Oracle® TimesTen In-Memory Database

C Developer's Guide

Release 22.1

F35396-05

February 2023

Oracle TimesTen In-Memory Database C Developer's Guide, Release 22.1

F35396-05

Contents

3 TimesTen Support for OCI

4 TimesTen Support for Pro*C/C++

5 XLA and TimesTen Event Management

10 TimesTen ODBC Support

About This Content

This document covers TimesTen support for ODBC, OCI, and Pro*C/C++.

Audience

This guide is for anyone developing or supporting applications that use TimesTen through ODBC, OCI, or Pro*C/C++.

In addition to familiarity with the particular programming interface you use, you should be familiar with TimesTen, SQL (Structured Query Language), and database operations.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Related Resources

Oracle Database documentation is available on the Oracle documentation website. This may be especially useful for Oracle Database features that TimesTen supports but does not attempt to fully document. In particular, the following Oracle Database guides may be of interest.

- *Oracle Call Interface Programmer's Guide*
- *Pro*C/C++ Programmer's Guide*
- *Oracle Database Globalization Support Guide*
- *Oracle Database Net Services Administrator's Guide*
- *Oracle Database SQL Language Reference*

This document frequently refers to ODBC API reference documentation for further information. This is available from Microsoft or a variety of third parties. For example:

`https://docs.microsoft.com/en-us/sql/odbc/reference/syntax/odbc-api-reference`

See TimesTen ODBC Support for details of TimesTen ODBC support.

Conventions

The following text conventions are used in this document.

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.

Convention	Meaning
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

What's New

This section summarizes new features and functionality of TimesTen Release 22.1, providing links into the guide for more information where applicable.

New features in Release 22.1.1.1.0

- For data returned from a `SELECT` statement in Client/Server, the buffer size for the data returned to the client is programmatically configurable to allow adjustments for better performance. See "Configuring the Result Set Buffer Size in Client/Server Using ODBC" and "Configuring the Result Set Buffer Size in Client/Server Using OCI".
- Multiple output REF CURSORs are supported. See "Working With REF CURSORs".
- The TimesTen driver manager is included in this release. See "Introduction to the TimesTen Driver Manager".

1 C Development Environment

This chapter provides information about the C development environment and related considerations for compiling and linking TimesTen applications, including discussion of the TimesTen driver manager.

These topics are covered:

TimesTen Environment Variable Settings

There are environment variables that must be set appropriately for proper operation of TimesTen. Scripts to set these environment variables are in the `timesten_home/bin` directory: `ttenv.sh` and `ttenv.csh` for Linux and UNIX platforms (depending on your shell) and `ttenv.bat` for Windows platforms.

Environment variable settings for TimesTen are discussed in Environment Variables in the *Oracle TimesTen In-Memory Database Installation, Migration, and Upgrade Guide*.

> **Note:**
>
> - The `ttenv` scripts also configure access to the Oracle Instant Client, required for OCI programming.
> - To ensure proper execution of OCI and Pro*C/C++ programs to be run on TimesTen, do not set `ORACLE_HOME` for OCI and Pro*C/C++ compilations (or unset it if it was set previously).

Introduction to the TimesTen Driver Manager

This section discusses the basic concepts and features of the TimesTen driver manager (TTDM).

TTDM is a lightweight ODBC driver manager that is designed and optimized for use with the TimesTen database. It provides access to TimesTen-specific features, extensions, and ODBC API support, offering 100% equivalent functionality to using the TimesTen direct driver or client/server driver directly.

An application links directly to the TTDM library, and TTDM dynamically loads the relevant ODBC driver libraries and passes ODBC calls from the application as needed. TTDM allows an application to use both direct and client/server connections at the same time.

TTDM offers these advantages over other driver managers:

- Its performance overhead is negligible.
- No source code changes or configuration changes are necessary in the application.

TTDM fully supports the TimesTen routing API and, for direct connections, TimesTen XLA and the TimesTen utility API. TTDM does not support XA.

To summarize:

- If you use only direct connections and have no need to use both direct and client/server connections from the same application process, then link directly with the TimesTen direct driver for maximum performance.
- If you need an application process to be able to use both direct and client/server connections, then link with TTDM for maximum performance and functionality with no application code changes.
- If you currently use a generic ODBC driver manager to enable use of both direct and client/server connections from the same process, consider using TTDM for improved performance and full availability of TimesTen features.

Refer to the following for information:

- Considerations for Linking With the TimesTen Driver Manager
- Compiling and Linking Applications With the TimesTen Driver Manager on Linux or UNIX
- Attribute Support for ODBC 3.5 SQLSetConnectAttr and SQLGetConnectAttr (for new attributes to support TTDM)
- Attribute Support for ODBC 3.5 SQLGetEnvAttr (for new attributes to support TTDM)

Linking Options

A TimesTen application can link directly with the TimesTen ODBC direct driver or ODBC client driver, link with a generic driver manager, or link with the TimesTen driver manager.

- Considerations for Linking Without an ODBC Driver Manager
- Considerations for Linking With a Generic ODBC Driver Manager
- Considerations for Linking With the TimesTen Driver Manager

Considerations for Linking Without an ODBC Driver Manager

Applications to be used solely with TimesTen can link directly with either the TimesTen ODBC direct driver or ODBC client driver, without a driver manager (or link with the TimesTen driver manager).

Linking without a generic driver manager avoids performance overhead and is a simple way to access TimesTen. However, developers of applications linked without a driver manager should be aware of the following issues.

- The application can connect only to a DSN (data source name) that uses the driver with which it is linked. It cannot connect to a database of any other vendor, nor can it connect to a TimesTen DSN of a different TimesTen driver. (A DSN is a logical name that identifies a TimesTen database and the set of connection attributes used for connecting to the database.)
- Windows ODBC tracing is not available.
- The ODBC cursor library is not available.
- Applications cannot use ODBC functions that are usually implemented by a driver manager, such as `SQLDataSources` and `SQLDrivers`.
- Applications that use `SQLCancel` to close a cursor instead of `SQLFreeStmt(..., SQL_CLOSE)` receive a return code of `SQL_SUCCESS_WITH_INFO` and a SQL state of `01S05`. This warning is intended to be used by the driver manager to manage its internal state. Applications should treat this warning as success.

Considerations for Linking With a Generic ODBC Driver Manager

Applications that link with an ODBC driver manager can connect to any DSN that references an ODBC driver and can even connect simultaneously to multiple DSNs that use different ODBC drivers.

Note, however, that driver managers are not available by default on most non-Windows platforms. In addition, using a generic driver manager may add significant synchronization overhead to every ODBC function call and has the following limitations:

- The TimesTen option `TT_PREFETCH_COUNT` cannot be used with applications that link with a driver manager. For more information on using `TT_PREFETCH_COUNT`, see Prefetching Multiple Rows of Data.
- Applications cannot set or reset the TimesTen-specific `TT_PREFETCH_CLOSE` connection option. See Optimizing Query Performance.
- Transaction Log API (XLA) calls cannot be used when applications are linked with a generic driver manager.
- The ODBC C types `SQL_C_BIGINT`, `SQL_C_TINYINT`, and `SQL_C_WCHAR` are not supported for an application linked with a generic driver manager when used with TimesTen. You cannot call methods that have any of these types in their signatures.

Considerations for Linking With the TimesTen Driver Manager

This section discusses behaviors for a developer to consider when you link an application with the TimesTen driver manager.

Limitations for ODBC driver managers noted in the preceding section, Considerations for Linking With a Generic ODBC Driver Manager, do not apply to TTDM.

Be aware of these behaviors when you link with TTDM:

- An ODBC 3 application is required to declare the ODBC version it is using by calling `SQLSetEnvAttr` to set `SQL_ATTR_ODBC_VERSION` to the value `SQL_OV_ODBC3`. TTDM detects the ODBC version (taken to be ODBC 2.5 if there is no SQL_OV_ODBC3 setting) and adapts some aspects of its behavior accordingly. It also passes the `SQLSetEnvAttr` call through to the underlying driver or drivers.

- The connection type (direct or client) is determined by TTDM at connection time based on the DSN or connection string used for the connection.
- When an application makes its first call to a function in TTDM, TTDM will attempt to dynamically load the direct driver library (`libtten.so`), the client library (`libttclient.so`), and the utility library (`libttutil.so`). Loading some of these libraries may fail depending on the TimesTen installation. For example, in a client-only instance there is no direct mode library or utility library. TTDM marks those features as unavailable, and if the application calls an unavailable function it will receive an error code indicating that.
- When an application calls an ODBC function or a TimesTen API function, it initially executes a TTDM-exported version of that function. In most cases, this TTDM stub function performs some minimal validation checks then calls the actual TimesTen function from the relevant TimesTen driver, passing the same parameters. Whatever the driver function returns is passed back to the calling application.

> **Tip:**
>
> Be aware of the following:
>
> - There are extensions for TTDM that can also be used by an application to programmatically determine if it is using TTDM or if it is linked directly with one of the TimesTen driver libraries. When any extension documented in Attribute Support for ODBC 3.5 SQLSetConnectAttr and SQLGetConnectAttr and Attribute Support for ODBC 3.5 SQLGetEnvAttr is called, a return value of `SQL_ERROR` indicates that the application is not using TTDM.
> - TTDM does not support the driver manager functions `SQLDrivers` and `SQLDataSources`.

Compiling and Linking Applications

There are methods for compiling and linking C applications on Windows and on Linux or UNIX.

- Compiling and Linking Applications on Windows
- Compiling and Linking Applications Directly With the TimesTen Drivers on Linux or UNIX
- Compiling and Linking Applications With the TimesTen Driver Manager on Linux or UNIX

Compiling and Linking Applications on Windows

There are methods for how to compile TimesTen applications on Windows. You are not required to specify the location of the ODBC include files. These files are included with Microsoft Visual C++. However, to use TimesTen features you must indicate the location of the TimesTen include files in the `/I` compiler option setting.
See TimesTen Include Files.

Link the appropriate libraries, as follows:

- Link directly to the native Windows driver manager, `odbc32.lib`
- Or link directly to one of the TimesTen drivers:
  - For direct mode: `tten221.lib` and `ttdv221.lib`
  - For client/server mode: `ttclient221.lib`

Link TimesTen libraries before any other libraries.

In addition, applications must do the following:

- Include `timesten.h`, the TimesTen include file. This automatically includes standard ODBC files as well. See TimesTen Include Files.
- Include TimesTen files before any other include files.

The Makefile in this example shows how to build a TimesTen application on Windows systems. This example assumes that *`timesten_home`*`\install\lib` has already been added to the `LIB` environment variable (which is accomplished when you execute `ttenv.bat`).

```
CFLAGS = "/Itimesten_home\install\include"
LIBSDM = ODBC32.LIB
LIBS = tten221.lib ttdv221.lib
LIBSDEBUG = tten221d.lib ttdv221d.lib
LIBSCS = ttclient221.lib

# Link with the ODBC driver manager
appldm.exe:appl.obj
           $(CC) /Feappldm.exe appl.obj $(LIBSDM)

# Link directly with the TimesTen
# ODBC production driver
appl.exe:appl.obj
         $(CC) /Feappl.exe appl.obj\
         $(LIBS)

# Link directly with the TimesTen
# ODBC debug driver
appldebug.exe:appl.obj
              $(CC) /Feappldebug.exe appl.obj\
              $(LIBSDEBUG)

# Link directly with the TimesTen
# ODBC client driver
applcs.exe:appl.obj
           $(CC) /Feapplcs.exe appl.obj\
           $(LIBSCS)
```

Note:

- TimesTen defaults to ODBC 3.5. To use ODBC 2.5 definitions and types, use the compiler setting `-DODBCVER=0x0250`.
- On Windows, there is only one TimesTen instance per installation, and *`timesten_home`* refers to *`installation_dir`*`\instance`.
- The *`timesten_home`*`\install` directory is a symbolic link to *`installation_dir`*.

Compiling and Linking Applications Directly With the TimesTen Drivers on Linux or UNIX

There are methods on how to compile TimesTen applications directly with the TimesTen drivers on Linux or UNIX platforms.

- Compile TimesTen applications using the TimesTen header files in the `include` directory of the TimesTen installation.
- Link with the TimesTen direct driver or the TimesTen client driver, each of which is provided as a shared library.
- Link TimesTen libraries before any other libraries.

In addition, applications must do the following:

- Include `timesten.h`, the TimesTen include file. This automatically includes standard ODBC files as well. See TimesTen Include Files.
- Include TimesTen files before any other include files.

To use the TimesTen include files if you are using TimesTen features, add the following to the C compiler command.

```
-Itimesten_home/install/include
```

To link with the TimesTen ODBC direct driver, add the following to the link command for the `libtten.so` library:

```
-Ltimesten_home/install/lib -ltten
```

The `-L` option tells the linker to search the TimesTen `lib` directory for library files. The `-ltten` option links in the TimesTen ODBC direct driver.

To link with the TimesTen ODBC client driver, add the following to the link command for the `libttclient.so` library:

```
-Ltimesten_home/install/lib -lttclient
```

On AIX, when linking applications with the TimesTen ODBC client driver, the C++ runtime library must be included in the link command (because the client driver is written in C++ and AIX does not link it automatically) and must follow the client driver:

```
-Ltimesten_home/install/lib -lttclient -lC_r
```

You do not have to include this library if you are linking with the TimesTen driver manager, discussed in Compiling and Linking Applications With the TimesTen Driver Manager on Linux or UNIX.

You can use Makefiles in subdirectories under the Quick Start `sample_code` directory (see About TimesTen Quick Start and Sample Applications), or you can use this example to guide you in creating your own Makefile.

```
CFLAGS = -Itimesten_home/install/include
LIBS = -Ltimesten_home/install/lib -ltten
LIBSDEBUG = -Ltimesten_home/install/lib -lttenD
LIBSCS = -Ltimesten_home/install/lib -lttclient

# Link directly with the TimesTen
# ODBC production driver
appl:appl.o
     $(CC) -o appl appl.o $(LIBS)

# Link directly with the TimesTen ODBC debug driver
appldebug:appl.o
     $(CC) -o appldebug appl.o $(LIBSDEBUG)

# Link directly with the TimesTen client driver
applcs:appl.o
     $(CC) -o applcs appl.o $(LIBSCS)
```

Note:

- TimesTen compiles against ODBC 3.5 by default. To compile an ODBC 2.5 application, use the compilation option setting `-DODBCVER=0x0250`.
- To directly link your application to the debug TimesTen ODBC driver, substitute `-lttenD` for `-ltten` on the link line.

Compiling and Linking Applications With the TimesTen Driver Manager on Linux or UNIX

This section discusses and shows commands for linking applications with the TimesTen driver manager (which is not supported on Windows).

With a few exceptions for specific discussion of the TimesTen direct and client/server drivers, discussion in the preceding section, Compiling and Linking Applications Directly With the TimesTen Drivers on Linux or UNIX, applies when you use TTDM, but note the following:

- Include `timesten.h` as you would normally. In addition, include any other TimesTen header files that you would normally. If your application uses XLA, include `tt_xla.h`. If your application uses the TimesTen utility API, include `tt_utillib.h` and `ttutil.h`.
- Link with the TTDM library, `libttdrvmgr.so`, instead of the ODBC direct driver or client/server driver. Do not link with any other TimesTen library.
- Do not link with any third-party driver manager library.

For example:

```
-Ltimesten_home/install/lib -lttdrvmgr
```

In a Makefile:

```
CFLAGS = -Itimesten_home/install/include
LIBTTDM = -Ltimesten_home/install/lib -lttdrvmgr
# Link with TTDM
applcs:appl.o
     $(CC) -o applcs appl.o $(LIBTTDM)
```

About TimesTen Quick Start and Sample Applications

The TimesTen Classic Quick Start and TimesTen Scaleout sample applications exhibit a variety of TimesTen features.

The sample applications are available from the TimesTen GitHub location. For the TimesTen Classic Quick Start, there is a complete set of tutorials, how-to instructions, and sample applications. For TimesTen Scaleout, there are ODBC and JDBC sample applications.

After you have configured your environment, you can confirm that everything is set up correctly by compiling and running the sample applications. For TimesTen Classic, applications are located under the Quick Start `sample_code` directory. For instructions on compiling and running them, see the instructions in the subdirectories. For TimesTen Scaleout, clone the `oracle-timesten-examples` GitHub repository and follow the instructions in the README files.

For TimesTen Classic, the following are included:

- Schema and setup: The `build_sampledb` script (`.sh` on Linux or UNIX or `.bat` on Windows) creates a sample database and schema. Run this script before using the sample applications.
- Environment and setup: The `ttquickstartenv` script (`.sh` or `.csh` on Linux or UNIX, `.bat` on Windows, or as applicable for your system), a superset of the `ttenv` script typically used for TimesTen setup, sets up the environment. Run this script each time you enter a session where you want to compile or run any of the sample applications.
- Sample applications and setup: The Quick Start provides sample applications and their source code for ODBC, OCI, and Pro*C/C++.

2 Working With TimesTen Databases in ODBC

This chapter covers TimesTen programming features and describes how to use ODBC to connect to and use the TimesTen database.

It includes the following topics:

- Management of TimesTen Database Connections
- Database Operations in ODBC
- TimesTen Features and Operations in Your Application
- Error Handling
- ODBC Support for Automatic Client Failover
- Client Routing API for TimesTen Scaleout

> **Note:**
>
> - For using OCI to access TimesTen from a C application, see TimesTen Support for OCI.
> - For using Pro*C/C++ to access TimesTen from a C application, see TimesTen Support for Pro*C/C++.
> - For accessing TimesTen from a C++ application, see Understanding and Using TTClasses in the *Oracle TimesTen In-Memory Database TTClasses Guide*.
> - For accessing TimesTen from a C# application, see Getting started with ODP.NET in the *Oracle Data Provider for .NET Oracle TimesTen In-Memory Database Support User's Guide*.

TimesTen supports:

- ODBC 2.5, Extension Level 1, as well as Extension Level 2 features that are documented in TimesTen ODBC Support
- ODBC 3.51 core interface conformance

Management of TimesTen Database Connections

There are methods to manage TimesTen database connections.

- Overview of TimesTen Connections
- SQLConnect, SQLDriverConnect, SQLAllocConnect, SQLDisconnect Functions
- Use of the Default DSN
- Connecting To and Disconnecting From a Database

- Setting Connection Attributes Programmatically

Overview of TimesTen Connections

ODBC applications can connect to a database by referencing either its attributes (host, port number, and so on) or its data source name (DSN). In TimesTen Classic, users can create DSNs directly. In TimesTen Scaleout, a DSN is created for each connectable you define in the grid.

This section covers some basics regarding TimesTen connections and provides references for details.

For TimesTen Scaleout, refer to *Oracle TimesTen In-Memory Database Scaleout User's Guide* for information about creating a database and connecting to a database, using either a direct connection or a client/server connection. See Creating a Database and Connecting to a Database.

For TimesTen Classic, *Oracle TimesTen In-Memory Database Operations Guide* contains information about creating a DSN for the database. The type of DSN you create depends on whether your application connects directly to the database or connects through a client:

- If you intend to connect directly to the database, refer to Managing TimesTen Databases in *Oracle TimesTen In-Memory Database Operations Guide*. There are sections on creating a DSN for a direct connection from Linux or UNIX or from Windows.
- If you intend to create a client connection to the database, refer to Working With the TimesTen Client and Server in *Oracle TimesTen In-Memory Database Operations Guide*. There are sections on creating a DSN for a client/server connection from Linux or UNIX or from Windows.

> **Note:**
>
> - In TimesTen, the user name and password must be for a valid user who has been granted `CREATE SESSION` privilege to connect to the database.
> - A TimesTen connection cannot be inherited from a parent process. If a process opens a database connection before creating (forking) a child process, the child must not use the connection.

SQLConnect, SQLDriverConnect, SQLAllocConnect, SQLDisconnect Functions

There are ODBC functions that are available for connecting to a database, allocating memory for the connection, and disconnecting from the database.

- `SQLConnect`: Loads a driver and connects to the database. The connection handle points to where information about the connection is stored, including status, transaction state, results, and error information.

  Here is the `SQLConnect` calling sequence:

```
SQLRETURN SQLConnect(
          SQLHDBC         ConnectionHandle,
          SQLCHAR *       ServerName,
          SQLSMALLINT     NameLength1,
          SQLCHAR *       UserName,
          SQLSMALLINT     NameLength2,
          SQLCHAR *       Authentication,
          SQLSMALLINT     NameLength3);
```

- `SQLDriverConnect`: This is an alternative to `SQLConnect` when more information is required than what is supported by `SQLConnect`, which is just data source (the database), user name, and password.

  Here is the `SQLDriverConnect` calling sequence:

  ```
  SQLRETURN SQLDriverConnect(
            SQLHDBC          ConnectionHandle,
            SQLHWND          WindowHandle,
            SQLCHAR *        InConnectionString,
            SQLSMALLINT      StringLength1,
            SQLCHAR *        OutConnectionString,
            SQLSMALLINT      BufferLength,
            SQLSMALLINT *    StringLength2Ptr,
            SQLUSMALLINT     DriverCompletion);
  ```

- `SQLAllocConnect`: Allocates memory for a connection handle within the specified environment.

  Here is the `SQLAllocConnect` calling sequence:

  ```
  SQLRETURN SQLAllocConnect(
             SQLHENV        EnvironmentHandle,
             SQLHDBC        PointerToConnectionHandle);
  ```

- `SQLDisconnect`: Disconnect from the database. Takes the existing connection handle as its only argument.

  Here is the `SQLDisconnect` calling sequence:

  ```
  SQLRETURN SQLDisconnect(
            SQLHDBC         ConnectionHandle);
  ```

Refer to ODBC API reference documentation for additional details about these functions.

Use of the Default DSN

This lists circumstances when a default DSN is used.

In TimesTen Classic, a default DSN, simply named `default`, can be defined in the `odbc.ini` or `sys.odbc.ini` file. See Setting Up a Default DSN in TimesTen Classic in *Oracle TimesTen In-Memory Database Operations Guide*.

For `SQLConnect`, if a default DSN has been defined, it is used in these circumstances:

- If *`ServerName`* specifies a data source that cannot be found.
- If *`ServerName`* is a null pointer.
- If `default` is specified as the server name. The user name and authentication values are used as is.

For `SQLDriverConnect`, if a default DSN has been defined, it is used in these circumstances:

- If the connection string does not include the `DSN` keyword.
- If the data source cannot be found.
- If `default` is specified as the DSN keyword. The user name and password are used as is.

Be aware of the following usage notes when in direct mode versus client/server mode with a generic driver manager:

- When you are not using a generic driver manager, TimesTen manages this functionality. The default DSN must be a TimesTen database.
- When you are using a generic driver manager, the driver manager manages this functionality. The default DSN need not be a TimesTen database.

Connecting To and Disconnecting From a Database

There are methods for connecting to and disconnecting from a database.

This code fragment invokes `SQLConnect` and `SQLDisconnect` to connect to and disconnect from the database named `FixedDs`. The first invocation of `SQLConnect` by any application causes the creation of the `FixedDs` database. Subsequent invocations of `SQLConnect` would connect to the existing database.

```
#include <timesten.h>
SQLRETURN retcode;
SQLHDBC hdbc;

...
retcode = SQLConnect(hdbc,
                     (SQLCHAR*)"FixedDs", SQL_NTS,
                     (SQLCHAR*)"johndoe", SQL_NTS,
                     (SQLCHAR*)"opensesame", SQL_NTS);
...
retcode = SQLDisconnect(hdbc);
...
```

The following is a complete program that creates, connects to, and disconnects from a database. The example uses `SQLDriverConnect` instead of `SQLConnect` to set up the connection, and uses `SQLAllocConnect` to allocate memory. It also shows how to get error messages. (In addition, you can refer to Error Handling.)

```
#include <timesten.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

static void chkReturnCode(SQLRETURN rc, SQLHENV henv,
                          SQLHDBC hdbc, SQLHSTMT hstmt,
                          char* msg, char* filename,
                          int lineno, BOOL err_is_fatal);
#define DEFAULT_CONNSTR "DSN=sampledb;PermSize=32"

int
main(int ac, char** av)
{
   SQLRETURN rc = SQL_SUCCESS;
                  /* General return code for the API */
   SQLHENV henv = SQL_NULL_HENV;
```

```
               /* Environment handle */
SQLHDBC hdbc = SQL_NULL_HDBC;
               /* Connection handle */
SQLHSTMT hstmt = SQL_NULL_HSTMT;
               /* Statement handle */
SQLCHAR connOut[255];
               /* Buffer for completed connection string */
SQLSMALLINT connOutLen;
               /* Number of bytes returned in ConnOut */
SQLCHAR *connStr = (SQLCHAR*)DEFAULT_CONNSTR;
               /* Connection string */
rc = SQLAllocEnv(&henv);
if (rc != SQL_SUCCESS) {
   fprintf(stderr, "Unable to allocate an "
          "environment handle\n");
 exit(1);
}
rc = SQLAllocConnect(henv, &hdbc);
chkReturnCode(rc, henv, SQL_NULL_HDBC,
           SQL_NULL_HSTMT,
           "Unable to allocate a "
           "connection handle\n",
           __FILE__, __LINE__, 1);

rc = SQLDriverConnect(hdbc, NULL,
                      connStr, SQL_NTS,
                      connOut, sizeof(connOut),
                      &connOutLen,
                      SQL_DRIVER_NOPROMPT);
chkReturnCode(rc, henv, hdbc, SQL_NULL_HSTMT,
              "Error in connecting to the"
              " database\n",
              __FILE__, __LINE__, 1);
rc = SQLAllocStmt(hdbc, &hstmt);
chkReturnCode(rc, henv, hdbc, SQL_NULL_HSTMT,
              "Unable to allocate a "
              "statement handle\n",
              __FILE__, __LINE__, 1);

/* Your application code here */

if (hstmt != SQL_NULL_HSTMT) {
  rc = SQLFreeStmt(hstmt, SQL_DROP);
  chkReturnCode(rc, henv, hdbc, hstmt,
                "Unable to free the "
                "statement handle\n",
                 __FILE__, __LINE__, 0);
}

rc = SQLDisconnect(hdbc);
chkReturnCode(rc, henv, hdbc,
              SQL_NULL_HSTMT,
              "Unable to close the "
              "connection\n",
              __FILE__, __LINE__, 0);

rc = SQLFreeConnect(hdbc);
chkReturnCode(rc, henv, hdbc,
              SQL_NULL_HSTMT,
              "Unable to free the "
              "connection handle\n",
```

```
                   __FILE__, __LINE__, 0);

   rc = SQLFreeEnv(henv);
   chkReturnCode(rc, henv, SQL_NULL_HDBC,
                 SQL_NULL_HSTMT,
                 "Unable to free the "
                 "environment handle\n",
                 __FILE__, __LINE__, 0);
     return 0;
   }
}

static void
chkReturnCode(SQLRETURN rc, SQLHENV henv,
              SQLHDBC hdbc, SQLHSTMT hstmt,
              char* msg, char* filename,
              int lineno, BOOL err_is_fatal)
{
   #define MSG_LNG 512
   SQLCHAR sqlState[MSG_LNG];
   /* SQL state string */
   SQLINTEGER nativeErr;
   /* Native error code */
   SQLCHAR errMsg[MSG_LNG];
   /* Error msg text buffer pointer */
   SQLSMALLINT errMsgLen;
   /* Error msg text Available bytes */
   SQLRETURN ret = SQL_SUCCESS;
   if (rc != SQL_SUCCESS &&
       rc != SQL_NO_DATA_FOUND ) {
      if (rc != SQL_SUCCESS_WITH_INFO) {
       /*
        * It's not just a warning
        */
      fprintf(stderr, "*** ERROR in %s, line %d:"
               " %s\n",
               filename, lineno, msg);
  }
  /*
   * Now see why the error/warning occurred
   */
  while (ret == SQL_SUCCESS ||
         ret == SQL_SUCCESS_WITH_INFO) {
    ret = SQLError(henv, hdbc, hstmt,
                   sqlState, &nativeErr,
                   errMsg, MSG_LNG,
                   &errMsgLen);
    switch (ret) {
      case SQL_SUCCESS:
        fprintf(stderr, "*** %s\n"
                 "*** ODBC Error/Warning = %s, "
                 "TimesTen Error/Warning "
                 " = %d\n",
                 errMsg, sqlState,
                 nativeErr);
      break;
    case SQL_SUCCESS_WITH_INFO:
      fprintf(stderr, "*** Call to SQLError"
              " failed with return code of "
              "SQL_SUCCESS_WITH_INFO.\n "
              "*** Need to increase size of"
```

```
                " message buffer.\n");
          break;
        case SQL_INVALID_HANDLE:
          fprintf(stderr, "*** Call to SQLError"
                  " failed with return code of "
                  "SQL_INVALID_HANDLE.\n");
          break;
        case SQL_ERROR:
          fprintf(stderr, "*** Call to SQLError"
                  " failed with return code of "
                  "SQL_ERROR.\n");
          break;
        case SQL_NO_DATA_FOUND:
          break;
        } /* switch */
      } /* while */
      if (rc != SQL_SUCCESS_WITH_INFO && err_is_fatal) {
        fprintf(stderr, "Exiting.\n");
        exit(-1);
      }
   }
```

Setting Connection Attributes Programmatically

This shows how to set or override connection attributes programmatically by specifying a connection string when you connect to a database.

This code fragment connects to a database named `mydsn` and indicates in the `SQLDriverConnect` call that the application should use a passthrough setting of 3. Note that `PassThrough` is a general connection attribute.

```
SQLHDBC hdbc;
SQLCHAR ConnStrOut[512];
SQLSMALLINT cbConnStrOut;
SQLRETURN rc;

rc = SQLDriverConnect(hdbc, NULL,
    "DSN=mydsn;PassThrough=3", SQL_NTS,
    ConnStrOut, sizeof (ConnStrOut),
    &cbConnStrOut, SQL_DRIVER_NOPROMPT);
```

> **Note:**
>
> - Each direct connection to a database opens several files. An application with many threads, each with a separate connection, has several files open for each thread. Such an application can exceed the maximum allowed (or configured maximum) number of file descriptors that may be simultaneously open on the operating system. In this case, configure your system to allow a larger number of open files. See Limits on Number of Open Files in *Oracle TimesTen In-Memory Database Reference*.
> - Refer to Managing TimesTen Databases in *Oracle TimesTen In-Memory Database Operations Guide* for general information about connection attributes. General connection attributes require no special privilege. First connection attributes are set when the database is first loaded, and persist for all connections. Only the instance administrator can load a database with changes to first connection attribute settings. Refer to Connection Attributes in *Oracle TimesTen In-Memory Database Reference*.

Database Operations in ODBC

There are basic methods for using ODBC in TimesTen.

- ODBC Functions to Execute SQL Statements
- Steps to Prepare and Execute Queries and Work With Cursors in ODBC
- Creating a Table in ODBC
- Preparing and Executing a Query in ODBC
- Committing Changes to the Database in ODBC

ODBC Functions to Execute SQL Statements

The `SQLExecute` and `SQLExecDirect` ODBC functions are used to execute SQL statements.

- `SQLExecute`: Executes a statement that has been prepared with `SQLPrepare`. After the application is done with the results, they can be discarded and `SQLExecute` can be run again using different parameter values.

  This is typically used for DML statements with bind parameters, or statements that are being executed more than once.

- `SQLExecDirect`: Prepares and executes a statement.

  This is typically used for DDL statements or for DML statements that would execute only a few times and without bind parameters.

Refer to ODBC API reference documentation for details about these functions.

Steps to Prepare and Execute Queries and Work With Cursors in ODBC

There are ODBC functions used to prepare and execute queries and work with cursors.

> **Note:**
>
> In TimesTen, any operation that ends your transaction, such as a commit or rollback, closes all cursors associated with the connection.

In ODBC, a cursor is always associated with a result set. This association is made by the ODBC driver. The application can control cursor characteristics, such as the number of rows to fetch at one time, using `SQLSetStmtAttr` attributes documented in Attribute Support for ODBC 3.5 SQLSetStmtAttr and SQLGetStmtAttr. The steps involved in executing a query typically include the following.

1. Use `SQLPrepare` to prepare the `SELECT` statement for execution.
2. Use `SQLBindParameter`, if the statement has parameters, to bind each parameter to an application address. See SQLBindParameter Function.
3. Call `SQLBindCol` to assign the storage and data type for a column of results, binding column results to local variable storage in your application.
4. Call `SQLExecute` to execute the `SELECT` statement. See ODBC Functions to Execute SQL Statements.
5. Call `SQLFetch` to fetch the results. Specify the statement handle.
6. Call `SQLFreeStmt` to free the statement handle. Specify the statement handle and either `SQL_CLOSE`, `SQL_DROP`, `SQL_UNBIND`, or `SQL_RESET_PARAMS`.

Refer to ODBC API reference documentation for details on these ODBC functions. Examples are shown throughout this chapter and in the TimesTen sample applications. See About TimesTen Quick Start and Sample Applications.

> **Note:**
>
> By default (when connection attribute `PrivateCommands=0`), TimesTen shares prepared statements between connections, so subsequent prepares of the same statement on different connections execute very quickly.

Creating a Table in ODBC

You can create a table in ODBC.

This example creates a table, `NameID`, with two columns: `CustID` and `CustName`. The table maps character names to integer identifiers.

```
#include <timesten.h>
SQLRETURN rc;
SQLHSTMT hstmt;
...
```

```
rc = SQLExecDirect(hstmt, (SQLCHAR*)
     "CREATE TABLE NameID (CustID INTEGER, CustName VARCHAR(50))",
     SQL_NTS);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
     ... /* handle error */
```

Preparing and Executing a Query in ODBC

This example prepares and executes a query.

Error checking has been omitted to simplify the example. In addition to ODBC functions mentioned previously, this example uses `SQLNumResultCols` to return the number of columns in the result set, `SQLDescribeCol` to return a description of one column of the result set (column name, type, precision, scale, and nullability), and `SQLBindCol` to assign the storage and data type for a column in the result set. These are all described in detail in ODBC API reference documentation.

```
#include <timesten.h>

SQLHSTMT hstmt;
SQLRETURN rc;
int i;
SQLSMALLINT numCols;
SQLCHAR colname[32];
SQLSMALLINT colnamelen, coltype, scale, nullable;
SQLULEN collen [MAXCOLS];
SQLLEN outlen [MAXCOLS];
SQLCHAR* data [MAXCOLS];

/* other declarations and program set-up here */

/* Prepare the SELECT statement */
rc = SQLPrepare(hstmt,
(SQLCHAR*) "SELECT * FROM EMP WHERE AGE>20",
SQL_NTS);
/* ... */

/* Determine number of columns in result rows */
rc = SQLNumResultCols(hstmt, &numCols);

/* ... */

/* Describe and bind the columns */
for (i = 0; i < numCols; i++) {
    rc = SQLDescribeCol(hstmt,
         (SQLSMALLINT) (i + 1),
         colname,(SQLSMALLINT)sizeof(colname), &colnamelen, &coltype, &collen[i],
         &scale, &nullable);

    /* ... */

   data[i] = (SQLCHAR*) malloc (collen[i] +1);  //Allocate space for column data.
   rc = SQLBindCol(hstmt, (SQLSMALLINT) (i + 1),
                   SQL_C_CHAR, data[i],
                   COL_LEN_MAX, &outlen[i]);

   /* ... */

}
/* Execute the SELECT statement */
```

```
rc = SQLExecute(hstmt);

/* ... */

/* Fetch the rows */
if (numCols > 0) {
  while ((rc = SQLFetch(hstmt)) == SQL_SUCCESS ||
          rc == SQL_SUCCESS_WITH_INFO) {
    /* ... "Process" the result row */
  } /* end of for-loop */
  if (rc != SQL_NO_DATA_FOUND)
    fprintf(stderr,
            "Unable to fetch the next row\n");

/* Close the cursor associated with the SELECT statement */
  rc = SQLFreeStmt(hstmt, SQL_CLOSE);
}
```

Committing Changes to the Database in ODBC

You can either autocommit or manually commit changes to the database. You can also disable autocommit and manually commit.

Autocommit is enabled by default (according to the ODBC specification), so that any DML change you make, such as an update, insert, or delete, is committed automatically. It is recommended, however, that you disable this feature and commit (or roll back) your changes explicitly. Use the `SQL_AUTOCOMMIT` option in a `SQLSetConnectOption` call to accomplish this:

```
rc = SQLSetConnectOption(hdbc, SQL_AUTOCOMMIT, SQL_AUTOCOMMIT_OFF);
```

With autocommit disabled, you can commit or roll back a transaction using the `SQLTransact` ODBC function, such as in the following example to commit:

```
rc = SQLTransact(henv, hdbc, SQL_COMMIT);
```

Refer to ODBC API reference documentation for details about these functions. Refer to Transaction Overview in *Oracle TimesTen In-Memory Database Operations Guide*.

Note:

- Autocommit mode applies only to the top-level statement executed by `SQLExecute` or `SQLExecDirect`. There is no awareness of what occurs inside the statement, and therefore no capability for intermediate autocommits of nested operations.
- All open cursors on the connection are closed upon transaction commit or rollback in TimesTen.
- The `SQLRowCount` function can be used to return information about SQL operations. For `UPDATE`, `INSERT`, and `DELETE` statements, the output argument returns the number of rows affected. See Retrieving Information About Cache Groups. Refer to ODBC API reference documentation for general information about `SQLRowCount` and its arguments.

This example prepares and executes a statement to give raises to selected employees, then manually commits the changes. Assume autocommit has been previously disabled.

```
update_example(SQLHDBC hdbc)
{
  SQLCHAR*      stmt_text;
  SQLHSTMT      hstmt;
  SQLINTEGER    raise_pct;
  char          hiredate_str[30];
  SQLLEN        hiredate_len;
  SQLLEN        numrows;

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

  /* prepare an update statement to give raises to employees hired before a
   * given date */
  stmt_text = (SQLCHAR*)
    "update employees "
    "set salary = salary * ((100 + :raise_pct) / 100.0) "
    "where hire_date < :hiredate";
  SQLPrepare(hstmt, stmt_text, SQL_NTS);

  /* bind parameter 1 (:raise_pct) to variable raise_pct */
  SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_SLONG,
                   SQL_DECIMAL, 0, 0, (SQLPOINTER)&raise_pct, 0, 0);

  /* bind parameter 2 (:hiredate) to variable hiredate_str */
  SQLBindParameter(hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR,
                   SQL_TIMESTAMP, 0, 0, (SQLPOINTER)hiredate_str,
                   sizeof(hiredate_str), &hiredate_len);

  /* set parameter values to give a 10% raise to employees hired before
   * January 1, 1996. */
  raise_pct = 10;
  strcpy(hiredate_str, "1996-01-01");
  hiredate_len = SQL_NTS;

  /* execute the update statement */
  SQLExecute(hstmt);

  /* print the number of employees who got raises  */
  SQLRowCount(hstmt, &numrows);
  printf("Gave raises to %d employees.\n", numrows);

  /* drop the statement handle */
  SQLFreeStmt(hstmt, SQL_DROP);

  /* commit the changes */
  SQLTransact(henv, hdbc, SQL_COMMIT);

}
```

TimesTen Features and Operations in Your Application

This section provides information about how an application works with data in a TimesTen database.

It includes the following topics. (See Working With Data in a TimesTen Database in *Oracle TimesTen In-Memory Database Operations Guide*.)

- TimesTen Include Files
- TimesTen Deferred Prepare
- Prefetching Multiple Rows of Data
- Optimizing Query Performance
- Parameter Binding and Statement Execution
- Working With REF CURSORs
- Working With DML Returning (RETURNING INTO Clause)
- Working With rowids
- Large Objects (LOBs)
- Using CALL to Execute Procedures and Functions
- Timeouts and Thresholds for Executing SQL Statements
- Configuring the Result Set Buffer Size in Client/Server Using ODBC
- Features for Cache

TimesTen Include Files

This section lists files you must include from your code in order to use TimesTen features. They are located in the `include` directory of the TimesTen installation.

Set the include path appropriately to access any files that are to be included. See Compiling and Linking Applications.

Include File	Description
`timesten.h`	TimesTen ODBC features This file includes the appropriate version of `sql.h`: the TimesTen version on Linux or UNIX systems or the system version on Windows systems. This file also includes `sqltypes.h`, `sqlext.h`, and `sqlucode.h`. On Windows systems, it also includes `windows.h`.
`tt_errCode.h`	TimesTen error codes (optional—see notes) This file maps TimesTen error codes to defined constants.

> **Note:**
>
> - If you include `sql.h` directly (instead of through `timesten.h`), on Windows you must include the system version of `sql.h`, not the TimesTen version.
> - Type definitions previously in `sqlunix.h` are now in `sqltypes.h`; however, `sqlunix.h` still exists (as an empty file) for backward compatibility.
> - There are alternatives to including `tt_errCode.h`. One is to move any desired constant definitions to `timesten.h`. Another is to reference the corresponding integer values directly in your code.

TimesTen Deferred Prepare

TimesTen has a deferred prepare feature to reduce round trips to the database.

In standard ODBC, a `SQLPrepare` call compiles a SQL statement so that information about the statement, such as column descriptions for the result set, is available to the application and accessible through calls such as `SQLDescribeCol`. To accomplish this, the `SQLPrepare` call must communicate with the server for processing.

This is in contrast, for example, to expected behavior under Oracle Call Interface (OCI), where a prepare call is expected to be a lightweight operation performed on the client to simply extract names and positions of parameters.

To avoid unwanted round trips between client and server, and also to make the behavior consistent with OCI expectations, the TimesTen client library implementation of `SQLPrepare` performs what is referred to as a "deferred prepare", where the request is not sent to the server until required. Examples of when the round trip would be required:

- When there is a `SQLExecute` call. Note that if there is a deferred prepare call that has not yet been sent to the server, a `SQLExecute` call on the client is converted to a `SQLExecDirect` call.
- When there is a request for information about the query that can only be supplied by the SQL engine, such as when there is a `SQLDescribeCol` call, for example. Many such calls in standard ODBC can access information previously returned by a `SQLPrepare` call, but with the deferred prepare functionality the `SQLPrepare` call is sent to the server and the information is returned to the application only as needed.

> **Note:**
>
> Deferred prepare functionality is not implemented (and not necessary) with the TimesTen direct driver.

The deferred prepare implementation requires no changes at the application or user level; however, be aware that calling any of the following functions may result in a round trip to the server if the required information from a previously prepared statement has not yet been retrieved:

- `SQLColAttributes`
- `SQLDescribeCol`
- `SQLDescribeParam`
- `SQLNumResultCols`
- `SQLNumParams`
- `SQLGetStmtOption` (for options that depend on the statement having been compiled by the SQL engine)

Also be aware that when calling any of these functions, any error from an earlier `SQLPrepare` call may be deferred until one of these calls is executed. In addition, these calls may return errors specific to `SQLPrepare` as well as errors specific to themselves.

Prefetching Multiple Rows of Data

A TimesTen extension to ODBC enables applications to prefetch multiple rows of data into the ODBC driver buffer. This can improve performance of client/server applications.

The `TT_PREFETCH_COUNT` ODBC statement option determines how many rows a `SQLFetch` call prefetches. Note that this option provides no benefit for an application using a direct connection to TimesTen.

You can set `TT_PREFETCH_COUNT` in a call to either `SQLSetStmtOption` or `SQLSetConnectOption` (which sets the option default value for all statements associated with the connection). The value can be any integer from 0 to 128, inclusive. Following is an example.

```
rc = SQLSetConnectOption(hdbc, TT_PREFETCH_COUNT, 100);
```

With this setting, the first `SQLFetch` call on the connection prefetches 100 rows. Subsequent `SQLFetch` calls fetch from the ODBC buffer instead of from the database, until the buffer is depleted. After it is depleted, the next `SQLFetch` call fetches another 100 rows into the buffer, and so on.

To disable prefetch, set `TT_PREFETCH_COUNT` to 1.

When you set the prefetch count to 0, TimesTen uses a default prefetch count according to the isolation level you have set for the database, and sets `TT_PREFETCH_COUNT` to that value. With Read Committed isolation level, the default prefetch value is 5. With Serializable isolation level, the default is 128. The default prefetch value is a good setting for most applications. Generally, a higher value may result in better performance for larger result sets, at the expense of slightly higher resource use.

You can also see Attribute Support for ODBC 3.5 SQLSetStmtAttr and SQLGetStmtAttr.

Optimizing Query Performance

A TimesTen extension to ODBC enables applications to optimize read-only query performance in client/server applications by using the `TT_PREFETCH_CLOSE` ODBC connection option.

Set `TT_PREFETCH_CLOSE` to `TT_PREFETCH_CLOSE_ON` using `SQLSetConnectOption`.

All transactions should be committed when executed, including read-only transactions. When TT_PREFETCH_CLOSE is set to TT_PREFETCH_CLOSE_ON, the server automatically closes the cursor and commits the transaction after the server has prefetched all rows of the result set for a read-only query. This enhances performance by reducing the number of network round-trips between client and server.

The client should still free the statement with SQLFreeStmt(SQL_CLOSE) and commit the transaction with SQLTransact(SQL_COMMIT), but those calls are executed in the client and do not require a network round trip between the client and server.

> **Note:**
>
> - Do not use multiple statement handles for the same connection when TT_PREFETCH_CLOSE is set to TT_PREFETCH_CLOSE_ON. The server may fetch all of the result set, commit the transaction, and close the statement handle before the client is finished, resulting in the closing of all statement handles.
> - This option is ignored for TimesTen direct connections and for SELECT FOR UPDATE statements.

The following example shows how to use the TT_PREFETCH_CLOSE option.

```
SQLSetConnectOption (hdbc, TT_PREFETCH_CLOSE, TT_PREFETCH_CLOSE_ON);
SQLExecDirect (hstmt, "SELECT * FROM T", SQL_NTS);
while (SQLFetch (hstmt) != SQL_NO_DATA_FOUND)
{
// do the processing and error checking
}
SQLFreeStmt (hstmt, SQL_CLOSE);
SQLTransact(SQL_COMMIT);
```

Parameter Binding and Statement Execution

There are methods for how to bind input or output parameters for SQL statements.

The following topics are covered.

- SQLBindParameter Function
- Parameter Type Assignments and Type Conversions
- ODBC SQL to TimesTen SQL or PL/SQL Type Mappings
- Binding Input Parameters
- Binding Output Parameters
- Binding Input/Output Parameters
- Binding of Duplicate Parameters in SQL Statements
- Binding of Duplicate Parameters in PL/SQL Statements
- Considerations for Floating Point Data
- Using SQL_WCHAR and SQL_WVARCHAR With a Driver Manager

> **Note:**
>
> The term "bind parameter" as used in TimesTen developer guides (in keeping with ODBC terminology) is equivalent to the term "bind variable" as used in TimesTen PL/SQL documents (in keeping with Oracle Database PL/SQL terminology).

SQLBindParameter Function

The ODBC `SQLBindParameter` function is used to bind parameters for SQL statements. This could include input, output, or input/output parameters.

To bind an input parameter through ODBC, use the `SQLBindParameter` function with a setting of `SQL_PARAM_INPUT` for the *`fParamType`* argument. Refer to ODBC API reference documentation for details about the `SQLBindParameter` function. Table 2-1 provides a brief summary of its arguments.

To bind an output or input/output parameter through ODBC, use the `SQLBindParameter` function with a setting of `SQL_PARAM_OUTPUT` or `SQL_PARAM_INPUT_OUTPUT`, respectively, for the *`fParamType`* argument. As with input parameters, use the *`fSqlType`*, *`cbColDef`*, and *`ibScale`* arguments (as applicable) to specify data types.

Table 2-1 SQLBindParameter Arguments

Argument	Type	Description
`hstmt`	`SQLHSTMT`	Statement handle
`ipar`	`SQLUSMALLINT`	Parameter number, sequentially from left to right, starting with 1
`fParamType`	`SQLSMALLINT`	Indicating input or output: `SQL_PARAM_INPUT`, `SQL_PARAM_OUTPUT`, or `SQL_PARAM_INPUT_OUTPUT`
`fCType`	`SQLSMALLINT`	C data type of the parameter
`fSqlType`	`SQLSMALLINT`	SQL data type of the parameter
`cbColDef`	`SQLULEN`	The precision of the parameter, such as the maximum number of bytes for binary data, the maximum number of digits for a number, or the maximum number of characters for character data
`ibScale`	`SQLSMALLINT`	The scale of the parameter, referring to the maximum number of digits to the right of the decimal point, where applicable
`rgbValue`	`SQLPOINTER`	Pointer to a buffer for the data of the parameter
`cbValueMax`	`SQLLEN`	Maximum length of the *`rgbValue`* buffer, in bytes
`pcbValue`	`SQLLEN*`	Pointer to a buffer for the length of the parameter

> **Note:**
>
> Refer to Data Types in *Oracle TimesTen In-Memory Database SQL Reference*.

Parameter Type Assignments and Type Conversions

Bind parameter type assignments are decided by different entities depending on where they are executed. Type conversions are performed by the ODBC driver.

This section discusses bind parameter type assignments, which are determined as follows:

- Parameter type assignments for statements that execute in TimesTen are determined by TimesTen. Specifically:
  - For SQL statements that execute within TimesTen, the TimesTen query optimizer determines data types of SQL parameters.
- Parameter type assignments for statements that execute in Oracle Database, or according to Oracle Database functionality, are determined by the application as follows.
  - For SQL statements that execute within Oracle Database—that is, passthrough statements from cache—the application must specify data types through its calls to the ODBC `SQLBindParameter` function, according to the *`fSqlType`*, *`cbColDef`*, and *`ibScale`* arguments of that function, as applicable.
  - For PL/SQL blocks or procedures that execute within TimesTen, where the PL/SQL execution engine has the same basic functionality as in Oracle Database, the application must specify data types through its calls to `SQLBindParameter` (the same as for SQL statements that execute within Oracle Database).

    So regarding host binds for PL/SQL (the variables, or parameters, that are preceded by a colon within a PL/SQL block), note that the type of a host bind is effectively declared by the call to `SQLBindParameter`, according to *`fSqlType`* and the other arguments as applicable, and is not declared within the PL/SQL block.

The ODBC driver performs any necessary type conversions between C values and SQL or PL/SQL types. For any C-to-SQL or C-to-PL/SQL combination that is not supported, an error occurs. These conversions can be from a C type to a SQL or PL/SQL type (input parameter), from a SQL or PL/SQL type to a C type (output parameter), or both (input/output parameter).

See the next section for information about type mappings between ODBC and TimesTen.

Note:

The TimesTen binding mechanism (early binding) differs from that of Oracle Database (late binding). TimesTen requires the data types before preparing queries. As a result, there will be an error if the data type of each bind parameter is not specified or cannot be inferred from the SQL statement. This would apply, for example, to the following statement:

```
SELECT 'x' FROM DUAL WHERE ? = ?;
```

You could address the issue as follows, for example:

```
SELECT 'x' from DUAL WHERE CAST(? as VARCHAR2(10)) =
                           CAST(? as VARCHAR2(10));
```

ODBC SQL to TimesTen SQL or PL/SQL Type Mappings

There are mappings from ODBC SQL to TimesTen SQL or PL/SQL.

Table 2-2 documents the mapping between ODBC types and SQL or PL/SQL types.

Table 2-2 ODBC SQL to TimesTen SQL or PL/SQL Type Mappings

ODBC Type (*fSqlType*)	SQL or PL/SQL Type	TimesTen Support Notes
SQL_BIGINT	NUMBER	No notes
SQL_BINARY	RAW(*p*)	No notes
SQL_BIT	PLS_INTEGER	No notes
SQL_CHAR	CHAR(*p*)	No notes
SQL_DATE	DATE	No notes
SQL_DECIMAL	NUMBER	No notes
SQL_DOUBLE	NUMBER	No notes
SQL_FLOAT	BINARY_DOUBLE	No notes
SQL_INTEGER	PLS_INTEGER	No notes
SQL_INTERVAL_DAY	N/A	See notes after this table.
SQL_INTERVAL_DAY_TO_HOUR	N/A	See notes after this table.
SQL_INTERVAL_DAY_TO_MINUTE	N/A	See notes after this table.
SQL_INTERVAL_DAY_TO_SECOND	N/A	See notes after this table.
SQL_INTERVAL_HOUR	N/A	See notes after this table.
SQL_INTERVAL_HOUR_TO_MINUTE	N/A	See notes after this table.
SQL_INTERVAL_HOUR_TO_SECOND	N/A	See notes after this table.
SQL_INTERVAL_MINUTE	N/A	See notes after this table.
SQL_INTERVAL_MINUTE_TO_SECOND	N/A	See notes after this table.
SQL_INTERVAL_MONTH	N/A	See notes after this table.

Table 2-2 (Cont.) ODBC SQL to TimesTen SQL or PL/SQL Type Mappings

ODBC Type (*fSqlType*)	SQL or PL/SQL Type	TimesTen Support Notes
SQL_INTERVAL_YEAR	N/A	See notes after this table.
SQL_INTERVAL_YEAR_TO_MONTH	N/A	See notes after this table.
SQL_INTERVAL_SECOND	N/A	See notes after this table.
SQL_NUMERIC	NUMBER	No notes
SQL_REAL	BINARY_FLOAT	No notes
SQL_REFCURSOR	REF CURSOR	No notes
SQL_ROWID	ROWID	No notes
SQL_SMALLINT	PLS_INTEGER	No notes
SQL_TIME	TIME	TimesTen does not support TIMEZONE. TIME data type values are stored without making any adjustment for time difference. Applications must assume one time zone and convert TIME values to that time zone before sending the values to the database.
SQL_TIMESTAMP	TIMESTAMP(*s*)	Same consideration as for TIME.
SQL_TINYINT	PLS_INTEGER	No notes
SQL_VARBINARY	RAW(*p*)	No notes
SQL_VARCHAR	VARCHAR2(*p*)	No notes
SQL_WCHAR	NCHAR(*p*)	No notes
SQL_WVARCHAR	NVARCHAR2(*p*)	No notes

Note:

- The notation (*p*) indicates precision is according to the SQLBindParameter argument *cbColDef*.
- The notation (*s*) indicates scale is according to the SQLBindParameter argument *ibScale*.
- The SQL_INTERVAL_*xxxx* types are supported only for computing values, such as in SQL expressions, not as database column types.
- Most applications should use SQL_VARCHAR rather than SQL_CHAR for binding character data. Use of SQL_CHAR may result in unwanted space padding to the full precision of the parameter type.
- Regarding TIME and TIMESTAMP, for example, an application can assume its time zone to be Pacific Standard Time. If the application is using TIME and TIMESTAMP values from Pacific Daylight Time or Eastern Standard Time, for example, the application must convert TIME and TIMESTAMP to Pacific Standard Time.

Binding Input Parameters

To bind input parameters to PL/SQL in TimesTen, use the *fSqlType*, *cbColDef*, and *ibScale* arguments (as applicable) of the ODBC SQLBindParameter function to specify data types.

This is in contrast to how SQL input parameters are supported, as noted in Parameter Type Assignments and Type Conversions.

In addition, use the *rgbValue*, *cbValueMax*, and *pcbValue* arguments of SQLBindParameter as follows for input parameters:

- *rgbValue*: Before statement execution, points to the buffer where the application places the parameter value to be passed to the application.
- *cbValueMax*: For character and binary data, indicates the maximum length of the incoming value that *rgbValue* points to, in bytes. For all other data types, *cbValueMax* is ignored, and the length of the value that *rgbValue* points to is determined by the length of the C data type specified in the *fCType* argument of SQLBindParameter.
- *pcbValue*: Points to a buffer that contains one of the following before statement execution:
  - The actual length of the value that *rgbValue* points to

    Note: For input parameters, this would be valid only for character or binary data.
  - SQL_NTS for a null-terminated string
  - SQL_NULL_DATA for a null value

Binding Output Parameters

To bind output parameters from PL/SQL in TimesTen, as noted for input parameters previously, use the *fSqlType*, *cbColDef*, and *ibScale* arguments (as applicable) of the ODBC SQLBindParameter function to specify data types.

In addition, use the *rgbValue*, *cbValueMax*, and *pcbValue* arguments of SQLBindParameter as follows for output parameters:

- *rgbValue*: During statement execution, points to the buffer where the value returned from the statement should be placed.
- *cbValueMax*: For character and binary data, indicates the maximum length of the outgoing value that *rgbValue* points to, in bytes. For all other data types, *cbValueMax* is ignored, and the length of the value that *rgbValue* points to is determined by the length of the C data type specified in the *fCType* argument of SQLBindParameter.

  Note that ODBC null-terminates all character data, even if the data is truncated. Therefore, when an output parameter has character data, *cbValueMax* must be large enough to accept the maximum data value plus a null terminator (one additional byte for CHAR and VARCHAR parameters, or two additional bytes for NCHAR and NVARCHAR parameters).
- *pcbValue*: Points to a buffer that contains one of the following after statement execution:
  - The actual length of the value that *rgbValue* points to (for all C types, not just character and binary data)

Note: This is the length of the full parameter value, regardless of whether the value can fit in the buffer that *rgbValue* points to.

– `SQL_NULL_DATA` for a null value

The following example shows how to prepare, bind, and execute a PL/SQL anonymous block.

- The anonymous block assigns bind parameter `a` the value '`abcde`' and bind parameter `b` the value `123`.
- `SQLPrepare` prepares the anonymous block.
- `SQLBindParameter` binds the first parameter (`a`) as an output parameter of type `SQL_VARCHAR` and binds the second parameter (`b`) as an output parameter of type `SQL_INTEGER`.
- `SQLExecute` executes the anonymous block.

```
{
  SQLHSTMT      hstmt;
  char          aval[11];
  SQLLEN        aval_len;
  SQLINTEGER    bval;
  SQLLEN        bval_len;

  SQLAllocStmt(hdbc, &hstmt);

  SQLPrepare(hstmt,
        (SQLCHAR*)"begin :a := 'abcde'; :b := 123; end;",
        SQL_NTS);

  SQLBindParameter(hstmt, 1, SQL_PARAM_OUTPUT, SQL_C_CHAR, SQL_VARCHAR,
         10, 0, (SQLPOINTER)aval, sizeof(aval), &aval_len);

  SQLBindParameter(hstmt, 2, SQL_PARAM_OUTPUT, SQL_C_SLONG, SQL_INTEGER,
         0, 0, (SQLPOINTER)&bval, sizeof(bval), &bval_len);

  SQLExecute(hstmt);
  printf("aval = [%s] (length = %d), bval = %d\n", aval, (int)aval_len, bval);
}
```

Binding Input/Output Parameters

To bind input/output parameters to and from PL/SQL in TimesTen, as noted for input and output parameters previously, use the *fSqlType*, *cbColDef*, and *ibScale* arguments (as applicable) of the ODBC `SQLBindParameter` function to specify data types.

In addition, use the *rgbValue*, *cbValueMax*, and *pcbValue* arguments of `SQLBindParameter` as follows for input/output parameters:

- *rgbValue*: This is first used before statement execution as described in Binding Input Parameters. Then it is used during statement execution as described in the preceding section, Binding Output Parameters. Note that for an input/output parameter, the outgoing value from a statement execution is the incoming value to the statement execution that immediately follows, unless that is overridden by the application. Also, for input/output values bound when you are using data-at-execution, the value of *rgbValue* serves as both the token that would be returned by the ODBC `SQLParamData` function and as the pointer to the buffer where the outgoing value is placed.

- *`cbValueMax`*: For character and binary data, this is first used as described in Binding Input Parameters. Then it is used as described in the preceding section, Binding Output Parameters. For all other data types, *`cbValueMax`* is ignored, and the length of the value that *`rgbValue`* points to is determined by the length of the C data type specified in the *`fCType`* argument of `SQLBindParameter`.

  Note that ODBC null-terminates all character data, even if the data is truncated. Therefore, when an input/output parameter has character data, *`cbValueMax`* must be large enough to accept the maximum data value plus a null terminator (one additional byte for `CHAR` and `VARCHAR` parameters, or two additional bytes for `NCHAR` and `NVARCHAR` parameters).

- *`pcbValue`*: This is first used before statement execution as described in Binding Input Parameters. Then it is used after statement execution as described in the preceding section, Binding Output Parameters.

> **Tip:**
>
> For character and binary data, carefully consider the value you use for *`cbValueMax`*. A value that is smaller than the actual buffer size may result in spurious truncation warnings. A value that is greater than the actual buffer size may cause the ODBC driver to overwrite the *`rgbValue`* buffer, resulting in memory corruption.

Binding of Duplicate Parameters in SQL Statements

TimesTen handles duplicate parameters in SQL. In TimesTen, multiple occurrences of the same parameter name in a SQL statement are considered to be distinct parameters. (This is consistent with Oracle Database support for binding duplicate parameters.)

> **Note:**
>
> - This discussion applies only to SQL statements issued directly from ODBC, not through PL/SQL, for example. (Regarding PL/SQL statements, see the next section Binding of Duplicate Parameters in PL/SQL Statements.)
> - "TimesTen mode" for binding duplicate parameters, and the `DuplicateBindMode` connection attribute, are deprecated.
> - The use of "`?`" for parameters, not supported in Oracle Database, is supported by TimesTen.

Consider this query:

```
SELECT * FROM employees
  WHERE employee_id < :a AND manager_id > :a AND salary < :b;
```

When parameter position numbers are assigned, a number is given to each parameter occurrence without regard to name duplication. The application must, at a minimum, bind a value for the first occurrence of each parameter name. For any subsequent occurrence of a given parameter name, the application has the following choices.

- It can bind a different value for the occurrence.
- It can leave the parameter occurrence unbound, in which case it takes the same value as the first occurrence.

In either case, each occurrence has a distinct parameter position number.

To use a different value for the second occurrence of `a` in the SQL statement above:

```
SQLBindParameter(..., 1, ...); /* first occurrence of :a */
SQLBindParameter(..., 2, ...); /* second occurrence of :a */
SQLBindParameter(..., 3, ...); /* occurrence of :b */
```

To use the same value for both occurrences of `a`:

```
SQLBindParameter(..., 1, ...); /* both occurrences of :a */
SQLBindParameter(..., 3, ...); /* occurrence of :b */
```

Parameter `b` is considered to be in position 3 regardless.

The `SQLNumParams` ODBC function returns 3 for the number of parameters in the example.

Binding of Duplicate Parameters in PL/SQL Statements

TimesTen handles duplicate parameters in PL/SQL. In PL/SQL, you bind a value for each unique parameter name. An application executing the following block, for example, would bind only one parameter, corresponding to `:a`.

Discussion in the preceding section, Binding of Duplicate Parameters in SQL Statements, does not apply to PL/SQL, which has its own semantics.

```
DECLARE
   x NUMBER;
   y NUMBER;
BEGIN
   x:=:a;
   y:=:a;
END;
```

An application executing the following block would also bind only one parameter:

```
BEGIN
   INSERT INTO tab1 VALUES(:a, :a);
END
```

And the same for the following `CALL` statement:

```
...CALL proc(:a, :a)...
```

An application executing the following block would bind two parameters, with `:a` as the first parameter and `:b` as the second parameter. The second parameter in each `INSERT` statement would take the same value as the first parameter in the first `INSERT` statement:

```
BEGIN
   INSERT INTO tab1 VALUES(:a, :a);
```

```
   INSERT INTO tab1 VALUES(:b, :a);
END
```

Considerations for Floating Point Data

There are considerations for floating point data.

The `BINARY_DOUBLE` and `BINARY_FLOAT` data types store and retrieve the IEEE floating point values `Inf`, `-Inf`, and `NaN`. If an application uses a C language facility such as `printf`, `scanf`, or `strtod` that requires conversion to character data, the floating point values are returned as "INF", "-INF", and "NAN". These character strings cannot be converted back to floating point values.

Using SQL_WCHAR and SQL_WVARCHAR With a Driver Manager

This section discusses how to avoid possible error conditions when using `SQL_WCHAR` or `SQL_WVARCHAR` with a driver manager.

Applications using the Windows driver manager may encounter errors from `SQLBindParameter` with SQL state `S1004` (SQL data type out of range) when passing an *`fSqlType`* value of `SQL_WCHAR` or `SQL_WVARCHAR`. This problem can be avoided by passing one of the following values for *`fSqlType`* instead.

- `SQL_WCHAR_DM_SQLBINDPARAMETER_BYPASS` instead of `SQL_WCHAR`
- `SQL_WVARCHAR_DM_SQLBINDPARAMETER_BYPASS` instead of `SQL_WVARCHAR`

These type codes are semantically identical to `SQL_WCHAR` and `SQL_WVARCHAR` but avoid the error from the Windows driver manager. They can be used in applications that link with the driver manager or link directly with the TimesTen ODBC direct driver or ODBC client driver.

See SQLBindParameter Function.

Working With REF CURSORs

REF CURSOR is a PL/SQL concept, a handle to a cursor over a SQL result set that can be passed between PL/SQL and an application. In TimesTen, the cursor can be opened in PL/SQL then the REF CURSOR can be passed to the application. The results can be processed in the application using ODBC calls.
This is an `OUT` REF CURSOR (an `OUT` parameter with respect to PL/SQL). The REF CURSOR is attached to a statement handle, enabling applications to describe and fetch result sets using the same APIs as for any result set.

Take the following steps to use a REF CURSOR. Assume a PL/SQL statement that returns a cursor through a REF CURSOR `OUT` parameter. Note that REF CURSORs use the same basic steps of prepare, bind, execute, and fetch as in the cursor example in Steps to Prepare and Execute Queries and Work With Cursors in ODBC.

1. Prepare the PL/SQL statement, using `SQLPrepare`, to be associated with the first statement handle.
2. Bind each parameter of the statement, using `SQLBindParameter`. When binding the REF CURSOR output parameter, use an allocated second statement handle as *`rgbValue`*, the pointer to the data buffer.

   The *`pcbValue`*, *`ibScale`*, *`cbValueMax`*, and *`pcbValue`* arguments are ignored for REF CURSORs.

See SQLBindParameter Function and Binding Output Parameters.

3. Call `SQLBindCol` to bind result columns to local variable storage.
4. Call `SQLExecute` to execute the statement.
5. Call `SQLFetch` to fetch the results. After a REF CURSOR is passed from PL/SQL to an application, the application can describe and fetch the results as it would for any result set.
6. Use `SQLFreeStmt` to free the statement handle.

These steps are demonstrated in the example that follows. Refer to ODBC API reference documentation for details on these functions. See PL/SQL REF CURSORs in *Oracle TimesTen In-Memory Database PL/SQL Developer's Guide*.

> **Note:**
>
> For passing REF CURSORs between PL/SQL and an application, TimesTen supports only `OUT` REF CURSORs, from PL/SQL to the application.

This example, using a REF CURSOR in a loop, demonstrates the basic steps of preparing a query, binding parameters, executing the query, binding results to local variable storage, and fetching the results. Error handling is omitted for simplicity. In addition to the ODBC functions summarized earlier, this example uses `SQLAllocStmt` to allocate memory for a statement handle.

```
refcursor_example(SQLHDBC hdbc)
{
  SQLCHAR*      stmt_text;
  SQLHSTMT      plsql_hstmt;
  SQLHSTMT      refcursor_hstmt;
  SQLINTEGER    deptid;
  SQLINTEGER    depts[3] = {10,30,40};
  SQLINTEGER    empid;
  SQLCHAR       lastname[30];
  SQLINTEGER    i;

  /* allocate 2 statement handles: one for the plsql statement and
   * one for the ref cursor */
  SQLAllocStmt(hdbc, &plsql_hstmt);
  SQLAllocStmt(hdbc, &refcursor_hstmt);

  /* prepare the plsql statement */
  stmt_text = (SQLCHAR*)
    "begin "
      "open :refc for "
        "select employee_id, last_name "
        "from employees "
        "where department_id = :dept; "
    "end;";
  SQLPrepare(plsql_hstmt, stmt_text, SQL_NTS);

  /* bind parameter 1 (:refc) to refcursor_hstmt */
  SQLBindParameter(plsql_hstmt, 1, SQL_PARAM_OUTPUT, SQL_C_REFCURSOR,
                   SQL_REFCURSOR, 0, 0, refcursor_hstmt, 0, 0);

  /* bind parameter 2 (:deptid) to local variable deptid */
```

```
  SQLBindParameter(plsql_hstmt, 2, SQL_PARAM_INPUT, SQL_C_SLONG,
                   SQL_INTEGER, 0, 0, &deptid, 0, 0);

  /* loop through values for :deptid */
  for (i=0; i<3; i++)
  {
     deptid = depts[i];

     /* execute the plsql statement */
     SQLExecute(plsql_hstmt);
     /*
      * The result set is now attached to refcursor_hstmt.
      * Bind the result columns and fetch the result set.
      */

     /* bind result column 1 to local variable empid */
     SQLBindCol(refcursor_hstmt, 1, SQL_C_SLONG,
                (SQLPOINTER)&empid, 0, 0);

     /* bind result column 2 to local variable lastname */
     SQLBindCol(refcursor_hstmt, 2, SQL_C_CHAR,
                (SQLPOINTER)lastname, sizeof(lastname), 0);

     /* fetch the result set */
     while(SQLFetch(refcursor_hstmt) != SQL_NO_DATA_FOUND){
       printf("%d, %s\n", empid, lastname);
     }

     /* close the ref cursor statement handle */
     SQLFreeStmt(refcursor_hstmt, SQL_CLOSE);
  }

  /* drop both handles */
  SQLFreeStmt(plsql_hstmt, SQL_DROP);
  SQLFreeStmt(refcursor_hstmt, SQL_DROP);
}
```

Working With DML Returning (RETURNING INTO Clause)

You can use a `RETURNING INTO` clause, referred to as *DML returning*, with an `INSERT`, `UPDATE`, or `DELETE` statement to return specified items from a row that was affected by the action.

This eliminates the need for a subsequent `SELECT` statement and separate round trip in case, for example, you want to confirm what was affected by the action.

With ODBC, DML returning is limited to returning items from a single-row operation. The clause returns the items into a list of output parameters. Bind the output parameters as discussed in Parameter Binding and Statement Execution.

SQL syntax and restrictions for the `RETURNING INTO` clause in TimesTen are documented as part of INSERT, UPDATE, and DELETE in *Oracle TimesTen In-Memory Database SQL Reference*.

Refer to RETURNING INTO Clause in *Oracle Database PL/SQL Language Reference* for details about DML returning.

This example is adapted from the example in Committing Changes to the Database in ODBC, with bold text highlighting key portions.

```
void
update_example(SQLHDBC hdbc)
{
   SQLCHAR*      stmt_text;
   SQLHSTMT      hstmt;
   SQLINTEGER    raise_pct;
   char          hiredate_str[30];
   char          last_name[30];
   SQLLEN        hiredate_len;
   SQLLEN        numrows;

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

   /* prepare an update statement to give a raise to one employee hired
      before a given date and return that employee's last name */
   stmt_text = (SQLCHAR*)
     "update employees "
     "set salary = salary * ((100 + :raise_pct) / 100.0) "
     "where hire_date < :hiredate and rownum = 1 returning last_name into "
                       ":last_name";
   SQLPrepare(hstmt, stmt_text, SQL_NTS);

   /* bind parameter 1 (:raise_pct) to variable raise_pct */
   SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_SLONG,
                    SQL_DECIMAL, 0, 0, (SQLPOINTER)&raise_pct, 0, 0);

   /* bind parameter 2 (:hiredate) to variable hiredate_str */
   SQLBindParameter(hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR,
                    SQL_TIMESTAMP, 0, 0, (SQLPOINTER)hiredate_str,
                    sizeof(hiredate_str), &hiredate_len);
   /* bind parameter 3 (:last_name) to variable last_name */
   SQLBindParameter(hstmt, 3, SQL_PARAM_OUTPUT, SQL_C_CHAR,
                    SQL_VARCHAR, 30, 0, (SQLPOINTER)last_name,
                    sizeof(last_name), NULL);
   /* set parameter values to give a 10% raise to an employee hired before
    * January 1, 1996. */
   raise_pct = 10;
   strcpy(hiredate_str, "1996-01-01");
   hiredate_len = SQL_NTS;

   /* execute the update statement */
   SQLExecute(hstmt);

   /* tell us who the lucky person is */
   printf("Gave raise to %s.\n", last_name );

   /* drop the statement handle */
   SQLFreeStmt(hstmt, SQL_DROP);

   /* commit the changes */
   SQLTransact(henv, hdbc, SQL_COMMIT);

}
```

This returns "King" as the recipient of the raise.

Working With rowids

Each row in a database table has a unique identifier known as its rowid. An application can retrieve the rowid of a row from the `ROWID` pseudocolumn. Rowids can be represented in either binary or character format.

An application can specify literal rowid values in SQL statements, such as in `WHERE` clauses, as `CHAR` constants enclosed in single quotes.

As noted in Table 2-2, the ODBC SQL type `SQL_ROWID` corresponds to the SQL type `ROWID`.

For parameters and result set columns, rowids are convertible to and from the C types `SQL_C_BINARY`, `SQL_C_WCHAR`, and `SQL_C_CHAR`. `SQL_C_CHAR` is the default C type for rowids. The size of a rowid would be 12 bytes as `SQL_C_BINARY`, 18 bytes as `SQL_C_CHAR`, and 36 bytes as `SQL_C_WCHAR`.

Refer to ROWID Data Type and ROWID Pseudocolumn in *Oracle TimesTen In-Memory Database SQL Reference*.

> **Note:**
>
> TimesTen does not support the PL/SQL type `UROWID`.

Large Objects (LOBs)

TimesTen Classic supports LOBs (large objects). This includes CLOBs (character LOBs), NCLOBs (national character LOBs), and BLOBs (binary LOBs).

These sections provide an overview of LOBs and discuss their use in ODBC, covering these topics:

- About LOBs
- Differences Between TimesTen LOBs and Oracle Database LOBs
- LOB Programmatic Approaches and Programming Interfaces
- Using the LOB Simple Data Interface in ODBC
- Using the LOB Piecewise Data Interface in ODBC
- Passthrough LOBs in ODBC

You can also refer to the following:

- LOBs in TimesTen OCI and LOBs in TimesTen Pro*C/C++ for information specific to those APIs
- LOB Data Types in *Oracle TimesTen In-Memory Database SQL Reference* for additional information about LOBs in TimesTen
- *Oracle Database SecureFiles and Large Objects Developer's Guide* for general information about programming with LOBs (but not specific to TimesTen functionality)

About LOBs

A LOB is a large binary object (BLOB) or character object (CLOB or NCLOB). In TimesTen, a BLOB can be up to 16 MB and a CLOB or NCLOB up to 4 MB. LOBs in TimesTen have essentially the same functionality as in Oracle Database, except as noted otherwise.

See Differences Between TimesTen LOBs and Oracle Database LOBs.

LOBs may be either persistent or temporary. A persistent LOB exists in a LOB column in the database. A temporary LOB exists only within an application. There are circumstances where a temporary LOB is created implicitly. For example, if a `SELECT` statement selects a LOB concatenated with an additional string of characters, TimesTen creates a temporary LOB to contain the concatenated data. In TimesTen ODBC, any temporary LOBs are managed implicitly.

Temporary LOBs are stored in the TimesTen temporary data region.

Differences Between TimesTen LOBs and Oracle Database LOBs

There are key differences between TimesTen and Oracle Database LOB functionality.

- In TimesTen, a LOB used in an application does not remain valid past the end of the transaction. All such LOBs are invalidated after a commit or rollback, whether explicit or implicit. This includes after any DDL statement.
- TimesTen does not support BFILEs, SecureFiles, array reads and writes for LOBs, or callback functions for LOBs.
- TimesTen does not support binding arrays of LOBs.
- TimesTen does not support batch processing of LOBs.
- Relevant to BLOBs, there are differences in the usage of hexadecimal literals in TimesTen. see the description of *`HexadecimalLiteral`* in Constants in *Oracle TimesTen In-Memory Database SQL Reference*.

LOB Programmatic Approaches and Programming Interfaces

There are three programmatic approaches for accessing LOBs from TimesTen in a C or C++ program.

- Simple data interface (ODBC, OCI, Pro*C/C++, TTClasses): Use binds and defines, as with other scalar types, to transfer LOB data in a single chunk.
- Piecewise data interface (ODBC): Use advanced forms of binds and defines to transfer LOB data in multiple pieces. This is sometimes referred to as *streaming* or using *data-at-exec* (at program execution time). TimesTen supports the piecewise data interface through polling loops to go piece-by-piece through the LOB data. (Another piecewise approach, using callback functions, is supported by Oracle Database but not by TimesTen.)

  The piecewise interface enables applications to access LOB data in portions, piece by piece. An application binds parameters or defines results similarly to how those actions are performed for the simple data interface, but indicates that the data is to be provided or retrieved at program execution time ("at exec"). In TimesTen, you can implement the piecewise data interface through a polling loop that is repeated until all the LOB data has been read or written.

- LOB locator interface (OCI, Pro*C/C++): Select LOB locators using SQL then access LOB data through APIs that are similar conceptually to those used in accessing a file system. Using the LOB locator interface, you can work with LOB data in pieces or in single chunks. See LOBs in TimesTen OCI and LOBs in TimesTen Pro*C/C++.

The LOB locator interface offers the most utility if it is feasible for you to use it.

Using the LOB Simple Data Interface in ODBC

For the simple data interface in ODBC, use `SQLBindParameter` to bind parameters and `SQLBindCol` to define result columns.

The application can bind or define using a SQL type that is compatible with the corresponding variable type, as follows.

- For BLOB data, use SQL type `SQL_LONGVARBINARY` and C type `SQL_C_BINARY`.
- For CLOB data, use SQL type `SQL_LONGVARCHAR` and C type `SQL_C_CHAR`.
- For NCLOB data, use SQL type `SQL_WLONGVARCHAR` and C type `SQL_C_WCHAR`.

`SQLBindParameter` and `SQLBindCol` calls for LOB data would be very similar to such calls for other data types, discussed earlier in this chapter.

> **Note:**
>
> Binding a CLOB or NCLOB with a C type of `SQL_C_BINARY` is prohibited.

Using the LOB Piecewise Data Interface in ODBC

For the piecewise data interface in ODBC, use `SQLParamData` with `SQLPutData` in a polling loop to bind parameters and `SQLGetData` in a polling loop to retrieve results.

See the preceding section, Using the LOB Simple Data Interface in ODBC, for information about supported SQL and C data types for BLOBs, CLOBs, and NCLOBs.

> **Note:**
>
> Similar piecewise data access has already been supported for the various APIs in previous releases of TimesTen, for `var` data types.

This program excerpt uses `SQLPutData` with `SQLParamData` in a polling loop to insert LOB data piece-by-piece into the database. The `CLOB` column contains the value "123ABC" when the code is executed.

```
...
  /* create a table */
  create_stmt = "create table clobtable ( c clob )";
  rc = SQLExecDirect(hstmt, (SQLCHAR *)create_stmt, SQL_NTS);
  if(rc != SQL_SUCCESS){/* ...error handling... */}

  /* initialize an insert statement */
  insert_stmt = "insert into clobtable values(?)";
```

```
  rc = SQLPrepare(hstmt, (SQLCHAR *)insert_stmt, SQL_NTS);
  if(rc != SQL_SUCCESS){/* ...error handling... */}

  /* bind the parameter and specify that we will be using
   * SQLParamData/SQLPutData */
  rc = SQLBindParameter(
    hstmt,            /* statement handle */
    1,                /* colnum number */
    SQL_PARAM_INPUT,  /* param type */
    SQL_C_CHAR,       /* C type */
    SQL_LONGVARCHAR,  /* SQL type (ignored) */
    2,                /* precision (ignored) */
    0,                /* scale (ignored) */
    0,                /* putdata token */
    0,                /* ignored */
    &pcbvalue);       /* indicates use of SQLPutData */
  if(rc != SQL_SUCCESS){/* ...error handling... */}

  pcbvalue = SQL_DATA_AT_EXEC;

  /* execute the statement -- this should return SQL_NEED_DATA */
  rc = SQLExecute(hstmt);
  if(rc != SQL_NEED_DATA){/* ...error handling... */}

  /* while we still have parameters that need data... */
  while((rc = SQLParamData(hstmt, &unused)) == SQL_NEED_DATA){

    memcpy(char_buf, "123", 3);
    rc = SQLPutData(hstmt, char_buf, 3);
    if(rc !=  SQL_SUCCESS){/* ...error handling... */}

    memcpy(char_buf, "ABC", 3);
    rc = SQLPutData(hstmt, char_buf, 3);
    if(rc !=  SQL_SUCCESS){/* ...error handling... */}

  }
...
```

Passthrough LOBs in ODBC

Passthrough LOBs, which are LOBs in Oracle Database accessed through TimesTen, are exposed as TimesTen LOBs and are supported by TimesTen in much the same way that any TimesTen LOB is supported.

Note the following:

- TimesTen LOB size limitations do not apply to storage of LOBs in the Oracle database through passthrough.
- As with TimesTen local LOBs, a passthrough LOB used in an application does not remain valid past the end of the transaction.

Using CALL to Execute Procedures and Functions

TimesTen Classic supports each of these syntax formats from any of its programming interfaces to call PL/SQL procedures (*procname*) or PL/SQL functions (*funcname*) that are standalone or part of a package, or to call TimesTen built-in procedures (*procname*).

```
CALL procname[(argumentlist)]

CALL funcname[(argumentlist)] INTO :returnparam

CALL funcname[(argumentlist)] INTO ?
```

TimesTen ODBC also supports each of the following syntax formats:

```
{ CALL procname[(argumentlist)] }

{ ? = [CALL] funcname[(argumentlist)] }

{ :returnparam = [CALL] funcname[(argumentlist)] }
```

The following ODBC example calls the TimesTen `ttCkpt` built-in procedure.

```
rc = SQLExecDirect (hstmt, (SQLCHAR*) "call ttCkpt",SQL_NTS);
```

These examples call a PL/SQL procedure `myproc` with two parameters:

```
rc = SQLExecDirect(hstmt, (SQLCHAR*) "{ call myproc(:param1, :param2) }",SQL_NTS);

rc = SQLExecDirect(hstmt, (SQLCHAR*) "{ call myproc(?, ?) }",SQL_NTS);
```

The following shows several ways to call a PL/SQL function `myfunc`:

```
rc = SQLExecDirect (hstmt, (SQLCHAR*) "CALL myfunc() INTO :retparam",SQL_NTS);

rc = SQLExecDirect (hstmt, (SQLCHAR*) "CALL myfunc() INTO ?",SQL_NTS);

rc = SQLExecDirect (hstmt, (SQLCHAR*) "{ :retparam = myfunc() }",SQL_NTS);

rc = SQLExecDirect (hstmt, (SQLCHAR*) "{ ? = myfunc() }",SQL_NTS);
```

See CALL in *Oracle TimesTen In-Memory Database SQL Reference*.

> **Note:**
>
> - A user's own procedure takes precedence over a TimesTen built-in procedure with the same name, but it is best to avoid such naming conflicts.
> - TimesTen does not support using `SQL_DEFAULT_PARAM` with `SQLBindParameter` for a `CALL` statement.

Timeouts and Thresholds for Executing SQL Statements

TimesTen offers two ways to limit the time for SQL statements or procedure calls to execute, by either setting a timeout duration or setting a threshold duration. This applies to any `SQLExecute`, `SQLExecDirect`, or `SQLFetch` call.

This section covers these topics:

- Setting a Timeout Duration for SQL Statements
- Setting a Threshold Duration for SQL Statements

Setting a Timeout Duration for SQL Statements

To control how long SQL statements should execute before timing out, you can set the `SQL_QUERY_TIMEOUT` option using a `SQLSetStmtOption` or `SQLSetConnectOption` call to specify a timeout value, in seconds. A value of 0 indicates no timeout. If a timeout duration is reached, the statement stops executing and an error is thrown.

> **Note:**
>
> Despite the name, this timeout value applies to any executable SQL statement, not just queries.

In TimesTen, you can specify this timeout value for a connection, and therefore any statement on the connection, by using either the `SQLQueryTimeout` general connection attribute (in seconds) or the `SQLQueryTimeoutMsec` general connection attribute (in milliseconds). The default value of each is 0, for no timeout. (Also see SQLQueryTimeout and SQLQueryTimeoutMsec in *Oracle TimesTen In-Memory Database Reference*.)

Despite the names, these timeout values apply to any executable SQL statement, not just queries.

A call to `SQLSetConnectOption` with the `SQL_QUERY_TIMEOUT` option overrides any previous query timeout setting. A call to `SQLSetStmtOption` with the `SQL_QUERY_TIMEOUT` option overrides the connection setting for the particular statement.

The query timeout limit has effect only when a SQL statement is actively executing. A timeout does not occur during commit or rollback. For transactions that update, insert, or delete a large number of rows, the commit or rollback phases may take a long time to complete. During that time the timeout value is ignored.

See Choose SQL and PL/SQL Timeout Values in *Oracle TimesTen In-Memory Database Operations Guide*.

> **Note:**
>
> If both a lock timeout value and a SQL query timeout value are specified, the lesser of the two values causes a timeout first. Regarding lock timeouts, you can refer to ttLockWait (built-in procedure) or LockWait (general connection attribute) in *Oracle TimesTen In-Memory Database Reference*, or to Check for Deadlocks and Timeouts in *Oracle TimesTen In-Memory Database Monitoring and Troubleshooting Guide*.

Setting a Threshold Duration for SQL Statements

You can configure TimesTen to write a warning to the support log when the execution of a SQL statement exceeds a specified time duration, in seconds. Execution continues and is not affected by the threshold.

By default, the application obtains the threshold from the `QueryThreshold` general connection attribute setting (refer to QueryThreshold in *Oracle TimesTen In-Memory Database Reference*). The default value is 0, for no warnings. Setting the `TT_QUERY_THRESHOLD` option in a `SQLSetConnectOption` call overrides the connection attribute setting for the current connection. Despite the name, the threshold applies to any executable SQL statement.

To set the threshold with `SQLSetConnectOption`:

```
RETCODE SQLSetConnectOption(hdbc, TT_QUERY_THRESHOLD, seconds);
```

Setting the `TT_QUERY_THRESHOLD` option in a `SQLSetStmtOption` call overrides the connection attribute setting, and any setting through `SQLSetConnectOption`, for the statement. It applies to SQL statements executed using the ODBC statement handle.

To set the threshold with `SQLSetStmtOption`:

```
RETCODE SQLSetStmtOption(hstmt, TT_QUERY_THRESHOLD, seconds);
```

You can retrieve the current value of `TT_QUERY_THRESHOLD` by using the `SQLGetConnectOption` or `SQLGetStmtOption` ODBC function:

```
RETCODE SQLGetConnectOption(hdbc, TT_QUERY_THRESHOLD, paramvalue);
```

```
RETCODE SQLGetStmtOption(hstmt, TT_QUERY_THRESHOLD, paramvalue);
```

Configuring the Result Set Buffer Size in Client/Server Using ODBC

For data returned from a `SELECT` statement in client/server, the buffer size for the data returned to the client is configurable to allow adjustments for better performance. (In earlier releases, the buffer size could not be changed.)

The buffer size can be set in terms of either rows of data or bytes of data. The lower limit takes precedence. It is suggested to use one limit and set the other to a value high enough to ensure that it is not reached first.

TimesTen provides these ODBC statement attributes:

- `TT_NET_MSG_MAX_ROWS`: Buffer size in rows (default 8192)
- `TT_NET_MSG_MAX_BYTES`: Buffer size in bytes (default 2097152, or 2 MB)

These can also be set at the connection level. When you set them on a connection handle, the new values will apply to any future statement handles created on the connection and also to any existing statement handles on the connection. It is recommended, though, to set them at statement level (or at connection level only to serve as initial values for statement handles to be created).

The attributes are supported either as ODBC 3.5 attributes, using `SQLSetStmtAttr()` or `SQLSetConnectAttr()`, or as ODBC 2.5 options, using `SQLSetStmtOption()` or `SQLSetConnectOption()`. You can retrieve the values with ODBC “get” functions only on statement handles, using `SQLGetStmtAttr()` in ODBC 3.5 or `SQLGetStmtOption()` in ODBC 2.5.

Here is an example:

```
SQLRETURN rc = SQL_SUCCESS;
/* Double the default number of rows */
UDWORD maxRows = 16384;
```

```
....
rc = SQLSetConnectAttr(hdbc, TT_NET_MSG_MAX_ROWS, (SQLPOINTER)
maxRows, SQL_IS_INTEGER);
```

> **Note:**
>
> - These attributes correspond to TimesTen connection attributes `TT_NetMsgMaxRows` and `TT_NetMsgMaxBytes`, which you can set in a TimesTen connection string or DSN, to serve as initial values for any statements created on the connection.
> - The minimum value of each attribute is 1 and at least one row is always returned. Setting either to a value of 0 results in the default value being used. There are no maximum settings other than the maximum value of the data type (32-bit unsigned integer).
> - If a client version that supports these attributes connects to a server version that does not, any settings are ignored.

Features for Cache

There are features related to the use of cache in TimesTen Classic.

- Setting Temporary Passthrough Level With the ttOptSetFlag Built-In Procedure
- Determining Passthrough Status
- Retrieving Information About Cache Groups

See *Oracle TimesTen In-Memory Database Cache Guide* for information about cache.

See PassThrough and Setting a Passthrough Level in *Oracle TimesTen In-Memory Database Cache Guide*.

Setting Temporary Passthrough Level With the ttOptSetFlag Built-In Procedure

TimesTen provides the `ttOptSetFlag` built-in procedure for setting various flags, including the `PassThrough` flag to temporarily set the passthrough level.

You can use `ttOptSetFlag` to set `PassThrough` in a C application as in the following example that sets the passthrough level to 1. The setting affects all statements that are prepared until the end of the transaction.

```
rc = SQLExecDirect (hstmt, "call ttOptSetFlag ('PassThrough', 1)",SQL_NTS);
```

See ttOptSetFlag in *Oracle TimesTen In-Memory Database Reference*.

Determining Passthrough Status

You can call the `SQLGetStmtOption` ODBC function with the `TT_STMT_PASSTHROUGH_TYPE` statement option to determine whether a SQL statement is to be executed in the TimesTen database or passed through to the Oracle database for execution.

This is shown in the following example.

```
rc = SQLGetStmtOption(hStmt, TT_STMT_PASSTHROUGH_TYPE, &passThroughType);
```

You can make this call after preparing the SQL statement. It is useful with `PassThrough` settings of 1 or 2, where the determination of whether a statement is actually passed through is not made until compilation time. If `TT_STMT_PASSTHROUGH_NONE` is returned, the statement is to be executed in TimesTen. If `TT_STMT_PASSTHROUGH_ORACLE` is returned, the statement is to be passed through to Oracle Database for execution.

See Setting a Passthrough Level in *Oracle TimesTen In-Memory Database Cache Guide*.

> **Note:**
>
> `TT_STMT_PASSTHROUGH_TYPE` is supported with `SQLGetStmtOption` only, not with `SQLSetStmtOption`.

Retrieving Information About Cache Groups

When using cache, following the execution of a `FLUSH CACHE GROUP`, `LOAD CACHE GROUP`, `REFRESH CACHE GROUP`, or `UNLOAD CACHE GROUP` statement, the ODBC function `SQLRowCount` returns the number of cache instances that were flushed, loaded, refreshed, or unloaded.

For related information, see Determining the Number of Cache Instances Affected by an Operation in *Oracle TimesTen In-Memory Database Cache Guide*.

Refer to ODBC API reference documentation for general information about `SQLRowCount`.

Error Handling

There are methods to check and handle different types of errors.

This section includes the following topics:

- Checking for Errors
- Error and Warning Levels
- Recovery After Fatal Errors
- Transient Errors (ODBC)

Checking for Errors

An application should check for errors and warnings on every call. This saves considerable time and effort during development and debugging. The sample applications provided with TimesTen show examples of error checking.

See About TimesTen Quick Start and Sample Applications.

Errors can be checked using either the TimesTen error code (error number) or error string, as defined in the *installation_dir*/include/tt_errCode.h file. Entries are in the following format:

```
#define tt_ErrMemoryLock            712
```

See List of Errors and Warnings in *Oracle TimesTen In-Memory Database Error Messages and SNMP Traps*.

After calling an ODBC function, check the return code. If the return code is not `SQL_SUCCESS`, use an error-handling routine that calls the ODBC function `SQLError` to retrieve the errors on the relevant ODBC handle. A single ODBC call may return multiple errors. The application should be written to return all errors by repeatedly calling the `SQLError` function until all errors are read from the error stack. Continue calling `SQLError` until the return code is `SQL_NO_DATA_FOUND`. (`SQL_NO_DATA_FOUND` is defined in `sqlext.h`, which is included by `timesten.h`.)

The following example shows that after a call to `SQLAllocConnect`, you can check for an error condition. If one is found, an error message is displayed and program execution is terminated.

```
rc = SQLAllocConnect(henv, &hdbc);

if (rc != SQL_SUCCESS) {
  handleError(rc, henv, hdbc, hstmt, err_buf, &native_error);
  fprintf(stderr,
          "Unable to allocate a connection handle:\n%s\n",
          err_buf);
  exit(-1);
}
```

Refer to ODBC API reference documentation for details about the `SQLError` function and its arguments.

See Retrieving Errors and Warnings in *Oracle TimesTen In-Memory Database Error Messages and SNMP Traps*.

Error and Warning Levels

When operations are not completely successful, TimesTen can return certain types of errors.

- Fatal Errors
- Non-Fatal Errors
- Warnings

> **Note:**
>
> In some cases of an unusual termination, such as process failure, no error is returned, but TimesTen automatically rolls back the transactions of the failed process.

Fatal Errors

Fatal errors are those that make the database inaccessible until after error recovery.

When a fatal error occurs, all database connections are required to disconnect. No further operations may complete. Fatal errors are indicated by TimesTen error codes 846 and 994. Error handling for these errors should be different from standard error handling. In particular, the application error-handling code should roll back the current transaction and disconnect from the database.

Also see Recovery After Fatal Errors.

Non-Fatal Errors

Non-fatal errors include errors such as an `INSERT` statement that violates unique constraints. This category also includes some classes of application and process failures.

TimesTen returns non-fatal errors through the standard error-handling process. Applications should check for errors and appropriately handle them.

When a database is affected by a non-fatal error, an error may be returned and the application should take appropriate action.

An application can handle non-fatal errors by modifying its actions or, in some cases, rolling back one or more offending transactions.

Warnings

TimesTen returns warnings when something unexpected occurs that you may want to know about.

Here are some events that cause TimesTen to issue a warning:

- Checkpoint failure
- Use of a deprecated feature
- Truncation of some data
- Execution of a recovery process upon connect
- Replication return receipt timeout

Application developers should have code that checks for warnings, as they can indicate application problems.

Recovery After Fatal Errors

When fatal errors occur, TimesTen performs a full cleanup and recovery procedure.

- Every connection to the database is invalidated. To avoid out-of-memory conditions in the server, applications are required to disconnect from the invalidated database. Shared memory from the old TimesTen instance is not freed until all active connections at the time of the error have disconnected. Inactive applications still connected to the old TimesTen instance may have to be manually terminated.
- The database is recovered from the checkpoint and transaction log files upon the first subsequent initial connection.
- The recovered database reflects the state of all durably committed transactions and possibly some transactions that were committed non-durably.
- No uncommitted or rolled back transactions are reflected.

Transient Errors (ODBC)

TimesTen automatically resolves most transient errors (which is particularly important for TimesTen Scaleout), but if your application detects the following `SQLSTATE` value, it is suggested to retry the current transaction.

- `TT005`: Transient transaction failure due to unavailability of resource. Roll back the transaction and try it again.

> **Note:**
>
> - Search the entire error stack for errors returning these SQL states before deciding whether it is appropriate to retry.
> - The example in Implementing Failover Delay and Retry Settings also shows how to retry for transient errors.

In ODBC 3.5, `SQLSTATE` is returned by the `SQLGetDiagRec` function or indicated in the `SQL_DIAG_SQLSTATE` field of the `SQLGetDiagField` function. In ODBC 2.5, `SQLSTATE` is returned by the `SQLError` function. This `SQLSTATE` may be encountered by any of the following functions. Unless indicated otherwise, these functions apply to either ODBC 2.5 or ODBC 3.5.

- Catalog functions (such as `SQLTables` and `SQLColumns`)
- `SQLCancel`
- `SQLCloseCursor` (ODBC 3.5)
- `SQLDisconnect`
- `SQLExecDirect`
- `SQLExecute`
- `SQLFetch`
- `SQLFetchScroll` (ODBC 3.5)
- `SQLFreeStmt` (ODBC 2.5)
- `SQLGetData`
- `SQLGetInfo`
- `SQLPrepare`
- `SQLPutData`
- `SQLEndTran` (ODBC 3.5)
- `SQLTransact` (ODBC 2.5)

ODBC Support for Automatic Client Failover

There is ODBC support of the TimesTen implementation of automatic client failover, as it applies to application developers.

For TimesTen Scaleout, see Client Connection Failover in *Oracle TimesTen In-Memory Database Scaleout User's Guide*. For TimesTen Classic, see Using Automatic Client Failover in *Oracle TimesTen In-Memory Database Operations Guide*.

About Automatic Client Failover

Automatic client failover is for use in high availability scenarios, for either TimesTen Scaleout or TimesTen Classic. There are two scenarios for TimesTen Classic, one with active standby pair replication and one referred to as *generic automatic client failover*.

If there is a failure of the database or database element to which the client is connected, then failover (connection transfer) to an alternate database or database element occurs:

- For TimesTen Scaleout, failover is to an element from a list returned by TimesTen of available elements in the grid.
- For TimesTen Classic with active standby replication, failover is to the new active (original standby) database.
- For TimesTen Classic using generic automatic client failover, where you can ensure that the schema and data are consistent on both databases, failover is to a database from a list that is configured in the client `odbc.ini` file.

  A typical use case for generic automatic failover is a set of databases using read-only caching, where each database has the same set of cached data. For example, if you have several read-only cache groups, then you would create the same read-only cache groups on all TimesTen Classic databases included in the list of failover servers. When the client connection fails over to an alternate TimesTen database, the cached data is consistent because cache operations automatically refresh the data (as needed) from the Oracle database.

Applications are automatically reconnected to the new database or database element. TimesTen provides features that enable applications to be alerted when this happens, so they can take any appropriate action.

> **Note:**
>
> - Automatic client failover applies only to client/server connections. The functionality described here does not apply to direct connections.
> - Automatic client failover is complementary to Oracle Clusterware in situations where Oracle Clusterware is used, though the two features are not dependent on each other. Refer to Using Oracle Clusterware to Manage Active Standby Pairs in *Oracle TimesTen In-Memory Database Replication Guide*.

Features and Functionality of ODBC Support for Automatic Client Failover

If a database or database element to which a client is connected fails, failover to an alternate database or database element occurs.

When failover occurs, be aware of the following:

- The client has a new connection but using the same ODBC connection handle. No state from the original connection is preserved, however, other than the handle itself. The application must open new ODBC statement handles and descriptor handles.
- If you register a failover callback function (see Implementing and Registering an ODBC Failover Callback Function.), a failover listener thread will be created within the client process to listen for failover event and invoke the callback function.

All client statement handles from the original connection are marked as invalid. API calls on these statement handles generally return `SQL_ERROR` with distinctive failover error codes defined in `tt_errCode.h`:

- Native error 30105 with SQL state 08006
- Native error 47137

The exception to this is for `SQLError`, `SQLFreeStmt`, `SQLGetDiagRec`, and `SQLGetDiagField` calls (depending on your version of ODBC), which behave typically.

In addition, note the following:

- The socket to the original database or database element is closed. There is no need to call `SQLDisconnect`. TimesTen performs the equivalent, cleaning up the connection handle and confirming resources are freed.
- In connecting to the new TimesTen database or database element, the same connection string and DSN definition from the original connection request are used, with the appropriate server name.
- It is up to the application to open new statement handles and reexecute necessary `SQLPrepare` calls.
- If a failover has already occurred and the client is already connected to the new database or database element:
  - For TimesTen Scaleout, the next failover request results in an attempt to connect to the next element in the list that was returned by TimesTen at the time of the original connection.
  - For TimesTen Classic with active standby replication, the next failover request results in an attempt to reconnect to the original active database. If that fails, alternating attempts are made to connect to the two servers until there is a timeout, and the connection is blocked during this period.
  - For TimesTen Classic using generic automatic client failover, the next failover request results in an attempt to connect to the next database in the list that is configured in the client `odbc.ini` file. This could be the next database sequentially or one chosen at random from the list, according to the setting of the `TTC_Random_Selection` connection attribute, which is described in Configuration of Automatic Client Failover.

The timeout value is according to the TimesTen client connection attribute `TTC_Timeout` (default 60 seconds). (Refer to TTC_Timeout in *Oracle TimesTen In-Memory Database Reference* for information about that attribute.)

- Failover connections are created only as needed, not in advance.

During failover, TimesTen can optionally make callbacks to a user-defined function that you register. This function takes care of any custom actions you want to occur in a failover situation. (See Implementing and Registering an ODBC Failover Callback Function.)

The following public connection options are propagated to the new connection. The corresponding general connection attribute is shown in parentheses where applicable. The `TT_REGISTER_FAILOVER_CALLBACK` option is used to register your callback function.

```
SQL_ACCESS_MODE
SQL_AUTOCOMMIT
SQL_TXN_ISOLATION (Isolation)
SQL_OPT_TRACE
SQL_QUIET_MODE
TT_PREFETCH_CLOSE
TT_CLIENT_TIMEOUT (TTC_TIMEOUT)
TT_REGISTER_FAILOVER_CALLBACK
```

The following options are propagated to the new connection if they were set through connection attributes or `SQLSetConnectOption` calls, but not if set through TimesTen built-in procedures or `ALTER SESSION`.

```
TT_NLS_SORT (NLS_SORT)
TT_NLS_LENGTH_SEMANTICS (NLS_LENGTH_SEMANTICS)
TT_NLS_NCHAR_CONV_EXCP (NLS_NCHAR_CONV_EXCP)
TT_DYNAMIC_LOAD_ENABLE (DynamicLoadEnable)
TT_DYNAMIC_LOAD_ERROR_MODE (DynamicLoadErrorMode)
TT_NO_RECONNECT_ON_FAILOVER (TTC_NoReconnectOnFailover)
```

The following options are propagated to the new connection if they were set on the connection handle.

```
SQL_QUERY_TIMEOUT
TT_PREFETCH_COUNT
```

See Connection Attributes in *Oracle TimesTen In-Memory Database Reference*.

> **Note:**
>
> If you issue an `ALTER SESSION` statement anytime after the initial database connection, you must re-issue the statement after a failover.

Configuration of Automatic Client Failover

In TimesTen Classic, failover DSNs must be specifically configured through `TTC_Server2` and `TTC_Server`*n* connection attributes.

Setting any of `TTC_Server2`, `TTC_Server_DSN2`, `TTC_Server`*n*, or `TCP_Port2` implies that you intend to use automatic client failover. For the active standby pair scenario, it also means a new thread is created for your application to support the failover mechanism.

Refer to Configuring Automatic Client Failover for TimesTen Classic in *Oracle TimesTen In-Memory Database Operations Guide* or Client Connection Failover in the *Oracle TimesTen In-Memory Database Scaleout User's Guide*.

Be aware of these TimesTen connection attributes:

- `TTC_NoReconnectOnFailover`: If this is set to 1 (enabled), TimesTen is instructed to do all the usual client failover processing except for the automatic reconnect. (For example, statement and connection handles are marked as invalid.) This is useful if the application does its own connection pooling or manages its own reconnection to the database after failover. The default value is 0 (reconnect). Also see TTC_NoReconnectOnFailover in *Oracle TimesTen In-Memory Database Reference*.
- `TTC_REDIRECT`: If this is set to 0 and the initial connection attempt to the desired database or database element fails, then an error is returned and there are no further connection attempts. This does not affect subsequent failovers on that connection. Also see "TTC_REDIRECT" in *Oracle TimesTen In-Memory Database Reference*.
- `TTC_Random_Selection`: For TimesTen Classic using generic automatic client failover, the default setting of 1 (enabled) specifies that when failover occurs, the client randomly selects an alternative server from the list provided in `TTC_Server`*`n`* attribute settings. If the client cannot connect to the selected server, it keeps redirecting until it successfully connects to one of the listed servers. With a setting of 0, TimesTen goes through the list of `TTC_Server`*`n`* servers sequentially. Also see TTC_Random_Selection in *Oracle TimesTen In-Memory Database Reference*.

> **Note:**
>
> If you set any of these in `odbc.ini` or the connection string, the settings are applied to the failover connection. They cannot be set as ODBC connection options or ALTER SESSION attributes.

Implementing and Registering an ODBC Failover Callback Function

If there are custom actions you would like to have occur when there is a failover, you can have TimesTen make a callback to a user-defined function for such actions.

This function is called when the attempt to connect to the new database or database element begins, and again after the attempt to connect is complete. This function could be used, for example, to cleanly restore statement handles.

The function API is defined as follows.

```
typedef SQLRETURN (*ttFailoverCallbackFcn_t)
  (SQLHDBC,       /* hdbc    */
   SQLPOINTER,    /* foCtx   */
   SQLUINTEGER,   /* foType  */
   SQLUINTEGER);  /* foEvent */
```

Where:

- *`hdbc`* is the ODBC connection handle for the connection that failed.

- *foCtx* is a pointer to an application-defined data structure, for use as needed.
- *foType* is the type of failover. In TimesTen, the only supported value for this is TT_FO_SESSION, which results in the session being reestablished. This does not result in statements being re-prepared.
- *foEvent* indicates the event that has occurred, with the following supported values:
  - TT_FO_BEGIN: Beginning failover.
  - TT_FO_ABORT: Failover failed. Retries were attempted for the interval specified by TTC_Timeout (minimum value 60 seconds for active standby failover) without success.
  - TT_FO_END: Successful end of failover.
  - TT_FO_ERROR: A failover connection failed but will be retried.

  Note that TT_FO_REAUTH is *not* supported by TimesTen client failover.

Use a SQLSetConnectOption call to set the TimesTen TT_REGISTER_FAILOVER_CALLBACK option to register the callback function, specifying an option value that is a pointer to a structure of C type ttFailoverCallback_t that is defined as follows in the timesten.h file and refers to the callback function.

```
typedef struct{
  SQLHDBC                  appHdbc;
  ttFailoverCallbackFcn_t callbackFcn;
  SQLPOINTER               foCtx;
} ttFailoverCallback_t;
```

Where:

- *appHdbc* is the ODBC connection handle, and should have the same value as *hdbc* in the SQLSetConnectOption calling sequence. (It is required in the data structure due to driver manager implementation details, in case you are using a driver manager.)
- *callbackFcn* specifies the callback function. (You can set this to NULL to cancel callbacks for the given connection. The failover would still happen, but the application would not be notified.)
- *foCtx* is a pointer to an application-defined data structure, as in the function description earlier.

Set TT_REGISTER_FAILOVER_CALLBACK for each connection for which a callback is desired. The values in the ttFailoverCallback_t structure are copied when the SQLSetConnectOption call is made. The structure need not be kept by the application. If TT_REGISTER_FAILOVER_CALLBACK is set multiple times for a connection, the last setting takes precedence.

> **Note:**
>
> - Because the callback function executes asynchronously to the main thread of your application, it should generally perform only simple tasks, such as setting flags that are polled by the application. However, there is no such restriction if the application is designed for multithreading. In that case, the function could even make ODBC calls, for example, but it is only safe to do so if the *`foEvent`* value `TT_FO_END` has been received.
> - It is up to the application to manage the data pointed to by the *`foCtx`* setting.

This example shows the following features.

- A globally defined user structure type, `FOINFO`, and the structure variable `foStatus` of type `FOINFO`
- A callback function, `FailoverCallback()`, that updates the `foStatus` structure whenever there is a failover
- A registration function, `RegisterCallback()`, that does the following:
  - Declares a structure, `failoverCallback`, of type `ttFailoverCallback_t`.
  - Initializes `foStatus` values.
  - Sets the `failoverCallback` data values, consisting of the connection handle, a pointer to `foStatus`, and the callback function (`FailoverCallback`).
  - Registers the callback function with a `SQLSetConnectOption` call that sets `TT_REGISTER_FAILOVER_CALLBACK` as a pointer to `failoverCallback`.

```
/* user defined structure */
struct FOINFO
{
   int callCount;
   SQLUINTEGER lastFoEvent;
};
/* global variable passed into the callback function */
struct FOINFO foStatus;

/* the callback function */
SQLRETURN FailoverCallback (SQLHDBC hdbc,
                            SQLPOINTER pCtx,
                            SQLUINTEGER FOType,
                            SQLUINTEGER FOEvent)
{
   struct FOINFO* pFoInfo = (struct FOINFO*) pCtx;

   /* update the user defined data */
   if (pFoInfo != NULL)
   {
      pFoInfo->callCount ++;
      pFoInfo->lastFoEvent = FOEvent;

      printf ("Failover Call #%d\n", pFoInfo->callCount);
   }
```

```
   /* the ODBC connection handle */
   printf ("Failover HDBC : %p\n", hdbc);

   /* pointer to user data */
   printf ("Failover Data : %p\n", pCtx);

   /* the type */
   switch (FOType)
   {
       case TT_FO_SESSION:
       printf ("Failover Type : TT_FO_SESSION\n");
       break;

     default:
       printf ("Failover Type : (unknown)\n");
   }

   /* the event */
   switch (FOEvent)
   {
     case TT_FO_BEGIN:
       printf ("Failover Event: TT_FO_BEGIN\n");
       break;

     case TT_FO_END:
       printf ("Failover Event: TT_FO_END\n");
       break;

     case TT_FO_ABORT:
       printf ("Failover Event: TT_FO_ABORT\n");
       break;

     case TT_FO_REAUTH:
       printf ("Failover Event: TT_FO_REAUTH\n");
       break;

     case TT_FO_ERROR:
       printf ("Failover Event: TT_FO_ERROR\n");
       break;

     default:
       printf ("Failover Event: (unknown)\n");
   }

 return SQL_SUCCESS;
}

/* function to register the callback with the failover connection */
SQLRETURN RegisterCallback (SQLHDBC hdbc)
{
   SQLRETURN rc;
   ttFailoverCallback_t failoverCallback;

   /* initialize the global user defined structure */
   foStatus.callCount = 0;
   foStatus.lastFoEvent = -1;

   /* register the connection handle, callback and the user defined structure */
   failoverCallback.appHdbc = hdbc;
   failoverCallback.foCtx = &foStatus;
   failoverCallback.callbackFcn = FailoverCallback;
```

```
   rc = SQLSetConnectOption (hdbc, TT_REGISTER_FAILOVER_CALLBACK,
     (SQLULEN)&failoverCallback);

   return rc;
}
```

When a failover occurs, the callback function would produce output such as the following:

```
Failover Call #1
Failover HDBC : 0x8198f50
Failover Data : 0x818f8ac
Failover Type : TT_FO_SESSION
Failover Event: TT_FO_BEGIN
```

ODBC Application Action in the Event of Failover

There are actions to perform in the event of a failover.

This section discusses these topics:

- Application Steps for Failover
- Implementing Failover Delay and Retry Settings

Application Steps for Failover

If you receive any error conditions in response to an operation in your application, then application failover is in progress.

See Features and Functionality of ODBC Support for Automatic Client Failover for a list of error conditions.

Perform these recovery actions:

1. Issue a rollback on the connection. Until you do this, no further processing is possible on the connection.
2. Clean up all objects from the previous connection. None of the state or objects associated with the previous connection are preserved, but proper cleanup through the relevant API calls is still strongly recommended.
3. Assuming `TTC_NoReconnectOnFailover=0` (the default), sleep briefly, as discussed in the next section, Implementing Failover Delay and Retry Settings. If `TTC_NoReconnectOnFailover=1`, then you must instead manually reconnect the application to an alternate database or database element.
4. Recreate and reprepare all objects related to your connection.
5. Restart any in-progress transactions from the beginning.

Implementing Failover Delay and Retry Settings

The reconnection to another database or database element during automatic client failover may take some time. Therefore, your application should place all recovery actions within a loop with a short delay before each subsequent attempt, where the total number of attempts is limited.

Note the following:

- If your application attempts recovery actions before TimesTen has completed its client failover process, you may receive another failover error condition as listed in Features and Functionality of ODBC Support for Automatic Client Failover.
- If you do not limit the number of attempts, the application may appear to freeze if the client failover process does not complete successfully. For example, your recovery loop could use a retry delay of 100 milliseconds with a maximum number of retries limited to 100 attempts. The ideal values depend on your particular application and configuration.

This example illustrates some of these points (as well as retrying transient errors, discussed in Transient Errors (ODBC)).

```
/*
 * The following code snippet is a simple illustration of how you might handle
 * the retrying of transient and connection failover errors in a C/ODBC
 * application. In the interests of simplicity code that is not directly
 * relevant to the example has been omitted (...). A real application
 * would of course be more complex.
 *
 * This example uses the ODBC 3.5 API.
 */

// define maximum retry counts and failover retry delay
#define  MAX_TE_RETRIES    30
#define  MAX_FO_RETRIES    100
#define  FO_RETRY_DELAY    100  // milliseconds

// function return values
#define  SUCCESS           0
#define  FAILURE         (-1)

// constants for categorising errors
#define  ERR_OTHER         1
#define  ERR_TRANSIENT     2
#define  ERR_FAILOVER      3

// SQLSTATES and native errors
#define    SQLSTATE_TRANSIENT   "TT005"
#define    SQLSTATE_FAILOVER    "08006"
#define    NATIVE_FAILOVER1     47137
#define    NATIVE_FAILOVER2     30105

// SQL statements
SQLCHAR * sqlQuery = (SQLCHAR *)"SELECT ...";
SQLCHAR * sqlUpdate = (SQLCHAR *)"UPDATE ...";

// Database connection handle
SQLHDBC        dbConn = SQL_NULL_HDBC;

// Statement handles
SQLHSTMT      stmtQuery = SQL_NULL_HSTMT;
SQLHSTMT      stmtUpdate = SQL_NULL_HSTMT;

// ODBC return code
SQLRETURN rc;

// Retry counters
int teRetries; // transient errors
int foRetries; // failover errors
int foDelay = FO_RETRY_DELAY; // failover retry delay in ms
```

```
// Function to sleep for a specified number of milliseconds
void
sleepMs( unsigned int ms)
{
    struct timespec rqtm, rmtm;

    rqtm.tv_sec = (time_t)(ms / 1000);
    rqtm.tv_nsec = (long)(ms % 1000000);

    while (  nanosleep( &rqtm, &rmtm )  )
        rqtm = rmtm;
} // sleepMs

// Function to check error stack for transient or failover errors.
// In a real application lots of other kinds of checking would also
// go in here to identify other errors of interest. We'd probably also
// log the errors to an error log.
int
errorCategory( SQLHANDLE handle, SQLSMALLINT handleType )
{
    SQLRETURN rc;
    SQLSMALLINT i = 1;
    SQLINTEGER native_error;
    SQLCHAR sqlstate[LEN_SQLSTATE+1];
    SQLCHAR msgbuff[1024];
    SQLSMALLINT msglen;

    native_error = 0;
    sqlstate[0] = '\0';
    rc = SQLGetDiagRec( handleType, handle, i, sqlstate, &native_error,
                        msgbuff, sizeof(msgbuff), &msglen );
    while (   rc == SQL_SUCCESS  )
    {
        if (  strcmp( sqlstate, SQLSTATE_TRANSIENT ) == 0  )
            return ERR_TRANSIENT;
        else
        if (  native_error == NATIVE_FAILOVER1  )
            return ERR_FAILOVER;
        else
        if (  ( strcmp( sqlstate, SQLSTATE_FAILOVER ) == 0 ) &&
              (native_error == NATIVE_FAILOVER2)  )
            return ERR_FAILOVER;
        rc = SQLGetDiagRec( handleType, handle, ++i, sqlstate,
                            &native_error, msgbuff, sizeof(msgbuff),
                            &msglen );
    }

    return ERR_OTHER;
} // errorCategory

// Function to perform a rollback
void
rollBack( SQLHDBC hDbc )
{
    SQLRETURN rc;

    rc = SQLEndTran( SQL_HANDLE_DBC, hDbc, SQL_ROLLBACK );
    // Report/log errors (a rollback failure is very, very bad).
    ...
} // rollBack
```

```
// Function to prepare all statements, bind parameters and bind
// columns.
int
prepareAll( void )
{
    SQLRETURN rc;

    // Prepare the SQL statements and check for errors.
    rc = SQLPrepare( stmtQuery, sqlQuery, SQL_NTS );
    if (  rc != SQL_SUCCESS  )
    {
        rollBack( dbConn );
        return errorCategory( stmtQuery, SQL_HANDLE_STMT );
    }
    rc = SQLPrepare( stmtUpdate, sqlUpdate, SQL_NTS );
...
    // Bind parameters and colums
...

    return SUCCESS; // indicate success
} // prepareAll

// Function to execute a specific application transaction handling
// retries.
int
txnSomeTransaction( ... )
{
    SQLRETURN rc;
    SQLLEN    rowcount = 0;
    int needReprepare = 0;
    int result;

    // Initialize retry counters
    teRetries = MAX_TE_RETRIES;
    foRetries = MAX_FO_RETRIES;

    // main retry loop
    while (  ( teRetries > 0 ) && ( foRetries > 0 )  )
    {

        // Do we need to re-prepare?
        while ( needReprepare && ( foRetries > 0 ) )
        {
            msSleep( retryDelay ); // delay before proceeding
            result = prepareAll();
            if (  result == SUCCESS  )
                needReprepare = 0;
            else
            if (  result != ERR_FAILOVER  )
                goto err;
            else
                foRetries--;
        }

        // First execute the query

        // Set input values for query
        ...

        // Execute query
        rc = SQLExecute( stmtQuery );
```

```
if (  rc != SQL_SUCCESS  )
{
    result = errorCategory( stmtQuery, SQL_HANDLE_STMT );
    rollBack( dbConn );
    switch (  result  )
    {
        case ERR_OTHER:
            goto err;
            break;
        case ERR_TRANSIENT:
            teRetries--;
            continue; // retry loop
            break;
        case ERR_FAILOVER:
            foRetries--;
            needReprepare = 1;
            continue; // retry loop
            break;
    }
}

// Process results
while (  (rc = SQLFetch( stmtQuery )) == SQL_SUCCESS  )
{
    // process next row
    ...
}
if (  (rc != SQL_SUCCESS) && (rc != SQL_NO_DATA)  )
{
    result = errorCategory( stmtQuery, SQL_HANDLE_STMT );
    rollBack( dbConn );
    switch (  result  )
    {
        case ERR_OTHER:
            goto err;
            break;
        case ERR_TRANSIENT:
            teRetries--;
            continue; // retry loop
            break;
        case ERR_FAILOVER:
            foRetries--;
            needReprepare = 1;
            continue; // retry loop
            break;
    }
}

// Now execute the update

// Set input values for update
...

// Execute update
rc = SQLExecute( stmtUpdate );
if (  rc != SQL_SUCCESS  )
{
    ...
}

// Check number of rows affected
```

```
            rc = SQLRowCount( stmtUpdate, &rowcount );
            if (  rc != SQL_SUCCESS  )
            {
                ...
            }
            // Check rowcount and handle unexpected cases
            if (  rowcount != 1  )
            {
                ...
            }

            // Finally, commit
            rc = SQLEndTran( SQL_HANDLE_DBC, dbConn, SQL_COMMIT );
            if (  rc != SQL_SUCCESS  )
            {
                ...
            }

            return SUCCESS; // all good
        } // retry loop

    err:
        // if we get here, we ran out of retries or had some other non-retryable
        // error. Report/log it etc. then return failure
        ...

        return FAILURE;
    } // txnSomeTransaction

    // Main code
    int
    main ( int argc, char * argv[] )
    {
        int status = 0; // final exit code

        ....

        // Open the connection  to the database and allocate statement handles
        ...

        // Disable auto-commit (this is essential)
        rc = SQLSetConnectAttr( dbConn,
                                SQL_ATTR_AUTOCOMMIT,
                                SQL_AUTOCOMMIT_OFF,
                                0 );
        ...

        // Prepare all statements, bind etc.
        if (  prepareAll() != SUCCESS  )
        {
            ...
        }

        // Do stuff until we are finished
        while (  ...  )
        {
            ...
            if (  txnSomeTransaction( ... ) != SUCCESS  )
            {
                ...
                goto fini;
```

```
            }
            ...
        }

fini:  // cleanup etc.
    // Release all resources (ODBC and non-ODBC)
    ...
    // Disconnect from database
    ...

    // Return final exit code
    return status;
} //main
```

Client Routing API for TimesTen Scaleout

These sections describe the client routing API for TimesTen Scaleout.

- Functionality of the Client Routing API
- Creating a Grid Map and Distribution
- Distribution Key Values
- Getting the Element Location Given a Set of Key Values
- Client Routing API With Functions in Use
- Supported Data Types
- Restrictions
- Failure Modes

Functionality of the Client Routing API

To increase performance, TimesTen Scaleout enables your client application to route connections to an element based on the key value for a hash distribution key.

You provide a key value and TimesTen Scaleout returns an array of element IDs (or the replica set ID) where the database allocated that value. This enables the client application to connect to the element that stores the row with the specified key value, avoiding unnecessary communication between the element storing the row and the one connected to your application.

> **Note:**
>
> This feature is not supported with generic driver managers but is supported with the TimesTen driver manager.

Creating a Grid Map and Distribution

These sections show how to create a grid map and distribution for client routing.

- Functions for the Grid Map and Distribution
- How to Create the Grid Map and Distribution

Functions for the Grid Map and Distribution

There are functions you can use for a grid map and distribution for client routing.

TimesTen Scaleout includes two new objects for a grid map and a grid distribution in the `timesten.h` file:

- `TTGRIDMAP`: A grid map is a lookup table that maps the topology of a grid. You create a grid map by calling the `ttGridMapCreate` function with a valid ODBC connection. The function returns a handle to a `TTGRIDMAP` object.

  Use the `ttGridMapFree` function to free a grid map.

  > **Note:**
  >
  > – A `TTGRIDMAP` object is not strongly associated with the HDBC connection. Freeing either object does not free the other.
  >
  > – A grid map can be shared among many grid distributions and across application threads. Only one grid map is required per application process per database.

- `TTGRIDDIST`: A grid distribution is an ordered set of types and values that represent the distribution key columns of a table or tables. For distribution keys composed of multiple columns, the order of the types and values must be the same as for the distribution key columns of the table.

  You create a grid distribution by calling the `ttGridDistCreate` function with the C type, SQL type, length, scale, and precision of the distribution key columns of a table. The function returns a handle to a `TTGRIDDIST` object. See Table 2-3.

  Use the `ttGridDistFree` function to free a grid distribution.

  > **Note:**
  >
  > – A `TTGRIDDIST` object is not associated with a given table. You can use the same `TTGRIDDIST` object for any table that uses the same types and values in their distribution key columns.
  >
  > – A grid distribution cannot be shared across threads. However, multiple grid distributions in different threads can be created using the same grid map.

Table 2-3 ttGridDistCreate Arguments

Argument	Type	Description
`hdbc`	`SQLHDBC`	Connection handle
`map`	`TTGRIDMAP`	Grid map handle
`cTypes[]`	`SQLSMALLINT`	Array of C bind types in the same order as the distribution key columns

Table 2-3 (Cont.) ttGridDistCreate Arguments

Argument	Type	Description
sqlTypes[]	SQLSMALLINT	Array of SQL bind types in the same order as the distribution key columns
precisions[]	SQLULEN	Array of precision values in the same order as the distribution key columns
scales[]	SQLSMALLINT	Array of scale values in the same order as the distribution key columns
maxSizes[]	SQLLEN	Array of maximum column size values in the same order as the distribution key columns
nCols	SQLUSMALLINT	Number of columns in the distribution key
*dist	TTGRIDDIST	Grid distribution handle (OUT)

Note:

The parameters for `ttGridDistCreate` are similar to those used in a subsequent `SQLBindParameter` ODBC call.

How to Create the Grid Map and Distribution

This example shows how to work with the grid map and distribution.

1. Create `TTGRIDMAP` and `TTGRIDDIST` objects.
2. Call the `ttGridMapCreate` function to create a grid map using an existing ODBC connection.
3. Call the `ttGridDistCreate` function to create a grid distribution based on a distribution key composed of two columns.
4. Free the grid distribution and map with the `ttGridDistFree` and `ttGridMapFree` functions, respectively.

```
TTGRIDMAP map;
TTGRIDDIST dist;

ttGridMapCreate(hdbc, &map);

SQLSMALLINT cTypes[] = { SQL_C_LONG, SQL_C_CHAR };
SQLSMALLINT sqlTypes[] = { SQL_INTEGER, SQL_CHAR };
SQLLEN maxSizes[] = { 4, 20 };

ttGridDistCreate(hdbc, map, cTypes, sqlTypes, NULL, NULL, maxSizes, 2, &dist);

...

ttGridDistFree(hdbc, dist);
ttGridMapFree(hdbc, map);
```

Distribution Key Values

There are methods to set the grid distribution key values for client routing.

- Function for Distribution Key Values
- Setting Distribution Key Values

Function for Distribution Key Values

Use the `ttGridDistValueSet` function to set the grid distribution key values for client routing.

With the grid map and distribution defined, you set the key values in order to determine the elements in which they are allocated. Call the `ttGridDistValueSet` function to set the key value for one of the columns in the distribution key. For distribution keys composed of multiple columns, call this function once for every column in the distribution key. Table 2-4 provides a brief summary of the arguments of the `ttGridDistValueSet` function.

Table 2-4 ttGridDistValueSet Arguments

Argument	Type	Description
`hdbc`	`SQLHDBC`	Connection handle
`dist`	`TTGRIDDIST`	Grid distribution handle
`position`	`SQLSMALLINT`	Position of the column in the distribution key
`value`	`SQLPOINTER`	Key value pointer
`valueLen`	`SQLLEN`	Length of the key value

Setting Distribution Key Values

This example first calls the `ttGridDistClear` function to clear any previously defined key values for the distribution key columns. Then, the example calls the `ttGridDistValueSet` function for every column in the distribution key and sets the key value for each column.

```
ttGridDistClear(hdbc, dist);

ttGridDistValueSet(hdbc, dist, 1, empId, sizeof(empId));
ttGridDistValueSet(hdbc, dist, 2, "SALES", SQL_NTS);
```

Getting the Element Location Given a Set of Key Values

Once you set the distribution key values, this section shows that you can call for the location of the key values either by element IDs or replica set ID.

These topics are covered:

- Function for Element IDs
- Getting the Element IDs
- Function for Replica Set ID
- Getting the Replica Set ID

Function for Element IDs

Call the `ttGridDistElementGet` function to obtain the corresponding element IDs that represent the location of the provided key values. The function returns an array of element

IDs. The application is responsible for allocating the return array. The length of the array is based on the value of K-safety of the grid.

For example, in a grid with K-safety set to 2, there must be at least two elements in the array. Table 2-5 provides a brief summary of the arguments of the ttGridDistElementGet function.

Table 2-5 ttGridDistElementGet Arguments

Argument	Type	Description
hdbc	SQLHDBC	Connection handle
dist	TTGRIDDIST	Grid distribution handle
elemIds[]	SQLSMALLINT	Array of element IDs where the key values are allocated (IN/OUT)
elemIdSize	SQLSMALLINT	Value of K-safety

Getting the Element IDs

These examples show how to get the element IDs and how to associate an element ID with a connection string.

This example gets the array of element IDs associated with the current key values (set by the ttGridDistValueSet function) by calling the ttGridDistElementGet function.

```
SQLSMALLINT elementIds[2];

ttGridDistElementGet(hdbc, dist, elementIds, 2);
```

> **Note:**
>
> The elementIds array must be of a length equal or greater than the value of K-safety of the grid.

With the location of the set of key values available, your application can use the element IDs to select a connection to one of the elements, prepare a statement, bind values, and execute the statement.

> **Note:**
>
> The connection attempt can be subject to a failover event and the application may not connect to the expected element.

The example that follows shows a query that may help you associate an element ID with a connection string. It assembles a connection string for each element of the database by querying the SYS.V$DISTRIBUTION_CURRENT system view. The connection string includes the TTC_REDIRECT=0 attribute to ensure a connection to the specified element or its replica. If the connection to all replicas fails, then a connection error is returned.

```
select 'TTC_REDIRECT=0;
TTC_SERVER='||hostexternaladdress||'/'||serverport,mappedelementid
 from SYS.V$DISTRIBUTION_CURRENT;
< TTC_REDIRECT=0;TTC_SERVER=ext-host3.example.com/6625, 1 >
< TTC_REDIRECT=0;TTC_SERVER=ext-host4.example.com/6625, 2 >
< TTC_REDIRECT=0;TTC_SERVER=ext-host5.example.com/6625, 3 >
< TTC_REDIRECT=0;TTC_SERVER=ext-host6.example.com/6625, 4 >
< TTC_REDIRECT=0;TTC_SERVER=ext-host7.example.com/6625, 5 >
< TTC_REDIRECT=0;TTC_SERVER=ext-host8.example.com/6625, 6 >
6 rows found.
```

Function for Replica Set ID

Call the `ttGridDistReplicaGet` function to obtain the corresponding replica set ID that represents the location of the provided key values.

Table 2-6 provides a brief summary of the arguments of the `ttGridDistReplicaGet` function.

Table 2-6 ttGridDistReplicaGet Arguments

Argument	Type	Description
`hdbc`	`SQLHDBC`	Connection handle
`dist`	`TTGRIDDIST`	Grid distribution handle
`*replicaSetId`	`SQLSMALLINT`	Replica set ID where the key values are allocated (`OUT`)

Getting the Replica Set ID

This example gets the replica set ID associated with the current key values (set by the `ttGridDistValueSet` function) by calling the `ttGridDistReplicaGet` function.

```
SQLSMALLINT replicaSetId;

ttGridDistReplicaGet(hdbc, dist, replicaSetId);
```

You can use the replica set ID with the `SYS.V$DISTRIBUTION_CURRENT` system view to look up the communication parameters of the elements in that replica set.

Client Routing API With Functions in Use

This partial example shows the client routing API with most of its objects and functions in use.

```
#include <timesten.h>

...

TTGRIDMAP map;
TTGRIDDIST dist;

/* Create a grid map using any existing connection. */
ttGridMapCreate(hdbc, &map);

/* The distribution key has two columns: one with TT_INTEGER as data type and
 * one with CHAR(20), in that order. Precision and scale are not necessary. */
SQLSMALLINT cTypes[] = { SQL_C_LONG, SQL_C_CHAR };
SQLSMALLINT sqlTypes[] = { SQL_INTEGER, SQL_CHAR };
```

```
SQLLEN maxSizes[] = { 4, 20 };

/* Create grid distribution from the grip map and the specified distribution
 * key column paremeters. */
ttGridDistCreate(hdbc, map, cTypes, sqlTypes, NULL, NULL, maxSizes, 2, &dist);

/* Execution loop. */
while ( ... )
{
      SQLSMALLINT elementIds[2];

      /* Clear the existing key values from the distribution map */
      ttGridDistClear(hdbc, dist);

      /* Set the key values for the grid distribution. */
      ttGridDistValueSet(hdbc, dist, 1, key1, sizeof(key1));
      ttGridDistValueSet(hdbc, dist, 2, key2, SQL_NTS);

      /* Get the corresponding element IDs for current key values*/
      ttGridDistElementGet(hdbc, dist, elementIds, 2);

      /* The application uses the element IDs to select a connection to
       * one of the elements, prepare a statement, bind values, and execute
       * the statement. */
      ...
}

/* Free the grid distribuion and map. */
ttGridDistFree(hdbc, dist);
ttGridMapFree(hdbc, map);
```

Supported Data Types

The `TTGRIDDIST` object is created using the C types and SQL types available from ODBC.

Table 2-7 shows the supported C types and SQL types with their corresponding Database SQL types.

Table 2-7 List of Supported Types

C Types	ODBC SQL Types	Database SQL Types
SQL_C_TINYINT	SQL_TINYINT	TT_TINYINT
SQL_C_SMALLINT	SQL_SMALLINT	TT_SMALLINT
SQL_C_LONG	SQL_INTEGER	TT_INTEGER
SQL_C_BIGINT	SQL_BIGINT	TT_BIGINT
SQL_C_CHAR	SQL_CHAR	CHAR
SQL_C_CHAR	SQL_VARCHAR	VARCHAR, VARCHAR2
SQL_C_WCHAR	SQL_WCHAR	NCHAR
SQL_C_WCHAR	SQL_WVARCHAR	NVARCHAR
SQL_C_SQLT_NUM	SQL_DOUBLE	NUMBER
SQL_C_SQLT_NUM	SQL_DECIMAL	NUMBER(p,s)

Table 2-7 (Cont.) List of Supported Types

C Types	ODBC SQL Types	Database SQL Types
SQL_C_SQLT_VNU	SQL_DOUBLE	NUMBER
SQL_C_SQLT_VNU	SQL_DECIMAL	NUMBER(p,s)

The TTGRIDDIST object supports all signed and unsigned data type variants. For example, it supports both SQL_C_SLONG and SQL_C_ULONG.

You can set NULL values by specifying SQL_NULL_DATA for the valueLen parameter of the ttGridDistValueSet function. The NULL value will always map to the same replica set or element IDs.

Restrictions

Client routing has certain restrictions.

- It does not have implicit connection or statement management.
- It does not support date, time, or timestamp data types.
- It does not support explicit type conversion. Applications must specify key values in canonical byte format.
- It does not support character set conversion. It ignores the connection character set.
- Changes in the topology of the grid require that applications free and recreate the grid map.

Failure Modes

The client routing API may return certain errors.

- *Incorrect types and values to describe the distribution key columns of the table*. In this case, the API will still compute an array of element IDs, but these may not correspond to the real location of the desired key values.
- *Unrecognized type codes*. If you call the ttGridDistCreate function with unrecognized type codes, the function returns an error.
- *Not enough values set for the grid distribution*. If you do not provide enough values for the distribution key through the ttGridDistValueSet function, then the ttGridDistElementGet or ttGridDistReplicaGet function would return an error.
- *Invalid size of the element IDs array*. If you do not provide an array of at least the size of the value of K-safety, the ttGridDistElementGet function would return an error.

3

TimesTen Support for OCI

TimesTen supports the Oracle Call Interface (OCI) for C or C++ programs.

This chapter provides an overview and TimesTen-specific information regarding OCI, especially emphasizing differences between using OCI with TimesTen versus with Oracle Database. For complete information about OCI, you can refer to *Oracle Call Interface Programmer's Guide* in the Oracle Database library.

Also note that Working With TimesTen Databases in ODBC contains information that may be of general interest regarding TimesTen features.

The following topics are covered:

- Overview of TimesTen OCI Support
- Getting Started With TimesTen OCI
- TimesTen Features With OCI
- TimesTen OCI Support Reference

Overview of TimesTen OCI Support

You can use OCI with TimesTen. For supported features, TimesTen OCI syntax and usage are the same as in Oracle Database.

These topics are covered:

- Overview of OCI
- TimesTen OCI Basics
- OCI in the TimesTen Architecture
- Globalization Support in TimesTen OCI
- TimesTen Restrictions and Limitations

Overview of OCI

OCI is an API that provides functions you can use to access the database and control SQL execution. OCI supports the data types, calling conventions, syntax, and semantics of the C and C++ programming languages.

You compile and link an OCI program much as you would any C or C++ program. There is no preprocessing or precompilation step.

The OCI library of database access and retrieval functions is in the form of a dynamic runtime library that can be linked into an application at runtime. The OCI library includes the following functional areas:

- SQL access functions
- Data type mapping and manipulation functions

The following are among the many useful features that OCI provides or supports:

- Statement caching
- Dynamic SQL
- Facilities to treat transaction control, session control, and system control statements like DML statements
- Description functionality to expose layers of server metadata
- Ability to associate commit requests with statement executions to reduce round trips
- Optimization of queries using transparent prefetch buffers to reduce round trips
- Thread safety that eliminates the need for mutual exclusive locks on OCI handles

For general information about OCI, you can refer to *Oracle Call Interface Programmer's Guide*, included with the Oracle Database documentation set.

TimesTen OCI Basics

TimesTen OCI support enables you to run many existing OCI applications with TimesTen direct connections or client/server connections. It also enables you to use other features, such as Pro*C/C++ and ODP.NET, that use OCI as a database interface. (You can also call PL/SQL from OCI, Pro*C/C++, and ODP.NET applications.)

TimesTen provides Oracle Instant Client as the OCI client library. This is configured through the appropriate `ttenv` script, discussed in Environment Variables in *Oracle TimesTen In-Memory Database Installation, Migration, and Upgrade Guide*.

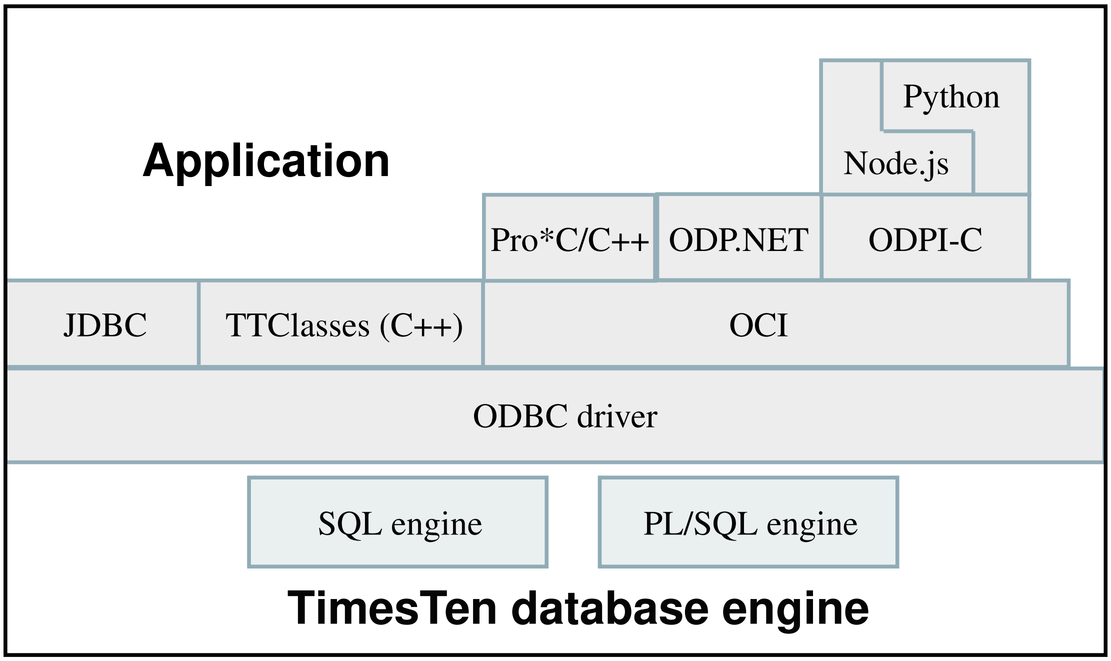
TimesTen Release 22.1 OCI is based on Oracle Database release 19c OCI and supports the contemporary OCI 8 style APIs.

OCI in the TimesTen Architecture

OCI support is positioned in the TimesTen architecture.

Figure 3-1 shows where OCI support is positioned in the TimesTen architecture.

Figure 3-1 OCI in the TimesTen Architecture



Globalization Support in TimesTen OCI

TimesTen OCI supports globalization.

- About TimesTen Support for Character Sets
- Specifying a Character Set
- Additional Globalization Features

About TimesTen Support for Character Sets

TimesTen character sets are compatible with Oracle Database.

The TimesTen default character set is `AMERICAN_AMERICA.US7ASCII` (but `AL32UTF8` is recommended as an alternative). Refer to Supported Character Sets in *Oracle TimesTen In-Memory Database Reference*.

Specifying a Character Set

To specify a character set for the connection, OCI programs can set the `NLS_LANG` environment variable or call `OCIEnvNlsCreate()`.

The `ConnectionCharacterSet` setting in the `sys.odbc.ini` or user `odbc.ini` file is used by default if not overridden by `NLS_LANG` or `OCIEnvNlsCreate()`. Setting the character set explicitly is recommended. The default is typically `AMERICAN_AMERICA.US7ASCII`, but `AL32UTF8` is recommended as an alternative.

Note that because TimesTen OCI does not support language or locale (territory) settings, the language and territory components of `NLS_LANG`, such as `AMERICAN_AMERICA` above, are ignored. Even when not specifying the language and locale, however, you must still have the period in front of the character set when setting NLS_LANG. For example, either of the following would work, although AMERICAN_AMERICA is ignored:

```
NLS_LANG=AMERICAN_AMERICA.WE8ISO8859P1
```

Or:

```
NLS_LANG=.WE8ISO8859P1
```

Note:

- An NLS_LANG environment setting overrides the TimesTen default character set.
- On Windows, the NLS_LANG setting is searched for in the registry if it is not in the environment. If your OCI or Pro*C/C++ program has trouble connecting to TimesTen, verify that the NLS_LANG setting under HKEY_LOCAL_MACHINE\Software\ORACLE\, if that key exists, is valid and indicates a character set supported by TimesTen.
- Refer to Choosing a Locale With the NLS_LANG Environment Variable in *Oracle Database Globalization Support Guide*.
- Refer to OCIEnvNlsCreate() in *Oracle Call Interface Programmer's Guide*.

Additional Globalization Features

TimesTen OCI also supports the globalization features referenced here. These can be set as environment variables, TimesTen general connection attributes, or TimesTen ODBC connection options.

For the connection options, the names here are prepended by "TT_". An environment variable setting takes precedence over a corresponding connection attribute or connection option setting. A connection option setting takes precedence over a corresponding connection attribute setting.

- NLS_LENGTH_SEMANTICS: By default, the lengths of character data types CHAR and VARCHAR2 are specified in bytes, not characters. For single-byte character encoding this works well. For multibyte character encoding, you can use NLS_LENGTH_SEMANTICS to create CHAR and VARCHAR2 columns using character-length semantics instead. Supported settings are BYTE (default) and CHAR. (NCHAR and NVARCHAR2 columns are always character-based. Existing columns are not affected.)
- NLS_SORT: This specifies the type of sort for character data. It overrides the default value from NLS_LANG. Valid values are BINARY or any linguistic sort name supported by TimesTen. For example, to specify the German linguistic sort sequence, set NLS_SORT=German.
- NLS_NCHAR_CONV_EXCP: This determines whether an error is reported when there is data loss during an implicit or explicit character type conversion between NCHAR or NVARCHAR data and CHAR or VARCHAR2 data. Valid settings are TRUE and FALSE. The default value is FALSE, resulting in no error being reported.

Refer to Globalization Support in *Oracle TimesTen In-Memory Database Operations Guide*, Setting Up a Globalization Support Environment in *Oracle Database Globalization Support Guide* and Attribute Support for ODBC 3.5 SQLSetConnectAttr and SQLGetConnectAttr.

TimesTen Restrictions and Limitations

This section discusses the following areas of restrictions for OCI in TimesTen compared to in Oracle Database:

- Oracle Database Features Not Supported by TimesTen
- TimesTen OCI Limitations

Oracle Database Features Not Supported by TimesTen

TimesTen does not support OCI calls that are related to functionality that does not exist in TimesTen, which do not support these Oracle Database features:

- Advanced Queuing
- Any Data
- Object support
- Collections
- Cartridge Services
- Direct path loading
- Date/time intervals
- Iterators
- BFILEs
- Cryptographic Toolkit
- XML DB support
- Spatial Services
- Event handling
- Session switching
- Scrollable cursors

TimesTen OCI Limitations

There are certain restrictions in TimesTen OCI.

- Asynchronous calls are not supported.
- Connection pooling and session pooling are not supported.
- Describing objects with `OCIDescribeAny()` is supported only by name. Describing PL/SQL objects is not supported. (Also see the entry for this function under Supported OCI Calls.)
- TimesTen client/server automatic client failover is not supported.
- The `TNSPING` utility does not recognize connections to TimesTen.

- Retrieving implicit ROWID values from INSERT, UPDATE, and DELETE statements is not supported. (This is supported for SELECT FOR UPDATE statements, however.)
- TimesTen built-in procedures that return result sets are not supported directly. You can, however, use PL/SQL for this purpose, as shown in Use of PL/SQL in OCI to Call a TimesTen Built-In Procedure.
- Only a single REF CURSOR can be returned from a PL/SQL block, procedure call, or function call.
- Binding and defining of structures through OCIBindArrayOfStruct() and OCIDefineArrayOfStruct() is supported for SQL statements but not for PL/SQL. (Also see the entries for these functions under Supported OCI Calls.)
- Oracle Database utilities such as SQL*Plus and SQL*Loader are not supported. (In TimesTen, you can use ttIsql instead of SQL*Plus and ttBulkCp instead of SQL*Loader. See Utilities in *Oracle TimesTen In-Memory Database Reference*.)
- Array binding, the ability to bind arrays into PL/SQL statements, is supported for associative arrays (index-by tables or PL/SQL tables) but is *not* supported for varrays (variable size arrays) or nested tables. (See Binding Associative Arrays in TimesTen OCI.)
- Both TimesTen and Oracle Database support XA, but TimesTen does not support XA through OCI.
- With OCI, TimesTen automatically disables autocommit for DML statements. Transactions should be explicitly committed or rolled back when finished.

Getting Started With TimesTen OCI

Use these methods for getting started with a TimesTen OCI application.

- Environment Variables for TimesTen OCI
- About Compiling and Linking OCI Applications
- Connecting to a TimesTen Database From OCI
- OCI Error Handling
- Signal Handling and Diagnostic Framework Considerations

Environment Variables for TimesTen OCI

You set certain environment variables for executing a TimesTen OCI application.

Settings apply to both direct connections and client/server connections except as noted.

See Table 3-1.

> **Note:**
>
> - After creating an instance, you can set your environment as appropriate through the `timesten_home/bin/ttenv` script applicable to your operating system. See Environment Variables in the *Oracle TimesTen In-Memory Database Installation, Migration, and Upgrade Guide* for information about `ttenv` scripts.
> - To ensure proper generation of OCI programs to be run on TimesTen, `ORACLE_HOME` cannot be set for OCI compilations.

Table 3-1 Environment Variables for TimesTen OCI

Variable	Required or Optional	Settings
`LD_LIBRARY_PATH` (Linux or UNIX) `PATH` (Windows)	Required	Must be set so that the TimesTen Instant Client directory precedes the Oracle Database libraries in the path. The path is set properly if you use the following script under *timesten_home*: `bin/ttenv`
`TNS_ADMIN`	Required if you use the `tnsnames` naming method	Specifies the directory where the `tnsnames.ora` file is located. This is also where TimesTen looks for a `sqlnet.ora` file. See Connecting to a TimesTen Database From OCI.
`TWO_TASK` (Linux or UNIX) `LOCAL` (Windows)	Optional	You can use this, whichever is appropriate for your platform, instead of specifying the *dbname* argument in your OCI logon call. The setting consists of a valid TNS name or easy connect string. See Connecting to a TimesTen Database From OCI.
`NLS_LANG`	Optional	See Specifying a Character Set. Only the character set component is honored and it must indicate a character set supported by TimesTen. The language and territory values are ignored. This environment variable overrides the TimesTen default character set.
`NLS_SORT`	Optional	See Additional Globalization Features. The sort order must be a value supported by TimesTen. This overrides the TimesTen `NLS_SORT` general connection attribute.

Table 3-1 (Cont.) Environment Variables for TimesTen OCI

Variable	Required or Optional	Settings
NLS_LENGTH_SEMANTICS	Optional	See Additional Globalization Features. This overrides the TimesTen NLS_LENGTH_SEMANTICS general connection attribute.
NLS_NCHAR_CONV_EXCP	Optional	See Additional Globalization Features. This overrides the TimesTen NLS_NCHAR_CONV_EXCP general connection attribute.

> **Note:**
>
> Refer to NLS General Connection Attributes in *Oracle TimesTen In-Memory Database Reference*.

About Compiling and Linking OCI Applications

No changes are required between Oracle Database and TimesTen for the steps to compile and link an OCI application.

OCI programs that use the Oracle Client library shipped with TimesTen do not have to be recompiled or relinked to be executed with TimesTen unless there has been a major upgrade to the Oracle version provided with TimesTen.

Connecting to a TimesTen Database From OCI

TimesTen OCI uses the Oracle Instant Client to connect to the TimesTen database. You can connect to the database through either the `tnsnames` or the *easy connect* naming method, similarly to how you would connect to an Oracle database through those methods.

This section covers the following topics. All but the first apply to only TimesTen Classic:

- About Configuring OCI Connections in TimesTen Scaleout
- Using the tnsnames Naming Method to Connect
- Using an Easy Connect String to Connect
- Configuring Whether to Use tnsnames.ora or Easy Connect

Refer to Configuring Naming Methods in *Oracle Database Net Services Administrator's Guide* for additional information about `tnsnames`, easy connect, and the `tnsnames.ora` file.

> **Note:**
>
> Although the sqlnet mechanism is used for a TimesTen OCI connection, the connection goes through the TimesTen ODBC driver, not the Oracle Database sqlnet driver.

About Configuring OCI Connections in TimesTen Scaleout

In TimesTen Scaleout, TimesTen will automatically populate the tnsnames.ora file and sqlnet.ora file, as applicable, on all instances with entries for all TimesTen connectables you have defined.

See Connectable Operations in *Oracle TimesTen In-Memory Database Reference*.

The instructions here are not relevant, as the user is not allowed to manually configure those entries. The tnsnames, sqlnet, and related information for additional entries, such as for Oracle database connections (as applicable), is brought in and distributed through the ttGridAdmin TNSNamesImport and SQLNetImport commands. See Oracle Database Operations in *Oracle TimesTen In-Memory Database Reference*.

Using the tnsnames Naming Method to Connect

TimesTen supports tnsnames syntax. You can use a TimesTen tnsnames.ora entry the same way you would use an Oracle Database tnsnames.ora entry.

The syntax of a TimesTen entry in tnsnames.ora is as follows:

```
tns_entry = (DESCRIPTION =
              (CONNECT_DATA =
                (SERVICE_NAME = dsn)
                (SERVER = timesten_direct | timesten_client)))
```

Where *tns_entry* is the arbitrary TNS name you assign to the entry. You can use this as the *dbname* argument in OCILogon(), OCILogon2(), and OCIServerAttach() calls.

DESCRIPTION and CONNECT_DATA are required as shown.

For SERVICE_NAME, *dsn* must be a TimesTen DSN that is configured in the sys.odbc.ini or user odbc.ini file that is visible to a user running the OCI application. On Windows, the DSN can be specified by using the ODBC Data Source Administrator. See Managing TimesTen Databases in *Oracle TimesTen In-Memory Database Operations Guide*.

For SERVER, timesten_direct specifies a direct connection to TimesTen or timesten_client specifies a client/server connection. If you choose timesten_client, the DSN must be configured as a client/server database.

As always, the host and port of the TimesTen server are determined from entries in the sys.ttconnect.ini file, according to the DSN. See Working With the TimesTen Client and Server in *Oracle TimesTen In-Memory Database Operations Guide*.

Here is a sample tnsnames.ora entry for a direct connection:

```
my_tnsname = (DESCRIPTION =
               (CONNECT_DATA =
```

```
                   (SERVICE_NAME = my_dsn)
                   (SERVER = timesten_direct)))
```

You can use the TNS name, `my_tnsname`, in either of the following ways:

- Specify "`my_tnsname`" for the *`dbname`* argument in your OCI logon call.
- Specify an empty string for *`dbname`* and set `TWO_TASK` or `LOCAL` to "`my_tnsname`".

For example:

```
OCILogon2(envhp, errhp, &svchp,
         (text *)"user1", (ub4)strlen("user1"),
         (text *)"pwd1", (ub4)strlen("pwd1"),
         (text *)"my_tnsname", (ub4)strlen((char*)"my_tnsname"), OCI_DEFAULT));
```

Refer to Connect, Authorize, and Initialize Functions in *Oracle Call Interface Programmer's Guide* for details about OCI logon calling sequences.

Or on a UNIX system, for example, you can set `TWO_TASK` to "`my_tnsname`" and use an OCI logon call with an empty string for *`dbname`*:

```
OCILogon2(envhp, errhp, &svchp,
         (text *)"user1", (ub4)strlen("user1"),
         (text *)"pwd1", (ub4)strlen("pwd1"),
         (text *)"", (ub4)0, OCI_DEFAULT));
```

> **Note:**
>
> For TimesTen Classic, you can use the `ttInstanceCreate -tnsadmin` option or the `ttInstanceModify -tns_admin` option (in addition to the `TNS_ADMIN` environment variable) to set the `tnsnames` location. See ttInstanceCreate and ttInstanceModify in *Oracle TimesTen In-Memory Database Reference*.

Using an Easy Connect String to Connect

TimesTen supports easy connect syntax, which enhances the Instant Client package by enabling connections to be made without configuring `tnsnames.ora`.

An easy connect string has syntax similar to a URL, in the following format:

```
[//]host[:port]/service_name:server[/instance]
```

The initial double-slash is optional. A host name must be specified to satisfy easy connect syntax, but is otherwise ignored by TimesTen. The name "`localhost`" is typically used by convention. Any value specified for the port is also ignored. For client/server connections, the host and port of the TimesTen server are determined from entries in the `sys.ttconnect.ini` file, according to the TimesTen DSN.

Specify the DSN for *`service_name`*. Specify `timesten_client` or `timesten_direct`, as appropriate, for *`server`*.

TimesTen ignores the *`instance`* field and does not require that it be specified.

For example, the following easy connect string connects to a TimesTen server using the client/server libraries. Assume a DSN `ttclient` in the `sys.odbc.ini` file is resolved as a client/server data source and connects to the corresponding host and port specified in the `sys.ttconnect.ini` file:

```
"localhost/ttclient:timesten_client"
```

The following easy connect string is for a direct connection to TimesTen. Assume the DSN `ttdirect` is defined in `sys.odbc.ini`:

```
"localhost/ttdirect:timesten_direct"
```

You can use an easy connect string in either of the following ways:

- Specify it for the *`dbname`* argument in your OCI logon call.
- Specify an empty string for *`dbname`* and set `TWO_TASK` or `LOCAL` to the easy connect string, in quotes.

For example:

```
OCILogon2(envhp, errhp, &svchp,
         (text *)"user1", (ub4)strlen("user1"),
         (text *)"pwd1", (ub4)strlen("pwd1"),
         (text *)"localhost/ttclient:timesten_client",
         (ub4)strlen((char*)"localhost/ttclient:timesten_client"), OCI_DEFAULT));
```

Refer to Connect, Authorize, and Initialize Functions in *Oracle Call Interface Programmer's Guide* for details about OCI logon calling sequences.

Or on a UNIX system, for example, you can set `TWO_TASK` to "`localhost/ttclient:timesten_client`" and use an OCI logon call with an empty string for *`dbname`*, as follows.

```
OCILogon2(envhp, errhp, &svchp,
         (text *)"user1", (ub4)strlen("user1"),
         (text *)"pwd1", (ub4)strlen("pwd1"),
         (text *)"", (ub4)0, OCI_DEFAULT));
```

Configuring Whether to Use tnsnames.ora or Easy Connect

If a `sqlnet.ora` file is present, it specifies the naming methods that are tried and the order in which they are tried.

The Instant Client looks for a `sqlnet.ora` file at the `TNS_ADMIN` location, if applicable. If `TNS_ADMIN` has not been set but `ORACLE_HOME` has been (such as if you had a previous Instant Client installation), the default `sqlnet.ora` location is in *`ORACLE_HOME`*`/network/admin` as noted in Parameters for the sqlnet.ora File in *Oracle Database Net Services Reference*.

If `sqlnet.ora` is found and does not indicate a particular naming method, you cannot use that method. If `sqlnet.ora` is not found, you can use either method.

In TimesTen, you can access sample copies of `tnsnames.ora` and `sqlnet.ora` in the *`timesten_home`*`/install/network/admin/samples` directory. Here is the `sqlnet.ora` file that TimesTen provides, which supports both `tnsnames` and easy connect ("`EZCONNECT`"):

```
# To use ezconnect syntax or tnsnames, the following entries must be
# included in the sqlnet.ora configuration.
#
NAMES.DIRECTORY_PATH= (TNSNAMES, EZCONNECT)
```

With this file, TimesTen first looks for `tnsnames` syntax in your OCI logon calls. If it cannot find `tnsnames` syntax, it looks for easy connect syntax.

OCI Error Handling

OCI error handling includes error reporting and transient errors.

This section discusses these topics:

- OCI Error Reporting
- Transient Errors (OCI)

OCI Error Reporting

Errors under TimesTen OCI applications return Oracle Database error codes.

TimesTen attempts to report the same error code as Oracle Database would under similar conditions. The error messages may come from either the TimesTen error catalog or the Oracle Database error catalog. Some error messages may indicate the accompanying TimesTen error code if appropriate.

Fatal errors are those that make the database inaccessible until after error recovery. When a fatal error occurs, all database connections are required to disconnect in order to avoid out-of-memory conditions. No further operations may complete. Shared memory from the old TimesTen instance is not freed until all active connections at the time of the error have disconnected.

Fatal errors in OCI are indicated by the Oracle Database error code `ORA-03135` or `ORA-00600`. Error handling for these errors should be different from standard error handling. In particular, the application error-handling code should have a disconnect from the database.

Transient Errors (OCI)

TimesTen automatically resolves most transient errors (which is particularly important for TimesTen Scaleout), but if your application detects the following error, it is suggested to retry the current transaction.

- `ORA-57005`: Transient transaction failure due to unavailability of resource. Roll back the transaction and try it again.

> **Note:**
>
> Search the entire error stack for errors returning these error types before deciding whether it is appropriate to retry.

This is returned in the `errcodep` parameter in `OCIErrorGet()` and may be encountered by any of the following OCI calls:

- `OCIBindArrayOfStruct()`
- `OCIBindByName()`
- `OCIBindByPos()`

- `OCIDefineArrayOfStruct()`
- `OCIDefineByPos()`
- `OCIDescribeAny()`
- `OCILogoff()`
- `OCILogon()`
- `OCILogon2()`
- `OCIPing()`
- `OCISessionBegin()`
- `OCISessionEnd()`
- `OCISessionGet()`
- `OCISessionRelease()`
- `OCIStmtExecute()`
- `OCIStmtFetch()`
- `OCIStmtFetch2()`
- `OCIStmtGetBindInfo()`
- `OCIStmtPrepare()`
- `OCIStmtPrepare2()`
- `OCIStmtRelease()`
- `OCITransCommit()`
- `OCITransRollback()`

Signal Handling and Diagnostic Framework Considerations

The OCI diagnostic framework installs signal handlers that may impact any signal handling that you use in your application. You can disable OCI signal handling by setting `DIAG_SIGHANDLER_ENABLED=FALSE` in the `sqlnet.ora` file.

Refer to Controlling ADR Creation and Disabling Fault Diagnosability Using sqlnet.ora in *Oracle Call Interface Programmer's Guide*.

TimesTen Features With OCI

This section covers the following topics for developers using TimesTen OCI.

- TimesTen Deferred Prepare
- Parameter Binding Features in TimesTen OCI
- Using Cache Operations With TimesTen OCI
- LOBs in TimesTen OCI
- Configuring the Result Set Buffer Size in Client/Server Using OCI
- Use of PL/SQL in OCI to Call a TimesTen Built-In Procedure

TimesTen Deferred Prepare

In OCI, a prepare call is expected to be a lightweight operation performed on the client.

To enable TimesTen to be consistent with this expectation, and to avoid unwanted round trips between client and server, the TimesTen client library implementation of `SQLPrepare` performs what is referred to as a *deferred prepare*, where the request is not sent to the server until required. See TimesTen Deferred Prepare.

Parameter Binding Features in TimesTen OCI

There are features relating to binding parameters into SQL or PL/SQL from an OCI application.

- Binding Duplicate Parameters in TimesTen OCI
- Binding Associative Arrays in TimesTen OCI

Binding Duplicate Parameters in TimesTen OCI

In TimesTen OCI, as in ODBC, multiple occurrences of the same parameter name are considered to be distinct parameters. However, OCI allows multiple occurrences to be bound with a single call to `OCIBindByPos()`.

See Binding of Duplicate Parameters in SQL Statements.

Consider this query:

```
SELECT * FROM employees
  WHERE employee_id < :a AND manager_id > :a AND salary < :b;
```

The two occurrences of parameter `a` are considered to be separate parameters, but you have the option of binding both occurrences with a single call to `OCIBindByPos()`:

```
OCIBindByPos(..., 1, ...); /* both occurrences of :a */
OCIBindByPos(..., 3, ...); /* occurrence of :b */
```

Or you can bind the two occurrences of `a` separately:

```
OCIBindByPos(..., 1, ...); /* first occurrence of :a */
OCIBindByPos(..., 2, ...); /* second occurrence of :a */
OCIBindByPos(..., 3, ...); /* occurrence of :b */
```

Note that in both cases, parameter `b` is considered to be in position 3.

Note:

OCI also allows parameters to be bound by name, rather than by position, using `OCIBindByName()`. In this case, the same value is used for any parameters that have the same name.

Binding Associative Arrays in TimesTen OCI

Associative arrays, formerly known as index-by tables or PL/SQL tables, are supported as `IN`, `OUT`, or `IN OUT` bind parameters in TimesTen PL/SQL, such as from an OCI application. This enables arrays of data to be passed efficiently between an application and the database.

An associative array is a set of key-value pairs. In TimesTen, for associative array binding (but not for use of associative arrays only within PL/SQL), the keys, or indexes, must be integers—`BINARY_INTEGER` or `PLS_INTEGER`. The values must be simple scalar values of the same data type. For example, there could be an array of department managers indexed by department numbers. Indexes are stored in sort order, not creation order.

You can declare an associative array type and then an associative array from PL/SQL as in the following example (note the `INDEX BY`):

```
declare
   TYPE VARCHARARRTYP IS TABLE OF VARCHAR2(30) INDEX BY BINARY_INTEGER;
   x VARCHARARRTYP;
   ...
```

For Pro*C/C++, see Associative Array Bindings in TimesTen Pro*C/C++.

For related information, see Using Associative Arrays From Applications in *Oracle TimesTen In-Memory Database PL/SQL Developer's Guide*.

> **Note:**
>
> Be aware of the following restrictions in TimesTen:
>
> - The following types are not supported in binding associative arrays: LOBs, REF CURSORs, `TIMESTAMP`, `ROWID`.
> - Associative array binding is not allowed in passthrough statements.
> - General bulk binding of arrays is not supported in TimesTen OCI. Varrays and nested tables are not supported as bind parameters.

TimesTen supports associative array binds in OCI by supporting the `maxarr_len` and `*curelep` parameters of the `OCIBindByName()` and `OCIBindByPos()` functions. These parameters are used to indicate that the binding is for an associative array.

The complete calling sequences for those functions are as follows:

```
sword OCIBindByName ( OCIStmt *stmtp,
                      OCIBind **bindpp,
                      OCIError *errhp,
                      const OraText *placeholder,
                      sb4 placeh_len,
                      void *valuep,
                      sb4 value_sz,
                      ub2 dty,
                      void *indp,
                      ub2 *alenp,
                      ub2 *rcodep,
                      ub4 maxarr_len,
                      ub4 *curelep,
```

```
                     ub4 mode );

sword OCIBindByPos ( OCIStmt *stmtp,
                     OCIBind **bindpp,
                     OCIError *errhp,
                     ub4 position,
                     void *valuep,
                     sb4 value_sz,
                     ub2 dty,
                     void *indp,
                     ub2 *alenp,
                     ub2 *rcodep,
                     ub4 maxarr_len,
                     ub4 *curelep,
                     ub4 mode );
```

The `maxarr_len` and `*curelep` parameters are used as follows when you bind an associative array. (They should be set to 0 if you are not binding an associative array.)

- `maxarr_len`: This is an input parameter indicating the maximum array length. This is the maximum number of elements that the associative array can accommodate.
- `*curelep`: This is an input/output parameter indicating the current array length. It is a pointer to the actual number of elements in the associative array before and after statement execution.

See OCIBindByName() and OCIBindByPos() in *Oracle Call Interface Programmer's Guide*.

In the following example, an OCI application binds an integer array and a character array to corresponding `OUT` associative arrays in a PL/SQL procedure.

Assume this SQL setup:

```
DROP TABLE FOO;

CREATE TABLE FOO (CNUM INTEGER,
                  CVC2 VARCHAR2(20));

INSERT INTO FOO VALUES ( null,
     'VARCHAR  1');
INSERT INTO FOO VALUES (-102,
     null);
INSERT INTO FOO VALUES ( 103,
     'VARCHAR  3');
INSERT INTO FOO VALUES (-104,
     'VARCHAR  4');
INSERT INTO FOO VALUES ( 105,
     'VARCHAR  5');
INSERT INTO FOO VALUES ( 106,
     'VARCHAR  6');
INSERT INTO FOO VALUES ( 107,
     'VARCHAR  7');
INSERT INTO FOO VALUES ( 108,
     'VARCHAR  8');

COMMIT;
```

Assume this PL/SQL package definition. This has the `INTEGER` associative array type `NUMARRTYP` and the `VARCHAR2` associative array type `VCHARRTYP`, used for output associative arrays `c1` and `c2`, respectively, in the definition of procedure `P1`.

```
CREATE OR REPLACE PACKAGE PKG1 AS
  TYPE NUMARRTYP IS TABLE OF INTEGER INDEX BY BINARY_INTEGER;
  TYPE VCHARRTYP IS TABLE OF VARCHAR2(20) INDEX BY BINARY_INTEGER;

  PROCEDURE P1(c1 OUT NUMARRTYP,c2 OUT VCHARRTYP);

END PKG1;
/

CREATE OR REPLACE PACKAGE BODY PKG1 AS

  CURSOR CUR1 IS SELECT CNUM, CVC2 FROM FOO;

  PROCEDURE P1(c1 OUT NUMARRTYP,c2 OUT VCHARRTYP) IS
  BEGIN
    IF NOT CUR1%ISOPEN  THEN
      OPEN CUR1;
    END IF;
    FOR i IN 1..8 LOOP
      FETCH CUR1 INTO c1(i), c2(i);
      IF CUR1%NOTFOUND THEN
        CLOSE CUR1;
        EXIT;
      END IF;
    END LOOP;
  END P1;

END PKG1;
```

The following OCI program calls `PKG1.P1`, binds arrays to the `P1` output associative arrays, and prints the contents of those associative arrays. Note in particular the `OCIBindByName()` function calls to do the binding.

```
static OCIEnv *envhp;
static OCIServer *srvhp;
static OCISvcCtx *svchp;
static OCIError *errhp;
static OCISession *authp;
static OCIStmt *stmthp;
static OCIBind *bndhp[MAXCOLS];
static OCIBind *dfnhp[MAXCOLS];

STATICF VOID outbnd_1()
{
  int   i;
  int   num[MAXROWS];
  char*   vch[MAXROWS][20];

  unsigned int    numcnt   = 5;
  unsigned int    vchcnt = 5;

  unsigned short    alen_num[MAXROWS];
  unsigned short    alen_vch[MAXROWS];
  unsigned short    rc_num[MAXROWS];
  unsigned short    rc_vch[MAXROWS];

  short    indp_num[MAXROWS];
  short    indp_vch[MAXROWS];

/* Assume the process is connected and srvhp, svchp, errhp, authp, and stmthp
   are all allocated/initialized/etc. */
```

```
char  *sqlstmt = (char *)"BEGIN PKG1.P1(:c1, :c2); END; ";

for (i = 0; i < MAXROWS; i++)
{
  alen_num[i] = 0;
  alen_vch[i] = 0;
  rc_num[i] = 0;
  rc_vch[i] = 0;
  indp_num[i] = 0;
  indp_vch[i] = 0;
}

DISCARD printf("Running outbnd_1.\n");
DISCARD printf("\n----> %s\n", sqlstmt);
checkerr(errhp, OCIStmtPrepare(stmthp, errhp, sqlstmt,
         (unsigned int)strlen((char *)sqlstmt),
         (unsigned int) OCI_NTV_SYNTAX, (unsigned int) OCI_DEFAULT));

bndhp[0] = 0;
bndhp[1] = 0;

checkerr(errhp, OCIBindByName(stmthp, &bndhp[0], errhp,
                (char *) ":c1", (sb4) strlen((char *) ":c1"),
                (dvoid *) &num[0], (sb4) sizeof(num[0]), SQLT_INT,
                (dvoid *) &indp_num[0], (unsigned short *) &alen_num[0],
                (unsigned short *) &rc_num[0],
                (unsigned int) MAXROWS, (unsigned int *) &numcnt,
                (unsigned int) OCI_DEFAULT));

checkerr(errhp, OCIBindByName(stmthp, &bndhp[1], errhp,
                (char *) ":c2", (sb4) strlen((char *) ":c2"),
                (dvoid *) vch[0], (sb4) sizeof(vch[0]), SQLT_CHR,
                (dvoid *) &indp_vch[0], (unsigned short *) &alen_vch[0],
                (unsigned short *) &rc_vch[0],
                (unsigned int) MAXROWS, (unsigned int *) &vchcnt,
                (unsigned int) OCI_DEFAULT));

DISCARD printf("\nTo execute the PL/SQL statement.\n");

checkerr(errhp, OCIStmtExecute(svchp, stmthp, errhp, (unsigned int) 1,
                (unsigned int) 0, (const OCISnapshot*) 0,
                (OCISnapshot*) 0,  (unsigned int) OCI_DEFAULT));

DISCARD printf("\nHere are the results:\n\n");

DISCARD printf("Column 1, INTEGER: \n");
for (i = 0; i < numcnt; i++)
{
  if (indp_num[i] == -1)
    DISCARD printf("-NULL- ");
  else
    DISCARD printf("%5d, ", num[i]);
  DISCARD printf("ind = %d,  len = %d, rc = %d\n",
                           indp_num[i], alen_num[i], rc_num[i]);
}

DISCARD printf("\nColumn 2, VARCHAR2(20): \n");
for (i = 0; i < vchcnt; i++)
{
  if (indp_vch[i] == -1)
```

```
      DISCARD printf("-NULL-      ");
    else
      DISCARD printf("%.*s, ", alen_vch[i], vch[i]);
    DISCARD printf("ind = %d,  len = %d, rc = %d\n",
                         indp_vch[i], alen_vch[i], rc_vch[i]);
  }

  DISCARD printf("\nDone\n");
  return;
}
```

> **Note:**
>
> The `alen_*` arrays are arrays of lengths; the `rc_*` arrays are arrays of return codes; the `indp_*` arrays are arrays of indicators.

Using Cache Operations With TimesTen OCI

This section discusses TimesTen OCI features related to using cache operations in TimesTen Classic:

- Specifying the Oracle Database Password in OCI for Cache
- Determining the Number of Cache Instances Affected by an Action

Specifying the Oracle Database Password in OCI for Cache

To use cache, there must be a cache administration user in the TimesTen database with the same name as an Oracle Database cache administration user who can select from and update the cached Oracle Database tables. This Oracle Database cache administration user could alternatively be a schema user. The password of the TimesTen cache administration user can be different from the password of the Oracle Database cache administration user. See Setting Up a Caching Infrastructure in *Oracle TimesTen In-Memory Database Cache Guide*.

For use of OCI with cache operations, TimesTen enables you to pass the Oracle Database cache administration user's password through OCI by appending it to the password field in an `OCILogon()` or `OCILogon2()` call when you log in to TimesTen. Use the attribute `OraclePWD` in the connect string, such as in the following example:

```
text *cacheadmin = (text *)"cacheadmin1";
text *cachepwds = (text *)"ttpassword;OraclePWD=oraclepassword";
text *ttdbname = (text *)"tt_tnsname";
....
OCILogon2(envhp, errhp, &svchp,
       (text *)cacheadmin, (ub4)strlen(cacheadmin),
       (text *)cachepwds, (ub4)strlen(cachepwds),
       (text *)ttdbname, (ub4)strlen(ttdbname), OCI_DEFAULT));
```

You must always specify `OraclePWD`, even if the Oracle Database cache administration user's password is the same as the TimesTen cache administration user's password.

Note the following for the example:

- The name of the TimesTen cache administration user, as well as the name of the Oracle Database cache administration user who can access the cached Oracle Database tables, is `cacheadmin1`.
- The password of the TimesTen cache administration user is *`ttpassword`*.
- The password of the Oracle Database cache administration user is *`oraclepassword`*.
- The TNS name of the TimesTen database being connected to is `tt_tnsname`.

The Oracle database is specified through the TimesTen `OracleNetServiceName` general connection attribute in the `sys.odbc.ini` or user `odbc.ini` file.

Alternatively, instead of using a TNS name, you could use easy connect syntax or the `TWO_TASK` or `LOCAL` environment variable, as discussed in preceding sections.

Determining the Number of Cache Instances Affected by an Action

In TimesTen OCI, following the execution of a `FLUSH CACHE GROUP`, `LOAD CACHE GROUP`, `REFRESH CACHE GROUP`, or `UNLOAD CACHE GROUP` statement, the OCI function `OCIAttrGet()` with the `OCI_ATTR_ROW_COUNT` argument returns the number of cache instances that were flushed, loaded, refreshed, or unloaded.

See Determining the Number of Cache Instances Affected by an Operation in the *Oracle TimesTen In-Memory Database Cache Guide*.

LOBs in TimesTen OCI

TimesTen Classic supports LOBs (large objects). This includes CLOBs (character LOBs), NCLOBs (national character LOBs), and BLOBs (binary LOBs). This section focuses on LOB locators, temporary LOBs, and OCI LOB APIs and features.

The following topics are covered here for OCI:

- LOB Locators in OCI
- Temporary LOBs in OCI
- Differences Between TimesTen LOBs and Oracle Database LOBs in OCI
- Using the LOB Simple Data Interface in OCI
- About Using the LOB Locator Interface in OCI
- Creating a Temporary LOB in OCI
- Accessing the Locator of a Persistent LOB in OCI
- Reading and Writing LOB Data Using the OCI LOB Locator Interface
- OCI Client-Side Buffering
- LOB Prefetching in OCI
- Passthrough LOBs in OCI

See the following for additional information:

- Large Objects (LOBs), which is ODBC-oriented but also provides general overview of LOBs, differences between TimesTen and Oracle Database LOBs, and LOB programming interfaces

- LOB Data Types in *Oracle TimesTen In-Memory Database SQL Reference* for additional information about LOBs in TimesTen
- LOB and BFILE Operations in *Oracle Call Interface Programmer's Guide* for complete information about LOBs and how to use them in OCI, keeping in mind that TimesTen does not support BFILEs, SecureFiles, array reads and writes for LOBs, or callback functions for LOBs

> **Note:**
>
> The LOB piecewise data interface is not applicable to OCI applications. (You can, however, manipulate LOB data in pieces through features of the LOB locator interface.)

LOB Locators in OCI

OCI provides the LOB locator interface, where a LOB consists of a LOB locator and a LOB value. The locator acts as a handle to the value. When an application selects a LOB from the database, it receives a locator. When it updates the LOB, it does so through the locator. And when it passes a LOB as a parameter, it is passing the locator, not the actual value. See About Using the LOB Locator Interface in OCI. (Note that in OCI it is also possible to use the simple data interface, which does not involve a locator. See Using the LOB Simple Data Interface in OCI.)

To update a LOB, your transaction must have an exclusive lock on the row containing the LOB. You can accomplish this by selecting the LOB with a `SELECT ... FOR UPDATE` statement. This results in a writable locator. With a simple `SELECT` statement, the locator is read-only. Read-only and writable locators behave as follows:

- A read-only locator is *read consistent*, meaning that throughout its lifetime, it sees only the contents of the LOB as of the time it was selected. Note that this would include any uncommitted updates made to the LOB within the same transaction before the LOB was selected.
- A writable locator is updated with the latest data from the database each time a write is made through the locator. So each write is made to the most current data of the LOB, including updates that have been made through other locators.

The following example details behavior for two writable locators for the same LOB:

1. The LOB column contains "XY".
2. Select locator `L1` for update.
3. Select locator `L2` for update.
4. Write "Z" through `L1` at offset 1.
5. Read through locator `L1`. This would return "ZY".
6. Read through locator `L2`. This would return "XY", because `L2` remains read-consistent until it is used for a write.
7. Write "W" through `L2` at offset 2.
8. Read through locator `L2`. This would return "ZW". Prior to the write in the preceding step, the locator was updated with the latest data ("ZY").

Temporary LOBs in OCI

A temporary LOB exists only within an application, and in TimesTen OCI has a lifetime no longer than the transaction in which it was created (as is the case with the lifetime of any LOB locator in TimesTen). You can think of a temporary LOB as a scratch area for LOB data.

An OCI application can instantiate a temporary LOB explicitly, for use within the application, through the appropriate API. (See About Using the LOB Locator Interface in OCI.) A temporary LOB may also be created implicitly by TimesTen. For example, if a `SELECT` statement selects a LOB concatenated with an additional string of characters, TimesTen implicitly creates a temporary LOB to contain the concatenated data and an OCI application would receive a locator for the temporary LOB.

Temporary LOBs are stored in the TimesTen temporary data region.

Differences Between TimesTen LOBs and Oracle Database LOBs in OCI

A key difference between the LOB implementation for TimesTen versus Oracle Database is that in TimesTen, LOB locators do not remain valid past the end of the transaction. All LOB locators are invalidated after a commit or rollback, whether explicit or implicit. This includes after any DDL statement.

Also see Differences Between TimesTen LOBs and Oracle Database LOBs.

Using the LOB Simple Data Interface in OCI

The simple data interface enables applications to access LOB data by binding and defining, as with other scalar data types. The application can use a LOB type that is compatible with the corresponding variable type.

Use `OCIStmtPrepare()` to prepare a statement. For binding parameters, use `OCIBindByName()` or `OCIBindByPos()`. For defining result columns, use `OCIDefineByPos()`.

For example, an OCI application can bind a CLOB parameter by calling `OCIBindByName()` with a data type of `SQLT_CHR`. Use `OCIStmtExecute()` to execute the statement. For an NCLOB parameter, use data type `SQLT_CHR` and set the OCI `csform` attribute (`OCI_ATTR_CHARSET_FORM`) to `SQLCS_NCHAR`. For a BLOB parameter, you can use data type `SQLT_BIN`.

Use of the simple data interface through OCI is shown in the following examples.

> **Note:**
>
> The simple data interface, through `OCIBindByName()`, `OCIBindByPos()`, or `OCIDefineByPos()`, limits bind sizes to 64 KB.

For examples that follow, assume the table and variables shown here.

```
person(ssn number, resume clob)

OCIEnv *envhp;
```

```
OCIServer *srvhp;
OCISvcCtx *svchp;
OCIError *errhp;
OCISession *authp;
OCIStmt *stmthp;

/* Bind Handles */
OCIBind *bndp1 = (OCIBind *) NULL;
OCIBind *bndp2 = (OCIBind *) NULL;

/* Define Handles */
OCIDefine *defnp1 = (OCIDefine *) NULL;
OCIDefine *defnp2 = (OCIDefine *) NULL;

#define DATA_SIZE 50
#define PIECE_SIZE 10
#define NPIECE (DATA_SIZE/PIECE_SIZE)

char col2[DATA_SIZE];
char col2Res[DATA_SIZE];
ub2 col2len = DATA_SIZE;
sb4 ssn = 123456;
...

text *ins_stmt = (text *)"INSERT INTO PERSON VALUES (:1, :2)";
text *sel_stmt = (text *)"SELECT * FROM PERSON_1 ORDER BY SSN";
...
```

The following example executes an `INSERT` statement using the simple data interface in OCI.

```
for (i=0;i<DATA_SIZE;i++)
  col2[i] = 'A';

/* prepare SQL insert statement */
OCIStmtPrepare (stmthp, errhp, ins_stmt, strlen(ins_stmt), OCI_NTV_SYNTAX,
    OCI_DEFAULT);

/* bind parameters 1 and 2 using OCI_DEFAULT (not data-at-exec) */
OCIBindByPos (stmthp, &bndp1, errhp, 1, (dvoid *) &ssn, sizeof(ssn),
 SQLT_INT, 0, 0, 0, 0, 0, OCI_DEFAULT);
OCIBindByPos (stmthp, &bndp2, errhp, 2, (dvoid *) col2, col2len,
                           SQLT_CHR, 0, 0, 0, 0, 0, OCI_DEFAULT);

/* execute insert statement */
OCIStmtExecute (svchp, stmthp, errhp, 1, 0, 0, 0, OCI_DEFAULT);
```

This next example executes a `SELECT` statement using the simple data interface in OCI. It uses the `SELECT` statement through the variable `sel_stmt` defined above.

```
/* prepare select statement */
OCIStmtPrepare (stmthp, errhp, sel_stmt, strlen(sel_stmt), OCI_NTV_SYNTAX,
    OCI_DEFAULT);

/* define result columns 1 and 2 using OCI_DEFAULT (not data-at-exec) */
OCIDefineByPos (stmthp, &defnp1, errhp, 1, (dvoid*) &ssn, sizeof(ssn),
                              SQLT_INT, 0, 0, 0, OCI_DEFAULT);
OCIDefineByPos (stmthp, &defnp2, errhp, 2, (dvoid *) col2Res, sizeof(col2),
                              SQLT_CHR, 0, &col2len, 0, OCI_DEFAULT);

/* execute select statement */
OCIStmtExecute (svchp, stmthp, errhp, (ub4)1, (ub4)0, (OCISnapshot *) NULL,
```

```
                              (OCISnapshot *) NULL, OCI_DEFAULT));

/* col2Res should now have a DATA_SIZE sized string of 'A's. */
```

About Using the LOB Locator Interface in OCI

You can use the OCI LOB locator interface to work with either a LOB from the database or a temporary LOB, either piece-by-piece or in whole chunks.

In order to use the LOB locator interface, the application must have a valid LOB locator. For a temporary LOB, this may be obtained explicitly through an `OCILobCreateTemporary()` call, or implicitly through a SQL statement that results in creation of a temporary LOB (such as `SELECT c1 || c2 FROM myclob`). For a persistent LOB, use a SQL statement to obtain the LOB locator from the database. (There are examples later in this section.)

Bind types are `SQLT_CLOB` for CLOBs and `SQLT_BLOB` for BLOBs. For NCLOBs, use `SQLT_CLOB` and also set the OCI `csform` attribute (`OCI_ATTR_CHARSET_FORM`) to `SQLCS_NCHAR`.

Sections that follow discuss using LOB locators in various scenarios.

Refer to LOB Functions in *Oracle Call Interface Programmer's Guide* for detailed information and additional examples for OCI LOB functions, noting that TimesTen does not support features specifically intended for BFILEs, SecureFiles, array reads and writes for LOBs, or callback functions for LOBs.

> **Tip:**
>
> LOB manipulations through APIs that use LOB locators result in usage of TimesTen temporary space. Any significant number of such manipulations may necessitate a size increase for the TimesTen temporary data region. See TempSize in *Oracle TimesTen In-Memory Database Reference*.

> **Note:**
>
> - If an invalid LOB locator is assigned to another LOB locator using `OCILobLocatorAssign()`, the target of the assignment is also freed and marked as invalid.
> - `OCILobLocatorAssign()` can be used on a temporary LOB, but `OCILobAssign()` cannot.

Creating a Temporary LOB in OCI

An OCI application can create a temporary LOB by using the `OCILobCreateTemporary()` function, which has an input/output parameter for the LOB locator, after first calling `OCIDescriptorAlloc()` to allocate the locator. When you are finished, use `OCIDescriptorFree()` to free the allocation for the locator and use `OCILobFreeTemporary()` to free the temporary LOB itself.

> **Tip:**
>
> In TimesTen, creation of a temporary LOB results in creation of a database transaction if one is not already in progress. To avoid error conditions, you must execute a commit or rollback to close the transaction.

In TimesTen, any duration supported by Oracle Database (`OCI_DURATION_SESSION`, `OCI_DURATION_TRANSACTION`, or `OCI_DURATION_CALL`) is permissible in the `OCILobCreateTemporary()` call; however, in TimesTen the lifetime of the temporary LOB itself is no longer than the lifetime of the transaction.

Note that the lifetime of a temporary LOB can be shorter than the lifetime of the transaction in the following scenarios:

- If `OCI_DURATION_CALL` is specified
- If the application calls `OCILobFreeTemporary()` on the locator before the end of the transaction
- If the application calls `OCIDurationBegin()` to start a user-specified duration for the temporary LOB, then calls `OCIDurationEnd()` before the end of the transaction

Following are examples of some of the OCI LOB functions mentioned above. See Temporary LOB Support in *Oracle Call Interface Programmer's Guide*.

```
if (OCIDescriptorAlloc((void*)envhp, (void **)&tblob,(ub4)OCI_DTYPE_LOB,
    (size_t)0, (void**)0))
{
   printf("failed in OCIDescriptor Alloc in select_and_createtemp \n");
   return OCI_ERROR;
}

...

if (OCILobCreateTemporary(svchp, errhp, tblob, (ub2)0, SQLCS_IMPLICIT,
    OCI_TEMP_BLOB, OCI_ATTR_NOCACHE, OCI_DURATION_TRANSACTION))
{
   (void) printf("FAILED: OCILobCreateTemporary() \n");
   return OCI_ERROR;
}

...

if(OCILobFreeTemporary(svchp,errhp,tblob))
{
   printf ("FAILED: OCILobFreeTemporary() call \n");
   return OCI_ERROR;
}
```

Accessing the Locator of a Persistent LOB in OCI

An application typically accesses a LOB from the database by using a SQL statement to obtain or access a LOB locator, then passing the locator to an appropriate API function.

A LOB that has been created using the `EMPTY_CLOB()` or `EMPTY_BLOB()` SQL function has a valid locator, which an application can then use to insert data into the LOB by selecting it.

Assume the following table definition:

```
CREATE TABLE clobtable (x NUMBER, y DATE, z VARCHAR2(30), lobcol CLOB);
```

1. Prepare an `INSERT` statement. For example:

   ```
   INSERT INTO clobtable ( x, y, z, lobcol )
      VALUES ( 81, sysdate, 'giants', EMPTY_CLOB() )
      RETURNING lobcol INTO :a;
   ```

   Or, to initialize the LOB with some data:

   ```
   INSERT INTO clobtable ( x, y, z, lobcol )
      VALUES ( 81, sysdate, 'giants', 'The Giants finally won a World Series' )
      RETURNING lobcol INTO :a;
   ```

2. Bind the LOB locator to `:a` as shown.
3. Execute the statement. After execution, the locator refers to the newly created LOB.

Then the application can use the LOB locator interface to read or write LOB data through the locator.

Alternatively, an application can use a `SELECT` statement to access the locator of an existing LOB.

The example that follows uses this table:

```
person(ssn number, resume clob)
```

It selects the locator for the LOB column in the `PERSON` table.

```
text *ins_stmt = (text *)"INSERT INTO PERSON VALUES (:1, :2)";
text *sel_stmt = (text *)"SELECT * FROM PERSON WHERE SSN = 123456";
text *ins_empty = (text *)"INSERT INTO PERSON VALUES ( 1, EMPTY_CLOB())";

OCILobLocator *lobp;

ub4   amtp = DATA_SIZE;
ub4   remainder = DATA_SIZE;
ub4    nbytes = PIECE_SIZE;

/* Allocate lob locator */
OCIDescriptorAlloc (envhp, &lobp, OCI_DTYPE_LOB, 0, 0);

/* Insert an empty locator */
OCIStmtPrepare (stmhp, errhp, ins_empty, strlen(ins_empty), OCI_NTV_SYNTAX,
                               OCI_DEFAULT);
OCIStmtExecute (svchp, stmhp, errhp, 1, 0, 0, 0, OCI_DEFAULT);

/* Now select the locator */

OCIStmtPrepare (stmhp, errhp, sel_stmt, strlen(sel_stmt), OCI_NTV_SYNTAX,
                OCI_DEFAULT);

/* Call define for the lob column */
OCIDefineByPos (stmthp, &defnp2, errhp, 1, &lobp, 0 , SQLT_CLOB, 0, 0, 0,
                OCI_DEFAULT);

OCIStmtExecute (svchp, stmhp, errhp, 1, 0, 0, 0, OCI_DEFAULT);
```

Reading and Writing LOB Data Using the OCI LOB Locator Interface

An OCI application can use `OCILobOpen()` and `OCILobClose()` to open and close a LOB. If you do not explicitly open and close a LOB, it is opened implicitly before a read or write and closed implicitly at the end of the transaction.

An application can use `OCILobRead()` or `OCILobRead2()` to read LOB data, `OCILobWrite()` or `OCILobWrite2()` to write LOB data, `OCILobWriteAppend()` or `OCILobWriteAppend2()` to append LOB data, `OCILobErase()` or `OCILobErase2()` to erase LOB data, and various other OCI functions to perform a variety of other actions.

For example, consider a CLOB with the content "Hello World!" You can overwrite and append data by calling `OCILobWrite()` with an offset of 7 to write "I am a new string". This would result in CLOB content being updated to "Hello I am a new string". Or, to erase data from the original "Hello World!" CLOB, you can call `OCILobErase()` with an offset of 7 and an amount (number of characters) of 5, for example, to update the CLOB to "Hello   !" (six spaces).

All the OCI LOB locator interface functions are covered in detail in LOB Functions in *Oracle Call Interface Programmer's Guide*.

> **Note:**
>
> - Oracle Database emphasizes use of the "2" versions of the OCI read and write functions for LOBs (the non-"2" versions were deprecated in the Oracle Database 11.2 release); however, currently in TimesTen there is no technical advantage in using `OCILobRead2()`, `OCILobWrite2()`, and `OCILobWriteAppend2()`, which are intended for LOBs larger than what TimesTen supports.
> - In using any of the LOB read or write functions, be aware that the callback function parameter must be set to `NULL` or 0, because TimesTen does not support callback functions for LOB manipulation.
> - Because TimesTen does not support binding arrays of LOBs, the `OCILobArrayRead()` and `OCILobArrayWrite()` functions are not supported.

The following example shows how to write LOB data using the OCI LOB function `OCILobWrite()` and how to read data using `OCILobRead()`. It uses the table and variables from the example in the preceding section.

```
for (i=0;i<DATA_SIZE;i++)
  col2[i] = 'A';

/*************** Writing to the LOB *****************/

amt = DATA_SIZE;
offset = 1;

/* Write contents of col2 buffer into the LOB in a single chunk via locator lobp */
OCILobWrite (svchp, errhp, lobp, &amt, offset, col2, DATA_SIZE, OCI_ONE_PIECE,
                        0, 0, 0, SQLCS_IMPLICIT);

/*************** Reading from the LOB *****************/
```

```
/* Get the length of the LOB */
OCILobGetLength (svchp, errhp, lobp, &len);
amt = len;

/* Read the LOB data in col2Res in a single chunk */
OCILobRead (svchp, errhp, lobp, &amt, offset, col2Res, DATA_SIZE, 0, 0, 0,
                        SQLCS_IMPLICIT);
```

OCI Client-Side Buffering

OCI provides a facility for client-side buffering on a per-LOB basis. It is enabled for a LOB by a call to `OCILobEnableBuffering()` and disabled by a call to `OCILobDisableBuffering()`.

Enabling buffering for a LOB locator creates a 512 KB write buffer. This size is not configurable. Data written by the application through the LOB locator is buffered. When possible, the client library satisfies LOB read requests from the buffer as well. An application can flush the buffer by a call to `OCILobFlushBuffer()`. Note that buffers are not flushed automatically when they become full, and an attempt to write to the LOB through the locator when the buffer is full results in an error.

The following restrictions apply when you use client-side buffering:

- Buffering is incompatible with the following functions: `OCILobAppend()`, `OCILobCopy()`, `OCILobCopy2()`, `OCILobErase()`, `OCILobGetLength()`, `OCILobTrim()`, `OCILobWriteAppend()`, and `OCILobWriteAppend2()`.
- An application can use `OCILobWrite()` or `OCILobWrite2()` only to append to the end of a LOB.
- LOB data becomes visible to SQL and PL/SQL (server-side) operations only after the application has flushed the buffer.
- When a LOB is selected while there are unflushed client-side writes in its buffer, the unflushed data is not included in the select.

LOB Prefetching in OCI

To reduce round trips to the server in client/server connections, LOB data can be prefetched from the database and cached on the client side during fetch operations. LOB prefetching in OCI has the same functionality in TimesTen as in Oracle Database.

Configure LOB prefetching through the following OCI attributes. Note that size refers to bytes for BLOBs and to characters for CLOBs or NCLOBs.

- `OCI_ATTR_DEFAULT_LOBPREFETCH_SIZE`: Use this to enable prefetching and specify the default prefetch size. A value of 0 (default) disables prefetching.
- `OCI_ATTR_LOBPREFETCH_SIZE`: Set this attribute for a column define handle to specify the prefetch size for the particular LOB column.
- `OCI_ATTR_LOBPREFETCH_LENGTH`: This attribute can be set `TRUE` or `FALSE` (default) to prefetch LOB metadata such as LOB length and chunk size.

The `OCI_ATTR_DEFAULT_LOBPREFETCH_SIZE` and `OCI_ATTR_LOBPREFETCH_LENGTH` settings are independent of each other. You can use LOB data prefetching independently of LOB metadata prefetching.

Refer to Prefetching of LOB Data, Length, and Chunk Size in *Oracle Call Interface Programmer's Guide*.

> **Note:**
>
> The above attribute settings are ignored for direct connections to the database.

Passthrough LOBs in OCI

Passthrough LOBs (LOBs in Oracle Database accessed through TimesTen) are exposed as TimesTen LOBs and are supported by TimesTen in much the same way that any TimesTen LOB is supported.

Note the following:

- You cannot use `OCILobCreateTemporary()` to create a passthrough LOB.
- In addition to copying from one TimesTen LOB to another TimesTen LOB—such as through `OCILobCopy()`, `OCILobCopy2()`, or `OCILobAppend()`—you can copy from a TimesTen LOB to a passthrough LOB, from a passthrough LOB to a TimesTen LOB, or from one passthrough LOB to another passthrough LOB. Any of these copies the LOB value to the target destination. For example, copying a passthrough LOB to a TimesTen LOB copies the LOB value into the TimesTen database.

  An attempt to copy a passthrough LOB to a TimesTen LOB when the passthrough LOB is larger than the TimesTen LOB size limit results in an error.
- TimesTen LOB size limitations do not apply to storage of LOBs in the Oracle database through passthrough. If a passthrough LOB is copied to a TimesTen LOB, the size limit applies to the copy.
- As with TimesTen local LOBs, a locator for a passthrough LOB does not remain valid past the end of the transaction.

The examples that follow highlight key functionality in copying between TimesTen LOBs and passthrough LOBs on Oracle Database. After the table and data setup, the first example uses `OCILobAppend()` to copy LOB data from Oracle Database to TimesTen and the second example uses `OCILobCopy()` to copy LOB data from TimesTen to Oracle Database. (Either call could be used in either case.) Then, for contrast, the third example uses an `UPDATE` statement to copy LOB data from Oracle Database to TimesTen and the fourth example uses an `INSERT` statement to copy LOB data from TimesTen to Oracle Database.

```
 /* Table and data setup */
call ttoptsetflag(''passthrough'', 3)';
DROP TABLE oratab';
CREATE TABLE oratab (i INT, c CLOB)';
INSERT INTO oratab VALUES (1, ''Copy from Oracle to TimesTen'')';
INSERT INTO oratab VALUES (2, EMPTY_CLOB())';
COMMIT;

call ttoptsetflag(''passthrough'', 0)';
DROP TABLE tttab';
CREATE TABLE tttab (i INT, c CLOB)';
INSERT INTO tttab VALUES (1, ''Copy from TimesTen to Oracle'')';
INSERT INTO tttab VALUES (2, EMPTY_CLOB())';
INSERT INTO tttab VALUES (3, NULL)';
COMMIT;
/* Table and data setup end */

/*
```

```
 * Below are four OCI pseudocode examples, for copying LOBs between
 * TimesTen and Oracle using OCI API and INSERT/UPDATE statements.
 */

/* Init OCI Env */

/* Set the passthrough level to 1 */
OCIStmtPrepare (..., "call ttoptsetflag(''passthrough'', 1)'", ...);
OCIStmtExecute (...);

/*
 * 1. Copy a passthrough LOB on Oracle to a TimesTen LOB */

/* Select a passthrough locator on Oracle */
OCIStmtPrepare (..., "SELECT c FROM oratab WHERE i = 1", ...);
OCIDefineByPos (..., (dvoid *)&ora_loc_1, 0 , SQLT_CLOB, ...);
OCIStmtExecute (...);

/* Select a locator on TimesTen for update */
OCIStmtPrepare (..., "SELECT c FROM tttab WHERE i = 2 FOR UPDATE", ...);
OCIDefineByPos (..., (dvoid *)&tt_loc_2, 0 , SQLT_CLOB, ...);
OCIStmtExecute (...);

/* Copy a passthrough LOB on Oracle to a TimesTen LOB */
OCILobAppend(..., tt_loc_2, ora_loc_1);

/*
 * 2. Copy a TimesTen LOB to a passthrough LOB on Oracle */

/* Select a passthrough locator on Oracle for update */
OCIStmtPrepare (..., "SELECT c FROM oratab WHERE i = 2 FOR UPDATE", ...);
OCIDefineByPos (..., (dvoid *)&ora_loc_2, 0 , SQLT_CLOB, ...);
OCIStmtExecute (...);

/* Select a locator on TimesTen */
OCIStmtPrepare (..., "SELECT c FROM tttab WHERE i = 1", ...);
OCIDefineByPos (..., (dvoid *)&tt_loc_1, 0 , SQLT_CLOB, ...);
OCIStmtExecute (...);

/* Copy a passthrough LOB on Oracle to a TimesTen LOB */
OCILobCopy(..., ora_loc_2, tt_loc_1, 28, 1, 1);

/*
 * 3. UPDATE a TimesTen LOB with a passthrough LOB on Oracle */

/* A passthrough LOB, (selected above in case 1) is bound to an UPDATE
statement
 * on TimesTen table */
OCIStmtPrepare (..., "UPDATE tttab SET c = :1 WHERE i = 3", ...);
OCIBindByPos (..., (dvoid *)&ora_loc_1, 0 , SQLT_CLOB, ...);
OCIStmtExecute (...);

/*
 * 4. INSERT a passthrough table on Oracle with a TimesTen LOB */

/* A TimesTen LOB, (selected above in case 2) is bound to an INSERT statement
 * on a passthough table on Oracle */
OCIStmtPrepare (..., "INSERT INTO oratab VALUES (3, :1)", ...);
OCIBindByPos (..., (dvoid *)&tt_loc_1, 0 , SQLT_CLOB, ...);
OCIStmtExecute (...);
```

```
  OCITransCommit (...);

  /* Cleanup OCI Env */
```

Configuring the Result Set Buffer Size in Client/Server Using OCI

For data returned from a `SELECT` statement in client/server, the buffer size for the data returned to the client is configurable to allow adjustments for better performance. (In earlier releases, the buffer size could not be changed.)

The buffer size can be set in terms of either rows of data or bytes of data. The lower limit takes precedence. It is suggested to use one limit and set the other to a value high enough to ensure that it is not reached first.

Use these OCI statement attributes:

- `OCI_ATTR_PREFETCH_ROWS`: Buffer size in rows (default 8192)
- `OCI_ATTR_PREFETCH_MEMORY`: Buffer size in bytes (default 2097152, or 2 MB)

You can set these attributes but not get them.

Here is an example:

```
/* Double the row limit. */
ub4 rowsvalue = 16384;
...
OCIAttrSet(stmthp, handle_type, (dvoid *)&rowsvalue,
           sizeof(ub4), OCI_ATTR_PREFETCH_ROWS, errhp);
```

Note:

- These attributes correspond to TimesTen connection attributes `TT_NetMsgMaxRows` and `TT_NetMsgMaxBytes`, which you can set in a TimesTen connection string or DSN, to serve as initial values for any statements created on the connection.
- The minimum value of each attribute is 1 and at least one row is always returned. Setting either to a value of 0 results in the default value being used. There are no maximum settings other than the maximum value of the datatype (32-bit unsigned integer).
- If a client version that supports these attributes connects to a server version that does not, any settings are ignored.

Use of PL/SQL in OCI to Call a TimesTen Built-In Procedure

TimesTen built-in procedures that return result sets are not supported directly through OCI.

This example shows how to use PL/SQL for this purpose.

```
plsql_resultset_example(OCIEnv *envhp, OCIError *errhp, OCISvcCtx *svchp)
{
  OCIStmt   *stmhp;
```

```
  OCIBind   *bindp;

  sb4        passThruValue = -1;
  char       v_name[255];
  text      *stmt_text;

  /* prepare the plsql statement */
  stmt_text = (text *)
    "declare v_name varchar2(255); "
    "begin execute immediate "
       "'call ttOptGetFlag(''passthrough'')' into v_name, :rc1; "
    "end;";
  OCIStmtPrepare2(svchp, &stmhp, errhp, (text *)stmt_text,
                  (ub4)strlen((char *)stmt_text),
                  (text *)0, (ub4)0,
                  (ub4)OCI_NTV_SYNTAX, (ub4)OCI_DEFAULT);

  /* bind parameter 1 (:v_name) to varchar2 out-parameter */
  OCIBindByPos(stmhp, &bindp, errhp, 1,
               (dvoid*)&v_name, sizeof(v_name), SQLT_CHR,
               (dvoid*)0, (ub2*)0, (ub2*)0, (ub4)0, (ub4*)0,
               OCI_DEFAULT);

  /* execute the plsql statement */
  OCIStmtExecute(svchp, stmhp, errhp, (ub4)1, (ub4)0,
                 (OCISnapshot *)0, (OCISnapshot *)0, (ub4)OCI_DEFAULT);

  /* convert the passthrough string value to an integer */
  passThruValue = (sb4)atoi((const char *)v_name);
  printf("Value of the passthrough flag is %d\n", passThruValue);

  /* drop the statement handle */
  OCIStmtRelease(stmhp, errhp, (text *)0, (ub4)0, (ub4)OCI_DEFAULT);
}
```

TimesTen OCI Support Reference

This is a reference section for TimesTen support of OCI features.

- Supported OCI Calls
- Supported Handles and Attributes
- Supported Descriptors
- Supported OCI-Defined Constants
- Supported Parameter Attributes

Supported OCI Calls

There is TimesTen support for OCI calls.

Table 3-2 lists TimesTen support for OCI calls that are documented for Oracle Database 19c releases.

Some groups of calls are represented with an asterisk in the name. For example, the calls related to Advanced Queuing, which TimesTen does not support, have names that start with `OCIAQ` and are represented in the table as `OCIAQ*()`. OCI date functions, which TimesTen does support, are designated by `OCIDate*()`.

> **Note:**
>
> TimesTen does not support the following features or related calls: Advanced Queueing, Any Data, collections, Data Cartridge, Direct Path Loading, user-defined objects, XML DB.

Table 3-2 TimesTen OCI Supported Calls

OCI Call	Notes
`OCIAppCtxClearAll()`	No notes
`OCIAppCtxSet()`	No notes
`OCIArrayDescriptorAlloc()`	No notes
`OCIArrayDescriptorFree()`	No notes
`OCIAttrGet()`	See Supported Handles and Attributes. TimesTen support includes special usage with cache groups. See Using Cache Operations With TimesTen OCI.
`OCIAttrSet()`	See Supported Handles and Attributes.
`OCIBindArrayOfStruct()`	This is supported for SQL statements but not PL/SQL.
`OCIBindByName()`	The following is an unsupported value for the *mode* parameter: • `OCI_IOV`
`OCIBindByPos()`	The following is an unsupported value for the *mode* parameter: • `OCI_IOV`
`OCIBindDynamic()`	No notes
`OCICharSetConversionIsReplacementUsed()`	No notes
`OCICharSetToUnicode()`	No notes
`OCIClientVersion()`	No notes
`OCIDate*()`	See Table 3-4.
`OCIDefineArrayOfStruct()`	This is supported for SQL statements but not PL/SQL.
`OCIDefineByPos()`	The following is an unsupported value for the *mode* parameter: • `OCI_IOV`
`OCIDefineDynamic()`	No notes

Table 3-2 (Cont.) TimesTen OCI Supported Calls

OCI Call	Notes
`OCIDescribeAny()`	PL/SQL objects are not supported. Describing objects is supported only by name. See Supported Parameter Attributes. The following are unsupported values for the *objptr_typ* parameter: • `OCI_OTYPE_REF` • `OCI_OTYPE_PTR` The following are unsupported values for the *objtyp* parameter: • `OCI_PTYPE_PKG` • `OCI_PTYPE_FUNC` • `OCI_PTYPE_PROC` • `OCI_PTYPE_SYN` • `OCI_PTYPE_TYPE` When you use the setting `OCI_PTYPE_DATABASE` for the *objtyp* parameter, use the predetermined name `$TT_DB_NAME$` as the database name for the **objptr* parameter.
`OCIDescriptorAlloc()`	No notes
`OCIDescriptorFree()`	No notes
`OCIDurationBegin()`	Supported for LOBs. Regardless of the duration setting, the duration cannot exceed the lifetime of the transaction.
`OCIDurationEnd()`	Supported for LOBs. Regardless of the duration setting, the duration cannot exceed the lifetime of the transaction.
`OCIEnvCreate()`	The following are unsupported values for the *mode* parameter: • `OCI_EVENTS` • `OCI_NEW_LENGTH_SEMANTICS` • `OCI_NCHAR_LITERAL_REPLACE_ON` • `OCI_NCHAR_LITERAL_REPLACE_OFF` • `OCI_NO_MUTEX` (Instead use `OCI_ENV_NO_MUTEX`.)
`OCIEnvInit()`	The following are unsupported values for the *mode* parameter: • `OCI_NO_MUTEX` • `OCI_ENV_NO_MUTEX` **Note**: Use `OCIEnvCreate()` instead of `OCIEnvInit()`. `OCIEnvInit()` is supported for backward compatibility.

Table 3-2 (Cont.) TimesTen OCI Supported Calls

OCI Call	Notes
`OCIEnvNlsCreate()`	The following are unsupported values for the *mode* parameter: • `OCI_EVENTS` • `OCI_NCHAR_LITERAL_REPLACE_ON` • `OCI_NCHAR_LITERAL_REPLACE_OFF` • `OCI_NO_MUTEX` (Instead use `OCI_ENV_NO_MUTEX`.)
`OCIErrorGet()`	No notes
`OCIHandleAlloc()`	No notes
`OCIHandleFree()`	No notes
`OCIInitialize()`	The following are unsupported values for the *mode* parameter: • `OCI_NO_MUTEX` • `OCI_ENV_NO_MUTEX` **Note**: Use `OCIEnvCreate()` instead of `OCIInitialize()`. `OCIInitialize()` is supported for backward compatibility.
`OCIInterval*()`	See Table 3-4.
`OCILob*()`	TimesTen supports `OCILob*()` functions other than the following: • Functions specifically intended for array reads and writes • Functions specifically intended for BFILEs • Functions specifically intended for SecureFiles **Notes:** • Regardless of the duration setting in an `OCILobCreateTemporary()` call, the LOB lifetime is no longer than the lifetime of the transaction. • See Reading and Writing LOB Data Using the OCI LOB Locator Interface regarding `OCILobRead2()`, `OCILobWrite2()`, and `OCILobWriteAppend2()`.
`OCILogoff()`	No notes
`OCILogon()`	No notes
`OCILogon2()`	`OCI_DEFAULT` is the only supported value for the *mode* parameter.
`OCIMultiByte*()`	No notes
`OCINls*()`	No notes
`OCINumber*()`	No notes
`OCIParamGet()`	No notes
`OCIParamSet()`	No notes

Table 3-2 (Cont.) TimesTen OCI Supported Calls

OCI Call	Notes
`OCIPing()`	No notes
`OCIRaw*()`	No notes
`OCIRowidToChar()`	No notes
`OCIServer*()`	`OCI_DEFAULT` is the only supported value for the *`mode`* parameter of `OCIServerAttach`.
`OCISessionBegin()`	`OCI_CRED_RDBMS` is the only supported value for the *`credt`* parameter. `OCI_DEFAULT` is the only supported value for the *`mode`* parameter.
`OCISessionEnd()`	No notes
`OCISessionGet()`	TimesTen does not support switching between sessions.
`OCISessionRelease()`	No notes
`OCIStmtExecute()`	The following are unsupported values for the *`mode`* parameter: • `OCI_BATCH_ERRORS` • `OCI_STMT_SCROLLABLE_READONLY` **Note**: Using `OCI_COMMIT_ON_SUCCESS` results in improved performance, avoiding an extra round trip to the server to commit a transaction.
`OCIStmtFetch()`	No notes
`OCIStmtFetch2()`	The only supported values for the *`orientation`* parameter are `OCI_DEFAULT` and `OCI_FETCH_NEXT`.
`OCIStmtGetBindInfo()`	No notes
`OCIStmtPrepare()`	The only supported value for the *`language`* parameter is `OCI_NTV_SYNTAX`.
`OCIStmtPrepare2()`	The only supported value for the *`mode`* parameter is `OCI_DEFAULT`. For statement caching, TimesTen supports the *`key`* argument to tag a statement for future calls to `OCIStmtPrepare2()` or `OCIStmtRelease()`.
`OCIStmtRelease()`	The only supported value for the *`mode`* parameter is `OCI_DEFAULT`. For statement caching, TimesTen supports the *`key`* argument to tag a statement. This can be the key from `OCIStmtPrepare2()`.
`OCIString*()`	No notes
`OCIThread*()`	No notes
`OCITransCommit()`	The only supported value for the *`mode`* parameter is `OCI_DEFAULT`.
`OCITransRollback()`	No notes

Table 3-2 (Cont.) TimesTen OCI Supported Calls

OCI Call	Notes
`OCIUnicodeToCharSet()`	No notes
`OCIUserCallbackGet()`	No notes
`OCIUserCallbackRegister()`	No notes
`OCIWideChar*()`	No notes

Supported Handles and Attributes

There are handles and attributes that TimesTen OCI supports for `OCIAttrGet()` and `OCIAttrSet()` calls.

Table 3-3 lists the handles and attributes that TimesTen OCI supports for `OCIAttrGet()` and `OCIAttrSet()` calls.

See Handle and Descriptor Attributes in *Oracle Call Interface Programmer's Guide*.

Table 3-3 TimesTen OCI Supported Handles and Attributes

Handle	C Object	Supported Attributes

Environment	`OCIEnv`	`OCI_ATTR_ENV_CHARSET_ID`
		`OCI_ATTR_ENV_NCHARSET_ID`
		`OCI_ATTR_ENV_UTF16`
		`OCI_ATTR_OBJECT`

Error	`OCIError`	`OCI_ATTR_DML_ROW_OFFSET`

Service context	`OCISvcCtx`	`OCI_ATTR_ENV`
		`OCI_ATTR_IN_V8_MODE`
		`OCI_ATTR_SERVER`
		`OCI_ATTR_SESSION`
		`OCI_ATTR_TRANS`

Table 3-3 (Cont.) TimesTen OCI Supported Handles and Attributes

Handle	C Object	Supported Attributes
Statement	OCIStmt	OCI_ATTR_BIND_COUNT OCI_ATTR_CURRENT_POSITION OCI_ATTR_ENV OCI_ATTR_FETCH_ROWID OCI_ATTR_NUM_DML_ERRORS OCI_ATTR_PARAM_COUNT OCI_ATTR_PREFETCH_MEMORY (Refer to Configuring the Result Set Buffer Size in Client/Server Using OCI) OCI_ATTR_PREFETCH_ROWS (Refer to Configuring the Result Set Buffer Size in Client/Server Using OCI) OCI_ATTR_ROW_COUNT OCI_ATTR_ROWID OCI_ATTR_ROWS_FETCHED OCI_ATTR_SQLFNCODE OCI_ATTR_STATEMENT OCI_ATTR_STMT_TYPE
Bind	OCIBind	OCI_ATTR_CHARSET_FORM OCI_ATTR_CHARSET_ID OCI_ATTR_MAXCHAR_SIZE OCI_ATTR_MAXDATA_SIZE
Define	OCIDefine	OCI_ATTR_CHARSET_FORM OCI_ATTR_CHARSET_ID OCI_ATTR_MAXCHAR_SIZE
Describe	OCIDescribe	OCI_ATTR_PARAM OCI_ATTR_PARAM_COUNT
Server	OCIServer	OCI_ATTR_ENV OCI_ATTR_IN_V8_MODE OCI_ATTR_SERVER_GROUP OCI_ATTR_SERVER_STATUS
User session	OCISession	OCI_ATTR_ACTION OCI_ATTR_CLIENT_IDENTIFIER OCI_ATTR_CLIENT_INFO OCI_ATTR_CURRENT_SCHEMA OCI_ATTR_DRIVER_NAME OCI_ATTR_INITIAL_CLIENT_ROLES OCI_ATTR_MODULE OCI_ATTR_PROXY_CREDENTIALS OCI_ATTR_USERNAME
Authentication	OCIAuthInfo	Same as for user session handle

Table 3-3 (Cont.) TimesTen OCI Supported Handles and Attributes

Handle	C Object	Supported Attributes
Transaction	OCITrans	OCI_ATTR_TRANS_NAME OCI_ATTR_TRANS_TIMEOUT
Thread	OCIThreadHandle	N/A

Supported Descriptors

There are descriptors that TimesTen OCI supports.

Table 3-4 lists the descriptors that TimesTen OCI supports.

Table 3-4 TimesTen OCI Supported Descriptors

Descriptor	C Object
Parameter (read-only)	OCIParam
ROWID	OCIRowid
ANSI DATE	OCIDateTime
TIMESTAMP	OCIDateTime
TIMESTAMP WITH TIME ZONE	OCIDateTime
TIMESTAMP WITH LOCAL TIME ZONE	OCIDateTime
INTERVAL YEAR TO MONTH	OCIInterval
INTERVAL DAY TO SECOND	OCIInterval
User callback	OCIUcb

Supported OCI-Defined Constants

There are OCI-defined constants that TimesTen OCI supports as well as mappings to TimesTen SQL types.

Table 3-5 lists the OCI-defined constants that TimesTen OCI supports and the mappings to TimesTen SQL types.

Table 3-5 TimesTen OCI Supported OCI-Defined Constants

OCI-defined Constant	TimesTen SQL Type	Notes
SQLT_AFC	CHAR	No notes
SQLT_AVC	CHAR	No notes
SQLT_BDOUBLE	BINARY_DOUBLE	No notes
SQLT_BFLOAT	BINARY_FLOAT	No notes
SQLT_BIN	VARBINARY	No notes
SQLT_BLOB	BLOB	No notes

Table 3-5 (Cont.) TimesTen OCI Supported OCI-Defined Constants

OCI-defined Constant	TimesTen SQL Type	Notes
SQLT_CHR	VARCHAR2	No notes
SQLT_CLOB	CLOB	To write to or read from an NCLOB, set the character set form (csfrm) parameter to SQLCS_NCHAR for applicable function calls.
SQLT_DAT	DATE	No notes
SQLT_DATE	DATE	No notes
SQLT_FLT	NUMBER, BINARY_FLOAT	No notes
SQLT_IBDOUBLE	BINARY_DOUBLE	No notes
SQLT_IBFLOAT	BINARY_FLOAT	No notes
SQLT_INT	NUMBER, TT_INTEGER, TT_BIGINT, TT_SMALLINT, TT_TINYINT	No notes
SQLT_INTERVAL_DS	N/A	Not stored in TimesTen.
SQLT_INTERVAL_YM	N/A	Not stored in TimesTen.
SQLT_LBI	VARBINARY	No notes
SQLT_LNG	VARCHAR2	No notes
SQLT_LVB	VARBINARY	Truncated at 4 MB when stored in TimesTen.
SQLT_LVC	VARCHAR2	Truncated at 4 MB when stored in TimesTen.
SQLT_NUM	NUMBER	No notes
SQLT_ODT	DATE	No notes
SQLT_RDD	ROWID	Rowids are returned in Oracle Database format.
SQLT_RSET	N/A	Only one result set parameter is allowed for each statement. Not stored in TimesTen
SQLT_STR	VARCHAR2	Null-terminated.
SQLT_TIMESTAMP	TIMESTAMP	No notes
SQLT_TIMESTAMP_LTZ	TIMESTAMP	Time zone ignored when stored in TimesTen.
SQLT_TIMESTAMP_TZ	TIMESTAMP	Time zone ignored when stored in TimesTen.
SQLT_UIN	NUMBER, TT_INTEGER, TT_BIGINT, TT_SMALLINT, TT_TINYINT	No notes
SQLT_VBI	VARBINARY	No notes
SQLT_VCS	VARCHAR2	No notes

Table 3-5 (Cont.) TimesTen OCI Supported OCI-Defined Constants

OCI-defined Constant	TimesTen SQL Type	Notes
SQLT_VNU	NUMBER	First byte indicates length of number (length of succeeding bytes).
SQLT_VST	CHAR, VARCHAR2	No notes

Supported Parameter Attributes

There are supported parameter attributes for `OCIDescribeAny()` calls.

Table 3-6 that follows lists supported parameter attributes for `OCIDescribeAny()` calls.

See Describing Schema Metadata in *Oracle Call Interface Programmer's Guide* for information about supported attributes.

Table 3-6 TimesTen OCI Supported Parameter Attributes

Parameter	Supported Attributes
All parameters	OCI_ATTR_NUM_PARAMS OCI_ATTR_OBJ_NAME OCI_ATTR_OBJ_SCHEMA OCI_ATTR_PTYPE
Table and view parameters	OCI_ATTR_NUM_COLS OCI_ATTR_LIST_COLUMNS
PL/SQL procedure and function parameters	OCI_ATTR_LIST_ARGUMENTS
PL/SQL package subprogram parameters	OCI_ATTR_LIST_ARGUMENTS OCI_ATTR_NAME
PL/SQL package parameters	OCI_ATTR_LIST_SUBPROGRAMS
Sequence parameters	OCI_ATTR_OBJID OCI_ATTR_MIN OCI_ATTR_MAX OCI_ATTR_INCR OCI_ATTR_CACHE OCI_ATTR_ORDER OCI_ATTR_HW_MARK

Table 3-6 (Cont.) TimesTen OCI Supported Parameter Attributes

Parameter	Supported Attributes
Column parameters	OCI_ATTR_CHAR_USED OCI_ATTR_CHAR_SIZE OCI_ATTR_DATA_SIZE OCI_ATTR_DATA_TYPE OCI_ATTR_NAME OCI_ATTR_PRECISION OCI_ATTR_SCALE OCI_ATTR_IS_NULL OCI_ATTR_TYPE_NAME OCI_ATTR_SCHEMA_NAME OCI_ATTR_CHARSET_ID OCI_ATTR_CHARSET_FORM
Argument and result parameters	OCI_ATTR_NAME OCI_ATTR_POSITION OCI_ATTR_DATA_TYPE OCI_ATTR_DATA_SIZE OCI_ATTR_PRECISION OCI_ATTR_SCALE OCI_ATTR_LEVEL OCI_ATTR_IS_NULL OCI_ATTR_CHARSET_ID OCI_ATTR_CHARSET_FORM
List parameters	OCI_LTYPE_COLUMN OCI_LTYPE_SCH_OBJ OCI_LTYPE_DB_SCH
Database parameters	OCI_ATTR_VERSION OCI_ATTR_CHARSET_ID OCI_ATTR_NCHARSET_ID OCI_ATTR_LIST_SCHEMAS OCI_ATTR_MAX_PROC_LEN OCI_ATTR_MAX_COLUMN_LEN OCI_ATTR_CURSOR_COMMIT_BEHAVIOR OCI_ATTR_MAX_CATALOG_NAMELEN OCI_ATTR_CATALOG_LOCATION OCI_ATTR_SAVEPOINT_SUPPORT OCI_ATTR_NOWAIT_SUPPORT OCI_ATTR_AUTOCOMMIT_DDL OCI_ATTR_LOCKING_MODE

4 TimesTen Support for Pro*C/C++

TimesTen supports the Oracle Pro*C/C++ Precompiler for C and C++ applications. You can use the precompiler with embedded SQL and PL/SQL applications that access a TimesTen database.

This chapter provides an overview and TimesTen-specific information regarding Pro*C/C++, especially emphasizing differences between using Pro*C/C++ with TimesTen versus with Oracle Database. For complete information about Pro*C/C++, you can refer to *Pro*C/C++ Programmer's Guide* in the Oracle Database library.

Also note that Working With TimesTen Databases in ODBC contains information that may be of general interest regarding TimesTen features.

This chapter includes the following topics:

- Overview of TimesTen Support for Pro*C/C++
- Getting Started With TimesTen Pro*C/C++
- TimesTen Features With Pro*C/C++
- TimesTen Pro*C/C++ Precompiler Options

Overview of TimesTen Support for Pro*C/C++

TimesTen support for the Oracle Pro*C/C++ Precompiler depends on TimesTen OCI. TimesTen OCI depends on the Oracle client library and the TimesTen ODBC libraries.

See Figure 3-1 to see where OCI and Pro*C/C++ fit in the TimesTen architecture.

This chapter contains information specific to using the Oracle Pro*C/C++ Precompiler with TimesTen. The syntax and usage of the Oracle Pro*C/C++ Precompiler with TimesTen is essentially the same as with Oracle Database.

The rest of this section includes the following topics.

- Overview of the Oracle Pro*C/C++ Precompiler
- TimesTen OCI Support With Respect to Pro*C/C++
- Restrictions in TimesTen Support for Pro*C/C++

Overview of the Oracle Pro*C/C++ Precompiler

The Oracle Pro*C/C++ Precompiler enables you to embed SQL statements or PL/SQL blocks directly into C or C++ code. Further, you can use your C or C++ program host variables in your embedded SQL or PL/SQL.

You use a precompilation step to convert the Pro*C/C++ source file into a C or C++ source file. The precompiler accepts the Pro*C/C++ file as input, translates embedded SQL statements into standard Oracle Database runtime library calls, and generates a modified source code file that you can then compile and link. Pro*C/C++ code is linked against the

Oracle Database precompiler `SQLLIB` library, which is included in the TimesTen distribution as part of the Oracle Instant Client.

TimesTen OCI Support With Respect to Pro*C/C++

TimesTen support of the Oracle Pro*C/C++ Precompiler depends on TimesTen OCI support. Because of this, restrictions for TimesTen OCI apply to Pro*C/C++ applications.

In addition, TimesTen does not support OCI calls that are related to functionality that does not exist in TimesTen.

See TimesTen Support for OCI. Much of the information there applies to Pro*C/C++ applications as well.

Restrictions in TimesTen Support for Pro*C/C++

There are restrictions when using TimesTen support for Pro*C/C++.

- Embedded SQL Support and Restrictions
- Semantic Checking Restrictions
- Embedded PL/SQL Restrictions
- Transaction Restrictions
- Connection Restrictions
- Summary of Unsupported or Restricted Executable Commands and Clauses

Embedded SQL Support and Restrictions

The TimesTen Pro*C/C++ Precompiler does not support embedded SQL for functionality that TimesTen does not support.

See TimesTen Restrictions and Limitations.

TimesTen provides the following support for `SQLLIB` functions:

- `SQLErrorGetText` (`sqlglmt`) is supported.
- `SQLRowidGet()` is supported following only `SELECT FOR UPDATE` statements.

In addition, TimesTen support for the Oracle Pro*C/C++ Precompiler has the following restrictions:

- `REGISTER CONNECT` is not supported.
- Stored Java subprograms are not supported.

Semantic Checking Restrictions

TimesTen support for the Oracle Pro*C/C++ Precompiler does not provide semantic checking during precompilation. A `SQLCHECK` precompiler option setting that specifies semantic checking is permissible but has no effect.

It is important to be aware, however, that a setting of `SEMANTICS` results in a database connection even though precompilation semantic checking is not performed. Therefore, a setting of `SEMANTICS` requires the following during precompilation:

- The database must be running.
- The `USERID` precompiler option must be set, either on the command line or in the `pcscfg.cfg` configuration file. You must provide the user name and password for an existing TimesTen user, and a TNS name that points to the database. In the following example, you are prompted for the password:

  ```
  USERID=user1@my_tnsname
  ```

  Alternatively, you can enter `USERID=user1/`*`password`*`@my_tnsname`, but for security reasons it is not advisable to specify a password on a command line or in a configuration file.

See Connecting to a TimesTen Database From Pro*C/C++ for information about usage and syntax for TNS names.

See the next section, Embedded PL/SQL Restrictions, for related information about Pro*C/C++ programs that use PL/SQL.

Embedded PL/SQL Restrictions

In TimesTen, if a Pro*C/C++ application contains PL/SQL blocks, then Pro*C/C++ acts as though the `SQLCHECK` setting is `SEMANTICS`. It is important to be aware that this results in a database connection even though precompilation semantic checking is not performed.

Therefore, using PL/SQL in a Pro*C/C++ application requires the following during precompilation:

- The database must be running.
- The `USERID` precompiler option must be set, specifying an existing TimesTen user. See the preceding section, Semantic Checking Restrictions.

Transaction Restrictions

Regarding transactions, TimesTen support for the Oracle Pro*C/C++ Precompiler has some restrictions.

The following is not supported:

- `SAVEPOINT` SQL statement
- `SET TRANSACTION` SQL statement

  You can still have transactions with commit and rollback, just not the `SET TRANSACTION` SQL statement.
- Fetch across commits
- Distributed transactions

Connection Restrictions

Regarding connections, TimesTen support for the Oracle Pro*C/C++ Precompiler does not provide certain features.

- `ALTER AUTHORIZATION` clause
- Automatic connections to the database

- Making connections to the database with SYSDBA or SYSOPER privilege, given that these privileges do not exist in TimesTen
- Implicit connections (dblinks) to a TimesTen or Oracle Database

For information about supported connection syntax, see Connecting to a TimesTen Database From Pro*C/C++.

Summary of Unsupported or Restricted Executable Commands and Clauses

Given TimesTen restrictions, including those noted in the preceding sections, this section summarizes the Pro*C/C++ EXEC SQL executable commands, categories of commands, and command clauses that TimesTen does not support or supports only partially.

- ALTER AUTHORIZATION
- CACHE FREE ALL
- CALL

  This is supported only for calling PL/SQL. To call TimesTen built-in procedures, use dynamic SQL statements.
- Any "COLLECTION..." command
- COMMIT FORCE 'some text'
- COMMIT WORK COMMENT 'some text' RELEASE

  The COMMENT clause is not supported.
- CONNECT BY
- CONTEXT OBJECT OPTION GET
- CONTEXT OBJECT OPTION SET
- DECLARE CURSOR

  The WITH HOLD clause is not supported.
- DECLARE TABLE

  Only Oracle Database data types are supported.
- DECLARE TYPE
- EXPLAIN PLAN
- IN SYSDBA MODE
- IN SYSOPER MODE
- LOCK TABLE
- Any "OBJECT..." command
- PARTITION
- REGISTER CONNECT
- RETURN
- RETURNING
- SAVEPOINT

- `SET DESCRIPTOR`

  You cannot set `CHARACTER_SET_NAME`.
- `SET TRANSACTION`
- `START WITH`
- `TO SAVEPOINT`

Getting Started With TimesTen Pro*C/C++

There are methods to get started with a Pro*C/C++ application for TimesTen.

- Environment and Configuration for TimesTen Pro*C/C++
- Building a Pro*C/C++ Application
- Connecting to a TimesTen Database From Pro*C/C++
- Error Reporting and Handling

Environment and Configuration for TimesTen Pro*C/C++

The Pro*C/C++ system configuration file `pcscfg.cfg` contains the precompiler options for precompilation of your Pro*C/C++ source code. In TimesTen, you must use the version of this file that TimesTen provides. This typically happens automatically if you ensure appropriate configuration for TimesTen through the *`timesten_home`*`/bin/ttenv` script.

Before building a Pro*C/C++ application, you must set up your environment. You can use the TimesTen Classic Quick Start OCI and Pro*C/C++ Makefiles to implement appropriate environment settings. See About TimesTen Quick Start and Sample Applications.

Then confirm `LD_LIBRARY_PATH` or `PATH` is set so that the Oracle Instant Client directory precedes the Oracle Database libraries in the path. The path is set properly if you use the `ttenv` script.

See Environment Variables in the *Oracle TimesTen In-Memory Database Installation, Migration, and Upgrade Guide* for information about `ttenv`.

> **Note:**
>
> To ensure proper generation of OCI and Pro*C/C++ programs to be run on TimesTen, do not set `ORACLE_HOME` for OCI and Pro*C/C++ compilations (or unset it if it was set previously).

Building a Pro*C/C++ Application

Once you have set up the environment, use steps to build a Pro*C/C++ application.

The steps shown here present a basic example for a UNIX system and assume the program has no other includes (`#include`) or links to other libraries. The designation *`instantclient`* represents the directory where Oracle Instant Client is installed.

1. Precompile the Pro*C/C++ source file by using the `proc` command from your system prompt. For example:

   ```
   % proc iname=sample.pc
   ```

   The `proc` utility takes a `.pc` source file as input and produces a `.c` file.

2. Compile the resulting C code file. On Linux platforms, enter a command similar to the following:

   ```
   % gcc -c sample.c -I(instantclient)/sdk/include
   ```

3. Link the resulting object modules with modules in `SQLLIB`. For example:

   ```
   % gcc -o sample sample.o -L(instantclient) -lclntsh
   ```

Connecting to a TimesTen Database From Pro*C/C++

There are methods for connecting to a TimesTen database from a Pro*C/C++ application. TimesTen Pro*C/C++ and TimesTen OCI use the Oracle Instant Client to connect to the database.

Refer to Connecting to a TimesTen Database From OCI for additional configuration steps to use the `tnsnames` naming method or easy connect naming method to connect to the database.

The following topics are covered here for TimesTen Classic:

- Connection Syntax and Parameters
- Using tnsnames or Easy Connect
- Specifying the Oracle Database Password in Pro*C/C++ for Cache

> **Note:**
>
> - Be aware that in TimesTen Scaleout, TimesTen will automatically populate the `tnsnames.ora` file and `sqlnet.ora` file, as applicable, on all instances with entries for all TimesTen connectables you have defined. See Connecting to a TimesTen Database From OCI.
> - A TimesTen connection cannot be inherited from a parent process. If a process opens a database connection before creating (forking) a child process, the child must not use the connection. In Pro*C/C++, to avoid having a child process inadvertently inherit a connection from its parent, use `EXEC SQL COMMIT RELEASE` in the parent before creating the child.

Connection Syntax and Parameters

TimesTen requires a connection syntax.

```
EXEC SQL CONNECT{:user IDENTIFIED BY :pwd | :user_string}
  [[AT{dbname |:host_variable}]USING :connect_string];
```

The parameters are described in Table 4-1.

Table 4-1 Connection Parameters

Parameter	Description
user	User name
pwd	Password
user_string	Alternative to separate *user* and *pwd* entries This is a user name and password separated by a slash, such as `user1/`*`password`*. After an "@" sign, you can also have a database identifier, instead of using *dbname*, or a TNS name or easy connect string, instead of using *connect_string*. See examples in the next section, Using tnsnames or Easy Connect.
dbname	Database identifier declared in a previous `DECLARE DATABASE` statement
host_variable	Variable whose value is a database identifier
connect_string	Valid TNS name or easy connect string for a TimesTen database

Using tnsnames or Easy Connect

To connect to a TimesTen database from a Pro*C/C++ application, you must configure a TNS name or easy connect string for the database.

Perform the `tnsnames` or easy connect steps described under Connecting to a TimesTen Database From OCI.

From Pro*C/C++, you can use a host variable to specify the user name, password, and a TNS name. For example:

```
EXEC SQL CONNECT :dbstring
```

Where `dbstring` is set to "`user1/`*`password`*`@my_tnsname`".

Alternatively, the host variable could specify the user name, password, and an easy connect string. For example, `dbstring` could be set to "`user1/`*`password`*`@localhost/ttclient:timesten_client`".

Or, if the `TWO_TASK` or `LOCAL` environment variable, as applicable for your operating system, is set to "`my_tnsname`" or "`localhost/ttclient:timesten_client`", you could connect as in the following example:

```
EXEC SQL CONNECT :user1 IDENTIFIED BY :pwd1
```

Specifying the Oracle Database Password in Pro*C/C++ for Cache

For use of Pro*C/C++ with cache operations, TimesTen enables you to pass the Oracle Database cache administration user's password through Pro*C/C++ by appending it to the password field in an `EXEC SQL CONNECT` call when you log in to TimesTen.

To use cache operations, there must be a cache administration user in the TimesTen Classic database with the same name as an Oracle Database cache administration user who can select from and update the cached Oracle Database tables. This Oracle Database cache administration user could also be a schema user. The password of the TimesTen cache administration user can be different from the password of the Oracle Database cache administration user. See Setting Up a Caching Infrastructure in *Oracle TimesTen In-Memory Database Cache Guide*.

Use the attribute `OraclePWD` in the connect string, such as in the following example:

```
text *cacheadmin = (text *)"cacheadmin1";
text *cachepwds = (text *)"ttpassword;OraclePWD=oraclepassword";
text *dbname = (text *)"tt_tnsname";
....
EXEC SQL CONNECT :cacheadmin IDENTIFIED BY :cachepassword AT :dbname
```

You must always specify `OraclePWD`, even if the Oracle Database cache administration user's password is the same as the TimesTen cache administration user's password. Furthermore, in the circumstance of specifying an Oracle Database password for cache operations, you must use a form of `EXEC SQL CONNECT` that specifies the password as a separate host variable. In this example, `cacheadmin1` is the name of the TimesTen cache administration user as well as the name of the Oracle Database cache administration user who can access the cached Oracle Database tables, *`ttpassword`* is the password of the TimesTen cache administration user, *`oraclepassword`* is the password of the Oracle Database cache administration user, and `tt_tnsname` is the TNS name of the TimesTen database being connected to. The Oracle database is specified through the TimesTen `OracleNetServiceName` general connection attribute in the `sys.odbc.ini` or user `odbc.ini` file.

Alternatively, instead of using the `AT` clause with a TNS name, you could use the `TWO_TASK` or `LOCAL` environment variable, as discussed in Connecting to a TimesTen Database From OCI.

Error Reporting and Handling

Be aware of restrictions regarding certain error conditions and error reporting.

- Errors under TimesTen Pro*C/C++ applications return Oracle Database error codes. TimesTen attempts to report the same error code as Oracle Database would under similar conditions. The error messages may come from either the TimesTen catalog or the Oracle Database catalog. Some error messages may indicate the accompanying TimesTen error code if appropriate. Pro*C/C++ applications that rely on parsing error codes should be checked.
- TimesTen automatically resolves most transient errors (which is particularly important for TimesTen Scaleout), but if your application detects an `ORA-57005` or `ORA-57007` error, it is suggested to retry the current transaction or most recent API call, as applicable. See Transient Errors (OCI).
- TimesTen supports the `WHENEVER SQLERROR` directive, to go to an error handler if an error occurs, and the `WHENEVER NOT FOUND` directive, to go to a handling section if a "no data found" condition occurs. TimesTen does *not* support the `WHENEVER SQLWARNING` directive.

  Examples:

  ```
  EXEC SQL WHENEVER NOT FOUND GOTO close_cursor;
  ...
  EXEC SQL WHENEVER SQLERROR GOTO error_handler;
  ```

TimesTen Features With Pro*C/C++

This section covers additional features you can use with Pro*C/C++ in TimesTen.

- Associative Array Bindings in TimesTen Pro*C/C++
- LOBs in TimesTen Pro*C/C++

Associative Array Bindings in TimesTen Pro*C/C++

You can pass associative arrays between PL/SQL blocks and Pro*C/C++ applications as well as OCI applications. They can be indexed by a PL/SQL variable of type `BINARY_INTEGER` or `PLS_INTEGER`.

As discussed in Binding Associative Arrays in TimesTen OCI, associative arrays, formerly known as index-by tables or PL/SQL tables, are supported as `IN`, `OUT`, or `IN OUT` bind parameters in TimesTen PL/SQL. See that section for additional information and limitations.

Typically, the entire host array is passed to PL/SQL, but you can use the Pro*C/C++ `ARRAYLEN` statement to specify a smaller array dimension.

See PL/SQL Tables, Host Arrays, and ARRAYLEN Statement in Embedded PL/SQL in *Pro*C/C++ Programmer's Guide*.

The following code excerpt shows the array `salary[]` being bound from Pro*C/C++ into the associative array `num_tab` in PL/SQL.

```
...
float salary[100];
/* populate the host array */
EXEC SQL EXECUTE
  DECLARE
    TYPE NumTabTyp IS TABLE OF REAL
                   INDEX BY BINARY_INTEGER;
    median_salary REAL;
    n BINARY_INTEGER;
...
  FUNCTION median (num_tab NumTabTyp, n INTEGER)
    RETURN REAL IS
  BEGIN
    -- compute median
  END;
  BEGIN
    n := 100;
    median_salary := median(:salary, n);
    ...
  END;
END-EXEC;
...
```

LOBs in TimesTen Pro*C/C++

TimesTen Classic supports LOBs (large objects). This includes CLOBs (character LOBs), NCLOBs (national character LOBs), and BLOBs (binary LOBs).

This section focuses on key Pro*C/C++ LOB features and TimesTen-specific support and restrictions.

These topics are covered:

- Using the LOB Simple Data Interface in Pro*C/C++
- Using the LOB Locator Interface in Pro*C/C++

See the following for additional information:

- Large Objects (LOBs). That section is ODBC-oriented but also provides a general overview of LOBs, differences between TimesTen and Oracle Database LOBs, and LOB programming interfaces.
- LOBs in TimesTen OCI for information about LOB locators, temporary LOBs, using the simple data interface or LOB locator interface in OCI, and additional OCI LOB features.
- LOB Data Types in *Oracle TimesTen In-Memory Database SQL Reference* for additional information about LOBs in TimesTen.
- LOBs in *Pro*C/C++ Programmer's Guide* for complete information about LOBs and how to use them in Pro*C/C++, keeping in mind that TimesTen does not support BFILEs, SecureFiles, array reads and writes for LOBs, or callback functions for LOBs. In particular, see "How to Use LOBs in Your Program" within that chapter.

> **Note:**
>
> - As indicated in the OCI chapter, in TimesTen a LOB used in an application does not remain valid past the end of the transaction.
> - The LOB piecewise data interface is not applicable to OCI or Pro*C/C++ applications in TimesTen. (You can, however, manipulate LOB data in pieces through features of the LOB locator interface.)

Using the LOB Simple Data Interface in Pro*C/C++

The simple data interface enables applications to manipulate LOB data similarly to how they would manipulate other types of scalar data, such as by using `EXEC SQL INSERT` and `EXEC SQL SELECT`. The application can use a LOB type that is compatible with the corresponding variable type.

An application can use the `EMPTY_BLOB()` or `EMPTY_CLOB()` function, as appropriate, to initialize a persistent LOB. This is similar to using `ALLOCATE` in the LOB locator interface, discussed next. Consider the following tables:

```
EXEC SQL CREATE TABLE lob_table ( a_blob BLOB, a_clob CLOB );
...
EXEC SQL INSERT INTO lob_table (a_blob, a_clob)
                VALUES (EMPTY_BLOB(), EMPTY_CLOB());
...
EXEC SQL CREATE TABLE data_table
         ( name VARCHAR2(30), length NUMBER(10), bincol BLOB, charcol CLOB );
```

The following selects LOB data from `data_table` into `myblob` and `myclob`, then inserts the LOB data into `lob_table`.

```
...
OCIBlobLocator *myblob;
OCIClobLocator *myclob;
...
EXEC SQL SELECT bincol, charcol INTO :myblob, :myclob FROM data_table
                WHERE name = :key;
...
// Put data into lob_table.
...
EXEC SQL INSERT INTO lob_table (a_blob, a_clob) VALUES (:myblob, :myclob);
```

To use an NCLOB, declare the variable as follows:

```
OCIClobLocator CHARACTER SET IS NCHAR_CS *mynclob;
```

> **Note:**
>
> The simple data interface, through OCI or Pro*C/C++, limits bind sizes to 64 KB.

Using the LOB Locator Interface in Pro*C/C++

You can use the Pro*C/C++ LOB locator interface to work with either LOBs from the database or temporary LOBs, either piece-by-piece or in whole chunks.

Refer to LOB Statements in *Pro*C/C++ Programmer's Guide* for detailed information about Pro*C/C++ statements for LOBs, noting that TimesTen does not support features specifically intended for BFILEs, SecureFiles, array reads and writes for LOBs, or callback functions for LOBs.

See lobdemo1.pc in *Pro*C/C++ Programmer's Guide* for an end-to-end example.

Also see About Using the LOB Locator Interface in OCI.

> **Tip:**
>
> If Pro*C/C++ syntax does not provide enough functionality to fully specify what you want to accomplish for any operation, you can use the corresponding OCI function as an alternative.

In Pro*C/C++, an application can create a temporary LOB by using the `CREATE TEMPORARY` embedded SQL feature, after first using the `ALLOCATE` feature to allocate the locator. Use `FREE` to free the allocation for the locator and `FREE TEMPORARY` to free the temporary LOB itself. This is shown immediately below.

Also see Creating a Temporary LOB in OCI.

Tip:

In TimesTen, creation of a temporary LOB results in creation of a database transaction if one is not already in progress. To avoid error conditions, execute a commit or rollback to close the transaction.

```
OCIClobLocator *tempclob;
EXEC SQL ALLOCATE :tempclob;
EXEC SQL LOB CREATE TEMPORARY :tempclob;
...
// (Manipulate LOB as desired.)
...
EXEC SQL FREE TEMPORARY :tempclob;
EXEC SQL FREE :tempclob;
```

Alternatively, to specify the LOB character set (here `NCHAR`), you can use the corresponding OCI function:

```
status = OCILobCreateTemporary(svc, err, tempclob, OCI_DEFAULT, SQLCS_NCHAR,
         OCI_TEMP_CLOB, TRUE, OCI_DURATION_TRANSACTION);
```

To access the locator of a persistent LOB in Pro*C/C++, an application typically accesses the LOB from the database by using a SQL statement to obtain the locator, then passing the locator to an appropriate API function.

The following excerpts are from the previously mentioned example in lobdemo1.pc in *Pro*C/C++ Programmer's Guide*. The example uses a CLOB `license_txt` and table `license_table` whose columns are social security number, name, and text summarizing driving offenses (a `CLOB` column). Also see Accessing the Locator of a Persistent LOB in OCI.

```
OCIClobLocator *license_txt;
...
EXEC SQL ALLOCATE :license_txt;
...
EXEC SQL SELECT name, txt_summary INTO :name, :license_txt FROM license_table
                WHERE sss = :sss;
```

To read and write LOB data using the LOB locator interface, a Pro*C/C++ application can use `LOB OPEN` and `LOB CLOSE` to open and close a LOB, `LOB READ` to read LOB data, `LOB WRITE` or `LOB WRITE APPEND` to write or append LOB data, `LOB DESCRIBE` to obtain information about a LOB, and various other Pro*C/C++ features to perform a variety of other actions. All the Pro*C/C++ LOB locator interface features are covered in detail in LOBs in *Pro*C/C++ Programmer's Guide*. Also see Reading and Writing LOB Data Using the OCI LOB Locator Interface in this document.

To write data, use `LOB WRITE ONE` to write the data in a single chunk. TimesTen does not support `LOB WRITE FIRST`, `LOB WRITE NEXT`, or `LOB WRITE LAST` (features of the piecewise data interface).

Here is an example of an `EXEC SQL LOB READ` statement:

```
EXEC SQL LOB READ :amt FROM :blob INTO :buffer;
```

Refer to READ a BLOB, Write a File Example in *Pro*C/C++ Programmer's Guide* for additional information.

Here is an example of an `EXEC SQL LOB WRITE` statement (writing the LOB data in one chunk):

```
EXEC SQL LOB WRITE ONE :amt FROM :buffer INTO :blob;
```

Refer to Read a File, WRITE a BLOB Example in *Pro*C/C++ Programmer's Guide*.

Here is an example of an `EXEC SQL LOB WRITE APPEND` statement:

```
EXEC SQL LOB WRITE APPEND :amt FROM :writebuf INTO :blob;
```

> **Note:**
>
> Opening a LOB is similar conceptually, but not technically, to opening a file. Opening a LOB is more like a hint regarding resources to be required.
>
> Be aware that a LOB being accessed by `OCILobRead()`, `OCILobWrite()`, or equivalent functionality is opened automatically as necessary.

The following excerpt is from the previously mentioned example in lobdemo1.pc in *Pro*C/C++ Programmer's Guide*.

```
...
OCIClobLocator *a_clob;
char *charbuf;
ub4 ClobLen, WriteAmt;
int CharLen = strlen(charbuf);
int NewCharbufLen = CharLen + DATELENGTH + 4;
varchar *NewCharbuf;
NewCharbuf = (varchar *)malloc(2 + NewCharbufLen);
NewCharbuf->arr[0] = '\n';
NewCharbuf->arr[1] = '\0';
strcat((char *)NewCharbuf->arr, charbuf);
NewCharbuf->arr[CharLen + 1] = '\0';
strcat((char *)NewCharbuf->arr, curdate);
NewCharbuf->len = NewCharbufLen;
EXEC SQL LOB DESCRIBE :a_clob GET LENGTH INTO :ClobLen;
WriteAmt = NewCharbufLen;
EXEC SQL LOB WRITE ONE :WriteAmt FROM :NewCharbuf WITH LENGTH :NewCharbufLen
                       INTO :a_clob;
...
```

The next example, like the preceding one, uses `LOB WRITE ONE`. Then it also uses `LOB WRITE APPEND` to append additional data. It writes or appends to the BLOB in 1 K chunks up to `MAX_CHUNKS`.

```
...
 EXEC SQL select b into :blob from t where pk = 1 for update;
 EXEC SQL LOB OPEN :blob READ WRITE;

 // Write/append to the BLOB
 for (i = 0; i < MAX_CHUNKS; i++) {
   if (i==0) { // FIRST CHUNK
     /*
     Write the first piece
     */
     EXEC SQL LOB WRITE ONE :amt FROM :writebuf INTO :blob;
```

```
        }
        else { // All Other Chunks
          /*
          At this point, APPEND all the next pieces
          */
          EXEC SQL LOB WRITE APPEND :amt FROM :writebuf INTO :blob ;
        }
        ...
      }
   ...
```

TimesTen Pro*C/C++ Precompiler Options

This section discusses the use of Pro*C/C++ Precompiler options in TimesTen.

- Precompiler Option Support
- Setting Precompiler Options

Precompiler Option Support

There are supported precompiler options in TimesTen.

SeeTable 4-2.

> **Note:**
>
> TimesTen does not support the following features or related options: Advanced Queueing, database optimization, user-defined objects. Also, TimesTen supports only `CPOOL=NO` and does not support related options.

Table 4-2 TimesTen Pro*C/C++ Precompiler Option Support

Option	Notes
`AUTO_CONNECT`	Supported value: `NO` (default)
`CHAR_MAP`	No notes
`CLOSE_ON_COMMIT`	Supported value: `YES` The Oracle Database default value of `NO` is overridden by TimesTen.
`CODE`	No notes
`COMP_CHARSET`	No notes
`CONFIG`	No notes
`CPOOL`	Supported value: `NO` (default)
`CPP_SUFFIX`	No notes
`DB2_ARRAY`	No notes
`DBMS`	Supported value: `NATIVE` (default)
`DEF_SQLCODE`	No notes

Table 4-2 (Cont.) TimesTen Pro*C/C++ Precompiler Option Support

Option	Notes
DEFINE	No notes
DYNAMIC	No notes
ERRORS	No notes
FIPS	No notes
HEADER	No notes
HOLD_CURSOR	No notes
IMPLICIT_SVPT	Supported value: NO (default)
INAME	No notes
INCLUDE	No notes
INTYPE	No notes
LINES	No notes
LNAME	No notes
LTYPE	No notes
MAX_ROW_INSERT	No notes
MAXLITERAL	No notes
MAXOPENCURSORS	No notes
MODE	No notes
NATIVE_TYPES	No notes
NLS_CHAR	No notes
NLS_LOCAL	Supported value: NO (default)
ONAME	No notes
ORACA	No notes
PAGELEN	No notes
PARSE	No notes
PREFETCH	No notes
RELEASE_CURSOR	No notes
SELECT_ERROR	No notes
SQLCHECK	Not applicable Any of the SQLCHECK settings is allowed, but TimesTen does not support semantic checking during precompilation. Whenever a Pro*C/C++ application uses PL/SQL, Pro*C/C++ acts as though the SQLCHECK setting is SEMANTICS. **Important**: A setting of SEMANTICS (or FULL, which is synonymous) always results in a connection to the database, even though precompilation semantic checking is not performed. See Semantic Checking Restrictions.
STMT_CACHE	No notes

Table 4-2 (Cont.) TimesTen Pro*C/C++ Precompiler Option Support

Option	Notes
SYS_INCLUDE	No notes
THREADS	No notes
TYPE_CODE	No notes
UNSAFE_NULL	No notes
USERID	No notes
UTF16_CHARSET	Supported value: NCHAR_CHARSET
VARCHAR	No notes

Note:

TimesTen does not support the default value for CLOSE_ON_COMMIT. TimesTen supports only CLOSE_ON_COMMIT=YES.

Setting Precompiler Options

You can set precompiler options in one of several ways.

- At compile time, either in the configuration file pcscfg.cfg or on the Pro*C/C++ command line

  A command line setting takes precedence over a setting in the configuration file.
- At runtime through the EXEC ORACLE OPTION command

  A runtime setting takes precedence over a compile-time setting.

For example, the following shows portions of the configuration file that ships with TimesTen.

```
ltype=short
parse=full
close_on_commit=yes
...
```

The following command line would override the ltype=short setting from the configuration file:

```
% proc ltype=long ... iname=sample.pc
```

The following runtime command would override the ltype=long setting from the command line:

```
EXEC ORACLE OPTION LTYPE=NONE;
```

5 XLA and TimesTen Event Management

The TimesTen Transaction Log API (XLA), supported by TimesTen Classic, is a set of C language functions that enable you to implement applications.

You can perform the following:

- Monitor TimesTen for changes to specified tables in a local database.
- Receive real-time notification of these changes.

The primary purpose of XLA is as a high-performance, asynchronous alternative to triggers.

> **Note:**
>
> In the unlikely event that TimesTen replication solutions described in Overview of TimesTen Replication in the *Oracle TimesTen In-Memory Database Replication Guide* do not meet your needs, it is possible to use XLA functions to build a custom data replication solution.

This chapter includes the following topics:

- Overview of TimesTen XLA
- Writing an XLA Event-Handler Application
- Using XLA as a Replication Mechanism
- Other XLA Features

See XLA Reference.

Overview of TimesTen XLA

There are ways to use XLA in TimesTen Classic.

- XLA Basics
- How XLA Reads Records From the Transaction Log
- About XLA and Materialized Views
- About XLA Bookmarks
- XLA Data Types
- XLA System Privilege
- XLA Limitations
- About the XLA Sample Application

XLA functions mentioned here are documented in XLA Reference.

XLA Basics

TimesTen XLA obtains update records directly from the transaction log buffer or transaction log files, so the records are available for as long as they are needed. The logging model also enables multiple readers to simultaneously read transaction log updates.

The ttXlaPersistOpen XLA function opens a connection to the database.

When initially created, TimesTen configures a transaction log handle for the same version as the TimesTen release to which the application is linked.

How XLA Reads Records From the Transaction Log

As applications modify a database, TimesTen generates transaction log records that describe the changes made to the data and other events such as transaction commits. New transaction log records are always written to the end of the log buffer as they are generated.

Transaction log records are periodically flushed in batches from the log buffer in memory to transaction log files on the file system. When XLA is initialized, the XLA application does not have to be concerned with which portions of the transaction log are on the file system or in memory. Therefore, the term "transaction log" as used in this chapter refers to the "virtual" source of transaction update records, regardless of whether those records are physically located in memory or on the file system.

Applications can use XLA to monitor the transaction log for changes to the database. XLA reads through the transaction log, filters the log records, and delivers to XLA applications a list of transaction records that contain the changes to the tables and columns of interest.

XLA sorts the records into discrete transactions. If multiple applications are updating the database simultaneously, transaction log records from the different applications are interleaved in the transaction log.

XLA transparently extracts all transaction log records associated with a particular transaction and delivers them in a contiguous list to the application.

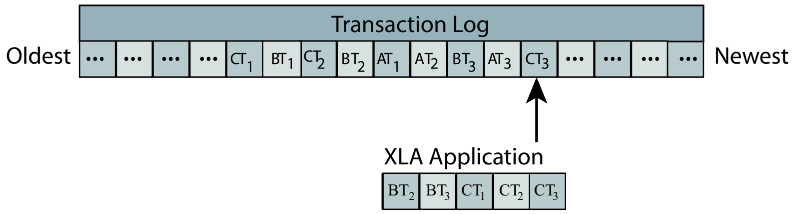
Only the records for committed transactions are returned. They are returned in the order in which their final commit record appears in the transaction log. XLA filters out records associated with changes to the database that have not yet been committed.

If a change is made but then rolled back, XLA does not deliver the records for the canceled transaction to the application.

Most of these basic XLA concepts are demonstrated in the example that follows and summarized in the bulleted list following the example.

Consider the example transaction log illustrated in Figure 5-1.

Figure 5-1 Records Extracted From the Transaction Log



In this example, the transaction log contains the following records:

`CT1` - Application `C` updates row 1 of table `W` with value 7.7.
`BT1` - Application `B` updates row 3 of table `X` with value 2.
`CT2` - Application `C` updates row 9 of table `W` with value 5.6.
`BT2` - Application `B` updates row 2 of table `Y` with value "XYZ".
`AT1` - Application `A` updates row 1 of table `Z` with value 3.
`AT2` - Application `A` updates row 3 of table `Z` with value 4.
`BT3` - Application `B` commits its transaction.
`AT3` - Application `A` rolls back its transaction.
`CT3` - Application `C` commits its transaction.

An XLA application that is set up to detect changes to tables `W`, `Y`, and `Z` would see the following:

`BT2` and `BT3` - Update row 2 of table `Y` with value "XYZ" and commit.
`CT1` - Update row 1 of table `W` with value 7.7.
`CT2` and `CT3` - Update row 9 of table `W` with value 5.6 and commit.

This example demonstrates the following:

- Transaction records of applications `B` and `C` all appear together.
- Although the records for application `C` begin to appear in the transaction log before those for application `B`, the commit for application `B` (`BT3`) appears in the transaction log before the commit for application `C` (`CT3`). As a result, the records for application `B` are returned to the XLA application ahead of those for application `C`.
- The application `B` update to table `X` (`BT1`) is not presented because XLA is not set up to detect changes to table `X`.
- The application `A` updates to table `Z` (`AT1` and `AT2`) are never presented because it did not commit and was rolled back (`AT3`).

About XLA and Materialized Views

You can use XLA to track changes to both tables and materialized views.

A materialized view provides a single source from which you can track changes to selected rows and columns in multiple detail tables. Without a materialized view, the XLA application would have to monitor and filter the update records from all of the detail tables, including records reflecting updates to rows and columns of no interest to the application.

In general, there are no operational differences between the XLA mechanisms used to track changes to a table or a materialized view.

For more information about materialized views, see the following:

- CREATE MATERIALIZED VIEW in *Oracle TimesTen In-Memory Database SQL Reference*
- Understanding Materialized Views in *Oracle TimesTen In-Memory Database Operations Guide*

About XLA Bookmarks

Each XLA reader uses XLA bookmarks to maintain its position in the log update stream.

These topics are covered:

- XLA Log Record Identifiers
- Creating or Reusing a Bookmark
- How Bookmarks Work
- Replicated Bookmarks
- XLA Bookmarks and Transaction Log Holds

XLA Log Record Identifiers

Each bookmark consists of two pointers that track update records in the transaction log by using *log record identifiers*.

- An Initial Read log record identifier points to the most recently acknowledged transaction log record. Initial Read log record identifiers are stored in the database, so they are persistent across database connections, shutdowns, and failures.
- A Current Read log record identifier points to the record currently being read from the transaction log.

Creating or Reusing a Bookmark

When you call the `ttXlaPersistOpen` function to initialize an XLA handle, you have a *`tag`* parameter to identify either a new bookmark or one that exists in the system, and an *`options`* parameter to specify whether it is a new non-replicated bookmark, a new replicated bookmark, or an existing (reused) bookmark.

See ttXlaPersistOpen and Initializing XLA and Obtaining an XLA Handle.

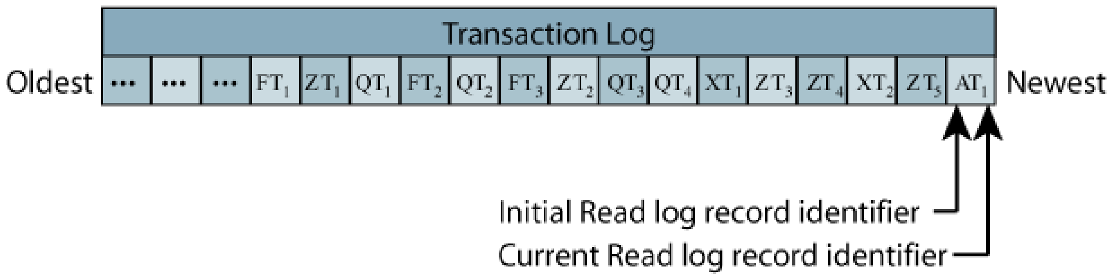
At this point, the Initial Read log record identifier associated with the bookmark is read from the database and cached in the XLA handle (`ttXlaHandle_h`). It designates the start position of the reader in the transaction log.

See ttLogHolds in *Oracle TimesTen In-Memory Database Reference*. That TimesTen built-in procedure returns information about transaction log holds.

How Bookmarks Work

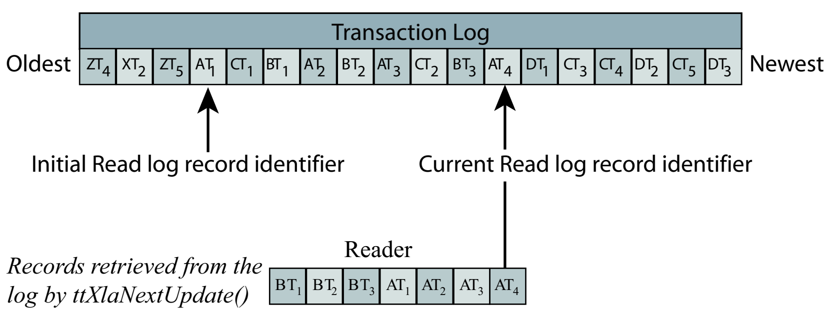
When an application first initializes XLA and obtains an XLA handle, its Current Read log record identifier and Initial Read log record identifier both point to the last record written to the database.

Figure 5-2 Log Record Indicator Positions Upon Initializing an XLA Handle



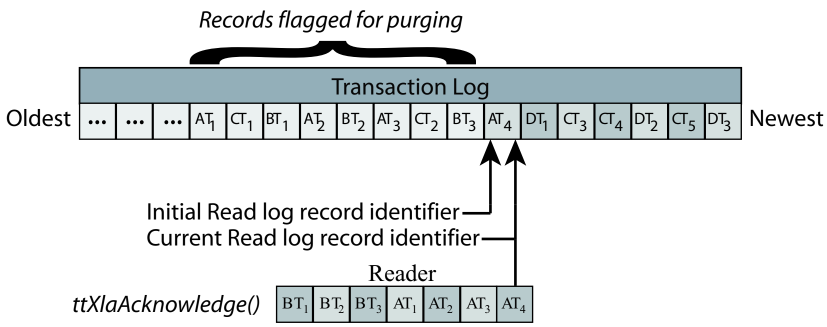
As described in Retrieving Update Records From the Transaction Log, use the ttXlaNextUpdate or ttXlaNextUpdateWait function to return a batch of records for committed transactions from the transaction log in the order in which they were committed. Each call to `ttXlaNextUpdate` resets the Current Read log record identifier of the bookmark to the last record read, as shown in Figure 5-3. The Current Read log record identifier marks the start position for the next call to `ttXlaNextUpdate`.

Figure 5-3 Records Retrieved by ttXlaNextUpdate



You can use the ttXlaGetLSN and ttXlaSetLSN functions to reread records, as described in Changing the Location of a Bookmark. However, calling the ttXlaAcknowledge function permanently resets the Initial Read log record identifier of the bookmark to its Current Read log record identifier, as shown in Figure 5-4. After you have called the `ttXlaAcknowledge` function to reset the Initial Read log record identifier, all previously read transaction records are flagged for purging by TimesTen. Once the Initial Read log record identifier is reset, you cannot use `ttXlaSetLSN` to go back and reread any of the previously read transactions.

Figure 5-4 ttXlaAcknowledge Resets Bookmark



> **Note:**
>
> A `ttXlaAcknowledge` call resets the bookmark even if there are no relevant update records to acknowledge. This may be useful in managing transaction log space, but should be balanced against the expense of the operation. Be aware that XLA purges transaction logs a file at a time. Refer to ttXlaAcknowledge for details on how the operation works.

The number of bookmarks created in a database is limited to 64. Each bookmark can be associated with only one active connection at a time. However, a bookmark over its lifetime may be associated with many connections. An application can open a connection, create a new bookmark, associate the bookmark with the connection, read a few records using the bookmark, disconnect from the database, reconnect to the database, create a new connection, associate this new connection with the bookmark, and continue reading transaction log records from where the old connection stopped.

Replicated Bookmarks

If you are using an active standby pair replication scheme, you have the option of using *replicated bookmarks* according to the *`options`* settings in your `ttXlaPersistOpen` calls.

See ttXlaPersistOpen.

For a replicated bookmark, operations on the bookmark are replicated to the standby database as appropriate. This results in more efficient recovery of your bookmark positions in the event of failover. Reading resumes from the stream of XLA records close to the point at which they left off before the switchover to the new active store. Without replicated bookmarks, reading must go through numerous duplicate records that were returned on the old active store.

To use replicated bookmarks, complete steps in this order:

1. Create the active standby pair replication scheme. (This is accomplished by the `create active standby pair` operation, or by the `ttCWAdmin -create` command in a Clusterware-managed environment.)
2. Create the bookmarks.
3. Subscribe the bookmarks.

4. Start the active standby pair, at which time duplication to the standby occurs and replication begins. (This is accomplished by the `ttRepAdmin -duplicate` command, or by the `ttCWAdmin -start` command in a Clusterware-managed environment.)

Be aware of the following usage notes:

- The position of the bookmark in the standby database is very close to that of the bookmark in the active database; however, because the replication of acknowledge operations is asynchronous, you may see a small window of duplicate updates in the event of a failover, depending on how often acknowledge operations are performed.
- You should close and reopen all bookmarks on a database after it changes from standby to active status, using the ttXlaClose and ttXlaPersistOpen functions. The state of a replicated bookmark on a standby database does change during XLA processing, as the replication agent automatically repositions bookmarks as appropriate on standby databases. If you attempt to use a bookmark that was open before the database changed to active status, you receive an error indicating that the state of the bookmark was reset and that it has been repositioned. While it is permissible to continue reading from the repositioned bookmark in this scenario, you can avoid the error by closing and reopening bookmarks.
- It is permissible to drop the active standby pair scheme while replicated bookmarks exist. The bookmarks of course cease to be replicated at that point, but are not deleted. If you subsequently re-enable the active standby pair scheme, these bookmarks are automatically added to the scheme.
- You cannot delete replicated bookmarks as long as the replication agent is running.
- You can only read and acknowledge a replicated bookmark in the active database. Each time you acknowledge a replicated bookmark, the acknowledge operation is asynchronously replicated to the standby database.

XLA Bookmarks and Transaction Log Holds

When XLA is in use, there is a hold on TimesTen transaction log files until the XLA bookmark advances.

The hold prevents transaction log files from being purged until XLA can confirm it no longer needs them. If a bookmark becomes stuck, which can occur if an XLA application terminates unexpectedly or disconnects without first deleting its bookmark or disabling change tracking, the log hold persists and there may be an excessive accumulation of transaction log files. This accumulation may result in file system space being filled.

See Monitoring Accumulation of Transaction Log Files in *Oracle TimesTen In-Memory Database Operations Guide*.

XLA Data Types

There is a data type mapping between internal SQL data types and XLA data types before release 7.0 and since release 7.0.

See Data Types in *Oracle TimesTen In-Memory Database SQL Reference*.

Table 5-1 XLA Data Type Mapping

Internal SQL Data Type	XLA Data Type
TT_CHAR	TTXLA_CHAR_TT

Table 5-1 (Cont.) XLA Data Type Mapping

Internal SQL Data Type	XLA Data Type
TT_VARCHAR	TTXLA_VARCHAR_TT
TT_NCHAR	TTXLA_NCHAR_TT
TT_NVARCHAR	TTXLA_NVARCHAR_TT
CHAR	TTXLA_CHAR
NCHAR	TTXLA_NCHAR
VARCHAR2	TTXLA_VARCHAR
NVARCHAR2	TTXLA_NVARCHAR
TT_TINYINT	TTXLA_TINYINT
TT_SMALLINT	TTXLA_SMALLINT
TT_INTEGER	TTXLA_INTEGER
TT_BIGINT	TTXLA_BIGINT
BINARY_FLOAT	TTXLA_BINARY_FLOAT
BINARY_DOUBLE	TTXLA_BINARY_DOUBLE
NUMBER	TTXLA_NUMBER
NUMBER(*p*,*s*)	TTXLA_NUMBER
FLOAT	TTXLA_NUMBER
TT_TIME	TTXLA_TIME
TT_DATE	TTXLA_DATE_TT
TT_TIMESTAMP	TTXLA_TIMESTAMP_TT
DATE	TTXLA_DATE
TIMESTAMP	TTXLA_TIMESTAMP
TT_BINARY	TTXLA_BINARY
TT_VARBINARY	TTXLA_VARBINARY
ROWID	TTXLA_ROWID
BLOB	TTXLA_BLOB
CLOB	TTXLA_CLOB
NCLOB	TTXLA_NCLOB

XLA offers functions to convert between internal SQL data types and external programmatic data types. For example, you can use ttXlaNumberToCString to convert NUMBER columns to character strings. TimesTen provides the following XLA data type conversion functions:

- ttXlaDateToODBCCType
- ttXlaDecimalToCString
- ttXlaNumberToCString
- ttXlaNumberToDouble

- ttXlaNumberToBigInt
- ttXlaNumberToInt
- ttXlaNumberToSmallInt
- ttXlaNumberToTinyInt
- ttXlaNumberToUInt
- ttXlaOraDateToODBCTimeStamp
- ttXlaOraTimeStampToODBCTimeStamp
- ttXlaRowidToCString
- ttXlaTimeToODBCCType
- ttXlaTimeStampToODBCCType

XLA System Privilege

An XLA user must have the XLA system privilege.

- Any XLA functionality, such as the following, requires the system privilege `XLA`:
  - Connecting to TimesTen (which also requires the `CREATE SESSION` privilege) as an XLA reader, such as by the `ttXlaPersistOpen` C function
  - Executing any other XLA-related TimesTen C functions, documented in XLA Reference
  - Executing any XLA-related TimesTen built-in procedures

    The procedures `ttXlaBookmarkCreate`, `ttXlaBookmarkDelete`, `ttXlaSubscribe`, and `ttXlaUnsubscribe` are documented in Built-In Procedures in *Oracle TimesTen In-Memory Database Reference*.
- A user with the `XLA` privilege has capabilities equivalent to the `SELECT ANY TABLE`, `SELECT ANY VIEW`, and `SELECT ANY SEQUENCE` system privileges, and can capture DDL statement records that occur in the database. Note that as a result, the user can obtain information about database objects that the user has not otherwise been granted access to.

XLA Limitations

This section lists TimesTen XLA limitations.

- XLA is available on all platforms supported by TimesTen. However, XLA does not support data transfer between different platforms.
- XLA support for LOBs is limited. See Specifying Which Tables to Monitor for Updates.
- XLA does not support applications linked with a generic driver manager library or linked directly with the client/server library. (XLA supports applications linked directly with the direct driver library or linked with the TimesTen driver manager for direct connections.)
- An XLA reader cannot subscribe to a table that uses in-memory column-based compression.
- For autorefresh cache groups, the change-tracking trigger on Oracle Database does not have column-level resolution. (To have that would be very expensive.) Therefore, the autorefresh feature updates all the columns in the row, and XLA can only report that all the columns have changed, even if data did not actually change in all columns.

About the XLA Sample Application

The TimesTen Classic Quick Start provides the `xlaSimple` sample application showing how to use many of the XLA functions described in this chapter.

See About TimesTen Quick Start and Sample Applications.

Most of this chapter, including the sample code shown in Writing an XLA Event-Handler Application starting immediately below, is based on the `xlaSimple` application. For this application, a table `MYDATA` is created in the `APPUSER` schema. While you are logged in as `APPUSER`, you make updates to the table. While you are logged in as `XLAUSER`, the `xlaSimple` application reports on the updates.

To run the application, execute `xlaSimple` at one command prompt. You are prompted for the password of `XLAUSER` (determined when the sample database is created). Start `ttIsql` at a separate command prompt, connecting to the TimesTen sample database as `APPUSER`. You are prompted for the password of `APPUSER` (also determined when the sample database is created).

At the `ttIsql` command prompt you can enter DML statements to alter the table. Then you can view the XLA output in the `xlaSimple` window.

Writing an XLA Event-Handler Application

There are general procedures for writing an XLA application that detects and reports changes to selected tables in a database.

This section describes these general procedures with the possible exception of Inspecting Column Data, the procedures described in this section are applicable to most XLA applications.

The following procedures are described:

- Obtaining a Database Connection Handle
- Initializing XLA and Obtaining an XLA Handle
- Specifying Which Tables to Monitor for Updates
- Retrieving Update Records From the Transaction Log
- Inspecting Record Headers and Locating Row Addresses
- Inspecting Column Data
- XLA Error Handling
- Dropping a Table That Has an XLA Bookmark
- Deleting Bookmarks
- Terminating an XLA Application

The example code in this section is based on the `xlaSimple` sample application.

XLA functions mentioned here are documented in XLA Reference.

Tip:

In addition to files noted in TimesTen Include Files, an XLA application must include `tt_xla.h`.

Note:

To simplify the code examples, routine error checking code for each function call has been omitted. See XLA Error Handling.

Obtaining a Database Connection Handle

As with every ODBC application, an XLA application must initialize ODBC, obtain an environment handle (`henv`), and obtain a connection handle (`hdbc`) to communicate with the specific database.

This section shows how to obtain a connection handle.

Initialize the environment and connection handles:

```
SQLHENV henv = SQL_NULL_HENV;
SQLHDBC hdbc = SQL_NULL_HDBC;
```

Pass the address of `henv` to the `SQLAllocEnv` ODBC function to allocate an environment handle:

```
rc = SQLAllocEnv(&henv);
```

Pass the address of `hdbc` to the `SQLAllocConnect` ODBC function to allocate a connection handle for the database:

```
rc = SQLAllocConnect(henv, &hdbc);
```

Call the `SQLDriverConnect` ODBC function to connect to the database specified by the connection string (`connStr`), which in this example is passed from the command line:

```
static char connstr[CONN_STR_LEN];
...
rc = SQLDriverConnect(hdbc, NULL, (SQLCHAR*)connstr, SQL_NTS, NULL, 0,
                      NULL, SQL_DRIVER_COMPLETE);
```

Note:

After an ODBC connection handle is opened for use by an XLA application, the ODBC handle cannot be used for ODBC operations until the corresponding XLA handle is closed by calling ttXlaClose.

Call the `SQLSetConnectOption` ODBC function to turn autocommit off:

```
rc = SQLSetConnectOption(hdbc, SQL_AUTOCOMMIT, SQL_AUTOCOMMIT_OFF);
```

Initializing XLA and Obtaining an XLA Handle

After initializing ODBC and obtaining an environment and connection handle, you can initialize XLA and obtain an XLA handle to access the transaction log.

See Obtaining a Database Connection Handle.

Create only one XLA handle per ODBC connection. If your application uses multiple XLA reader threads (each connected to its own XLA bookmark), create a separate XLA handle and ODBC connection for each thread.

This section describes how to initialize XLA. Before initializing XLA, initialize a bookmark. Then initialize an XLA handle as type `ttXlaHandle_h`:

```
unsigned char bookmarkName [32];
...
strcpy((char*)bookmarkName, "xlaSimple");
...
ttXlaHandle_h xla_handle = NULL;
```

Pass `bookmarkName` and the address of `xla_handle` to the ttXlaPersistOpen function to obtain an XLA handle:

```
rc = ttXlaPersistOpen(hdbc, bookmarkName, XLACREAT, &xla_handle);
```

The `XLACREAT` option is used to create a new non-replicated bookmark. Alternatively, use the `XLAREPL` option to create a replicated bookmark. In either case, the operation fails if the bookmark already exists.

To use a bookmark that already exists, call `ttXlaPersistOpen` with the `XLAREUSE` option, as shown in the following example.

```
#include <tt_errCode.h>      /* TimesTen Native Error codes */
...
    if ( native_error == 907 ) { /* tt_ErrKeyExists */
      rc = ttXlaPersistOpen(hdbc, bookmarkName, XLAREUSE, &xla_handle);
    ...
    }
```

If `ttXlaPersistOpen` is given invalid parameters, or the application was unable to allocate memory for the handle, the return code is `SQL_INVALID_HANDLE`. In this situation, ttXlaError cannot be used to detect this or any further errors.

If `ttXlaPersistOpen` fails but still creates a handle, the handle must be closed to prevent memory leaks.

Specifying Which Tables to Monitor for Updates

After initializing XLA and obtaining an XLA handle, you can specify which tables or materialized views you want to monitor for update events.

See Initializing XLA and Obtaining an XLA Handle.

You can determine which tables a bookmark is subscribed to by querying the `SYS.XLASUBSCRIPTIONS` table. You can also use `SYS.XLASUBSCRIPTIONS` to determine which bookmarks have subscribed to a specific table.

The ttXlaNextUpdate and ttXlaNextUpdateWait functions retrieve XLA records associated with DDL events. DDL XLA records are available to any XLA bookmark. DDL events include `CREATAB`, `DROPTAB`, `CREAIND`, `DROPIND`, `CREATVIEW`, `DROPVIEW`, `CREATSEQ`, `DROPSEQ`, `CREATSYN`, `DROPSYN`, `ADDCOLS`, `DRPCOLS`, and `TRUNCATE` transactions. See ttXlaUpdateDesc_t.

The ttXlaTableStatus function subscribes the current bookmark to updates to the specified table. Or it determines whether the current bookmark is already monitoring DML records associated with the table.

Call the ttXlaTableByName function to obtain both the system and user identifiers for a named table or materialized view. Then call the `ttXlaTableStatus` function to enable XLA to monitor changes to the table or materialized view.

> **Note:**
>
> LOB support in XLA is limited, as follows:
>
> - You can subscribe to tables containing LOB columns, but information about the LOB value itself is unavailable.
> - ttXlaGetColumnInfo returns information about LOB columns.
> - Columns containing LOBs are reported as empty (zero length) or null (if the value is actually `NULL`). In this way, you can tell the difference between a null column and a non-null column.

This example tracks changes to the `MYDATA` table.

```
#define TABLE_OWNER "APPUSER"
#define TABLE_NAME "MYDATA"
...
SQLUBIGINT SYSTEM_TABLE_ID = 0;
...
SQLUBIGINT userID;

rc = ttXlaTableByName(xla_handle, TABLE_OWNER, TABLE_NAME,
                      &SYSTEM_TABLE_ID, &userID);
```

When you have the table identifiers, you can use the `ttXlaTableStatus` function to enable XLA update tracking to detect changes to the `MYDATA` table. Setting the `newstatus` parameter to a nonzero value results in XLA tracking changes made to the specified table.

```
SQLINTEGER oldstatus;
SQLINTEGER newstatus = 1;
...
rc = ttXlaTableStatus(xla_handle, SYSTEM_TABLE_ID, 0,
                      &oldstatus, &newstatus);
```

The `oldstatus` parameter is output to indicate the status of the table at the time of the call.

At any time, you can use `ttXlaTableStatus` to return the current XLA status of a table by leaving `newstatus` null and returning only `oldstatus`. For example:

```
rc = ttXlaTableStatus(xla_handle, SYSTEM_TABLE_ID, 0,
                      &oldstatus, NULL);
...
if (oldstatus != 0)
```

```
     printf("XLA is currently tracking changes to table %s.%s\n",
             TABLE_OWNER, TABLE_NAME);
else
     printf("XLA is not tracking changes to table %s.%s\n",
             TABLE_OWNER, TABLE_NAME);
```

Retrieving Update Records From the Transaction Log

Once you have specified which tables to monitor for updates, you can call the `ttXlaNextUpdate` or `ttXlaNextUpdateWait` function to return a batch of records from the transaction log.

See ttXlaNextUpdate and ttXlaNextUpdateWait.

Only records for committed transactions are returned. They are returned in the order in which they were committed. You must periodically call the ttXlaAcknowledge function to acknowledge receipt of the transactions so that XLA can determine which records are no longer needed and can be purged from the transaction log. These functions impact the position of the application bookmark in the transaction log, as described in How Bookmarks Work. Also see ttLogHolds in *Oracle TimesTen In-Memory Database Reference*. That TimesTen built-in procedure returns information about transaction log holds.

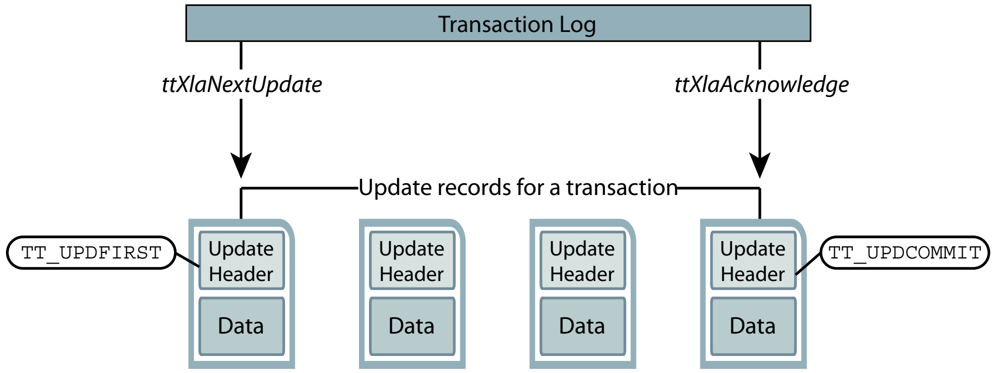
> **Note:**
>
> The `ttXlaAcknowledge` function is an expensive operation and should be used only as necessary.

Each update record in a transaction returned by `ttXlaNextUpdate` begins with an update header described by the ttXlaUpdateDesc_t structure. This update header contains a flag indicating if the record is the first in the transaction (`TT_UPDFIRST`) or the last commit record (`TT_UPDCOMMIT`). The update header also identifies the table affected by the update. Following the update header are zero to two rows of data that describe the update made to that table in the database.

Figure 5-5 that follows shows a call to `ttXlaNextUpdate` that returns a transaction consisting of four update records from the transaction log. Receipt of the returned transaction is acknowledged by calling `ttXlaAcknowledge`, which resets the bookmark.

> **Note:**
>
> This example is simplified for clarity. An actual XLA application would likely read records for multiple transactions before calling `ttXlaAcknowledge`.

Figure 5-5   Update Records



In this example, the `xlaSimple` application continues to monitor our table for updates until stopped by the user.

Before calling `ttXlaNextUpdateWait`, the example initializes a pointer to the buffer to hold the returned `ttXlaUpdateDesc_t` records (`arry`) and a variable to hold the actual number of returned records (`records`). Because the example calls `ttXlaNextUpdateWait`, it also specifies the number of seconds to wait (`FETCH_WAIT_SECS`) if no records are found in the transaction log buffer.

Next, call `ttXlaNextUpdateWait`, passing these values to obtain a batch of `ttXlaUpdateDesc_t` records in `arry`. Then process each record in `arry` by passing it to the `HandleChange()` function described in the example in Inspecting Record Headers and Locating Row Addresses. After all records are processed, call `ttXlaAcknowledge` to reset the bookmark position.

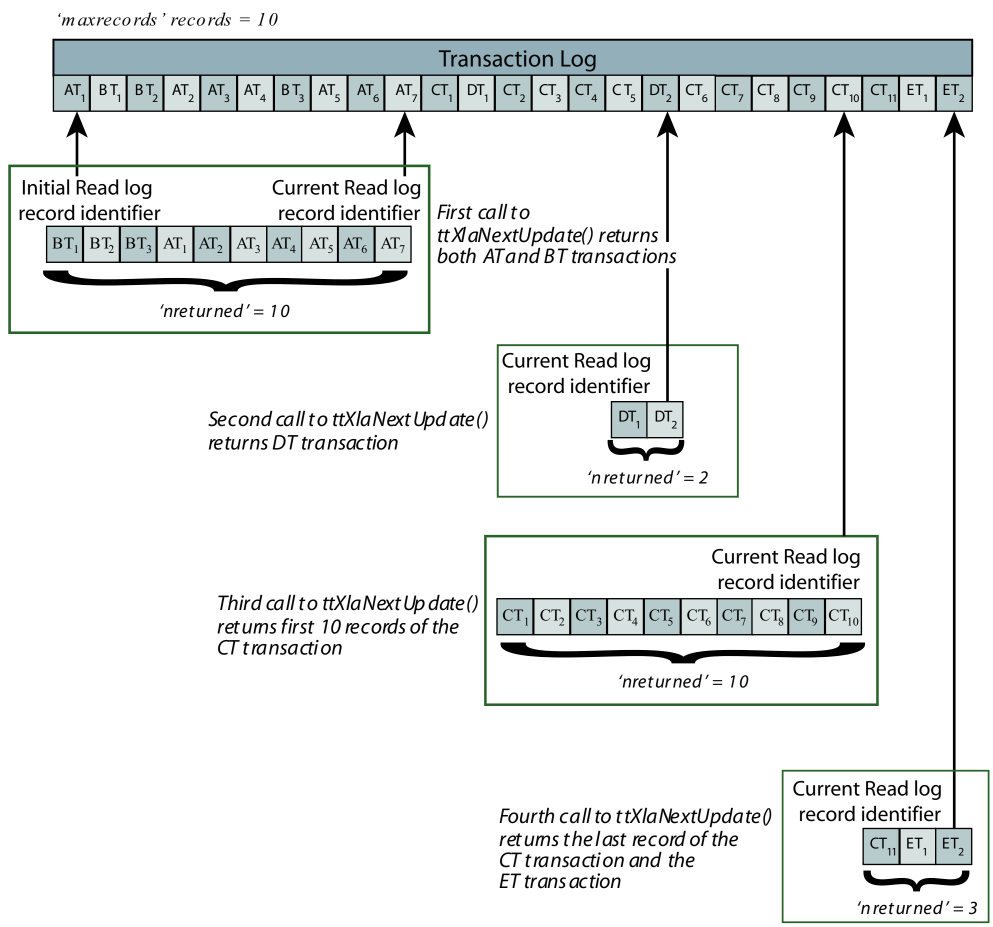
```
#define FETCH_WAIT_SECS 5
...
SQLINTEGER records;
ttXlaUpdateDesc_t** arry;
int j;

while (!StopRequested()) {

    /* Get a batch of update records */
    rc = ttXlaNextUpdateWait(xla_handle, &arry, 100,
                             &records, FETCH_WAIT_SECS);
    if (rc != SQL_SUCCESS {
      /* See XLA Error Handling */
    }

    /* Process the records */
    for(j=0; j < records; j++){
      ttXlaUpdateDesc_t* p;
      p = arry[j];
      HandleChange(p); /* Described in the next section */
    }

    /* After each batch, Acknowledge updates to reset bookmark.*/
    rc = ttXlaAcknowledge(xla_handle);
    if (rc != SQL_SUCCESS {
      /* See XLA Error Handling */
```

```
    }
} /* end while !StopRequested() */
```

The actual number of records returned by `ttXlaNextUpdate` or `ttXlaNextUpdateWait`, as indicated by the *`nreturned`* output parameter of those functions, may be less than the value of the *`maxrecords`* parameter. Figure 5-6 shows an example where *`maxrecords`* is 10, the transaction log contains transaction `AT` that is made up of seven records, and transaction `BT` that is made up of three records. In this case, both transactions are returned in the same batch and both *`maxrecords`* and *`nreturned`* values are 10. However, the next three transactions in the log are `CT` with 11 records, `DT` with two records, and `ET` with two records. Because the commit record for the `DT` transaction appears before the `CT` commit record, the next call to `ttXlaNextUpdate` returns the two records for the `DT` transaction and the value of *`nreturned`* is 2. In the next call to `ttXlaNextUpdate`, XLA detects that the total records for the `CT` transaction exceeds *`maxrecords`*, so it returns the records for this transaction in two batches. The first batch contains the first 10 records for `CT` (*`nreturned`* = 10). The second batch contains the last `CT` record and the two records for the `ET` transaction, assuming no commit record for a transaction following `ET` is detected within the next seven records.

See ttXlaNextUpdate and ttXlaNextUpdateWait.

Figure 5-6 Records Retrieved When maxrecords=10



XLA reads records from either a memory buffer or transaction log files on the file system, as described in How XLA Reads Records From the Transaction Log. To minimize latency, records from the memory buffer are returned as soon as they are available, while records not in the buffer are returned only if the buffer is empty. This design enables XLA applications to see changes as soon as the changes are made and with minimal latency. The trade-off is that there may be times when fewer changes are returned than the number requested by the `ttXlaNextUpdate` or `ttXlaNextUpdateWait` *`maxrecords`* parameter.

> **Note:**
>
> For optimal throughput, XLA applications should make the "fetch" and "process record" procedures asynchronous. For example, you can create one thread to fetch and store the records and one or more other threads to process the stored records.

Inspecting Record Headers and Locating Row Addresses

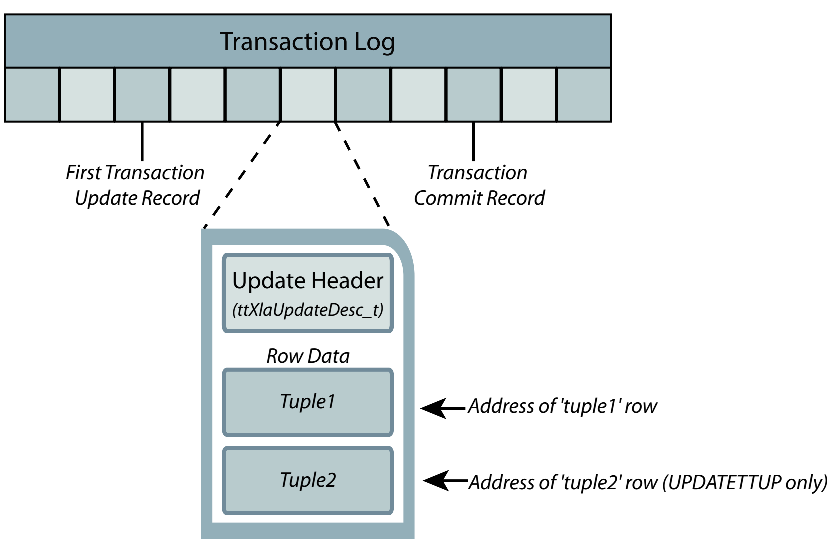
Now that there is an array of update records where the type of operation each record represents is known, the returned row data can be inspected.

Each record returned by the ttXlaNextUpdate or ttXlaNextUpdateWait function begins with an ttXlaUpdateDesc_t header that describes the following:

- The table on which the operation was performed
- Whether the record is the first or last (commit) record in the transaction
- The type of operation it represents
- The length of the returned row data, if any
- Which columns in the row were updated, if any

Figure 5-7 shows one of the update records in the transaction log.

Figure 5-7 Address of Row Data Returned in an XLA Update Record



The `ttXlaUpdateDesc_t` header has a fixed length and, depending on the type of operation, is followed by zero to two rows (or tuples) from the database. You can locate the address of the first returned row by obtaining the address of the `ttXlaUpdateDesc_t` header and adding it to `sizeof(ttXlaUpdateDesc_t)`:

```
tup1 = (void*) ((char*) ttXlaUpdateDesc_t + sizeof(ttXlaUpdateDesc_t));
```

This is shown in the example below.

The `ttXlaUpdateDesc_t ->`*`type`* field describes the type of SQL operation that generated the update. Transaction records of type `UPDATETTUP` describe `UPDATE` operations, so they return two rows to report the row data before and after the update. You can locate the address of the second returned row that holds the value after the update by adding the address of the first row in the record to its length:

```
if (ttXlaUpdateDesc_t->type == UPDATETUP) {
  tup2 = (void*) ((char*) tup1 + ttXlaUpdateDesc_t->tuple1);
}
```

This is also shown in the following example, which passes each record returned by the `ttXlaNextUpdateWait` function to a `HandleChange()` function, which determines whether the record is related to an `INSERT`, `UPDATE`, or `CREATE VIEW` operation. In this example, all other operations are ignored.

The `HandleChange()` function handles each type of SQL operation differently before calling the `PrintColValues()` function described in the example in Putting It All Together With a PrintColValues() Function.

```
void HandleChange(ttXlaUpdateDesc_t* xlaP)
{
  void*  tup1;
  void*  tup2;

  /* First confirm that the XLA update is for the table we care about. */
  if (xlaP->sysTableID != SYSTEM_TABLE_ID)
    return ;

  /* OK, it is for the table we are monitoring. */

  /* The last record in the ttXlaUpdateDesc_t record is the "tuple2"
   * field.  Immediately following this field is the first XLA record "row". */

  tup1 = (void*) ((char*) xlaP + sizeof(ttXlaUpdateDesc_t));

  switch(xlaP->type) {

  case INSERTTUP:
    printf("Inserted new row:\n");
    PrintColValues(tup1);
    break;

  case UPDATETUP:

    /* If this is an update ttXlaUpdateDesc_t, then following that is
     * the second XLA record "row".
     */

    tup2 = (void*) ((char*) tup1 + xlaP->tuple1);
    printf("Updated row:\n");
    PrintColValues(tup1);
```

```
      printf("To:\n");
      PrintColValues(tup2);
      break;

    case DELETETUP:
      printf("Deleted row:\n");
      PrintColValues(tup1);
      break;

    default:
      /* Ignore any XLA records that are not for inserts/update/delete SQL ops. */
      break;

  } /* switch (xlaP->type) */
}
```

Inspecting Column Data

There are methods for inspecting column data.

- Data Returned in an Update Record
- Obtaining Column Descriptions
- Reading Fixed-Length Column Data
- Reading NOT INLINE Variable-Length Column Data
- Null-Terminating Returned Strings
- Converting Complex Data Types
- Detecting Null Values
- Putting It All Together With a PrintColValues() Function

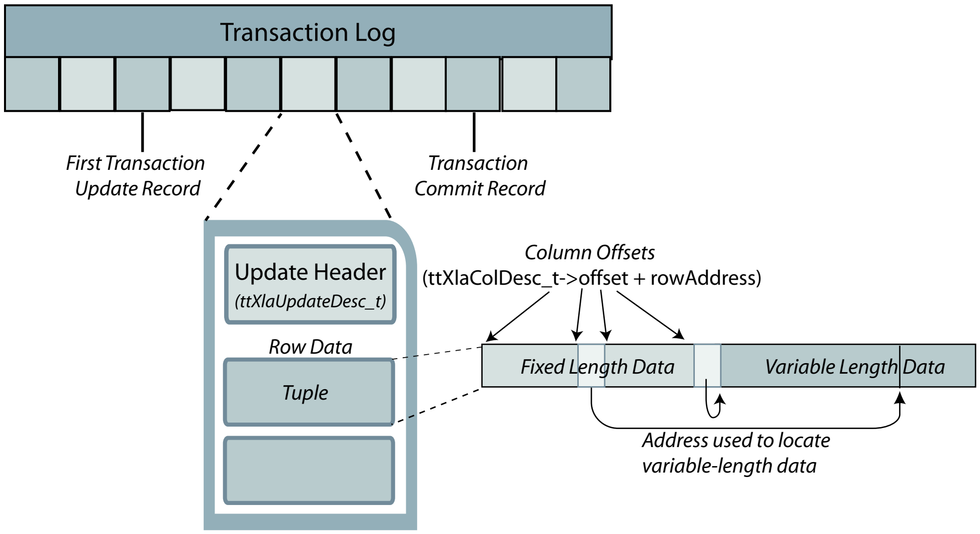
Data Returned in an Update Record

Zero to two rows of data may be returned in an update record after the `ttXlaUpdateDesc_t` structure.

See ttXlaUpdateDesc_t and Inspecting Record Headers and Locating Row Addresses.

For each row, the first portion of the data is the fixed-length data, which is followed by any variable-length data, as shown in Figure 5-8.

Figure 5-8 Column Offsets in a Row Returned in an XLA Update Record



Obtaining Column Descriptions

To read the column values from the returned row, you must first know the offset of each column in that row.

The column offsets and other column metadata can be obtained for a particular table by calling the ttXlaGetColumnInfo function, which returns a separate ttXlaColDesc_t structure for each column in the table. You should call the `ttXlaGetColumnInfo` function as part of your initialization procedure. This call was omitted from the discussion in Initializing XLA and Obtaining an XLA Handle for simplicity.

When calling `ttXlaGetColumnInfo`, specify a *`colinfo`* parameter to create a pointer to a buffer to hold the list of returned `ttXlaColDesc_t` structures. Use the *`maxcols`* parameter to define the size of the buffer.

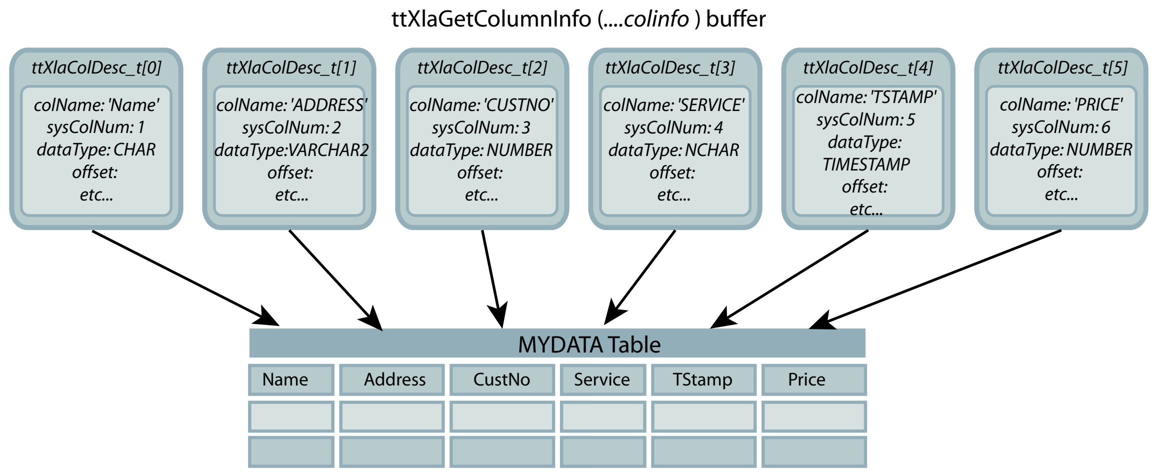
The sample code from the `xlaSimple` application below guesses the maximum number of returned columns (`MAX_XLA_COLUMNS`), which sets the size of the buffer `xla_column_defs` to hold the returned `ttXlaColDesc_t` structures. An alternative and more precise way to set the *`maxcols`* parameter would be to call the ttXlaGetTableInfo function and use the value returned in `ttXlaColDesc_t ->`*`columns`*.

```
#define MAX_XLA_COLUMNS 128
...
SQLINTEGER ncols;
...
ttXlaColDesc_t xla_column_defs[MAX_XLA_COLUMNS];
...
rc = ttXlaGetColumnInfo(xla_handle, SYSTEM_TABLE_ID, userID,
             xla_column_defs, MAX_XLA_COLUMNS, &ncols);
  if (rc != SQL_SUCCESS {
    /* See "XLA Error Handling" */
}
```

As shown in Figure 5-9, the `ttXlaGetColumnInfo` function produces the following output:

- A list, `xla_column_defs`, of `ttXlaColDesc_t` structures into the buffer pointed to by the `ttXlaGetColumnInfo` *`colinfo`* parameter
- An *`nreturned`* value, `ncols`, that holds the actual number of columns returned in the `xla_column_defs` buffer

Figure 5-9 ttXlaColDesc_t structures Returned by ttXlaGetColumnInfo



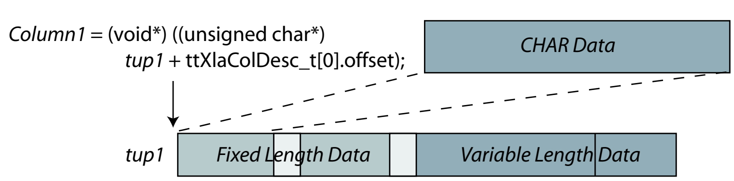
Each `ttXlaColDesc_t` structure returned by `ttXlaGetColumnInfo` has an offset value that describes the offset location of that column. How you use this offset value to read the column data depends on whether the column contains fixed-length data (such as `CHAR`, `NCHAR`, `INTEGER`, `BINARY`, `DOUBLE`, `FLOAT`, `DATE`, `TIME`, `TIMESTAMP`, and so on) or variable-length data (such as `VARCHAR`, `NVARCHAR`, or `VARBINARY`).

Reading Fixed-Length Column Data

For fixed-length column data, the address of a column is the offset value in the `ttXlaColDesc_t` structure, plus the address of the row.

See ttXlaColDesc_t.

Figure 5-10 Locating Fixed-Length Data in a Row



See the example in Putting It All Together With a PrintColValues() Function for a complete working example of computations such as those shown here.

The first column in the `MYDATA` table is of type `CHAR`. If you use the address of the `tup1` row obtained earlier in the `HandleChange()` function (in the example in Inspecting Record Headers and Locating Row Addresses) and the offset from the first `ttXlaColDesc_t` structure returned by the ttXlaGetColumnInfo function (in the example in Obtaining Column Descriptions), you can obtain the value of the first column with computations such as the following:

```
char*  Column1;

Column1 = ((unsigned char*) tup1 + xla_column_defs[0].offset);
```

The third column in the `MYDATA` table is of type `INTEGER`, so you can use the offset from the third `ttXlaColDesc_t` structure to locate the value and recast it as an integer using computations such as the following. The data is guaranteed to be aligned properly.

```
int Column3;

Column3 = *((int*) ((unsigned char*) tup +
           xla_column_defs[2].offset));
```

The fourth column in the `MYDATA` table is of type `NCHAR`, so you can use the offset from the fourth `ttXlaColDesc_t` structure to locate the value and recast it as a `SQLWCHAR` type, with computations such as the following:

```
SQLWCHAR*  Column4;

Column4 = (SQLWCHAR*) ((unsigned char*) tup +
                      xla_column_defs[3].offset);
```

Unlike the column values obtained in the above examples, `Column4` points to an array of two-byte Unicode characters. You must iterate through each element in this array to obtain the string, as shown for the `SQL_WCHAR` case in the example in Putting It All Together With a PrintColValues() Function.

Other fixed-length data types can be cast to their corresponding C types. Complex fixed-length data types, such as `DATE`, `TIME`, and `DECIMAL` values, are stored in an internal TimesTen format, but can be converted by applications to their corresponding ODBC C value using the XLA conversion functions, as described in Converting Complex Data Types.

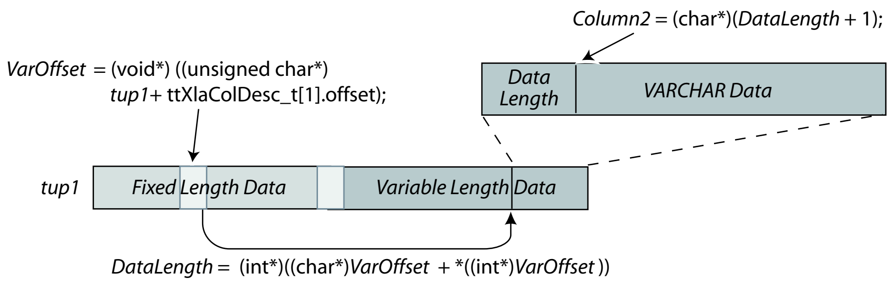
> **Note:**
>
> Strings returned by XLA are not null-terminated. See Null-Terminating Returned Strings.

Reading NOT INLINE Variable-Length Column Data

For `NOT INLINE` variable-length data (`VARCHAR`, `NVARCHAR`, and `VARBINARY`), the data located at `ttXlaColDesc_t ->`*`offset`* is a four-byte offset value that points to the location of the data in the variable-length portion of the returned row.

By adding the offset address to the offset value, you can obtain the address of the column data in the variable-length portion of the row. The first eight bytes at this location is the length of the data, followed by the actual data. For variable-length data, the `ttXlaColDesc_t ->`*`size`* value is the maximum allowable column size. Figure 5-11 shows how to locate `NOT INLINE` variable-length data in a row.

Figure 5-11 Locating NOT INLINE Variable-Length Data in a Row



See the example in Putting It All Together With a PrintColValues() Function for a complete working example of computations such as those shown here.

Continuing with our example, the second column in the returned row (`tup1`) is of type `VARCHAR`. To locate the variable-length data in the row, first locate the value at the column's `ttXlaColDesc_t ->offset` in the fixed-length portion of the row, as shown in Figure 5-11 above. The value at this address is the four-byte offset of the data in the variable-length portion of the row (`VarOffset`). Next, obtain a pointer to the beginning of the variable-length column data (`DataLength`) by adding the `VarOffset` offset value to the address of `VarOffset`. The first eight bytes at the `DataLength` location is the length of the data. The next byte after `DataLength` is the beginning of the actual data (`Column2`).

```
void*  VarOffset; /* offset of data */
long*  DataLength; /* length of data */
char*  Column2; /* pointer to data */

VarOffset = (void*) ((unsigned char*) tup1 +
             xla_column_defs[1].offset);
/*
 * If column is out-of-line, pColVal points to an offset
 * else column is inline so pColVal points directly to the string length.
 */

if (xla_column_defs[1].flags & TT_COLOUTOFLINE)
   DataLength = (long*)((char*)VarOffset + *((int*)VarOffset));
else
   DataLength = (long*)VarOffset;
Column2 = (char*)(DataLength+1);
```

`VARBINARY` types are handled in a manner similar to `VARCHAR` types. If `Column2` were an `NVARCHAR` type, you could initialize it as a `SQLWCHAR`, get the value as shown in the above `VARCHAR` case, then iterate through the `Column2` array, as shown for the `NCHAR` value, `CharBuf`, in Putting It All Together With a PrintColValues() Function.

> **Note:**
>
> In the example, `DataLength` is type `long`, which is a 64-bit (eight-byte) type on UNIX-based 64-bit systems. On Windows 64-bit systems, where `long` is a four-byte type, the eight-byte type `__int64` would be used instead.

Null-Terminating Returned Strings

Strings returned from record row data are not terminated with a null character. You can null-terminate a string by copying it into a buffer and adding a null character, '\0', after the last character in the string.

The procedures for null-terminating fixed-length and variable-length strings are slightly different. The examples that follow show the processes for null-terminating fixed-length strings, null-terminating variable-length strings of a known size, and null-terminating variable-length strings of an unknown size.

See the example inPutting It All Together With a PrintColValues() Function for a complete working example of computations such as those shown here.

To null-terminate the fixed-length CHAR(10) Column1 string returned in the example in "Reading Fixed-Length Column Data", establish a buffer large enough to hold the string plus null character. Next, obtain the size of the string from ttXlaColDesc_t ->*size*, copy the string into the buffer, and null-terminate the end of the string, using computations such as the following. You can now use the contents of the buffer. In this example, the string is printed:

```
char buffer[10+1];
int size;

size = xla_column_defs[0].size;
memcpy(buffer, Column1, size);
buffer[size] = '\0';

printf(" Row %s is %s\n", ((unsigned char*) xla_column_defs[0].colName), buffer);
```

Null-terminating a variable-length string is similar to the procedure for fixed-length strings, only the size of the string is the value located at the beginning of the variable-length data offset, as described in Reading NOT INLINE Variable-Length Column Data.

If the Column2 string obtained in the example in Reading NOT INLINE Variable-Length Column Data is a VARCHAR(32), establish a buffer large enough to hold the string plus null character. Use the value located at the DataLength offset to determine the size of the string, using computations such as the following:

```
char buffer[32+1];

memcpy(buffer, Column2, *DataLength);
buffer[*DataLength] = '\0';

printf(" Row %s is %s\n", ((unsigned char*) xla_column_defs[1].colName), buffer);
```

If you are writing general purpose code to read all data types, you cannot make any assumptions about the size of a returned string. For strings of an unknown size, statically allocate a buffer large enough to hold the majority of returned strings. If a returned string is larger than the buffer, dynamically allocate the correct size buffer, as shown in the example immediately below.

If the Column2 string obtained in the example in Reading NOT INLINE Variable-Length Column Data is of an unknown size, you might statically allocate a buffer large enough to hold a string of up to 10,000 characters. Then check that the DataLength value obtained at the beginning of the variable-length data offset is less than the size of the buffer. If the string is larger than the buffer, use malloc() to dynamically allocate the buffer to the correct size.

```
#define STACKBUFSIZE 10000
char VarStackBuf[STACKBUFSIZE];
char*  buffer;

buffer = (*DataLength+1 <= STACKBUFSIZE) ? VarStackBuf :
           malloc(*DataLength+1);

memcpy(buffer,Column2,*DataLength);
buffer[*DataLength] = '\0';

printf(" Row %s is %s\n", ((unsigned char*) xla_column_defs[1].colName), buffer);
if (buffer != VarStackBuf) /* buffer was allocated */
        free(buffer);
```

Converting Complex Data Types

There are methods to convert complex data types.

See the example in Putting It All Together With a PrintColValues() Function for a complete working example of computations such as those shown here.

Values for complex data types such as TT_DATE and TT_TIME are stored in an internal TimesTen format that can be converted into corresponding ODBC C types using XLA type conversion functions described in XLA Data Type Conversion Functions.

If you use the address of the tup1 row obtained earlier in the HandleChange() function (see the example in Inspecting Record Headers and Locating Row Addresses) and the offset from the fifth ttXlaColDesc_t structure returned by the ttXlaGetColumnInfo function (see the example in Obtaining Column Descriptions), you can locate a column value of type TIMESTAMP. Use the ttXlaTimeStampToODBCCType function to convert the column data from TimesTen format and store the converted time value in an ODBC TIMESTAMP_STRUCT. You could use code such as the following to print the values:

```
void*  Column5;
TIMESTAMP_STRUCT timestamp;

Column5 = (void*) ((unsigned char*) tup1 +
                  xla_column_defs[4].offset);

rc = ttXlaTimeStampToODBCCType(Column5, &timestamp);
if (rc != SQL_SUCCESS) {
  /* See XLA Error Handling */
}
printf(" %s: %04d-%02d-%02d %02d:%02d:%02d.%06d\n",
      ((unsigned char*) xla_column_defs[i].colName),
        timestamp.year,timestamp.month, timestamp.day,
        timestamp.hour,timestamp.minute,timestamp.second,
        timestamp.fraction);
```

If you use the address of the tup1 row obtained earlier in the HandleChange() function and the offset from the sixth ttXlaColDesc_t structure returned by the ttXlaGetColumnInfo function, you can locate a column value of type DECIMAL. Use the ttXlaDecimalToCString function to convert the column data from TimesTen decimal format to a string. You could use code such as the following to print the values.

```
char decimalData[50];

Column6 = (float*) ((unsigned char*) tup +
           xla_column_defs[5].offset);
precision = (short) (xla_column_defs[5].precision);
scale = (short) (xla_column_defs[5].scale);

rc = ttXlaDecimalToCString(Column6, (char*)&decimalData,
                           precision, scale);
if (rc != SQL_SUCCESS) {
  /* See XLA Error Handling */
}

printf(" %s: %s\n", ((unsigned char*) xla_column_defs[5].colName), decimalData);
```

Detecting Null Values

For nullable table columns, `ttXlaColDesc_t ->nullOffset` points to the column's null byte in the record. This field is 0 (zero) if the column is not nullable, or greater than 0 if the column can be null.

For nullable columns (`ttXlaColDesc_t ->nullOffset` > 0), to determine if the column is null, add the null offset to the address of `ttXlaUpdate_t*` and check the (`unsigned char`) byte there to see if it is 1 (`NULL`) or 0 (`NOT NULL`).

Check whether `Column6` is null as follows:

```
if (xla_column_defs[5].nullOffset != 0) {
  if (*((unsigned char*) tup +
     xla_column_defs[5].nullOffset) == 1) {
         printf("Column6 is NULL\n");
  }
}
```

XLA Data Type Conversion Functions

This section lists XLA data type conversion functions that convert from internal TimesTen formats to ODBC C types.

These conversion functions can be used on row data in the ttXlaUpdateDesc_t types: `UPDATETUP`, `INSERTTUP` and `DELETETUP`.

Table 5-2 XLA Data Type Conversion Functions

Function	Converts
ttXlaDateToODBCCType	Internal `TT_DATE` value to an ODBC C value
ttXlaTimeToODBCCType	Internal `TT_TIME` value to an ODBC C value
ttXlaTimeStampToODBCCType	Internal `TT_TIMESTAMP` value to an ODBC C value
ttXlaDecimalToCString	Internal `TTXLA_DECIMAL_TT` value to a string value
ttXlaDateToODBCCType	Internal `TTXLA_DATE_TT` value to an ODBC C value

Table 5-2 (Cont.) XLA Data Type Conversion Functions

Function	Converts
ttXlaDecimalToCString	Internal TTXLA_DECIMAL_TT value to a character string
ttXlaNumberToBigInt	Internal TTXLA_NUMBER value to a TT_BIGINT value
ttXlaNumberToCString	Internal TTXLA_NUMBER value to a character string
ttXlaNumberToDouble	Internal TTXLA_NUMBER value to a long floating point number value
ttXlaNumberToInt	Internal TTXLA_NUMBER value to an integer
ttXlaNumberToSmallInt	Internal TTXLA_NUMBER value to a TT_SMALLINT value
ttXlaNumberToTinyInt	Internal TTXLA_NUMBER value to a TT_TINYINT value
ttXlaNumberToUInt	Internal TTXLA_NUMBER value to an unsigned integer
ttXlaOraDateToODBCTimeStamp	Internal TTXLA_DATE value to an ODBC timestamp
ttXlaOraTimeStampToODBCTimeStamp	Internal TTXLA_TIMESTAMP value to an ODBC timestamp
ttXlaTimeToODBCCType	Internal TTXLA_TIME value to an ODBC C value
ttXlaTimeStampToODBCCType	Internal TTXLA_TIMESTAMP_TT value to an ODBC C value

Putting It All Together With a PrintColValues() Function

There is a function that checks the `ttXlaColDesc_t ->`*`dataType`* of each column to locate columns with a data type of `CHAR`, `NCHAR`, `INTEGER`, `TIMESTAMP`, `DECIMAL`, and `VARCHAR`, then prints the values.

This is just one possible approach. Another option, for example, would be to check the `ttXlaColDesc_t ->`*`ColName`* values to locate specific columns by name.

The `PrintColValues()` function handles `CHAR` and `VARCHAR` strings up to 50 bytes in length. `NCHAR` characters must belong to the ASCII character set.

The function in this example first checks `ttXlaColDesc_t ->`*`nullOffset`* to see if the column is null. Next it checks the `ttXlaColDesc_t ->`*`dataType`* field to determine the data type for the column. For simple fixed-length data (`CHAR`, `NCHAR`, and `INTEGER`), it casts the value located at `ttXlaColDesc_t ->`*`offset`* to the appropriate C type. The complex data types, `TIMESTAMP` and `DECIMAL`, are converted from their TimesTen formats to ODBC C values using the ttXlaTimeStampToODBCCType and ttXlaDecimalToCString functions.

For variable-length data (`VARCHAR`), the function locates the data in the variable-length portion of the row, as described in XLA Error Handling.

```
void PrintColValues(void* tup)
{

  SQLRETURN rc ;
  SQLINTEGER native_error;

  void* pColVal;
  char buffer[50+1]; /* No strings over 50 bytes */
  int i;

  for (i = 0; i < ncols; i++)
  {

    if (xla_column_defs[i].nullOffset != 0) {  /* See if column is NULL */
      /* this means col could be NULL */
      if (*((unsigned char*) tup + xla_column_defs[i].nullOffset) == 1) {
        /* this means that value is SQL NULL */
        printf("  %s: NULL\n",
               ((unsigned char*) xla_column_defs[i].colName));
        continue; /* Skip rest and re-loop */
      }
    }

    /* Fixed-length data types: */
    /* For INTEGER, recast as int */

    if (xla_column_defs[i].dataType == TTXLA_INTEGER) {

      printf("  %s: %d\n",
             ((unsigned char*) xla_column_defs[i].colName),
             *((int*) ((unsigned char*) tup + xla_column_defs[i].offset)));
    }

    /* For CHAR, just get value and null-terminate string */

    else if (   xla_column_defs[i].dataType == TTXLA_CHAR_TT
             || xla_column_defs[i].dataType == TTXLA_CHAR) {

      pColVal = (void*) ((unsigned char*) tup + xla_column_defs[i].offset);

      memcpy(buffer, pColVal, xla_column_defs[i].size);
      buffer[xla_column_defs[i].size] = '\0';

      printf("  %s: %s\n", ((unsigned char*) xla_column_defs[i].colName),
buffer);
    }

    /* For NCHAR, recast as SQLWCHAR.
       NCHAR strings must be parsed one character at a time */

    else if (   xla_column_defs[i].dataType == TTXLA_NCHAR_TT
             || xla_column_defs[i].dataType == TTXLA_NCHAR ) {
      SQLUINTEGER j;
      SQLWCHAR* CharBuf;

      CharBuf = (SQLWCHAR*) ((unsigned char*) tup + xla_column_defs[i].offset);

      printf("  %s: ", ((unsigned char*) xla_column_defs[i].colName));

      for (j = 0; j < xla_column_defs[i].size / 2; j++)
      {
```

```
    printf("%c", CharBuf[j]);
  }
  printf("\n");
}
/* Variable-length data types:
   For VARCHAR, locate value at its variable-length offset and null-terminate.
   VARBINARY types are handled in a similar manner.
   For NVARCHARs, initialize 'var_data' as a SQLWCHAR, get the value as shown
   below, then iterate through 'var_len' as shown for NCHAR above */

else if (   xla_column_defs[i].dataType == TTXLA_VARCHAR
         || xla_column_defs[i].dataType == TTXLA_VARCHAR_TT) {

  long*  var_len;
  char* var_data;
  pColVal = (void*) ((unsigned char*) tup + xla_column_defs[i].offset);
  /*
   * If column is out-of-line, pColVal points to an offset
   * else column is inline so pColVal points directly to the string length.
   */
  if (xla_column_defs[i].flags & TT_COLOUTOFLINE)
    var_len = (long*)((char*)pColVal + *((int*)pColVal));
  else
    var_len = (long*)pColVal;

  var_data = (char*)(var_len+1);

  memcpy(buffer,var_data,*var_len);
  buffer[*var_len] = '\0';

  printf("  %s: %s\n", ((unsigned char*) xla_column_defs[i].colName), buffer);
}
/* Complex data types require conversion by the XLA conversion methods
   Read and convert a TimesTen TIMESTAMP value.
   DATE and TIME types are handled in a similar manner  */

else if (   xla_column_defs[i].dataType == TTXLA_TIMESTAMP
         || xla_column_defs[i].dataType == TTXLA_TIMESTAMP_TT) {

  TIMESTAMP_STRUCT timestamp;
  char* convFunc;

  pColVal = (void*) ((unsigned char*) tup + xla_column_defs[i].offset);

  if (xla_column_defs[i].dataType == TTXLA_TIMESTAMP_TT) {
    rc = ttXlaTimeStampToODBCCType(pColVal, &timestamp);
    convFunc="ttXlaTimeStampToODBCCType";
  }
  else {
    rc = ttXlaOraTimeStampToODBCTimeStamp(pColVal, &timestamp);
    convFunc="ttXlaOraTimeStampToODBCTimeStamp";
  }

  if (rc != SQL_SUCCESS) {
    handleXLAerror (rc, xla_handle, err_buf, &native_error);
    fprintf(stderr, "%s() returns an error <%d>: %s",
            convFunc, rc, err_buf);
    TerminateGracefully(1);
  }

  printf("  %s: %04d-%02d-%02d %02d:%02d:%02d.%06d\n",
```

```
                 ((unsigned char*) xla_column_defs[i].colName),
                 timestamp.year,timestamp.month, timestamp.day,
                 timestamp.hour,timestamp.minute,timestamp.second,
                 timestamp.fraction);
        }

        /* Read and convert a TimesTen DECIMAL value to a string. */

        else if (xla_column_defs[i].dataType == TTXLA_DECIMAL_TT) {

          char decimalData[50];
          short precision, scale;
          pColVal = (float*) ((unsigned char*) tup + xla_column_defs[i].offset);
          precision = (short) (xla_column_defs[i].precision);
          scale = (short) (xla_column_defs[i].scale);

          rc = ttXlaDecimalToCString(pColVal, (char*)&decimalData, precision, scale);
          if (rc != SQL_SUCCESS) {
            handleXLAerror (rc, xla_handle, err_buf, &native_error);
            fprintf(stderr, "ttXlaDecimalToCString() returns an error <%d>: %s",
                    rc, err_buf);
            TerminateGracefully(1);
          }

          printf("  %s: %s\n", ((unsigned char*) xla_column_defs[i].colName),
                 decimalData);
        }
        else if (xla_column_defs[i].dataType == TTXLA_NUMBER) {
          char numbuf[32];
          pColVal = (void*) ((unsigned char*) tup + xla_column_defs[i].offset);

          rc=ttXlaNumberToCString(xla_handle, pColVal, numbuf, sizeof(numbuf));
          if (rc != SQL_SUCCESS) {
            handleXLAerror (rc, xla_handle, err_buf, &native_error);
            fprintf(stderr, "ttXlaNumberToDouble() returns an error <%d>: %s",
                    rc, err_buf);
            TerminateGracefully(1);
          }
          printf("  %s: %s\n", ((unsigned char*) xla_column_defs[i].colName),
numbuf);
        }

  } /* End FOR loop */
}
```

> **Note:**
>
> - In the example, `var_len` is type `long`, which is a 64-bit (eight-byte) type on UNIX-based 64-bit systems. On Windows 64-bit systems, where `long` is a four-byte type, `__int64` would be used instead.
> - See Terminating an XLA Application for a sample `TerminateGracefully()` method.

XLA Error Handling

This section discusses XLA error handling.

- XLA Errors and Codes
- How to Handle XLA Errors

XLA Errors and Codes

This section documents XLA errors and their error codes.

Table 5-3 XLA Errors and Codes

Error	Code
`tt_ErrDbAllocFailed`	802 (transient)
`tt_ErrCondLockConflict`	6001 (transient)
`tt_ErrDeadlockVictim`	6002 (transient)
`tt_ErrTimeoutVictim`	6003 (transient)
`tt_ErrPermSpaceExhausted`	6220 (transient)
`tt_ErrTempSpaceExhausted`	6221 (transient)
`tt_ErrBadXlaRecord`	8024
`tt_ErrXlaBookmarkUsed`	8029
`tt_ErrXlaLsnBad`	8031
`tt_ErrXlaNoSQL`	8034
`tt_ErrXlaNoLogging`	8035
`tt_ErrXlaParameter`	8036
`tt_ErrXlaTableDiff`	8037
`tt_ErrXlaTableSystem`	8038
`tt_ErrXlaTupleMismatch`	8046
`tt_ErrXlaDedicatedConnection`	8047

How to Handle XLA Errors

Each time you call an ODBC or XLA function, you must check the return code for any errors. If the error is fatal, terminate the program.

See Terminating an XLA Application.

Depending on your application, you may be required to act on specific XLA errors, including those shown in Table 5-3 in the preceding section.

You can check an error using either its error code (error number) or `tt_Err` string. For the complete list of TimesTen error codes and error strings, see the *`timesten_home`*`/install/include/tt_errCode.h` file. For a description of each message, see List of Errors and Warnings in *Oracle TimesTen In-Memory Database Error Messages and SNMP Traps*.

If the return code from an XLA function is not `SQL_SUCCESS`, use the ttXlaError function to retrieve XLA-specific errors on the XLA handle.

Also see Checking for Errors.

The following example, after calling the XLA function ttXlaTableByName, checks to see if the return code is `SQL_SUCCESS`. If not, it calls an XLA error-handling function followed by a function to terminate the application. See Terminating an XLA Application.

```
rc = ttXlaTableByName(xla_handle, TABLE_OWNER, TABLE_NAME,
                      &SYSTEM_TABLE_ID, &userID);
if (rc != SQL_SUCCESS) {
  handleXLAerror (rc, xla_handle, err_buf, &native_error);
  fprintf(stderr,
    "ttXlaTableByName() returns an error <%d>: %s", rc, err_buf);
  TerminateGracefully(1);
}
```

Your XLA error-handling function should repeatedly call `ttXlaError` until all XLA errors are read from the error stack, proceeding until the return code from ttXlaError is `SQL_NO_DATA_FOUND`. If you must reread the errors, you can call the ttXlaErrorRestart function to reset the error stack pointer to the first error. (`SQL_NO_DATA_FOUND` is defined in `sqlext.h`, which is included by `timesten.h`.)

The error stack is cleared after a call to any XLA function other than `ttXlaError` or `ttXlaErrorRestart`.

Note:

In cases where ttXlaPersistOpen cannot create an XLA handle, it returns the error code `SQL_INVALID_HANDLE`. Because no XLA handle has been created, `ttXlaError` cannot be used to detect this error. `SQL_INVALID_HANDLE` is returned only in cases where no memory can be allocated or the parameters provided are invalid.

The following example shows `handleXLAerror()`, the error function for the `xlaSimple` application program.

```
void handleXLAerror(SQLRETURN rc, ttXlaHandle_h xlaHandle,
                    SQLCHAR* err_msg, SQLINTEGER* native_error)
{
  SQLINTEGER retLen;
  SQLINTEGER code;
  char* err_msg_ptr;

  /* initialize return codes */
  rc = SQL_ERROR;
  *native_error = -1;
  err_msg[0] = '\0';

  err_msg_ptr = (char*)err_msg;

  while (1)
  {
    int rc = ttXlaError(xlaHandle, &code, err_msg_ptr,
                        ERR_BUF_LEN - (err_msg_ptr - (char*)err_msg), &retLen);
```

```
        if (rc == SQL_NO_DATA_FOUND)
        {
          break;
        }
        if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO) {
          sprintf(err_msg_ptr,
                  "*** Error fetching error message via ttXlaError(); rc=<%d>.",rc) ;
          break;
        }
        rc = SQL_ERROR;
        *native_error = code ;
        /* append any other error messages */
        err_msg_ptr += retLen;
      }
    }
```

Dropping a Table That Has an XLA Bookmark

Before you can drop a table that is subscribed to by an XLA bookmark, you must unsubscribe the table from the bookmark. There are several ways to unsubscribe a table from a bookmark, depending on whether the application is connected to the bookmark.

If XLA applications are connected and using bookmarks that are tracking the table to be dropped, then perform the following tasks.

1. Each XLA application must call the ttXlaTableStatus function and set the *newstatus* parameter to 0. This unsubscribes the table from the XLA bookmark in use by the application.
2. Drop the table.

If XLA applications are not connected and using bookmarks associated with the table to be dropped, then perform the following tasks:

1. Query the SYS.XLASUBSCRIPTIONS system table to see which bookmarks have subscribed to the table you want to drop.
2. Use the ttXlaUnsubscribe built-in procedure to unsubscribe the table from each XLA bookmark with a subscription to the table.
3. Drop the table.

Deleting bookmarks also unsubscribes the table from the XLA bookmarks. See Deleting Bookmarks.

Deleting Bookmarks

You may want to delete bookmarks when you terminate an application or drop a table.

Use the ttXlaDeleteBookmark function to delete XLA bookmarks if the application is connected and using the bookmarks.

As described in About XLA Bookmarks, a bookmark may be reused by a new connection after its previous connection has closed. The new connection can resume reading from the transaction log from where the previous connection stopped. Note the following:

- If you delete the bookmark, subsequent checkpoint operations such as the ttCkpt or ttCkptBlocking built-in procedure free the file system space associated with any unread update records in the transaction log.

- If you do not delete the bookmark, when an XLA application connects and reuses the bookmark, all unread update records that have accumulated since the program terminated are read by the application. This is because the update records are persistent in the TimesTen transaction log. However, the danger is that these unread records can build up in the transaction log files and consume a lot of file system space.

> **Note:**
>
> - You cannot delete replicated bookmarks while the replication agent is running.
> - When you reuse a bookmark, you start with the Initial Read log record identifier in the transaction log file. To ensure that a connection that reuses a bookmark begins reading where the prior connection left off, the prior connection should call ttXlaAcknowledge to reset the bookmark position to the currently accessed record before disconnecting.
> - See ttLogHolds in *Oracle TimesTen In-Memory Database Reference*. That TimesTen built-in procedure returns information about transaction log holds.
> - Be aware that `ttCkpt` and `ttCkptBlocking` require `ADMIN` privilege. TimesTen built-in procedures and any required privileges are documented in Built-In Procedures in *Oracle TimesTen In-Memory Database Reference*.

The `InitHandler()` function in the `xlaSimple` application deletes the XLA bookmark upon exit, as shown in the following example.

```
if (deleteBookmark) {
    ttXlaDeleteBookmark(xla_handle);
    if (rc != SQL_SUCCESS) {
    /* See XLA Error Handling */
    }
    xla_handle = NULL; /* Deleting the bookmark has the */
                       /* effect of disconnecting from XLA. */
}
/* Close the XLA connection as described in the next section,
Terminating an XLA Application. */
```

If the application is not connected and using the XLA bookmark, you can delete the bookmark either of the following ways:

- Close the bookmark and call the `ttXlaBookmarkDelete` built-in procedure.
- Close the bookmark and use the `ttIsql` command `xladeletebookmark`.

Terminating an XLA Application

When your XLA application has finished reading from the transaction log, gracefully exit by rolling back uncommitted transactions and freeing all handles.

There are two approaches to this:

- Unsubscribe from all tables and materialized views, delete the XLA bookmark, and disconnect from the database.

Or:

- Disconnect from the database but keep the XLA bookmark in place. When you reconnect at a later time, you can resume reading records from the bookmark.

For the first approach, complete the following steps.

1. Call ttXlaTableStatus to unsubscribe from each table and materialized view, setting the *newstatus* parameter to 0.
2. Call `ttXlaDeleteBookmark` to delete the bookmark. See Deleting Bookmarks.
3. Call ttXlaClose to disconnect the XLA handle.
4. Call the ODBC function `SQLTransact` with the `SQL_ROLLBACK` setting to roll back any uncommitted transaction.
5. Call the ODBC function `SQLDisconnect` to disconnect from the TimesTen database.
6. Call the ODBC function `SQLFreeConnect` to free memory allocated for the ODBC connection handle.
7. Call the ODBC function `SQLFreeEnv` to free the ODBC environment handle.

For the second approach, maintaining the bookmark, skip the first two steps but complete the remaining steps.

Be aware that resources should be freed in reverse order of allocation. For example, the ODBC environment handle is allocated before the ODBC connection handle, so for cleanup free the connection handle before the environment handle.

This example shows `TerminateGracefully()`, the termination function in the `xlaSimple` application.

```
void TerminateGracefully(int status)
{

  SQLRETURN      rc;
  SQLINTEGER     native_error ;
  SQLINTEGER     oldstatus;
  SQLINTEGER     newstatus = 0;

  /* If the table has been subscribed to through XLA, unsubscribe it. */

  if (SYSTEM_TABLE_ID != 0) {
    rc = ttXlaTableStatus(xla_handle, SYSTEM_TABLE_ID, 0,
                          &oldstatus, &newstatus);
    if (rc != SQL_SUCCESS) {
      handleXLAerror (rc, xla_handle, err_buf, &native_error);
      fprintf(stderr, "Error when unsubscribing from "TABLE_OWNER"."TABLE_NAME
              " table <%d>: %s", rc, err_buf);
    }
    SYSTEM_TABLE_ID = 0;
  }

  /* Close the XLA connection. */

  if (xla_handle != NULL) {
    rc = ttXlaClose(xla_handle);
    if (rc != SQL_SUCCESS) {
```

```
        fprintf(stderr, "Error when disconnecting from XLA:<%d>", rc);
      }
      xla_handle = NULL;
    }

    if (hstmt != SQL_NULL_HSTMT) {
      rc = SQLFreeStmt(hstmt, SQL_DROP);
      if (rc != SQL_SUCCESS) {
        handleError(rc, henv, hdbc, hstmt, err_buf, &native_error);
        fprintf(stderr, "Error when freeing statement handle:\n%s\n", err_buf);
      }
      hstmt = SQL_NULL_HSTMT;
    }

    /* Disconnect from TimesTen entirely. */

    if (hdbc != SQL_NULL_HDBC) {
      rc = SQLTransact(henv, hdbc, SQL_ROLLBACK);
      if (rc != SQL_SUCCESS) {
        handleError(rc, henv, hdbc, hstmt, err_buf, &native_error);
        fprintf(stderr, "Error when rolling back transaction:\n%s\n", err_buf);
      }

      rc = SQLDisconnect(hdbc);
      if (rc != SQL_SUCCESS) {
        handleError(rc, henv, hdbc, hstmt, err_buf, &native_error);
        fprintf(stderr, "Error when disconnecting from TimesTen:\n%s\n", err_buf);
      }

      rc = SQLFreeConnect(hdbc);
      if (rc != SQL_SUCCESS) {
        handleError(rc, henv, hdbc, hstmt, err_buf, &native_error);
        fprintf(stderr, "Error when freeing connection handle:\n%s\n", err_buf);
      }
      hdbc = SQL_NULL_HDBC;
    }

    if (henv != SQL_NULL_HENV) {
      rc = SQLFreeEnv(henv);
      if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO) {
        handleError(rc, henv, hdbc, hstmt, err_buf, &native_error);
        fprintf(stderr, "Error when freeing environment handle:\n%s\n", err_buf);
      }
      henv = SQL_NULL_HENV;
    }
    exit(status);
  }
```

Using XLA as a Replication Mechanism

This section discusses using XLA as a substitute for TimesTen replication.

- About XLA as a Replication Mechanism
- Checking Table Compatibility Between Databases
- Replicating Updates Between Databases
- Handling Timeout and Deadlock Errors
- Checking for Update Conflicts

- Replicating Updates to a Non-TimesTen Database

XLA functions mentioned here are documented in XLA Reference.

About XLA as a Replication Mechanism

TimesTen replication is sufficient for most customer needs; however, it is also possible to use XLA functions to replicate updates from one database to another. Implementing your own replication scheme on top of XLA in this way is fairly complicated, but can be considered if TimesTen replication is not feasible for some reason. See Overview of TimesTen Replication in the *Oracle TimesTen In-Memory Database Replication Guide*

> **Note:**
>
> You cannot use XLA to replicate updates between different platforms.

To use XLA to replicate changes between databases, first use the ttXlaPersistOpen function to initialize the XLA handles, as described in Initializing XLA and Obtaining an XLA Handle.

After the XLA handles have been initialized for the databases, the sections that follow describe these parts of the process:

- Check table compatibility between databases.
- Replicate updates between databases.
- Handle timeout and deadlock errors.
- Check for update conflicts.

Throughout the discussion, the sending database is referred to as the master and the receiving database as the subscriber.

Checking Table Compatibility Between Databases

Before transferring update records from one database to the other, verify that the tables in the master and subscriber databases are compatible with one another.

- You can check the descriptions of a table and its columns by using the ttXlaTableByName, ttXlaGetTableInfo, and ttXlaGetColumnInfo functions.
- You can check the table and column versions of a specific XLA record by using the ttXlaVersionTableInfo and ttXlaVersionColumnInfo functions.

These approaches are described in the sections that follow:

- Checking Table and Column Descriptions
- Checking Table and Column Versions

Checking Table and Column Descriptions

Use the `ttXlaTableByName`, `ttXlaGetTableInfo`, and `ttXlaGetColumnInfo` functions to return `ttXlaTblDesc_t` and `ttXlaColDesc_t` descriptions for each table you want to replicate.

See ttXlaTblDesc_t, ttXlaColDesc_t, Specifying Which Tables to Monitor for Updates and Obtaining Column Descriptions.

You can then pass these descriptions to the ttXlaTableCheck function. The output parameter, `compat`, specifies whether the tables are compatible. A value of 1 indicates compatibility and 0 indicates non-compatibility. The following example demonstrates this.

```
SQLINTEGER compat;
ttXlaTblDesc_t table;
ttXlaColDesc_t columns[20];

rc = ttXlaTableCheck(xla_handle, &table, columns, &compat);
if (compat) {
    /* Go ahead and start replicating */
}
else {
    /* Not compatible or some other error occurred */
}
```

Checking Table and Column Versions

Use the `ttXlaVersionTableInfo` and `ttXlaVersionColumnInfo` functions to retrieve the table structure information of an update record at the time the record was generated.

See ttXlaVersionTableInfo and ttXlaVersionColumnInfo.

The following example verifies that the table associated with the *`pXlaRecord`* update record from the *`pCmd`* source is compatible with the *`hXlaTarget`* target.

```
BOOL CUTLCheckXlaTable (SCOMMAND* pCmd,
                        ttXlaHandle_h hXlaTarget,
                        const ttXlaUpdateDesc_t* pXlaRecord)
{
  /* locals */
  ttXlaTblVerDesc_t tblVerDescSource;
  ttXlaColDesc_t colDescSource [255];
  SQLINTEGER iColsReturned = 0;
  SQLINTEGER iCompatible = 0;
  SQLRETURN rc;

  /* only certain update record types should be checked */
  if (pXlaRecord->type == INSERTTUP ||
      pXlaRecord->type == UPDATETUP ||
      pXlaRecord->type == DELETETUP)
  {
     /* Get source table description associated with this record */
     /* from the time it was generated. */
     rc = ttXlaVersionTableInfo (pCmd->pCtx->con->hXla,
             (ttXlaUpdateDesc_t*) pXlaRecord, &tblVerDescSource);
```

```
         if (rc == SQL_SUCCESS)
         {
             /* Get the source column descriptors for this table */
             /* at the time the record was generated. */
             rc = ttXlaVersionColumnInfo (pCmd->pCtx->con->hXla,
                     (ttXlaUpdateDesc_t*) pXlaRecord,
                     colDescSource, 255, &iColsReturned);

             if (rc == SQL_SUCCESS)
             {
                 /* Check compatibility. */
                 rc = ttXlaTableCheck (hXlaTarget,
                         &tblVerDescSource.tblDesc, colDescSource,
                         &iCompatible);
             }
         }
      }
    }
```

Replicating Updates Between Databases

When you are ready to begin replication, use the `ttXlaNextUpdate` or `ttXlaNextUpdateWait` function to obtain batches of update records from the master database and `ttXlaApply` to write the records to the subscriber database.

See ttXlaNextUpdate, ttXlaNextUpdateWait, and ttXlaApply.

The following example shows this.

```
int j;
ttXlaHandle_h h;
SQLINTEGER records;
ttXlaUpdateDesc_t** arry;

  do {
    /* get up to 15 updates */
    rc = ttXlaNextUpdate(h,&arry,15,&records);
    if (rc != SQL_SUCCESS) {
      /* See XLA Error Handling */
    }

    /* print number of updates returned */
    printf("Records returned by ttXlaNextUpdate : %d\n",records);

    /* apply the received updates */
    for (j=0;j < records;j++) {
      ttXlaUpdateDesc_t* p;

      p = arry[j];
      rc = ttXlaApply(h, p, 0);
      if (rc != SQL_SUCCESS){
      /* See XLA Error Handling and */
      /* Handling Timeout and Deadlock Errors below */
      }
    }

    /* print number of updates applied */
    printf("Records applied successfully : %d\n",records);

  } while (records != 0);
```

> **Tip:**
>
> - To ensure that you are sending XLA updates between databases that have compatible versions of XLA records, use the ttXlaGetVersion and ttXlaVersionCompare functions on all databases.
> - If you are packaging data to be replicated across a network, or anywhere between processes not using the same memory space, you must ensure that the `ttXlaUpdateDesc_t` data structure is shipped in its entirely. Its length is indicated by `ttXlaUpdateDesc_t ->`*`header.length`*, where the *`header`* element is a `ttXlaNodeHdr_t` structure that in turn has a *`length`* element. Also see ttXlaUpdateDesc_t and ttXlaNodeHdr_t.

Handling Timeout and Deadlock Errors

The return code from `ttXlaApply` indicates whether the update was successful.

See ttXlaApply.

If the return code is not `SQL_SUCCESS`, then the update may have encountered a transient problem, such as a deadlock or timeout, or a persistent problem. You can use ttXlaError to check for errors, such as `tt_ErrDeadlockVictim` or `tt_ErrTimeoutVictim`. Recovery from transient errors is possible by rolling back the replicated transaction and reexecuting it. Other errors may be persistent, such as those for duplicate key violations or key not found. Such errors are likely to repeat if the transaction is reexecuted.

If `ttXlaApply` returns a timeout or deadlock error before applying the commit record (`ttXlaUpdateDesc_t ->`*`flags`*` = TT_UPDCOMMIT`) for a transaction to the subscriber database, you can do either of the following:

- Use ttXlaRollback to roll back the transaction.
- Use ttXlaCommit to commit the changes in the records that have been applied to the subscriber database.

To enable recovery from transient errors, you should keep track of transaction boundaries on the master database and store the records associated with the transaction currently being applied to the subscriber in a user buffer, so you can reapply them if necessary. The transaction boundaries can be found by checking the *`flags`* member of the ttXlaUpdateDesc_t structure. Consider the following example. If this condition is true, then the record was committed:

```
(pXlaRecords [iRecordIndex]->flags & TT_UPDCOMMIT)
```

If you encounter an error that requires you to roll back a transaction, call `ttXlaRollback` to roll back the records applied to the subscriber database. Then call `ttXlaApply` to reapply all the rolled back records stored in your buffer.

> **Note:**
>
> An alternative to buffering the transaction records in a user buffer is to call ttXlaGetLSN to get the transaction log record identifier of each commit record in the transaction log, as described in Changing the Location of a Bookmark. If you encounter an error that requires you to roll back a transaction, you can call ttXlaSetLSN to reset the bookmark to the beginning of the transaction in the transaction log and reapply the records. However, the extra overhead associated with the `ttXlaGetLSN` function may make this a less efficient option.

Checking for Update Conflicts

If you have applications making simultaneous updates to both your master and subscriber databases, you may encounter update conflicts.

Update conflicts are described in detail in Resolving Replication Conflicts in *Oracle TimesTen In-Memory Database Replication Guide*.

To check for update conflicts in XLA, you can set the ttXlaApply *`test`* parameter to compare the old row value (`ttXlaUpdateDesc_t ->`*`tuple1`*) in each record of type `UPDATETUP` with the existing row in the subscriber database. If the old row value in the update description does not match the corresponding row in the subscriber database, an update conflict is probably the reason. In this case, `ttXlaApply` does not apply the update to the subscriber and returns an `sb_ErrXlaTupleMismatch` error.

Replicating Updates to a Non-TimesTen Database

If you are replicating changes to a non-TimesTen database, you can use the `ttXlaGenerateSQL` function to convert the record data into a SQL statement that can be read by the non-TimesTen subscriber. For update and delete records, `ttXlaGenerateSQL` requires a primary key or a unique index on a non-nullable column to generate the correct SQL. See ttXlaGenerateSQL.

The `ttXlaGenerateSQL` function accepts a ttXlaUpdateDesc_t record as a parameter and outputs its SQL equivalent into a buffer.

> **Note:**
>
> The SQL returned by `ttXlaGenerateSQL` uses TimesTen SQL syntax. The SQL statement may fail on a non-TimesTen subscriber if there are SQL syntax incompatibilities between the two systems. In addition, the SQL statement is encoded in the connection character set associated with the XLA handle.

This example translates a record (`record`) and stores the resulting SQL output in a 200-character buffer (`buffer`). The actual size of the buffer is returned in the `actualLength` parameter.

```
ttXlaUpdateDesc_t record;
char buffer[200];
SQLINTEGER actualLength;
```

```
rc = ttXlaGenerateSQL(xla_handle, &record, buffer, 200, &actualLength);

if (rc != SQL_SUCCESS) {
    handleXLAerror (rc, xla_handle, err_buf, &native_error);
    if ( native_error == 8034 ) { // tt_ErrXlaNoSQL
      printf("Unable to translate to SQL\n");
    }
}
```

Other XLA Features

Changing the location of a bookmark and passing the application context are additional XLA features.

- Changing the Location of a Bookmark
- Passing Application Context

Changing the Location of a Bookmark

At any point during a connection, you can call the `ttXlaGetLSN` function to query the system for the Current Read log record identifier.

See ttXlaGetLSN.

If you must replay a set of updates, you can use the ttXlaSetLSN function to reset the Current Read log record identifier to any valid value larger than the Initial Read log record identifier set by the last ttXlaAcknowledge call. In this context, "larger" only applies if the log record identifiers being compared are from records in the same transaction. If that is not the case, then any log record identifier from a transaction that committed before another transaction is the "smaller" log record identifier, even if the numeric value of the log record identifier is larger. The only way to enable the Initial Read log record identifier to move forward to the Current Read log record identifier is by calling the `ttXlaAcknowledge` function, which indicates that you have received and processed all transaction log records up to the Current Read log record identifier. Once you have called `ttXlaAcknowledge` on a particular bookmark, you can no longer access transaction log records with a log record identifier smaller than the Current Read log record identifier.

Passing Application Context

Although it is not an XLA function, writers to the transaction log can call the `ttApplicationContext` built-in procedure to pass binary data associated with an application to XLA readers.

The `ttApplicationContext` built-in procedure specifies a single `VARBINARY` value that is returned in the next update record produced by the current transaction. XLA readers can obtain a pointer to this value as described in Reading NOT INLINE Variable-Length Column Data.

> **Note:**
>
> A context value is applied to only one update record. After it has been applied it is reset. If the same context value should be applied to multiple updates, then it must be reestablished before each update.

To set the context:

1. Declare two program variables for invoking the `ttApplicationContext` procedure. The variable `contextBuffer` is a `CHAR` array that is declared to be large enough to accommodate the longest application context that you use. The variable `contextBufferLen` is of type `INTEGER` and is used to convey the actual length of the context on each call to `ttApplicationContext`.
2. Initialize a statement handle with a compiled invocation of the `ttApplicationContext` built-in procedure:

   ```
   rc = SQLPrepare(hstmt, "call ttApplicationContext(?)", SQL_NTS);
   rc = SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_BINARY,
                         SQL_VARBINARY, 0, 0, &contextBuffer,
                         sizeof contextBuffer, &contextBufferLen);
   ```

3. When the application context must be set later, copy the context value into `contextBuffer`, assign the length of the context to `contextBufferLen`, and invoke `ttApplicationContext` with the call:

   ```
   rc = SQLExecute(hstmt);
   ```

   The transaction is then committed with the usual call on `SQLTransact`:

   ```
   rc = SQLTransact(NULL, hdbc, SQL_COMMIT);
   ```

   > **Note:**
   >
   > If a SQL operation fails after a call to `ttApplicationContext`, the context may not be stored in the next SQL operation and therefore may be lost. If this happens, the application can call `ttApplicationContext` again before the next SQL operation.

6 Distributed Transaction Processing: XA

TimesTen Classic implementats the X/Open XA standard.

- Overview of XA
- XA in TimesTen
- XA Support Through the Windows ODBC Driver Manager
- Configuring Tuxedo to Use TimesTen XA

Also refer to the following documents:

- X/Open CAE Specification, *Distributed Transaction Processing: The XA Specification*, published by the The Open Group (`http://www.opengroup.org`)
- Tuxedo documentation, available at this location:

  `https://www.oracle.com/middleware/technologies/tuxedo.html`
- WebLogic documentation, available at this location:

  `https://www.oracle.com/middleware/technologies/weblogic.html`

Overview of XA

This section provides a brief overview of XA concepts.

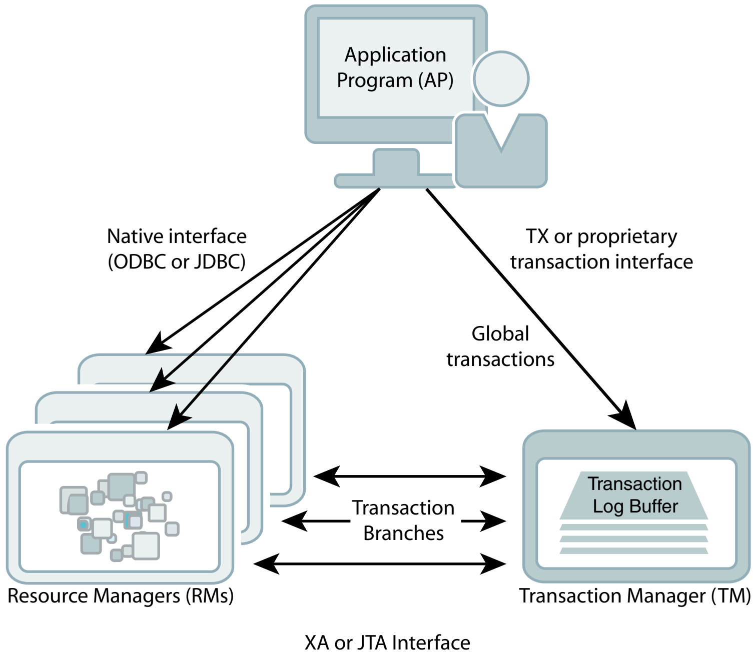
- X/Open DTP Model
- Two-Phase Commit

X/Open DTP Model

Applications use the TX interface to communicate with a transaction manager. In the DTP model, the transaction manager breaks each global transaction down into multiple branches and distributes them to separate resource managers for service. It uses the XA interface to coordinate each transaction branch with the appropriate resource manager.

Figure 6-1 that follows illustrates the interfaces defined by the X/Open DTP model and shows an application communicating global transactions to the transaction manager.

Figure 6-1 Distributed Transaction Processing Model



Global transaction control provided by the TX and XA interfaces is distinct from local transaction control provided by the native ODBC interface. It is generally best to maintain separate connections for local and global transactions. Applications can obtain a connection handle to a TimesTen resource manager in order to initiate both local and global transactions over the same connection. See TimesTen tt_xa_context Function to Obtain ODBC Handle From XA Connection.

Two-Phase Commit

In an XA implementation, the transaction manager commits the distributed branches of a global transaction by using a two-phase commit protocol.

1. In phase one, the transaction manager directs each resource manager to prepare to commit, which is to verify and guarantee it can commit its respective branch of the global transaction. If a resource manager cannot commit its branch, the transaction manager rolls back the entire transaction in phase two.
2. In phase two, the transaction manager either directs each resource manager to commit its branch or, if a resource manager reported it was unable to commit in phase one, rolls back the global transaction.

Note the following optimizations:

- If a global transaction is determined by the transaction manager to have involved only one branch, it skips phase one and commits the transaction in phase two.
- If a global transaction branch is read-only, where it does not generate any transaction log records, the transaction manager commits the branch in phase one and skips phase two for that branch.

> **Note:**
>
> The transaction manager considers the global transaction committed if and only if all branches successfully commit.

XA in TimesTen

The implementation of XA for TimesTen Classic provides an API that is consistent with the API specified in *Distributed Transaction Processing: The XA Specification*.

This section describes what you should know when using the TimesTen implementation of XA, covering the following topics:

- Introduction to the TimesTen XA Implementation and Limitations
- TimesTen Database Requirements for XA
- Global Transaction Recovery in TimesTen
- Considerations in Using Standard XA Functions With TimesTen
- TimesTen tt_xa_context Function to Obtain ODBC Handle From XA Connection
- Considerations in Calling ODBC Functions Over XA Connections in TimesTen
- XA Resource Manager Switch
- XA Error Handling in TimesTen

Introduction to the TimesTen XA Implementation and Limitations

The TimesTen implementation of the XA interfaces is intended for use by transaction managers in distributed transaction processing (DTP) environments. You can use these interfaces to write a new transaction manager or to adapt an existing transaction manager, such as Oracle Tuxedo, to operate with TimesTen resource managers.

In the context of TimesTen XA, the resource managers can be a collection of TimesTen databases, or databases in combination with other commercial databases that support XA.

Note these important limitations:

- The TimesTen XA implementation does not work with cache. The start of any XA transaction fails if the cache agent is running.
- You cannot execute an XA transaction if TimesTen replication is enabled.
- The TimesTen driver manager does not support XA.
- Do not execute DDL statements within an XA transaction.

TimesTen Database Requirements for XA

To guarantee global transaction consistency, TimesTen XA transaction branches must be durable.

The TimesTen implementation of the `xa_prepare()`, `xa_rollback()`, and `xa_commit()` functions log their actions to the file system, regardless of the value set in the `DurableCommits` general connection attribute or by the `ttDurableCommit` built-in procedure.

(The behavior is equivalent to what occurs with a setting of `DurableCommits=1`. See DurableCommits in *Oracle TimesTen In-Memory Database Reference*.) If you must recover from a failure, both the resource manager and the TimesTen transaction manager have a consistent view of which transaction branches were active in a prepared state at the time of failure.

Global Transaction Recovery in TimesTen

When a database is loaded from the file system to recover after a failure or unexpected termination, any global transactions that were prepared but not committed are left pending, or in doubt. Regular processing is not enabled until the disposition of all in-doubt transactions has been resolved.

After connection and recovery are complete, TimesTen checks for in-doubt transactions. If there are no in-doubt transactions, operation proceeds as expected. If there are in-doubt transactions, other connections may be created, but virtually all operations are prohibited on those connections until the in-doubt transactions are resolved. Any other ODBC or JDBC calls result in the following error:

```
Error 11035 - "In-doubt transactions awaiting resolution in recovery must be
resolved first"
```

The list of in-doubt transactions can be retrieved through the XA implementation of `xa_recover()`, then dealt with through the XA call `xa_commit()`, `xa_rollback()`, or `xa_forget()`, as appropriate. After all of the in-doubt transactions are cleared, operation proceeds as expected.

This scheme should be adequate for systems that operate strictly under control of the transaction manager, since the first thing the transaction manager should do after connect is to call `xa_recover()`.

If the transaction manager is unavailable or cannot resolve an in-doubt transaction, you can use the `ttXactAdmin` utility `-HCommit` or `-HAbort` option to independently commit or roll back the individual transaction branches. Be aware, however, that these `ttXactAdmin` options require `ADMIN` privilege. See ttXactAdmin in *Oracle TimesTen In-Memory Database Reference*.

Considerations in Using Standard XA Functions With TimesTen

There are some issues concerning the use of standard XA functions in TimesTen, for those creating your own transaction manager.

Discussion covers the following:

- xa_open() Function
- xa_close() Function
- Transaction Id (XID) Parameter

xa_open() Function

The *`xa_info`* string used by `xa_open()` should be a connection string identical to that supplied to `SQLDriverConnect`.

```
"DSN=DataStoreResource;UID=MyName"
```

XA limits the length of the string to 256 characters. See `MAXINFOSIZE` in the `xa.h` header file.

The `xa_open()` function automatically turns off autocommit when it opens an XA connection.

A connection opened with `xa_open()` must be closed with a call to `xa_close()`.

xa_close() Function

The *`xa_info`* string used by `xa_close()` should be empty.

Transaction Id (XID) Parameter

XA uniquely identifies global transactions by using a transaction ID, referred to as an *XID*. The XID is a required parameter for XA functions that manipulate a transaction. Internally, TimesTen maps XIDs to its own transaction identifiers.

The XID defined by the XA standard has some of its members (such as `formatID`, `gtrid_length`, and `bqual_length`) defined as type `long`. Historically, this could cause problems when a 32-bit client application connected to a 64-bit server, or a 64-bit client application connected to a 32-bit server, because `long` is a 32-bit integer on 32-bit platforms and a 64-bit integer on 64-bit platforms (other than 64-bit Windows). Because of this, TimesTen internally uses only the 32 least significant bits of those XID members regardless of the platform type of client or server. TimesTen does not support any value in those XID members that does not fit in a 32-bit integer.

TimesTen tt_xa_context Function to Obtain ODBC Handle From XA Connection

The TimesTen function `tt_xa_context()` enables you to acquire the ODBC connection handle associated with an XA connection opened by `xa_open()`.

- tt_xa_context Syntax and Parameter Descriptions
- Using tt_xa_context

tt_xa_context Syntax and Parameter Descriptions

Syntax for the `tt_xa_context` function is as follows:

```
#include <tt_xa.h>
int tt_xa_context(int* rmid, SQLHENV* henv, SQLHDBC* hdbc);
```

This table describes the parameters:

Parameter	Type	Description
rmid	int	The specified resource manager ID If this is non-null, the function returns the handles associated with the *rmid* value. If the specified *rmid* is null, the function returns the handles associated with the first connection on this thread. For example, specify a null value if the connection has been opened outside the scope of the user-written code, where *rmid* is unknown. This establishes context in the application environment.
henv	out SQLHENV	The environment handle associated with the current xa_open() context
hdbc	out SQLHDBC	The connection handle associated with the current xa_open() context

The function returns these values:

0: Success

1: *rmid* not found

-1: Invalid parameter

Using tt_xa_context

This section shows how to use tt_xa_context.

In the following example, assume Tuxedo has used xa_open() and xa_start() to open a connection to the database and start a transaction. To do further ODBC processing on the connection, use the tt_xa_context() function to locate the SQLHENV and SQLHDBC handles allocated by xa_open().

```
do_insert()
{

    SQLHENV henv;
    SQLHDBC hdbc;
    SQLHSTMT hstmt;

    /* retrieve the handles for the current connection */
    tt_xa_context(NULL, &henv, &hdbc);

    /* now we can do our ODBC programming as usual */
    SQLAllocStmt(hdbc, &hstmt);

    SQLExecDirect(hstmt, "insert into t1 values (1)", SQL_NTS);

    SQLFreeStmt(hstmt, SQL_DROP);
}
```

Considerations in Calling ODBC Functions Over XA Connections in TimesTen

There are factors to consider when calling ODBC functions using an ODBC handle associated with an XA connection opened by `xa_open()`.

These are discussed in the following sections:

- Autocommit
- Local Transaction COMMIT and ROLLBACK
- Close Open Cursors

Autocommit

To simplify operation and prevent possible contradictions, `xa_open()` automatically turns off autocommit when it opens an XA connection.

Autocommit may subsequently be turned on or off during local transaction work, but must be turned off before `xa_start()` is called to begin work on a global transaction branch. If autocommit is on, a call to `xa_start()` returns the following error:

```
Error 11030 - "Autocommit must be turned off when working on global (XA) transactions"
```

Once `xa_start()` has been called to begin work on a global transaction branch, autocommit may not be turned on until such work has been completed through a call to `xa_end()`. Any attempt to turn on autocommit in this case results in the same error as above.

Local Transaction COMMIT and ROLLBACK

Once work on a global transaction branch has commenced through a call to `xa_start()`, attempts to perform a local commit or rollback using `SQLTransact` results in an error.

```
Error 11031 - "Invalid combination of local transaction and global (XA) transaction"
```

Close Open Cursors

Any open statement cursors must be closed using `SQLFreeStmt` with a value of `SQL_CLOSE` before calling `xa_end()` to end work on a global transaction branch.

Otherwise, the following error is returned:

```
Error 11032 - "XA request failed due to open cursors"
```

XA Resource Manager Switch

This section discusses the XA resource manager switch.

- About the Resource Manager Switch
- XA Switch xa_switch_t
- TimesTen Switch tt_xa_switch

About the Resource Manager Switch

Each resource manager defines a switch in its `xa.h` header file that provides the transaction manager with access to the XA functions in the resource managers.

The transaction manager never directly calls an XA interface function. Instead, it calls the function in the switch table that, in turn, points to the appropriate function in the resource manager. Then resource managers can be added and removed without the requirement to recompile the applications.

In the TimesTen implementation of XA, the functions in the XA switch, `xa_switch_t`, point to their respective functions defined in a TimesTen switch, `tt_xa_switch`.

XA Switch xa_switch_t

The `xa_switch_t` structure is defined by the XA specification.

```
/* XA Switch Data Structure */
#define RMNAMESZ       32        /* length of resource manager name, */
                                 /* including the null terminator */
#define MAXINFOSIZE    256       /* maximum size in bytes of xa_info strings, */
                                 /* including the null terminator */

struct xa_switch_t
{

    char name[RMNAMESZ];               /* name of resource manager */
    long flags;                        /* resource manager specific options */
    long version;                      /* must be 0 */

int (*xa_open_entry)(char*, int, long);        /* xa_open function pointer */
int (*xa_close_entry)(char*, int, long);       /* xa_close function pointer*/
int (*xa_start_entry)(XID*, int, long);        /* xa_start function pointer */
int (*xa_end_entry)(XID*, int, long);          /* xa_end function pointer */
int (*xa_rollback_entry)(XID*, int, long);     /* xa_rollback function pointer */
int (*xa_prepare_entry)(XID*, int, long);      /* xa_prepare function pointer */
int (*xa_commit_entry)(XID*, int, long);       /* xa_commit function pointer */
int (*xa_recover_entry)(XID*, long, int, long); /* xa_recover function pointer*/
int (*xa_forget_entry)(XID*, int, long);        /* xa_forget function pointer */
int (*xa_complete_entry)(int*, int*, int, long);/* xa_complete function pointer
*/
};

typedef struct xa_switch_t xa_switch_t;
/*
 * Flag definitions for the RM switch
 */
#define TMNOFLAGS 0x00000000L     /* no resource manager features selected */
#define TMREGISTER 0x00000001L    /* resource manager dynamically registers */
#define TMNOMIGRATE 0x00000002L   /* RM does not support association migration */
#define TMUSEASYNC 0x00000004L    /* RM supports asynchronous operations */
```

TimesTen Switch tt_xa_switch

The `tt_xa_switch` names the actual functions implemented by a TimesTen resource manager. It also indicates explicitly that association migration is not supported. In addition, dynamic registration and asynchronous operations are not supported.

```
struct xa_switch_t
tt_xa_switch =
{
    "TimesTen", /* name of resource manager */
    TMNOMIGRATE, /* RM does not support association migration */
    0,
    tt_xa_open,
    tt_xa_close,
    tt_xa_start,
    tt_xa_end,
    tt_xa_rollback,
    tt_xa_prepare,
    tt_xa_commit,
    tt_xa_recover,
    tt_xa_forget,
    tt_xa_complete
};
```

XA Error Handling in TimesTen

The XA specification has a limited and strictly defined set of errors that can be returned from XA interface calls. The ODBC `SQLError` function returns XA-defined errors along with any additional information.

The TimesTen XA-related errors begin at number 11000. Errors 11002 through 11020 correspond to the errors defined by the XA standard.

See Errors and Warnings in *Oracle TimesTen In-Memory Database Error Messages and SNMP Traps*.

XA Support Through the Windows ODBC Driver Manager

There are issues and procedures for using XA with the Windows ODBC driver manager. Linux or UNIX ODBC driver managers are not considered.

- Issues to Consider With the Driver Manager
- Linking to the TimesTen ODBC XA Driver Manager Extension Library

Issues to Consider With the Driver Manager

XA support through the ODBC driver manager requires special handling.

There are two fundamental issues:

- The XA interface is not part of the defined ODBC interface. If the XA symbols are directly referenced in an application, it is not possible to link with only the driver manager library to resolve all the external references.
- By design, the driver manager determines which driver `.dll` file to load at connect time, when you call `SQLConnect` or `SQLDriverConnect`. XA dictates that the connection should

be opened through `xa_open()`. But the correct `xa_open()` entry point cannot be located until the `.dll` is loaded during the connect operation itself.

Note that the driver manager objective of database portability is generally not applicable here, since each XA implementation is essentially proprietary. The primary benefit of driver manager support for XA-enabled applications is to enable TimesTen-specific applications to run transparently with either the TimesTen direct driver or the TimesTen client/server driver.

Linking to the TimesTen ODBC XA Driver Manager Extension Library

On Windows installations, TimesTen provides a driver manager extension library, `ttxadm221.dll`, for XA functions. Applications can make XA calls directly, but must link in the extension library.

To link with the `ttxadm221.dll` library, applications must include `ttxadm221.lib` before `odbc32.lib` in their link line. For example:

```
# Link with the ODBC driver manager
appldm.exe:appl.obj
       $(CC) /Feappldm.exe appl.obj ttxadm221.lib odbc32.lib
```

Configuring Tuxedo to Use TimesTen XA

There are tasks to configure Tuxedo to use the TimesTen resource managers.

- Introductory Notes and Cautions
- Update the $TUXDIR/udataobj/RM File
- Build the Tuxedo Transaction Manager Server
- Update the GROUPS Section in the UBBCONFIG File
- Compile the Servers

Introductory Notes and Cautions

The examples in the sections that follow use the direct driver. You can also use the client/server library or driver manager library with the XA extension library.

Information on configuring TimesTen for object-relational mapping frameworks and application servers, including Oracle WebLogic Application Server, is available in the TimesTen Classic Quick Start. See About TimesTen Quick Start and Sample Applications.

> **Note:**
>
> Though TimesTen XA has been demonstrated to work with the Oracle Tuxedo transaction manager, TimesTen cannot guarantee the operation of DTP software beyond the TimesTen implementation of XA.

Update the $TUXDIR/udataobj/RM File

To integrate the TimesTen XA resource manager into the Oracle Tuxedo system, update the `$TUXDIR/udataobj/RM` file to identify the TimesTen resource manager, the name of the TimesTen resource manager switch (`tt_xa_switch`), and the name of the library for the resource manager.

On Linux or UNIX platforms, add:

```
TimesTen:tt_xa_switch:-Ltimesten_home/install/lib -ltten
```

On Windows platforms, add:

```
TimesTen;tt_xa_switch;timesten_home\install\lib\ttdv221.lib
```

> **Note:**
>
> - The *`timesten_home`*`/install` directory is a symbolic link to *`installation_dir`*, the path to the TimesTen installation directory.
> - On Windows, there is only one TimesTen instance, and *`timesten_home`* refers to *`installation_dir`*`\instance`.

Build the Tuxedo Transaction Manager Server

Use the `buildtms` command to build a transaction manager server for the TimesTen resource manager. Then copy the `TMS_TT` file created by `buildtms` to the `$TUXDIR/bin` directory.

On Linux or UNIX platforms, the commands are the following:

```
buildtms -o TMS_TT -r TimesTen -v
cp TMS_TT $TUXDIR/bin
```

On Windows platforms, the commands are the following:

```
buildtms -o TMS_TT -r TimesTen -v
copy TMS_TT.exe %TUXDIR%\bin
```

Update the GROUPS Section in the UBBCONFIG File

For `TMSNAME`, specify the `TMS_TT` file created by the `buildtms` command.

```
TMSNAME=TMS_TT
```

Enter a line for each TimesTen resource manager that specifies a group name, followed by the `LMID`, `GRPNO`, and `OPENINFO` parameters. Your `OPENINFO` string should look like this:

```
OPENINFO="TimesTen:DSN=DSNname"
```

Where *`DSNname`* is the name of the database.

Note that on Windows, Tuxedo servers run as user `SYSTEM`. Add the `UID` general connection attribute to the `OPENINFO` string to specify a user other than `SYSTEM`:

```
OPENINFO="TimesTen:DSN=DSNname;UID=user"
```

Do not specify a `CLOSEINFO` parameter for any TimesTen resource manager.

The following example shows the portions of a `UBBCONFIG` file used to configure two TimesTen resource managers, `GROUP1` and `GROUP2`.

```
*RESOURCES
...
*MACHINES
...
ENGSERV LMID=simple
*GROUPS
DEFAULT: TMSNAME=TMS_TT TMSCOUNT=2
GROUP1
    LMID=simple GRPNO=1 OPENINFO="TimesTen:DSN=MyDSN1;UID=MyName"
GROUP2
    LMID=simple GRPNO=2 OPENINFO="TimesTen:DSN=MyDSN2;UID=MyName"
*SERVERS
DEFAULT:
    CLOPT="-A"
simpserv1 SRVGRP=GROUP1 SRVID=1
simpserv2 SRVGRP=GROUP2 SRVID=2

*SERVICES
TOUPPER
TOLOWER
```

Compile the Servers

Set the `CFLAGS` environment variable to include the *`timesten_home`*`/install/include` directory that contains the TimesTen include files. Then use the `buildserver` command to construct an Oracle Tuxedo ATMI server load module.

On Linux or UNIX platforms, enter the following.

```
export CFLAGS=-Itimesten_home/install
buildserver -o server -f server.c -r TimesTen -s SERVICE
```

On Windows platforms, enter the following.

```
set CFLAGS=-Itimesten_home\install
buildserver -o server -f server.c -r TimesTen -s SERVICE
```

Note:

- The *`timesten_home`*`/install` directory is a symbolic link to *`installation_dir`*, the path to the TimesTen installation directory.
- On Windows, there is only one TimesTen instance, and *`timesten_home`* refers to *`installation_dir`*`\instance`.

The following example shows how to use the `buildclient` command to construct the client module (`simpcl`) and the `buildserver` command to construct the two server modules described in the `UBBCONFIG` file in the example in the preceding section, Update the GROUPS Section in the UBBCONFIG File.

```
set CFLAGS=-Itimesten_home\install
buildclient -o simpcl -f simpcl.c
buildserver -v -t -o simpserv1 -f simpserv1.c -r TimesTen -s TOUPPER
buildserver -v -t -o simpserv2 -f simpserv2.c -r TimesTen -s TOLOWER
```

7 ODBC Application Tuning

You can tune an ODBC application to run optimally on a TimesTen database.

See TimesTen Database Performance Tuning in *Oracle TimesTen In-Memory Database Operations Guide* for more general tuning tips.

This chapter includes the following topics:

- Avoid Generic Driver Managers If Possible
- Use Arrays of Parameters for Batch Execution
- Avoid Excessive Binds
- Avoid SQLGetData
- Avoid Data Type Conversions
- Bulk Fetch Rows of TimesTen Data
- Optimize Queries

Avoid Generic Driver Managers If Possible

Applications that do not need ODBC access to database systems other than TimesTen should not use a generic driver manager as it adds unnecessary overhead.

Applications that need to use TimesTen direct and client/server connections concurrently must link with the TimesTen driver manager. The performance improvement is significant compared to generic driver managers. Otherwise, applications can be linked directly with the TimesTen direct or client/server driver, as described in Linking Options. The performance improvement is significant.

Use Arrays of Parameters for Batch Execution

You can improve performance by using groups, referred to as *batches*, of statement executions in your application.

The `SQLParamOptions` ODBC function enables an application to specify multiple values for the set of parameters assigned by `SQLBindParameter`. This is useful for processing the same SQL statement multiple times with various parameter values. For example, your application can specify multiple sets of values for the set of parameters associated with an `INSERT` statement, and then execute the `INSERT` statement once to perform all the insert operations.

TimesTen supports the use of `SQLParamOptions` with `INSERT`, `UPDATE`, `DELETE`, and `MERGE` statements, but not with `SELECT` statements. (TimesTen Scaleout does not support `MERGE` statements.)

The ideal batch size for any of these database operations varies according to details of the user environment and requires testing and experimentation to determine.

> **Tip:**
>
> In TimesTen Classic, it is important to use a batch size that is an exact multiple of 256 for inserts, in order to optimize the insert mechanism.

Table 7-1 provides a summary of `SQLParamOptions` arguments. Refer to ODBC API reference documentation for details.

Table 7-1 SQLParamOptions Arguments

Argument	Type	Description
hstmt	SQLHSTMT	Statement handle
crow	SQLULEN	Number of values for each parameter
pirow	SQLULEN	Pointer to storage for the current row number

Assuming the *crow* value is greater than 1, the *rgbValue* argument of `SQLBindParameter` points to an array of parameter values and the *pcbValue* argument points to an array of lengths. (Also see SQLBindParameter Function.)

In the TimesTen Classic Quick Start, refer to source file `bulkinsert.c` for a complete working example of batching. (Also, for programming in C++ with TTClasses, see `bulktest.cpp`.) See About TimesTen Quick Start and Sample Applications.

> **Note:**
>
> When using `SQLParamOptions` with the TimesTen client/server driver, data-at-execution parameters are not supported. (An application can pass the value for a parameter either in the `SQLBindParameter` *rgbValue* buffer or with one or more calls to `SQLPutData`. Parameters whose data is passed with `SQLPutData` are known as *data-at-execution* parameters. These are commonly used to send data for `SQL_LONGVARBINARY` and `SQL_LONGVARCHAR` parameters and can be mixed with other parameters.)

Avoid Excessive Binds

The purpose of a `SQLBindCol` or `SQLBindParameter` call is to associate a type conversion and program buffer with a data column or parameter.

For a given SQL statement, if the type conversion or memory buffer for a given data column or parameter is not going to change over repeated executions of the statement, it is better not to make repeated calls to `SQLBindCol` or `SQLBindParameter`. Simply prepare once and bind once to execute many times.

Avoid SQLGetData

`SQLGetData` can be used for fetching data without binding columns.

This can sometimes have a negative impact on performance because applications have to issue a `SQLGetData` ODBC call for every column of every row that is fetched. In contrast, using bound columns requires only one ODBC call for each fetched column. Further, the TimesTen ODBC driver is more highly optimized for the bound columns method of fetching data.

`SQLGetData` can be very useful, though, for doing piecewise fetches of data from long character or binary columns. (This is discussed for LOBs in Using the LOB Piecewise Data Interface in ODBC.)

Avoid Data Type Conversions

TimesTen instruction paths are so short that even small delays due to data conversion can cause a relatively large percentage increase in transaction time.

To avoid data type conversions:

- Match input argument types to expression types.
- Match the types of output buffers to the types of the fetched values.
- Match the connection character set to the database character set.

Bulk Fetch Rows of TimesTen Data

TimesTen provides the `TT_PREFETCH_COUNT` ODBC statement option to enable an application to fetch multiple rows of data.

This feature is available for applications that use the Read Committed isolation level. For applications that retrieve large amounts of TimesTen data, fetching multiple rows can increase performance greatly. However, locks are held on all rows being retrieved until the application has received all the data, decreasing concurrency. See Prefetching Multiple Rows of Data.

Optimize Queries

TimesTen provides the `TT_PREFETCH_CLOSE` ODBC connection option to optimize query performance.

See Optimizing Query Performance.

8 TimesTen Utility API

This chapter provides reference information for TimesTen C language functions, referred to as *utility functions* or the *utility library*, beginning with some overview.

Overview of the TimesTen Utility Library

This section provides some overview for the TimesTen utility library.

- About the Utility Library
- Requirements for the Utility Library
- Utility Function Return Codes

About the Utility Library

The TimesTen utility library C language functions provide programmatic interfaces to some of the command line utilities.

See "Utilities" in *Oracle TimesTen In-Memory Database Reference*.

These functions are supported for TimesTen ODBC applications using the direct driver or using the TimesTen driver manager for direct connections.

Requirements for the Utility Library

Applications that use this set of C language functions must include `ttutillib.h` and `ttutil.h` and link with the `libttutil.so` TimesTen utility library.

Refer to Compiling and Linking Applications Directly With the TimesTen Drivers on Linux or UNIX, Compiling and Linking Applications on Windows, and Compiling and Linking Applications With the TimesTen Driver Manager on Linux or UNIX.

> **Tip:**
>
> Applications must call the ttUtilAllocEnv C function before calling any other TimesTen utility library function. In addition, applications must call the ttUtilFreeEnv C function when done using the TimesTen utility library interface.

Utility Function Return Codes

Unless otherwise indicated, the utility functions return these codes as defined in `ttutillib.h`.

The application must call the ttUtilGetError C function to retrieve all actual error or warning information.

Code	Description
`TTUTIL_SUCCESS`	Indicates success.
`TTUTIL_ERROR`	Indicates an error occurs.
`TTUTIL_WARNING`	Upon success, indicates a warning has been generated.
`TTUTIL_INVALID_HANDLE`	Indicates an invalid utility library handle is specified.

ttBackup

Description

Creates either a full or an incremental backup copy of the database specified by *connStr*. You can back up a database either to a set of files or to a stream. You can restore the database at a later time using either the ttRestore function or the `ttRestore` utility.

Also see ttBackup in *Oracle TimesTen In-Memory Database Reference*.

Required Privilege

`ADMIN`

Syntax

```
ttBackup (ttUtilHandle handle, const char* connStr,
          ttBackUpType type, ttBooleanType atomic,
          const char* backupDir, const char* baseName,
          ttUtFileHandle stream)
```

Parameters

Parameter	Type	Description
handle	`ttUtilHandle`	Specifies the TimesTen utility library environment handle allocated using ttUtilAllocEnv.
connStr	`const char*`	This is a null-terminated string specifying a connection string that describes the database to be backed up.

Parameter	Type	Description
type	ttBackupType	Specifies the type of backup to be performed. Valid values are as follows: • TT_BACKUP_FILE_FULL: Performs a full file backup to the backup path specified by the *backupDir* and *baseName* parameters. The resulting backup is not enabled for incremental backup. • TT_BACKUP_FILE_FULL_ENABLE: Performs a full file backup to the backup path specified by the *backupDir* and *baseName* parameters. The resulting backup is enabled for incremental backup. • TT_BACKUP_FILE_INCREMENTAL: Performs an incremental file backup to the backup path specified by the *backupDir* and *baseName* parameters, if that backup path contains an incremental-enabled backup of the database. Otherwise, an error is returned. • TT_BACKUP_FILE_INCR_OR_FULL: Performs an incremental file backup to the backup path specified by the *backupDir* and *baseName* parameters of that backup path contains an incremental-enabled backup of the database. Otherwise, it performs a full file backup of the database and marks it incremental enabled. • TT_BACKUP_STREAM_FULL: Performs a stream backup to the stream specified by the *stream* parameter. • TT_BACKUP__INCREMENTAL_STOP: Does not perform a backup. Disables incremental backups for the backup path specified by the *backupDir* and *baseName* parameters. This prevents transaction log files from accumulating for an incremental backup.

Parameter	Type	Description
atomic	ttBooleanType	Specifies the disposition of an existing backup with the same *baseName* and *backupDir* while the new backup is being created. This parameter has an effect only on full file backups when there is an existing backup with the same *baseName* and *backupDir*. It is ignored for incremental backups because they augment, rather than replace, an existing backup. It is ignored for stream backups because they write to the given stream, ignoring the *baseName* and *backupDir* parameters. The following are valid values: • TT_FALSE: The existing backup is destroyed before the new backup begins. If the new backup fails to complete, neither the new, incomplete, backup nor the existing backup can be used to restore the database. This option should be used only when the database is being backed up for the first time, when there is another backup of the database that uses a different *baseName* or *backupDir*, or when the application can tolerate a window of time (typically tens of minutes long for large databases) during which no backup of the database exists. • TT_TRUE: The existing backup is destroyed only after the new backup has completed successfully. If the new backup fails to complete, the old backup is retained and can be used to restore the database. If there is an existing backup with the same *baseName* and *backupDir*, the use of this option ensures that there is no window of time during which neither the existing backup nor the new backup is available for restoring the database, and it ensures that the existing backup is destroyed only if it has been successfully superseded by the new backup. However, it does require enough file system space for both the existing and new backups to reside in the *backupDir* at the same time.
backupDir	const char*	Specifies the backup directory for file backups. It is ignored for stream backups. Otherwise it must be non-null. For TT_BACKUP_INCREMENTAL_STOP, it specifies the directory portion of the backup path that is to be disabled. For TT_BACKUP_INCREMENTAL_STOP or a file backup, an error is returned if NULL is specified.

Parameter	Type	Description
baseName	const char*	Specifies the file prefix for the backup files in the backup directory specified by the *backupDir* parameter for file backups. It is ignored for stream backups. If NULL is specified for this parameter, the file prefix for the backup files is the file name portion of the DataStore attribute in the ODBC definition of the database. For TT_BACKUP_INCREMENTAL_STOP, this parameter specifies the base name portion of the backup path that is to be disabled.
stream	ttUtFileHandle	For stream backups, this parameter specifies the stream to which the backup is to be written. On Linux or UNIX, it is an integer file descriptor that can be written to by using write(2). Pass 1 to write the backup to stdout. On Windows, it is a handle that can be written to using WriteFile. Pass the result of GetStdHandle(STD_OUTPUT_HANDLE) to write the backup to the standard output. This parameter is ignored for file backups. The application can pass TTUTIL_INVALID_FILE_HANDLE for this parameter.

Example

This example backs up the database for the payroll DSN into C:\backup.

```
ttUtilHandle  utilHandle;
int           rc;
rc = ttBackup (utilHandle,  "DSN=payroll", TT_BACKUP_FILE_FULL,
     TT_TRUE, "c:\\backup", NULL, TTUTIL_INVALID_FILE_HANDLE);
```

Upon successful backup, all files are created in the C:\backup directory.

Note

Each database supports only eight incremental-enabled backups.

See Also

ttRestore

ttBackup and ttRestore utilities in *Oracle TimesTen In-Memory Database Reference*

ttDestroyDataStore

Description

Destroys a database, including all checkpoint files, transaction logs and daemon catalog entries corresponding to the database specified by the connection string. It does not delete the DSN itself defined in the `sys.odbc.ini` or user `odbc.ini` file on Linux or UNIX platforms or in the Windows registry on Windows platforms.

Required Privilege

Instance administrator

Syntax

```
ttDestroyDataStore (ttUtilHandle handle, const char* connStr,
                    unsigned int timeout)
```

Parameters

Parameter	Type	Description
handle	ttUtilHandle	Specifies the TimesTen utility library environment handle allocated using ttUtilAllocEnv.
connStr	const char*	This is a null-terminated string specifying the connection string that describes the database to be destroyed. All attributes in this connection string, except the DSN and the `DataStore` attribute, are ignored.
timeout	unsigned int	Specifies the number of times to retry before returning to the caller. `ttDestroyDataStore` continually retries the destroy operation every second until it is successful or the timeout is reached. This is useful in those situations where the destroy fails due to some temporary condition, such as when the database is in use. No retry is performed if this parameter value is 0.

Example

This example destroys a database defined by the `payroll` DSN, consisting of files `C:\dsns\payroll.ds0`, `C:\dsns\payroll.ds1`, and several transaction log files `C:\dsns\payroll.log`*n*.

```
char            errBuff [256];
int             rc;
unsigned int    retCode;
ttUtilErrType   retType;
ttUtilHandle    utilHandle;
...
...
rc = ttDestroyDataStore (utilHandle, "DSN=payroll", 30);
if (rc == TTUTIL_SUCCESS)
   printf ("Datastore payroll successfully destroyed.\n");
else if (rc == TTUTIL_INVALID_HANDLE)
   printf ("TimesTen utility library handle is invalid.\n");
else
  while ((rc = ttUtilGetError (utilHandle, 0, &retCode,
           &retType, errBuff, sizeof (errBuff), NULL)) !=
           TTUTIL_NODATA)
    {
```

```
    ...
    ...
}
```

ttDestroyDataStoreForce

Description

Destroys a database, including all checkpoint files, transaction logs and daemon catalog entries corresponding to the database specified by the connection string. It does not delete the DSN itself defined in the `sys.odbc.ini` or user `odbc.ini` file on Linux or UNIX platforms or in the Windows registry on Windows platforms.

Required Privilege

Instance administrator

Syntax

```
ttDestroyDataStoreForce (ttUtilHandle handle, const char* connstr,
                         unsigned int timeout)
```

Parameters

Parameter	Type	Description
handle	`ttUtilHandle`	Specifies the TimesTen utility library environment handle allocated using ttUtilAllocEnv.
connStr	`const char*`	This is a null-terminated string specifying the connection string that describes the database to be destroyed. All attributes in this connection string, except the DSN and the `DataStore` attribute, are ignored.
timeout	`unsigned int`	Specifies the number of seconds to retry before returning to the caller. The `ttDestroyDataStoreForce` utility continually retries the destroy operation every second until it is successful or the timeout is reached. This is useful when the destroy fails due to some temporary condition, such as when the database is in use. No retry is performed if this parameter value is 0.

Example

This example destroys a database defined by the `payroll` DSN, consisting of files `C:\dsns\payroll.ds0`, `C:\dsns\payroll.ds1`, and several transaction log files `C:\dsns\payroll.log`*n*.

```
char             errBuff [256];
int              rc;
unsigned int     retCode;
ttUtilErrType    retType;
ttUtilHandle     utilHandle;
```

```
...
...
rc = ttDestroyDataStoreForce (utilHandle, "DSN=payroll", 30);
if (rc == TTUTIL_SUCCESS)
  printf ("Datastore payroll successfully destroyed.\n");
else if (rc == TTUTIL_INVALID_HANDLE)
  printf ("TimesTen utility library handle is invalid.\n");
else
  while ((rc = ttUtilGetError (utilHandle, 0, &retCode,
                               &retType, errBuff, sizeof (errBuff),
NULL)) !=
                               TTUTIL_NODATA)
    {
  ...
  ...
}
```

ttRamGrace

Description

Specifies the number of seconds the database specified by the connection string is kept in RAM by TimesTen after the last application disconnects from the database. TimesTen then unloads the database. This grace period can be set or reset at any time but is only in effect if the RAM policy is `TT_RAMPOL_INUSE`.

Required Privilege

Instance administrator

Syntax

```
ttRamGrace (ttUtilHandle handle, const char* connStr, unsigned int
seconds)
```

Parameters

Parameter	Type	Description
handle	`ttUtilHandle`	Specifies the TimesTen utility library environment handle allocated using ttUtilAllocEnv.
connStr	`const char*`	This is a null-terminated string specifying a connection string that describes the database for which the RAM grace period is set.
seconds	`unsigned int`	Specifies the number of seconds TimesTen keeps the database in RAM after the last application disconnects from the database. TimesTen then unloads the database.

Example

This example sets the RAM grace period of 10 seconds for the `payroll` DSN.

```
ttUtilHandle   utilHandle;
int            rc;
rc = ttRamGrace (utilHandle, "DSN=payroll", 10);
```

See Also

- ttRamLoad
- ttRamPolicy
- ttRamUnload

ttRamLoad

Description

Causes TimesTen to load the database specified by the connection string into the system RAM. A call to `ttRamLoad` is valid only when `RamPolicy` is set to `TT_RAMPOL_MANUAL`.

Refer to ttRamPolicySet in *Oracle TimesTen In-Memory Database Reference* or to ttRamPolicy.

Required privilege

Instance administrator

Syntax

```
ttRamLoad (ttUtilHandle handle, const char* connStr)
```

Parameters

Parameter	Type	Description
handle	`ttUtilHandle`	Specifies the TimesTen utility library environment handle allocated using ttUtilAllocEnv.
connStr	`const char*`	This is a null-terminated string specifying a connection string that describes the database to be loaded into RAM.

Example

This example loads the database for the `payroll` DSN.

```
ttUtilHandle   utilHandle;
int            rc;
rc = ttRamLoad (utilHandle, "DSN=payroll");
```

See Also

- ttRamGrace
- ttRamPolicy
- ttRamUnload

ttRamPolicy

Description

Defines the policy used to determine when TimesTen loads the database specified by the connection string into the system RAM.

Required Privilege

Instance administrator

Syntax

```
ttRamPolicy (ttUtilHandle handle, const char* connStr,
             ttRamPolicyType policy)
```

Parameters

Parameter	Type	Description
handle	`ttUtilHandle`	Specifies the TimesTen utility library environment handle allocated using ttUtilAllocEnv.
connStr	`const char*`	This is a null-terminated string specifying a connection string that describes the database for which the RAM policy is to be set.
policy	`ttRamPolicyType`	Specifies the policy used to determine when TimesTen loads the specified database into system RAM. Valid values are the following: • `TT_RAMPOL_ALWAYS`: Specifies that the database should always remain in RAM. • `TT_RAMPOL_MANUAL`: Specifies that the database can be loaded into RAM explicitly using either the ttRamLoad C function or the `ttAdmin -ramLoad` command. Similarly, the database can be unloaded from RAM explicitly by using ttRamUnload C function or using `ttAdmin -ramUnload` command. • `TT_RAMPOL_INUSE`: Specifies that the database is to be loaded into RAM when an application wants to connect to the database. This RAM policy may be further modified using the ttRamGrace C function or using the `ttAdmin -ramGrace` command. If you do not explicitly set the RAM policy for the specified database, the default RAM policy is `TT_RAMPOL_INUSE`. **Note**: `TT_RAMPOL_INUSE` is not supported by TimesTen Scaleout.

Example

This example sets the RAM policy to manual for the `payroll` DSN.

```
ttUtilHandle   utilHandle;
int            rc;
rc = ttRamPolicy (utilHandle, "DSN=payroll", TT_RAMPOL_MANUAL);
```

See Also

ttRamGrace
ttRamLoad
ttRamUnload

ttRamUnload

Description

Causes TimesTen to unload the database specified by the connection string from the system RAM if the TimesTen RAM policy is set to `manual`. This call is valid only when RAM policy is set to `TT_RAMPOL_MANUAL`.

Refer to ttRamPolicySet in *Oracle TimesTen In-Memory Database Reference* or to ttRamPolicy.

Required Privilege

Instance administrator

Syntax

```
ttRamUnload (ttUtilHandle handle, const char* connStr)
```

Parameters

Parameter	Type	Description
handle	`ttUtilHandle`	Specifies the TimesTen utility library environment handle allocated using ttUtilAllocEnv.
connStr	`const char*`	This is a null-terminated string specifying a connection string for the database to be unloaded from RAM.

Example

This example unloads the database from RAM for the `payroll` DSN.

```
ttUtilHandle   utilHandle;
int            rc;
rc = ttRamUnload (utilHandle, "DSN=payroll");
```

See Also

ttRamGrace
ttRamLoad

ttRamPolicy

ttRepDuplicateEx

Description

Creates a replica of a remote database on the local system. The process is initiated from the receiving local system. From there, a connection is made to the remote source database to perform the duplicate operation.

> **Note:**
>
> - This utility has features to recover from a site failure by creating a disaster recovery (DR) read-only subscriber as part of the active standby pair replication scheme. See Using a Disaster Recovery Subscriber in an Active Standby Pair in *Oracle TimesTen In-Memory Database Replication Guide*.
> - If the database does not use cache groups, the following items discussed below are not relevant: `cacheuid` and `cachepwd` data structure elements; `TT_REPDUP_NOKEEPCG`, `TT_REPDUP_RECOVERINGNODE`, `TT_REPDUP_INITCACHEDR`, and `TT_REPDUP_DEFERCACHEUPDATE` flag values.
> - There are elements in the `ttRepDuplicateExArg` structure that is a parameter of this utility, *`localIP`* and *`remoteIP`*, that enable you to optionally specify which local network interface to use, which remote network interface to use, or both.

Required Privilege

Requires an instance administrator on the receiving local database (where `ttRepDuplicateEx` is called) and a user with `ADMIN` privilege on the remote source database. Create the internal user on the remote source store as necessary.

In addition, be aware of the following requirements to execute `ttRepDuplicateEx`:

- The operating system user name of the instance administrator on the receiving local database must be the same as the operating system user name of the instance administrator on the remote source database.
- When `ttRepDuplicateEx` is called, the `uid` and `pwd` data structure elements must specify the user name and password of the user with `ADMIN` privilege on the remote source database. This user name is used to connect to the remote source database to perform the duplicate operation.

Syntax

```
ttRepDuplicateEx (ttUtilHandle handle,
                  const char* destConnStr,
                  const char* srcDatabase,
                  const char* remoteHost,
                  ttRepDuplicateExArg* arg
                  )
```

```
typedef struct
{
      unsigned int size; /*set to size of(ttRepDuplicateExArg) */
      unsigned int flags;
      const char* uid;
      const char* pwd;
      const char* pwdcrypt;
      const char* cacheuid;
      const char* cachepwd;
      const char* localHost;
      int truncListLen;
      const char** truncList;
      int dropListLen;
      const char** dropList;
      int maxkbytesPerSec;
      int remoteDaemonPort;
      int nThreads4initDR;
      const char* localIP
      const char* remoteIP
      int crsManaged;
} ttRepDuplicateExArg
```

Parameters

Parameter	Type	Description
handle	`ttUtilHandle`	Specifies the TimesTen utility library environment handle allocated using ttUtilAllocEnv.
destConnStr	`const char*`	This is a null-terminated string specifying the connection string for a local database into which the replica of the remote database is created.
srcDatabase	`const char*`	This is a null-terminated string specifying the remote source database name. This name is the last component of the database path name.
remoteHost	`const char*`	This is a null-terminated string specifying the TCP/IP host name of the system where the remote source database is located.
arg	`ttRepDuplicateExArg*`	This is the address of the structure containing the desired `ttRepDuplicateEx` arguments. If `NULL` is passed in for *`arg`* or if the value of *`arg`* `->`*`size`* is invalid, TimesTen returns error 12230, "`Invalid argument value`", and `TTUTIL_ERROR`.

Struct Elements

The `ttRepDuplicateExArg` structure contains these elements:

Element	Type	Description
size	`unsigned int`	Size This must be set up to *`sizeof`* (`ttRepDuplicateExArg`).

Element	Type	Description
flags	unsigned int	Bit-wise union of values chosen from the list in the table of flag values
uid	const char*	User name of a user on the remote source database with ADMIN privileges This user name is used to connect to the remote source database to perform the duplicate operation.
pwd	const char*	Password associated with the user ID
pwdcrypt	const char*	Encrypted password associated with the user ID
cacheuid	const char*	TimesTen cache administration user ID
cachepwd	const char*	TimesTen cache administration user password
localHost	const char*	Null-terminated string specifying the TCP/IP host name of the local system This element is ignored if *remoteRepStart* is TT_FALSE. This explicitly identifies the local host. This parameter can be null, which is useful if the local host uses a nonstandard name such as an IP address.
truncListLen	int	Number of elements in the *truncList*
truncList	const char**	List of non-replicated tables to truncate after duplicate
dropListLen	int	Number of elements in *dropList*
dropList	const char**	List of non-replicated tables to drop after the duplicate operation
maxkbytesPerSec	int	Maximum kilobytes per second Setting this to a nonzero value specifies that the duplicate operation should not put more than *maxkbytesPerSec* kilobytes of data per second onto the network. Setting it to 0 or a negative number indicates that the duplicate operation should not attempt to limit its bandwidth.
remoteDaemonPort	int	Remote daemon port Setting this to 0 results in the daemon port number for the target database being set to the port number used for the daemon on the source database. This option cannot be used in duplicate operations for databases with automatic port configuration.
nThreads4initDR	int	Number of threads for initialization For the disaster recovery subscriber, this determines the number of threads used to initialize the Oracle database on the disaster recovery site. After the TimesTen database is copied to the disaster recovery system, the Oracle database tables are truncated and the data from the TimesTen Classic cache groups is copied to the Oracle database on the disaster recovery system. Also see the TT_REPDUP_INITCACHEDR flag below.

Element	Type	Description
localIP	const char*	A null-terminated string specifying the alias or IP address (IPv4 or IPv6) of the local network interface to use for the duplicate operation. Set this to NULL if you do not want to specify the local network interface, in which case any compatible interface may be used.
remoteIP	const char*	A null-terminated string specifying the alias or IP address (IPv4 or IPv6) of the remote network interface to use for the duplicate operation. Set this to NULL if you do not want to specify the remote network interface, in which case any compatible interface may be used. **Note**: You can specify both *localIP* and *remoteIP*, or either one by itself, or neither.
crsManaged	int	For internal use This should be set to 0 (default).

The ttRepDuplicateExArg *flags* element is constructed from these values:

Value	Description
TT_REPDUP_NOFLAGS	Indicates no flags.
TT_REPDUP_COMPRESS	Enables compression of the data transmitted over the network for the duplicate operation.
TT_REPDUP_REPSTART	Directs ttRepDuplicateEx to set the replication state (with respect to the local database) in the remote database to the start state before the remote database is copied across the network. This ensures that all updates made after the duplicate operation are replicated from the remote database to the newly created or restored local database.
TT_REPDUP_RAMLOAD	Keeps the database in memory upon completion of the duplicate operation. It changes the RAM policy for the database to manual.
TT_REPDUP_DELXLA	Directs ttRepDuplicateEx to remove all the XLA bookmarks as part of the duplicate operation.
TT_REPDUP_NOKEEPCG	Do not preserve the cache group definitions; ttRepDuplicateEx converts all cache group tables into regular tables. By default, cache group definitions are preserved.

Value	Description
TT_REPDUP_RECOVERINGNODE	Specifies that ttRepDuplicateEx is being used to recover a failed node for a replication scheme that has an AWT or autorefresh cache group. Do not specify TT_REPDUP_RECOVERINGNODE when rolling out a new or modified replication scheme to a node. If ttRepDuplicateEx cannot update metadata stored on the Oracle database and all incremental autorefresh cache groups are replicated, then updates to the metadata are automatically deferred until the cache and replication agents are started.
TT_REPDUP_DEFERCACHEUPDATE	Forces the deferral of changes to metadata stored on the Oracle database until the cache and replication agents are started and the agents can connect to the Oracle database. Using this option can cause a full autorefresh if some incremental cache groups are not replicated or if ttRepDuplicateEx is being used for rolling out a new or modified replication scheme to a node.
TT_REPDUP_INITCACHEDR	Initializes disaster recovery. You must also specify *cacheuid* and *cachepwd* in the data structure. Also see *nThreads4initDR* in the data structure.

See Also

ttRepAdmin -duplicate in *Oracle TimesTen In-Memory Database Reference*

The following built-in procedures are described in Built-In Procedures in *Oracle TimesTen In-Memory Database Reference*.

```
ttReplicationStatus
ttRepPolicySet
ttRepStop
ttRepSubscriberStateSet
ttRepSyncGet
ttRepSyncSet
```

ttRestore

Description

Restores a database specified by the connection string from a backup that has been created using the ttBackup C function or ttBackup utility. If the database already exists, ttRestore does not overwrite it.

Also see ttRestore in *Oracle TimesTen In-Memory Database Reference*.

Required Privilege

Instance administrator

Syntax

```
ttRestore (ttUtilHandle handle, const char* connStr,
           ttRestoreType type, const char* backupDir,
           const char* baseName, ttUtFileHandle stream,
           unsigned intflags)
```

Parameters

Parameter	Type	Description
handle	ttUtilHandle	Specifies the TimesTen utility library environment handle allocated using ttUtilAllocEnv.
connStr	const char*	This is a null-terminated string specifying a connection string that describes the database to be restored.
type	ttRestoreType	Indicates whether the database is to be restored from a file or a stream backup. Valid values are the following: • TT_RESTORE_FILE: The database is to be restored from a file backup located at the backup path specified by the *backupDir* and *baseName* parameters. • TT_RESTORE_STREAM: The database is to be restored from a stream backup read from the given stream.
backupDir	const char*	For TT_RESTORE_FILE, specifies the directory where the backup files are stored. For TT_RESTORE_STREAM, this parameter is ignored.
baseName	const char*	For TT_RESTORE_FILE, specifies the file prefix for the backup files in the backup directory specified by the *backupDir* parameter. If NULL is specified, the file prefix for the backup files is the file name portion of the DataStore attribute of the database ODBC definition. For TT_RESTORE_STREAM, this parameter is ignored.
stream	ttUtFileHandle	For TT_RESTORE_STREAM, specifies the stream from which the backup is to be read. On Linux or UNIX, it is an integer file descriptor that can be read from using read(2). Pass 0 to read the backup from stdin. On Windows, it is a handle that can be read from using ReadFile. Pass the result of GetStdHandle(STD_INPUT_HANDLE) to read from the standard input. For TT_RESTORE_FILE, this parameter is ignored. The application can pass TTUTIL_INVALID_FILE_HANDLE for this parameter.

Parameter	Type	Description
flags	`unsigned int`	This is reserved for future use. Set it to 0.

See Also

ttBackup

ttBackup and ttRestore utilities in *Oracle TimesTen In-Memory Database Reference*

ttUtilAllocEnv

Description

Allocates memory for a TimesTen utility library environment handle and initializes the TimesTen utility library interface for use by an application. An application must call `ttUtilAllocEnv` before calling any other TimesTen utility library function. In addition, an application should call ttUtilFreeEnv when it is done using the TimesTen utility library interface.

Required Privilege

None

Syntax

```
ttUtilAllocEnv (ttUtilHandle* handle_ptr, char* errBuff,
                unsigned int buffLen, unsigned int* errLen)
```

Parameters

Parameter	Type	Description
handle_ptr	`ttUtilHandle*`	Specifies a pointer to storage where the TimesTen utility library environment handle is returned.
errBuff	`char*`	This is a user allocated buffer where error messages (if any) are returned. The returned error message is a null-terminated string. If the length of the error message exceeds *buffLen*-1, it is truncated to *buffLen*-1. If this parameter is null, *buffLen* is ignored and TimesTen does not return error messages to the calling application.
buffLen	`unsigned int`	Specifies the size of the buffer *errBuff*. If this parameter is 0, TimesTen does not return error messages to the calling application.
errLen	`unsigned int*`	This is a pointer to an unsigned integer where the actual length of the error message is returned. If it is `NULL`, this parameter is ignored.

Return Codes

This utility returns the following code as defined in `ttutillib.h`.

Code	Description
`TTUTIL_SUCCESS`	Returned upon success.

Otherwise, it returns a TimesTen-specific error message as defined in `tt_errCode.h` and a corresponding error message in the buffer provided by the caller.

Example

This example allocates and initializes a TimesTen utility library environment handle with the name `utilHandle`.

```
char           errBuff [256];
int            rc;
ttUtilHandle   utilHandle;

rc = ttUtilAllocEnv (&utilHandle, errBuff, sizeof(errBuff), NULL);
```

See Also

ttUtilFreeEnv
ttUtilGetError
ttUtilGetErrorCount

ttUtilFreeEnv

Description

Frees memory associated with the TimesTen utility library handle.

An application must call ttUtilAllocEnv before calling any other TimesTen utility library function. In addition, an application should call `ttUtilFreeEnv` when it is done using the TimesTen utility library interface.

Required Privilege

None

Syntax

```
ttUtilFreeEnv (ttUtilHandle handle, char* errBuff,
                unsigned int buffLen, unsigned int* errLen)
```

Parameters

Parameter	Type	Description
handle	`ttUtilHandle`	Specifies the TimesTen utility library environment handle allocated using ttUtilAllocEnv.

Parameter	Type	Description
errBuff	char*	This is a user-allocated buffer where error messages are to be returned. The returned error message is a null-terminated string. If the length of the error message exceeds *buffLen*-1, it is truncated to *buffLen*-1. If this parameter is NULL, *buffLen* is ignored and TimesTen does not return error messages to the calling application.
buffLen	unsigned int	Specifies the size of the buffer *errBuff*. If this parameter is 0, TimesTen does not return error messages to the calling application.
errLen	unsigned int*	This is a pointer to an unsigned integer where the actual length of the error message is returned. If it is NULL, this parameter is ignored.

Return Codes

This utility returns the following codes as defined in ttutillib.h.

Code	Description
TTUTIL_SUCCESS	Returned upon success.
TTUTIL_INVALID_HANDLE	Returned if an invalid utility library handle is specified.

Otherwise, it returns a TimesTen-specific error message as defined in tt_errCode.h and a corresponding error message in the buffer provided by the caller.

Example

This example frees a TimesTen utility library environment handle named utilHandle.

```
char           errBuff [256];
int            rc;
ttUtilHandle   utilHandle;

rc = ttUtilFreeEnv (utilHandle, errBuff, sizeof(errBuff), NULL);
```

See Also

- ttUtilAllocEnv
- ttUtilGetError
- ttUtilGetErrorCount

ttUtilGetError

Description

Retrieves the errors and warnings generated by the last call to the TimesTen C utility library functions excluding ttUtilAllocEnv and ttUtilFreeEnv.

Required Privilege

None

Syntax

```
ttUtilGetError (ttUtilHandle handle, unsigned int errIndex,
                unsigned int* retCode, ttUtilErrType* retType,
                char* errbuff, unsigned int buffLen,
                unsigned int* errLen)
```

Parameters

Parameter	Type	Description
handle	ttUtilHandle	Specifies the TimesTen utility library environment handle allocated using ttUtilAllocEnv.
errIndex	unsigned int	Indicates error or warning record to be retrieved from the TimesTen utility library error array. Valid values are as follows: • 0: Retrieve the next record from the utility library error array. • 1...*n*: Retrieve the specified record from the utility library error array, where *n* is the error count returned by the ttUtilGetErrorCount call.
retCode	unsigned int*	Returns the TimesTen-specific error or warning codes as defined in tt_errCode.h.
retType	ttUtilErrType*	Indicates whether the returned message is an error or warning. The following are valid return values: • TTUTIL_ERROR • TTUTIL_WARNING
errBuff	char*	This is a user allocated buffer where error messages (if any) are to be returned. The returned error message is a null-terminated string. If the length of the error message exceeds *buffLen*-1, it is truncated to *buffLen*-1. If this parameter is NULL, *buffLen* is ignored and TimesTen does not return error messages to the calling application.
buffLen	unsigned int	Specifies the size of the buffer *errBuff*. If this parameter is 0, TimesTen does not return error messages to the calling application.
errLen	unsigned int*	A pointer to an unsigned integer where the actual length of the error message is returned. If it is NULL, TimesTen ignores this parameter.

Return Codes

This utility returns the following codes as defined in ttutillib.h.

Code	Description
TTUTIL_SUCCESS	Returned upon success.
TTUTIL_INVALID_HANDLE	Returned if an invalid utility library handle is specified.
TTUTIL_NODATA	Returned if no error or warming information is retrieved.

Example

This example retrieves all error or warning information after calling ttDestroyDataStore for the DSN named `payroll`.

```
char            errBuff[256];
int             rc;
unsigned int    retCode;
ttUtilErrType   retType;
ttUtilHandle    utilHandle;

rc = ttDestroyDataStore (utilHandle, "DSN=PAYROLL", 30);
if ((rc == TTUTIL_SUCCESS)
  printf ("Datastore payroll successfully destroyed.\n");
else if (rc == TTUTIL_INVALID_HANDLE)
  printf ("TimesTen utility library handle is invalid.\n");
else
    while ((rc = ttUtilGetError (utilHandle, 0,
        &retCode, &retType, errBuff, sizeof (errBuff),
        NULL)) != TTUTIL_NODATA)
    {
...
...
}
```

Notes

Each of the TimesTen C functions can potentially generate multiple errors and warnings for a single call from an application. To retrieve all of these errors and warnings, the application must make repeated calls to ttUtilGetError until it returns `TTUTIL_NODATA`.

See Also

- ttUtilAllocEnv
- ttUtilFreeEnv
- ttUtilGetErrorCount

ttUtilGetErrorCount

Description

Retrieves the number of errors and warnings generated by the last call to the TimesTen C utility library functions, excluding ttUtilAllocEnv and ttUtilFreeEnv. Each of these functions can potentially generate multiple errors and warnings for a single call from an application. To retrieve all of these errors and warnings, the application must make repeated calls to ttUtilGetError until it returns `TTUTIL_NODATA`.

Required Privilege

None

Syntax

```
ttUtilGetErrorCount (ttUtilHandle handle,
                     unsigned int* errCount)
```

Parameters

Parameter	Type	Description
handle	`ttUtilHandle`	Specifies the TimesTen utility library environment handle allocated using ttUtilAllocEnv.
errCount	`unsigned int*`	Indicates the number of errors and warnings generated by the last call, excluding `ttUtilAllocEnv` and `ttUtilFreeEnv`, to the TimesTen utility library.

Return Codes

The utility returns the following codes as defined in `ttutillib.h`.

Code	Description
`TTUTIL_SUCCESS`	Returned upon success.
`TTUTIL_INVALID_HANDLE`	Returned if an invalid utility library handle is specified.

Example

This example retrieves the error and warning count information after calling ttDestroyDataStore for the DSN named `payroll`.

```
int             rc;
unsigned int    errCount;
ttUtilHandle    utilHandle;

rc = ttDestroyDataStore (utilHandle, "DSN=payroll", 30);
if (rc == TTUTIL_SUCCESS)
  printf ("Datastore payroll successfully destroyed.\n")

else if (rc == TTUTIL_INVALID_HANDLE)
  printf ("TimesTen utility library handle is invalid.\n");
else
{
rc = ttUtilGetErrorCount(utilHandle, &errCount);
  ...
  ...
}
```

Notes

Each of the TimesTen utility library functions can potentially generate multiple errors and warnings for a single call from an application. To retrieve all of these errors and warnings, the application must make repeated calls to `ttUtilGetError` until it returns `TTUTIL_NODATA`.

See Also

ttUtilAllocEnv
ttUtilFreeEnv
ttUtilGetError

ttXactIdRollback

Description

Rolls back the transaction indicated by the transaction ID that is specified. The intended user of `ttXactIdRollback` is the `ttXactAdmin` utility. However, programs that want to have a thread with the power to roll back the work of other threads must ensure that those threads call the `ttXactIdGet` built-in procedure before beginning work and put the results into a location known to the thread that wishes to roll back the transaction. (Refer to ttXactIdGet in *Oracle TimesTen In-Memory Database Reference*.)

Required Privilege

`ADMIN`

Syntax

```
ttXactIdRollback (ttUtilHandle handle, const char* connStr,
                  const char* xactId)
```

Parameters

Parameter	Type	Description
handle	`ttUtilHandle`	Specifies the TimesTen utility library environment handle allocated using ttUtilAllocEnv.
connStr	`const char**`	Specifies the connection string of the database, which contains the transaction to be rolled back.
xactId	`const char*`	Indicates the transaction ID for the transaction to be rolled back.

Example

This example rolls back a transaction with the ID 3.4567 in the database named `payroll`.

```
char            errBuff [256];
int             rc;
unsigned int    retCode;
ttUtilErrType   retType;
ttUtilHandle    utilHandle;
...
rc = ttXactIdRollback (utilHandle, "DSN=payroll", "3.4567");
if (rc == TTUTIL_SUCCESS)
```

```
  printf ("Transaction ID successfully rolled back.\n");
else if (rc == TTUTIL_INVALID_HANDLE)
  printf ("TimesTen utility library handle is invalid.\n");
else
  while ((rc = ttUtilGetError (utilHandle, 0, &retCode,
  &retType, errBuff, sizeof (errBuff), NULL)) != TTUTIL_NODATA)
   {
  ...
}
```

9
XLA Reference

This chapter provides reference information for the Transaction Log API (XLA).

See XLA and TimesTen Event Management.

It includes the following topics:

- Overview of XLA Functions
- Summary of XLA Functions by Category
- XLA Function Reference
- XLA Replication Function Reference
- C Data Structures Used by XLA

Overview of XLA Functions

This section provides general information about XLA functions for TimesTen Classic.

- XLA Function Return Codes
- XLA Function Parameter Types (Input, Output, Input/Output)
- Results Output by XLA Functions
- XLA Function Required Privileges

XLA Function Return Codes

All of the XLA API functions described in this chapter return a value of type `SQLRETURN`.

`SQLRETURN` is defined by ODBC to have one of the following values:

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`
- `SQL_NO_DATA_FOUND`
- `SQL_ERROR`

See XLA Error Handling.

> **Note:**
>
> `SQL_NO_DATA_FOUND` is defined in `sqlext.h`, which is included by `timesten.h`.

XLA Function Parameter Types (Input, Output, Input/Output)

There are three XLA function parameter types.

In the function descriptions:

- All parameters are input-only unless otherwise indicated.
- Output parameters are prefixed with `OUT`.
- Input/output parameters are prefixed with `IN OUT`.

Results Output by XLA Functions

Most routines in this API copy results to application buffers. Those few routines that produce pointers to buffers containing results are guaranteed as valid only until the next call with the same XLA handle.

Exceptions to this rule include the following.

- Buffers remain valid across calls to the ttXlaError function that supplies diagnostic information.
- Results returned by ttXlaNextUpdate remain valid until the next call to `ttXlaNextUpdate`.
- For ttXlaAcknowledge, if the application must retain access to the buffers for a longer time, it must copy the information from the buffer returned by XLA to an application-owned buffer.

Character string values in XLA are null-terminated, except for actual column values. Fixed-length `CHAR` columns are space-padded to their full length. `VARCHAR` columns have an explicit length encoded.

XLA uses the same data structures for 64-bit platforms as it has for 32-bit platforms. The types `SQLUINTEGER` and `SQLUBIGINT` refer to 64-bit and 32-bit integers unambiguously. Issues of alignment and padding are addressed by filling the type definition so that each `SQLUINTEGER` value is on a four-byte boundary and each `SQLUBIGINT` value is on an eight-byte boundary. For a description of storage requirements for other TimesTen data types, see Understanding Rows in *Oracle TimesTen In-Memory Database Operations Guide*.

XLA Function Required Privileges

Any XLA functionality requires the system privilege `XLA`.

XLA System Privilege introduces the effects of TimesTen access control features on XLA functionality.

Summary of XLA Functions by Category

TimesTen XLA can be used to detect updates on a TimesTen Classic database or as a toolkit to build your own replication solution.

See XLA and TimesTen Event Management.

This section categorizes the XLA functions based on their use and provides a brief description of each function. It includes the following categories:

- XLA Core Functions
- XLA Data Type Conversion Functions
- XLA Replication Functions

XLA Core Functions

The following table lists all the XLA functions used in typical XLA operations, aside from data conversion functions which are listed separately below.

See Writing an XLA Event-Handler Application for a discussion on how to use most of these functions.

Function	Description
ttXlaAcknowledge	Acknowledges receipt of one or more transaction update records from the transaction log.
ttXlaClose	Closes the XLA handle opened by ttXlaPersistOpen.
ttXlaConvertCharType	Converts column data into the connection character set.
ttXlaDeleteBookmark	Deletes a transaction log bookmark.
ttXlaError	Retrieves error information.
ttXlaErrorRestart	Resets error stack information.
ttXlaGetColumnInfo	Retrieves information about all the columns in the table.
ttXlaGetLSN	Retrieves the log record identifier of the current bookmark for a database.
ttXlaGetTableInfo	Retrieves information about a table.
ttXlaGetVersion	Retrieves the current version of XLA.
ttXlaNextUpdate	Retrieves a batch of updates from TimesTen.
ttXlaNextUpdateWait	Retrieves a batch of updates from TimesTen. Waits for a specified time if no updates are available in the transaction log.
ttXlaPersistOpen	Initializes a handle to a database to access the transaction log.
ttXlaSetLSN	Sets the log record identifier of the current bookmark for a database.
ttXlaSetVersion	Sets the XLA version to be used.
ttXlaTableByName	Finds the system and user table identifiers for a table given the table owner and name.
ttXlaTableStatus	Sets and retrieves XLA status for a table.
ttXlaTableVersionVerify	Checks whether the cached table definitions are compatible with the XLA record being processed.
ttXlaVersionColumnInfo	Retrieves information about the columns in a table for which a change update record must be processed.
ttXlaVersionCompare	Compares two XLA versions.

XLA Data Type Conversion Functions

The following table lists data type conversion functions.

See XLA Data Types.

Function	Description
ttXlaDateToODBCCType	Converts a `TTXLA_DATE_TT` value to an ODBC C value usable by applications.
ttXlaDecimalToCString	Converts a `TTXLA_DECIMAL_TT` value to a character string usable by applications.
ttXlaNumberToBigInt	Converts a `TTXLA_NUMBER` value to a `SQLBIGINT` C value usable by applications.
ttXlaNumberToCString	Converts a `TTXLA_NUMBER` value to a character string usable by applications.
ttXlaNumberToDouble	Converts a `TTXLA_NUMBER` value to a long floating point number value usable by applications.
ttXlaNumberToInt	Converts a `TTXLA_NUMBER` value to an integer usable by applications.
ttXlaNumberToSmallInt	Converts a `TTXLA_NUMBER` value to a `SQLSMALLINT` C value usable by applications.
ttXlaNumberToTinyInt	Converts a `TTXLA_NUMBER` value to a `SQLCHAR` C value usable by applications.
ttXlaNumberToUInt	Converts a `TTXLA_NUMBER` value to an unsigned integer usable by applications.
ttXlaOraDateToODBCTimeStamp	Converts a `TTXLA_DATE` value to an ODBC timestamp usable by applications.
ttXlaOraTimeStampToODBCTimeStamp	Converts a `TTXLA_TIMESTAMP` value to an ODBC timestamp usable by applications.
ttXlaRowidToCString	Converts a `ROWID` value to a character string value usable by applications.
ttXlaTimeToODBCCType	Converts a `TTXLA_TIME` value to an ODBC C value usable by applications.
ttXlaTimeStampToODBCCType	Converts a `TTXLA_TIMESTAMP_TT` value to an ODBC C value usable by applications.

XLA Replication Functions

TimesTen replication as described in Overview of TimesTen Replication in the *Oracle TimesTen In-Memory Database Replication Guide* is sufficient for most TimesTen customer needs. However, it is also possible to use XLA functions to replicate updates from one database to another. Implementing your own replication scheme on top of XLA in this way is fairly complicated, but can be considered if TimesTen replication is not feasible for some reason.

The following table lists functions used exclusively for XLA as a replication mechanism. (Reference information for these functions is in a separate section from other XLA functions, XLA Replication Function Reference.)

Function	Description
ttXlaApply	Applies the update to the database associated with the XLA handle.
ttXlaCommit	Commits a transaction.
ttXlaGenerateSQL	Generates a SQL statement that expresses the effect of an update record.
ttXlaLookup	Looks for an update record for a table with a specific key value.
ttXlaRollback	Rolls back a transaction.
ttXlaTableCheck	Verifies that the named table in the table description received from the sending database is compatible with the receiving database.

See Using XLA as a Replication Mechanism for a discussion on how to use these functions.

XLA Function Reference

This section provides reference information for XLA core functions and XLA data type conversion functions.

The functions are listed in alphabetical order.

> **Note:**
>
> Functions used exclusively for XLA as a replication mechanism are documented in a separate section, XLA Replication Function Reference.

ttXlaAcknowledge

Description

This function is used to acknowledge that one or more records have been read from the transaction log by the ttXlaNextUpdate or ttXlaNextUpdateWait function.

After you make this call, the bookmark is reset so that you cannot reread any of the previously returned records. Call `ttXlaAcknowledge` only when messages have been completely processed.

> **Note:**
>
> - The bookmark is only reset for the specified handle. Other handles in the system may still be able to access those earlier transactions.
> - The bookmark is reset even if there are no relevant update records to acknowledge.

Note that `ttXlaAcknowledge` is an expensive operation that should be used only as necessary. Calling `ttXlaAcknowledge` more than once per reading of the transaction log file does not reduce the volume of the transaction log since XLA only purges transaction logs a file at a time. To detect when a new transaction log file is generated, you can find out which log file a bookmark is in by examining the `purgeLSN` (represented by the `PURGELSNHIGH` and `PURGELSNLOW` values) for the bookmark in the system table `SYS.TRANSACTION_LOG_API`. You can then call `ttXlaAcknowledge` to purge the old transaction log files. (Note that you must have `ADMIN` or `SELECT ANY TABLE` privilege to view this table.)

The second purpose of `ttXlaAcknowledge` is to ensure that the XLA application does not see the acknowledged records if it were to connect to a previously used bookmark by calling the ttXlaPersistOpen function with the `XLAREUSE` option. If you intend to reuse a bookmark, call `ttXlaAcknowledge` to reset the bookmark position to the current record before calling ttXlaClose.

See Retrieving Update Records From the Transaction Log.

Required Privilege

XLA

Syntax

```
SQLRETURN ttXlaAcknowledge(ttXlaHandle_h handle)
```

Parameters

Parameter	Type	Description
handle	`ttXlaHandle_h`	Transaction log handle

Returns

Returns `SQL_SUCCESS` if call is successful. Otherwise, use ttXlaError to report the error.

Example

```
rc = ttXlaAcknowledge(xlahandle);
```

See Also

ttXlaNextUpdate
ttXlaNextUpdateWait

ttXlaClose

Description

Closes an XLA handle that was opened by ttXlaPersistOpen. See Terminating an XLA Application.

Required Privilege

XLA

Syntax

```
SQLRETURN ttXlaClose(ttXlaHandle_h handle)
```

Parameters

Parameter	Type	Description
handle	`ttXlaHandle_h`	ODBC handle for the database

Returns

Returns `SQL_SUCCESS` if call is successful. Otherwise, use ttXlaError to report the error.

To close the XLA handle opened in the previous example, use the following call:

```
rc = ttXlaClose(xlahandle);
```

See Also

ttXlaPersistOpen

ttXlaConvertCharType

Description

Converts the column data indicated by the *colinfo* and *tup* parameters into the connection character set associated with the transaction log handle and places the result in a buffer.

Required Privilege

XLA

Syntax

```
SQLRETURN ttXlaConvertCharType (ttXlaHandle_h handle,
                                ttXlaColDesc_t*  colinfo,
                                void* tup,
                                void* buf,
                                size_t buflen)
```

Parameters

Parameter	Type	Description
handle	`ttXlaHandle_h`	Transaction log handle for the database
colinfo	ttXlaColDesc_t*	Pointer to the buffer that holds the column descriptions
tup	`void*`	Data to be converted
buf	`void*`	Location where the converted data is placed
buflen	`size_t`	Size of the buffer where the converted data is placed

Returns

Returns SQL_SUCCESS if call is successful. Otherwise, use ttXlaError to report the error.

ttXlaDateToODBCCType

Description

Converts a TTXLA_DATE_TT value to an ODBC C value usable by applications. See Converting Complex Data Types.

Call this function only on a column of data type TTXLA_DATE_TT. The data type can be obtained from the ttXlaColDesc_t structure returned by the ttXlaGetColumnInfo function.

Required Privilege

XLA

Syntax

```
SQLRETURN ttXlaDateToODBCCType(void* fromData,
                               out DATE_STRUCT* returnData)
```

Parameters

Parameter	Type	Description
fromData	void*	Pointer to the date value returned from the transaction log
returnData	DATE_STRUCT*	Pointer to storage allocated to hold the converted date

Returns

Returns SQL_SUCCESS if call is successful. Otherwise, use ttXlaError to report the error.

ttXlaDecimalToCString

Converts a TTXLA_DECIMAL_TT value to a string usable by applications.

Description

The scale and precision values can be obtained from the ttXlaColDesc_t structure returned by the ttXlaGetColumnInfo function. The *scale* parameter specifies the maximum number of digits after the decimal point. If the decimal value is larger than 1, the *precision* parameter should specify the maximum number of digits before and after the decimal point. If the decimal value is less than 1, *precision* equals *scale*.

Call this function only for a column of type TTXLA_DECIMAL_TT. The data type can be obtained from the ttXlaColDesc_t structure returned by the ttXlaGetColumnInfo function.

See Converting Complex Data Types.

Required Privilege

XLA

Syntax

```
SQLRETURN ttXlaDecimalToCString(void* fromData,
                                out char* returnData,
                                SQLSMALLINT precision,
                                SQLSMALLINT scale)
```

Parameters

Parameter	Type	Description
fromData	void*	Pointer to the decimal value returned from the transaction log
returnData	char*	Pointer to storage allocated to hold the converted string
precision	SQLSMALLINT	If *fromData* is greater than 1, the maximum number of digits before and after the decimal point If *fromData* is less than 1, same as scale
scale	SQLSMALLINT	Maximum number of digits after the decimal point

Returns

Returns SQL_SUCCESS if call is successful. Otherwise, use ttXlaError to report the error.

Example

This example assumes you have obtained the *offset*, *precision*, and *scale* values from a ttXlaColDesc_t structure and used the offset to obtain a decimal value, *pColVal*, in a row returned in a transaction log record.

```
char decimalData[50];
static ttXlaColDesc_t colDesc[255];

rc = ttXlaDecimalToCString(pColVal, (char*)&decimalData,
                                    colDesc->precision,
                                    colDesc->scale);
```

ttXlaDeleteBookmark

Description

Deletes the bookmark associated with the specified transaction log handle. After the bookmark has been deleted, it is no longer accessible and its identifier may be reused for another bookmark. The deleted bookmark is no longer associated with the database handle and the effect is the same as having opened the connection with the XLANONE option.

If the bookmark is in use, it cannot be deleted until it is no longer in use.

See Deleting Bookmarks.

> **Note:**
>
> - Do not confuse this with the TimesTen built-in procedure `ttXlaBookmarkDelete`, documented in ttXlaBookmarkDelete in *Oracle TimesTen In-Memory Database Reference*.
> - You cannot delete replicated bookmarks while the replication agent is running.

Required Privilege

XLA

Syntax

```
SQLRETURN ttXlaDeleteBookmark(ttXlaHandle_h handle)
```

Parameters

Parameter	Type	Description
handle	ttXlaHandle_h	Transaction log handle

Returns

Returns `SQL_SUCCESS` if call is successful. Otherwise, use ttXlaError to report the error.

Example

Delete the bookmark for `xlahandle`:

```
rc = ttXlaDeleteBookmark(xlahandle);
```

See Also

ttXlaPersistOpen
ttXlaGetLSN
ttXlaSetLSN

ttXlaError

Description

Reports details of any errors encountered from the previous call on the given transaction log handle. Multiple errors may be returned through subsequent calls to `ttXlaError`. The error stack is cleared following each call to a function other than `ttXlaError` itself and ttXlaErrorRestart.

See "XLA Error Handling" for a discussion about using this function.

Required Privilege

XLA

Syntax

```
SQLRETURN ttXlaError(ttXlaHandle_h handle,
                     out SQLINTEGER* errCode,
                     out char* errMessage,
                     SQLINTEGER maxLen,
                     out SQLINTEGER* retLen)
```

Parameters

Parameter	Type	Description
handle	ttXlaHandle_h	Transaction log handle for the database
errCode	SQLINTEGER*	Code of the error message to be copied into the *errMessage* buffer
errMessage	char*	Buffer to hold the error text
maxLen	SQLINTEGER	Maximum length of the *errMessage* buffer
retLen	SQLINTEGER*	Actual size of the error message

Returns

Returns SQL_SUCCESS if error information is returned, or SQL_NO_DATA_FOUND if no more errors are found in the error stack. If the *errMessage* buffer is not large enough, ttXlaError returns SQL_SUCCESS_WITH_INFO.

> **Note:**
>
> SQL_NO_DATA_FOUND is defined in sqlext.h, which is included by timesten.h.

Example

There can be multiple errors on the error stack. This example shows how to read them all.

```
char message[100];
SQLINTEGER code;

for (;;) {
  rc = ttXlaError(xlahandle, &code, message, sizeof (message), &retLen);
  if (rc == SQL_NO_DATA_FOUND)
      break;
  if (rc == SQL_ERROR) {
      printf("Error in fetching error message\n");
      break;
  }
  else {
      printf("Error code %d: %s\n", code, message);
  }
}
```

Note

If you use multiple threads to access a TimesTen transaction log over a single XLA connection, TimesTen creates a latch to control concurrent access. If for some reason the latch cannot be acquired by a thread, the XLA function returns `SQL_INVALID_HANDLE`.

See Also

ttXlaErrorRestart

ttXlaErrorRestart

Description

Resets the error stack so that an application can reread the errors. See XLA Error Handling.

Required Privilege

`XLA`

Syntax

```
SQLRETURN ttXlaErrorRestart(ttXlaHandle_h handle)
```

Parameters

Parameter	Type	Description
handle	`ttXlaHandle_h`	Transaction log handle for the database

Returns

Returns `SQL_SUCCESS` if call is successful. Otherwise, use ttXlaError to report the error.

Example

```
rc = ttXlaErrorRestart(xlahandle);
```

See Also

ttXlaError

ttXlaGetColumnInfo

Description

Retrieves information about all the columns in the table. The output parameter for number of columns returned, *nreturned*, is set to the number of columns returned in *colinfo*. The *systemTableID* or *userTableID* parameter describes the desired table. This call is serialized with respect to changes in the table definition.

See Obtaining Column Descriptions.

Required Privilege

XLA

Syntax

```
SQLRETURN ttXlaGetColumnInfo(ttXlaHandle_h handle,
                             SQLUBIGINT systemTableID,
                             SQLUBIGINT userTableID,
                             out ttXlaColDesc_t* colinfo,
                             SQLINTEGER maxcols,
                             out SQLINTEGER* nreturned)
```

Parameters

Parameter	Type	Description
handle	`ttXlaHandle_h`	Transaction log handle for the database
systemTableID	`SQLUBIGINT`	System ID of table
userTableID	`SQLUBIGINT`	User ID of table
colinfo	ttXlaColDesc_t*	Pointer to the buffer large enough to hold a separate description for *maxcols* columns
maxcols	`SQLINTEGER`	Maximum number of columns that can be stored in the *colInfo* buffer If the table contains more than *maxcols* columns, an error is returned.
nreturned	`SQLINTEGER*`	Number of columns returned

Returns

Returns `SQL_SUCCESS` if call is successful. Otherwise, use ttXlaError to report the error.

Example

For this example, assume the following definitions:

```
ttXlaColDesc_t colinfo[20];
SQLUBIGINT systemTableID, userTableID;
SQLINTEGER ncols;
```

To get the description of up to 20 columns using the system table identifier, issue the following call.

```
rc = ttXlaGetColumnInfo(xlahandle, systemTableID, 0, colinfo, 20, &ncols);
```

Likewise, the user table identifier can be used:

```
rc = ttXlaGetColumnInfo(xlahandle, 0, userTableID, colinfo, 20, &ncols);
```

See ttXlaColDesc_t for details and an example on how to access the column data in a returned row.

See Also

ttXlaGetTableInfo
ttXlaDecimalToCString
ttXlaDateToODBCCType
ttXlaTimeToODBCCType
ttXlaTimeStampToODBCCType

ttXlaGetLSN

Description

Returns the Current Read log record identifier for the connection specified by the transaction log handle. See How Bookmarks Work.

Required Privilege

XLA

Syntax

```
SQLRETURN ttXlaGetLSN(ttXlaHandle_h handle,
                      out tt_XlaLsn_t* LSN)
```

Parameters

Parameter	Type	Description
handle	ttXlaHandle_h	Transaction log handle for the database
LSN	tt_XlaLsn_t*	Current Read log record identifier for the handle

> **Note:**
>
> Be aware that tt_XlaLsn_t, particularly the *logFile* and *logOffset* fields, is used differently than in earlier releases, referring to log record identifiers rather than sequentially increasing LSNs. See the note in tt_XlaLsn_t.

Returns

Returns SQL_SUCCESS if call is successful. Otherwise, use ttXlaError to report the error.

Example

This example returns the Current Read log record identifier, CurLSN.

```
tt_XlaLsn_t CurLSN;

rc = ttXlaGetLSN(xlahandle, &CurLSN);
```

See Also

ttXlaSetLSN

ttXlaGetTableInfo

Description

Retrieves information about the rows in the table (refer to the description of the ttXlaTblDesc_t data type.) If the *userTableID* parameter is nonzero, then it is used to locate the desired table. Otherwise, the *systemTableID* value is used to locate the table. If both are zero, an error is returned. The description is stored in the output parameter *tblinfo*. This call is serialized with respect to changes in the table definition.

Required Privilege

XLA

Syntax

```
SQLRETURN ttXlaGetTableInfo(ttXlaHandle_h handle,
                            SQLUBIGINT systemTableID,
                            SQLUBIGINT userTableID,
                            out ttXlaTblDesc_t* tblinfo)
```

Parameters

Parameter	Type	Description
handle	ttXlaHandle_h	Transaction log handle for the database
systemTableID	SQLUBIGINT	System table ID
userTableID	SQLUBIGINT	User table ID
tblinfo	ttXlaTblDesc_t*	Row information

Returns

Returns SQL_SUCCESS if call is successful. Otherwise, use ttXlaError to report the error.

Example

For this example, assume the following definitions:

```
ttXlaTblDesc_t tabinfo;
SQLUBIGINT systemTableID, userTableID;
```

To get table information using a system identifier, find the system table identifier using ttXlaTableByName or other means and issue the following call:

```
rc = ttXlaGetTableInfo(xlahandle, systemTableID, 0, &tabinfo);
```

Alternatively, the table information can be retrieved using a user table identifier:

```
rc = ttXlaGetTableInfo(xlahandle, 0, userTableID, &tabinfo);
```

See Also

ttXlaGetColumnInfo

ttXlaGetVersion

Description

This function is used in combination with ttXlaSetVersion to ensure XLA applications written for older versions of XLA operate on a new version. The configured version is typically the older version, while the actual version is the newer one.

The function retrieves the currently configured XLA version and stores it into *configuredVersion* parameter. The actual version of the underlying XLA is stored in *actualVersion*. Due to calls on `ttXlaSetVersion`, the results in *configuredVersion* may vary from one call to the next, but the results in *actualVersion* remain the same.

Required Privilege

XLA

Syntax

```
SQLRETURN ttXlaGetVersion(ttXlaHandle_h handle,
                          out ttXlaVersion_t* configuredVersion,
                          out ttXlaVersion_t* actualVersion)
```

Parameters

Parameter	Type	Description
handle	`ttXlaHandle_h`	Transaction log handle for the database
configuredVersion	ttXlaVersion_t*	Configured version of XLA
actualVersion	`ttXlaVersion_t*`	Actual version of XLA

Returns

Returns `SQL_SUCCESS` if call is successful. Otherwise, use ttXlaError to report the error.

Example

Assume the following directions for this example:

```
ttXlaVersion_t configured, actual;
```

To determine the current version configuration, use the following call:

```
rc = ttXlaGetVersion(xlahandle, &configured, &actual);
```

See Also

ttXlaVersionCompare
ttXlaSetVersion

ttXlaNextUpdate

Description

This function fetches up to a specified maximum number of update records from the transaction log and returns the records associated with committed transactions to a specified buffer. The actual number of returned records is reported in the *nreturned* output parameter. This function requires a bookmark to be present in the database and to be associated with the connection used by the function.

Each call to `ttXlaNextUpdate` resets the bookmark to the last record read to enable the next call to `ttXlaNextUpdate` to return the next list of records.

See Retrieving Update Records From the Transaction Log.

Required Privilege

XLA

Syntax

```
SQLRETURN ttXlaNextUpdate(ttXlaHandle_h handle,
                          out ttXlaUpdateDesc_t*** records,
                          SQLINTEGER maxrecords,
                          out SQLINTEGER* nreturned)
```

Parameters

Parameter	Type	Description
handle	`ttXlaHandle_h`	Transaction log handle for the database
records	ttXlaUpdateDesc_t***	Buffer to hold the completed transaction records
maxrecords	`SQLINTEGER`	Maximum number of records to be fetched
nreturned	`SQLINTEGER*`	Actual number of returned records, where 0 is returned if no update data is available

Returns

Returns `SQL_SUCCESS` if call is successful. Otherwise, use ttXlaError to report the error.

Example

This example retrieves up to 100 records and describes a loop in which each record can be processed:

```
ttXlaUpdateDesc_t** records;
SQLINTEGER nreturned;
SQLINTEGER i;

rc = ttXlaNextUpdate(xlahandle, &records, 100, &nreturned);
/* Check for errors; if none, process the records */
for (i = 0; i < nreturned; i++) {
  process(records[i]);
}
```

Notes

Updates are generated for all data definition statements, regardless of tracking status. Updates are generated for data update operations for all tracked tables associated with the bookmark.

In addition, updates are generated for certain special operations, including assigning application-level identifiers for tables and columns and changing the tracking status of a table.

See Also

ttXlaNextUpdateWait
ttXlaAcknowledge

ttXlaNextUpdateWait

Description

This is similar to the ttXlaNextUpdate function, with the addition of a *seconds* parameter that specifies the number of seconds to wait if no records are available in the transaction log. The actual number of seconds of wait time can be up to two seconds more than the specified *seconds* value.

Also see Retrieving Update Records From the Transaction Log.

Required Privilege

XLA

Syntax

```
SQLRETURN ttXlaNextUpdateWait(ttXlaHandle_h handle,
                              out ttXlaUpdateDesc_t*** records,
                              SQLINTEGER maxrecords,
                              out SQLINTEGER* nreturned,
                              SQLINTEGER seconds)
```

Parameters

Parameter	Type	Description
handle	ttXlaHandle_h	Transaction log handle for the database
records	ttXlaUpdateDesc_t***	Buffer to hold completed transaction records
maxrecords	SQLINTEGER	Maximum number of records to be fetched **Note**: The largest effective value is 1000 records.
nreturned	SQLINTEGER*	Actual number of records returned, where 0 is returned if no update data is available within the seconds wait period
seconds	SQLINTEGER	Number of seconds to wait if the log is empty

Returns

Returns SQL_SUCCESS if call is successful. Otherwise, use ttXlaError to report the error.

Example

This example retrieves up to 100 records and waits for up to 60 seconds if there are no records available in the transaction log.

```
ttXlaUpdateDesc_t** records;
SQLINTEGER nreturned;
SQLINTEGER i;

rc = ttXlaNextUpdateWait(xlahandle, &records, 100, &nreturned, 60);
/* Check for errors; if none, process the records */
for (i = 0; i < nreturned; i++) {
  process(records[i]);
}
```

See Also

ttXlaNextUpdate
ttXlaAcknowledge

ttXlaNumberToBigInt

Description

Converts a TTXLA_NUMBER value to a SQLBIGINT value usable by an application.

Call this function only for a column of type TTXLA_NUMBER. The data type can be obtained from the ttXlaColDesc_t structure returned by the ttXlaGetColumnInfo function.

Required Privilege

XLA

Syntax

```
SQLRETURN ttXlaNumberToBigInt(void* fromData,
                              SQLBIGINT* bint)
```

Parameters

Parameter	Type	Description
fromData	void*	Pointer to the number value returned from the transaction log
bint	SQLBIGINT*	The SQLBIGINT value converted from the XLA number value

Returns

Returns SQL_SUCCESS if call is successful. Otherwise, use ttXlaError to report the error.

ttXlaNumberToCString

Description

Converts a TTXLA_NUMBER value to a character string usable by an application.

Call this function only for a column of type TTXLA_NUMBER. The data type can be obtained from the ttXlaColDesc_t structure returned by the ttXlaGetColumnInfo function.

Required Privilege

XLA

Syntax

```
SQLRETURN ttXlaNumberToCString(ttXlaHandle_h handle,
                               void* fromData,
                               char* buf,
                               int buflen
                               int* reslen)
```

Parameters

Parameter	Type	Description
fromData	void*	Pointer to the number value returned from the transaction log
buf	char*	Location where the converted data is placed
buflen	int	Size of the buffer where the converted data is placed
reslen	int*	Number of bytes that were written, assuming *buflen* is large enough (otherwise, the number of bytes that would have been written)

Returns

Returns SQL_SUCCESS if call is successful. Otherwise, use ttXlaError to report the error.

ttXlaNumberToDouble

Description

Converts a TTXLA_NUMBER value to a long floating point number value usable by applications.

Call this function only for a column of type TTXLA_NUMBER. The data type can be obtained from the ttXlaColDesc_t structure returned by the ttXlaGetColumnInfo function.

Required Privilege

XLA

Syntax

```
SQLRETURN ttXlaNumberToDouble(void* fromData,
                              double* dbl)
```

Parameters

Parameter	Type	Description
fromData	void*	Pointer to the number value returned from the transaction log
dbl	double*	The long floating point number value converted from the XLA number value

Returns

Returns `SQL_SUCCESS` if call is successful. Otherwise, use ttXlaError to report the error.

ttXlaNumberToInt

Description

Converts a `TTXLA_NUMBER` value to a `SQLINTEGER` value usable by an application.

Call this function only for a column of type `TTXLA_NUMBER`. The data type can be obtained from the ttXlaColDesc_t structure returned by the ttXlaGetColumnInfo function.

Required Privilege

XLA

Syntax

```
SQLRETURN ttXlaNumberToInt(void* fromData,
                           SQLINTEGER* ival)
```

Parameters

Parameter	Type	Description
fromData	void*	Pointer to the number value returned from the transaction log
ival	SQLINTEGER*	The `SQLINTEGER` value converted from the XLA number value

Returns

Returns `SQL_SUCCESS` if call is successful. Otherwise, use ttXlaError to report the error.

ttXlaNumberToSmallInt

Description

Converts a TTXLA_NUMBER value to a SQLSMALLINT value usable by an application.

Call this function only for a column of type TTXLA_NUMBER. The data type can be obtained from the ttXlaColDesc_t structure returned by the ttXlaGetColumnInfo function.

Required Privilege

XLA

Syntax

```
SQLRETURN ttXlaNumberToSmallInt(void* fromData,
                                SQLSMALLINT* smint)
```

Parameters

Parameter	Type	Description
fromData	void*	Pointer to the number value returned from the transaction log
smint	SQLSMALLINT*	The SQLSMALLINT value converted from the XLA number value

Returns

Returns SQL_SUCCESS if call is successful. Otherwise, use ttXlaError to report the error.

ttXlaNumberToTinyInt

Description

Converts a TTXLA_NUMBER value to a tiny integer value usable by an application.

Call this function only for a column of type TTXLA_NUMBER. The data type can be obtained from the ttXlaColDesc_t structure returned by the ttXlaGetColumnInfo function.

Required Privilege

XLA

Syntax

```
SQLRETURN ttXlaNumberToTinyInt(void* fromData,
                               SQLCHAR* tiny)
```

Parameters

Parameter	Type	Description
fromData	void*	Pointer to the number value returned from the transaction log
tiny	SQLCHAR*	The tiny integer value converted from the XLA number value

Returns

Returns SQL_SUCCESS if call is successful. Otherwise, use ttXlaError to report the error.

ttXlaNumberToUInt

Description

Converts a TTXLA_NUMBER value to an unsigned integer value usable by an application.

Call this function only for a column of type TTXLA_NUMBER. The data type can be obtained from the ttXlaColDesc_t structure returned by the ttXlaGetColumnInfo function.

Required Privilege

XLA

Syntax

```
SQLRETURN ttXlaNumberToInt(void* fromData,
                           SQLUINTEGER* ival)
```

Parameters

Parameter	Type	Description
fromData	void*	Pointer to the number value returned from the transaction log
ival	SQLUINTEGER*	The integer value converted from the XLA number value

Returns

Returns SQL_SUCCESS if call is successful. Otherwise, use ttXlaError to report the error.

ttXlaOraDateToODBCTimeStamp

Description

Converts a TTXLA_DATE value to an ODBC timestamp.

Call this function only for a column of type TTXLA_DATE. The data type can be obtained from the ttXlaColDesc_t structure returned by the ttXlaGetColumnInfo function.

Required Privilege

XLA

Syntax

```
SQLRETURN ttXlaOraDateToODBCTimeStamp(void* fromData,
                                      TIMESTAMP_STRUCT* returnData)
```

Parameters

Parameter	Type	Description
fromData	void*	Pointer to the number value returned from the transaction log
returnData	TIMESTAMP_STRUCT*	ODBC timestamp value converted from the XLA Oracle Database DATE value

Returns

Returns SQL_SUCCESS if call is successful. Otherwise, use ttXlaError to report the error.

ttXlaOraTimeStampToODBCTimeStamp

Description

Converts a TTXLA_TIMESTAMP value to an ODBC timestamp.

Call this function only for a column of type TTXLA_TIMESTAMP. The data type can be obtained from the ttXlaColDesc_t structure returned by the ttXlaGetColumnInfo function.

Syntax

```
SQLRETURN ttXlaOraTimeStampToODBCTimeStamp(void* fromData,
                                           TIMESTAMP_STRUCT*
returnData)
```

Required Privilege

XLA

Parameters

Parameter	Type	Description
fromData	void*	Pointer to the number value returned from the transaction log
returnData	TIMESTAMP_STRUCT*	ODBC timestamp value converted from the XLA Oracle Database TIMESTAMP value

Returns

Returns SQL_SUCCESS if call is successful. Otherwise, use ttXlaError to report the error.

ttXlaPersistOpen

Description

Initializes a transaction log handle to a database to enable access to the transaction log. The *hdbc* parameter is an ODBC connection handle to a database. Create only one XLA handle for each ODBC connection. After you have created an XLA handle on an ODBC connection, do not issue any other ODBC calls over the ODBC connection until it is closed by ttXlaClose.

The *tag* is a string that identifies the XLA bookmark (see About XLA Bookmarks). The *tag* can identify a new bookmark, either non-replicated or replicated, or one that exists in the system, as specified by the *options* parameter. The *handle* parameter is initialized by this call and must be provided on each subsequent call to XLA.

Some actions can be done without a bookmark. When performing these types of actions, you can use the XLANONE option to access the transaction log without a bookmark. Actions that *cannot* be done without a bookmark are the following:

- ttXlaAcknowledge
- ttXlaGetLSN
- ttXlaSetLSN
- ttXlaNextUpdate
- ttXlaNextUpdateWait

Multiple applications can concurrently read from the transaction log. See Initializing XLA and Obtaining an XLA Handle.

When this function is successful, XLA sets the autocommit mode to off.

If this function fails but still creates a handle, the handle must be closed to prevent memory leaks.

> **Note:**
>
> Space is allocated by this call. Call ttXlaClose to free space when you are finished.

Required Privilege

XLA

Syntax

```
SQLRETURN ttXlaPersistOpen(SQLHDBC hdbc,
                           SQLCHAR* tag,
                           SQLUINTEGER options,
                           out ttXlaHandle_h* handle)
```

Parameters

Parameter	Type	Description
hdbc	SQLHDBC	ODBC handle for the database
tag	SQLCHAR*	Identifier for the XLA bookmark This can be null, in which case *options* should be set to XLANONE. Maximum allowed length is 31.
options	SQLUINTEGER	Bookmark options: • XLANONE: Connect without a bookmark. The *tag* field is ignored. • XLACREAT: Create a new non-replicated bookmark. Fails if a bookmark already exists. • XLAREPL: Create a new replicated bookmark. Fails if a bookmark already exists. • XLAREUSE: Associate with an existing bookmark (non-replicated or replicated). Fails if the bookmark does not exist.
handle	ttXlaHandle_h*	Transaction log handle returned by this call

Returns

Returns SQL_SUCCESS or SQL_SUCCESS_WITH_INFO if call is successful. Otherwise, call SQLError on the HDBC connection handle that was passed in as an argument.

Example

This example opens a transaction log, returns a handle named xlahandle, and creates a new non-replicated bookmark named mybookmark:

```
SQLHDBC hdbc;
ttXlaHandle_h xlahandle;

rc = ttXlaPersistOpen(hdbc, ( SQLCHAR*)mybookmark,
                      XLACREAT, &xlahandle);
```

Alternatively, create a new replicated bookmark as follows:

```
SQLHDBC hdbc;
ttXlaHandle_h xlahandle;

rc = ttXlaPersistOpen(hdbc, ( SQLCHAR*)mybookmark,
                      XLAREPL, &xlahandle);
```

Note

Multithreaded applications should create a separate XLA handle for each thread. If multiple threads must use the same XLA handle, use a mutex to serialize thread access to that XLA handle so that only one thread can execute an XLA operation at a time.

See Also

ttXlaClose
ttXlaDeleteBookmark
ttXlaGetLSN
ttXlaSetLSN

ttXlaRowidToCString

Description

Converts a `ROWID` value to a string value usable by applications.

Required Privilege

`XLA`

Syntax

```
SQLRETURN ttXlaRowidToCString(void* fromData, char* buf, int buflen)
```

Parameters

Parameter	Type	Description
fromData	`void*`	Pointer to the `ROWID` value returned from the transaction log
buf	`char*`	Pointer to storage allocated to hold the converted string
buflen	`int`	Length of the converted string

Returns

Returns `SQL_SUCCESS` if call is successful. Otherwise, use ttXlaError to report the error.

Example

```
char charbuf[18];
void* rowiddata;
/* ... */
rc = ttXlaRowidToCString(rowiddata, charbuf, sizeof(charbuf));
```

ttXlaSetLSN

Description

Sets the Current Read log record identifier for the database specified by the transaction handle. The specified *LSN* value should be returned from `ttXlaGetLSN`. It cannot be a user-created value and cannot be earlier than the current bookmark Initial Read log record identifier.

See "About XLA Bookmarks" for a discussion about using this function.

Required Privilege

XLA

Syntax

```
SQLRETURN ttXlaSetLSN(ttXlaHandle_h handle,
                      tt_XlaLsn_t* LSN)
```

Parameters

Parameter	Type	Description
handle	ttXlaHandle_h	Transaction log handle for the database
LSN	tt_XlaLsn_t*	New log record identifier for the handle

Note:

Be aware that tt_XlaLsn_t, particularly the *logFile* and *logOffset* fields, is used differently than in earlier releases, referring to log record identifiers rather than sequentially increasing LSNs. See the note in tt_XlaLsn_t.

Returns

Returns SQL_SUCCESS if call is successful. Otherwise, use ttXlaError to report the error.

Example

This example sets the Current Read log record identifier to CurLSN.

```
tt_XlaLsn_t CurLSN;

rc = ttXlaSetLSN(xlahandle, &CurLSN);
```

See Also

ttXlaGetLSN

ttXlaSetVersion

Description

Sets the version of XLA to be used by the application. This version must be either the same as the version received from ttXlaGetVersion or from an earlier version.

Required Privilege

XLA

Syntax

```
SQLRETURN ttXlaSetVersion(ttXlaHandle_h handle,
                          ttXlaVersion_t* version)
```

Parameters

Parameter	Type	Description
handle	ttXlaHandle_h	Transaction log handle for the database
version	ttXlaVersion_t*	Desired version of XLA

Returns

Returns `SQL_SUCCESS` if call is successful. Otherwise, use ttXlaError to report the error.

Example

To set the configured version to the value specified in `requestedVersion`, issue the following call:

```
rc = ttXlaSetVersion(xlahandle, &requestedVersion);
```

See Also

ttXlaVersionCompare
ttXlaGetVersion

ttXlaTableByName

Description

Finds the system and user table identifiers for a table or materialized view by providing the owner and name of the table or view. See Specifying Which Tables to Monitor for Updates.

Required Privilege

XLA

Syntax

```
SQLRETURN ttXlaTableByName(ttXlaHandle_h handle,
                           char* owner,
                           char* name,
                           out SQLUBIGINT* sysTableID,
                           out SQLUBIGINT* userTableID)
```

Parameters

Parameter	Type	Description
handle	ttXlaHandle_h	Transaction log handle for the database

Parameter	Type	Description
owner	char*	Owner for the table or view as a string
name	char*	Name of the table or view
sysTableID	SQLUBIGINT*	System table ID
userTableID	SQLUBIGINT*	User table ID

Returns

Returns SQL_SUCCESS if call is successful. Otherwise, use ttXlaError to report the error.

Example

To get the system and user table IDs associated with the table PURCHASING.INVOICES, use the following call:

```
SQLUBIGINT sysTableID;
SQLUBIGINT userTableID;

rc = ttXlaTableByName(xlahandle, "PURCHASING", "INVOICES",
                      &sysTableID, &userTableID);
```

See Also

ttXlaTableStatus

ttXlaTableStatus

Description

Returns the update status for a table. Identify the table by specifying either a user ID (*userTableID*) or a system ID (*systemTableID*). If *userTableID* is nonzero, it is used to locate the table. Otherwise *systemTableID* is used. If both are zero, an error is returned.

Specifying a value for *newstatus* sets the update status to **newstatus*. A nonzero status means the table specified by *systemTableID* is available through XLA. Zero means the table is not tracked. Changes to table update status are effective immediately.

Updates to a table are tracked only if update tracking was enabled for the table at the time the update was performed. This call is serialized with respect to updates to the underlying table. Therefore, transactions that update the table run either completely before or completely after the change to table status.

To use ttXlaTableStatus, the user must be connected to a bookmark. The function reports inserts, updates, and deletes only to the bookmark that has subscribed to the table. It reports DDL events to all bookmarks. DDL events include CREATAB, DROPTAB, CREAIND, DROPIND, CREATVIEW, DROPVIEW, CREATSEQ, DROPSEQ, CREATSYN, DROPSYN, ADDCOLS, DRPCOLS, TRUNCATE, SETTBL1, and SETCOL1 transactions. See ttXlaUpdateDesc_t.

See "Specifying Which Tables to Monitor for Updates" for a discussion about using this function.

> **Note:**
>
> DML updates to a table being tracked through XLA do not prevent `ttXlaTableStatus` from running. However, DDL updates to the table being tracked, which take a lock on `SYS.TABLES`, do delay `ttXlaTableStatus` from running in serializable isolation against `SYS.TABLES`.

Required Privilege

XLA

Syntax

```
SQLRETURN ttXlaTableStatus(ttXlaHandle_h handle,
                           SQLUBIGINT systemTableID,
                           SQLUBIGINT userTableID,
                           out SQLINTEGER* oldstatus,
                           SQLINTEGER* newstatus)
```

Parameters

Parameter	Type	Description
handle	`ttXlaHandle_h`	Transaction log handle for the database
systemTableID	`SQLUBIGINT`	System ID of table
userTableID	`SQLUBIGINT`	User ID of table
oldstatus	`SQLINTEGER*`	XLA old status: • 1: On • 0: Off
newstatus	`SQLINTEGER*`	XLA new status: • 1: On • 0: Off

Returns

Returns `SQL_SUCCESS` if call is successful. Otherwise, use ttXlaError to report the error.

Example

The following examples assume that the system or user table identifiers are found using ttXlaTableByName or some other means.

Assume these declarations for the example:

```
SQLUBIGINT systemTableID;
SQLUBIGINT userTableID;
SQLINTEGER currentStatus, requestedStatus;
```

To find the status of a table given its system table identifier, use the following call:

```
/* Get system table identifier into systemTableID, then ... */
rc = ttXlaTableStatus(xlahandle, systemTableID, 0,
                      &currentStatus, NULL);
```

The *currentStatus* value is nonzero if update tracking for the table is enabled, or zero otherwise.

To enable update tracking for a table given a system table identifier, set the requested status to 1 as follows:

```
requestedStatus = 1;

rc = ttXlaTableStatus(xlahandle, systemTableID, 0,
                      NULL, &requestedStatus);
```

You can set a new update tracking status and retrieve the current status in a single call, as in the following example:

```
requestedStatus = 1;

rc = ttXlaTableStatus(xlahandle, systemTableID, 0,
                      &currentStatus, &requestedStatus);
```

The above call enables update tracking for a table by system table identifier and retrieves the prior update tracking status in the variable *currentStatus*.

All of these examples can be done using user table identifiers as well. To retrieve the update tracking status of a table through its user table identifier, use the following call:

```
/* Get system table identifier into userTableID, then ... */

rc = ttXlaTableStatus(xlahandle, 0, userTableID,
                      &currentStatus, NULL);
```

See Also

ttXlaTableByName

ttXlaTableVersionVerify

Description

Verifies that the cached table definitions are compatible with the XLA record being processed. Table definitions change only when the `ALTER TABLE` statement is used to add or remove columns.

You can monitor the XLA stream for XLA records of transaction type `ADDCOLS` and `DRPCOLS` to avoid the overhead of using this function. When an XLA record of transaction type `ADDCOLS` or `DROPCOLS` is encountered, refresh the table and column definitions. See Inspecting Record Headers and Locating Row Addresses for information about monitoring XLA records for transaction type.

Required Privilege

XLA

Syntax

```
SQLRETURN ttXlaTableVersionVerify(ttXlaHandle_h handle
                                  ttXlaTblVerDesc_t* table,
                                  ttXlaUpdateDesc_t* record
                                  out SQLINTEGER* compat)
```

Parameters

Parameter	Type	Description
handle	ttXlaHandle_h	Transaction log handle for the database
table	ttXlaTblVerDesc_t*	A cached table description
record	ttXlaUpdateDesc_t*	XLA record to be processed
compat	SQLINTEGER*	Compatibility information: • 1: Tables are compatible. • 0: Tables are not compatible.

Returns

Returns SQL_SUCCESS if cached table definition is compatible with the XLA record being processed. Otherwise, use ttXlaError to report the error.

Example

This example checks the compatibility of a table.

```
SQLINTEGER compat;
ttXlaTbVerDesc_t table;
ttXlaUpdateDesc_t* record;
/*
 * Get the desired table definitions into the variable "table"
 */
rc = ttXlaTableVersionVerify(xlahandle, &table, record, &compat);
if (compat) {
/*
 * Compatible
 */
}
else {
/*
 * Not compatible or some other error occurred
 * If not compatible, issue a call to ttXlaVersionTableInfo and
 * ttXlaVersionColumnInfo to get the new definition.
 */
}
```

See Also

ttXlaVersionColumnInfo
ttXlaVersionTableInfo

ttXlaTimeToODBCCType

Description

Converts a TTXLA_TIME value to an ODBC C value usable by applications. See Converting Complex Data Types for a discussion about using this function.

Call this function only for a column of type TTXLA_TIME. The data type can be obtained from the ttXlaColDesc_t structure returned by the ttXlaGetColumnInfo function.

Required Privilege

XLA

Syntax

```
SQLRETURN ttXlaTimeToODBCCType (void* fromData,
                                out TIME_STRUCT* returnData)
```

Parameters

Parameter	Type	Description
fromData	void*	Pointer to the time value returned from the transaction log
returnData	TIME_STRUCT*	Pointer to storage allocated to hold the converted time

Returns

Returns SQL_SUCCESS if call is successful. Otherwise, use ttXlaError to report the error.

Example

This example assumes you have used the *offset* value returned in a ttXlaColDesc_t structure to obtain a time value, *pColVal*, from a row returned in a transaction log record.

```
TIME_STRUCT time;

rc = ttXlaTimeToODBCCType(pColVal, &time);
```

ttXlaTimeStampToODBCCType

Description

Converts a TTXLA_TIMSTAMP_TT value to an ODBC C value usable by applications. See Converting Complex Data Types for a discussion about using this function.

Call this function only for a column of type TTXLA_TIMSTAMP_TT. The data type can be obtained from the ttXlaColDesc_t structure returned by the ttXlaGetColumnInfo function.

Required Privilege

XLA

Syntax

```
SQLRETURN ttXlaTimeStampToODBCCType(void* fromData,
                                    out TIMESTAMP_STRUCT* returnData)
```

Parameters

Parameter	Type	Description
fromData	void*	Pointer to the timestamp value returned from the transaction log
returnData	TIMESTAMP_STRUCT*	Pointer to storage allocated to hold the converted timestamp

Returns

Returns `SQL_SUCCESS` if call is successful. Otherwise, use ttXlaError to report the error.

Example

This example assumes you have used the *offset* value returned in a ttXlaColDesc_t structure to obtain a timestamp value, *pColVal*, from a row returned in a transaction log record.

```
TIMESTAMP_STRUCT timestamp;

rc = ttXlaTimeStampToODBCCType(pColVal, &timestamp);
```

ttXlaVersionColumnInfo

Description

Retrieves information about the columns in a table for which a change update XLA record must be processed.

Required privilege

XLA

Syntax

```
SQLRETURN ttXlaVersionColumnInfo(ttXlaHandle_h handle,
                                 ttXlaUpdateDesc_t* record,
                                 out ttXlaColDesc_t* colinfo,
                                 SQLINTEGER maxcols,
                                 out SQLINTEGER* nreturned)
```

Parameters

Parameter	Type	Description
handle	ttXlaHandle_h	Transaction log handle for the database
record	ttXlaUpdateDesc_t*	XLA record to be processed
colinfo	ttXlaColDesc_t*	A pointer to the buffer large enough to hold a description for *maxcols* columns
maxcols	SQLINTEGER	Maximum number of columns the table can have **Note**: If the table contains more than *maxcols* columns, an error is returned.

Parameter	Type	Description
nreturned	SQLINTEGER*	Number of columns returned

Returns

Returns `SQL_SUCCESS` if call is successful. Otherwise, use ttXlaError to report the error.

Example

For this example, assume the following definitions:

```
ttXlaHandle_h xlahandle
ttXlaUpdateDesc_t* record;
ttXlaColDesc_t colinfo[20];
SQLINTEGER ncols;
```

The following call retrieves the description of up to 20 columns:

```
rc = ttXlaVersionColumnInfo(xlahandle, record, colinfo, 20, &ncols);
```

ttXlaVersionCompare

Description

Compares two XLA versions and returns a result indicating either that the versions are the same, or which version is earlier.

Required Privilege

XLA

Syntax

```
SQLRETURN ttXlaVersionCompare(ttXlaHandle_h handle,
                              ttXlaVersion_t* version1,
                              ttXlaVersion_t* version2,
                              out SQLINTEGER* comparison)
```

Parameters

Parameter	Type	Description
handle	ttXlaHandle_h	Transaction log handle for the database
version1	ttXlaVersion_t*	Version of XLA to compare with *version2*
version2	ttXlaVersion_t*	Version of XLA to compare with *version1*
comparison	SQLINTEGER*	Comparison result: • 0: Indicates *version1* and *version2* match. • -1: Indicates *version1* is earlier than *version2*. • +1: Indicates *version1* is later than *version2*.

Returns

Returns `SQL_SUCCESS` if call is successful. Otherwise, use ttXlaError to report the error.

Example

To compare the `configured` version against the `actual` version of XLA, issue the following call:

```
ttXlaVersion_t configured, actual;
SQLINTEGER comparision;

rc = ttXlaGetVersion (xlahandle, &configured, &actual);
rc = ttXlaVersionCompare (xlahandle, &configured, &actual,
                          &comparison);
```

Notes

When connecting two systems with XLA-based replication, use the following protocol.

1. At the primary site, retrieve the XLA version using `ttXlaGetVersion`. Send this version information to the standby site.
2. At the standby site, retrieve the XLA version using `ttXlaGetVersion`. Use `ttXlaVersionCompare` to determine which version is earlier. The earlier version number must be used to ensure proper operation between the two sites. Use `ttXlaSetVersion` to specify the version of the interface to use at the standby site. Send the earlier version number back to the primary site.
3. When the chosen version is received at the primary site, use `ttXlaSetVersion` to specify the version of XLA to use.

See Also

ttXlaGetVersion
ttXlaSetVersion

ttXlaVersionTableInfo

Description

Retrieves the table definition for the change update record that must be processed. The table description is stored in the *`tableinfo`* output parameter.

Required Privilege

`XLA`

Syntax

```
SQLRETURN ttXlaVersionTableInfo(ttXlaHandle_h handle,
                                ttXlaUpdateDesc_t* record,
                                out ttXlaTblVerDesc_t* tblinfo)
```

Parameters

Parameter	Type	Description
handle	ttXlaHandle_h	Transaction log handle for the database
record	ttXlaUpdateDesc_t*	XLA record to be processed
tableinfo	ttXlaTblVerDesc_t*	Information about table definition

Returns

Returns `SQL_SUCCESS` if call is successful. Otherwise, use ttXlaError to report the error.

Example

For this example, assume the following definitions:

```
ttXlaHandle_h xlahandle;
ttXlaUpdateDesc_t* record;
ttXlaTblVerDesc_t tabinfo;
```

The following call retrieves a table definition:

```
rc = ttXlaVersionTableInfo(xlahandle, record, &tabinfo);
```

XLA Replication Function Reference

TimesTen replication as described in Overview of TimesTen Replication in *Oracle TimesTen In-Memory Database Replication Guide* is sufficient for most customer needs. However, it is also possible to use XLA functions to replicate updates from one database to another. Implementing your own replication scheme on top of XLA in this way is fairly complicated, but can be considered if TimesTen replication is not feasible for some reason.
This section documents the functions that are exclusive to using XLA as a replication mechanism. Functions are listed in alphabetical order.

ttXlaApply

This function is part of XLA replication functionality and is not appropriate for use in a typical XLA application.

Description

Applies an update to the database associated with the transaction log handle. The return value indicates whether the update was successful. The return also shows if the update encountered a persistent problem. (To see whether the update encountered a transient problem such as a deadlock or timeout, you must call ttXlaError and check the error code.)

If the ttXlaUpdateDesc_t record is a transaction commit, the underlying database transaction is committed. No other transaction commits are performed by `ttXlaApply`. If the parameter *test* (see syntax below) is true, the "old values" in the update description are compared against the current contents of the database for record updates and deletions. If the old value in the update description does not match the corresponding row in the database, this function rejects the update and returns an `sb_ErrXlaTupleMismatch` error.

See Using XLA as a Replication Mechanism.

> **Note:**
>
> `ttXlaApply` cannot be used if the table definition was updated since it was originally written to the transaction log. Unique key and foreign key constraints are checked at the row level rather than at the statement level.

Required Privilege

`ADMIN`

Additional privileges may be required on the target database for the `ttXlaApply` operation. For example, to apply a `CREATETAB` (create table) record to the target database, you must have `CREATE TABLE` or `CREATE ANY TABLE` privilege, as appropriate.

Syntax

```
SQLRETURN ttXlaApply(ttXlaHandle_h handle,
                     ttXlaUpdateDesc_t* record,
                     SQLINTEGER test)
```

Parameters

Parameter	Type	Description
handle	`ttXlaHandle_h`	Transaction log handle for the database
record	`ttXlaUpdateDesc_t*`	Transaction to generate SQL statement
test	`SQLINTEGER`	Test for old values: • 1: Test on • 0: Test off

Returns

Returns `SQL_SUCCESS` if call is successful. Otherwise, use ttXlaError to report the error.

If *test* is 1 and `ttXlaApply` detects an update conflict, an `sb_ErrXlaTupleMismatch` error is returned.

Example

This example applies an update to a database without testing for the previous value of the existing record:

```
ttXlaUpdateDesc_t record;
rc = ttXlaApply(xlahandle, &record, 0);
```

Note

When calling `ttXlaApply`, it is possible for the update to timeout or deadlock with concurrent transactions. In such cases, it is the application's responsibility to roll the transaction back and reapply the updates.

See Also

ttXlaCommit
ttXlaRollback
ttXlaLookup
ttXlaTableCheck
ttXlaGenerateSQL

ttXlaCommit

This function is part of XLA replication functionality and is not appropriate for use in a typical XLA application.

Description

Commits the current transaction being applied on the transaction log handle. This routine commits the transaction regardless of whether the transaction has completed. You can call this routine to respond to transient errors (timeout or deadlock) reported by ttXlaApply, which applies the current transaction if it does not encounter an error.

See Handling Timeout and Deadlock Errors.

Required Privilege

`XLA`

Syntax

```
SQLRETURN ttXlaCommit(ttXlaHandle_h handle)
```

Parameters

Parameter	Type	Description
handle	`ttXlaHandle_h`	Transaction log handle for the database

Returns

Returns `SQL_SUCCESS` if call is successful. Otherwise, use ttXlaError to report the error.

Example

```
rc = ttXlaCommit(xlahandle);
```

See Also

ttXlaApply
ttXlaRollback

ttXlaLookup
ttXlaTableCheck
ttXlaGenerateSQL

ttXlaGenerateSQL

This function is part of XLA replication functionality and is not appropriate for use in a typical XLA application.

> **Note:**
>
> This function does not currently work with LOB locators.

Description

Generates a SQL DML or DDL statement that expresses the effect of the update record. The generated statement is not applied to any database. Instead, the statement is returned in the given buffer, whose maximum size is specified by the *maxLen* parameter. The actual size of the buffer is returned in *actualLen*. For update and delete records, ttXlaGenerateSQL requires a primary key or a unique index on a non-nullable column to generate the correct SQL.

The generated SQL statement is encoded in the connection character set that is associated with the ODBC connection of the XLA handle.

Also see Replicating Updates to a Non-TimesTen Database.

Required Privilege

XLA

Syntax

```
SQLRETURN ttXlaGenerateSQL(ttXlaHandle_h handle,
                           ttXlaUpdateDesc_t* record,
                           out char* buffer,
                           SQLINTEGER maxLen,
                           out SQLINTEGER* actualLen)
```

Parameters

Parameter	Type	Description
handle	ttXlaHandle_h	Transaction log handle for the database
record	ttXlaUpdateDesc_t*	Record to be translated into SQL
buffer	char*	Location of the translated SQL statement
maxLen	SQLINTEGER	Maximum length of the buffer, in bytes
actualLen	SQLINTEGER*	Actual length of the buffer, in bytes

Returns

Returns `SQL_SUCCESS` if call is successful. Otherwise, use ttXlaError to report the error.

Example

This example generates the text of a SQL statement that is equivalent to the `UPDATE` expressed by an update record:

```
ttXlaUpdateDesc_t record;
char buffer[200];
/*
 * Get the desired update record into the varable record.
 */

SQLINTEGER actualLength;

rc = ttXlaGenerateSQL(xlahandle, &record, buffer, 200,
                      &actualLength);
```

Note

The `ttXlaGenerateSQL` function cannot generate SQL statements for update records associated with a table that has been dropped or altered since the record was generated.

See Also

ttXlaApply
ttXlaCommit
ttXlaRollback
ttXlaLookup
ttXlaTableCheck

ttXlaLookup

This function is part of XLA replication functionality and is not appropriate for use in a typical XLA application.

Description

This function looks for a record in the given table with key values according to the *`keys`* parameter. The formats of the *`keys`* and *`result`* records are the same as for ordinary rows. This function requires a primary key on the underlying table.

Required Privilege

XLA

Syntax

```
SQLRETURN ttXlaLookup(ttXlaHandle_h handle,
                        ttXlaTableDesc_t* table,
                        void* keys,
                        out void* result,
```

```
SQLINTEGER maxsize,
out SQLINTEGER* retsize)
```

Parameters

Parameter	Type	Description
handle	ttXlaHandle_h	Transaction log handle for the database
table	ttXlaTblDesc_t*	Table to search
keys	void*	A record in the defined structure for the table Only those columns of the keys record that are part of the primary key for the table are examined.
result	void*	Where the located record is copied If no record exists with the matching key columns, an error is returned.
maxsize	SQLINTEGER	Size of the largest record that can fit into the result buffer
retsize	SQLINTEGER*	Actual size of the record

Returns

Returns SQL_SUCCESS if call is successful. Otherwise, use ttXlaError to report the error.

Example

This example looks up a record given a pair of integer key values. Before this call, *table* should describe the desired table and *keybuffer* contains a record with the key columns set.

```
char keybuffer[100];
char recbuffer[2000];
ttXlaTableDesc_t table;
SQLINTEGER recordSize;

rc = ttXlaLookup(xlahandle, &table, keybuffer, recbuffer,
                 sizeof (recbuffer), &recordSize);
```

See Also

- ttXlaApply
- ttXlaCommit
- ttXlaRollback
- ttXlaTableCheck
- ttXlaGenerateSQL

ttXlaRollback

This function is part of XLA replication functionality and is not appropriate for use in a typical XLA application.

Description

Rolls back the current transaction being applied on the transaction log handle. You can call this routine to respond to transient errors (timeout or deadlock) reported by ttXlaApply.

See Handling Timeout and Deadlock Errors.

Required Privilege

XLA

Syntax

```
SQLRETURN ttXlaRollback(ttXlaHandle_h handle)
```

Parameters

Parameter	Type	Description
handle	ttXlaHandle_h	Transaction log handle for the database

Returns

Returns `SQL_SUCCESS` if call is successful. Otherwise, use ttXlaError to report the error.

Example

```
rc = ttXlaRollback(xlahandle);
```

See Also

ttXlaApply
ttXlaCommit
ttXlaLookup
ttXlaTableCheck
ttXlaGenerateSQL

ttXlaTableCheck

This function is part of XLA replication functionality and is not appropriate for use in a typical XLA application.

Description

When using XLA as a replication mechanism, this function verifies that the named table in the ttXlaTblDesc_t structure received from a master database is compatible with a subscriber database or database associated with the transaction log handle. The *compat* parameter indicates whether the tables are compatible.

See Checking Table Compatibility Between Databases.

Required Privilege

XLA

Syntax

```
SQLRETURN ttXlaTableCheck(ttXlaHandle_h handle,
                          ttXlaTblDesc_t* table,
```

```
ttXlaColDesc_t* columns,
out SQLINTEGER* compat)
```

Parameters

Parameter	Type	Description
handle	`ttXlaHandle_h`	Transaction log handle for the database
table	ttXlaTblDesc_t*	Table description
columns	ttXlaColDesc_t*	Column description for the table
compat	`SQLINTEGER*`	Compatibility information: • 1: Tables are compatible. • 0: Tables are not compatible.

Returns

Returns `SQL_SUCCESS` if call is successful. Otherwise, use ttXlaError to report the error.

Example

This example checks the compatibility of a table:

```
SQLINTEGER compat;
ttXlaTblDesc_t table;
ttXlaColDesc_t columns[20];
/*
 * Get the desired table and column definitions into
 * the variables "table" and "columns"
 */
rc = ttXlaTableCheck(xlahandle, &table, columns, &compat);
if (compat) {
    /* Compatible */
}
else {
    /*
     * Not compatible or some other error occurred
     */
}
```

See Also

ttXlaApply
ttXlaCommit
ttXlaRollback
ttXlaLookup
ttXlaGenerateSQL

C Data Structures Used by XLA

This section describes the C data structures used by the XLA functions described in this chapter.

These structures are defined in the following file:

installation_dir`/include/tt_xla.h`

You must include this file when building your XLA application.

Table 9-1 Summary of C Data Structures

C Data Structure	Description
ttXlaNodeHdr_t	Describes the record type. Used at the beginning of records returned by XLA.
ttXlaUpdateDesc_t	Describes an update record.
ttXlaVersion_t	Describes XLA version information returned by ttXlaGetVersion.
ttXlaTblDesc_t	Describes table information returned by ttXlaGetTableInfo.
ttXlaTblVerDesc_t	Describes table version returned by ttXlaVersionTableInfo.
ttXlaColDesc_t	Describes table column information returned by ttXlaGetColumnInfo.
tt_LSN_t	Describes a log record identifier used by bookmarks. This structure is used by the ttXlaUpdateDesc_t structure.
tt_XlaLsn_t	Describes a log record identifier used by an XLA bookmark.

ttXlaNodeHdr_t

Most C data structures begin with a standard header that describes the data record type and length. The standard header has the type `ttXlaNodeHdr_t`.

This header has the following fields.

Field	Type	Description
nodeType	`char`	The type of record: • `TTXLANHVERSION`: Version • `TTXLANHUPDATE`: Update • `TTXLANHTABLEDESC`: Table description • `TTXLANHCOLDESC`: Column description • `TTXLANHSTATUS`: Status • `TTXLANHINVALID`: Invalid
byteOrder	`char`	Byte order of the record: • "1": Big-endian • "2": Little-endian
length	`SQLUINTEGER`	Total length of record, including all attachments

ttXlaUpdateDesc_t

This structure describes an update operation to a single row (or *tuple*) in the database.

Each update record returned by a ttXlaNextUpdate or ttXlaNextUpdateWait function begins with a fixed length `ttXlaUpdateDesc_t` header followed by zero to two rows from the database. The row data differs depending on the record type reported in the `ttXlaUpdateDesc_t` header:

- No rows are present in a `COMMITONLY` record.
- One row is present in `INSERTTUP` or `DELETETUP`.

- Two rows are present in an `UPDATETUP` record to report the row data before and after the update, respectively.
- Special format rows are present in `CREATAB`, `DROPTAB`, `CREAIND`, `DROPIND`, `CREATVIEW`, `DROPVIEW`, `CREATSEQ`, `DROPSEQ`, `CREATSYN`, `DROPSYN`, `ADDCOLS`, and `DRPCOLS` records, which are described in Special Update Data Formats.

The *`flags`* field is a bit-map of special options for the record update.

The *`connID`* field identifies the ODBC connection handle that initiated the update. This value can be used to determine if updates came from the same connection.

A separate commit XLA record is generated when a call to the `ttApplicationContext` procedure is not followed by an operation that generates an XLA record. See Passing Application Context for a description of the `ttApplicationContext` procedure.

Note

XLA cannot receive notification of the following:

- `CREATE VIEW` or `DROP VIEW` for a non-materialized view
- `CREATE GLOBAL TEMPORARY TABLE` or `DROP TABLE` for a temporary table

The only XLA records that can be generated from an `ALTER TABLE` operation are of the following types:

- `ADDCOLS` or `DRPCOLS` when columns are added or dropped
- `CREAIND` or `DROPIND` when a unique attribute of a column is modified

While sequence creates (`CREATESEQ`) and drops (`DROPSEQ`) are visible through XLA, sequence increments are not.

All deletes resulting from cascading deletes and aging are visible through XLA. The *`flags`* value (discussed in the following table) indicates when deletes are due to cascading or aging.

The fields of the update header defined by `ttXlaUpdateDesc_t` are as follows.

Field	Type	Description
`header`	ttXlaNodeHdr_t	Standard data header

Field	Type	Description
type	SQLUSMALLINT	Record type: • CREATAB: Create table. • DROPTAB: Drop table. • CREAIND: Create index. • DROPIND: Drop index. • CREATVIEW: Create view. • DROPVIEW: Drop view. • CREATSEQ: Create sequence. • DROPSEQ: Drop sequence. • CREATSYN: Create synonym. • DROPSYN: Drop synonym. • ADDCOLS: Add columns. • DRPCOLS: Drop columns. • TRUNCATE: Truncate table. • INSERTTUP: Insert. • UPDATETUP: Update. • DELETETUP: Delete. • COMMITONLY: Commit.

<table>
<tr><th>Field</th><th>Type</th><th>Description</th></tr>
<tr><td>flags</td><td><code>SQLUSMALLINT</code></td><td>Special options on record update:

<code>TT_UPDCOMMIT</code>: Indicates that the update record is the last record for the transaction. (Implied commit.)
<code>TT_UPDFIRST</code>: Indicates that the update record is the first record for the transaction.
<code>TT_UPDREPL</code>: Indicates that this update was the result of a non-XLA TimesTen replicated update from another database.
<code>TT_UPDCOLS</code>: Indicates the presence of a list following the last returned row that specifies which columns in the row were updated. The list consists of an array of <code>SQLUSMALLINT</code> values, the first of which is the number of columns that were updated, followed by the column numbers of the updated columns. For example, if the first and third columns are updated, the array is (2, 1, 3) or (2, 3, 1), depending on the <code>UPDATE</code> statement used. This array is in all <code>UPDATETUP</code> records.
<code>TT_UPDDEFAULT</code>: Indicates that the update record (either a <code>CREATAB</code> or <code>ADDCOLS</code>) contains default column values. If set, the default columns are presented as an array of <code>SQLUSMALLINT</code> values followed by a string with all the default values concatenated. The number of <code>SQLUSMALLINT</code> values in the array equals the number of columns in the <code>CREATAB</code> or <code>ADDCOLS</code> record.
<code>TT_CASCDEL</code>: Indicates that the XLA update was generated as part of a cascade delete operation.
<code>TT_AGING</code>: Indicates that the XLA update was generated as part of an aging operation.

If the value of a specific column is 0, it indicates that column does not have a default value. The defaults for all nonzero values are concatenated in a string and are presented in order, with the array value indicating the length of the default value. For example, three columns with defaults 1 of type <code>INTEGER</code>, no default, and "apple" of type <code>VARCHAR2(10)</code> is (1,0,5)"1apple".

Decimal values for each of these flags bits is as follows. (Note that some flag values are for internal use only.)
<pre>
TT_UPDCOMMIT   1
TT_UPDFIRST   2
TT_UPDREPL   4
TT_UPDCOLS   8
TT_UPDDEFAULT   64
TT_CASCDEL   256
TT_AGING   512
</pre></td></tr>
<tr><td>contextOffset</td><td><code>SQLUINTEGER</code></td><td>Offset to application-provided context value

This value is 0 if there is no context. A nonzero value indicates the location of the context relative to the beginning of the XLA record.</td></tr>
<tr><td>connID</td><td><code>SQLUBIGINT</code></td><td>Connection ID owning the transaction</td></tr>
</table>

Field	Type	Description
sysTableID	SQLUBIGINT	System-provided identifier of the affected table
userTableID	SQLUBIGINT	Application-defined table ID of the affected table
tranID	SQLUBIGINT	Read-only, system-provided transaction identifier
LSN	tt_LSN_t	Transaction log record identifier of this operation, used for diagnostics
tuple1	SQLUINTEGER	Length of first row (tuple), or zero
tuple2	SQLUINTEGER	Length of second row (tuple), or zero

Note:

Be aware that tt_LSN_t, particularly the *logFile* and *logOffset* fields, is used differently than in earlier releases, referring to log record identifiers rather than sequentially increasing LSNs. See the note in "tt_LSN_t".

Special Update Data Formats

The data contained in an update record follows the ttXlaTblDesc_t header.

This section describes the data formats for the special update records related to specific SQL operations. See ttXlaTblDesc_t.

CREATE TABLE

For a CREATE TABLE operation, the special row value consists of the ttXlaTblDesc_t record describing the new table, followed by the ttXlaColDesc_t records that describe each column.

See ttXlaColDesc_t.

ALTER TABLE

For an ALTER TABLE operation, the special row value consists of a ttXlaDropTableTup_t or ttXlaAddColumnTup_t value, followed by a ttXlaColDesc_t record that describes the column.

ttXlaDropTableTup_t

For a DROP TABLE operation, the row value is as follows.

Field	Type	Description
tblName	char(31)	Name of the dropped table
tblOwner	char(31)	Owner of the dropped table

ttXlaTruncateTableTup_t

For a `TRUNCATE TABLE` operation, the row value is as follows.

Field	Type	Description
tblName	char(31)	Name of the truncated table
tblOwner	char(31)	Owner of the truncated table

ttXlaCreateIndexTup_t

For a `CREATE INDEX` operation, the row value is as follows.

Field	Type	Description
tblName	char(31)	Name of the table on which the index is defined
tblOwner	char(31)	Owner of the table on which the index is defined
ixName	char(31)	Name of the new index
flag	char(31)	Index flag: • "P": Primary key • "F": Foreign key • "R": Regular
nixcols	SQLUINTEGER	Number of indexed columns
ixColsSys	SQLUINTEGER(16)	Indexed column numbers using system numbers
ixColsUser	SQLUINTEGER(16)	Indexed column numbers using user-defined column IDs
ixType	char	Type of index: • "T": Range • "H": Hash • "B": Bit map
ixUnique	char	Uniqueness of index: • "U": Unique • "N": Non-unique
pages	SQLUINTEGER	Number of pages for hash indexes

ttXlaDropIndexTup_t

For a `DROP INDEX` operation, the row value is as follows.

Field	Type	Description
tblName	char(31)	Name of the table on which the index was dropped
tblOwner	char(31)	Owner of the table on which the index was dropped
ixName	char(31)	Name of the dropped index

ttXlaAddColumnTup_t

For an `ADD COLUMN` operation, the row value is as follows.

Field	Type	Description
ncols	SQLUINTEGER	Number of additional columns

Following this special row are the `ttXlaColDesc_t` records describing the new columns.

ttXlaDropColumnTup_t

For a `DROP COLUMN` operation, the row value is as follows.

Field	Type	Description
ncols	SQLUINTEGER	Number of dropped columns

Following this special row is an array of `ttXlaColDesc_t` records describing the columns that were dropped.

ttXlaCreateSeqTup_t

For a `CREATE SEQUENCE` operation, the row value is as follows.

Field	Type	Description
sqName	char(31)	Name of sequence
sqOwner	char(31)	Owner of sequence
cycle	char	Cycle flag Indicates whether the sequence number generator continues to generate numbers after it reaches the maximum or minimum value: • "1": Yes • "0": No
minval	SQLBIGINT	Minimum value of sequence
maxval	SQLBIGINT	Maximum value of sequence
incr	SQLBIGINT	Increment between sequence numbers Positive numbers indicate an ascending sequence and negative numbers indicate a descending sequence. In a descending sequence, the range goes from *maxval* to *minval*. In an ascending sequence, the range goes from *minval* to *maxval*.

ttXlaDropSeqTup_t

For a `DROP SEQUENCE` operation, the row value is as follows.

Field	Type	Description
sqName	char(31)	Name of sequence
sqOwner	char(31)	Owner of sequence

ttXlaViewDesc_t

For a `CREATE VIEW` operation, the row value is as follows.

> **Note:**
>
> This applies to either materialized or non-materialized views.

Field	Type	Description
vwName	`char(31)`	Name of view
vwOwner	`char(31)`	Owner of view
sysTableID	`SQLUBIGINT`	System table ID stored in `SYS.TABLES`

ttXlaDropViewTup_t

For a `DROP VIEW` operation, the row value is as follows.

> **Note:**
>
> This applies to either materialized or non-materialized views.

Field	Type	Description
vwName	`char(31)`	Name of view
vwOwner	`char(31)`	Owner of view

ttXlaCreateSynTup_t

For a `CREATE SYNONYM` operation, the row value is as follows.

Field	Type	Description
synName	`char(31)`	Name of synonym
synOwner	`char(31)`	Owner of synonym
objName	`char(31)`	Name of object the synonym points to
objOwner	`char(31)`	Owner of object the synonym points to
isPublic	`char`	Indicates whether the synonym is public: • "1": True • "0": False
isReplace	`char`	Indicates whether the synonym was created using `CREATE OR REPLACE`: • "1": True • "0": False

ttXlaDropSynTup_t

For a DROP SYNONYM operation, the row value is as follows.

Field	Type	Description
synName	char(31)	Name of synonym
synOwner	char(31)	Owner of synonym
isPublic	char	Indicates whether the synonym is public: • "1": True • "0": False

ttXlaSetTableTup_t

The description of the SET TABLE ID operation uses the previously assigned application table identifier in the main part of the update record and provides the new value of the application table identifier in the following special row.

Field	Type	Description
newID	SQLUBIGINT	New user-defined table ID

ttXlaSetColumnTup_t

The description of the SET COLUMN ID operation provides the following special row:

Field	Type	Description
oldUserColID	SQLUINTEGER	Previous user-defined column ID value
newUserColID	SQLUINTEGER	New user-defined column ID value
sysColID	SQLUINTEGER	System column ID

ttXlaSetStatusTup_t

A change in a table's replication status provides the following special row:

Field	Type	Description
oldStatus	SQLUINTEGER	Previous replication status
newStatus	SQLUINTEGER	New replication status

Locating the Row Data Following a ttXlaUpdateDesc_t Header

The update header is immediately followed by the row data. The row data is stored in an internal format with the offsets given in the ttXlaColDesc_t structure returned by ttXlaGetColumnInfo.

See Retrieving Update Records From the Transaction Log and Inspecting Record Headers and Locating Row Addresses for a detailed discussion on obtaining update records and inspecting the contents of ttXlaUpdateDesc_t headers. Below is a summary of these procedures.

See ttXlaGetColumnInfo.

You can locate the address of the row data by adding the address of the update header to its size.

For example:

```
char* Row = (char*)&ttXlaUpdateDesc_t +
             sizeof(ttXlaUpdateDesc_t);
```

For UPDATETUP records, there are two rows of data following the ttXlaUpdateDesc_t header. The first row contains the data before the update, and the second row the data after the update.

Since the new row is right after the old row, you can calculate its address by adding the address of the old row to its length (tuple1).

For example:

```
char* oldRow = (char*)&ttXlaUpdateDesc_t +
                sizeof(ttXlaUpdateDesc_t);
char* newRow = oldRow + ttXlaUpdateDesc_t.tuple1;
```

See ttXlaColDesc_t for details on how to access the column data in a returned row.

ttXlaVersion_t

To permit future extensions to XLA, a version structure ttXlaVersion_t describes the current XLA version and structure byte order.

This structure is returned by the ttXlaGetVersion function.

This structure has the following fields:

Field	Type	Description
header	ttXlaNodeHdr_t	Standard data header
hardware	char(16)	Name of hardware platform
wordSize	SQLUINTEGER	Native word size (32 or 64 bits)
TTMajor	SQLUINTEGER	TimesTen major version
TTMinor	SQLUINTEGER	TimesTen minor version
TTPatch	SQLUINTEGER	TimesTen point release number
OS	char(16)	Name of operating system
OSMajor	SQLUINTEGER	Operating system major version
OSMinor	SQLUINTEGER	Operating system minor version

ttXlaTblDesc_t

Table information is portrayed through the ttXlaTblDesc_t structure.

This structure is returned by the ttXlaGetTableInfo function.

This structure has the following fields:

Field	Type	Description
header	ttXlaNodeHdr_t	Standard data header
tblName	char(31)	Name of the table, null-terminated
tblOwner	char(31)	Owner of the table, null-terminated
sysTableID	SQLUBIGINT	Unique system-defined table identifier
userTableId	SQLUBIGINT	User-defined table identifier
columns	SQLUINTEGER	Number of columns
width	SQLUINTEGER	Inline row size
nPrimCols	SQLUINTEGER	Number of primary columns
primColsSys	SQLUINTEGER(16)	System primary key column numbers
primColsUser	SQLUINTEGER(16)	User-defined primary key column numbers

The inline row size includes space for all fixed-width columns, null column flags, and pointer information for variable-length columns. Each varying-length column occupies four bytes of inline row space.

Note the following if the table has a declared primary key:

- The *nPrimCols* value is greater than 0.
- The *primColsSys* array contains the column numbers of the primary key, in the same order in which they were originally declared with the CREATE TABLE statement.
- The *primColsUser* array contains the corresponding application-specified column identifiers.

ttXlaTblVerDesc_t

This data structure contains the table version number and ttXlaTblDesc_t.

It is returned by ttXlaVersionTableInfo. This structure has the following fields:

Field	Type	Description
tblDesc	ttXlaTblDesc_t	Table description
tblVer	SQLBIGINT	System-generated table version number

ttXlaColDesc_t

Column information is given through this structure, which is returned by the ttXlaGetColumnInfo function.

The structure has the following fields:

Field	Type	Description
header	ttXlaNodeHdr_t	Standard data header
colName [tt_NameLenMax]	char	Name of the column

Field	Type	Description
pad0	SQLUINTEGER	Pad to four-byte boundary
sysColNum	SQLUINTEGER	Ordinal number of the column as determined when the table is created or subsequently altered This is the same as the corresponding COLNUM value in SYS.COLUMNS. (See SYS.COLUMNS in *Oracle TimesTen In-Memory Database System Tables and Views Reference*.)
userColNum	SQLUINTEGER	Ordinal number of the column if optionally specified by the user This is zero or a column number specified through the ttSetUserColumnID TimesTen built-in procedure. (See ttSetUserColumnID in *Oracle TimesTen In-Memory Database Reference*.)
dataType	SQLUINTEGER	Structure in ODBC TTXLA_* code See XLA Data Types.
size	SQLUINTEGER	Maximum or basic size of column
offset	SQLUINTEGER	Offset to fixed-length part of column
nullOffset	SQLUINTEGER	Offset to null byte, or zero if not nullable
precision	SQLSMALLINT	Numeric precision for decimal types
scale	SQLSMALLINT	Numeric scale for decimal types
flags	SQLUINTEGER	Column flag: • TT_COLPRIMKEY: Column is primary key. • TT_COLVARYING: Column is stored out of line. • TT_COLNULLABLE: Column is nullable. • TT_COLUNIQUE: Column has a unique attribute defined on it.

The procedures for obtaining a ttXlaColDesc_t structure and inspecting its contents are described in Inspecting Column Data. Below is a summary of these procedures.

The ttXlaColDesc_t structure is returned by the ttXlaGetColumnInfo function. This structure contains the metadata needed to access column information in a particular table. For example, you can use the *offset* field to locate specific column data in the row or rows returned in an update record after the ttXlaColDesc_t structure. By adding the *offset* to the address of a returned row, you can locate the address to the column value. You can then cast this value to the corresponding C types according to the *dataType* field, or pass it to one of the conversion routines described in Converting Complex Data Types.

TimesTen row data consists of fixed-length data followed by any variable-length data.

- For fixed length column data, ttXlaColDesc_t returns the *offset* and *size* of the column data. The *offset* is relative to the beginning of the fixed part of the record. See the example below.

- For variable-length column data (VARCHAR2, NVARCHAR2, and VARBINARY), *offset* is an address that points to a four-byte offset value. By adding the offset address to the offset value, you can obtain the address of the column data in the variable-length portion of the row. The first eight bytes at this location is the length of the data, followed by the actual data. For variable-length data, the returned size value is the maximum allowable column size. See the example below.

For columns that can have null values, *nullOffset* points to a null byte in the record. This value is 1 if the column is null, or 0 if it is not null. See Detecting Null Values.

The *flags* bits define whether the column is nullable, part of a primary key, or stored out of line.

The *sysColNum* value is the system column number to assign to the column. This value begins with 1 for the first column.

Note:

LOB support in XLA is limited, as follows:

- You can subscribe to tables containing LOB columns, but information about the LOB value itself is unavailable.
- ttXlaGetColumnInfo returns information about LOB columns.
- Columns containing LOBs are reported as empty (zero length) or null (if the value is actually NULL). In this way, you can tell the difference between a null column and a non-null column.

For fixed-length column data, the address of a column is the *offset* value in the ttXlaColDesc_t structure, plus the address of the row as follows:

```
ttXlaColDesc_t colDesc;

void* pColVal = colDesc->offset + row;
```

The value of the column can be obtained by dereferencing this pointer using a type pointer that corresponds to the data type. For example, for SQL_INTEGER, the ODBC type is SQLINTEGER and the value of the column can be obtained by the following:

```
*((SQLINTEGER*) pColVal))
```

In the case of variable-length column data, the *pColVal* calculated above is the address of a four-byte offset value. Adding this offset value to the address of *pColVal* provides a pointer to the beginning of the variable-length column data. The first eight bytes at this location is the length of this data (var_len), followed by the actual data (var_data).

In this example, a VARCHAR string is copied and printed.

```
tt_ptrint* var_len = (tt_ptrint*)((char*)pColVal +
                     *((int*)pColVal));
char* var_data = (char*)(var_len+1);
char* buffer = malloc(*var_len+1);
memcpy(buffer,var_data,*var_len);
buffer[*var_len] = (char)NULL; /* NULL terminate the string */
```

```
printf("%s\n",buffer);
free(buffer);
```

tt_LSN_t

Description of log record identifier used by bookmarks.

This structure is used by the ttXlaUpdateDesc_t structure.

Field	Type	Description
logFile	SQLUBIGINT	Higher order portion of log record identifier
logOffset	SQLUBIGINT	Lower order portion of log record identifier

Note:

The *logFile* and *logOffset* field names are retained for backward compatibility, although their usage has changed. In previous releases the values referred to LSNs, which increased sequentially, and the values had very specific meanings, indicating the log file number plus byte offset. Now they refer to log record identifiers, which are more abstract and do not have a direct relationship to the log file number and byte offset. All you can assume about a sequence of log record identifiers is that a log record identifier B read at a later time than a log record identifier A has a higher value.

tt_XlaLsn_t

Description of a log record identifier used by bookmarks.

This structure is returned by the ttXlaGetLSN function and used by the ttXlaSetLSN function.

The *checksum* is specific to an XLA handle to ensure that every log record identifier is related to a known XLA connection.

Field	Type	Description
checksum	SQLUINTEGER	Checksum used to ensure that it is a valid log record identifier handle
xid	SQLUSMALLINT	Transaction ID
logFile	SQLUBIGINT	Higher order portion of log record identifier
logOffset	SQLUBIGINT	Lower order portion of log record identifier

> **Note:**
>
> The `logFile` and `logOffset` field names are retained for backward compatibility, although their usage has changed. In previous releases the values referred to LSNs, which increased sequentially, and the values had very specific meanings, indicating the log file number plus byte offset. Now they refer to log record identifiers, which are more abstract and do not have a direct relationship to the log file number and byte offset. All you can assume about a sequence of log record identifiers is that a log record identifier B read at a later time than a log record identifier A has a higher value.

10 TimesTen ODBC Support

TimesTen provides an ODBC 3.51 driver that also supports ODBC 2.5.

- For ODBC 3.5, TimesTen supports ODBC 3.51 core interface conformance.
- For ODBC 2.5, TimesTen supports Extension Level 1, as well as Extension Level 2 features that are documented in this chapter.

This chapter covers the details of TimesTen ODBC support, discussing the following topics:

- TimesTen ODBC 3.5 Support
- TimesTen ODBC 2.5 Support
- ODBC API Incompatibilities With Previous Versions of TimesTen

You can also refer to the following additional resources.

- Backward compatibility and standards compliance:

  `https://docs.microsoft.com/en-us/sql/odbc/reference/develop-app/backward-compatibility-and-standards-compliance`
- Summary of differences between ODBC 2.5 and ODBC 3.5:

  `https://docs.microsoft.com/en-us/sql/odbc/reference/appendixes/behavioral-changes-and-odbc-3-x-drivers`
- Additional behavioral changes:

  `https://docs.microsoft.com/en-us/sql/odbc/reference/develop-app/behavioral-changes`
- Writing ODBC 3.x applications:

  `https://docs.microsoft.com/en-us/sql/odbc/reference/develop-app/writing-odbc-3-x-applications`
- ODBC API reference documentation:

  `https://docs.microsoft.com/en-us/sql/odbc/reference/syntax/odbc-api-reference`

Also see TimesTen Include Files, for information about `#include` files for TimesTen extensions.

TimesTen ODBC 3.5 Support

This section covers theses topics for TimesTen ODBC 3.5 support.

- Using ODBC 3.5 With TimesTen
- Client/Server Cross-Release Restrictions With ODBC 3.5
- ODBC 3.5 New and Replacement Function Support
- ODBC 3.5 Data Type Support Notes

- Environment Attribute Support for ODBC 3.5
- Attribute Support for ODBC 3.5 SQLSetConnectAttr and SQLGetConnectAttr
- Attribute Support for ODBC 3.5 SQLSetStmtAttr and SQLGetStmtAttr
- Attribute Support for ODBC 3.5 SQLGetEnvAttr
- TimesTen Field Identifiers for ODBC 3.5 SQLColAttribute
- Information Type Support for ODBC 3.5 SQLGetInfo
- TimesTen SQL Keywords for ODBC 3.5

Using ODBC 3.5 With TimesTen

In accordance with the ODBC 3.5 specification, an ODBC 3.5 application calls `SQLSetEnvAttr` to set `SQL_ATTR_ODBC_VERSION` to `SQL_OV_ODBC3` directly after calling `SQLAllocHandle`.

For example:

```
RetCode = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &hEnv);
...
RetCode = SQLSetEnvAttr(hEnv, SQL_ATTR_ODBC_VERSION, (SQLPOINTER)SQL_OV_ODBC3,
0);
...
RetCode = SQLAllocHandle(SQL_HANDLE_DBC, hEnv, &hDbc);
...
RetCode = SQLDriverConnect(hDbc, winHandle, connStr, SQL_NTS,
                           outConnStr, outConnStrBufferLen,
                           &outConnStrLen, SQL_DRIVER_NOPROMPT);
...
```

Tip:

Because TimesTen Release 22.1 is a major release, you should recompile and relink existing ODBC applications. Also see ODBC API Incompatibilities With Previous Versions of TimesTen.

It is also advisable to link your applications dynamically rather than statically.

Client/Server Cross-Release Restrictions With ODBC 3.5

Previous TimesTen releases support cross-release client/server connections, where the client version could be either newer or older than the server version (such as a 22.1 client connecting to an 18.1 server, or an 18.1 client connecting to a 22.1 server).

Due to changes in ODBC 3.5 functionality, TimesTen clients of Release 18.1 or later cannot connect to an older TimesTen server when the client declares itself to be ODBC 3.*x* compliant by specifying `SQL_ODBC_OV3` in a `SQLSetEnvAttr` call (such as shown in the preceding section).

> **Note:**
>
> This limitation does not impact ODBC 2.5 applications.

ODBC 3.5 New and Replacement Function Support

There are new and replacement ODBC 3.5 functions supported by TimesTen.

> **Note:**
>
> TimesTen supports wide-character (W) function versions for applications not using a generic driver manager, as indicated in Table 10-1 and Table 10-11.

Table 10-1 Supported ODBC 3.5 New and Replacement Functions

Function	Notes
`SQLAllocHandle`	With applicable settings for *`HandleType`*, replaces ODBC 2.5 functions `SQLAllocEnv`, `SQLAllocConnect`, and `SQLAllocStmt`.
`SQLBulkOperations`	Call returns "Driver not capable."
`SQLCloseCursor`	Replaces the ODBC 2.5 function `SQLFreeStmt` when that function is used with the `SQL_CLOSE` option.
`SQLColAttribute` and `SQLColAttributeW`	Replaces the ODBC 2.5 function `SQLColAttributes`. See TimesTen Field Identifiers for ODBC 3.5 SQLColAttribute.
`SQLCopyDesc`	No notes
`SQLEndTran`	Replaces the ODBC 2.5 function `SQLTransact`.
`SQLFetchScroll`	TimesTen supports only the `SQL_FETCH_NEXT` option (forward scroll).
`SQLFreeHandle`	With applicable settings for *`HandleType`*, replaces ODBC 2.5 functions `SQLFreeEnv`, `SQLFreeConnect`, and `SQLFreeStmt`.
`SQLGetConnectAttr` and `SQLGetConnectAttrW`	Replaces the ODBC 2.5 function `SQLGetConnectOption`. Support is added for the TimesTen driver manager (TTDM). See Attribute Support for ODBC 3.5 SQLSetConnectAttr and SQLGetConnectAttr.
`SQLGetDescField` and `SQLGetDescFieldW`	No notes
`SQLGetDescRec` and `SQLGetDescRecW`	No notes

Table 10-1 (Cont.) Supported ODBC 3.5 New and Replacement Functions

Function	Notes
`SQLGetDiagField` and `SQLGetDiagFieldW`	Replaces the ODBC 2.5 function `SQLError`. Native error codes are TimesTen errors. You may receive generic errors such as, "Execution at Oracle failed. Oracle error code *nnn*." When using `SQLGetDiagField` or `SQLGetDiagFieldW`: • Use `TT_MAX_MESSAGE_LENGTH` instead of `SQL_MAX_MESSAGE_LENGTH` (which is a limit of 512 bytes). • Handle a possible return of `SQL_SUCCESS_WITH_INFO` (for example, in case the message length exceeded the input buffer size).
`SQLGetDiagRec` and `SQLGetDiagRecW`	Replaces the ODBC 2.5 function `SQLError`. Native error codes are TimesTen errors. You may receive generic errors such as, "Execution at Oracle failed. Oracle error code *nnn*." When using `SQLGetDiagRec` or `SQLGetDiagRecW`: • Use `TT_MAX_MESSAGE_LENGTH` instead of `SQL_MAX_MESSAGE_LENGTH` (which is a limit of 512 bytes). • Handle a possible return of `SQL_SUCCESS_WITH_INFO` (for example, in case the message length exceeded the input buffer size).
`SQLGetEnvAttr`	Support is added for the TimesTen driver manager (TTDM). See Attribute Support for ODBC 3.5 SQLGetEnvAttr.
`SQLGetInfo`	See Information Type Support for ODBC 3.5 SQLGetInfo.
`SQLGetStmtAttr` and `SQLGetStmtAttrW`	Replaces the ODBC 2.5 function `SQLGetStmtOption`. See Attribute Support for ODBC 3.5 SQLSetStmtAttr and SQLGetStmtAttr.
`SQLSetConnectAttr` and `SQLSetConnectAttrW`	Replaces the ODBC 2.5 function `SQLSetConnectOption`. See Attribute Support for ODBC 3.5 SQLSetConnectAttr and SQLGetConnectAttr.
`SQLSetDescField`	No notes
`SQLSetDescRec`	No notes
`SQLSetEnvAttr`	Required for ODBC applications to set `SQL_ATTR_ODBC_VERSION` to `SQL_OV_ODBC3`.
`SQLSetStmtAttr` and `SQLSetStmtAttrW`	Replaces the ODBC 2.5 function `SQLSetStmtOption`. See Attribute Support for ODBC 3.5 SQLSetStmtAttr and SQLGetStmtAttr.

ODBC 3.5 Data Type Support Notes

TimesTen supports new ODBC 3.5 data types.

- `SQL_C_NUMERIC`
- `SQL_C_TYPE_DATE`

- SQL_C_TYPE_TIME
- SQL_C_TYPE_TIMESTAMP

TimesTen does not support these data types or has limited support:

- SQL_GUID: TimesTen does not support conversion of this type to a C type.
- SQL_INTERVAL_*xxxx*: TimesTen does not support conversion of interval types to C types.
- SQL_WCHAR: TimesTen does not support conversion of this type to C numeric types.

Environment Attribute Support for ODBC 3.5

There are standard environment attributes supported by TimesTen in ODBC 3.5.

Table 10-2 Standard Environment Attributes (ODBC 3.5)

Attribute	Notes
SQL_ATTR_ODBC_VERSION	Supported values SQL_OV_ODBC3 and SQL_OV_ODBC2.
SQL_ATTR_OUTPUT_NTS	Supported value SQL_TRUE.

Attribute Support for ODBC 3.5 SQLSetConnectAttr and SQLGetConnectAttr

There are support of standard attributes by TimesTen for the ODBC 3.5 SQLSetConnectAttr and SQLGetConnectAttr functions. Also, there are TimesTen-specific connection attributes, which are supported in both ODBC 3.5 and ODBC 2.5.

Table 10-3 lists support of standard attributes by TimesTen for the ODBC 3.5 SQLSetConnectAttr and SQLGetConnectAttr functions. Table 10-4 lists TimesTen-specific connection attributes, which are supported in both ODBC 3.5 and ODBC 2.5. These functions enable you to set connection attributes after the initial connection or retrieve those settings.

Also see Attribute Support for ODBC 3.5 SQLSetStmtAttr and SQLGetStmtAttr. Those attributes can also be set using SQLSetConnectAttr, in which case the value serves as a default for all statements on the connection.

Note:

- An attribute setting through SQLSetConnectAttr or SQLSetStmtAttr overrides the setting of the corresponding connection attribute (as applicable).
- The documentation here also applies to SQLSetConnectAttrW and SQLGetConnectAttrW.
- The TimesTen driver manager (TTDM) is supported through the TT_TTDM_CONNECTION_TYPE attribute, as described in Table 10-4.

Table 10-3 Standard Connection Attributes (ODBC 3.5)

Attribute	Notes
SQL_ATTR_ASYNC_ENABLE	Supported setting SQL_ASYNC_ENABLE_OFF.
SQL_ATTR_AUTO_IPD	Read-only (get); value is always SQL_TRUE.
SQL_ATTR_CONNECTION_DEAD	Read-only (get).
SQL_ATTR_CONNECTION_TIMEOUT	Supported setting 0; any other setting reverts to 0.
SQL_ATTR_ENLIST_IN_DTC	Driver not capable.
SQL_ATTR_METADATA_ID	Supported setting SQL_FALSE.

Table 10-4 TimesTen Connection Attributes

Attribute	Notes
TT_CLIENT_TIMEOUT	This is for client/server only and has the same functionality as the TTC_Timeout TimesTen client connection attribute. Also see Choose SQL and PL/SQL Timeout Values in *Oracle TimesTen In-Memory Database Operations Guide*.
TT_DYNAMIC_LOAD_ENABLE	See Enabling or Disabling Dynamic Load in *Oracle TimesTen In-Memory Database Cache Guide*. This has the same functionality as the DynamicLoadEnable cache general connection attribute.
TT_DYNAMIC_LOAD_ERROR_MODE	See Returning Errors for Dynamic Load in *Oracle TimesTen In-Memory Database Cache Guide*. This has the same functionality as the DynamicLoadErrorMode cache connection attribute.
TT_GRID_ENABLED_DATABASE	Read-only (get). This indicates whether the database is from a TimesTen instance enabled for TimesTen Scaleout.
TT_NLS_LENGTH_SEMANTICS	This has the same functionality as the NLS_LENGTH_SEMANTICS general connection attribute. See Additional Globalization Features.
TT_NLS_NCHAR_CONV_EXCP	This has the same functionality as the NLS_NCHAR_CONV_EXCP general connection attribute. See Additional Globalization Features.
TT_NLS_SORT	This has the same functionality as the NLS_SORT general connection attribute. There is related information about the functionality in Additional Globalization Features.
TT_NO_RECONNECT_ON_FAILOVER	Read-only (get). See Configuration of Automatic Client Failover. This indicates the setting of the TimesTen connection attribute TTC_NoReconnectOnFailover (for client connections only).

Table 10-4 (Cont.) TimesTen Connection Attributes

Attribute	Notes
TT_PREFETCH_CLOSE	Set to TT_PREFETCH_CLOSE_ON to optimize query performance. The default setting is TT_PREFETCH_CLOSE_OFF. Refer to Optimizing Query Performance.
TT_REGISTER_FAILOVER_CALLBACK	See ODBC Support for Automatic Client Failover. This attribute is client-only. If you attempt to use it in TimesTen direct mode, SQL_SUCCESS is returned but no action is taken.
TT_REPLICATION_TRACK	For ODBC applications that use parallel replication and specify replication tracks, this has the same functionality as the ReplicationTrack general connection attribute, to specify a track number for the connection.
TT_TTDM_CONNECTION_TYPE	For ODBC applications using TTDM, specifying this as the attribute returns a value indicating the type of connection represented by the HDBC object, one of the following: • TT_TTDM_CONN_NONE: The HDBC is not connected. • TT_TTDM_CONN_DIRECT: The HDBC is connected in direct mode. • TT_TTDM_CONN_CLIENT: The HDBC is connected in client/server mode. See the example below.

This example shows the use of SQLGetConnectAttr with the attribute TT_TTDM_CONNECTION_TYPE for an application using TTDM. The connection type, as documented in the table immediately above, is returned in connType.

```
SQLINTEGER connType = 0;
rc = SQLGetConnectAttr(hdbc, TT_TTDM_CONNECTION_TYPE, &connType, SQL_IS_INTEGER, NULL);
```

Attribute Support for ODBC 3.5 SQLSetStmtAttr and SQLGetStmtAttr

There are standard attributes supported by TimesTen for the ODBC 3.5 SQLSetStmtAttr and SQLGetStmtAttr functions. Also, there are TimesTen-specific statement attributes, which are supported in both ODBC 3.5 and ODBC 2.5.

Table 10-5 lists standard attributes supported by TimesTen for the ODBC 3.5 SQLSetStmtAttr and SQLGetStmtAttr functions. Table 10-6 lists TimesTen-specific statement attributes, which are supported in both ODBC 3.5 and ODBC 2.5. These functions enable you to set or retrieve statement attribute settings.

To set an attribute default value for all statements associated with a connection, use SQLSetConnectAttr.

> **Note:**
>
> - An attribute setting through SQLSetConnectAttr or SQLSetStmtAttr overrides the setting of the corresponding connection attribute (as applicable).
> - TimesTen also supports the options listed in Option Support for ODBC 2.5 SQLSetStmtOption and SQLGetStmtOption.

Table 10-5 Standard Statement Attributes (ODBC 3.5)

Attribute	Notes
SQL_ATTR_APP_PARAM_DESC	No notes
SQL_ATTR_APP_ROW_DESC	No notes
SQL_ATTR_CURSOR_SCROLLABLE	Supported setting SQL_NONSCROLLABLE.
SQL_ATTR_CURSOR_SENSITIVITY	Supported setting SQL_INSENSITIVE.
SQL_ATTR_ENABLE_AUTO_IPD	No notes
SQL_ATTR_IMP_PARAM_DESC	Read-only (get).
SQL_ATTR_IMP_ROW_DESC	Read-only (get).
SQL_ATTR_METADATA_ID	Supported setting SQL_FALSE.
SQL_ATTR_PARAM_BIND_OFFSET_PTR	No notes
SQL_ATTR_PARAM_BIND_TYPE	No notes
SQL_ATTR_PARAM_OPERATION_PTR	No notes
SQL_ATTR_PARAM_STATUS_PTR	No notes
SQL_ATTR_PARAMS_PROCESSED_PTR	No notes
SQL_ATTR_PARAMSET_SIZE	No notes
SQL_ATTR_ROW_ARRAY_SIZE	No notes
SQL_ATTR_ROW_BIND_OFFSET_PTR	No notes
SQL_ATTR_ROW_STATUS_PTR	No notes
SQL_ATTR_ROWS_FETCHED_PTR	No notes

Table 10-6 TimesTen Statement Attributes

Attribute	Notes
TT_NET_MSG_MAX_BYTES	In client/server, determines the maximum number of bytes in the result set buffer. See Configuring the Result Set Buffer Size in Client/Server Using ODBC.
TT_NET_MSG_MAX_ROWS	In client/server, determines the maximum number of rows in the result set buffer. See Configuring the Result Set Buffer Size in Client/Server Using ODBC.

Table 10-6 (Cont.) TimesTen Statement Attributes

Attribute	Notes
TT_PREFETCH_COUNT	See Prefetching Multiple Rows of Data.
TT_QUERY_THRESHOLD	See Setting a Threshold Duration for SQL Statements. This is to specify a time threshold for SQL statements, in seconds, after which TimesTen writes a warning to the support log.
TT_PRIVATE_COMMANDS	Commands are not shared with any other connection. See PrivateCommands in *Oracle TimesTen In-Memory Database Reference*.
TT_STMT_PASSTHROUGH_TYPE	Determines whether a specific prepared statement is passed through to Oracle Database by the cache passthrough feature. The value returned by `SQLGetStmtOption` can be either `TT_STMT_PASSTHROUGH_NONE` or `TT_STMT_PASSTHROUGH_ORACLE`. **Note**: In TimesTen, this option is supported only with `SQLGetStmtOption`. See Determining Passthrough Status. Also see Setting a Passthrough Level in *Oracle TimesTen In-Memory Database Cache Guide*.

Attribute Support for ODBC 3.5 SQLGetEnvAttr

This section describes TimesTen environmental attributes for `SQLGetEnvAttr`.

Table 10-7 lists TimesTen-specific environment attributes, which are supported in both ODBC 3.5 and ODBC 2.5. These attributes support the TimesTen driver manager (TTDM).

Table 10-7 TimesTen Environment Attributes

Attribute	Notes
TT_TTDM_CAPABILITIES	For ODBC applications using TTDM, if you specify this as the attribute and pass a pointer to a `SQLINTEGER` for the `ValuePtr` parameter, then a value is returned indicating the capabilities that are currently available through TTDM. The value is expressed as a bit-wise `OR` of these constants: • `TT_TTDM_CLIENT`: Client/server driver capabilities are available. • `TT_TTDM_DIRECT`: Direct driver capabilities are available. • `TT_TTDM_XLA`: XLA capabilities are available. • `TT_TTDM_ROUTING`: Routing API capabilities are available. • `TT_TTDM_UTILITY`: C utility API functions are available. See the example below. **Note:** The available capabilities depend on what is available in the TimesTen environment where the TTDM-based application is executing.

Table 10-7 (Cont.) TimesTen Environment Attributes

Attribute	Notes
`TT_TTDM_VERSION`	For ODBC applications using TTDM, if you specify this as the attribute and pass a pointer to a `SQLINTEGER` for the `ValuePtr` parameter, then the value returned indicates the TimesTen release (specifically, of the TTDM library) that the application is using, such as 18.1.4.9.0 or 22.1.1.1.0. See the example below.

This example shows the use of `SQLGetEnvAttr` with the attribute `TT_TTDM_VERSION` for an application using TTDM.

```
SQLCHAR ttdmver[21];
rc = SQLGetEnvAttr(henv, TT_TTDM_VERSION, (SQLPOINTER)ttdmver, sizeof(ttdmver),
NULL);
```

The value returned in `ttdmver` indicates the TimesTen release of the TTDM library that the application is using.

The next example shows the use of `SQLGetEnvAttr` with the attribute `TT_TTDM_CAPABILITIES` for an application using TTDM.

```
SQLINTEGER ttdmcap = 0;
rc = SQLGetEnvAttr(henv, TT_TTDM_CAPABILITIES, (SQLPOINTER)&ttdmcap, 0, NULL);
```

The value returned in `ttdmcap` is a bitwise `OR` of the constants documented in the table above, indicating what capabilities are currently available through TTDM. Then check what capabilities are supported:

```
...
if ( ttdmcap & TT_TTDM_CLIENT )
     printf("TTDM: Client driver functions are available\n");
if ( ttdmcap & TT_TTDM_DIRECT )
     printf("TTDM: Direct driver functions are available\n");
if ( ttdmcap & TT_TTDM_XLA )
     printf("TTDM: XLA functions are available\n");
if ( ttdmcap & TT_TTDM_ROUTING )
     printf("TTDM: Routing API functions are available\n");
if ( ttdmcap & TT_TTDM_UTILITY )
     printf("TTDM: Utility functions are available\n");
...
```

TimesTen Field Identifiers for ODBC 3.5 SQLColAttribute

The `SQLColAttribute` function returns descriptor information for a column in a result set.

Refer to ODBC API reference documentation for complete information about this function and standard column descriptors.

Note:

This replaces SQLColAttributes (plural) in ODBC 2.5.

See Table 10-8.

Table 10-8 TimesTen Field Identifiers: SQLColAttribute (ODBC 3.5)

Descriptor	Comment/Description
TT_COLUMN_INLINE	Returns TRUE for columns with inline data, or FALSE otherwise. This is returned in the SQLColAttribute *CharacterAttributePtr* parameter.
TT_COLUMN_LENGTH_SEMANTICS	For character-type columns, this returns "BYTE" for columns with byte length semantics and "CHAR" for columns with character length semantics. For non-character columns, it returns "". The information is returned in the SQLColAttribute *CharacterAttributePtr* parameter. This information refers to whether data length is measured in bytes or characters. Length semantics in TimesTen are the same as in Oracle Database. See Length Semantics in *Oracle Database Globalization Support Guide*.

Information Type Support for ODBC 3.5 SQLGetInfo

There is support in the TimesTen ODBC 3.5 implementation for standard and TimesTen-specific information types for the ODBC function SQLGetInfo.

Table 10-9 documents TimesTen support for standard information types that were introduced or renamed in ODBC 3.0, noting the TimesTen-specific correct value or values returned.

Refer to the following location for standard information:

https://docs.microsoft.com/en-us/sql/odbc/reference/syntax/sqlgetinfo-function

Also see Information Type Support for ODBC 2.5 SQLGetInfo. Those information types are still supported by the TimesTen ODBC 3.5 driver (with some renamed, as noted).

Table 10-9 TimesTen Support for Standard Information Types: SQLGetInfo (ODBC 3.5)

Information Type	Notes and Correct Values Returned by TimesTen
SQL_ACTIVE_ENVIRONMENTS	0: Environment objects are allocated from heap.
SQL_AGGREGATE_FUNCTIONS	SQL_AF_ALL, SQL_AF_AVG, SQL_AF_COUNT, SQL_AF_DISTINCT, SQL_AF_MAX, SQL_AF_MIN, SQL_AF_SUM
SQL_ALTER_DOMAIN	0: ALTER DOMAIN statement not supported.

Table 10-9 (Cont.) TimesTen Support for Standard Information Types: SQLGetInfo (ODBC 3.5)

Information Type	Notes and Correct Values Returned by TimesTen
SQL_ALTER_TABLE	SQL_AT_ADD_COLUMN_DEFAULT: ADD COLUMN clause is supported, with facility to specify column defaults (FIPS transitional level). SQL_AT_ADD_COLUMN_SINGLE: ADD COLUMN clause is supported (FIPS transitional level). SQL_AT_ADD_CONSTRAINT: ADD COLUMN clause is supported, with facility to specify column constraints (FIPS transitional level). SQL_AT_ADD_TABLE_CONSTRAINT: ADD TABLE CONSTRAINT clause is supported (FIPS transitional level). SQL_AT_DROP_COLUMN_CASCADE: DROP COLUMN ... CASCADE clause is supported (FIPS transitional level). SQL_AT_DROP_COLUMN_DEFAULT: ALTER COLUMN ... DROP COLUMN DEFAULT clause is supported (Intermediate level).
SQL_ASYNC_MODE	SQL_AM_NONE: Asynchronous mode not supported.
SQL_BATCH_ROW_COUNT	0: Batches of SQL statements not supported.
SQL_BATCH_SUPPORT	0: Batches of SQL statements not supported.
SQL_CATALOG_LOCATION	0: Catalog names as qualifiers not supported. SQL_QUALIFIER_LOCATION in ODBC 2.5.
SQL_CATALOG_NAME	"N": Catalog names as qualifiers not supported.
SQL_CATALOG_NAME_SEPARATOR	NULL: Not supported. SQL_QUALIFIER_NAME_SEPARATOR in ODBC 2.5.
SQL_CATALOG_TERM	"data store" SQL_QUALIFIER_TERM in ODBC 2.5.
SQL_CATALOG_USAGE	0: Catalogs not supported. SQL_QUALIFIER_USAGE in ODBC 2.5.
SQL_COLLATION_SEQ	Current value of the NLS_SORT database parameter. **Note**: Because TimesTen does not have a default character set, default collation for the default character is set is not applicable. NLS_SORT is the collation for the current character set.
SQL_CONVERT_GUID	0: CONVERT function not supported.
SQL_CONVERT_INTERVAL_DAY_TIME	0: CONVERT function not supported.
SQL_CONVERT_INTERVAL_YEAR_MONTH	0: CONVERT function not supported.
SQL_CONVERT_WCHAR	0: CONVERT function not supported.
SQL_CONVERT_WLONGVARCHAR	0: CONVERT function not supported.
SQL_CONVERT_WVARCHAR	0: CONVERT function not supported.
SQL_CREATE_ASSERTION	0: CREATE ASSERTION statement not supported.

Table 10-9 (Cont.) TimesTen Support for Standard Information Types: SQLGetInfo (ODBC 3.5)

Information Type	Notes and Correct Values Returned by TimesTen
SQL_CREATE_CHARACTER_SET	0: CREATE CHARACTER SET statement not supported.
SQL_CREATE_COLLATION	0: CREATE COLLATION statement not supported.
SQL_CREATE_DOMAIN	0: CREATE DOMAIN statement not supported.
SQL_CREATE_SCHEMA	0: CREATE SCHEMA statement not supported.
SQL_CREATE_TABLE	To determine which clauses are supported: SQL_CT_CREATE_TABLE: CREATE TABLE statement is supported (entry level). SQL_CT_TABLE_CONSTRAINT: Specifying table constraints is supported (FIPS transitional level). SQL_CT_CONSTRAINT_NAME_DEFINITION: <constraint name definition> clause is supported for naming column and table constraints (intermediate level). To specify the ability to create temporary tables: SQL_CT_COMMIT_PRESERVE: Deleted rows are preserved on commit (full level). SQL_CT_COMMIT_DELETE: Deleted rows are deleted on commit (full level). SQL_CT_GLOBAL_TEMPORARY: Global temporary tables can be created (full level). To specify the ability to create column constraints: SQL_CT_COLUMN_CONSTRAINT: Specifying column constraints is supported (FIPS transitional level). SQL_CT_COLUMN_DEFAULT: Specifying column defaults is supported (FIPS transitional level).
SQL_CREATE_TRANSLATION	0: CREATE TRANSLATION statement not supported.
SQL_CREATE_VIEW	SQL_CV_CREATE_VIEWS
SQL_CURSOR_SENSITIVITY	SQL_SENSITIVE: Cursors are sensitive to changes made by other cursors within the same transaction.
SQL_DATETIME_LITERALS	SQL_DL_SQL92_DATE, SQL_DL_SQL92_TIME, SQL_DL_SQL92_TIMESTAMP
SQL_DDL_INDEX	SQL_DI_CREATE_INDEX, SQL_DI_DROP_INDEX
SQL_DESCRIBE_PARAMETER	"Y": Parameters can be described.
SQL_DM_VER	ERROR IM001: Driver does not support this function. Applies to driver manager only.
SQL_DRIVER_HDESC	Pointer to driver descriptor handle.
SQL_DROP_ASSERTION	0: DROP ASSERTION statement not supported.
SQL_DROP_CHARACTER_SET	0: DROP_CHARACTER_SET statement not supported.
SQL_DROP_COLLATION	0: DROP_COLLATION statement not supported.
SQL_DROP_DOMAIN	0: DROP_DOMAIN statement not supported.

Table 10-9 (Cont.) TimesTen Support for Standard Information Types: SQLGetInfo (ODBC 3.5)

Information Type	Notes and Correct Values Returned by TimesTen
SQL_DROP_SCHEMA	0: DROP_SCHEMA statement not supported.
SQL_DROP_TABLE	SQL_DT_DROP_TABLE
SQL_DROP_TRANSLATION	0: DROP_TRANSLATION statement not supported.
SQL_DROP_VIEW	SQL_DV_DROP_VIEW
SQL_DYNAMIC_CURSOR_ATTRIBUTES1	None: Dynamic cursors not supported.
SQL_DYNAMIC_CURSOR_ATTRIBUTES2	None: Dynamic cursors not supported.
SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES1	SQL_CA1_NEXT, SQL_CA1_SELECT_FOR_UPDATE
SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES2	SQL_CA2_READ_ONLY_CONCURRENCY, SQL_CA2_MAX_ROWS_SELECT
SQL_INDEX_KEYWORDS	SQL_IK_ALL: All keywords supported.
SQL_INFO_SCHEMA_VIEWS	None: Views in the INFORMATION_SCHEMA not supported.
SQL_INSERT_STATEMENT	SQL_IS_INSERT_LITERALS, SQL_IS_INSERT_SEARCHED, SQL_IS_SELECT_INTO
SQL_INTEGRITY	"N" SQL_ODBC_SQL_OPT_IEF in ODBC 2.5.
SQL_KEYSET_CURSOR_ATTRIBUTES1	None: Keyset cursors not supported.
SQL_KEYSET_CURSOR_ATTRIBUTES2	None: Keyset cursors not supported.
SQL_KEYWORDS	TT_SQL_KEYWORDS: A character string that contains a comma-separated list of TimesTen-specific SQL keywords. See TimesTen SQL Keywords for ODBC 3.5.
SQL_MAX_ASYNC_CONCURRENT_STATEMENTS	0: No specific limit to number of active concurrent statements in asynchronous mode.
SQL_MAX_CATALOG_NAME_LEN Alias SQL_MAXIMUM_CATALOG_NAME_LENGTH	0: No specific maximum length. SQL_MAX_QUALIFIER_NAME_LEN in ODBC 2.5.
SQL_MAX_CONCURRENT_ACTIVITIES Alias SQL_MAXIMUM_CONCURRENT_ACTIVITIES	0: Allocated from heap, no limit on concurrency. SQL_ACTIVE_STATEMENTS in ODBC 2.5.
SQL_MAX_DRIVER_CONNECTIONS Alias SQL_MAXIMUM_DRIVER_CONNECTIONS	sb_DbConnMaxUser: Daemon connections limited to this value. SQL_ACTIVE_CONNECTIONS in ODBC 2.5.
SQL_MAX_IDENTIFIER_LEN Alias SQL_MAXIMUM_IDENTIFIER_LENGTH	sb_ObjNameLenMax
SQL_MAX_ROW_SIZE_INCLUDES_LONG	"N"
SQL_MAX_SCHEMA_NAME_LEN Alias SQL_MAXIMUM_SCHEMA_NAME_LENGTH	sb_ObjNameLenMax SQL_MAX_OWNER_NAME_LEN in ODBC 2.5.

Table 10-9 (Cont.) TimesTen Support for Standard Information Types: SQLGetInfo (ODBC 3.5)

Information Type	Notes and Correct Values Returned by TimesTen
SQL_ODBC_INTERFACE_CONFORMANCE	SQL_OIC_CORE: Minimum level, including basic interface elements such as connection functions, functions for preparing and executing an SQL statement, basic result set metadata functions, and basic catalog functions.
SQL_PARAM_ARRAY_ROW_COUNTS	SQL_PARC_NO_BATCH
SQL_PARAM_ARRAY_SELECTS	SQL_PAS_NO_SELECT
SQL_SCHEMA_TERM	"owner" SQL_OWNER_TERM in ODBC 2.5.
SQL_SCHEMA_USAGE	SQL_OU_DML_STATEMENTS: Schemas supported in all DML statements. SQL_OU_PROCEDURE_INVOCATION: Schemas supported in the ODBC procedure invocation statement. SQL_OU_TABLE_DEFINITION: Schemas supported in CREATE TABLE, CREATE VIEW, ALTER TABLE, DROP TABLE, and DROP VIEW statements. SQL_OU_INDEX_DEFINITION: Schemas supported in CREATE INDEX and DROP INDEX statements. SQL_OU_PRIVILEGE_DEFINITION: Schemas are supported in GRANT and REVOKE statements. SQL_OWNER_USAGE in ODBC 2.5.
SQL_SQL_CONFORMANCE	SQL_SC_SQL92_ENTRY: Entry level SQL-92 compliant.
SQL_SQL92_DATETIME_FUNCTIONS	None: Datetime scalar functions not supported.
SQL_SQL92_FOREIGN_KEY_DELETE_RULE	SQL_SFKD_CASCADE
SQL_SQL92_FOREIGN_KEY_UPDATE_RULE	SQL_SFKU_SET_DEFAULT, SQL_SFKU_SET_NULL
SQL_SQL92_GRANT	SQL_SG_DELETE_TABLE, SQL_SG_INSERT_TABLE, SQL_SG_REFERENCES_TABLE, SQL_SG_SELECT_TABLE, SQL_SG_UPDATE_TABLE (all entry level)
SQL_SQL92_NUMERIC_VALUE_FUNCTIONS	SQL_SNVF_EXTRACT
SQL_SQL92_PREDICATES	SQL_SP_BETWEEN, SQL_SP_COMPARISON, SQL_SP_EXISTS, SQL_SP_IN, SQL_SP_ISNOTNULL, SQL_SP_ISNULL, SQL_SP_LIKE (all entry level)
SQL_SQL92_RELATIONAL_JOIN_OPERATORS	SQL_SRJO_CROSS_JOIN (full level), SQL_SRJO_INNER_JOIN (FIPS transitional level), SQL_SRJO_LEFT_OUTER_JOIN (FIPS transitional level), SQL_SRJO_RIGHT_OUTER_JOIN (FIPS transitional level)

Table 10-9 (Cont.) TimesTen Support for Standard Information Types: SQLGetInfo (ODBC 3.5)

Information Type	Notes and Correct Values Returned by TimesTen
SQL_SQL92_REVOKE	SQL_SR_DELETE_TABLE, SQL_SR_INSERT_TABLE, SQL_SR_REFERENCES_TABLE, SQL_SR_SELECT_TABLE, SQL_SR_UPDATE_TABLE (all entry level)
SQL_SQL92_ROW_VALUE_CONSTRUCTOR	None: Row value constructor expressions not supported.
SQL_SQL92_STRING_FUNCTIONS	None: String scalar functions not supported.
SQL_SQL92_VALUE_EXPRESSIONS	SQL_SVE_CASE (intermediate level), SQL_SVE_CAST (FIPS transitional level), SQL_SVE_NULLIF (intermediate level)
SQL_STANDARD_CLI_CONFORMANCE	None: Driver does not conform to CLI standards.
SQL_STATIC_CURSOR_ATTRIBUTES1	SQL_CA1_NEXT, SQL_CA1_SELECT_FOR_UPDATE
SQL_STATIC_CURSOR_ATTRIBUTES2	SQL_CA2_READ_ONLY_CONCURRENCY, SQL_CA2_MAX_ROWS_SELECT
SQL_TIMEDATE_FUNCTIONS	SQL_FN_TD_EXTRACT, SQL_FN_TD_NOW, SQL_FN_TD_TIMESTAMPADD, SQL_FN_TD_TIMESTAMPDIFF
SQL_UNION_STATEMENT	SQL_U_UNION: Data source supports UNION clause. SQL_U_UNION_ALL: Data source supports ALL keyword in the UNION clause. (SQLGetInfo returns both SQL_U_UNION and SQL_U_UNION_ALL in this case.) SQL_UNION in ODBC 2.5.
SQL_XOPEN_CLI_YEAR	ERROR IM001: Driver does not support this function. Applies to driver manager only.

Table 10-10 describes TimesTen-specific information types.

Table 10-10 TimesTen Information Types: SQLGetInfo

Information Type	Data Type	Description
TT_DATA_STORE_INVALID	SQLINTEGER	Returns 1 if the database is in invalid state, such as due to a system or application failure, or 0 if not. **Note**: Fatal errors, such as error 846 or 994, invalidate a TimesTen database, causing this item to be set to 1.
TT_DATABASE_CHARACTER_SET	SQLCHAR	Returns the name of the database character set.
TT_DATABASE_CHARACTER_SET_SIZE	SQLINTEGER	Returns the maximum size of a character in the database character set, in bytes.

Table 10-10 (Cont.) TimesTen Information Types: SQLGetInfo

Information Type	Data Type	Description
TT_PLATFORM_INFO	Bit mask	Returns a bit mask indicating platform information. Bit 0 has the value 1 for a 64-bit platform. Bit 1 has the value 1 for big-endian, or the value 0 for little-endian.
TT_REPLICATION_INVALID	SQLINTEGER	Returns 1 if replication is in a failed state, or 0 if not. See Subscriber Failures in *Oracle TimesTen In-Memory Database Replication Guide*.

TimesTen SQL Keywords for ODBC 3.5

The list of TimesTen SQL keywords returned for SQL_KEYWORDS in a SQLGetInfo call is the same in TimesTen ODBC 3.5 support as in ODBC 2.5 support.

See TimesTen SQL Keywords for ODBC 2.5.

This is different from the list of TimesTen reserved words. See Reserved Words in *Oracle TimesTen In-Memory Database SQL Reference*.

TimesTen ODBC 2.5 Support

This section covers these topics for TimesTen 2.5 support.

- Using ODBC 2.5 With TimesTen
- ODBC 2.5 Function Support
- Option Support for ODBC 2.5 SQLSetConnectOption and SQLGetConnectOption
- Option Support for ODBC 2.5 SQLSetStmtOption and SQLGetStmtOption
- Column Descriptor Support for ODBC 2.5 SQLColAttributes
- Information Type Support for ODBC 2.5 SQLGetInfo
- TimesTen SQL Keywords for ODBC 2.5

Using ODBC 2.5 With TimesTen

An ODBC 2.5 application not using a driver manager will continue to work with the TimesTen ODBC driver through its call to SQLAllocEnv.

Tip:

Because TimesTen Release 22.1 is a major release, you should recompile and relink existing ODBC applications. Also see ODBC API Incompatibilities With Previous Versions of TimesTen.

It is also advisable to link your applications dynamically rather than statically.

ODBC 2.5 Function Support

TimesTen supports certain ODBC 2.5 functions.

Note:

- The TimesTen ODBC driver supports wide-character (W) function versions for applications not using a generic driver manager, as indicated in Table 10-11.
- In ODBC 2.5, TimesTen supports some ODBC 3.0 handle types (such as `SQLHDBC` and `SQLHENV`) as well as ODBC 2.0 handle types (such as `HDBC` and `HENV`). TimesTen recommends using ODBC 3.0 handle types. The `FAR` modifier that is mentioned in ODBC 2.0 documentation is not required.

Table 10-11 Supported ODBC 2.5 Functions

Function	Notes
`SQLAllocConnect`	No notes
`SQLAllocEnv`	No notes
`SQLAllocStmt`	No notes
`SQLBindCol`	No notes
`SQLBindParameter`	See SQLBindParameter Function.
`SQLCancel`	`SQLCancel` can cancel the following: • An operation running on an *`hstmt`* on another thread • An operation running on an *`hstmt`* that needs data `SQLCancel` *cannot* cancel the following: • Cache administrative operations Do not call `SQLCancel` directly from a signal handler. Such code may not be portable.
`SQLColAttributes` and `SQLColAttributesW`	See Column Descriptor Support for ODBC 2.5 SQLColAttributes. Also see ODBC 2.5 Function Signatures That Have Changed.
`SQLColumnPrivileges`	Call returns "driver not capable".

Table 10-11 (Cont.) Supported ODBC 2.5 Functions

Function	Notes
SQLColumns and SQLColumnsW	For catalog functions, TimesTen supports only an empty string or NULL as the qualifier.
SQLConnect and ttSQLConnectW	Note the TimesTen name for the "W" function.
SQLDataSources and SQLDataSourcesW	Available only to programs using a driver manager.
SQLDescribeCol and SQLDescribeColW	No notes
SQLDescribeParam	No notes
SQLDisconnect	No notes
SQLDriverConnect and SQLDriverConnectW	No notes
SQLDrivers and SQLDriversW	Available only to programs using a driver manager.
SQLError and SQLErrorW	Native error codes are TimesTen errors. You may receive generic errors such as, "Execution at Oracle failed. Oracle error code *nnn*." When using SQLError or SQLErrorW: • Use TT_MAX_MESSAGE_LENGTH (which is a higher limit) instead of SQL_MAX_MESSAGE_LENGTH (which is a limit of 512 bytes). • Handle a possible return of SQL_SUCCESS_WITH_INFO (for example, in case the message length exceeded the input buffer size).
SQLExecDirect	See SQLExecute.
SQLExecute	TimesTen does not support asynchronous statement execution. (TimesTen does not support the SQL_ASYNC_ENABLE statement option, as noted later in this chapter.)
SQLFetch	The return code is defined as SQL_NO_DATA_FOUND when no more rows are returned. SQL_NO_DATA_FOUND is defined in sqlext.h, which is included by timesten.h.
SQLForeignKeys and SQLForeignKeysW	For catalog functions, TimesTen supports only an empty string or NULL as the qualifier.
SQLFreeConnect	No notes
SQLFreeEnv	No notes
SQLFreeStmt	No notes

Table 10-11 (Cont.) Supported ODBC 2.5 Functions

Function	Notes
`SQLGetConnectOption` and `SQLGetConnectOptionW`	See Option Support for ODBC 2.5 SQLSetConnectOption and SQLGetConnectOption. Also see ODBC 2.5 Function Signatures That Have Changed. Support is added for the TimesTen driver manager (TTDM). This is documented in Attribute Support for ODBC 3.5 SQLSetConnectAttr and SQLGetConnectAttr.
`SQLGetCursorName` and `SQLGetCursorNameW`	You can set or get a cursor name but not reference it, such as in a `WHERE CURRENT OF` clause for a positioned update or delete. TimesTen does not support positioned update or delete statements.
`SQLGetData`	See Avoid SQLGetData.
`SQLGetFunctions`	No notes
`SQLGetInfo` and `SQLGetInfoW`	See Information Type Support for ODBC 2.5 SQLGetInfo. Also see ODBC 2.5 Function Signatures That Have Changed.
`SQLGetStmtOption`	See Option Support for ODBC 2.5 SQLSetStmtOption and SQLGetStmtOption. Also see ODBC 2.5 Function Signatures That Have Changed.
`SQLGetTypeInfo` and `SQLGetTypeInfoW`	For catalog functions, TimesTen supports only an empty string or `NULL` as the qualifier.
`SQLNativeSql` and `SQLNativeSqlW`	No notes
`SQLNumParams`	No notes
`SQLNumResultCols`	No notes
`SQLParamData`	No notes
`SQLParamOptions`	See ODBC 2.5 Function Signatures That Have Changed.
`SQLPrepare` and `SQLPrepareW`	No notes
`SQLPrimaryKeys` and `SQLPrimaryKeysW`	For catalog functions, TimesTen supports only an empty string or `NULL` as the qualifier.
`SQLProcedureColumns` and `SQLProcedureColumnsW`	For catalog functions, TimesTen supports only an empty string or `NULL` as the qualifier.
`SQLProcedures` and `SQLProceduresW`	For catalog functions, TimesTen supports only an empty string or `NULL` as the qualifier.
`SQLPutData`	No notes
`SQLRowCount`	In addition to its standard functionality, this is used with TimesTen cache groups. See Retrieving Information About Cache Groups.
`SQLSetConnectOption` and `SQLSetConnectOptionW`	See Option Support for ODBC 2.5 SQLSetConnectOption and SQLGetConnectOption under the next section. Also see ODBC 2.5 Function Signatures That Have Changed.

Table 10-11 (Cont.) Supported ODBC 2.5 Functions

Function	Notes
SQLSetCursorName and SQLSetCursorNameW	You can set or get a cursor name but not reference it, such as in a WHERE CURRENT OF clause for a positioned update or delete.
SQLSetParam	This is an ODBC 1.0 function, replaced by SQLBindParameter in ODBC 2.0. Retained for backward compatibility.
SQLSetPos	Call returns "driver not capable".
SQLSetStmtOption	See Option Support for ODBC 2.5 SQLSetStmtOption and SQLGetStmtOption. Also see ODBC 2.5 Function Signatures That Have Changed.
SQLSpecialColumns and SQLSpecialColumnsW	TimesTen supports only the SQL_BEST_ROWID option. For catalog functions, TimesTen supports only an empty string or NULL as the qualifier.
SQLStatistics and SQLStatisticsW	For catalog functions, TimesTen supports only an empty string or NULL as the qualifier.
SQLTablePrivileges	Call returns "driver not capable".
SQLTables and SQLTablesW	For catalog functions, TimesTen supports only an empty string or NULL as the qualifier.
SQLTransact	No notes

Option Support for ODBC 2.5 SQLSetConnectOption and SQLGetConnectOption

The ODBC 2.5 SQLSetConnectOption and SQLGetConnectOption functions enable you to set connection options after the initial connection or retrieve those settings. Some of these correspond to connection attributes you can set during the connection process.

Table 10-12 lists standard options supported by TimesTen for The ODBC 2.5 SQLSetConnectOption and SQLGetConnectOption functions.

For TimesTen-specific connection options, see Table 10-4. These options are supported for both ODBC 2.5 and ODBC 3.5.

Also see Option Support for ODBC 2.5 SQLSetStmtOption and SQLGetStmtOption. Those options can also be set using SQLSetConnectOption, in which case the value serves as a default for all statements on the connection.

Note:

- An option setting through `SQLSetConnectOption` or `SQLSetStmtOption` overrides the setting of the corresponding connection attribute (as applicable).
- The documentation here also applies to `SQLSetConnectOptionW` and `SQLGetConnectOptionW`.
- Where TimesTen connection attributes are mentioned as being equivalent to ODBC connection options, see Connection Attributes in *Oracle TimesTen In-Memory Database Reference*.

Table 10-12 Standard connection options (ODBC 2.5)

Option	Notes
`SQL_AUTOCOMMIT`	No notes
`SQL_MAX_ROWS`	See ODBC 2.5 Function Signatures That Have Changed (refer to `SQLGetStmtOption` or `SQLSetStmtOption` there).
`SQL_NOSCAN`	No notes
`SQL_ODBC_CURSORS`	Supported for programs using a driver manager
`SQL_OPT_TRACE`	Supported for programs using a driver manager
`SQL_OPT_TRACEFILE`	Supported for programs using a driver manager
`SQL_TXN_ISOLATION`	Supported for *`vParam`* is `SQL_TXN_READ_COMMITTED` or `SQL_TXN_SERIALIZABLE` See Prefetching Multiple Rows of Data for information about the relationship between prefetching and isolation level. Also see Concurrency Control Through Isolation and Locking in *Oracle TimesTen In-Memory Database Operations Guide* and Isolation in *Oracle TimesTen In-Memory Database Reference*.

Option Support for ODBC 2.5 SQLSetStmtOption and SQLGetStmtOption

The ODBC 2.5 `SQLSetStmtOption` and `SQLGetStmtOption` functions enable you to set or retrieve statement option settings.

Table 10-13 lists standard options supported by TimesTen for the ODBC 2.5 `SQLSetStmtOption` and `SQLGetStmtOption` functions, with notes about the support.

For TimesTen-specific statement options, see Table 10-6. These options are supported for both ODBC 2.5 and ODBC 3.5.

To set an option default value for all statements associated with a connection, use `SQLSetConnectOption`.

Note:

An option setting through `SQLSetConnectOption` or `SQLSetStmtOption` overrides the setting of the corresponding connection attribute (as applicable).

Table 10-13 Standard Statement Options (ODBC 2.5)

Option	Notes
`SQL_MAX_ROWS`	See ODBC 2.5 Function Signatures That Have Changed.
`SQL_NOSCAN`	No notes
`SQL_QUERY_TIMEOUT`	See Setting a Timeout Duration for SQL Statements.

Note:

The `SQL_MAX_LENGTH` option can be set, but any specified value is overridden with 0 (return all available data).

Column Descriptor Support for ODBC 2.5 SQLColAttributes

The `SQLColAttributes` function returns descriptor information for a column in a result set.

Refer to ODBC API reference documentation for complete information about this function and standard column descriptors.

Table 10-14 describes TimesTen-specific column descriptors.

Table 10-14 TimesTen Column Descriptors: SQLColAttributes

Descriptor	Comment/Description
`TT_COLUMN_INLINE`	Returns `TRUE` for columns with inline data, or `FALSE` otherwise. This is returned in the `SQLColAttributes` *`pfDesc`* parameter.
`TT_COLUMN_LENGTH_SEMANTICS`	For character-type columns, this returns "BYTE" for columns with byte length semantics and "CHAR" for columns with character length semantics. For non-character columns, it returns "". The information is returned in the `SQLColAttributes` *`rgbDesc`* parameter. This information refers to whether data length is measured in bytes or characters. Length semantics in TimesTen are the same as in Oracle Database. See Length Semantics in *Oracle Database Globalization Support Guide*.

Information Type Support for ODBC 2.5 SQLGetInfo

There is support in the TimesTen ODBC 2.5 implementation for information types for the ODBC function SQLGetInfo.

Table 10-15 documents TimesTen support for standard information types introduced in ODBC 1.0 and 2.0, as well as ODBC 3.0 information types supported by the TimesTen ODBC 2.5 implementation (as indicated), noting the TimesTen-specific correct value or values returned.

See Information Type Support for ODBC 3.5 SQLGetInfo for TimesTen-specific information types, which are supported for both ODBC 3.5 and ODBC 2.5.

Table 10-15 TimesTen Support for Standard Information Types: SQLGetInfo (ODBC 2.5)

Information Type	Notes and Correct Values Returned by TimesTen
SQL_ACCESSIBLE_PROCEDURES	"Y"
SQL_ACCESSIBLE_TABLES	"Y"
SQL_ACTIVE_CONNECTIONS	sb_DbConnMaxUser: Daemon connections limited to this value.
SQL_ACTIVE_STATEMENTS	0: Allocated from heap, no limit on concurrency.
SQL_AGGREGATE_FUNCTIONS	ODBC 3.0 information type supported by TimesTen ODBC 2.5 implementation. See Information Type Support for ODBC 3.5 SQLGetInfo.
SQL_ALTER_TABLE	SQL_AT_ADD_COLUMN, SQL_AT_DROP_COLUMN
SQL_BOOKMARK_PERSISTENCE	0: Bookmarks persist through none of the operations.
SQL_COLUMN_ALIAS	"Y"
SQL_CONCAT_NULL_BEHAVIOR	SQL_CB_NON_NULL: Result is concatenation of column or columns with non-null values.
SQL_CONVERT_FUNCTIONS	SQL_FN_CVT_CAST
SQL_CONVERT_*xxxx*	0: CONVERT function not supported.
SQL_CONVERT_WCHAR	ODBC 3.0 information type supported by TimesTen ODBC 2.5 implementation. See Information Type Support for ODBC 3.5 SQLGetInfo.
SQL_CONVERT_WLONGVARCHAR	ODBC 3.0 information type supported by TimesTen ODBC 2.5 implementation. See Information Type Support for ODBC 3.5 SQLGetInfo.
SQL_CONVERT_WVARCHAR	ODBC 3.0 information type supported by TimesTen ODBC 2.5 implementation. See Information Type Support for ODBC 3.5 SQLGetInfo.
SQL_CORRELATION_NAME	SQL_CN_ANY: Correlation names are supported and can be any valid user-defined name.

Table 10-15 (Cont.) TimesTen Support for Standard Information Types: SQLGetInfo (ODBC 2.5)

Information Type	Notes and Correct Values Returned by TimesTen
`SQL_CREATE_VIEW`	ODBC 3.0 information type supported by TimesTen ODBC 2.5 implementation. See Information Type Support for ODBC 3.5 SQLGetInfo.
`SQL_CURSOR_COMMIT_BEHAVIOR`	`SQL_CB_CLOSE`: Close cursors. For prepared statements, the application can call `SQLExecute` on the statement without calling `SQLPrepare` again.
`SQL_CURSOR_ROLLBACK_BEHAVIOR`	`SQL_CB_CLOSE`: Close cursors. For prepared statements, the application can call `SQLExecute` on the statement without calling `SQLPrepare` again.
`SQL_DATA_SOURCE_NAME`	"": Empty string.
`SQL_DATA_SOURCE_READ_ONLY`	"N"
`SQL_DATETIME_LITERALS`	ODBC 3.0 information type supported by TimesTen ODBC 2.5 implementation. See Information Type Support for ODBC 3.5 SQLGetInfo.
`SQL_DEFAULT_TXN_ISOLATION`	`SQL_TXN_READ_COMMITTED`: Dirty reads are not possible. Non-repeatable reads and phantoms are possible. `SQL_TXN_SERIALIZABLE`: Transactions are serializable. Dirty reads, non-repeatable reads, or phantoms are now allowed.
`SQL_DRIVER_HDBC`	Pointer to driver connection handle.
`SQL_DRIVER_HENV`	Pointer to driver environment handle.
`SQL_DRIVER_HLIB`	`NULL` **Note**: If you use a driver manager, this returns the pointer to the TimesTen library.
`SQL_DRIVER_HSTMT`	Pointer to driver statement handle.
`SQL_DRIVER_NAME`	The file name of the TimesTen ODBC driver library for your platform.
`SQL_DRIVER_ODBC_VER`	"3.51" for ODBC 3.5; "2.50" for ODBC 2.5.
`SQL_DRIVER_VER`	A string indicating the TimesTen version. For example, "22.01.0001.0001 Oracle TimesTen version 22.1.1.1.0".
`SQL_DROP_VIEW.`	ODBC 3.0 information type supported by TimesTen ODBC 2.5 implementation. See Information Type Support for ODBC 3.5 SQLGetInfo.
`SQL_EXPRESSIONS_IN_ORDERBY`	"Y"
`SQL_FETCH_DIRECTION`	`SQL_FD_FETCH_NEXT`

Table 10-15 (Cont.) TimesTen Support for Standard Information Types: SQLGetInfo (ODBC 2.5)

Information Type	Notes and Correct Values Returned by TimesTen
SQL_FILE_USAGE	SQL_FILE_NOT_SUPPORTED: Driver is not a single-tier driver.
SQL_GETDATA_EXTENSIONS	SQL_GD_ANY_COLUMN: SQLGetData can be called for any unbound column, including those before the last bound column. The columns must be called in order of ascending column number unless SQL_GD_ANY_ORDER is also returned. SQL_GD_ANY_ORDER: SQLGetData can be called for unbound columns in any order. Note that SQLGetData can be called only for columns after the last bound column unless SQL_GD_ANY_COLUMN is also returned. SQL_GD_BOUND: SQLGetData can be called for bound columns in addition to unbound columns. A driver cannot return this value unless it also returns SQL_GD_ANY_COLUMN.
SQL_GROUP_BY	SQL_GB_GROUP_BY_CONTAINS_SELECT: GROUP BY clause must contain all nonaggregated columns in the select list, but can also contain columns that are not in the select list. For example: `SELECT dept, MAX(salary) FROM employee GROUP BY dept, age;`
SQL_IDENTIFIER_CASE	SQL_IC_UPPER: SQL identifiers are not case-sensitive and are stored in uppercase in system catalog.
SQL_IDENTIFIER_QUOTE_CHAR	""": A string with one quote mark, which is the quote character.
SQL_KEYWORDS	TT_SQL_KEYWORDS: A character string that contains a comma-separated list of TimesTen-specific SQL keywords. See TimesTen SQL Keywords for ODBC 2.5.
SQL_LIKE_ESCAPE_CLAUSE	"Y"
SQL_MAX_BINARY_LITERAL_LEN	16384
SQL_MAX_CHAR_LITERAL_LEN	YY_BUF_SIZE
SQL_MAX_COLUMN_NAME_LEN Alias SQL_MAXIMUM_COLUMN_NAME_LENGTH	sb_ObjNameLenMax
SQL_MAX_COLUMNS_IN_GROUP_BY Alias SQL_MAXIMUM_COLUMNS_IN_GROUP_BY	MAX_COLUMNS_IN_GB
SQL_MAX_COLUMNS_IN_INDEX	MAX_COLUMNS_IN_IDX

Table 10-15 (Cont.) TimesTen Support for Standard Information Types: SQLGetInfo (ODBC 2.5)

Information Type	Notes and Correct Values Returned by TimesTen
SQL_MAX_COLUMNS_IN_ORDER_BY Alias SQL_MAXIMUM_COLUMNS_IN_ORDER_BY	MAX_COLUMNS_IN_OB
SQL_MAX_COLUMNS_IN_SELECT Alias SQL_MAXIMUM_COLUMNS_IN_SELECT	MAX_COLUMNS_IN_SELECT
SQL_MAX_COLUMNS_IN_TABLE Alias SQL_MAXIMUM_COLUMNS_IN_TABLE	MAX_COLUMNS_IN_TBL
SQL_MAX_CURSOR_NAME_LEN Alias SQL_MAXIMUM_CURSOR_NAME_LENGTH	18
SQL_MAX_INDEX_SIZE	4194304
SQL_MAX_OWNER_NAME_LEN	sb_ObjNameLenMax
SQL_MAX_PROCEDURE_NAME_LEN	sb_NameLenMax - 1
SQL_MAX_QUALIFIER_NAME_LEN	0: No specific maximum length.
SQL_MAX_ROW_SIZE	4194304
SQL_MAX_STATEMENT_LEN Alias SQL_MAXIMUM_STATEMENT_LENGTH	sb_SqlStringLenMax
SQL_MAX_TABLE_NAME_LEN Alias SQL_MAXIMUM_TABLE_NAME_LENGTH	sb_ObjNameLenMax
SQL_MAX_TABLES_IN_SELECT Alias SQL_MAXIMUM_TABLES_IN_SELECT	sb_SqlCorrMax
SQL_MAX_USER_NAME_LEN Alias SQL_MAXIMUM_USER_NAME_LENGTH	sb_ObjNameLenMax
SQL_MULT_RESULT_SETS	"N"
SQL_MULTIPLE_ACTIVE_TXN	"Y"
SQL_NEED_LONG_DATA_LEN	"N"
SQL_NON_NULLABLE_COLUMNS	SQL_NNC_NON_NULL: Columns cannot be nullable. (The data source supports the NOT NULL column constraint in CREATE TABLE statements.)
SQL_NULL_COLLATION	SQL_NC_HIGH: Null values are sorted at the high end of the result set, depending on the ASC or DESC keyword.
SQL_NUMERIC_FUNCTIONS	SQL_FN_NUM_ABS, SQL_FN_NUM_CEILING, SQL_FN_NUM_FLOOR, SQL_FN_NUM_MOD, SQL_FN_NUM_POWER, SQL_FN_NUM_ROUND, SQL_FN_NUM_SIGN, SQL_FN_NUM_SQRT
SQL_ODBC_SQL_OPT_IEF	"N"
SQL_ODBC_VER	N/A, implemented by the driver manager.

Table 10-15 (Cont.) TimesTen Support for Standard Information Types: SQLGetInfo (ODBC 2.5)

Information Type	Notes and Correct Values Returned by TimesTen
SQL_OJ_CAPABILITIES Alias SQL_OUTER_JOIN_CAPABILITIES	SQL_OJ_LEFT: Left outer joins supported. SQL_OJ_RIGHT: Right outer joins supported. SQL_OJ_NOT_ORDERED: Column names in the ON clause of the outer join do not have to be in the same order as their respective table names in the OUTER JOIN clause. SQL_OJ_INNER: Inner table (right table in a left outer join or left table in a right outer join) can also be used in an inner join. This does not apply to full outer joins, which do not have an inner table. SQL_OJ_ALL_COMPARISON_OPS: Comparison operator in the ON clause can be any of the ODBC comparison operators. If this bit is not set, only the equals (=) comparison operator can be used in outer joins.
SQL_ORDER_BY_COLUMNS_IN_SELECT	"Y"
SQL_OUTER_JOINS	"Y"
SQL_OWNER_TERM	"owner"
SQL_OWNER_USAGE	SQL_OU_DML_STATEMENTS: Schemas supported in all DML statements. SQL_OU_PROCEDURE_INVOCATION: Schemas supported in the ODBC procedure invocation statement. SQL_OU_TABLE_DEFINITION: Schemas supported in CREATE TABLE, CREATE VIEW, ALTER TABLE, DROP TABLE, and DROP VIEW statements. SQL_OU_INDEX_DEFINITION: Schemas supported in CREATE INDEX and DROP INDEX statements. SQL_OU_PRIVILEGE_DEFINITION: Schemas are supported in GRANT and REVOKE statements.
SQL_PARAM_ARRAY_ROW_COUNTS	ODBC 3.0 information type supported by TimesTen ODBC 2.5 implementation. See Information Type Support for ODBC 3.5 SQLGetInfo.
SQL_PARAM_ARRAY_SELECTS	ODBC 3.0 information type supported by TimesTen ODBC 2.5 implementation. See Information Type Support for ODBC 3.5 SQLGetInfo.
SQL_POS_OPERATIONS	0: Scrollable cursors not supported.
SQL_PROCEDURE_TERM	"procedure"
SQL_PROCEDURES	"Y"
SQL_QUALIFIER_LOCATION	0: Catalog names as qualifiers not supported.

Table 10-15 (Cont.) TimesTen Support for Standard Information Types: SQLGetInfo (ODBC 2.5)

Information Type	Notes and Correct Values Returned by TimesTen
SQL_QUALIFIER_NAME_SEPARATOR	NULL: Not supported.
SQL_QUALIFIER_TERM	"data store"
SQL_QUALIFIER_USAGE	0: Catalogs not supported.
SQL_QUOTED_IDENTIFIER_CASE	SQL_IC_SENSITIVE: Quoted identifiers in SQL are case-sensitive and stored in mixed-case in the system catalog.
SQL_ROW_UPDATES	"N"
SQL_SCROLL_OPTIONS	SQL_SO_FORWARD_ONLY: Cursors can scroll only forward.
SQL_SEARCH_PATTERN_ESCAPE	"\\"
SQL_SERVER_NAME	"": Empty string.
SQL_SPECIAL_CHARACTERS	"@#$": A string indicating the special characters.
SQL_SQL92_RELATIONAL_JOIN_OPERATORS	ODBC 3.0 information type supported by TimesTen ODBC 2.5 implementation. See Information Type Support for ODBC 3.5 SQLGetInfo.
SQL_SQL92_VALUE_EXPRESSIONS	ODBC 3.0 information type supported by TimesTen ODBC 2.5 implementation. See Information Type Support for ODBC 3.5 SQLGetInfo.
SQL_STRING_FUNCTIONS	SQL_FN_STR_CHAR, SQL_FN_STR_CONCAT, SQL_FN_STR_LCASE, SQL_FN_STR_LEFT, SQL_FN_STR_LENGTH, SQL_FN_STR_LOCATE, SQL_FN_STR_LOCATE_2, SQL_FN_STR_LTRIM, SQL_FN_STR_REPLACE, SQL_FN_STR_RIGHT, SQL_FN_STR_RTRIM, SQL_FN_STR_SOUNDEX, SQL_FN_STR_SPACE, SQL_FN_STR_SUBSTRING, SQL_FN_STR_UCASE
SQL_SUBQUERIES	SQL_SQ_CORRELATED_SUBQUERIES, SQL_SQ_COMPARISON, SQL_SQ_EXISTS, SQL_SQ_IN, SQL_SQ_INSQL_SQ_QUANTIFIED
SQL_SYSTEM_FUNCTIONS	SQL_FN_SYS_IFNULL, SQL_FN_SYS_USERNAME
SQL_TABLE_TERM	"table"
SQL_TIMEDATE_ADD_INTERVALS	SQL_FN_TSI_FRAC_SECOND, SQL_FN_TSI_SECOND, SQL_FN_TSI_MINUTE, SQL_FN_TSI_HOUR, SQL_FN_TSI_DAY, SQL_FN_TSI_WEEK, SQL_FN_TSI_MONTH, SQL_FN_TSI_QUARTER, SQL_FN_TSI_YEAR
SQL_TIMEDATE_DIFF_INTERVALS	SQL_FN_TSI_FRAC_SECOND, SQL_FN_TSI_SECOND, SQL_FN_TSI_MINUTE, SQL_FN_TSI_HOUR, SQL_FN_TSI_DAY, SQL_FN_TSI_WEEK, SQL_FN_TSI_MONTH, SQL_FN_TSI_QUARTER, SQL_FN_TSI_YEAR

Table 10-15 (Cont.) TimesTen Support for Standard Information Types: SQLGetInfo (ODBC 2.5)

Information Type	Notes and Correct Values Returned by TimesTen
SQL_TIMEDATE_FUNCTIONS	SQL_FN_TD_TIMESTAMPADD, SQL_FN_TD_NOW, SQL_FN_TD_TIMESTAMPDIFF
SQL_TXN_CAPABLE Alias SQL_TRANSACTION_CAPABLE	SQL_TC_DDL_COMMIT: According to the ODBC 2.0 standard, this indicates that transactions can contain only DML statements, and that DDL statements encountered in a transaction cause the transaction to be committed. TimesTen implements Oracle Database semantics, which allow both DML and DDL in a transaction, but a DDL statement causes the transaction to commit.
SQL_TXN_ISOLATION_OPTION Alias SQL_TRANSACTION_ISOLATION_OPTION	SQL_TXN_READ_COMMITTED, SQL_TXN_SERIALIZABLE
SQL_UNION	SQL_U_UNION: Data source supports UNION clause. SQL_U_UNION_ALL: Data source supports ALL keyword in the UNION clause. (SQLGetInfo returns both SQL_U_UNION and SQL_U_UNION_ALL in this case.)
SQL_USER_NAME	At runtime, returns a string containing the user name.

Note:

If you use *InfoType* value SQL_DRIVER_HDBC, SQL_DRIVER_HENV, or SQL_DRIVER_HSTMT, refer to ODBC 2.5 Function Signatures That Have Changed.

TimesTen SQL Keywords for ODBC 2.5

This section lists the TimesTen SQL keywords returned for SQL_KEYWORDS in a SQLGetInfo call.

This is different from the list of TimesTen reserved words. See Reserved Words in *Oracle TimesTen In-Memory Database SQL Reference*.

```
ABS, ACCOUNT, ACTIVE, ADDMONTHS, ADMIN, AFFINITY, AGENT, AGING, ALLOW,
ASCIISTR, ASYNCHRONOUS, AUTHID, AUTOREFRESH, AWT, BATCH, BIG, BIGINT,
BINARY, BINARY_DOUBLE, BINARY_DOUBLE_INFINITY, BINARY_DOUBLE_NAN,
BINARY_FLOAT, BINARY_FLOAT_INFINITY, BINARY_FLOAT_NAN, BITAND, BITMAP,
BITNOT, BITOR, BITXOR, BLOB, BODY, BYTE, BYTES, CACHE, CACHEONLY,
CACHE_MANAGER, CALL, CHECKING, CHR, CLOB, COLUMNAR, COMMITTED, COMPILE,
COMPLETE, COMPRESS, CONCAT, CONFLICT, CONFLICTS, CS, CUBE, CURRENT_SCHEMA,
CURRVAL, CYCLE, DATASTORE, DATASTORE_OWNER, DAYS, DEBUG, DECODE, DEFINED,
```

```
DEFINER, DEFINITION, DELETE_FT, DESTROY, DICTIONARY, DIRECTORY, DISABLE,
DISTRIBUTE, DUPLICATE, DURABLE, DURATION, DYNAMIC, ELEMENT, ENABLE, ENCRYPTED,
ENDSEQ, EVERY, EXACT, EXCLUDE, EXIT, EXPIRE, EXTERNALLY, FACTOR, FAILTHRESHOLD,
FAST, FIRST_VALUE, FLUSH, FOLLOWING, FORCE, FORMAT, FUNCTION, GETDATE, GRID,
GROUPING, GROUPING_ID, GROUP_ID, HASH, HEARTBEAT, HIERARCHY, HOURS, ID,
IDENTIFIED, IGNORE, INCREMENT, INCREMENTAL, INFINITE, INLINE, INSERTONLY,
INSTANCE, INSTR, INSTR4, INSTRB, LAST_VALUE, LATENCY, LENGTH, LENGTH4, LENGTHB,
LIBRARY, LIFETIME, LIMIT, LIMIT_FT, LOAD, LOAD_FT, LOCK, LOG, LONG, LRU, MASTER,
MASTERIP, MATCHED, MATERIALIZED, MAXVALUE, MAXVALUES, MERGE, MIGRATORY,
MILLISECOND, MILLISECONDS, MINUS, MINUTES, MINVALUE, MOD, MODE, MODIFY, MULTI,
NAME, NAN, NCHAR_CS, NCHR, NCLOB, NEXTVAL, NLSSORT, NOBATCH, NOCACHE, NOCYCLE,
NOMAXVALUE, NOMINVALUE, NONDURABLE, NOORDER, NOWAIT, NULLS, NUMBER,
NUMTODSINTERVAL, NUMTOYMINTERVAL, NVARCHAR, NVARCHAR2, NVL, OFF, OPTIMIZED,
ORACLE, ORA_CHAR, ORA_DATE, ORA_FLOAT, ORA_NCHAR, ORA_NVARCHAR2, ORA_SYSDATE,
ORA_TIMESTAMP, ORA_VARCHAR2, OUT, OVER, PACKAGE, PAGES, PAIR, PARALLEL,
PARTITION, PASSWORD, PAUSED, PLSQL_WARNINGS, PORT, PRECEDING, PRIORITY, PRIVATE,
PROPAGATE, PROPAGATOR, PUBLICREAD, PUBLICROW, QUIT, RANGE, RC, READERS,
READONLY, RECEIPT, REFERENCE, REFRESH, REFRESH_FT, RELAXED, RELEASE, RENAME,
REPLACE, REPLICATION, REPORT, REPORTING, REQUEST, RESUME, RETURN, RETURNING,
REUSE, RLE, ROLLUP, ROUTE, ROW, ROWID, ROWIDONLY, ROWNUM, RR, RTRIM, RU,
SECONDS, SELF, SEQBATCH, SEQCACHE, SEQUENCE, SERVICES, SETS, SETTINGS,
SPECIFICATION, SQL_TSI_DAY, SQL_TSI_FRAC_SECOND, SQL_TSI_HOUR, SQL_TSI_MINUTE,
SQL_TSI_MONTH, SQL_TSI_QUARTER, SQL_TSI_SECOND, SQL_TSI_WEEK, SQL_TSI_YEAR,
STANDARD, STANDBY, START, STARTSEQ, STATE, STATIC, STOPPED, STORE, SUBSCRIBER,
SUBSCRIBERIP, SUBSTR, SUBSTR4, SUBSTRB, SUSPEND, SYNCHRONOUS, SYNONYM, SYSDATE,
SYSDBA, SYSTEM, TAG, TIMEOUT, TIMESTAMPADD, TIMESTAMPDIFF, TINYINT, TO_BLOB,
TO_CHAR, TO_CLOB, TO_DATE, TO_LOB, TO_NCLOB, TO_NUMBER, TO_TIMESTAMP, TRAFFIC,
TRANSMIT, TREE, TRUNC, TRUNCATE, TRUSTED, TT_BIGINT, TT_BINARY, TT_CHAR,
TT_DATE, TT_DECIMAL, TT_HASH, TT_INT, TT_INTEGER, TT_INTERVAL, TT_NCHAR,
TT_NVARCHAR, TT_SMALLINT, TT_SYSDATE, TT_TIME, TT_TIMESTAMP, TT_TINYINT,
TT_VARBINARY, TT_VARCHAR, TWOSAFE, UID, UNBOUNDED, UNISTR, UNLOAD, UNLOCK, USE,
USERMANAGED, VARBINARY, VARCHAR2, WAIT, WRAPPED, WRITETHROUGH, XLA, XML, XYZZY
```

ODBC API Incompatibilities With Previous Versions of TimesTen

There are changes introduced in the TimesTen 18.1 release that impact ODBC applications used with previous versions of TimesTen.

These topics are covered:

- Overview of ODBC API Incompatibilities
- ODBC 3.5 Function Signatures That Have Changed
- ODBC 2.5 Function Signatures That Have Changed
- ODBC Data Types That Have Changed

Overview of ODBC API Incompatibilities

The TimesTen driver is ODBC-compliant; however, beginning in Release 18.1, more recent ODBC header files are provided in the `include` directory of the TimesTen installation on Linux and UNIX platforms.

Changes were also made to update some ODBC types and functions to make them 64-bit compatible.

These and other changes may necessitate code changes on any platform. ODBC changes requiring code updates for ODBC applications fall into the following categories:

- ODBC function changes
  - Function signature changes: A number of function signatures have changed for 64-bit programming.
  - Changes to the size of option or attribute values: This refers to values of connection options, statement options, column attributes, or driver and data source information, either passed or returned. These are now 64-bit values in the circumstances indicated below.
- ODBC data type changes

> **Tip:**
>
> Even if none of the required code changes applies to your applications, you should recompile and relink existing ODBC applications the first time you use a TimesTen 22.1 release.

If your existing TimesTen ODBC application uses features described in the sections that follow, you must update the application as necessary:

ODBC 3.5 Function Signatures That Have Changed

There are ODBC 3.5 functions with changes requiring code updates.

In previous releases, TimesTen provided partial support for ODBC 3.5 functionality, including:

- Handle allocation methods
- Diagnostic records
- Wide character functions
- Attribute set and get functions for handles
- `SQLColAttribute`
- Miscellaneous functions that map directly to 2.5 functionality such as `SQLCloseCursor` and `SQLEndTran`

The functions listed in Table 10-16 have changes to the signature or changes to the size of attribute values, requiring code updates for ODBC 3.5 applications, as indicated. Sizes of attribute values apply to values of connection and statement attributes, either passed or returned.

Note:

- Signature changes apply to either 64-bit or 32-bit environments. Size changes in option and attribute values apply only to 64-bit environments.
- TimesTen ODBC does not return values for options or attributes related to features that TimesTen does not support. For example: `SQL_ATTR_ASYNC_ENABLE`, `SQL_ATTR_ENLIST_IN_DTC`, `SQL_ATTR_CURSOR_SCROLLABLE`, `SQL_ATTR_CURSOR_SENSITIVITY`, `SQL_ATTR_FETCH_BOOKMARK_PTR`, `SQL_ATTR_METADATA_ID`, `SQL_ATTR_RETRIEVE_DATA`, `SQL_ATTR_SIMULATE_CURSOR`, `SQL_ATTR_USE_BOOKMARKS`.

Table 10-16 Changes in ODBC 3.5 Functions

Function	Signature Changes	Size Changes in Option and Attribute Values

`SQLColAttribute`
`SQLColAttributeW`
	N/A	On UNIX platforms: For the following *`FieldIdentifier`* values, a 64-bit value is returned in *`*NumericAttributePtr`*:
		`SQL_DESC_AUTO_UNIQUE_VALUE`
		`SQL_DESC_CASE_SENSITIVE`
		`SQL_DESC_CONCISE_TYPE`
		`SQL_DESC_COUNT`
		`SQL_DESC_DISPLAY_SIZE`
		`SQL_DESC_FIXED_PREC_SCALE`
		`SQL_DESC_LENGTH`
		`SQL_DESC_NULLABLE`
		`SQL_DESC_NUM_PREC_RADIX`
		`SQL_DESC_OCTET_LENGTH`
		`SQL_DESC_PRECISION`
		`SQL_DESC_SCALE`
		`SQL_DESC_SEARCHABLE`
		`SQL_DESC_TYPE`
		`SQL_DESC_UNNAMED`
		`SQL_DESC_UNSIGNED`
		`SQL_DESC_UPDATABLE`

`SQLGetConnectAttr`
`SQLGetConnectAttrW`
	`*ValuePtr` must be `SQLUINTEGER` or `SQLULEN`, depending on the attribute you are getting.
	Note: TimesTen-specific attributes (prefixed with `TT_`) remain the same data types.
		On UNIX platforms: For the following attributes, a 64-bit value is returned in *`*ValuePtr`*:
		`SQL_ATTR_ASYNC_ENABLE`
		`SQL_ATTR_ENLIST_IN_DTC`
		`SQL_ATTR_ODBC_CURSORS`
		`SQL_ATTR_QUIET_MODE`

Table 10-16 (Cont.) Changes in ODBC 3.5 Functions

Function	Signature Changes	Size Changes in Option and Attribute Values
SQLGetStmtAttr SQLGetStmtAttrW	*ValuePtr must be SQLUINTEGER or SQLULEN, depending on the attribute you are getting. **Note**: TimesTen-specific attributes (prefixed with TT_) remain the same data types.	On UNIX platforms: For the following attributes, a 64-bit value is returned in **ValuePtr*: SQL_ATTR_APP_PARAM_DESC SQL_ATTR_APP_ROW_DESC SQL_ATTR_ASYNC_ENABLE SQL_ATTR_CONCURRENCY SQL_ATTR_CURSOR_SCROLLABLE SQL_ATTR_CURSOR_SENSITIVITY SQL_ATTR_CURSOR_TYPE SQL_ATTR_ENABLE_AUTO_IPD SQL_ATTR_FETCH_BOOKMARK_PTR SQL_ATTR_ROWS_FETCHED_PTR SQL_ATTR_IMP_PARAM_DESC SQL_ATTR_IMP_ROW_DESC SQL_ATTR_KEYSET_SIZE SQL_ATTR_MAX_LENGTH SQL_ATTR_MAX_ROWS SQL_ATTR_METADATA_ID SQL_ATTR_NOSCAN SQL_ATTR_PARAM_BIND_OFFSET_PTR SQL_ATTR_PARAM_BIND_TYPE SQL_ATTR_PARAM_OPERATION_PTR SQL_ATTR_PARAM_STATUS_PTR SQL_ATTR_PARAMS_PROCESSED_PTR SQL_ATTR_PARAMSET_SIZE SQL_ATTR_QUERY_TIMEOUT SQL_ATTR_RETRIEVE_DATA SQL_ATTR_ROW_ARRAY_SIZE SQL_ATTR_ROW_BIND_OFFSET_PTR SQL_ATTR_ROW_NUMBER SQL_ATTR_ROW_OPERATION_PTR SQL_ATTR_ROW_STATUS_PTR SQL_ATTR_SIMULATE_CURSOR SQL_ATTR_USE_BOOKMARKS

Table 10-16 (Cont.) Changes in ODBC 3.5 Functions

Function	Signature Changes	Size Changes in Option and Attribute Values
SQLSetConnectAttr SQLSetConnectAttrW	**ValuePtr* must be SQLUINTEGER or SQLULEN, depending on the attribute you are setting. **Note**: TimesTen-specific attributes (prefixed with TT_) remain the same data types.	On UNIX platforms: For the following attributes, a 64-bit value is passed in **ValuePtr*: SQL_ATTR_ASYNC_ENABLE SQL_ATTR_ENLIST_IN_DTC SQL_ATTR_ODBC_CURSORS SQL_ATTR_QUIET_MODE

Table 10-16 (Cont.) Changes in ODBC 3.5 Functions

Function	Signature Changes	Size Changes in Option and Attribute Values
SQLSetStmtAttr SQLSetStmtAttrW	**ValuePtr* must be SQLUINTEGER or SQLULEN, depending on the attribute you are setting. **Note**: TimesTen-specific attributes (prefixed with TT_) remain the same data types.	On UNIX platforms: For the following attributes, a 64-bit value is passed in **ValuePtr*: SQL_ATTR_APP_PARAM_DESC SQL_ATTR_APP_ROW_DESC SQL_ATTR_ASYNC_ENABLE SQL_ATTR_CONCURRENCY SQL_ATTR_CURSOR_SCROLLABLE SQL_ATTR_CURSOR_SENSITIVITY SQL_ATTR_CURSOR_TYPE SQL_ATTR_ENABLE_AUTO_IPD SQL_ATTR_FETCH_BOOKMARK_PTR SQL_ATTR_IMP_PARAM_DESC SQL_ATTR_IMP_ROW_DESC SQL_ATTR_KEYSET_SIZE SQL_ATTR_MAX_LENGTH SQL_ATTR_MAX_ROWS SQL_ATTR_METADATA_ID SQL_ATTR_NOSCAN SQL_ATTR_PARAM_BIND_OFFSET_PTR SQL_ATTR_PARAM_BIND_TYPE SQL_ATTR_PARAM_OPERATION_PTR SQL_ATTR_PARAM_STATUS_PTR SQL_ATTR_PARAMS_PROCESSED_PTR SQL_ATTR_PARAMSET_SIZE SQL_ATTR_QUERY_TIMEOUT SQL_ATTR_RETRIEVE_DATA SQL_ATTR_ROW_ARRAY_SIZE SQL_ATTR_ROW_BIND_OFFSET_PTR SQL_ATTR_ROW_NUMBER SQL_ATTR_ROW_OPERATION_PTR SQL_ATTR_ROW_STATUS_PTR SQL_ATTR_ROWS_FETCHED_PTR SQL_ATTR_SIMULATE_CURSOR SQL_ATTR_USE_BOOKMARKS

ODBC 2.5 Function Signatures That Have Changed

There are ODBC 2.5 functions with changes requiring code updates.

The functions listed in Table 10-17 have changes to the signature or changes to the size of option or attribute values, requiring code updates for ODBC 2.5 applications, as indicated. Sizes of option or attribute values apply to values of connection options, statement options, column attributes, or driver and data source information, either passed or returned.

Table 10-17   Changes in ODBC 2.5 Functions

Function	Signature Changes	Size Changes in Option and Attribute Values
SQLColAttributes SQLColAttributesW	N/A	On Linux and UNIX platforms: For the following *fDescType* values, a SQLLEN value is returned in **pfDesc*: SQL_COLUMN_COUNT SQL_COLUMN_DISPLAY_SIZE SQL_COLUMN_LENGTH SQL_DESC_AUTO_UNIQUE_VALUE SQL_DESC_CASE_SENSITIVE SQL_DESC_CONCISE_TYPE SQL_DESC_FIXED_PREC_SCALE SQL_DESC_SEARCHABLE SQL_DESC_UNSIGNED SQL_DESC_UPDATABLE
SQLGetConnectOption SQLGetConnectOptionW	The *Value* parameter must be SQLUINTEGER or SQLULEN, depending on the option you are getting. **Note**: TimesTen-specific options (prefixed with TT_) remain the same data types.	On Linux and UNIX platforms: For the option SQL_ATTR_QUIET_MODE, an HWND value (void * pointer to a window) is returned in *Value*.
SQLGetInfo SQLGetInfoW	N/A	On Linux and UNIX platforms: For the following *InfoType* values, a SQLPOINTER value is returned in **InfoValuePtr*: SQL_DRIVER_HDBC SQL_DRIVER_HENV SQL_DRIVER_HSTMT
SQLGetStmtOption	The *Value* parameter must be SQLUINTEGER or SQLULEN, depending on the option you are getting. **Note**: TimesTen-specific options (prefixed with TT_) remain the same data types.	On Linux and UNIX platforms: For the following options, a SQLPOINTER value is returned in *Value*: SQL_KEYSET_SIZE SQL_MAX_LENGTH SQL_MAX_ROWS SQL_ROWSET_SIZE

Table 10-17 (Cont.) Changes in ODBC 2.5 Functions

Function	Signature Changes	Size Changes in Option and Attribute Values
SQLParamOptions	On Linux and UNIX platforms: The *crow* and *pirow* parameters are now declared as SQLULEN.	N/A
SQLSetConnectOption SQLSetConnectOptionW	The *Value* parameter must be SQLUINTEGER or SQLULEN, depending on the option you are setting. **Note**: TimesTen-specific options (prefixed with TT_) remain the same data types.	On Linux and UNIX platforms: For the option SQL_ATTR_QUIET_MODE, an HWND value (void * pointer to a window) is passed in *Value*.
SQLSetPos	TimesTen does not support scrollable cursors. This function returns a "Driver not capable" error (S1C00). **Note**: The ODBC definition of SQLSETPOSIROW, the data type for the irow parameter, has changed. (See the next section, ODBC Data Types That Have Changed.)	N/A
SQLSetStmtOption	The *Value* parameter must be SQLUINTEGER or SQLULEN, depending on the option you are setting. **Note**: TimesTen-specific options (prefixed with TT_) remain the same data types.	On Linux and UNIX platforms: For the following options, a SQLPOINTER value is passed in *Value*: SQL_KEYSET_SIZE SQL_MAX_LENGTH SQL_MAX_ROWS SQL_ROWSET_SIZE

ODBC Data Types That Have Changed

There are changes to data types that require code updates for ODBC applications.

Table 10-18 ODBC 2.5 Data Types That Have Changed

Data Types	Explanation
HANDLE HINSTANCE	On Linux and UNIX platforms: These data types have been redefined as (void *).
SQLROWCOUNT SQLROWSETSIZE SQLTRANSID	These data types have been deprecated. Use SQLULEN instead.
SQLROWOFFSET	This data type has been deprecated. Use SQLLEN instead.
SQLSETPOSIROW	On Linux and UNIX platforms: This data type has been redefined as SQLULEN. It is advisable to use SQLULEN directly instead.