Oracle® TimesTen In-Memory Database

C Developer's Guide

Release 11.2.1

E13066-02

August 2009

Oracle TimesTen In-Memory Database C Developer's Guide, Release 11.2.1

E13066-02

Contents

Preface

Oracle TimesTen In-Memory Database is a memory-optimized relational database. Deployed in the application tier, TimesTen operates on databases that fit entirely in physical memory using standard SQL interfaces. High availability for the in-memory database is provided through real-time transactional replication.

TimesTen supports a variety of programming interfaces, including ODBC (Open Database Connectivity), OCI (Oracle Call Interface), Oracle Pro*C/C++ (precompiler for embedded SQL and PL/SQL instructions in C or C++ code), and PL/SQL (Oracle procedural language extension for SQL).

This preface covers the following topics:

- Audience
- Related documents
- Conventions
- Documentation Accessibility
- Technical support

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.

Related documents

TimesTen documentation is available on the product distribution media and on the Oracle Technology Network:

`http://www.oracle.com/technology/documentation/timesten_doc.html`

Oracle documentation is also available on the Oracle Technology network. This may be especially useful for Oracle features that TimesTen supports but does not attempt to fully document, such as OCI and Pro*C/C++:

`http://www.oracle.com/technology/documentation/database.html`

In particular, these Oracle documents 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 manual frequently refers to ODBC API reference documentation for further information. This is available from Microsoft or a variety of third parties. For example:

`http://msdn.microsoft.com/en-us/library/ms714562(VS.85).aspx`

Conventions

TimesTen supports multiple platforms. Unless otherwise indicated, the information in this guide applies to all supported platforms. The term Windows refers to Windows 2000, Windows XP, and Windows Server 2003. The term UNIX refers to Solaris, Linux, HP-UX, and AIX.

This document uses the following text conventions:

Convention	Meaning
italic	Italic type indicates terms defined in text, book titles, or emphasis.
`monospace`	Monospace type indicates commands, URLs, function names, attribute names, directory names, file names, text that appears on the screen, or text that you enter.
`italic monospace`	Italic monospace type indicates a placeholder or a variable in a code example for which you specify or use a particular value. For example: `Driver=`*`install_dir`*`/lib/libtten.sl` Replace *`install_dir`* with the path of your TimesTen installation directory.
[ ]	Square brackets indicate that an item in a command line is optional.
{ }	Curly braces indicated that you must choose one of the items separated by a vertical bar (|) in a command line.
|	A vertical bar (or pipe) separates alternative arguments.
. . .	An ellipsis (. . .) after an argument indicates that you may use more than one argument on a single command line. An ellipsis in a code example indicates that what is shown is only a partial example.
%	The percent sign indicates the UNIX shell prompt.

In addition, TimesTen documentation uses the following special conventions:

Convention	Meaning
`install_dir`	The path that represents the directory where TimesTen is installed.
`TTinstance`	The instance name for your specific installation of TimesTen. Each installation of TimesTen must be identified at installation time with a unique alphanumeric instance name. This name appears in the installation path.
`bits` or *`bb`*	Two digits, either 32 or 64, that represent either a 32-bit or 64-bit operating system.
`release` or *`rr`*	Numbers that represent a major TimesTen release, with or without dots. For example, 1121 or 11.2.1 represents TimesTen Release 11.2.1.

Convention	Meaning
DSN	TimesTen data source name.

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible to all users, including users that are disabled. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Accessibility standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For more information, visit the Oracle Accessibility Program Web site at `http://www.oracle.com/accessibility/`.

Accessibility of Code Examples in Documentation

Screen readers may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, some screen readers may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation

This documentation may contain links to Web sites of other companies or organizations that Oracle does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

Deaf/Hard of Hearing Access to Oracle Support Services

To reach Oracle Support Services, use a telecommunications relay service (TRS) to call Oracle Support at 1.800.223.1711. An Oracle Support Services engineer will handle technical issues and provide customer support according to the Oracle service request process. Information about TRS is available at `http://www.fcc.gov/cgb/consumerfacts/trs.html`, and a list of phone numbers is available at `http://www.fcc.gov/cgb/dro/trsphonebk.html`.

Technical support

For information about obtaining technical support for TimesTen products, go to the following Web address:

`http://www.oracle.com/support/contact.html`

What's New

This section summarizes new features and functionality of Oracle TimesTen In-Memory Database Release 11.2.1 that are documented in this guide, providing links into the guide for more information.

New Features for Release 11.2.1

TimesTen Release 11.2.1 includes the following new features covered in this guide:

- Quick Start demos

  This release includes an optional Quick Start feature with introductory information, tutorials, and new or reworked demo applications. Note that the demos have mostly the same names as in earlier releases, but in a different location.

  See "About the TimesTen C demos" on page 2-4 and `install_dir/quickstart.html` in your installation.

- Oracle Call Interface (OCI) support

  OCI is an API that provides functions you can use to access the database server 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.

  See Chapter 3, "TimesTen Support for Oracle Call Interface."

- Pro*C/C++ support

  The Oracle Pro*C/C++ Precompiler allows you to embed SQL statements or PL/SQL blocks directly into C or C++ code. You use a precompilation step to convert the Pro*C/C++ source file into a C or C++ source file.

  See Chapter 4, "TimesTen Support for Oracle Pro*C/C++ Precompiler."

- Access control

  Perhaps the most significant overall change to previous functionality in this release is access control. TimesTen has new features to control database access with object-level resolution for database objects such as tables, views, materialized views, and sequences. This also affects access to certain TimesTen built-in procedures, utilities, and connection attributes.

  See "Considering TimesTen features for access control" on page 1-27. For general information, see "Managing Access Control" in *Oracle TimesTen In-Memory Database Operations Guide*.

- OUT parameters

  Discussion of binding parameters includes new support for binding OUT and IN OUT parameters.

  See appropriate subsections under "Binding parameters" on page 1-10.

- Duplicate parameters

  TimesTen now supports either of two modes for binding duplicate parameters in a SQL statement. Use the `DuplicateBindMode` general connection attribute to choose between Oracle mode (now the default) and traditional TimesTen mode.

  See "Binding duplicate parameters in SQL statements" on page 1-15.

- REF CURSORs

  REF CURSOR is a PL/SQL concept, where a REF CURSOR is a handle to a cursor over a SQL result set and can be passed between PL/SQL and an application.

  See "Working with REF CURSORs" on page 1-17.

- Automatic client failover

  Automatic client failover, used in High Availability scenarios when failure of a TimesTen node results in failover (transfer) to an alternate node, automatically reconnects applications to the new node. TimesTen provides features that allow applications to be alerted when this happens, so they can take any appropriate action.

  See "Automatic client failover" on page 1-29.

- Deferred prepare

  To make its behavior consistent with OCI expectations 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" on page 1-9.

- Parallel log manager

  As a result of new multistrand functionality of the log manager, some terminology has changed in Chapter 5, "XLA and TimesTen Event Management," and Chapter 9, "XLA Reference." For discussion in those chapters, the term "log sequence number" (LSN) is replaced by "log record identifier". There are still LSNs, but in a more limited and specific context. Only some of what used to be called LSNs are still LSNs in the new usage. Names of functions, data structures, and so on where "LSN" appears are not changed due to backward compatibility considerations.

  In particular, note that the multistrand functionality affects the `tt_XlaLsn_t` structure used by XLA functions `ttXlaGetLSN` and `ttXlaSetLSN`. It also affects the `tt_LSN_t` structure that is a field of the `ttXlaUpdateDesc_t` structure. See "ttXlaGetLSN" on page 9-27, "ttXlaSetLSN" on page 9-52, and "ttXlaUpdateDesc_t" on page 9-70.

- Rowids

  Each row in a TimesTen database table has a unique identifier known as its rowid. TimesTen now supports Oracle-style rowids. An application can retrieve the rowid of a row from the ROWID pseudocolumn. Rowids can be represented in either binary or character format.

See "Working with rowids" on page 1-19.

- DML returning (RETURNING INTO clause)

  TimesTen now supports the 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.

  See "Using DML returning (RETURNING INTO clause)" on page 1-22.

- Execution time threshold for SQL statements

  You can configure TimesTen to write a warning to the support log and throw an SNMP trap when the execution of a SQL statement exceeds a specified time duration, in seconds. This feature was added in a 7.0.x maintenance release but not documented in this manual. Note that this feature is similar to but differs from the previously existing timeout value for SQL statements.

  See "Setting a timeout or threshold for executing SQL statements" on page 1-23.

- "T-tree" indexes are now referred to as "range" indexes.
- C utility function changes

  The `ttRepDuplicateEx` function in particular is affected by access control. See "ttRepDuplicateEx" on page 8-15.

- XLA replicated bookmarks

  If you are using an active standby pair replication scheme, you now have the option of using replicated bookmarks. For a replicated bookmark, operations on the bookmark are replicated to the standby database as appropriate. This allows more efficient recovery of your bookmark positions in the event of failover.

  See the section on replicated bookmarks under "About XLA bookmarks" on page 5-4.

- Additional XLA changes

  – Use of XLA in non-persistent mode is discouraged. Use the persistent mode.

    See "XLA persistent mode" on page 5-2.

  – There is a new XLA type conversion function for rowids, `ttXlaRowidToCString`.

    See "ttXlaRowidToCString" on page 9-51.

  – XLA indicates whether an update was generated as part of a cascading delete or aging operation, through new values for the *`flags`* field in the `ttXlaUpdateDesc_t` structure.

    See "ttXlaUpdateDesc_t" on page 9-70.

1 Working with TimesTen Data Stores

This chapter describes how to use ODBC to connect to and use an Oracle TimesTen In-Memory Database data store. It includes the following topics:

- Managing TimesTen data store connections
- Managing TimesTen data
- Considering TimesTen features for access control
- Managing error and failure conditions

Managing TimesTen data store connections

The *Oracle TimesTen In-Memory Database Operations Guide* contains information about creating a DSN for a TimesTen data store. The type of DSN you create depends on whether your application will connect directly to the data store or will connect by a client.

If you intend to connect directly to the data store, refer to "Creating TimesTen Data Stores" in *Oracle TimesTen In-Memory Database Operations Guide*. There are sections on creating a DSN for a direct connection from UNIX or Windows.

If you intend to create a client connection to the data store, 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 UNIX or Windows.

Notes:

- 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 a child process, the child must not use the connection.

The rest of this section covers the following topics:

- SQLConnect, SQLDriverConnect, SQLAllocConnect, SQLDisconnect functions
- Connecting to and disconnecting from a data store
- Setting connection attributes programmatically
- Access control for connections

SQLConnect, SQLDriverConnect, SQLAllocConnect, SQLDisconnect functions

The following ODBC functions are available for connecting to a TimesTen data store and related functionality:

- `SQLConnect`: Loads a driver and connects to the data store. The connection handle points to where information about the connection is stored, including status, transaction state, results, and error information.
- `SQLDriverConnect`: This is an alternative to `SQLConnect` when more information is required than what is supported by `SQLConnect`, which is just data source (the TimesTen data store), user name, and password.
- `SQLAllocConnect`: Allocates memory for a connection handle within the specified environment.
- `SQLDisconnect`: Disconnect from the data store. Takes the existing connection handle as its only argument.

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

Connecting to and disconnecting from a data store

This section provides examples of connecting to and disconnecting from a TimesTen data store.

Example 1–1 Connect and disconnect (excerpt)

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

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

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

Example 1–2 Connect and disconnect (complete program)

This example contains a complete program that creates, connects to, and disconnects from a data store. 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" on page 1-28.)

```
#ifdef WIN32
# include <windows.h>
#else
# include <sqlunix.h>
#endif
#include <sql.h>
#include <sqlext.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_1121;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"
              " data store\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

You can set or override connection attributes programmatically by specifying a connection string when you connect to a data store.

Refer to *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 *Oracle TimesTen In-Memory Database Reference* for specific information about any particular connection attribute.

Example 1–3 Connect and use store-level locking

This code fragment connects to a data store called `mydsn` and indicates in the `SQLDriverConnect` call that the application should use data store-level locking rather than the default row-level locking. Note that `LockLevel` is a general connection attribute.

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

rc = SQLDriverConnect(hdbc, NULL,
    "DSN=mydsn;LockLevel=1", SQL_NTS,
    ConnStrOut, sizeof (ConnStrOut),
    &cbConnStrOut, SQL_DRIVER_NOPROMPT);
```

Note: Each connection to a TimesTen data store opens several files. This means that an application with many threads, each with a separate connection, has several files open for each thread. Such an application can exceed the 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*.

Access control for connections

Privilege to connect to a TimesTen data store must be explicitly granted to every user other than the instance administrator, through the CREATE SESSION privilege. This is a system privilege. It must be granted by an administrator to the user, either directly or through the PUBLIC role. Refer to "Managing Access Control" in *Oracle TimesTen In-Memory Database Operations Guide* for additional information and examples.

Managing TimesTen data

This section provides detailed information on working with data in a TimesTen data store. It includes the following topics:

- TimesTen #include files
- SQL statement execution within C applications
- Preparing and executing queries and working with cursors
- TimesTen deferred prepare
- Binding parameters
- Working with REF CURSORs
- Working with rowids
- Prefetching multiple rows of data
- Making and committing changes to the database
- Using DML returning (RETURNING INTO clause)
- Calling TimesTen built-in procedures within C applications
- Setting a timeout or threshold for executing SQL statements
- Managing cache groups
- Setting globalization options
- ODBC 3.0 data types

TimesTen #include files

In addition to standard C `#include` files, your application must include the following TimesTen `#include` files:

Include file	Description
`timesten.h`	TimesTen ODBC `#include` file
`tt_errCode.h`	TimesTen native error codes

SQL statement execution within C applications

"Working with Data in a TimesTen Data Store" in *Oracle TimesTen In-Memory Database Operations Guide* describes how to use SQL to manage data in a TimesTen data store. This section describes general formats used to call a SQL statement within a C application. The following topics are covered:

- SQLExecDirect and SQLExecute functions
- Executing a SQL statement

Note: Access control privileges are checked both when SQL is prepared and when it is executed in the database. Refer to "Considering TimesTen features for access control" on page 1-27 for related information.

SQLExecDirect and SQLExecute functions

There are two ODBC functions to execute SQL statements:

- `SQLExecute`: Executes a statement that has already been prepared. This is used in conjunction 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 a relatively large number of times.

- `SQLExecDirect`: Prepares and executes a statement.

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

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

Executing a SQL statement

You can use the `SQLExecDirect` function as shown in Example 1–4.

The next section, "Preparing and executing queries and working with cursors", shows usage of the `SQLExecute` and `SQLPrepare` functions.

Example 1–4   Executing a SQL statement with SQLExecDirect

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

```
#include <sql.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 queries and working with cursors

This section shows the basic steps of preparing and executing a query and working with cursors. Applications use cursors to scroll through the results of a query, examining one result row at a time.

In the ODBC setting, 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 number of rows to fetch at one time, using `SQLSetStmtOption` options documented in "Option support for SQLSetStmtOption and SQLGetStmtOption" on page 10-5. 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" on page 1-12. (Note that Example 1–5 below does not bind parameters.)
3. Call `SQLExecute` to initiate the SELECT statement. See "SQLExecDirect and SQLExecute functions" on page 1-7.
4. Call `SQLBindCol` to assign the storage and data type for a column of results, binding column results to local variable storage in your application.
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.

Note: Access control privileges are checked both when SQL is prepared and when it is executed in the database. Refer to "Considering TimesTen features for access control" on page 1-27 for related information.

Example 1–5 Executing a query and working with the cursor

This example illustrates how to prepare and execute a query using ODBC calls. 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 <sql.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);
   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);
}
```

TimesTen deferred prepare

In standard ODBC, a `SQLPrepare` call is expected to be compiled by the SQL engine so that information about the SQL statement, such as column descriptions for the result set, is available to the application and accessible through calls such as `SQLDescribeCol`. To achieve this functionality, the `SQLPrepare` call must be sent to 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, as well as making 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 relevant, 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.

Binding parameters

This sections discusses how to bind input or output parameters for SQL statements. The following topics are covered:

- ODBC to SQL or PL/SQL type mappings
- SQLBindParameter function
- Considerations for SQL parameter type assignments
- Binding IN parameters
- Binding OUT parameters

- Binding IN OUT parameters
- Binding duplicate parameters in SQL statements
- Binding duplicate parameters in PL/SQL
- Considerations for floating point data

ODBC to SQL or PL/SQL type mappings

Table 1–1 documents the mapping between ODBC SQL types and Oracle SQL or PL/SQL types.

Table 1–1 ODBC SQL to TimesTen SQL or PL/SQL type mappings

ODBC type	SQL or PL/SQL type
SQL_BIGINT	NUMBER
SQL_BINARY	RAW(*p*)
SQL_BIT	PLS_INTEGER
SQL_CHAR	CHAR(*p*)
SQL_DATE	DATE
SQL_DECIMAL	NUMBER
SQL_DOUBLE	NUMBER
SQL_FLOAT	BINARY_DOUBLE
SQL_INTEGER	PLS_INTEGER
SQL_NUMERIC	NUMBER
SQL_REAL	BINARY_FLOAT
SQL_REFCURSOR	REF CURSOR
SQL_ROWID	ROWID
SQL_SMALLINT	PLS_INTEGER
SQL_TIMESTAMP	TIMESTAMP(*s*)
SQL_TINYINT	PLS_INTEGER
SQL_VARBINARY	RAW(*p*)
SQL_VARCHAR	VARCHAR2(*p*)
SQL_WCHAR	NCHAR(*p*)
SQL_WVARCHAR	NVARCHAR2(*p*)

Notes:

- The notation (*p*) indicates precision is according to the `SQLBindParameter` argument *`cbColDef`*.
- The notation (*s*) indicates scale is according to the `SQLBindParameter` argument *`ibScale`*.
- For SQL types, the table applies only to passthrough parameters for Oracle In-Memory Database Cache (IMDB Cache). Otherwise, refer to "Considerations for SQL parameter type assignments" on page 1-13.
- For ODBC types that map to the NUMBER type, TimesTen ignores any precision and scale specified in the `SQLBindParameter` call. Thus TimesTen avoids the overhead of checking constraints on the data.

SQLBindParameter function

The ODBC `SQLBindParameter` function is used to bind parameters for SQL statements. This could include IN, OUT, or IN OUT 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 1–2 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) of the ODBC `SQLBindParameter` function to specify data types. In addition, use the *`rgbValue`*, *`cbValueMax`*, and *`pcbValue`* arguments of `SQLBindParameter`.

Table 1–2 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 column or expression of the parameter marker.
`ibScale`	SQLSMALLINT	The scale of the column or expression of the parameter marker.
`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.

Considerations for SQL parameter type assignments

There is a notable difference between TimesTen and Oracle Database functionality regarding SQL parameter type assignments. For statements that execute within TimesTen, the TimesTen query optimizer determines data types of SQL parameters. Oracle, by contrast, requires the application to specify data types through the *`fSqlType`*, *`cbColDef`*, and *`ibScale`* arguments (as applicable) of the `SQLBindParameter` function. TimesTen ignores these arguments. An application cannot change or influence type assignments.

Note that the TimesTen behavior does not apply to passthrough statements, when TimesTen is used as IMDB Cache. When statements are passed through to Oracle for execution, the data types must be specified using *`fSqlType`*, *`cbColDef`*, and *`ibScale`*, as applicable.

The TimesTen behavior also does not apply to PL/SQL statements.

Considerations for C parameters used in PL/SQL

When a C parameter is used by a PL/SQL block or procedure, a type conversion may occur, depending on the C type of the parameter and the PL/SQL type or types that are acceptable in that context. If the combination is not supported, an error will occur. These conversions can be from a C type to a SQL or PL/SQL type (IN parameter), from a SQL or PL/SQL type to a C type (OUT parameter), or both (IN OUT parameter).

Binding IN parameters

For IN parameters for use with 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 "Considerations for SQL parameter type assignments" above.

In addition, the *`rgbValue`*, *`cbValueMax`*, and *`pcbValue`* arguments of `SQLBindParameter` are used as follows for IN 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. For IN parameters, this would be valid for only character or binary data.
  - SQL_NTS for a null-terminated string.
  - SQL_NULL_DATA for a null value.

Binding OUT parameters

The *`rgbValue`*, *`cbValueMax`*, and *`pcbValue`* arguments of the ODBC `SQLBindParameter` function are used as follows for OUT 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, in the case of an OUT parameter that has character data, *cbValueMax* is greater by 1 than the longest value that can be accepted. And in the case of an OUT parameter that has C NCHAR data, which uses a two-byte null terminator, *cbValueMax* is greater by 2 than the longest value that can be accepted.
- *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). 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.

Example 1–6 Binding output parameters

This example shows how to prepare, bind, and execute a PL/SQL anonymous block. The anonymous block assigns bind variable `a` the value `'abcde'` and bind variable `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 IN OUT parameters

The *rgbValue*, *cbValueMax*, and *pcbValue* arguments of the ODBC `SQLBindParameter` function are used as follows for IN OUT parameters:

- *`rgbValue`*: This is first used before statement execution as described in "Binding IN parameters" on page 1-13. Then it is used during statement execution as described in the preceding section, "Binding OUT parameters". Note that for an IN OUT parameter, the outgoing value from a statement execution will be the incoming value to the statement execution that immediately follows, unless that is overridden by the application. Also, for IN OUT 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 will be placed.
- *`cbValueMax`*: For character and binary data, this is first used as described in "Binding IN parameters" on page 1-13. Then it is used as described in the preceding section, "Binding OUT 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, in the case of an IN OUT parameter that has character data, *`cbValueMax`* is greater by 1 than the longest value that can be accepted. And in the case of an IN OUT parameter that has C NCHAR data (which uses a two-byte null terminator), *`cbValueMax`* is greater by 2 than the longest value that can be accepted.
- *`pcbValue`*: This is first used before statement execution as described in "Binding IN parameters" on page 1-13. Then it is used after statement execution as described in the preceding section, "Binding OUT parameters".

Important:

- 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.
- TimesTen will return SQL_SUCCESS_WITH_INFO if there are errors in converting OUT or IN OUT parameters. If `SQLExecute`, `SQLExecDirect`, or `SQLParamData` returns SQL_SUCCESS_WITH_INFO, then the values of all OUT and IN OUT parameters are undefined.

Binding duplicate parameters in SQL statements

TimesTen supports either of two modes for binding duplicate parameters in a SQL statement. (Regarding PL/SQL statements, see "Binding duplicate parameters in PL/SQL" on page 1-17.)

- Oracle mode, where multiple instances of the same parameter name are considered to be distinct parameters.
- Traditional TimesTen mode, as in earlier releases, where multiple instances of the same parameter name are considered to be the same parameter.

You can choose the desired mode through the `DuplicateBindMode` general connection attribute. `DuplicateBindMode=0` (the default) is for Oracle mode, and `DuplicateBindMode=1` is for TimesTen mode. Because this is a general connection attribute, different concurrent connections to the same database can use different values. Refer to "DuplicateBindMode" in *Oracle TimesTen In-Memory Database Reference* for additional information about this attribute.

The rest of this section provides details for each mode, considering the following query:

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

Notes:

- This discussion applies only to SQL statements issued directly from ODBC (not through PL/SQL, for example).
- The use of "?" for parameters, not supported in Oracle Database, is supported by TimesTen in either mode.

Oracle mode for duplicate parameters In Oracle mode, where `DuplicateBindMode=0`, multiple instances of the same parameter name in a SQL statement are considered to be different parameters. 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 still 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.

In Oracle mode, the `SQLNumParams` ODBC function returns 3 for the number of parameters in the example.

TimesTen mode for duplicate parameters In TimesTen mode, where `DuplicateBindMode=1`, SQL statements containing duplicate parameters are parsed such that only distinct parameter names are considered as separate parameters.

Binding is based on the position of the first occurrence of a parameter name. Subsequent occurrences of the parameter name are not given their own position numbers. All occurrences of the same parameter name take on the same value.

For the SQL statement above, the two occurrences of `a` are considered to be a single parameter, so cannot be bound separately:

```
SQLBindParameter(..., 1, ...); /* both occurrences of :a */
SQLBindParameter(..., 2, ...); /* occurrence of :b */
```

Note that in TimesTen mode, parameter `b` is considered to be in position 2, not position 3.

In TimesTen mode, the `SQLNumParams` ODBC function returns 2 for the number of parameters in the example.

Binding duplicate parameters in PL/SQL

The preceding discussion does not apply within PL/SQL. Instead, PL/SQL semantics apply, whereby you bind a value for each unique parameter. An application calling the following block, for example, would bind only one parameter, corresponding to `:a`.

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

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

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

An application calling the following block would bind two parameters, with `:a` as parameter #1 and `:b` as parameter #2. 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

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.

Working with REF CURSORs

REF CURSOR is a PL/SQL concept, where a REF CURSOR is a handle to a cursor over a SQL result set and 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, allowing 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 the same basic steps of prepare, bind, execute, and fetch as in the cursor example in "Preparing and executing queries and working with cursors" on page 1-8.

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" on page 1-12 and "Binding OUT parameters" on page 1-13 for information about these and other `SQLBindParameter` arguments.
3. Call `SQLExecute` to execute the statement.
4. Call `SQLBindCol` to bind result columns to local variable storage.
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.

Important: For passing REF CURSORs between PL/SQL and an application, TimesTen supports only OUT REF CURSORs, from PL/SQL to the application, and supports a statement returning only a single REF CURSOR.

Example 1–7 Executing a query and working with a REF CURSOR

This example uses a REF CURSOR and 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 omitted for simplicity. In addition to 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    empid;
  SQLCHAR       lastname[30];

  /* 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);

  /* set the value for :deptid */
  deptid = 30;

  /* 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's statement handle and drop both handles */
  SQLFreeStmt(refcursor_hstmt, SQL_DROP);
  SQLFreeStmt(plsql_hstmt, SQL_DROP);
}
```

Working with rowids

Each row in a TimesTen 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 1–1 on page 1-11, 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 specification" in *Oracle TimesTen In-Memory Database SQL Reference* for additional information about rowids and the ROWID data type, including usage and life.

Note: Oracle TimesTen In-Memory Database does not support the PL/SQL type UROWID.

Prefetching multiple rows of data

A TimesTen extension to ODBC allows applications to prefetch multiple rows of data from a TimesTen data store into the ODBC driver buffer. This can increase the performance of applications that use the read-committed or serializable isolation level.

The TT_PREFETCH_COUNT connection option determines how many rows a `SQLFetch` call will prefetch. This option is available for both direct access and client/server applications.

TT_PREFETCH_COUNT can be set in a call to either `SQLSetConnectOption` or `SQLSetStmtOption`. The value can be any integer from 0 to 128, inclusive. For example:

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

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

To disable prefetch, set TT_PREFETCH_COUNT to 1.

When the prefetch count is set to 0, TimesTen uses a default value, depending on the isolation level you have set for the data store. In read-committed isolation mode, the default prefetch value is 5. In serializable isolation mode, the default is 128. The default prefetch value is the optimum 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 set the isolation level as follows:

```
rc = SQLSetConnectOption(hdbc, SQL_TXN_ISOLATION, SQL_TXN_READ_COMMITTED);
```

Or:

```
rc = SQLSetConnectOption(hdbc, SQL_TXN_ISOLATION, SQL_TXN_SERIALIZABLE);
```

Making and committing changes to the database

By default in TimesTen, autocommit is enabled, so that any DML change you make (update, insert, or delete) is committed automatically. It is recommended, however, that you disable this feature and commit (or roll back) your changes manually. You can refer to "Transaction semantics" in *Oracle TimesTen In-Memory Database Operations Guide* for information about autocommit.

With autocommit disabled, you can manually commit or roll back a transaction using the `SQLTransact` ODBC function. Refer to ODBC API reference documentation for details about this function.

Notes:

- 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 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. For other operations, the driver can define the usage of this argument. See "Managing cache groups" on page 1-25 regarding special TimesTen functionality. Refer to ODBC API reference documentation for general information about `SQLRowCount` and its arguments.

Example 1–8 Updating the database and committing the change

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. See  */
  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);

}
```

Using 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 OUT parameters. Bind the OUT parameters as discussed in "Binding parameters" on page 1-10.

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.

Example 1–9   DML returning

This example is adapted from Example 1–8 in the previous section.

```
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_CHAR, 0, 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.

Calling TimesTen built-in procedures within C applications

"Built-In Procedures" in *Oracle TimesTen In-Memory Database Reference* describes TimesTen built-in procedures that extend standard ODBC functionality. You can invoke these procedures using ODBC call syntax. Use the following format:

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

This ODBC example calls the `ttCkpt` procedure to initiate a fuzzy checkpoint.

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

Note: The *Oracle TimesTen In-Memory Database Reference* documents any required privileges for TimesTen built-in procedures. For example, `ttCkpt` requires ADMIN privilege.

Setting a timeout or threshold for executing SQL statements

TimesTen offers two ways to limit the time for SQL statements or procedure calls to execute, applying to any `SQLExecute`, `SQLExecDirect`, or `SQLFetch` call.

- Setting a timeout value for SQL statements
- Setting a threshold value for SQL statements

For the former, if the timeout duration is reached, the statement stops executing and an error is thrown. For the latter, if the threshold is reached, an SNMP trap is thrown but execution continues.

Setting a timeout value 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. Despite the name, this timeout value applies to any executable SQL statement, not just queries.

In TimesTen you can specify this timeout value for any connection, and hence for any statement, by using the `SqlQueryTimeout` DSN attribute. If you set `SqlQueryTimeout` in the DSN specification, its value becomes the default value for all subsequent connections to the data store. A call to `SQLSetConnectOption` with the SQL_QUERY_TIMEOUT option overrides any default value that a connection may have inherited and applies to any statement from that connection. A call to `SQLSetStmtOption` with the SQL_QUERY_TIMEOUT option overrides any default value inherited from the connection as well as any value set using `SQLSetConnectOption`, but applies only to the 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 execute a large number of UPDATE, DELETE or INSERT statements, the commit or rollback phases may take a long time to complete. During that time the timeout value is ignored.

Note: If both a lock timeout and a `SqlQueryTimeout` 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" (connection attribute) in *Oracle TimesTen In-Memory Database Reference*, or to "Check for deadlocks and timeouts" in *Oracle TimesTen In-Memory Database Troubleshooting Procedures Guide*.

Setting a threshold value for SQL statements

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

The name of the SNMP trap is `ttQueryThresholdWarnTrap`. See *Oracle TimesTen In-Memory Database Error Messages and SNMP Traps* for information about configuring SNMP traps. Despite the name, this threshold applies to any executable SQL statement.

By default, the application obtains the threshold from the `QueryThreshold` connection attribute setting. Setting the TT_QUERY_THRESHOLD option in a `SQLSetConnectOption` call overrides the connection attribute setting for the current connection.

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 as well as any setting through `SQLSetConnectOption`. 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);
```

Managing cache groups

In IMDB 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 In-Memory Database Cache User's Guide*.

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

Setting globalization options

TimesTen extensions to ODBC enable an application to set options for linguistic sorts, length semantics for character columns, and error reporting during character set conversion. These options can be used in a call to `SQLSetConnectOption`. The options are defined in the `timesten.h` `#include` file (noted in "TimesTen #include files" on page 1-7).

For more information about linguistic sorts, length semantics, and character sets, see "Globalization Support" in *Oracle TimesTen In-Memory Database Operations Guide*.

This section includes the following TimesTen ODBC globalization options:

- TT_NLS_SORT
- TT_NLS_LENGTH_SEMANTICS
- TT_NLS_NCHAR_CONV_EXCP

TT_NLS_SORT

This option specifies the collating sequence used for linguistic comparisons. See "Monolingual linguistic sorts" and "Multilingual linguistic sorts" in *Oracle TimesTen In-Memory Database Operations Guide* for supported linguistic sorts.

It takes a string value. The default is "BINARY".

Also see the description of the `NLS_SORT` connection attribute, which has the same functionality, in "NLS_SORT" in *Oracle TimesTen In-Memory Database Reference*. Note that `TT_NLS_SORT`, being a runtime option, takes precedence over the `NLS_SORT` connection attribute.

TT_NLS_LENGTH_SEMANTICS

This option specifies whether byte or character semantics is used. The possible values are:

- TT_NLS_LENGTH_SEMANTICS_BYTE (default)

- TT_NLS_LENGTH_SEMANTICS_CHAR

Also see the description of the `NLS_LENGTH_SEMANTICS` connection attribute, which has the same functionality, in "NLS_LENGTH_SEMANTICS" in *Oracle TimesTen In-Memory Database Reference*. Note that `TT_NLS_LENGTH_SEMANTICS`, being a runtime option, takes precedence over the `NLS_LENGTH_SEMANTICS` connection attribute.

TT_NLS_NCHAR_CONV_EXCP

This option specifies whether an error is reported when there is data loss during an implicit or explicit character type conversion between NCHAR or NVARCHAR2 data and CHAR or VARCHAR2 data during SQL operations. The option does not apply to conversions done by ODBC as a result of binding.

The possible values are:

- TRUE: Errors during conversion are reported.
- FALSE: Errors during conversion are not reported (default).

Also see the description of the `NLS_NCHAR_CONV_EXCP` connection attribute, which has the same functionality, in "NLS_NCHAR_CONV_EXCP" in *Oracle TimesTen In-Memory Database Reference*. Note that `TT_NLS_NCHAR_CONV_EXCP`, being a runtime option, takes precedence over the `NLS_NCHAR_CONV_EXCP` connection attribute.

ODBC 3.0 data types

The data types used in ODBC 2.0 and prior have been renamed in ODBC 3.0 for ISO 92 standards compliance. The sample programs shipped with TimesTen have been written using SQL 3.0 data types. The following table maps 2.0 types to their 3.0 equivalents.

ODBC 2.0 data type	ODBC 3.0 data type
HDBC	SQLHDBC
HENV	SQLHENV
HSTMT	SQLHSTMT
HWND	SQLHWND
LDOUBLE	SQLDOUBLE
RETCODE	SQLRETURN
SCHAR	SQLSCHAR
SDOUBLE	SQLFLOATS
SDWORD	SQLINTEGER
SFLOAT	SQLREAL
SWORD	SQLSMALLINT
UCHAR	SQLCHAR
UDWORD	SQLUINTEGER
UWORD	SQLUSMALLINT

Either version of data types may be used with TimesTen without restriction.

Note also that the `FAR` modifier that is mentioned in ODBC 2.0 documentation is not required.

Considering TimesTen features for access control

TimesTen has features to control database access with object-level resolution for database objects such as tables, views, materialized views, and sequences. You can refer to "Managing Access Control" in *Oracle TimesTen In-Memory Database Operations Guide* for introductory information about these features.

This section introduces access control as it relates to SQL operations, data store connections, XLA, and C utility functions.

For any query, SQL DML statement, or SQL DDL statement discussed in this document or used in an example, it is assumed that the user has appropriate privileges to execute the statement. For example, a SELECT statement on a table requires ownership of the table, SELECT privilege granted for the table, or the SELECT ANY TABLE system privilege. Similarly, any DML statement requires table ownership, the applicable DML privilege (such as UPDATE) granted for the table, or the applicable ANY TABLE privilege (such as UPDATE ANY TABLE).

For DDL statements, CREATE TABLE requires the CREATE TABLE privilege in the user's schema, or CREATE ANY TABLE in any other schema. ALTER TABLE requires ownership or the ALTER ANY TABLE system privilege. DROP TABLE requires ownership or the DROP ANY TABLE system privilege. There are no object-level ALTER or DROP privileges.

Refer to "SQL Statements" in *Oracle TimesTen In-Memory Database SQL Reference* for the privilege required for any given SQL statement.

Privileges are granted through the SQL statement GRANT and revoked through the statement REVOKE. Some privileges are granted to all users through the PUBLIC role, of which each user is a member. See "The PUBLIC role" in *Oracle TimesTen In-Memory Database SQL Reference* for information about that role.

In addition, access control affects the following topics covered in this document:

- Connecting to a data store. Refer to "Access control for connections" on page 1-6.
- Setting connection attributes. Refer to "Setting connection attributes programmatically" on page 1-5.
- Configuring and managing XLA and using XLA functions. Refer to "Access control impact on XLA" on page 5-8. Also refer to Chapter 9, "XLA Reference." The documentation for each XLA function notes the required privilege.
- Executing C utility functions. Refer to Chapter 8, "TimesTen Utility API." The documentation for each utility mentions whether any privilege is required.

Notes:

- Access control cannot be disabled.
- Access control privileges are checked both when SQL is prepared and when it is executed in the database, with most of the performance cost coming at prepare time.

Managing error and failure conditions

This section discusses the following topics:

- Error handling
- Automatic client failover

Error handling

This section includes the following topics:

- Checking for errors
- Error and warning levels
- Recovering after fatal errors

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 demo programs provided with TimesTen show examples of error checking.

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

```
#define tt_ErrMemoryLock             712
```

For a description of each message, 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.

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

For more information about writing a function to handle standard ODBC errors, see "Retrieving errors and warnings" in *Oracle TimesTen In-Memory Database Error Messages and SNMP Traps*.

Example 1–10 Checking an ODBC function call for errors

This 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);
  TerminateGracefully(1);
}
```

Error and warning levels

TimesTen can return fatal errors, non-fatal errors, or warnings.

Fatal errors Fatal errors are those that make the data store inaccessible until after error recovery. When a fatal error occurs, all data store 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 include a disconnect from the data store.

Non-fatal errors Non-fatal errors include simple 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 normal error-handling process and requires the application to check for and identify them.

When a data store is affected by a non-fatal error, an error may be returned and the application should take appropriate action. In some cases, such as process failure, no error is returned, but TimesTen automatically rolls back the transactions of the failed process.

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. Some examples of events that cause TimesTen to issue a warning include:

- Checkpoint failure
- Use of a deprecated TimesTen feature
- Truncation of some data
- Execution of a recovery process upon connect

We strongly encourage application developers to include code that checks for warnings, as they can indicate application problems.

Recovering after fatal errors

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

- Every connection to the data store is invalidated and applications are required to disconnect from the invalidated data store.
- The data store is recovered from the checkpoint and transaction log files upon the first subsequent initial connection.
- The recovered data store 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.

If no checkpoint or transaction log files exist and the `AutoCreate` DSN attribute is set, TimesTen creates an empty data store.

Automatic client failover

Automatic client failover, used in High Availability scenarios when failure of a TimesTen node results in failover (transfer) to an alternate node, automatically reconnects applications to the new node. The standby node becomes the active node due to failure of the previously active node. TimesTen provides features that allow applications to be alerted when this happens, so they can take any appropriate action.

This section discusses the TimesTen implementation of automatic client failover, covering the following topics:

- Features and functionality of automatic client failover
- Failover callback functions

Automatic client failover is complementary to Oracle Clusterware, though the two features are not dependent on each other. You can also refer to "Using Oracle Clusterware to Manage Active Standby Pairs" in *Oracle TimesTen In-Memory Database TimesTen to TimesTen Replication Guide*.

Features and functionality of automatic client failover

When a client failover occurs, no state other than the connection handle is preserved. All client statement handles are marked as invalid. API calls on these statement handles will generally return SQL_ERROR with a distinctive failover error code, defined in `tt_errCode.h`, such as:

```
SQLSTATE = S1000 "General Error", native error = tt_ErrFailoverInvalidation
```

The exception to this is for `SQLError` and `SQLFreeStmt` calls, which would behave normally.

In addition, note the following:

- The socket to the old server is closed. There is no attempt to call `SQLDisconnect`.
- In connecting to the alternate TimesTen node, the same connection string that was returned from the original connection request is used, other than resetting attributes as appropriate to indicate the new server DSN.
- 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 alternate server, the next failover request results in an attempt to reconnect to the original server. If that fails, alternating attempts are made to connect to the two servers until a timeout value specified by the DSN attribute TTC_TIMEOUT is reached.
- Failover connections are created only as needed, not in advance.

When failover occurs, TimesTen makes a callback to a user-defined function that you register. This function takes care of any custom actions you want to occur in a failover situation.

Notes:

- The features described here apply only in client/server mode, not for direct connections.
- TimesTen supports automatic client failover only in the active standby pair replication configuration, where the clients are to be connected to the node currently in the active role. When a failover connection is attempted, the server will reject it if it is not active.
- Functionality is similar to that of Oracle TAF (Transparent Application Failover) and FAN (Fast Application Notification), but TimesTen does not use the FAN or TAF libraries.

The following public connection options will be propagated to the new connection. The corresponding connection attribute 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_WARN_POSSIBLE_TRUNC_BINDING
TT_WARN_SQLCBIGINT_BINDING
TT_CONNECTION_CHARACTER_SET (ConnectionCharacterSet)
TT_REGISTER_FAILOVER_CALLBACK
```

The following options will be 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)
```

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

```
SQL_QUERY_TIMEOUT
TT_PREFETCH_COUNT
```

The following attributes for the logical server DSN in `sys.ttconnect.ini` are equivalent to `TTC_Server`, `TTC_Server_DSN`, and `TCP_Port`, but for the alternate server.

```
TTC_Server2
TTC_Server_DSN2
TCP_Port2
```

Notes:

- Like other DSN attributes, `TTC_Server2`, `TTC_Server_DSN2`, and `TCP_Port2` can be specified in the connection string, overriding any settings in the DSN.
- If `TTC_Server2` is specified but `TTC_Server_DSN2` and `TCP_Port2` are not, then `TTC_Server_DSN2` is set to the `TTC_Server_DSN` value and `TCP_Port2` is set to the `TCP_Port` value.
- `TTC_Server` and `TTC_Server2` can have the same setting if it is a virtual IP address.

Setting any of `TTC_Server2`, `TTC_Server_DSN2`, or `TCP_Port2` implies the following:

- You intend to use automatic client failover.

- You understand that a new thread will be created for your application to support the failover mechanism.
- You have linked your application with a thread library.

The following new DSN attribute specifies a port range for the port where the failover thread will listen for failover notifications:

```
TTC_FAILOVERPORTRANGE
```

Set this as a lower and upper value separated by hyphen. TimesTen supports setting a port range to accommodate firewalls between the client and server. By default, a port chosen by the operating system will be used.

Notes:

- If the client library cannot connect to `TTC_Server_DSN`, it will try the failover alternative, as if it had received an explicit failover request.
- If the client library loses the connection to the server, it will fail over and attempt to switch to the alternate node.
- If the active node fails before the client registration is successfully propagated by replication to the standby, the client will not receive a failover message and the registration will be lost. However, the client library will eventually notice (through TCP) that its connection to the former active server has been lost, and it can then initiate a failover attempt.

Failover callback functions

When failover occurs, TimesTen makes a callback to your user-defined function for any desired action. This function is called when the attempt to connect to the alternate server 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 (modeled on a corresponding TAF function):

```
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, as would be the case with TAF.
- *`foEvent`* indicates the event that has occurred, with supported values as for FAN and TAF:
  - TT_FO_BEGIN: Beginning failover.
  - TT_FO_ABORT: Failover failed. Retries were attempted for the interval specified by TTC_TIMEOUT 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`, which 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 the driver manager.)
- *`callbackFcn`* specifies the callback function. (You can set this to NULL to cancel callbacks for the given connection. The failover will still happen, but the application will 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 will be 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.

Notes:

- 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.

Example 1–11 Failover callback function and registration

This example shows the following:

- 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
```

2 Compiling and Linking TimesTen Applications

This chapter describes how to compile and link TimesTen applications for UNIX and Windows platforms and introduces Quick Start demo applications. The following topics are covered:

- Setting environment variables
- Linking options
- Compiling and linking applications on Windows
- Compiling and linking applications on UNIX
- Testing link options
- About the TimesTen C demos

Setting environment variables

Environment variable settings for TimesTen are discussed in "Environment variables" in the *Oracle TimesTen In-Memory Database Installation Guide*.

UNIX

On UNIX platforms, you can set environment variables for TimesTen by executing one of the following scripts:

```
install_dir/bin/ttenv.sh
install_dir/bin/ttenv.csh
```

WINDOWS

On Windows, you can either set environment variables during installation or run:

```
install_dir\bin\ttenv.bat
```

Linking options

A TimesTen application can link directly with the TimesTen Data Manager ODBC driver or TimesTen Client ODBC driver, or can link with a driver manager.

Linking directly with a TimesTen ODBC driver

Applications that only must use TimesTen can link directly with either the TimesTen Data Manager ODBC driver or the TimesTen Client ODBC driver. Direct linking avoids the performance overhead of a driver manager and is the simplest way to access TimesTen. However, developers of direct-linked applications should be aware of the following issues associated with direct linking.

- The application can only connect to a DSN that uses the driver with which it is linked. It cannot connect to a data store of any other vendor, nor can it connect to a TimesTen DSN of a different TimesTen driver or a different version or type.
- Windows ODBC tracing is not available to direct-linked applications.
- The ODBC cursor library is not available to direct-linked applications.
- Applications cannot use the ODBC functions that are usually implemented by a driver manager. These functions include `SQLDataSources` and `SQLDrivers`.
- Applications that use `SQLCancel` to close a cursor instead of `SQLFreeStmt(..., SQL_CLOSE)` will 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.

Linking with a driver manager

Applications that link with the 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. However, using a driver manager has the following limitations:

- The driver manager adds synchronization overhead to every ODBC function call. This overhead may be significant for some applications.
- 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 option, see "Prefetching multiple rows of data" on page 1-20.
- Applications cannot set or reset the TimesTen-specific TT_PREFETCH_CLOSE connection option. For more information about using the TT_PREFETCH_CLOSE connection option, see "Enable TT_PREFETCH_CLOSE for serializable transactions" in the *Oracle TimesTen In-Memory Database Operations Guide*.
- Driver managers are not available by default on most non-Windows platforms. TimesTen supplies a driver manager for either Windows or UNIX with the Quick Start sample applications.
- Transaction Log API (XLA) calls cannot be used when applications are linked with a driver manager.

Compiling and linking applications on Windows

WINDOWS

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, you must indicate the location of TimesTen `#include` files by using the `/I` compiler option.

The Makefile in Example 2–1 shows how to build a TimesTen application on Windows systems. This example assumes that *`install_dir`*`\lib` has already been added to the LIB environment variable.

Example 2–1 Building a TimesTen application in Windows

```
CFLAGS = "/Iinstall_dir\include"
LIBSDM = ODBC32.LIB
LIBS = tten1121.lib ttdv1121.lib
LIBSDEBUG = tten1121d.lib ttdv1121d.lib
```

```
LIBSCS = ttclient1121.lib

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

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

# Link directly with the TimesTen
# Data Manager 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)
```

Compiling and linking applications on UNIX

On UNIX platforms:

- Compile TimesTen applications using the TimesTen header files from the TimesTen installation directory.
- Link with the TimesTen Data Manager ODBC driver or TimesTen Client ODBC driver, which is provided as a shared library.

On UNIX, applications using the ULONG, SLONG, USHORT or SSHORT ODBC data types must specify the `TT_USE_ALL_TYPES` preprocessor option while compiling. This is typically done using the `-DTT_USE_ALL_TYPES` C compiler option.

To use the TimesTen `#include` files, add the following to the C compiler command, where *`install_dir`* is the TimesTen installation directory path:

```
-Iinstall_dir/include
```

To link with the TimesTen Data Manager ODBC driver, add the following to the link command:

```
-Linstall_dir/lib -ltten
```

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

To link with the TimesTen Client ODBC driver, add the following to the link command:

```
-Linstall_dir/lib -lttclient
```

On Solaris, the default TimesTen Client ODBC driver was compiled with Studio 11. The library allows you to link an application compiled with the Sun Studio 11 C/C++ compiler directly with the TimesTen client.

You can use Makefiles in subdirectories under the `quickstart/sample_code` directory, or you can use Example 2–2 to guide you in creating your own Makefile.

Example 2–2 Makefile to link the application

```
CFLAGS = -Iinstall_dir/include
LIBS = -Linstall_dir/lib -ltten
LIBSDEBUG = -Linstall_dir/lib -lttenD
LIBSCS = -Linstall_dir/lib -lttclient

# Link directly with the TimesTen
# Data Manager 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)
```

Notes:

- To directly link your application to the TimesTen Data Manager debug ODBC driver, substitute `-lttenD` for `-ltten` on the link line.
- On Solaris, when compiling with Sun C/C++ compilers, TimesTen applications must be compiled and linked with the `-mt` option.

Testing link options

To test whether an application was directly linked, you can call `SQLGetInfo` to examine the driver release of the data store connection handle, as shown in Example 2–3.

For direct-linked applications, the call to `SQLGetInfo` returns the unchanged connection handle. For applications that use a driver manager, the returned connection handle differs from the passed-in handle.

Example 2–3 Testing whether an application is directly linked

```
RetCode = SQLDriverConnect(hdbc,NULL,szConnString,
     SQL_NTS,szConnout,255,&cbConnOut,SQL_DRIVER_NOPROMPT);
rc = SQLGetInfo(hdbc, SQL_DRIVER_HDBC, &drhdbc,
     sizeof (drhdbc), &drhdbclen);
if (drhdbc != NULL && drhdbc != hdbc) {
     /* Linked with driver manager */
}
else {
     /* Directly linked with TimesTen driver */
}
```

About the TimesTen C demos

After you have configured your C environment, you can confirm that everything is set up correctly by compiling and running TimesTen Quick Start demo applications. Refer to the Quick Start welcome page at `install_dir/quickstart.html`, especially the links under Sample Programs, for information on the following topics.

- Demo schema and setup

  The `build_sampledb` script creates a sample database and demo schema. You must run this before you start using the demos.

- Demo environment and setup

  The `ttquickstartenv` script, a superset of the `ttenv` script generally used for TimesTen setup, sets up the demo environment. You must run this each time you enter a session where you want to compile and run any of the demos.

- Demos and setup

  TimesTen provides demos for ODBC, XLA, OCI, and Pro*C/C++ in subdirectories under the `quickstart/sample_code` directory. For instructions on compiling and running the demos, see the README files in the subdirectories.

- What the demos do

  A synopsis of each demo is provided.

3 TimesTen Support for Oracle Call Interface

Oracle TimesTen In-Memory Database and Oracle In-Memory Database Cache (IMDB Cache) support the Oracle Call Interface (OCI).

This chapter includes the following sections:

This chapter focuses on TimesTen-specific information regarding OCI support. For complete information, you can refer to *Oracle Call Interface Programmer's Guide* in the Oracle Database library.

Overview of OCI

OCI is an API that provides functions you can use to access the database server 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
- External procedure functions, for writing C callbacks to PL/SQL

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.

Overview of TimesTen OCI support

This chapter contains information specific to using OCI with TimesTen and IMDB Cache. TimesTen OCI syntax and usage is the same as that in Oracle Database.

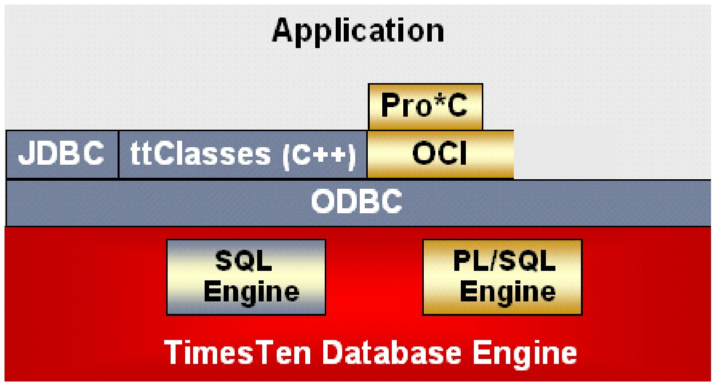
This section covers the following topics:

- OCI libraries and architecture
- Globalization support
- TimesTen restrictions
- The ttSrcScan utility

OCI libraries and architecture

TimesTen OCI depends on the Oracle client library and the TimesTen ODBC libraries. TimesTen OCI support enables you to run many existing OCI applications with TimesTen in direct mode or client/server mode. It also enables you to use other Oracle products, such as Pro*C/C++, that use OCI as a database interface. You can also call PL/SQL from OCI applications. Figure 3–1 shows where OCI support is positioned in the TimesTen architecture.

Figure 3–1 OCI in the TimesTen architecture



TimesTen includes Oracle Instant Client as the OCI client library.

TimesTen Release 11.2.1 OCI is based on Oracle Release 11.1.0.7 OCI and supports the contemporary OCI 8 style APIs. For example, the `OCIStmtExecute()` function is supported but not the older `oexec()` function. See "Obsolete OCI Routines" in *Oracle Call Interface Programmer's Guide* in the Oracle Database documentation.

Globalization support

This section discusses TimesTen OCI support for globalization.

Character sets

To specify a character set for the connection, OCI programs can set the NLS_LANG environment variable or call `OCIEnvNlsCreate()`. Any connection character set in the `odbc.ini` file is ignored. Setting the character set explicitly is recommended. The default is typically AMERICAN_AMERICA.US7ASCII.

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 include 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
```

Notes:

- An NLS_LANG environment setting overrides the TimesTen default character set.
- The TIMESTEN8 character set is not supported.
- On Windows, the NLS_LANG setting is taken from 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\ is valid and indicates a character set supported by TimesTen. (The NLS_LANG registry setting may be set to an invalid value, such as "NA". If the value is "NA", the TimesTen installer will replace it with AMERICAN_AMERICA.US7ASCII.) This is likely only an issue on systems that previously had Oracle9*i* or earlier Oracle versions installed.
- Refer to "Choosing a Locale with the NLS_LANG Environment Variable" in *Oracle Database Globalization Support Guide* for information about NLS_LANG.
- Refer to "OCIEnvNlsCreate()" in *Oracle Call Interface Programmer's Guide* for information about that OCI call.

Additional globalization features

TimesTen OCI also supports the following additional globalization features. These can be set either as environment variables or TimesTen general connection attributes. An environment variable setting takes precedence.

- 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_LANGUAGE. 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.

Note: These environment variables override the corresponding TimesTen connection attributes for OCI or Pro*C/C++ programs.

Refer to *Oracle TimesTen In-Memory Database Operations Guide* and *Oracle Database Globalization Support Guide* for additional information on these environment variables and related features.

TimesTen restrictions

TimesTen does not support OCI calls that are related to functionality that does not exist in TimesTen or IMDB Cache. For example, TimesTen and IMDB Cache do not support these Oracle Database features:

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

Both TimesTen and Oracle support XA, but TimesTen does not support XA through OCI.

In addition, TimesTen OCI has the following restrictions:

- The `TypeMode` database attribute must be set to 0, which corresponds to Oracle behavior.
- The `DuplicateBindMode` connection attribute must be set to 0, which corresponds to Oracle behavior.

- The `DDLCommitBehavior` connection attribute must be set to 0, which corresponds to Oracle behavior.
- 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.
- TimesTen Client/Server automatic client failover is not supported.
- The TNS Ping utility does not recognize connections to TimesTen.
- Retrieving implicit ROWID values from INSERT, UPDATE, and DELETE statements is not supported.
- TimesTen built-in procedures that return result sets are not supported directly.
- 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.
- Oracle utilities such as SQL*Plus and SQL*Loader are not supported.

The ttSrcScan utility

If you have an existing OCI program and want to see whether it uses OCI features that TimesTen does not support, you can use the `ttSrcScan` command line utility to scan your program for unsupported functions, packages, types, type codes, attributes, modes, and constants. This is a standalone utility that can be run without TimesTen or Oracle being installed and runs on any platform supported by TimesTen. It reads source code files as input and creates HTML and text files as output. If the utility finds unsupported items, then they are logged and alternatives are suggested. You can find the `ttSrcScan` executable in the `quickstart/sample_util` directory in your TimesTen installation.

Specify an input file or directory for the program to be scanned and an output directory for the `ttSrcScan` reports. Other options are available as well. See the README file in the `sample_util` directory for information.

Getting started with TimesTen OCI

This section discusses the following topics for getting started with a TimesTen OCI application:

- Environment variables for TimesTen OCI
- Compiling and linking OCI applications
- Connecting to a TimesTen data store from OCI
- Error reporting
- OCI demo programs

Environment variables for TimesTen OCI

Environment variables for executing a TimesTen OCI application are described in Table 3–1. Settings apply to both direct mode and client/server mode except as noted.

After installation, you can modify environment variables as appropriate through the TimesTen `install_dir/bin/ttenv` script or `quickstart/ttquickstartenv` script applicable to your operating system.

You can also use the TimesTen OCI and Pro*C/C++ Make files provided with the Quick Start demos to implement appropriate environment settings. These are in the following locations:

```
quickstart/sample_code/oci/
quickstart/sample_code/proc/
```

Refer to "Environment variables" in *Oracle TimesTen In-Memory Database Installation Guide* for additional information about environment variables and `ttenv`.

Table 3–1 Environment variables for TimesTen OCI

Variable	Required or optional	Settings

LD_LIBRARY_PATH (UNIX)
PATH (Windows)
	Required	Must be set so that the Oracle Instant Client directory precedes the Oracle Database libraries in the path. The path will be set properly if you use either of the following scripts under *`install_dir`*:
		`bin/ttenv`
		`quickstart/ttquickstartenv`
		(Unless you installed Quick Start in a different location.)

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.
TWO_TASK (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 data store from OCI" on page 3-7 for more information.
NLS_LANG	Optional	See "Character sets" on page 3-3. 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.

Table 3–1 (Cont.) Environment variables for TimesTen OCI

Variable	Required or optional	Settings
NLS_SORT	Optional	See "Additional globalization features" on page 3-3. The sort order must be a value supported by TimesTen. This overrides the TimesTen `NLS_SORT` connection attribute.
NLS_LENGTH_SEMANTICS	Optional	See "Additional globalization features" on page 3-3. This overrides the TimesTen `NLS_LENGTH_SEMANTICS` connection attribute.
NLS_NCHAR_CONV_EXCP	Optional	See "Additional globalization features" on page 3-3. This overrides the TimesTen `NLS_NCHAR_CONV_EXCP` connection attribute.

Note: Refer to "Data Store Attributes" in *Oracle TimesTen In-Memory Database Reference* for information about TimesTen connection attributes.

Compiling and linking OCI applications

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

OCI programs that use the Oracle Client 11.1.0.7 library do not have to be recompiled or relinked to be executed with TimesTen.

Connecting to a TimesTen data store 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:

- 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.

Notes:

- Although the `sqlnet` mechanism is used for a TimesTen OCI connection, the connection goes through the TimesTen ODBC driver, not the Oracle `sqlnet` driver.
- Privilege to connect to a TimesTen data store must be explicitly granted to every user other than the instance administrator, through the CREATE SESSION privilege. Refer to "Access control for connections" on page 1-6.
- If the operating system user name is the same as the instance administrator user name, you can connect through OCI as the instance administrator by specifying the user name in quotes and brackets and using an empty string for the password, such as "`[myname]`" for user and "" for password.

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 `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 `odbc.ini` or `sys.odbc.ini` file that is visible to a user running your OCI application. On Windows, the DSN can be specified by using the ODBC Data Source Administrator. See "Creating TimesTen Data Stores" 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 data store.

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));
```

Using an easy connect string to connect

TimesTen supports easy connect syntax, which enhances the Instant Client package by allowing 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. In client/server mode, 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 desired, 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 the DSN `ttclient` in the `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 `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`*:

```
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 will be tried and the order in which they will be tried. If `sqlnet.ora` exists and does not include a particular naming method, you cannot use that method. If `sqlnet.ora` does not exist, you can use any method. The Instant Client will look for a `sqlnet.ora` file at the TNS_ADMIN location, as applicable. The default location is *`ORACLE_HOME`*`/network/admin`.

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 will first look for `tnsnames` syntax in your OCI logon calls. If it cannot find `tnsnames` syntax, it will look for easy connect syntax.

Note: In TimesTen, the sample copies of `tnsnames.ora` and `sqlnet.ora` are in the *`install_dir`*`/network/admin/samples` directory.

Error reporting

Errors under TimesTen OCI applications return Oracle error codes. TimesTen attempts to report the same Oracle error code as Oracle would under similar conditions. The error messages may come from either the TimesTen catalog or the Oracle catalog. Some error messages may include the accompanying TimesTen error code if appropriate.

Fatal errors are those that make the data store inaccessible until after error recovery. When a fatal error occurs, all data store connections are required to disconnect. No further operations may complete. Fatal errors in OCI are indicated by the Oracle 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 include a disconnect from the data store.

OCI demo programs

TimesTen ships OCI demo programs. They are in the `quickstart/sample_code/oci` directory. The README file in the directory explains how to compile and run the demos.

Refer to the Quick Start welcome page at *`install_dir`*`/quickstart.html` for information.

Additional features of TimesTen OCI

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

- TimesTen deferred prepare
- Using IMDB Cache in OCI
- Duplicate parameter bindings in TimesTen OCI

TimesTen deferred prepare

In OCI, a "prepare" call is expected to be a lightweight operation performed on the client. To allow TimesTen to be consistent with this expectation, as well as 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" on page 3-11.

Using IMDB Cache in OCI

This section discusses TimesTen OCI features related using the IMDB Cache:

- Specifying the Oracle password in OCI for IMDB Cache
- Determining the number of cache groups affected by an action

Specifying the Oracle password in OCI for IMDB Cache

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

For use of OCI with the IMDB Cache, TimesTen allows you to pass the Oracle 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 *cacheuser = (text *)"cacheuser1";
text *cachepwds = (text *)"ttpwd;OraclePWD=orclpwd";
text *ttdbname = (text *)"tt_tnsname";
....
OCILogon2(envhp, errhp, &svchp,
       (text *)cacheuser, (ub4)strlen(cacheuser),
       (text *)cachepwds, (ub4)strlen(cachepwds),
       (text *)ttdbname, (ub4)strlen(ttdbname), OCI_DEFAULT));
```

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

Note the following for the example:

- `cacheuser1` is the name of the TimesTen cache user as well as the name of the Oracle user who can access the cached Oracle tables.
- `ttpwd` is the password of the TimesTen cache user.
- `orclpwd` is the password of the Oracle user.
- `tt_tnsname` is the TNS name of the TimesTen database being connected to.

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

For related information, see "Determining the number of cache instances affected by an operation" in the *Oracle In-Memory Database Cache User's Guide*.

Duplicate parameter bindings in TimesTen OCI

"Binding duplicate parameters in SQL statements" on page 1-15 discusses the two supported modes for binding duplicate parameters in a SQL statement, either the Oracle mode or the traditional TimesTen mode. In TimesTen OCI, only the Oracle mode is supported.

As in chapter 1, consider the following query:

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

In OCI, as in Oracle mode in general, two occurrences of parameter `a` are considered to be separate parameters. However, OCI allows both occurrences of `a` to be bound with a single call to `OCIBindByPos()`:

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

Alternatively, OCI also allows the two occurrences of `a` to be bound 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()`.

Call, handle, descriptor, SQL data type, and parameter attribute support

Table 3–2 lists TimesTen support for OCI calls that are documented for Oracle Database, release 11.1.0.7.

Groups of calls that are related to features that TimesTen does not support may be represented with an asterisk in the name. For example, TimesTen does not support Advanced Queueing. The calls related to Advanced Queueing have names that start with OCIAQ and are represented in the table as `OCIAQ*()`.

Some groups of calls that TimesTen does support are also represented in the table with an asterisk. For example, TimesTen OCI supports the OCI date functions, designated by `OCIDate*()`.

Table 3–2 TimesTen OCI call support

OCI call	Supported	Notes
`OCIAQ*()`	No	TimesTen does not support Advanced Queueing.
`OCIAnyData*()`	No	TimesTen does not support Any Data.
`OCIAppCtxClearAll()`	Yes	
`OCIAppCtxSet()`	Yes	
`OCIArrayDescriptorAlloc()`	Yes	
`OCIArrayDescriptorFree()`	Yes	
`OCIAttrGet()`	Yes	TimesTen support includes special usage with cache groups. See "Using IMDB Cache in OCI" on page 3-11.
`OCIAttrSet()`	Yes	
`OCIBinXml*()`	No	TimesTen does not support XML DB.
`OCIBindArrayOfStruct()`	Yes	Supported for SQL statements but not PL/SQL.
`OCIBindByName()`	Yes	Unsupported values for the *mode* parameter: ▪ OCI_DATA_AT_EXEC ▪ OCI_IOV
`OCIBindByPos()`	Yes	Unsupported values for the *mode* parameter: ▪ OCI_DATA_AT_EXEC ▪ OCI_IOV
`OCIBindDynamic()`	No	
`OCIBindObject()`	No	TimesTen does not support user-defined objects.
`OCIBreak()`	No	
`OCICache*()`	No	TimesTen does not support user-defined objects.
`OCICharSetConversionIsReplacementUsed()`	Yes	
`OCICharSetToUnicode()`	Yes	
`OCIClientVersion()`	Yes	
`OCIColl*()`	No	TimesTen does not support collections.
`OCIConnectionPoolCreate()`	No	
`OCIConnectionPoolDestroy()`	No	
`OCIContext*()`	No	TimesTen does not support Data Cartridge.
`OCIDBShutdown()`	No	
`OCIDBStartup()`	No	
`OCIDate*()`	Yes	See Table 3–4 for information about descriptor support.

Table 3–2 (Cont.) TimesTen OCI call support

OCI call	Supported	Notes
`OCIDefineArrayOfStruct()`	Yes	Supported for SQL statements but not PL/SQL.
`OCIDefineByPos()`	Yes	
`OCIDefineDynamic()`	No	
`OCIDefineObject()`	No	
`OCIDescribeAny()`	Yes	Unsupported values for the *`objptr_typ`* parameter: ▪ OCI_OTYPE_REF ▪ OCI_OTYPE_PTR Unsupported values for the *`objtyp`* parameter: ▪ OCI_PTYPE_PKG ▪ OCI_PTYPE_SYN ▪ OCI_PTYPE_TYPE
`OCIDescriptorAlloc()`	Yes	
`OCIDescriptorFree()`	Yes	
`OCIDirPath*()`	No	TimesTen does not support Direct Path Loading.
`OCIDuration*()`	No	TimesTen does not support Data Cartridge.
`OCIEnvCreate()`	Yes	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()`	Yes	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.
`OCIEnvNlsCreate()`	Yes	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()`	Yes	
`OCIExtProc*()`	No	TimesTen does not support Data Cartridge.
`OCIExtract*()`	No	TimesTen does not support Data Cartridge.
`OCIFile*()`	No	TimesTen does not support Data Cartridge.

Table 3–2 (Cont.) TimesTen OCI call support

OCI call	Supported	Notes
`OCIFormatInit()`	No	TimesTen does not support Data Cartridge.
`OCIFormatString()`	No	TimesTen does not support Data Cartridge.
`OCIFormatTerm()`	No	TimesTen does not support Data Cartridge.
`OCIHandleAlloc()`	Yes	
`OCIHandleFree()`	Yes	
`OCIInitialize()`	Yes	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*()`	Yes	See Table 3–4 for information about descriptor support.
`OCIIter*()`	No	TimesTen does not support collections.
`OCILdaToSvcCtx()`	No	
`OCILob*()`	No	TimesTen does not support LOB data types.
`OCILogoff()`	Yes	
`OCILogon()`	Yes	
`OCILogon2()`	Yes	OCI_DEFAULT is the only supported value for the *`mode`* parameter.
`OCIMemory*()`	No	TimesTen does not support Data Cartridge.
`OCIMessage*()`	No	TimesTen does not support Data Cartridge.
`OCIMultiByte*()`	Yes	
`OCINls*()`	Yes	
`OCINumber*()`	Yes	
`OCIObject*()`	No	TimesTen does not support user-defined objects.
`OCIParamGet()`	Yes	
`OCIParamSet()`	Yes	
`OCIPasswordChange()`	No	
`OCIPing()`	Yes	
`OCIRaw*()`	Yes	
`OCIRef*()`	No	
`OCIReset()`	No	
`OCIRowidToChar()`	Yes	
`OCIServer*()`	Yes	OCI_DEFAULT is the only supported value for the *`mode`* parameter of `OCIServerAttach`.

Table 3–2 (Cont.) TimesTen OCI call support

OCI call	Supported	Notes
`OCISessionBegin()`	Yes	OCI_CRED_RDBMS is the only supported value for the *credt* parameter. OCI_DEFAULT is the only supported value for the *mode* parameter.
`OCISessionEnd()`	Yes	
`OCISessionGet()`	Yes	
`OCISessionPoolCreate()`	No	
`OCISessionPoolDestroy()`	No	
`OCISessionRelease()`	Yes	
`OCISharedLibInit()`	No	
`OCIStmtExecute()`	Yes	Unsupported values for the *mode* parameter: ■ OCI_BATCH_ERRORS ■ OCI_EXACT_FETCH ■ 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()`	Yes	
`OCIStmtFetch2()`	Yes	The only supported values for the *orientation* parameter are OCI_DEFAULT and OCI_FETCH_NEXT.
`OCIStmtGetBindInfo()`	Yes	
`OCIStmtGetPieceInfo()`	No	
`OCIStmtPrepare()`	Yes	The only supported value for the *language* parameter is OCI_NTV_SYNTAX. **Note**: In TimesTen, `OCIStmtPrepare()` does not support statement caching. See `OCIStmtPrepare2()` that follows.
`OCIStmtPrepare2()`	Yes	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()`	Yes	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()`.
`OCIStmtSetPieceInfo()`	No	
`OCIString*()`	Yes	
`OCISubscription*()`	No	TimesTen does not support Advanced Queueing.
`OCISvcCtxToLda()`	No	

Table 3–2 (Cont.) TimesTen OCI call support

OCI call	Supported	Notes
`OCITable*()`	No	
`OCITerminate()`	No	
`OCIThread*()`	Yes	
`OCITransCommit()`	Yes	The only supported value for the *mode* parameter is OCI_DEFAULT.
`OCITransDetach()`	No	
`OCITransForget()`	No	
`OCITransMultiPrepare()`	No	
`OCITransPrepare()`	No	
`OCITransRollback()`	Yes	
`OCITransStart()`	No	
`OCIType*()`	No	
`OCIUnicodeToCharSet()`	Yes	
`OCIUserCallbackGet()`	Yes	
`OCIUserCallbackRegister()`	Yes	
`OCIWideChar*()`	Yes	
`OCIXmlDbFreeXmlCtx()`	No	TimesTen does not support XML DB.
`OCIXmlDbInitXmlCtx()`	No	TimesTen does not support XML DB.

Table 3–3 lists the handles and attributes that TimesTen OCI supports.

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_EVTCTX 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 OCI_ATTR_PREFETCH_ROWS 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_SERVER_STATUS
User session	OCISession	OCI_ATTR_CLIENT_IDENTIFER 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.
Transaction	OCITrans	OCI_ATTR_TRANS_NAME OCI_ATTR_TRANS_TIMEOUT
Thread	OCIThreadHandle	

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

Table 3–5 lists the SQL data types that TimesTen OCI supports.

Table 3–5 TimesTen OCI supported SQL data types

SQL data type	Notes
SQLT_AFC	
SQLT_AVC	
SQLT_BDOUBLE	
SQLT_BFLOAT	
SQLT_BIN	
SQLT_CHR	
SQLT_DAT	
SQLT_DATE	
SQLT_FLT	
SQLT_IBDOUBLE	
SQLT_IBFLOAT	
SQLT_INT	
SQLT_INTERVAL_DS	Not stored in TimesTen.
SQLT_INTERVAL_YM	Not stored in TimesTen.
SQLT_LBI	
SQLT_LNG	
SQLT_LVB	Truncated at 4 MB when stored in TimesTen.
SQLT_LVC	Truncated at 4 MB when stored in TimesTen.
SQLT_NUM	
SQLT_ODT	
SQLT_RDD	Rowids returned in Oracle format.
SQLT_RSET	Only one result set parameter is allowed for each statement.
SQLT_STR	
SQLT_TIME	

Table 3–5 (Cont.) TimesTen OCI supported SQL data types

SQL data type	Notes
SQLT_TIME_TZ	Time zone is ignored when stored in TimesTen.
SQLT_TIMESTAMP	
SQLT_TIMESTAMP_LTZ	Time zone is ignored when stored in TimesTen.
SQLT_TIMESTAMP_TZ	Time zone is ignored when stored in TimesTen.
SQLT_UIN	
SQLT_VBI	
SQLT_VCS	
SQLT_VNU	
SQLT_VST	

Table 3–6 lists supported parameter attributes.

Table 3–6 TimesTen OCI support for 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 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 support for 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_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 Oracle Pro*C/C++ Precompiler

Oracle TimesTen In-Memory Database and Oracle IMDB Cache support the Oracle Pro*C/C++ Precompiler. You can use the precompiler with embedded SQL and PL/SQL applications that access the TimesTen database.

This chapter includes the following topics:

- Overview of the Oracle Pro*C/C++ Precompiler
- Overview of TimesTen support for Pro*C/C++
- Getting started with TimesTen Pro*C/C++
- TimesTen Pro*C/C++ Precompiler options

It provides only an overview and TimesTen-specific information regarding Pro*C/C++. For complete general information, you can refer to *Pro*C/C++ Programmer's Guide* in the Oracle Database library.

Overview of the Oracle Pro*C/C++ Precompiler

The Oracle Pro*C/C++ Precompiler allows 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 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 SQLLIB library, which is shipped with TimesTen as part of the Oracle Instant Client.

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 on page 3-2 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:

- TimesTen OCI support
- Embedded SQL restrictions
- Semantic checking restrictions
- Embedded PL/SQL restrictions
- Transaction restrictions
- Connection restrictions
- Summary of unsupported or restricted executable commands and clauses
- The ttSrcScan utility

TimesTen OCI support

Because TimesTen support of the Oracle Pro*C/C++ Precompiler depends on TimesTen OCI support, 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.

For more information about TimesTen OCI support, see Chapter 3, "TimesTen Support for Oracle Call Interface." Much of the information there may apply to Pro*C/C++ applications as well.

Embedded SQL restrictions

TimesTen supports SQL92 standards. Oracle supports SQL99 standards.

The TimesTen Pro*C/C++ Precompiler does not support embedded SQL for functionality that TimesTen and IMDB Cache do not support. See "TimesTen restrictions" on page 3-4.

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.
- `SQLRowidGet()` is supported following only SELECT FOR UPDATE statements.

Semantic checking restrictions

TimesTen support for the Oracle Pro*C/C++ Precompiler does not include 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 TimesTen 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 TimesTen database. In the following example, you will be prompted for the password:

  ```
  USERID=user1@my_tnsname
  ```

Alternatively, you can enter `USERID=user1/mypassword@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 data store from Pro*C/C++" on page 4-6 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 TimesTen database must be running.
- The USERID precompiler option must be set, specifying an existing TimesTen user. See the preceding section, "Semantic checking restrictions", for details about setting this option.

Transaction restrictions

Regarding transactions, TimesTen support for the Oracle Pro*C/C++ Precompiler does not include the following:

- 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 include the following:

- ALTER AUTHORIZATION clause
- Automatic connections to a TimesTen database
- Making connections to a TimesTen 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 data store from Pro*C/C++" on page 4-6.

Summary of unsupported or restricted executable commands and clauses

Given 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:

- ALTER AUTHORIZATION
- CACHE FREE ALL
- CALL: 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 TABLE: Supports only Oracle data types.
- DECLARE TYPE
- EXPLAIN PLAN
- IN SYSDBA MODE
- IN SYSOPER MODE
- Any "LOB..." command
- LOCK TABLE
- Any "OBJECT..." command
- PARTITION
- REGISTER CONNECT
- RETURN
- RETURNING
- SAVEPOINT
- SET DESCRIPTOR: Cannot set CHARACTER_SET_NAME.
- SET TRANSACTION
- START WITH
- TO SAVEPOINT

The ttSrcScan utility

If you have an existing Pro*C/C++ program and want to see whether it uses Pro*C/C++ features that TimesTen does not support, you can use the `ttSrcScan` command line utility to scan your program for unsupported embedded SQL functions and types. This is a standalone utility that can be run without TimesTen or Oracle being installed and runs on any platform supported by TimesTen. It reads source code files as input and creates HTML and text files as output. If the utility finds unsupported items, they are logged and alternatives are suggested. You can find the `ttSrcScan` executable in the `quickstart/sample_util` directory in your TimesTen installation.

Specify an input file or directory for the program to be scanned and an output directory for the `ttSrcScan` reports. Other options are available as well. See the README file in the `sample_util` directory for information.

Getting started with TimesTen Pro*C/C++

This section covers the following topics for getting started with a Pro*C/C++ application for TimesTen:

- Building a Pro*C/C++ application
- Connecting to a TimesTen data store from Pro*C/C++
- Error reporting and handling
- Pro*C/C++ demo programs

Building a Pro*C/C++ application

Before building a Pro*C/C++ application, you must set up your environment:

1. You can use the TimesTen OCI and Pro*C/C++ Make files provided with the Quick Start demos to implement appropriate environment settings. These are in the following locations:

   ```
   install_dir/quickstart/sample_code/oci/
   install_dir/quickstart/sample_code/proc/
   ```

   (Unless you installed Quick Start in a different location.)

2. 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 will be set properly if you use the *`install_dir`*`/bin/ttenv` script or `quickstart/ttquickstartenv` script. See "Environment variables" in *Oracle TimesTen In-Memory Database Installation Guide* for information about environment variables and `ttenv`.

Then use steps such as the following 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 *`instant_client`* represents the directory where Oracle Instant Client is installed.

See the Quick Start Pro*C/C++ Makefile in the `quickstart/sample_code/proc` directory for complete, platform-specific examples.

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(instant_client)/sdk/include
   ```

3. Link the resulting object modules with modules in SQLLIB. For example:

   ```
   % gcc -o sample sample.o -L(instant_client)/lib -lclntsh
   ```

Connecting to a TimesTen data store from Pro*C/C++

This section provides information on connecting to TimesTen from a Pro*C/C++ application. Also see "Connecting to a TimesTen data store from OCI" on page 3-7 for information about using the `tnsnames` naming method or easy connect naming method to connect to the database.

> **Note:** A TimesTen connection cannot be inherited from a parent process. If a process opens a database connection before creating 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 supports the following 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	This is the user name.
pwd	This is the user password.
user_string	As an alternative to separate *user* and *pwd* entries, *user_string* is a user name and password separated by a slash, such as `user1/pwd1`. After an "`@`" sign, you can also include 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	This is a database identifier declared in a previous DECLARE DATABASE statement.
host_variable	This is a variable whose value is a database identifier.
connect_string	This is a valid TNS name or easy connect string for a TimesTen database.

Using tnsnames or easy connect

Your EXEC SQL CONNECT syntax can be simplified if you use the Oracle `tnsnames` or easy connect method.

From Pro*C/C++, you can use a host variable to include the user name, password, and a TNS name. For example:

```
EXEC SQL CONNECT :dbstring
```

Where `dbstring` is set to "`user1/pwd1@my_tnsname`".

Alternatively, the host variable could include the user name, password, and an easy connect string. For example, `dbstring` could be set to "`user1/pwd1@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 password in Pro*C/C++ for IMDB Cache

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

For use of Pro*C/C++ with IMDB Cache, TimesTen allows you to pass the Oracle 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. Use the attribute OraclePWD in the connect string, such as in the following example:

```
text *cacheuser = (text *)"cacheuser1";
text *cachepwds = (text *)"ttpwd;OraclePWD=orclpwd";
text *dbname = (text *)"tt_tnsname";
....
EXEC SQL CONNECT :cacheuser IDENTIFIED BY :cachepwds AT :dbname
```

You must always specify OraclePWD, even if the Oracle user's password is the same as the TimesTen user's password. Furthermore, in the circumstance of specifying an Oracle password for IMDB Cache, you must use a form of EXEC SQL CONNECT that specifies the password as a separate host variable. In this example, cacheuser1 is the name of the TimesTen cache user as well as the name of the Oracle user who can access the cached Oracle tables, ttpwd is the password of the TimesTen cache user, orclpwd is the password of the Oracle user, and tt_tnsname is the TNS name of the TimesTen database being connected to. The Oracle database is specified through the TimesTen OracleNetServiceName connection attribute in the odbc.ini or sys.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 data store from OCI" on page 3-7.

Error reporting and handling

Be aware of the following regarding error conditions and error reporting:

- Errors under TimesTen Pro*C/C++ applications return Oracle error codes. TimesTen attempts to report the same Oracle error code as Oracle would under similar conditions. The error messages may come from either the TimesTen catalog or the Oracle catalog. Some error messages may include the accompanying TimesTen error code if appropriate. Pro*C/C++ applications that rely on parsing error codes should be checked.
- 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;
  ```

Pro*C/C++ demo programs

TimesTen ships Pro*C/C++ demo programs. They are in the `quickstart/sample_code/proc` directory. The README file in the directory explains how to compile and run the demos.

Refer to the Quick Start welcome page at *`install_dir`*`/quickstart.html` for information.

TimesTen Pro*C/C++ Precompiler options

This section discusses Pro*C/C++ Precompiler option support by TimesTen.

Precompiler option support

Table 4–2 describes TimesTen Pro*C/C++ Precompiler option support.

Table 4–2 TimesTen Pro*C/C++ Precompiler option support

Option	Notes
AUTO_CONNECT	Supported value: NO (default)
CHAR_MAP	Supported
CINCR	Setting has no effect because TimesTen supports only CPOOL=NO.
CLOSE_ON_COMMIT	Supported value: YES The Oracle default value of NO is overridden by TimesTen.
CMAX	Setting has no effect because TimesTen supports only CPOOL=NO.
CMIN	Setting has no effect because TimesTen supports only CPOOL=NO.
CNOWAIT	Setting has no effect because TimesTen supports only CPOOL=NO.
CODE	Supported
COMP_CHARSET	Supported
CONFIG	Supported
CPOOL	Supported value: NO (default)
CPP_SUFFIX	Supported
CTIMEOUT	Setting has no effect because TimesTen supports only CPOOL=NO.
DB2_ARRAY	Supported
DBMS	Supported value: NATIVE (default)
DEF_SQLCODE	Supported
DEFINE	Supported
DURATION	Setting has no effect because TimesTen does not support objects.
DYNAMIC	Supported
ERRORS	Supported
ERRTYPE	Not supported

*Table 4–2 (Cont.) TimesTen Pro*C/C++ Precompiler option support*

Option	Notes
EVENTS	Both values allowed, but TimesTen OCI does not support Advanced Queueing.
FIPS	Supported
HEADER	Supported
HOLD_CURSOR	Supported
IMPLICIT_SVPT	Supported value: NO (default)
INAME	Supported
INCLUDE	Supported
INTYPE	Supported
LINES	Supported
LNAME	Supported
LTYPE	Supported
MAX_ROW_INSERT	Supported
MAXLITERAL	Supported
MAXOPENCURSORS	Supported
MODE	Supported
NATIVE_TYPES	Supported
NLS_CHAR	Supported
NLS_LOCAL	Supported value: NO (default)
OBJECTS	Setting has no effect because TimesTen does not support objects.
ONAME	Supported
ORACA	Supported
OUTLINE	All values are allowed, but TimesTen does not support Oracle optimization.
OUTLNPREFIX	Both values are allowed, but TimesTen does not support Oracle optimization.
PAGELEN	Supported
PARSE	Supported
PREFETCH	Supported
RELEASE_CURSOR	Supported
RUNOUTLINE	Not supported. Both values (`yes|no`) are allowed but ignored.
SELECT_ERROR	Supported
SQLCHECK	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" on page 4-2.

*Table 4–2 (Cont.) TimesTen Pro*C/C++ Precompiler option support*

Option	Notes
STMT_CACHE	Supported
SYS_INCLUDE	Supported
THREADS	Supported
TYPE_CODE	Supported
UNSAFE_NULL	Supported
USERID	Supported
UTF16_CHARSET	Only the NCHAR_CHARSET setting is supported.
VARCHAR	Supported
VERSION	Setting has no effect because TimesTen does not support objects.

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 the following ways:

- At compile time, either in the configuration file `pcscfg.cfg` or on the Pro*C/C++ command line. A setting on the command line 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, this 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 Transaction Log API (XLA) is a set of C language functions that enable you to implement applications perform the following:

- Monitor TimesTen for changes to specified tables in a local data store.
- Receive real-time notification of these changes.

One of the purposes of XLA is to provide a high-performance, asynchronous alternative to triggers.

XLA also provides functions that enable you to build a custom data replication solution if the TimesTen replication solutions described in *Oracle TimesTen In-Memory Database TimesTen to TimesTen Replication Guide* do not meet your needs.

For a complete description of each XLA function, see Chapter 9, "XLA Reference".

Notes:

- XLA is available on all platforms supported by TimesTen. However, XLA does not support data transfer between different platforms or between 32-bit and 64-bit versions of the same platform.
- XLA does not support applications linked with a driver manager library or the client/server library.

This chapter includes the following topics:

XLA concepts

This section includes the following topics:

- About XLA data types
- Access control impact on XLA
- XLA demo

XLA functions mentioned here are documented in Chapter 9, "XLA Reference".

XLA persistent mode

In normal usage, TimesTen XLA is initialized in persistent mode. In this mode, 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 persistent logging model also allows multiple readers to simultaneously read transaction log updates.

The ttXlaPersistOpen XLA function opens a connection to a TimesTen data store in persistent mode.

When initially created, TimesTen configures a transaction log handle for the same version as the TimesTen release to which the application is linked. You can also use the ttXlaGetVersion and ttXlaSetVersion XLA functions to interoperate with earlier XLA versions.

(It is possible, though not recommended, to use XLA in non-persistent mode. This is discussed in "Using XLA in non-persistent mode" on page 5-39.)

How XLA reads records from the transaction log

As applications modify a TimesTen data store, 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 disk. When XLA is initialized in persistent mode, the XLA application does not have to be concerned with which portions of the transaction log are on disk 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 disk.

Applications can use XLA to monitor the transaction log for changes to the TimesTen data store. 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 data store simultaneously, transaction log records from the different applications will be 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.

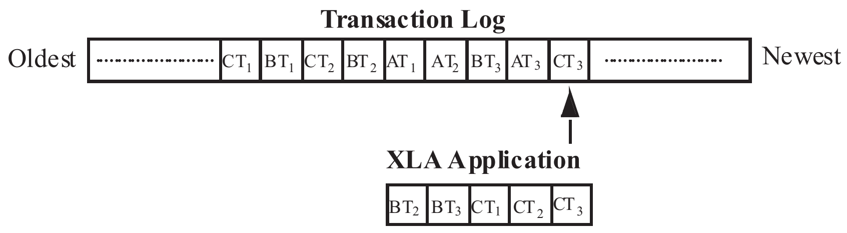
Only the records for committed transactions are returned. They are returned in the order in which their final commit record appears in the transaction log. XLA filters out records associated with changes to the data store that have not yet been committed.

If a change is made but then rolled back, XLA does not deliver the records for the aborted transaction to the application.

Most of these basic XLA concepts are demonstrated in Example 5–1 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



Example 5–1 Reading transaction log records

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:

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. However, for asynchronous materialized views, be aware that the order of XLA notifications for an asynchronous view is not necessarily the same as it would be for the associated detail tables, or the same as it would be for a synchronous view. For example, if there are two inserts to a detail table, they may be done in the opposite order in the asynchronous materialized view. Furthermore, updates may be treated as a delete followed by an insert. Also, multiple operations, such as multiple inserts or multiple deletes, may be combined. Applications that depend on ordering should not use asynchronous materialized views.

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 reader of a persistent transaction log uses a bookmark to maintain its position in the log update stream. 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 data store, so they are persistent across data store connections, shutdowns, and failures.
- A Current Read log record identifier points to the record currently being read from the transaction log.

The rest of this section covers the following:

- Creating or reusing a bookmark
- How bookmarks work
- Replicated bookmarks

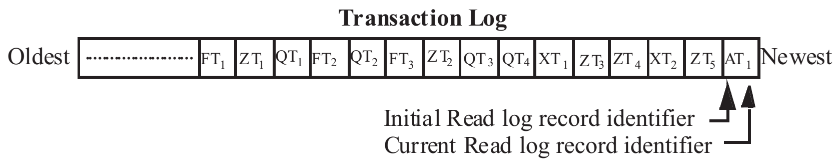
Creating or reusing a bookmark

As described in "Initializing XLA and obtaining an XLA handle" on page 5-10, when you call the ttXlaPersistOpen function to initialize a persistent XLA handle, you include a *`tag`* parameter to identify either a new bookmark or one that already exists in the system, and an *`options`* parameter to specify whether it is a new non-replicated bookmark, a new replicated bookmark, or an already existing (reused) bookmark. At this time, the Initial Read log record identifier associated with the bookmark is read from the data store and cached in the persistent XLA handle (`ttXlaHandle_h`). It designates the start position of the reader in the transaction log.

How bookmarks work

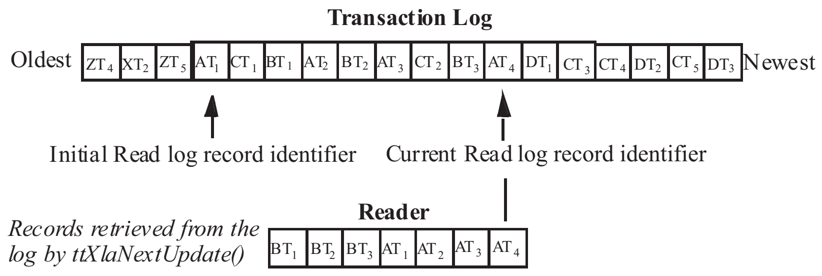
When an application first initializes XLA and obtains an XLA handle, its Current Read log record identifier and Initial Read log record identifier both point to the last record written to the data store, as shown in Figure 5–2.

Figure 5–2 Log record indicator positions upon initializing a persistent XLA handle


Transaction Log
Oldest
FT_1 ZT_1 QT_1 FT_2 QT_2 FT_3 ZT_2 QT_3 QT_4 XT_1 ZT_3 ZT_4 XT_2 ZT_5 AT_1
Newest
Initial Read log record identifier
Current Read log record identifier

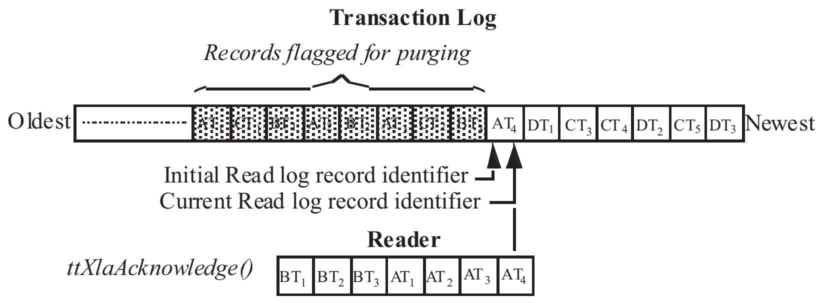
As described in "Retrieving update records from the transaction log" on page 5-12, you use the ttXlaNextUpdate or ttXlaNextUpdateWait function to return a batch of records for committed transactions from the transaction log in the order in which they were committed. Each call to ttXlaNextUpdate resets the bookmark's Current Read log record identifier 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


Transaction Log
Oldest
ZT_4 XT_2 ZT_5 AT_1 CT_1 BT_1 AT_2 BT_2 AT_3 CT_2 BT_3 AT_4 DT_1 CT_3 CT_4 DT_2 CT_5 DT_3
Newest
Initial Read log record identifier
Current Read log record identifier
Reader
Records retrieved from the log by ttXlaNextUpdate()
BT_1 BT_2 BT_3 AT_1 AT_2 AT_3 AT_4

You can use the ttXlaGetLSN and ttXlaSetLSN functions to reread records, as described in "Changing the location of a bookmark" on page 5-38. However, calling the ttXlaAcknowledge function permanently resets the bookmark's Initial Read log record identifier 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


Transaction Log
Records flagged for purging
Oldest
AT_4 DT_1 CT_3 CT_4 DT_2 CT_5 DT_3
Newest
Initial Read log record identifier
Current Read log record identifier
Reader
ttXlaAcknowledge()
BT_1 BT_2 BT_3 AT_1 AT_2 AT_3 AT_4

Note: A `ttXlaAcknowledge` call will reset the bookmark even in the absence of any 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" on page 9-7 for details on how the operation works.

The number of bookmarks created in a TimesTen data store is limited to 64. Each bookmark can be associated with only one active persistent connection at a time. However, a bookmark over its lifetime may be associated with many connections. An application can open a persistent connection, create a new bookmark, associate the bookmark with the connection, read a few records using the bookmark, disconnect from the data store, reconnect to the data store, create a new persistent connection, associate this new connection with the bookmark, and continue reading persistent 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. For a replicated bookmark, operations on the bookmark are replicated to the standby database as appropriate. This allows 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 already returned on the old active store.

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.

Be aware of the following usage notes:

- The position of the bookmark in the standby database will be 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.
- It is recommended that you close and reopen all bookmarks on a data store after it changes from standby to active status, using the ttXlaClose and ttXlaPersistOpen functions. The state of a replicated bookmark on a standby data store does change during normal XLA processing, as the replication agent automatically repositions bookmarks as appropriate on standby data stores. If you attempt to use a bookmark that was open before the data store changed to active status, you will 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.
- If replicated bookmarks already exist at the time you enable the active standby pair scheme, the bookmarks will automatically be added to the replication scheme.
- It is permissible to drop the active standby pair scheme while replicated bookmarks exist. The bookmarks will cease to be replicated at that point.
- You cannot delete replicated bookmarks as long as the replication agent is running.

About XLA data types

XLA data types supported by TimesTen are the same as previous data types when an equivalent data type existed before TimesTen release 7.0. Thus XLA applications that were written before release 7.0 should continue to work without code changes. If you change an XLA application that was written before release 7.0 so that it uses new data types, then you must also modify it to support the new data types.

Table 5–1 shows the data type mapping between internal SQL data types and XLA data types before release 7.0 and since release 7.0. For more information about TimesTen data types, 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 before Release 7.0	XLA data type since Release 7.0
TT_CHAR	SQL_CHAR	TTXLA_CHAR_TT
TT_VARCHAR	SQL_VARCHAR	TTXLA_VARCHAR_TT
TT_NCHAR	SQL_WCHAR	TTXLA_NCHAR_TT
TT_NVARCHAR	SQL_WVARCHAR	TTXLA_NVARCHAR_TT
CHAR	-	TTXLA_CHAR
NCHAR	-	TTXLA_NCHAR
VARCHAR2	-	TTXLA_VARCHAR
NVARCHAR2	-	TTXLA_NVARCHAR
TT_TINYINT	SQL_TINYINT	TTXLA_TINYINT
TT_SMALLINT	SQL_SMALLINT	TTXLA_SMALLINT
TT_INTEGER	SQL_INTEGER	TTXLA_INTEGER
TT_BIGINT	SQL_BIGINT	TTXLA_BIGINT
BINARY_FLOAT	SQL_REAL	TTXLA_BINARY_FLOAT
BINARY_DOUBLE	SQL_DOUBLE	TTXLA_BINARY_DOUBLE
TT_DECIMAL	SQL_DECIMAL	TTXLA_DECIMAL_TT
NUMBER	-	TTXLA_NUMBER
NUMBER(*p*,*s*)	-	TTXLA_NUMBER
FLOAT	-	TTXLA_NUMBER
TT_TIME	SQL_TIME	TTXLA_TIME
TT_DATE	SQL_DATE	TTXLA_DATE_TT
TT_TIMESTAMP	SQL_TIMESTAMP	TTXLA_TIMESTAMP_TT
DATE	-	TTXLA_DATE
TIMESTAMP	-	TTXLA_TIMESTAMP
TT_BINARY	SQL_BINARY	TTXLA_BINARY
TT_VARBINARY	SQL_VARBINARY	TTXLA_VARBINARY
ROWID	-	TTXLA_ROWID

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. XLA data type conversion functions include the following:

- ttXlaDateToODBCCType
- ttXlaDecimalToCString
- ttXlaNumberToCString

- ttXlaNumberToDouble
- ttXlaNumberToBigInt
- ttXlaNumberToInt
- ttXlaNumberToSmallInt
- ttXlaNumberToTinyInt
- ttXlaNumberToUInt
- ttXlaOraDateToODBCTimeStamp
- ttXlaOraTimeStampToODBCTimeStamp
- ttXlaRowidToCString
- ttXlaTimeToODBCCType
- ttXlaTimeStampToODBCCType

Access control impact on XLA

"Considering TimesTen features for access control" on page 1-27 provides a brief overview of how TimesTen access control affects operations in the database. Access control includes impact on XLA, as follows:

- Any XLA functionality requires the system privilege XLA. This includes:
  - Connecting to TimesTen as an XLA reader, such as by the `ttXlaPersistOpen` C function.
  - Executing any other XLA-related TimesTen C functions. These are documented in Chapter 9, "XLA Reference".
  - Executing any XLA-related TimesTen built-in functions. The functions `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 and SELECT ANY SEQUENCE system privileges.
- A user with the XLA privilege can capture DDL statement records that occur in the database. Note that as a result, the user can obtain information about database objects that he or she has not otherwise been granted access to.

XLA demo

TimesTen provides the `xlaSimple` demo showing how to use many of the XLA functions described in this chapter. It is located in the `quickstart/sample_code/odbc/xla` directory:

See "About the TimesTen C demos" on page 2-4 for an overview of TimesTen demo programs for C developers. Refer to *`install_dir`*`/quickstart.html` for details. The README file in the `odbc` directory contains instructions for building and running `xlaSimple`, among others.

Most of this chapter, including the sample code shown in "Writing an XLA event-handler application" starting immediately below, is based on the `xlaSimple` demo. For this demo, a table `MYDATA` has been created in the `APPUSER` schema. While you are logged in as `APPUSER`, you will be making updates to the table. While you are logged in as `XLAUSER`, the `xlaSimple` demo reports on the updates.

To run the demo, execute `xlaSimple` at one command prompt. You will be prompted for the password of `XLAUSER`, which is specified when the sample database is created. Start `ttIsql` at a separate command prompt, connecting to the TimesTen sample database as `APPUSER`. Again, you will be prompted for a password that is specified 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

This section describes the general procedures for writing an XLA application that detects and reports changes to selected tables in a data store. With the possible exception of "Inspecting column data" on page 5-17, the procedures described in this section are applicable to most XLA applications.

The procedures described in this section are:

- Obtaining a data store 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
- Handling XLA errors
- 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` demo application.

XLA functions mentioned here are documented in Chapter 9, "XLA Reference".

Important: In addition to `#include` files noted in "TimesTen #include files" on page 1-7, 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 "Handling XLA errors" on page 5-28 for information on error handling.

Obtaining a data store 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 TimesTen data store.

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 TimesTen data store:

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

Call the `SQLDriverConnect` ODBC function to connect to the data store 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 as described in "Obtaining a data store connection handle" on page 5-9, you can initialize XLA and obtain an XLA handle to access the transaction log. Create only one XLA handle per ODBC connection. If your application uses multiple XLA reader threads, create a separate XLA handle and ODBC connection for each thread.

This section describes how to initialize XLA in persistent mode, which is the recommended mode.

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 indicates that you want to create a new non-replicated bookmark. Alternatively, use the XLAREPL option to create a replicated bookmark. In either case, the operation will fail if the bookmark already exists.

To use a bookmark that already exists, call ttXlaPersistOpen with the XLAREUSE option. For 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 will be 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.

Note: When an XLA handle is initially created, TimesTen configures it for the same version as the TimesTen release to which the application is linked. If you must interoperate with earlier XLA versions, you can use the ttXlaGetVersion and ttXlaSetVersion functions.

Specifying which tables to monitor for updates

After initializing XLA and obtaining an XLA handle, as described in "Initializing XLA and obtaining an XLA handle" on page 5-10, you can specify which tables or materialized views you want to monitor for update events.

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, ADDCOLS, DRPCOLS, TRUNCATE, SETTBLI, and SETCOLI transactions.

The ttXlaTableStatus function indicates that DML records associated with the specified table should be monitored by the current bookmark. 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.

Example 5–2 Specifying a table to monitor for updates

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 indicates that you want XLA to track 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. 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's bookmark in the transaction log, as described in "How bookmarks work" on page 5-4.

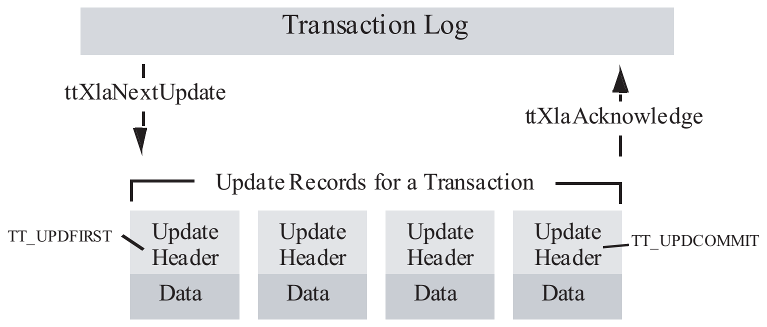
Note: The ttXlaAcknowledge function is an expensive operation and should be used only as necessary.

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 data store.

Figure 5–5 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



Example 5–3 Retrieving update records from the transaction log

The `xlaSimple` demo 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.

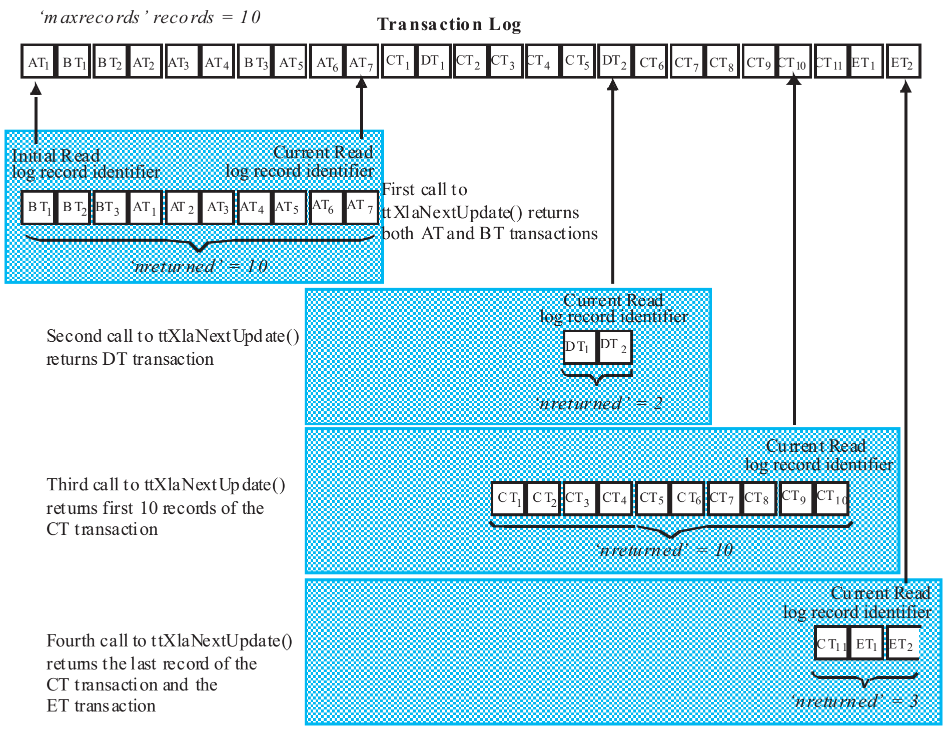
Next, call ttXlaNextUpdateWait, passing these values to obtain a batch of ttXlaUpdateDesc_t records in `arry`. Then process each record in `arry` by passing it to the `HandleChange()` function described in Example 5–4. After all records are processed, call ttXlaAcknowledge to reset the bookmark position.

```
#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 "Handling XLA errors" on page 5-28 */
      }

/* 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 "Handling XLA errors" on page 5-28 */
    }
} /* 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" on page 9-32 and "ttXlaNextUpdateWait" on page 9-34 for details of the parameters of these functions.

Figure 5–6 Records retrieved when maxrecords=10


'maxrecords' records = 10
Transaction Log
AT1 BT1 BT2 AT2 AT3 AT4 BT3 AT5 AT6 AT7 CT1 DT1 CT2 CT3 CT4 CT5 DT2 CT6 CT7 CT8 CT9 CT10 CT11 ET1 ET2
Initial Read log record identifier
Current Read log record identifier
BT1 BT2 BT3 AT1 AT2 AT3 AT4 AT5 AT6 AT7
'nreturned' = 10
First call to ttXlaNextUpdate() returns both AT and BT transactions
Second call to ttXlaNextUpdate() returns DT transaction
Current Read log record identifier
DT1 DT2
'nreturned' = 2
Third call to ttXlaNextUpdate() returns first 10 records of the CT transaction
Current Read log record identifier
CT1 CT2 CT3 CT4 CT5 CT6 CT7 CT8 CT9 CT10
'nreturned' = 10
Fourth call to ttXlaNextUpdate() returns the last record of the CT transaction and the ET transaction
Current Read log record identifier
CT11 ET1 ET2
'nreturned' = 3

XLA reads records from either a memory buffer or transaction log files on disk, as described in "How XLA reads records from the transaction log" on page 5-2. 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 allows XLA applications to see changes as soon as the changes are made and with minimal latency. The tradeoff is that there may be times when fewer changes are returned than the number requested by the ttXlaNextUpdate or ttXlaNextUpdateWait *maxrecords* parameter.

Note: Some XLA applications may improve performance by making 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

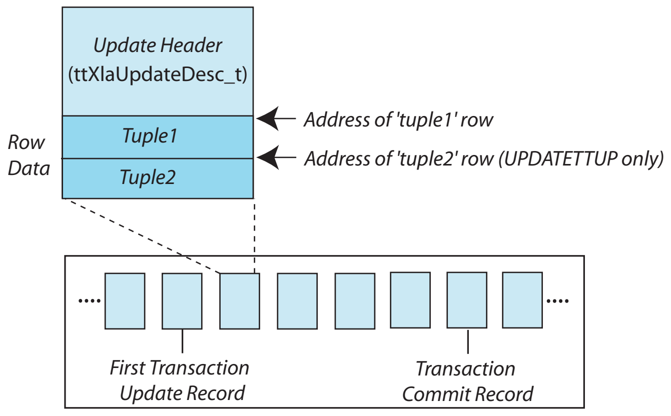
Now that there is an array of update records where the type of operation each record represents is known, the returned row data can be inspected.

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


Update Header
(ttXlaUpdateDesc_t)
Row Data
Tuple1
Tuple2
Address of 'tuple1' row
Address of 'tuple2' row (UPDATETTUP only)
....
....
First Transaction Update Record
Transaction Commit 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 data store. 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 Example 5–4 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 Example 5–4.

Example 5–4 Inspecting record headers for SQL operation type

This example 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. To keep this example simple, all other operations are ignored.

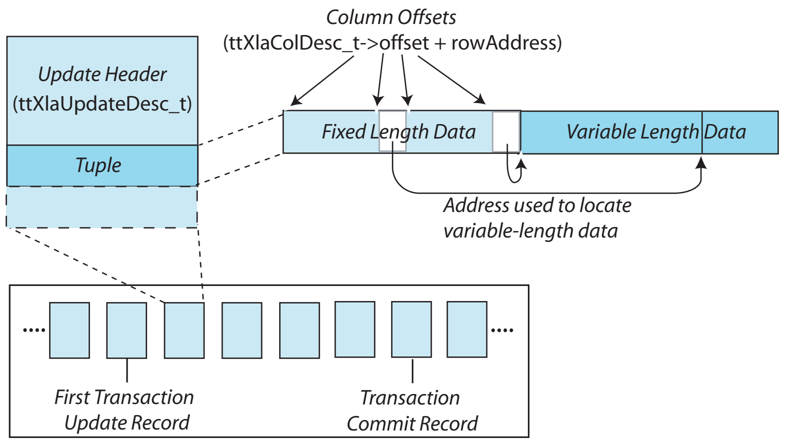
The `HandleChange()` function handles each type of SQL operation differently before calling the `PrintColValues()` function described in Example 5–13.

```
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's for the table we're 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

As described in "Inspecting record headers and locating row addresses" on page 5-15, zero to two rows of data may be returned in an update record after the ttXlaUpdateDesc_t structure. 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


Column Offsets
(ttXlaColDesc_t->offset + rowAddress)
Update Header
(ttXlaUpdateDesc_t)
Tuple
Fixed Length Data
Variable Length Data
Address used to locate variable-length data
First Transaction Update Record
Transaction Commit Record

The procedures for inspecting column data are described in the following sections:

- 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: a PrintColValues() function

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" on page 5-10 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.

Example 5–5 Using column descriptions

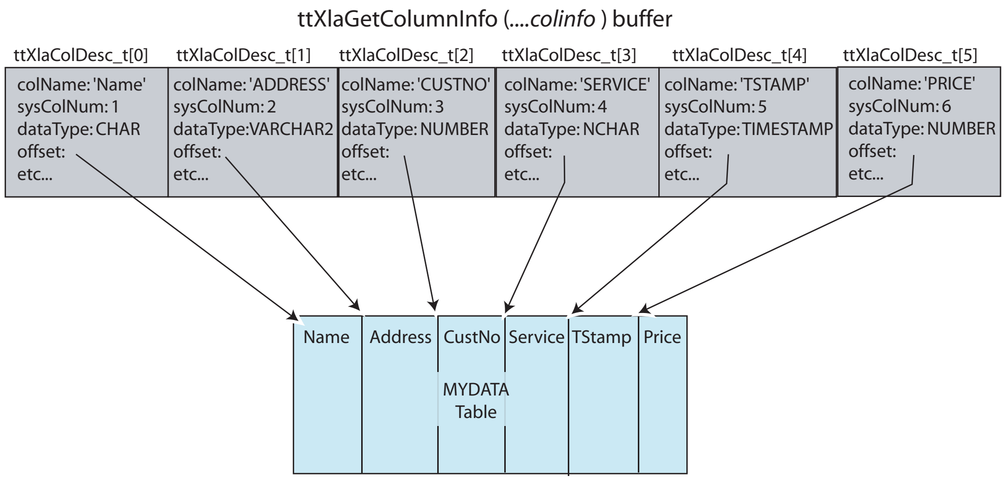
The sample code from the `xlaSimple` demo below guesses the maximum number of returned columns (MAX_XLA_COLUMNS), which sets the size of the buffer `xla_column_defs` to hold the returned ttXlaColDesc_t structures. An alternative and more precise way to set the *`maxcols`* parameter would be to call the 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 "Handling XLA errors" on page 5-28 */
}
```

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

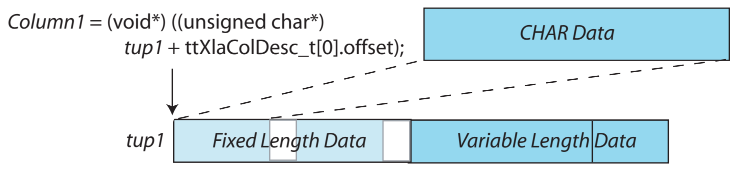

ttXlaGetColumnInfo (....colinfo) buffer
ttXlaColDesc_t[0]
ttXlaColDesc_t[1]
ttXlaColDesc_t[2]
ttXlaColDesc_t[3]
ttXlaColDesc_t[4]
ttXlaColDesc_t[5]
colName: 'Name' sysColNum: 1 dataType: CHAR offset: etc...
colName: 'ADDRESS' sysColNum: 2 dataType:VARCHAR2 offset: etc...
colName: 'CUSTNO' sysColNum: 3 dataType: NUMBER offset: etc...
colName: 'SERVICE' sysColNum: 4 dataType: NCHAR offset: etc...
colName: 'TSTAMP' sysColNum: 5 dataType: TIMESTAMP offset: etc...
colName: 'PRICE' sysColNum: 6 dataType: NUMBER offset: etc...
Name
Address
CustNo
Service
TStamp
Price
MYDATA Table

Each ttXlaColDesc_t structure returned by ttXlaGetColumnInfo includes 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.

Figure 5–10 Locating fixed-length data in a row


Column1 = (void*) ((unsigned char*)
tup1 + ttXlaColDesc_t[0].offset);
CHAR Data
tup1
Fixed Length Data
Variable Length Data

Example 5–6 Reading fixed-length column data

See Example 5–13, "Complete 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 (Example 5–4) and the offset from the first ttXlaColDesc_t structure returned by the ttXlaGetColumnInfo function (Example 5–5), 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 in order to obtain the string, as shown for the SQL_WCHAR case in Example 5–13.

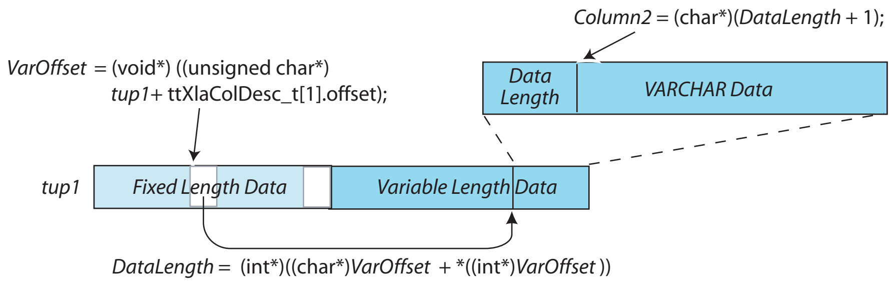
Other fixed-length data types can be cast to their corresponding C types. Complex fixed-length data types, such as DATE, TIME, and DECIMAL values, are stored in an internal TimesTen format, but can be converted by applications to their corresponding ODBC C value using the XLA conversion functions, as described in "Converting complex data types" on page 5-22.

Note: Strings returned by XLA are not null-terminated. See "Null-terminating returned strings" on page 5-21.

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 4-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 *n* bytes (where *n* is 4 on 32-bit platforms or 8 on 64-bit platforms) 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


Column2 = (char*)(DataLength + 1);
VarOffset = (void*) ((unsigned char*)
tup1+ ttXlaColDesc_t[1].offset);
Data Length
VARCHAR Data
tup1
Fixed Length Data
Variable Length Data
DataLength = (int*)((char*)VarOffset + *((int*)VarOffset))

Example 5–7 Reading NOT INLINE variable-length column data

See Example 5–13, "Complete 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. The value at this address is the 4-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`. Assuming the operation is performed on a 32-bit platform, the first four bytes at the `DataLength` location is the length of the data. The next byte after `DataLength` is the beginning of the actual data (`Column2`).

The sample code here assumes the operation is performed on a 32-bit platform, so `DataLength` is initialized as a 32-bit type. On a 64-bit platform, `DataLength` must be initialized as a 64-bit type and the `Column2` data would appear 64 bits + 1 after the offset address, `DataLength`.

```
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 Example 5–13.

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, such as `'\0'`, after the last character in the string.

The procedures for null-terminating fixed-length and variable-length strings are slightly different. The procedure for null-terminating fixed-length strings is described in Example 5–8. Example 5–9 describes the procedure for null-terminating variable-length strings of a known size. Example 5–10 describes the procedure for strings of an unknown size.

Example 5–8 Null-terminating fixed-length strings

See Example 5–13, "Complete 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 Example 5–6, 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" on page 5-19.

Example 5–9 Null-terminating variable-length strings of known size

(See Example 5–13, "Complete PrintColValues() function" for a complete working example of computations such as those shown here.)

If the `Column2` string obtained in Example 5–7 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 Example 5–10.

Example 5–10 Null-terminating variable-length strings of unknown size

If the `Column2` string obtained in Example 5–7 is of an unknown size, you might statically allocate a buffer large enough to hold a string of up to 10000 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

Values for complex data types such as TT_DATE, TT_TIME, and TT_DECIMAL are stored in an internal TimesTen format that can be converted into corresponding ODBC C types using the XLA type conversion functions. Table 5–2 contains descriptions of these conversion functions.

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 TT_DECIMAL value to a string value
ttXlaDateToODBCCType	Internal TTXLA_DATE_TT value to an ODBC C value
ttXlaDecimalToCString	Internal TTXLA_DECIMAL_TT value to a character string
ttXlaNumberToBigInt	Internal TTXLA_NUMBER value to a TT_BIGINT value

Table 5–2 (Cont.) XLA data type conversion functions

Function	**Converts**
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

These conversion functions can be used on row data included in the ttXlaUpdateDesc_t types: UPDATETUP, INSERTTUP and DELETETUP.

Example 5–11 Converting complex data types

(See Example 5–13, "Complete PrintColValues() function" for a complete working example of computations such as those shown here.)

If you use the address of the `tup1` row obtained earlier in the `HandleChange()` function (Example 5–4) and the offset from the fifth ttXlaColDesc_t structure returned by the ttXlaGetColumnInfo function (Example 5–5), 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 "Handling XLA errors" on page 5-28 */
  }
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 (Example 5–4) and the offset from the sixth ttXlaColDesc_t structure returned by the ttXlaGetColumnInfo function (Example 5–5), 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 "Handling XLA errors" on page 5-28 */
  }

printf(" %s: %s\n",
      ((unsigned char*) xla_column_defs[5].colName),
      decimalData);
```

Detecting null values

For columns that can have null values, `ttXlaColDesc_t ->`*`nullOffset`* points to a null byte in the record. The *`nullOffset`* is 1 if the column is null, or 0 if it is not null.

To determine if a column value is null, first check if the *`nullOffset`* is 0, in which case it is not a nullable value. If *`nullOffset`* is nullable, then check the value at the *`nullOffset`* to see if it is 1 or 0.

Example 5–12 Deleting null values

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");
  }
}
```

Putting it all together: a PrintColValues() function

Example 5–13 shows 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.

Example 5–13 Complete PrintColValues() function

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 "Handling XLA errors" on page 5-28.

```
void PrintColValues(void * tup)
{

  SQLRETURN rc ; /* make these global?? */
  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 */
  }
```

Handling 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 as described in "Terminating an XLA application" on page 5-31.

An error can be checked using either its error code (error number) or `tt_Err` string. For the complete list of TimesTen error codes and error strings, see the *`install_dir`*`/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" on page 1-28.

Example 5–14 Checking the return code and calling the error-handling function

This 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" on page 5-31.

```
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.

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.

Depending on your application, you may be required to act on specific XLA errors, including those shown in Table 5–3 below.

Table 5–3 XLA errors and codes

Error	Code
tt_ErrDbAllocFailed	802 (transient)
tt_ErrCacheXlaNotRead	5023
tt_ErrCacheXLARollbackFailed	5031
tt_ErrCondLockConflict	6001 (transient)

Table 5–3 (Cont.) XLA errors and codes

Error	Code
tt_ErrDeadlockVictim	6002 (transient)
tt_ErrTimeoutVictim	6003 (transient)
tt_ErrPermSpaceExhausted	6220 (transient)
tt_ErrTempSpaceExhausted	6221 (transient)
tt_ErrBadXlaRecord	8024
tt_ErrXlaBookmarkUsed	8029
tt_ErrXlaBookmarkBad	8030
tt_ErrXlaLsnBad	8031
tt_ErrXlaNoSQL	8034
tt_ErrXlaNoLogging	8035
tt_ErrXlaParameter	8036
tt_ErrXlaTableDiff	8037
tt_ErrXlaTableSystem	8038
tt_ErrXlaTupleMismatch	8046
tt_ErrXlaDedicatedConnection	8047

Example 5–15 Calling the handleXLAerror() function

This example shows `handleXLAerror()`, the error function for the `xlaSimple` demo 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 persistent XLA applications are connected and using bookmarks that are tracking the table to be dropped, then perform the following tasks:

1. Each persistent 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 persistent 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 the next section, "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" on page 5-4, 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 will free the disk 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 disk space.

Notes:

- 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.
- 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*.

Example 5–16 Deleting bookmarks

The `InitHandler()` function in the `xlaSimple` demo deletes the XLA bookmark upon exit, as shown in this example.

```
if (deleteBookmark) {
    ttXlaDeleteBookmark(xla_handle);
    if (rc != SQL_SUCCESS) {
    /* See "Handling XLA errors" on page 5-28 */
    }
    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, you should gracefully exit by rolling back uncommitted transactions and freeing all handles. Also unsubscribe the tables and materialized views being monitored, unless your application must capture updates that occur when it is not connected. You may or may not want to delete the XLA bookmark when the program terminates, as described in "Deleting bookmarks" on page 5-30.

Free your resources in reverse order of allocation. For each table and materialized view tracked by XLA, call the ttXlaTableStatus function and set the *`newstatus`* parameter to 0. This unsubscribes the table or materialized view from XLA. Next, call ttXlaClose to release the XLA handle.

Call appropriate ODBC functions. Call `SQLTransact` with SQL_ROLLBACK to roll back any uncommitted transaction. Next, call `SQLDisconnect` to close the connection to TimesTen. Finally, call `SQLFreeConnect` and `SQLFreeEnv` to release the connection handle (`hdbc`) and environment handle (`henv`) and to free the associated memory.

Example 5–17 Terminating an XLA application

This example shows `TerminateGracefully()`, the XLA termination function in the `xlaSimple` demo.

```
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

If the TimesTen replication solutions described in *Oracle TimesTen In-Memory Database TimesTen to TimesTen Replication Guide* do not meet your needs, you can use XLA functions to replicate updates from one data store to another.

Note: You cannot use XLA to replicate updates between different platforms or between 32-bit and 64-bit versions of the same platform.

In this section, the sending data store is referred to as the master and the receiving data store as the subscriber. To use XLA to replicate changes between data stores, first use the ttXlaPersistOpen function to initialize the XLA handles, as described in "Initializing XLA and obtaining an XLA handle" on page 5-10.

After the XLA handles have been initialized for the data stores, take the steps described in the following sections:

- Checking table compatibility between data stores
- Replicating updates between data stores
- Handling timeout and deadlock errors
- Checking for update conflicts

XLA functions mentioned here are documented in Chapter 9, "XLA Reference".

Checking table compatibility between data stores

Before transferring update records from one data store to the other, verify that the tables in the master and subscriber data stores are compatible with one another:

- You can check the descriptions of a table and its columns by using the ttXlaTableByName, ttXlaGetTableInfo, and ttXlaGetColumnInfo functions. See "Checking table and column descriptions" immediately below.
- You can check the table and column versions of a specific XLA record by using the ttXlaVersionTableInfo and ttXlaVersionColumnInfo functions. See "Checking table and column versions" on page 5-34.

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. These operations are described in "Specifying which tables to monitor for updates" on page 5-11 and "Obtaining column descriptions" on page 5-17. 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 shows this.

Example 5–18 Checking table and column descriptions for compatibility

```
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.

The following example verifies that the table associated with the *`pXlaRecord`* update record from the *`pCmd`* source is compatible with the *`hXlaTarget`* target.

Example 5–19 Checking table and column versions for compatibility

```
BOOL CUTLCheckXlaTable (SCOMMAND* pCmd,
                        ttXlaHandle_h hXlaTarget,
                        const ttXlaUpdateDesc_t* pXlaRecord)
{
  /* locals */
  BOOL bStatus = TRUE;
  SQLRETURN rc;
  ttXlaTblVerDesc_t tblVerDescSource;
  ttXlaColDesc_t colDescSource [255];
  SQLINTEGER iColumnCount = 0, iCompatible = 0;

  /* only certain update record types should be checked */
  if (pXlaRecord->type == INSERTUP ||
    pXlaRecord->type == UPDATETUP ||
    pXlaRecord->type == DELETETUP)
  {
 /* get source table description associated with this record */
    /* at the time it was generated */
    rc = ttXlaVersionTableInfo (pCmd->pCtx->con->hXla,
       (ttXlaUpdateDesc_t*) pXlaRecord,
       &tblVerDescSource);

 /* get the source column descriptors for this table */
    /* at the time the record was generated */
    if (bStatus)
    {
      SQLINTEGER iColsReturned = 0;
```

```
rc = ttXlaVersionColumnInfo (pCmd->pCtx->con->hXla,
  (ttXlaUpdateDesc_t*) pXlaRecord,
  colDescSource, 255, &iColsReturned);
```

You can then pass this information to the ttXlaTableCheck function to determine whether the record can be safely applied to another data store.

```
/* check compatibility */
    if (bStatus)
    {
      rc = ttXlaTableCheck (hXlaTarget,
        &tblVerDescSource.tblDesc, colDescSource,
        &iCompatible);
```

Replicating updates between data stores

When you are ready to begin replication, use the ttXlaNextUpdate or ttXlaNextUpdateWait function to obtain batches of update records from the master data store and ttXlaApply to write the records to the subscriber data store. The following example shows this.

Example 5–20 Replicating updates between data stores

```
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 "Handling XLA errors" on page 5-28 */
    }

    /* 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 "Handling XLA errors" on page 5-28 and */
      /* "Handling timeout and deadlock errors" on page 5-36 */
      }
    }

    /* print number of updates applied */
    printf("Records applied successfully : %d\n",records);

  } while (records != 0);
```

Important: 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" on page 9-70 and "ttXlaNodeHdr_t" on page 9-69.

Handling timeout and deadlock errors

The return code from ttXlaApply indicates whether the update was successful. 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 re-executing it. Other errors may be persistent, such as 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 data store, 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 already been applied to the subscriber data store.

To enable recovery from transient errors, you should keep track of transaction boundaries on the master data store 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 already applied to the subscriber data store. 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" on page 5-38. 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 data stores, you may encounter update conflicts. Update conflicts are described in detail in "Conflict Resolution and Failure Recovery" in *Oracle TimesTen In-Memory Database TimesTen to TimesTen 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 data store. If the old row value in the update description does not match the corresponding row in the subscriber data store, an update conflict is assumed. In this case, ttXlaApply does not apply the update to the subscriber and returns an `sb_ErrXlaTupleMismatch` error.

Replicating updates to a non-TimesTen data store

If you are replicating changes to a non-TimesTen data store, 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.

The ttXlaGenerateSQL function accepts a ttXlaUpdateDesc_t record as a parameter and outputs its SQL equivalent into a buffer.

Important: 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.

Example 5–21 Replicating updates to a non-TimesTen data store

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

The following sections describe how to use additional XLA features:

- Changing the location of a bookmark
- Passing application context
- Using XLA in non-persistent mode

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. 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. This 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" on page 5-19.

Note: A context value will be 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 `contextBuffer` is a CHAR array that is declared to be large enough to accommodate the longest application context that you will 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.

Using XLA in non-persistent mode

TimesTen XLA is normally used in persistent mode, but non-persistent mode is also supported. This is primarily for backward compatibility. In non-persistent mode, transaction log updates are maintained in an XLA staging buffer, which is where XLA stages the update records obtained from the transaction log and makes them available to be read by the application. However, the staging buffer can be accessed by only one reader at a time and all of the buffered data is lost when the computer or data store is shut down.

The ttXlaOpenTimesTen XLA function opens a connection to a TimesTen data store in non-persistent mode.

Information for operating XLA in non-persistent mode is described in the following sections:

- How non-persistent mode differs from persistent mode
- Initializing XLA in non-persistent mode
- Configuring the staging buffer
- Retrieving and resetting the buffer status

How non-persistent mode differs from persistent mode

Non-persistent mode differs from persistent mode as follows:

- Transaction update records are maintained in a transient staging buffer, rather than being obtained directly from a transaction log buffer or transaction log file on disk.
- If the staging buffer becomes full, transactions cannot complete until you empty the buffer.
- You cannot use XLA bookmarks.
- You must configure the size of the staging buffer by using ttXlaConfigBuffer.
- You can check the status of the staging buffer by calling the ttXlaStatus function.
- You can reset the staging buffer status by calling the ttXlaResetStatus function.
- Only one XLA application can read from the staging buffer at any one time.

All other XLA procedures, excluding those related to bookmarks, are the same as those described for persistent mode in "Writing an XLA event-handler application" on page 5-9.

Initializing XLA in non-persistent mode

After initializing ODBC and obtaining an environment handle, connection handle, and statement handle as described in "Obtaining a data store connection handle" on page 5-9, you can initialize XLA in non-persistent mode and obtain an XLA handle to access the transaction log. Though you may have multiple open XLA connections in non-persistent mode, you must coordinate reads so that only one connection accesses the staging buffer at any one time.

Initializing XLA in non-persistent mode is similar to initializing in persistent mode, as described in "Initializing XLA and obtaining an XLA handle" on page 5-10, but you are not required to identify a bookmark. Simply initialize an XLA handle as type `ttXlaHandle_h` and pass the address to the `ttXlaOpenTimesTen` function to obtain the XLA handle:

```
ttXlaHandle_h xla_handle;
rc = ttXlaOpenTimesTen(hdbc, &xla_handle);
```

Configuring the staging buffer

After initializing XLA in non-persistent mode, use the ttXlaConfigBuffer function to configure the size of the XLA staging buffer. Only one staging buffer may be configured for a data store. The staging buffer size setting is guaranteed to survive normal disconnects. However, the size setting may not survive an abnormal termination, depending on whether a checkpoint was done.

When finished using XLA, you can delete the staging buffer by setting its size to 0.

See "ttXlaConfigBuffer" on page 9-13 for details.

Retrieving and resetting the buffer status

When operating XLA in non-persistent mode, you can use the ttXlaStatus function to retrieve status information on the transaction log buffer and your XLA staging buffer.

See "ttXlaStatus" on page 9-54 for details.

6 Distributed Transaction Processing: XA

This chapter describes the TimesTen implementation of the X/Open XA standard.

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.

The purpose of this chapter is to provide information specific to the TimesTen implementation of XA and is intended to be used with 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 through the following location:

  `http://www.oracle.com/technology/documentation/bea.html`

This chapter includes the following topics:

- Overview of XA
- Using XA in TimesTen
- XA support through the Windows ODBC driver manager
- Configuring Tuxedo to use TimesTen XA

Important:

- The TimesTen XA implementation does not work with IMDB Cache. The start of any XA transaction will fail if the cache agent is running.
- You cannot execute an XA transaction if replication is enabled.
- DDL statements cannot be executed within an XA transaction.

Overview of XA

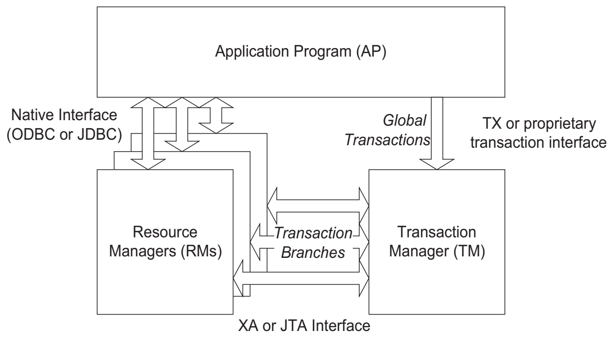
This section provides a brief overview of the following XA concepts:

- X/Open DTP model
- Two-phase commit

X/Open DTP model

Figure 6–1 illustrates the interfaces defined by the X/Open DTP model.

Figure 6–1 Distributed transaction processing model



The TX interface is what applications use to communicate with a transaction manager. The figure shows an application communicating global transactions to the 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.

In the context of TimesTen XA, the resource managers can be a collection of TimesTen data stores, or data stores in combination with other commercial databases that support XA.

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" on page 6-4 for more information.

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.

Using XA in TimesTen

The TimesTen implementation of XA 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:

TimesTen data store 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 disk, regardless of the value set in the `DurableCommits` connection attribute or by the `ttDurableCommit` built-in procedure. 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.

Rollback of transactions requires transaction logging, which is always enabled with XA.

Global transaction recovery in TimesTen

When a TimesTen data store is loaded from disk to recover after a failure or unexpected termination, any global transactions that were prepared but not committed are left pending, or in doubt. Normal 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 normal. 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 normally.

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 to independently commit or abort 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

This section describes some of the issues concerning the use of TimesTen XA functions, which are of interest if you are writing your own transaction manager.

xa_open()

The *`xa_info`* string used by `xa_open()` should be a connection string identical to that supplied to `SQLDriverConnect`, such as:

```
"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 AUTOCOMMIT OFF when it opens an XA connection.

A connection opened with `xa_open()` must be closed with a call to `xa_close()`.

Note: Privilege to connect to a TimesTen data store must be explicitly granted to every user other than the instance administrator, through the CREATE SESSION privilege. Refer to "Access control for connections" on page 1-6.

xa_close()

The *`xa_info`* string used by `xa_close()` should be empty.

Transaction id (XID) parameter

XA uniquely identifies global transactions through the use of 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`. Be aware that this can cause problems when 32-bit client applications connect to a 64-bit server, or 64-bit client applications connect to a 32-bit server. This is because `long` is a 32-bit integer on 32-bit platforms but a 64-bit integer on 64-bit platforms, other than 64-bit Windows. Hence, 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

TimesTen provides the function `tt_xa_context()`, which allows you to acquire the ODBC connection handle associated with an XA connection opened by `xa_open()`.

Syntax

```
#include <tt_xa.h>
int tt_xa_context(int *rmid, SQLHENV *henv, SQLHDBC *hdbc);
```

Parameters

Parameter	Type	Description
rmid (input)	int	If the specified resource manager ID (*rmid*) 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 (output)	SQLHENV	The environment handle associated with the current `xa_open()` context.
hdbc (output)	SQLHDBC	The connection handle associated with the current `xa_open()` context.

Return values

0: Success

1: *rmid* not found

-1: Invalid parameter

Example

In the following example, Tuxedo has already used `xa_open()` and `xa_start()` to open a connection to the data store and start a transaction. In order 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()`.

Example 6–1 Using tt_xa_context() to locate handles

```
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

This section describes some of the TimesTen issues to be aware of when calling ODBC functions using an ODBC handle associated with an XA connection opened by `xa_open()`.

AUTOCOMMIT

To simplify operation and prevent possible contradictions, `xa_open()` automatically turns AUTOCOMMIT OFF when it opens an XA connection.

AUTOCOMMIT may subsequently be turned ON or OFF while performing local transaction work, but must be turned OFF before calling `xa_start()` 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 AUTOCOMMIT ON in this case will result 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 do a local COMMIT or ROLLBACK using `SQLTransact` results in the following error:

```
Error 11031- "Illegal combination of local transaction and global (XA)
transaction"
```

Closing 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

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, which, in turn, points to the appropriate function in the resource manager. This allows resource managers to 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_t

The `xa_switch_t` structure defined by the XA specification is as follows:

```
/*
* 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 */
```

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, 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 "Warnings and Errors" in *Oracle TimesTen In-Memory Database Error Messages and SNMP Traps* for the complete list of errors.

XA support through the Windows ODBC driver manager

This section discusses issues and procedures for using XA with the Windows ODBC driver manager. (UNIX ODBC driver managers are not considered.)

Issues to consider

XA support through the ODBC driver manager requires special handling. There are two fundamental problems:

- 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()`. However, 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 DBMS 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 allow 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, `ttxadm1121.dll`, for XA functions. Applications can make XA calls directly, but must link in the extension library.

To link with the `ttxadm1121.dll` library, applications must include `ttxadm1121.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 ttxadm1121.lib odbc32.lib
```

Note: The XA driver manager extension is implemented only for 32-bit Windows applications.

Configuring Tuxedo to use TimesTen XA

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.

To configure Tuxedo to use the TimesTen resource managers, perform the following tasks:

- Update the $TUXDIR/udataobj/RM file
- Build the Tuxedo transaction manager server
- Update the GROUPS section in the UBBCONFIG file
- Compile the servers

Note: The examples in this section use the direct driver. You can also use the client/server library or driver manager library with the XA extension library.

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 UNIX platforms, add the following:

```
TimesTen:tt_xa_switch:-Linstall_dir/lib -ltten
```

For example:

```
TimesTen:tt_xa_switch:-L/opt/TimesTen/giraffe/lib -ltten
```

On Windows platforms, add the following:

```
TimesTen;tt_xa_switch;install_dir\lib\ttdv1121.lib
```

For example:

```
TimesTen;tt_xa_switch;C:\TimesTen\giraffe\lib\ttdv1121.lib
```

Note: The *`install_dir`* is the path to the TimesTen home directory.

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 UNIX platforms, the commands are:

```
buildtms -o TMS_TT -r TimesTen -v
cp TMS_TT $TUXDIR/bin
```

On Windows platforms, the commands are:

```
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 described in the preceding section.

```
TMSNAME=TMS_TT
```

Enter a line for each TimesTen resource manager that includes 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 TimesTen data store.

Note that on Windows, Tuxedo servers run as user SYSTEM. Add the UID connection attribute to the OPENINFO string to specify a user other than SYSTEM for the connection:

```
OPENINFO="TimesTen:DSN=DSNname;UID=user"
```

Do not specify a CLOSEINFO parameter for any TimesTen resource manager.

Example 6–2 shows the portions of a UBBCONFIG file used to configure two TimesTen resource managers, GROUP1 and GROUP2.

Example 6–2 Configuring TimesTen resource managers

```
*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 *`install_dir`*`/include` directory that holds the TimesTen header files. Then use the `buildserver` command to construct an Oracle Tuxedo ATMI server load module.

On UNIX platforms, enter the following.

```
export CFLAGS=-Iinstall_dir/include
buildserver -o server -f server.c -r TimesTen -s SERVICE
```

WINDOWS

On Windows platforms, enter the following.

```
set CFLAGS=-Iinstall_dir\Include
buildserver -o server -f server.c -r TimesTen -s SERVICE
```

Note: The *install_dir* is the path to the TimesTen home directory.

Example 6–3 shows an example of 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 Example 6–2.

Example 6–3 Construct server modules

```
set CFLAGS=-IC:\TimesTen\giraffe\Include
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 Application Tuning

This chapter describes how to tune a C application to run optimally on a TimesTen data store. See "TimesTen Database Performance Tuning" in *Oracle TimesTen In-Memory Database Operations Guide* for more general tuning tips.

This chapter includes the following topics:

Bypass driver manager if appropriate

TimesTen permits ODBC applications that do not need some of the functionality provided by the driver manager to link without it. In particular, applications that do not need ODBC access to database systems other than TimesTen should consider omitting the driver manager. This is done by linking the application directly with the TimesTen Data Manager or client driver, as described in "Linking options" on page 2-13. The performance improvement will be approximately 20 percent.

"Testing link options" on page 2-4 explains how to determine whether an application is linked directly with the driver or with the driver manager.

Note: It is permissible for some applications connected to a data store to be linked with the driver manager, while others connected to the same data store are direct-linked.

Using 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 allows 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 and DELETE statements, but not with SELECT statements. TimesTen recommends the following batch sizes for Release 11.2.1:

- 256 for INSERT statements
- 31 for UPDATE statements
- 31 for DELETE statements

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	SQLROWSETSIZE	Number of values for each parameter.
pirow	SQLROWSETSIZE	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" on page 1-12.)

Refer to the TimesTen Quick Start demo source file `bulkinsert.c` for a complete working example of batching. (Also, for programming in C++ with TTClasses, see `bulktest.cpp`.)

Note: When using `SQLParamOptions` with the TimesTen Client/Server driver, data-at-execution parameters are not supported.

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`.

Note: A call to `SQLFreeStmt` with the SQL_UNBIND option unbinds all columns.

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 piece-wise fetches of data from long character or binary columns.

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 option, which can be set through `SQLSetStmtOption` and allows 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 all the application has received all the data, decreasing concurrency. For more information on how to use TT_PREFETCH_COUNT, see "Prefetching multiple rows of data" on page 1-20.

8

TimesTen Utility API

The TimesTen Utility Library C language functions documented in this chapter provide a programmable interface to some of the command line utilities documented in "Utilities" in *Oracle TimesTen In-Memory Database Reference*.

Applications that use this set of C language functions must include `ttutillib.h` and link with both the TimesTen Data Manager library (`libtten` on UNIX or `ttdv1121.lib` and `tten1121.lib` on Windows) and the TimesTen utility library (`libttutil` on UNIX and `ttutil1121.lib` on Windows platforms).

Important: 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 it is done with the TimesTen utility library interface

These functions are not supported with TimesTen Client or for Java applications. They are supported only for TimesTen Data Manager ODBC applications.

Return codes

Unless otherwise indicated, the utility functions return these codes as defined in `ttutillib.h`.

Code	Description
TTUTIL_SUCCESS	Returned upon success.
TTUTIL_ERROR	Returned if an error occurs.
TTUTIL_WARNING	Returned upon success, when a warning has been generated.
TTUTIL_INVALID_HANDLE	Returned if an invalid utility library handle is specified.

Note: The application must call the ttUtilGetError C function to retrieve all actual error or warning information.

ttBackup

Description

Creates either a full or an incremental backup copy of the data store specified by *`connStr`*. You can back up a data store either to a set of files or to a stream. You can restore the data store at a later time using either the ttRestore function or the `ttRestore` utility. If the data store is in use at the time of the backup, it must be in shared mode to successfully complete this operation.

For an overview of the TimesTen backup and restore facility, see "Copying, migrating, backing up and restoring a data store" in the *Oracle TimesTen In-Memory Database Operations Guide*.

Required privilege

Requires 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 *	A NULL-terminated string specifying a connection string that describes the data store to be backed up.

Parameter	Type	Description
type	ttBackupType	Specified the type of backup to be performed. Valid values are: 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 data store. 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 data store. Otherwise, it performs a full file backup of the data store 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. 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 a 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* then 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 will be destroyed only if it has been successfully superseded by the new backup. However, it does require enough disk 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 data store. For TT_BACKUP_INCREMENTAL_STOP, this parameter specifies the basename 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 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

To back up the data store 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 data store supports only eight incremental-enabled backups.

See also

ttRestore

ttDestroyDataStore

Description

Destroys a data store including all checkpoint files, transaction logs and daemon catalog entries corresponding to the data store specified by the connection string. It does not delete the DSN itself defined in the `odbc.ini` file on the supported UNIX platforms or in Windows registry on the supported Windows platforms.

Required privilege

Requires 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 *	A NULL-terminated string specifying the connection string that describes the data store to be destroyed. All data store attributes in this connection string, except DSN and `DataStore` attributes, are ignored.
timeout	unsigned int	Specifies the number of times to retry before returning to the caller. `ttDestroyDataStore` continually retries the destroy operation every 1 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 data store is in use. No retry is performed if this parameter is 0.

Example

To destroy a data store 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*, call:

```
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 data store including all checkpoint files, transaction logs and daemon catalog entries corresponding to the data store specified by the connection string. It does not delete the DSN itself defined in the `odbc.ini` file on the supported UNIX platforms or in the Windows registry on supported Windows platforms.

Required privilege

Requires 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 *	A NULL-terminated string specifying the connection string that describes the data store to be destroyed. All data store attributes in this connection string except DSN and `DataStore` attributes are ignored.
timeout	unsigned int	Specifies the number of seconds to retry before returning to the caller. `ttDestroyDataStoreForce` continually retries the destroy operation every 1 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 data store is in use. No retry is performed if this parameter is 0.

Example

To destroy a data store 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*, call:

```
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 data store specified by the connection string is kept in RAM by TimesTen after the last application disconnects from the data store. TimesTen then unloads the data store. The 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

Requires 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 *	A NULL-terminated string specifying a connection string that describes the data store for which the RAM Grace period is set.
seconds	unsigned int	Specifies the number of seconds TimesTen keeps the data store in RAM after the last application disconnects from the data store. TimesTen then unloads the data store.

Example

To set 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 data store specified by the connection string into the system RAM. For a permanent data store, a call to `ttRamLoad` is valid only when `RamPolicy` is set to TT_RAMPOL_MANUAL. For a temporary data store, a call to `ttRamLoad` loads the data store into RAM.

Required privilege

Requires 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 *	A NULL-terminated string specifying a connection string that describes the data store for which the data store is to be loaded into RAM.

Example

To load the data store 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 data store into the system RAM for the data store specified by the connection string.

Required privilege

Requires 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 *	A NULL-terminated string specifying a connection string that describes the data store for which the RAM policy is to be set.
policy	ttRamPolicyType	Specifies the policy used to determine when TimesTen loads the data store into system RAM for the specified data store. Valid values are the following: ■ TT_RAMPOL_ALWAYS: Specifies that the data store should always remain in RAM. ■ TT_RAMPOL_MANUAL: Specifies that the data store can be loaded into RAM explicitly using either ttRamLoad C function or the `ttAdmin -ramLoad` command. Similarly, the data store can be unloaded from RAM explicitly by using ttRamUnload C function or using `ttAdmin -ramUnload` command. ■ TT_RAMPOL_INUSE: Specifies that the data store is to be loaded into RAM when an application wants to connect to the data store. 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 data store, the default RAM policy is TT_RAMPOL_INUSE.

Example

To set the RAM policy to manual for the `payroll` DSN:

```
ttUtilHandle   utilHandle;
int            rc;
rc = ttRamPolicy (utilHandle, "DSN=payroll", TT_RAMPOL_MANUAL);
```

Note

The policy cannot be set for a temporary data store.

See also

ttRamGrace
ttRamLoad
ttRamUnload

ttRamUnload

Description

Causes TimesTen to unload the data store specified by the connection string from the system RAM if the `RamPolicy` attribute is set to "manual". For a permanent data store, this call is valid only when RAM policy is set to TT_RAMPOL_MANUAL. For a temporary data store, a call to `ttRamUnload` always tries to unload the data store from RAM because RAM policy cannot be set for such a data store.

Required privilege

Requires 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 *	A NULL-terminated string specifying a connection string for which the data store is to be unloaded from RAM.

Example

To unload the data store from RAM for the `payroll` DSN:

```
ttUtilHandle   utilHandle;
int            rc;
rc = ttRamUnload (utilHandle, "DSN=payroll");
```

Notes

When using this function with a temporary data store, TimesTen always attempts to unload the data store.

See also

ttRamGrace
ttRamLoad
ttRamPolicy

ttRepDuplicateEx

Description

Creates a replica of a remote data store on the local system. The process is initiated from the receiving local system. From there, a connection is made to the remote source data store to perform the duplicate operation.

Note: If the data store 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, and TT_REPDUP_DEFERCACHEUPDATE flag values.

Required privilege

Requires an instance administrator on the receiving local instance (where `ttRepDuplicateEx` is called) and a user with ADMIN privilege on the remote source data store. Create the internal user on the remote source data 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 instance must be the same as the operating system user name of the instance administrator on the remote source data store.
- When `ttRepDuplicateEx` is called, the `uid` and `pwd` Struct elements must specify the user name and password of the user with ADMIN privilege on the remote source data store. This user name is used to connect to the remove source data store to perform the duplicate operation.

Syntax

```
ttRepDuplicateEx (ttUtilHandle handle,
                  const char *destConnStr,
                  const char *srcDatastore,
                  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
      /*new struct elements can only be added here at the end */
} ttRepDuplicateExArg
```

Parameters

Parameter	Type	Description
handle	ttUtilHandle	Specifies the TimesTen utility library environment handle allocated using ttUtilAllocEnv.
destConnStr	const char *	A NULL-terminated string specifying the connection string for a local data store into which the replica of the remote data store is created.
srcDatastore	const char *	A NULL-terminated string specifying the remote source data store name. This name is the last component of the data store path name.
remoteHost	const char *	A NULL-terminated string specifying the TCP/IP host name of the system where the remote source data store is located.
arg	ttRepDuplicateExArg*	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 `ttRepDuplicateEx` argument structure contains these elements:

Element	Type	Description
size	unsigned int	Must be set up to *sizeof* (`ttRepDuplicateExArg`).
flags	unsigned int	The bit-wise union of values chosen from the list in the table of flag values.
uid	const char *	The user name of a user on the remote source data store with ADMIN privileges. This user name is used to connect to the remote source data store to perform the duplicate operation.
pwd	const char *	The password associated with the `uid`.
pwdcrypt	const char *	The encrypted password associated with the user ID.
cacheuid	const char *	Cache administration user ID.
cachepwd	const char *	Cache administration user password.
localHost	const char *	A NULL-terminated string specifying the TCP/IP host name of the local system. *localHost* is ignored if *remoteRepStart* is TT_FALSE. This explicitly identifies the local host. This parameter can be NULL. This is useful if the local host uses a nonstandard name, such as an IP address.
truncListLen	int	The number of elements in the *truncList*.
truncList	const char**	A list of non-replicated tables to truncate after duplicate.

Element	Type	Description
dropListLen	int	The number of elements in *dropList*.
dropList	const char**	A list of non-replicated tables to drop after the duplicate operation.
maxkbytesPerSec	int	Setting *maxkbytesPerSec* to a nonzero value specifies that the duplicate operation should not put more than *maxkbytesPerSec* kilobytes of data per second onto the network. Setting *maxkbytesPerSec* to 0 or a negative number indicates that the duplicate operation should not attempt to limit its bandwidth.
remoteDaemonPort	int	Specifies the remote daemon port. Setting *remoteDaemonPort* to 0 results in the daemon port number for the target data store being set to the port number used for the daemon on the source data store. This option cannot be used in duplicate operations for data stores with automatic port configuration.

The `ttRepDuplicateExArg` flags element is constructed from these values:

Value	Description
TT_REPDUP_NOFLAGS	No flags.
TT_REPDUP_COMPRESS	Enables compression of the data transmitted over the network for the duplicate operation.
TT_REPDUP_REPSTART	`ttRepDuplicateEx` sets the replication state (with respect to the local data store) in the remote data store to the start state before the remote data store is copied across the network. This ensures that all updates made after the duplicate operation are replicated from the remote data store to the newly created or restored local data store.
TT_REPDUP_RAMLOAD	Keeps the data store in memory upon completion of the duplicate operation. It changes the RAM policy for the data store to "manual".
TT_REPDUP_DELXLA	`ttRepDuplicateEx` removes 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 includes 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 will be 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 of the incremental cache groups are not replicated or if `ttRepDuplicateEx` is being used for rolling out a new or modified replication scheme to a node.

Example

To create a replica of a remote TimesTen DSN, `remote_payroll`, whose data store path name is `C:\dsns\payroll` to a local DSN, `local_payroll`, use the following:

```
ttUtilHandle utilHandle;
int rc;
ttRepDuplicateExArg arg;

memset(&arg, 0, sizeof(arg));
arg.size = sizeof(ttRepDuplicateExArg);
arg.flags = TT_REPDUP_REPSTART | TT_REPDUP_DELXLA;
arg.localHost = "mylocalhost";
arg.uid="myuid";
arg.pwd="mypwd";
rc=ttRepDuplicateEx(utilHandle,"DSN=local_payroll","payroll","remotehost", &arg);
```

See also

These 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 data store specified by the connection string from a backup that has been created using the ttBackup C function or `ttBackup` utility. If the data store already exists, `ttRestore` will not overwrite it.

For an overview of the TimesTen backup and restore facility, see "Copying, migrating, backing up and restoring a data store" in *Oracle TimesTen In-Memory Database Operations Guide*.

Required privilege

Requires 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 *	A NULL-terminated string specifying a connection string that describes the data store to be restored.
`type`	ttRestoreType	Indicates whether the data store is to be restored from a file or a stream backup. Valid values are the following: ■ TT_RESTORE_FILE: The data store is to be restored from a file backup located at the backup path specified by the *`backupDir`* and *`baseName`* parameters. ■ TT_RESTORE_STREAM: The data store 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 data store's ODBC definition. For TT_RESTORE_STREAM, this parameter is ignored.

Parameter	Type	Description
stream	ttUtFileHandle	For TT_RESTORE_STREAM, specifies the stream from which the backup is to be read. On 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.
flags	unsigned int	Reserved for future use. Specify 0.

Example

To restore the data store for the `payroll` DSN from `C:\backup`, use:

```
ttUtilHandle   utilHandle;
int            rc;

rc = ttRestore (utilHandle, "DSN=payroll", TT_RESTORE_FILE,
              "c:\\backup", NULL, TTUTIL_INVALID_FILE_HANDLE, 0);
```

See also

ttBackup

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 must call ttUtilFreeEnv when it is done with 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 *	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 *	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

`ttUtilAllocEnv` returns these codes 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 must call `ttUtilFreeEnv` when it is done with 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.
errBuff	char *	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 *	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

`ttUtilFreeEnv` returns these 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

To free 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: ▪ 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. Valid return values are: ▪ TTUTIL_ERROR ▪ TTUTIL_WARNING
errBuff	char *	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

`ttUtilGetError` returns these 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

To retrieve 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 `ttUtilGetErrorCount` function returns these 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

To retrieve 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 rolling back thread.

Required privilege

Requires 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**	The connection string of the data store, which contains the transaction to be rolled back.
xactId	const char*	The transaction ID for the transaction to be rolled back.

Example

To roll back a transaction with the ID `3.4567`, in the data store 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) described in "XLA and TimesTen Event Management" on page 5-1. It includes the following topics:

- About XLA functions
- Summary of XLA functions by category
- XLA function reference
- C data structures used by XLA

About XLA functions

This section includes general information about XLA functions.

About return codes

All of the XLA API functions described in this chapter return a value of type SQLRETURN, which is defined by ODBC to have one of the following values:

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_NO_DATA_FOUND
- SQL_ERROR

See "Handling XLA errors" on page 5-28 for information on handling XLA errors.

About parameter types (input, output, input-output)

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.

About results output by functions

Most routines in this API copy results to application buffers. Those few routines that produce pointers to buffers containing results are guaranteed to remain 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.

  ttXlaConfigBuffer, or ttXlaAcknowledge (in persistent mode). If the application must retain access to the buffers for a longer time, the application 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 both 32- and 64-bit platforms. The types SQLUINTEGER and SQLUBIGINT are used to refer to 32- and 64-bit integers unambiguously. Issues of alignment and padding are addressed by filling the type definition so that each SQLUINTEGER value is on a 4-byte boundary and each SQLUBIGINT value is on an 8-byte boundary. For a description of storage requirements for other TimesTen data types, see "Understanding rows" in *Oracle TimesTen In-Memory Database Operations Guide*.

About required privileges

"Access control impact on XLA" on page 5-8 introduces the effects of TimesTen access control features on XLA functionality. Any XLA functionality requires the system privilege XLA.

Summary of XLA functions by category

As described in Chapter 5, "XLA and TimesTen Event Management", TimesTen XLA can be used to detect updates on a data store or as a toolkit to build your own replication solution. You can initialize XLA in either *persistent* or *non-persistent* mode.

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 including data type conversion functions
- XLA persistent mode functions
- XLA non-persistent mode functions
- XLA replication functions

XLA core functions including data type conversion functions

The following table lists core XLA functions that can be used by any XLA application:

Function	Description
ttXlaClose	Closes the XLA handle opened by ttXlaPersistOpen.
ttXlaConvertCharType	Converts column data into the connection character set.
ttXlaError	Retrieves error information.
ttXlaErrorRestart	Resets error stack information.
ttXlaGetColumnInfo	Retrieves information about all the columns in the table.
ttXlaGetTableInfo	Retrieves information about a table.

Function	Description
ttXlaGetVersion	Retrieves the current version of XLA.
ttXlaNextUpdate	Retrieves a batch of updates from TimesTen.
ttXlaNextUpdateWait	Retrieves a batch of updates from TimesTen. Will wait for a specified time if no updates are available in the transaction log.
ttXlaTableByName	Finds the system and user table identifiers for a table given the table's owner and name.
ttXlaTableStatus	Sets and retrieves XLA status for a table.
ttXlaSetVersion	Sets the XLA version to be used.
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.

See "Writing an XLA event-handler application" on page 5-9 for a discussion on how to use most of these functions.

The following table lists data type conversion functions that can be used by any XLA application:

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.

Function	Description
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.

For more information about XLA data types, see "About XLA data types" on page 5-6.

XLA persistent mode functions

The following table lists the functions that are exclusive to operating XLA in persistent mode:

Function	Description
ttXlaPersistOpen	Initializes a handle to a TimesTen data store to access the transaction log in persistent mode.
ttXlaAcknowledge	Acknowledges receipt of one or more transaction update records from the transaction log.
ttXlaDeleteBookmark	Deletes a transaction log bookmark.
ttXlaGetLSN	Retrieves the log record identifier of the current bookmark for a data store.
ttXlaSetLSN	Sets the log record identifier of the current bookmark for a data store.

See "Writing an XLA event-handler application" on page 5-9 for a discussion on how to use these functions.

XLA non-persistent mode functions

Note: TimesTen recommends using XLA in persistent mode.

The following table lists the functions that are exclusive to operating XLA in non-persistent mode:

Function	Description
ttXlaOpenTimesTen	Initializes a handle to a TimesTen data store to access the transaction log in non-persistent mode.
ttXlaConfigBuffer	Sets the size of the XLA staging buffer.
ttXlaStatus	Retrieves the current XLA status.
ttXlaResetStatus	Resets all the XLA statistics counters.

XLA replication functions

The following table lists the functions that are exclusive to using XLA as a replication mechanism include the following.

Function	Description
ttXlaApply	Applies the update to the data store or database associated with the XLA handle.
ttXlaTableCheck	Verifies that the named table in the table description received from the sending data store is compatible with the receiving data store.
ttXlaLookup	Looks for an update record for a table with a specific key value.
ttXlaRollback	Rolls back a transaction.
ttXlaCommit	Commits a transaction.
ttXlaGenerateSQL	Generates a SQL statement that expresses the effect of an update record.

See "Using XLA as a replication mechanism" on page 5-33 for a discussion on how to use these functions.

XLA function reference

This section provides reference information for each XLA function. Functions are listed in alphabetical order.

ttXlaAcknowledge

Description

This function is used in persistent mode 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.

Notes:

- 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 in the absence of any 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" on page 5-12 for a discussion about using this function.

Required privilege

Requires the system privilege XLA.

Syntax

```
SQLRETURN ttXlaAcknowledge(ttXlaHandle_h handle)
```

Parameters

Parameter	Type	Description
handle	ttXlaHandle_h	The transaction log handle.

Returns

SQL_SUCCESS if call is successful. Otherwise, use ttXlaError to report the error.

Example

```
rc = ttXlaAcknowledge(xlahandle);
```

See also

ttXlaNextUpdate
ttXlaNextUpdateWait

ttXlaApply

Description

Applies an update to the data store or database associated with the *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 data store or database transaction is committed. No other transaction commits are performed by `ttXlaApply`. If the parameter *test* is true, the "old values" in the update description are compared against the current contents of the data store for record updates and deletions. If the old value in the update description does not match the corresponding row in the data store, this function rejects the update and returns an `sb_ErrXlaTupleMismatch` error.

See "Using XLA as a replication mechanism" on page 5-33 for a discussion about using this function.

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

Requires the system 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	The transaction log handle for the data store.
record	ttXlaUpdateDesc_t *	Transaction to generate SQL statement.
test	SQLINTEGER	Test for old values (0 = test off; 1 = test on).

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 data store 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

ttXlaClose

Description

Closes an XLA *handle* that was opened by ttXlaPersistOpen. See "Terminating an XLA application" on page 5-31 for a discussion about using this function.

Required privilege

Requires the system privilege XLA.

Syntax

```
SQLRETURN ttXlaClose(ttXlaHandle_h handle)
```

Parameters

Parameter	Type	Description
handle	ttXlaHandle_h	The ODBC handle for the data store.

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 call:

```
rc = ttXlaClose(xlahandle);
```

See also

ttXlaPersistOpen

ttXlaCommit

Description

Commits the current transaction being applied on the *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. ttXlaApply applies the current transaction if it does not encounter an error.

See "Handling timeout and deadlock errors" on page 5-36 for a discussion about using this function.

Required privilege

Requires the system privilege XLA.

Syntax

```
SQLRETURN ttXlaCommit(ttXlaHandle_h handle)
```

Parameters

Parameter	Type	Description
handle	ttXlaHandle_h	The transaction log handle for the data store.

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

ttXlaConfigBuffer

Description

This function is valid only when XLA is in non-persistent mode.

You can use the `ttXlaConfigBuffer` function to both set and get the size of the XLA staging buffer, which is where XLA stages the update records obtained from the transaction log and makes them available to be read by the application.

To first set the size of the staging buffer, specify a value for the *`newSize`* parameter and a null value for the *`oldSize`* parameter. The new size of the staging buffer is retrieved from *`*newSize`*. A size of zero indicates no staging buffer should be allocated.

To change the size of the staging buffer, specify a value for *`newSize`* and provide an *`oldSize`* parameter. Upon return, *`*oldSize`* contains the previous size of the staging buffer, or 0 if the size had not been set.

To retrieve but not change the current size of the staging buffer, specify a null value for the *`newSize`* parameter. The current size of the staging buffer is returned in *`*oldSize`*.

When choosing the size of your staging buffer, consider that if the buffer is too small, TimesTen updates will exhaust the buffer, causing further updates to be rejected. Conversely, over-allocating space for the buffer wastes memory.

After setting the size of your staging buffer, you can resize it at any time. However, resizing may result in copying the current buffer and therefore incurring substantial performance penalties.

Changes to the staging buffer size are carried out immediately. When the buffer is resized, records that were returned by previous calls to ttXlaNextUpdate or ttXlaNextUpdateWait become invalid.

Only one buffer may be configured for a data store. When the buffer is resized, values returned by previous calls on ttXlaNextUpdate become invalid.

Notes:

- If the XLA staging buffer is set to a nonzero size and no XLA reader is connected, updates on the data store will be written into the buffer. When the staging buffer becomes full, database operations cannot successfully complete until you either delete the staging buffer (size set to 0) or connect an XLA reader and begin reading from the buffer.
- If a smaller size is specified for the staging buffer and the current contents will not fit in the smaller size, the buffer size is not changed and an error is returned.

Required privilege

Requires the system privilege XLA.

Syntax

```
SQLRETURN ttXlaConfigBuffer(ttXlaHandle_h handle,
                            out SQLUBIGINT *oldSize,
                            SQLUBIGINT *newSize)
```

Parameters

Parameter	Type	Description
handle	ttXlaHandle_h	The transaction log handle for the data store.
oldSize	out SQLUBIGINT *	Current size of the staging buffer.
newSize	SQLUBIGINT *	New size of the staging buffer.

Returns

SQL_SUCCESS if call is successful. Otherwise, use ttXlaError to report the error.

Example

Assume the following declarations for our examples:

```
SQLUBIGINT currentSize, requestedSize;
```

To find the current size of the staging buffer without changing the size:

```
rc = ttXlaConfigBuffer(xlahandle, &currentSize, NULL);
```

To set the size of the staging buffer to 400,000 bytes:

```
requestedSize = 400000;
...
rc = ttXlaConfigBuffer(xlahandle, NULL, &requestedSize);
```

To change the size of the staging buffer to 400,000 bytes and retrieve the previous size:

```
requestedSize = 400000;
...
rc = ttXlaConfigBuffer(xla_handle, &currentSize, &requestedSize);
```

To delete the staging buffer:

```
requestedSize = 0;
...
rc = ttXlaConfigBuffer(xlahandle, NULL, &requestedSize);
```

Note

Buffer resizing may copy the current buffer and therefore incur substantial performance penalties. If a smaller size is specified for the staging buffer and the current contents will not fit in the smaller size, the staging buffer size is not changed and an error is returned.

See also

ttXlaOpenTimesTen
ttXlaStatus
ttXlaResetStatus

ttXlaConvertCharType

Description

Converts the column data indicated by *colinfo* and *tup* into the connection character set associated with *handle* and places the result in *buf*.

Required privilege

Requires the system 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	The transaction log handle for the data store.
colinfo	ttXlaColDesc_t *	A pointer to the buffer that holds the column descriptions.
tup	void *	The data that is 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

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" on page 5-22 for a discussion about using this function.

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

Requires the system 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	out DATE_STRUCT *	Pointer to storage allocated to hold the converted date.

Returns

SQL_SUCCESS if call is successful. Otherwise, use ttXlaError to report the error.

ttXlaDecimalToCString

Description

Converts a TTXLA_DECIMAL_TT value to a string usable by applications. 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` is the same as `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" on page 5-22 for a discussion about using this function.

Required privilege

Requires the system 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	out char *	Pointer to storage allocated to hold the converted string.
precision	SQLSMALLINT	If `fromData` is larger than 1, precision is the maximum number of digits before and after the decimal point. If `fromData` is less than 1, precision is the same as scale.
scale	SQLSMALLINT	Maximum number of digits after the decimal point.

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 *`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 data store handle and the effect is the same as having opened the persistent 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" on page 5-30 for a discussion about using this function.

Notes:

- 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

Requires the system privilege XLA.

Syntax

```
SQLRETURN ttXlaDeleteBookmark(ttXlaHandle_h handle)
```

Parameters

Parameter	Type	Description
`handle`	ttXlaHandle_h	The transaction log handle.

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 the error(s) encountered from the previous call on the given `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 "Handling XLA errors" on page 5-28 for a discussion about using this function.

Required privilege

Requires the system 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	The transaction log handle for the data store.
errCode	out SQLINTEGER *	The code of the error message to be copied into the *errMessage* buffer.
errMessage	out char *	Buffer to hold the error text.
maxLen	SQLINTEGER	The maximum length of the *errMessage* buffer.
retLen	out SQLINTEGER *	The actual size of the error message.

Returns

SQL_SUCCESS if error information is returned and 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.

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 "Handling XLA errors" on page 5-28 for a discussion about using this function.

Required privilege

Requires the system privilege XLA.

Syntax

```
SQLRETURN ttXlaErrorRestart(ttXlaHandle_h handle)
```

Parameters

Parameter	Type	Description
handle	ttXlaHandle_h	The transaction log handle for the data store.

Returns

SQL_SUCCESS if call is successful. Otherwise, use ttXlaError to report the error.

Example

```
rc = ttXlaErrorRestart(xlahandle);
```

See also

ttXlaError

ttXlaGenerateSQL

Description

Generates a SQL DML or DDL statement that expresses the effect of the update `record`. The generated statement is not applied to any data store or 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 data store" on page 5-37.

Required privilege

Requires the system 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	The transaction log handle for the data store.
record	ttXlaUpdateDesc_t *	The record to be translated into SQL.
buffer	out char *	Location of the translated SQL statement.
maxLen	SQLINTEGER	The maximum length of the buffer, in bytes.
actualLen	out SQLINTEGER *	The actual length of the buffer, in bytes.

Returns

SQL_SUCCESS if call is successful. Otherwise, use ttXlaError to report the error.

Example

This example generates the text of an 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

ttXlaGetColumnInfo

Description

Retrieves information about all the columns in the table. Normally, `*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's definition.

See "Obtaining column descriptions" on page 5-17 for a discussion about using this function.

Required privilege

Requires the system 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	The transaction log handle for the data store.
systemTableID	SQLUBIGINT	System ID of table.
userTableID	SQLUBIGINT	User ID of table.
colinfo	out ttXlaColDesc_t *	A pointer to the buffer large enough to hold a separate description for *maxcols* columns.
maxcols	SQLINTEGER	The 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	out SQLINTEGER *	The number of columns returned.

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" on page 9-81 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 `handle`. See "How bookmarks work" on page 5-4 for a discussion about using this function.

Required privilege

Requires the system privilege XLA.

Syntax

```
SQLRETURN ttXlaGetLSN(ttXlaHandle_h handle,
                      out tt_XlaLsn_t *LSN)
```

Parameters

Parameter	Type	Description
handle	ttXlaHandle_h	The transaction log handle for the data store.
LSN	out tt_XlaLsn_t *	The 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" on page 9-84.

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; see the description of the ttXlaTblDesc_t data type. If `userTableID` 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

Requires the system privilege XLA.

Syntax

```
SQLRETURN ttXlaGetTableInfo(ttXlaHandle_h handle,
                            SQLUBIGINT systemTableID,
                            SQLUBIGINT userTableID,
                            out ttXlaTblDesc_t *tblinfo)
```

Parameters

Parameter	Type	Description
handle	ttXlaHandle_h	The transaction log handle for the data store.
systemTableID	SQLUBIGINT	System table ID.
userTableID	SQLUBIGINT	User table ID.
tblinfo	out ttXlaTblDesc_t *	Row information.

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 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.

`ttXlaGetVersion` 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.

See "XLA persistent mode" on page 5-2 for a discussion about using this function.

Required privilege

Requires the system privilege XLA.

Syntax

```
SQLRETURN ttXlaGetVersion(ttXlaHandle_h handle,
                          out ttXlaVersion_t *configuredVersion,
                          out ttXlaVersion_t *actualVersion)
```

Parameters

Parameter	Type	Description
handle	ttXlaHandle_h	The transaction log handle for the data store.
configuredVersion	out ttXlaVersion_t *	The configured version of XLA.
actualVersion	out ttXlaVersion_t *	The actual version of XLA.

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 call:

```
rc = ttXlaGetVersion(xlahandle, &configured, &actual);
```

See also

ttXlaVersionCompare
ttXlaSetVersion

ttXlaLookup

Description

This function looks for a record in the given table with the key values specified in the *keys* parameter. The formats of the *keys* and *result* record are the same as for ordinary rows. This function requires a primary key on the underlying table.

Required privilege

Requires the system 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	The transaction log handle for the data store.
table	ttXlaTblDesc_t *	The 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	out void *	The located record is copied into the result. If no record exists with the matching key columns, an error is returned.
maxsize	SQLINTEGER	The size of the largest record that can fit into the result buffer.
retsize	out SQLINTEGER *	The actual size of the record.

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

ttXlaNextUpdate

Description

This function fetches up to *maxrecords* update records from the transaction log and returns the records associated with committed transactions to the *records* buffer. The actual number of returned records is reported in the *nreturned* output parameter. This function requires a bookmark to be present in the data store and to be associated with the connection used by the function.

When operating the transaction log in persistent mode, 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" on page 5-12 for a discussion about using this function.

Required privilege

Requires the system privilege XLA.

Syntax

```
SQLRETURN ttXlaNextUpdate(ttXlaHandle_h handle,
                          out ttXlaUpdateDesc_t ***records,
                          SQLINTEGER maxrecords,
                          out SQLINTEGER *nreturned)
```

Parameters

Parameter	Type	Description
handle	ttXlaHandle_h	The transaction log handle for the data store.
records	out ttXlaUpdateDesc_t ***	The buffer to hold the completed transaction records.
maxrecords	SQLINTEGER	The maximum number of records to be fetched.
nreturned	out SQLINTEGER *	The actual number of returned records. 0 is returned if no update data is available.

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 a table's tracking status.

See also

ttXlaNextUpdateWait
ttXlaAcknowledge

ttXlaNextUpdateWait

Description

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" on page 5-12.

Required privilege

Requires the system 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	The transaction log handle for the data store.
records	out ttXlaUpdateDesc_t ***	The buffer to hold the completed transaction records.
maxrecords	SQLINTEGER	The maximum number of records to be fetched. **Note**: The largest effective value is 1000 records.
nreturned	out SQLINTEGER *	The actual number of records returned. 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

SQL_SUCCESS if call is successful. Otherwise, use ttXlaError to report the error.

Example

This example retrieves up to 100 records and will wait 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

Requires the system 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

SQL_SUCCESS if successful. Otherwise, use ttXlaError to report an 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

Requires the system 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 *	If *buflen* >= *reslen*, then *reslen* is the number of bytes that were written. If *buflen* <*reslen*, then *reslen* is the number of bytes that would have been written if the buffer had been large enough.

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

Requires the system 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

SQL_SUCCESS if call is successful. Otherwise, use ttXlaError to report an 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

Requires the system 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

SQL_SUCCESS if call is successful. Otherwise, use ttXlaError to report an 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

Requires the system 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

SQL_SUCCESS if call is successful. Otherwise, use ttXlaError to report an 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

Requires the system 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

SQL_SUCCESS if call is successful. Otherwise, use ttXlaError to report an 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

Requires the system 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

SQL_SUCCESS if call is successful. Otherwise, use ttXlaError to report an error.

ttXlaOpenTimesTen

Description

Initializes a transaction log handle to a TimesTen data store to enable access to the transaction log in non-persistent mode. The *hdbc* parameter is an ODBC connection handle to a TimesTen data store that will be used to apply updates. Do not issue any other ODBC calls against this connection until it is closed by `ttXlaClose`. The *handle* parameter is initialized by this call and must be provided on each subsequent call that applies updates.

In non-persistent mode, only one application can read from the transaction log at any point in time. See "Initializing XLA in non-persistent mode" on page 5-39 for related discussion.

Note: Most applications should use ttXlaPersistOpen to initialize XLA in persistent mode.

Required privilege

Requires the system privilege XLA.

Syntax

```
SQLRETURN ttXlaOpenTimesTen(SQLHDBC hdbc,
                            out ttXlaHandle_h *handle)
```

Parameters

Parameter	Type	Description
hdbc	SQLHDBC	The ODBC handle for the data store.
handle	out ttXlaHandle_h *	The transaction log handle for the data store.

Returns

SQL_SUCCESS if call is successful. Otherwise, use ttXlaError to report the error.

The following example opens a transaction log in non-persistent mode and returns a handle named `xlahandle` for the ODBC connection:

```
SQLHDBC hdbc;
ttXlaHandle_h xlahandle;
rc = ttXlaOpenTimesTen(hdbc, &xlahandle);
```

Note

Use of multiple threads over the same XLA handle is not recommended by TimesTen. Multithreaded applications should use ttXlaPersistOpen to 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

ttXlaConfigBuffer
ttXlaStatus

ttXlaResetStatus
ttXlaClose

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

Requires the system 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 *	An ODBC timestamp value converted from the XLA Oracle DATE value.

Returns

SQL_SUCCESS if call is successful. Otherwise, use ttXlaError to report an 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

Requires the system privilege XLA.

Parameters

Parameter	Type	Description
fromData	void *	Pointer to the number value returned from the transaction log.
returnData	TIMESTAMP_STRUCT *	An ODBC timestamp value converted from the XLA Oracle TIMESTAMP value.

Returns

SQL_SUCCESS if call is successful. Otherwise, use ttXlaError to report an error.

ttXlaPersistOpen

Description

Initializes a transaction log handle to a TimesTen data store to enable access to the transaction log in persistent mode. The *hdbc* parameter is an ODBC connection handle to a TimesTen data store. 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 persistent bookmark (see "About XLA bookmarks" on page 5-4). The *tag* can identify a new bookmark, either non-replicated or replicated, or one that already 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

In persistent mode, multiple applications can concurrently read from the transaction log. See "Initializing XLA and obtaining an XLA handle" on page 5-10 for a discussion about using this function.

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.

Required privilege

Requires the system privilege XLA.

Syntax

```
SQLRETURN ttXlaPersistOpen(SQLHDBC hdbc,
                           SQLCHAR * tag,
                           SQLUINTEGER options,
                           out ttXlaHandle_h * handle)
```

Parameters

Parameter	Type	Description
hdbc	SQLHDBC	The ODBC handle for the data store.
tag	SQLCHAR *	The identifier for the persistent bookmark. Can be null, in which case options should be set to XLANONE. Maximum allowed length is 31.

Parameter	Type	Description
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	out ttXlaHandle_h *	The transaction log handle returned by this call. Space is allocated by this call. User should call ttXlaClose to free space.

Returns

SQL_SUCCESS if call is successful. Otherwise, use ttXlaError to report the error.

Example

This example opens a transaction log in persistent mode, 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

ttXlaResetStatus

Description

This function is valid only when XLA is in non-persistent mode.

Resets all the XLA status counters reported in the ttXlaStatus_t structure returned by ttXlaStatus. Currently, only the *xlabufminfree* value is reset.

Required privilege

Requires the system privilege XLA.

Syntax

```
SQLRETURN ttXlaResetStatus(ttXlaHandle_h handle)
```

Parameters

Parameter	Type	Description
handle	ttXlaHandle_h	The transaction log handle for the data store.

Returns

SQL_SUCCESS if call is successful. Otherwise, use ttXlaError to report the error.

Example

The following example resets the XLA status counters:

```
rc = ttXlaResetStatus(xlahandle);
```

See also

ttXlaOpenTimesTen
ttXlaConfigBuffer
ttXlaStatus

ttXlaRollback

Description

Rolls back the current transaction being applied on the *handle*. You can call this routine to respond to transient errors (timeout or deadlock) reported by ttXlaApply.

See "Handling timeout and deadlock errors" on page 5-36 for a discussion about using this function.

Required privilege

Requires the system privilege XLA.

Syntax

```
SQLRETURN ttXlaRollback(ttXlaHandle_h handle)
```

Parameters

Parameter	Type	Description
handle	ttXlaHandle_h	The transaction log handle for the data store.

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

ttXlaRowidToCString

Description

Converts a ROWID value to a string value usable by applications.

Required privilege

Requires the system 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

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 data store specified by `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's Initial Read log record identifier.

See "About XLA bookmarks" on page 5-4 for a discussion about using this function.

Required privilege

Requires the system privilege XLA.

Syntax

```
SQLRETURN ttXlaSetLSN(ttXlaHandle_h handle,
                      tt_XlaLsn_t *LSN)
```

Parameters

Parameter	Type	Description
handle	ttXlaHandle_h	The transaction log handle for the data store.
LSN	tt_XlaLsn_t *	The 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" on page 9-84.

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.

See "XLA persistent mode" on page 5-2 for a discussion about using this function.

Required privilege

Requires the system privilege XLA.

Syntax

```
SQLRETURN ttXlaSetVersion(ttXlaHandle_h handle,
                          ttXlaVersion_t *version)
```

Parameters

Parameter	Type	Description
handle	ttXlaHandle_h	The transaction log handle for the data store.
version	ttXlaVersion_t *	The desired version of XLA.

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 call:

```
rc = ttXlaSetVersion(xlahandle, &requestedVersion);
```

See also

ttXlaVersionCompare
ttXlaGetVersion

ttXlaStatus

Description

This function is valid only when operating XLA in non-persistent mode.

Retrieves status information on the transaction log buffer and your XLA staging buffer and stores it in the `*status` parameter, which is of data type ttXlaStatus_t. This data structure includes:

- The free and occupied space in the staging buffer
- The number of transactions and records in the staging buffer
- The free and occupied space in the transaction log buffer
- Whether the system is accepting new transaction updates

The `ttXlaStatus_t ->xlabufminfree` value is the minimum number of free bytes in the transaction log buffer and is a useful statistic if you decide to recalculate the optimum size of the staging buffer. As the transaction log buffer expands and contracts, *`xlabufminfree`* may no longer accurately reflect the minimum space. You can call ttXlaResetStatus, generally used to reset the value of the `ttXlaStatus_t ->xlabufminfree` field, to null *`xlabufminfree`*. Then, at some later time, you can call ttXlaStatus to obtain a new minimum value before calculating the optimum *`newSize`* value to pass to the ttXlaConfigBuffer function.

Required privilege

Requires the system privilege XLA.

Syntax

```
ttXlaStatus(ttXlaHandle_h handle, out ttXlaStatus_t *status)
```

Parameters

Parameter	Type	Description
handle	ttXlaHandle_h	The transaction log handle for the data store.
status	out ttXlaStatus_t *	The current XLA status.

Returns

SQL_SUCCESS if call is successful. Otherwise, use ttXlaError to report the error.

Example

This example gets the current XLA status:

```
ttXlaStatus_t s;
rc = ttXlaStatus(xlahandle, &s);
```

See also

ttXlaOpenTimesTen
ttXlaConfigBuffer
ttXlaResetStatus

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" on page 5-11 for a discussion about using this function.

Required privilege

Requires the system 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	The transaction log handle for the data store.
owner	char *	The owner for the table or view as a string.
name	char *	The name of the table or view.
sysTableID	out SQLUBIGINT *	Where the system table ID is returned.
userTableID	out SQLUBIGINT *	Where the user table ID is returned.

Returns

SQL_SUCCESS if call is successful. Otherwise, use ttXlaError to report the error.

Example

To get the system and user table ID's associated with the table `PURCHASING.INVOICES`, use the call:

```
SQLUBIGINT sysTableID;
SQLUBIGINT userTableID;

rc = ttXlaTableByName(xlahandle, "PURCHASING", "INVOICES",
                      &sysTableID, &userTableID);
```

See also

ttXlaTableStatus

ttXlaTableCheck

Description

When using XLA as a replication mechanism, this function verifies that the named table in the ttXlaTblDesc_t structure received from a master data store is compatible with a subscriber data store or database associated with the *handle*. The *compat* parameter indicates whether the tables are compatible.

See "Checking table compatibility between data stores" on page 5-33 for a discussion about using this function.

Required privilege

Requires the system privilege XLA.

Syntax

```
SQLRETURN ttXlaTableCheck(ttXlaHandle_h handle,
                          ttXlaTblDesc_t *table,
                          ttXlaColDesc_t *columns,
                          out SQLINTEGER *compat)
```

Parameters

Parameter	Type	Description
handle	ttXlaHandle_h	The transaction log handle for the data store.
table	ttXlaTblDesc_t *	A table description.
columns	ttXlaColDesc_t *	Column description for the table.
compat	out SQLINTEGER *	Returns compatibility information. ■ 1: Tables are compatible. ■ 0: Tables are not compatible.

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

ttXlaTableStatus

Description

Returns the update status for a table in `*oldstatus`. You 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 in persistent mode. `ttXlaTableStatus` 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, ADDCOLS, DRPCOLS, TRUNCATE, SETTBL1, and SETCOL1 transactions.

See "Specifying which tables to monitor for updates" on page 5-11 for a discussion about using this function.

Required privilege

Requires the system privilege XLA.

Syntax

```
SQLRETURN ttXlaTableStatus(ttXlaHandle_h handle,
                           SQLUBIGINT systemTableID,
                           SQLUBIGINT userTableID,
                           out SQLINTEGER *oldstatus,
                           SQLINTEGER *newstatus)
```

Parameters

Parameter	Type	Description
handle	ttXlaHandle_h	The transaction log handle for the data store.
systemTableID	SQLUBIGINT	System ID of table.
userTableID	SQLUBIGINT	User ID of table.
oldstatus	out SQLINTEGER *	XLA status: 1 = On or 0 = Off.
newstatus	SQLINTEGER *	XLA status: 1 = On or 0 = Off.

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 call:

```
/* Get system table identifier into systemTableID, then ... */

rc = ttXlaTableStatus(xlahandle, systemTableID, 0,
                      &currentStatus, NULL);
```

The *currentStatus* value will be 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 this 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 by means of 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

ttXlaTimeToODBCCType

Description

Converts a TTXLA_TIME value to an ODBC C value usable by applications. See "Converting complex data types" on page 5-22 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

Requires the system 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	out TIME_STRUCT *	Pointer to storage allocated to hold the converted time.

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" on page 5-22 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

Requires the system 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	out TIMESTAMP_ STRUCT *	Pointer to storage allocated to hold the converted timestamp.

Returns

SQL_SUCCESS if 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);
```

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

Requires the system privilege XLA.

Syntax

```
SQLRETURN ttXlaTableVersionVerify(ttXlaHandle_h handle
                                  ttXlaTblVerDesc_t *table,
                                  ttXlaUpdateDesc_t *record
                                  out SQLINTEGER *compat)
```

Returns

SQL_SUCCESS if cached table definition is compatible with the XLA record being processed. Otherwise, use ttXlaError to report the error.

Parameters

Parameter	Type	Description
handle	ttXlaHandle_h	The transaction log handle for the data store.
table	ttXlaTblVerDesc_t *	A cached table description.
record	ttXlaUpdateDesc_t *	The XLA record that must be processed.
compat	out SQLINTEGER *	Returns compatibility information. ■ 1: Tables are compatible. ■ 0: Tables are not compatible.

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

ttXlaVersionColumnInfo

Description

Retrieves information about the columns in a table for which a change update XLA record must be processed.

Required privilege

Requires the system privilege XLA.

Syntax

```
SQLRETURN ttXlaVersionColumnInfo(ttXlaHandle_h handle,
                                 ttXlaUpdateDesc_t *record,
                                 out ttXlaColDesc_t *colinfo,
                                 SQLINTEGER maxcols,
                                 out SQLINTEGER *nreturned)
```

Returns

SQL_SUCCESS if call is successful. Otherwise, use ttXlaError to report the error.

Parameters

Parameter	Type	Description
handle	ttXlaHandle_h	The transaction log handle for the data store.
record	ttXlaUpdateDesc_t *	The XLA record that must be processed.
colinfo	out ttXlaColDesc_t *	A pointer to the buffer large enough to hold a description for *maxcols* columns.
maxcols	SQLINTEGER	The maximum number of columns the table can have. If the table contains more than *maxcols* columns, an error is returned.
nreturned	out SQLINTEGER *	The number of columns returned.

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 the result.

Required privilege

Requires the system privilege XLA.

Syntax

```
SQLRETURN ttXlaVersionCompare(ttXlaHandle_h handle,
                              ttXlaVersion_t *version1,
                              ttXlaVersion_t *version2,
                              out SQLINTEGER *comparison)
```

Parameters

Parameter	Type	Description
handle	ttXlaHandle_h	The transaction log handle for the data store.
version1	ttXlaVersion_t *	The version of XLA you want to compare with *version2*.
version2	ttXlaVersion_t *	The version of XLA you want to compare with *version1*.
comparison	out SQLINTEGER *	The comparison result. ■ 0: Indicates *version1* and *version2* match. ■ -1: Indicates *version1* is earlier than *version2*. ■ +1: Indicates *version1* is later than *version2*.

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 this 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, we recommend 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

Requires the system privilege XLA.

Syntax

```
SQLRETURN ttXlaVersionTableInfo(ttXlaHandle_h handle,
                                ttXlaUpdateDesc_t *record,
                                out ttXlaTblVerDesc_t *tblinfo)
```

Returns

SQL_SUCCESS if call is successful. Otherwise, use ttXlaError to report the error.

Parameters

Parameter	Type	Description
handle	ttXlaHandle_h	The transaction log handle for the data store.
record	ttXlaUpdateDesc_t *	The XLA record that must be processed.
tableinfo	out ttXlaTblVerDesc_t *	Information about table definition.

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);
```

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:

```
install_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.
ttXlaStatus_t	Describes XLA status information returned by ttXlaStatus.
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	Description of 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 includes the 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 data store. 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 data store. The row data differs depending on the record type reported in the `ttXlaUpdateDesc_t` header:

- No rows are included in a COMMITONLY record.
- One row is included in INSERTTUP, DELETETUP, or SETREPL records.
- Two rows are included in an UPDATETUP record to report the row data before and after the update, respectively.
- Special format rows are included in CREATAB, DROPTAB, CREAIND, DROPIND, CREATVIEW, DROPVIEW, CREATSEQ, DROPSEQ, ADDCOLS, DRPCOLS, SETTBLI, and SETCOLI records, which are described in "Special update data formats" on page 9-73.

Note: SETREPL, SETTBLI and SETCOLI records are not returned in persistent mode.

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" on page 5-38 for a description of the `ttApplicationContext` procedure.

Note

XLA cannot receive notification of:

- A CREATE VIEW or DROP VIEW for a non-materialized view.
- A 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 type:

- 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 from cascading deletes are visible through XLA; deletes from 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
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. ■ ADDCOLS - Add columns. ■ DRPCOLS - Drop columns. ■ SETREPL - Set table replication status. ■ SETTBLI - Set table user ID. ■ SETCOLI - Set column user ID. ■ TRUNCATE - Truncate table. ■ INSERTTUP - Insert. ■ UPDATETUP - Update. ■ DELETETUP - Delete. ■ COMMITONLY - Commit.

Field	Type	Description

flags	SQLUSMALLINT	Special options on record update:
		▪ TT_UPDCOMMIT - Indicates that the update record is the last record for the transaction. (Implied commit.)
		▪ TT_UPDFIRST - Indicates that the update record is the first record for the transaction.
		▪ TT_UPDREPL - Indicates that this update was the result of a non-XLA TimesTen replicated update from another data store.
		▪ TT_UPDCOLS - 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 SQLUSMALLINT 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 UPDATE statement used. This array is included with all UPDATETUP records.
		▪ TT_UPDDEFAULT - Indicates that the update record (either a CREATAB or ADDCOLS) contains default column values. If set, the default columns are presented as an array of SQLUSMALLINT values followed by a string with all the default values concatenated. The number of SQLUSMALLINT values in the array is equal to the number of columns in the CREATAB or ADDCOLS record.
		▪ TT_CASCDEL - Indicates that the XLA update was generated as part of a cascade delete operation.
		▪ TT_AGING - 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 INTEGER, no default, and "apple" of type VARCHAR2(10) 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.)
		`TT_UPDCOMMIT     1`
		`TT_UPDFIRST      2`
		`TT_UPDREPL       4`
		`TT_UPDCOLS       8`
		`TT_UPDDEFAULT   64`
		`TT_CASCDEL     256`
		`TT_AGING       512`

contextOffset	SQLUINTEGER	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.
connID	SQLUBIGINT	Connection ID owning the transaction.

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" on page 9-83.

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.

CREATE TABLE

For CREATE TABLE, the special row value consists of the ttXlaTblDesc_t record describing the new table, followed by the ttXlaColDesc_t records that describe each column.

ALTER TABLE

For ALTER TABLE, 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 ■ F = Foreign ■ 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	'T' = range, 'H' = hash.
ixUnique	char	'U' = unique index, '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 add columns, the row value is as follows:

Field	Type	Description
ncols	SQLUINTEGER	The number of additional columns.

Following this special row are the ttXlaColDesc_t records describing the new columns.

ttXlaDropColumnTup_t
For drop columns, the row value is as follows:

Field	Type	Description
ncols	SQLUINTEGER	The 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	Indicates whether the sequence number generator will continue to generate numbers after it reaches the maximum or minimum value.Values are: ■ 1 = Cycle. ■ 0 = Do not cycle.
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 MATERIALIZED VIEW operation, the row value is as follows:

Field	Type	Description
vwName	char(31)	Name of materialized view.
vwOwner	char(31)	Owner of materialized view.
sysTableID	SQLUBIGINT	System table ID stored in SYS.TABLES.

ttXlaDropViewTup_t

For a DROP VIEW operation on a materialized view, the row value is as follows:

Field	Type	Description
vwName	char(31)	Name of materialized view.
vwOwner	char(31)	Owner of materialized view.

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	The 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

See "Retrieving update records from the transaction log" on page 5-12 and "Inspecting record headers and locating row addresses" on page 5-15 for a detailed discussion on obtaining update records and inspecting the contents of ttXlaUpdateDesc_t headers. Below is a summary of these procedures.

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.

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" on page 9-81 for details on how to access the column data in a returned row.

ttXlaStatus_t

The `ttXlaStatus_t` structure shows runtime operational information about the XLA system. This structure is returned by the ttXlaStatus function when operating XLA in non-persistent mode.

Field	Type	Description
header	ttXlaNodeHdr_t	Standard data header.
xlabuffree	SQLUBIGINT	Free bytes in the staging buffer.
xlabufminfree	SQLUBIGINT	Minimum free bytes in the staging buffer.
xlabufalloc	SQLUBIGINT	Allocated bytes in the staging buffer.
xlabuftran	SQLUBIGINT	Number of transactions in the staging buffer.
xlabufrec	SQLUBIGINT	Number of records in the staging buffer.
logbuffree	SQLUBIGINT	Number of free bytes in the transaction log buffer.
logbufminfree	SQLUBIGINT	Minimum free bytes in the transaction log buffer.
logbufalloc	SQLUBIGINT	Number of allocated bytes in the transaction log buffer.
flags	SQLUINTEGER	A bit map of status flags. Currently, only the TTXLASTAT_STALLED flag is defined. If set, this flag specifies that the XLA staging buffer is full and new updates are being rejected.

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 includes 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).
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 includes 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 4 bytes of inline row space.

Note the following if the table has a declared primary key:

- *nprimcols* is larger than 0.
- *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.
- *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 includes 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 the `ttXlaColDesc_t` structure. This structure is returned by the ttXlaGetColumnInfo function.

This structure includes the following fields:

Field	Type	Description
header	ttXlaNodeHdr_t	Standard data header.
colName [tt_NameLenMax]	char	Name of the column.
pad0	SQLUINTEGER	Pad to 4-byte boundary.
sysColNum	SQLUINTEGER	Column number, numbered from 1.
userColNum	SQLUINTEGER	User-assigned column number.
dataType	SQLUINTEGER	Structure in ODBC TTXLA_* code. See "About XLA data types" on page 5-6.
size	SQLUINTEGER	Max or basic size of column.
offset	SQLUINTEGER	Offset to fixed-length part of column.
nullOffset	SQLUINTEGER	Offset to null byte; 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" on page 5-17. 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" on page 5-22.

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 Example 9–1.

- For variable-length column data (VARCHAR and VARBINARY), `offset` is an address that points to a 4-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 *n* bytes at this location is the length of the data, followed by the actual data (where *n* is 4 on 32-bit platforms or 8 on 64-bit platforms). For variable-length data, the returned size value is the maximum allowable column size. See Example 9–1.

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" on page 5-24 for a discussion.

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.

Example 9–1 Copying and printing a VARCHAR string

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, in the case of 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 4-byte offset value. Adding this offset value to the address of `pColVal` provides a pointer to the beginning of the variable-length column data. Assuming the operation is performed on a 64-bit platform, the first 8 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 will have 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 will have a higher value.

10

TimesTen ODBC Functions and Options

This chapter covers the topics noted below, listing ODBC functions supported by TimesTen and options supported by TimesTen for set and get functions for statements and connections. For complete function definitions, refer to ODBC API reference documentation.

TimesTen supports ODBC 2.5, Extension Level 1, with additional features for Extension Level 2 as indicated in this chapter.

- Supported ODBC functions
- Option support for ODBC connection and statement functions

Supported ODBC functions

This section lists ODBC function supported by TimesTen, with special notes as applicable.

Table 10–1 Supported ODBC functions

Function	Notes
`SQLAllocConnect`	
`SQLAllocEnv`	
`SQLAllocStmt`	
`SQLBindCol`	
`SQLBindParameter`	See "SQLBindParameter function" on page 1-12.
`SQLCancel`	In TimesTen, `SQLCancel` cannot cancel a function running asynchronously on the *`hstmt`* or one running on the *`hstmt`* on another thread.
`SQLColAttributes`	
`SQLColumns`	
`SQLConnect`	
`SQLDataSources`	Available only to programs using a driver manager.
`SQLDescribeCol`	
`SQLDescribeParam`	
`SQLDisconnect`	
`SQLDriverConnect`	
`SQLDrivers`	Available only to programs using a driver manager.

Table 10–1 (Cont.) Supported ODBC functions

Function	Notes
`SQLError`	
`SQLExecDirect`	
`SQLExecute`	
`SQLFetch`	
`SQLForeignKeys`	
`SQLFreeConnect`	
`SQLFreeEnv`	
`SQLFreeStmt`	
`SQLGetConnectOption`	See "Option support for SQLSetConnectOption and SQLGetConnectOption" on page 10-3.
`SQLGetCursorName`	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.
`SQLGetData`	
`SQLGetFunctions`	
`SQLGetInfo`	
`SQLGetStmtOption`	See "Option support for SQLSetStmtOption and SQLGetStmtOption" on page 10-5.
`SQLGetTypeInfo`	
`SQLNativeSql`	
`SQLNumParams`	
`SQLNumResultCols`	
`SQLParamData`	
`SQLParamOptions`	
`SQLPrepare`	
`SQLPrimaryKeys`	
`SQLProcedureColumns`	
`SQLProcedures`	
`SQLPutData`	
`SQLRowCount`	In addition to its standard functionality, this has special usage with cache groups. See "Managing cache groups" on page 1-25.
`SQLSetConnectOption`	See "Option support for SQLSetConnectOption and SQLGetConnectOption" on page 10-3.
`SQLSetCursorName`	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.
`SQLSetStmtOption`	See "Option support for SQLSetStmtOption and SQLGetStmtOption" on page 10-5.
`SQLSetParam`	ODBC 1.0 function, replaced by `SQLBindParameter` in ODBC 2.0. Retained for backward compatibility.

Table 10–1 (Cont.) Supported ODBC functions

Function	Notes
`SQLSpecialColumns`	
`SQLStatistics`	
`SQLTables`	
`SQLTransact`	

Option support for ODBC connection and statement functions

This section discusses TimesTen option support for the ODBC functions `SQLSetConnectOption`, `SQLGetConnectOption`, `SQLSetStmtOption`, and `SQLGetStmtOption`.

Refer to ODBC API reference documentation for general information about these functions.

Option support for SQLSetConnectOption and SQLGetConnectOption

Table 10–2 and Table 10–3 document TimesTen support for standard and TimesTen-specific options for the ODBC `SQLSetConnectOption` and `SQLGetConnectOption` functions. These functions let you 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, as noted.

Note: An option setting through `SQLSetConnectOption` or `SQLSetStmtOption` overrides the setting of the corresponding connection attribute (as applicable).

Table 10–2 Standard options: SQLSetConnectOption, SQLGetConnectOption

Option	Support
SQL_ACCESS_MODE	No
SQL_AUTOCOMMIT	Yes
SQL_CURRENT_QUALIFIER	No
SQL_LOGIN_TIMEOUT	No
SQL_MAX_ROWS	Yes
SQL_NOSCAN	Yes
SQL_ODBC_CURSORS	Available only to programs using a driver manager.
SQL_OPT_TRACE	Available only to programs using a driver manager.
SQL_OPT_TRACEFILE	Available only to programs using a driver manager.
SQL_PACKET_SIZE	No
SQL_QUIET_MODE	No
SQL_TRANSLATE_DLL	No
SQL_TRANSLATE_OPTION	No

Table 10–2 (Cont.) Standard options: SQLSetConnectOption, SQLGetConnectOption

Option	Support
SQL_TXN_ISOLATION	Supported only if *`vParam`* is SQL_TXN_READ_COMMITTED or SQL_TXN_SERIALIZABLE. See "Prefetching multiple rows of data" on page 1-20 and "Concurrency control" in *Oracle TimesTen In-Memory Database Operations Guide*. Same functionality as the `Isolation` connection attribute, as described in "Isolation" in *Oracle TimesTen In-Memory Database Reference*.

Table 10–3 TimesTen options: SQLSetConnectOption, SQLGetConnectOption

Option	Comments
TT_PREFETCH_CLOSE	See "Enable TT_PREFETCH_CLOSE for serializable transactions" in *Oracle TimesTen In-Memory Database Operations Guide*.
TT_CLIENT_TIMEOUT	For client/server only. Same functionality as the `TTC_Timeout` connection attribute, as described in "TTC_Timeout" in *Oracle TimesTen In-Memory Database Reference*.
TT_NLS_SORT	See "Setting globalization options" on page 1-25. Same functionality as the `NLS_SORT` connection attribute described in "NLS_SORT" in *Oracle TimesTen In-Memory Database Reference*. There is also related information about the functionality in "Additional globalization features" on page 3-3.
TT_NLS_LENGTH_SEMANTICS	See "Setting globalization options" on page 1-25. Same functionality as the `NLS_LENGTH_SEMANTICS` connection attribute described in "NLS_LENGTH_SEMANTICS" in *Oracle TimesTen In-Memory Database Reference*. There is also related information about the functionality in "Additional globalization features" on page 3-3.
TT_NLS_NCHAR_CONV_EXCP	See "Setting globalization options" on page 1-25. Same functionality as the `NLS_NCHAR_CONV_EXCP` connection attribute described in "NLS_NCHAR_CONV_EXCP" in *Oracle TimesTen In-Memory Database Reference*. There is also related information about the functionality in "Additional globalization features" on page 3-3.
TT_DYNAMIC_LOAD_ENABLE	See "Disabling dynamic loading" in *Oracle In-Memory Database Cache User's Guide*. This has the same functionality as the `DynamicLoadEnable` connection attribute described in "DynamicLoadEnable" in *Oracle TimesTen In-Memory Database Reference*.

Table 10–3 (Cont.) TimesTen options: SQLSetConnectOption, SQLGetConnectOption

Option	Comments
TT_DYNAMIC_LOAD_ERROR_MODE	See "Displaying dynamic load errors" in *Oracle In-Memory Database Cache User's Guide*. Same functionality as the `DynamicLoadErrorMode` connection attribute described in "DynamicLoadErrorMode" in *Oracle TimesTen In-Memory Database Reference*.
TT_REGISTER_FAILOVER_CALLBACK	See "Automatic client failover" on page 1-29.

Option support for SQLSetStmtOption and SQLGetStmtOption

Table 10–4 and Table 10–5 document TimesTen support for standard and TimesTen-specific options for the ODBC `SQLSetStmtOption` and `SQLGetStmtOption` functions, which let you set or retrieve statement option settings.

Note: An option setting through `SQLSetConnectOption` or `SQLSetStmtOption` overrides the setting of the corresponding connection attribute (as applicable).

Table 10–4 Standard options: SQLSetStmtOption, SQLGetStmtOption

Option	Support
SQL_ASYNC_ENABLE	No
SQL_BIND_TYPE	No
SQL_CONCURRENCY	No
SQL_CURSOR_TYPE	No
SQL_KEYSET_SIZE	No
SQL_MAX_LENGTH	No. SQL_MAX_LENGTH can be set, but any specified value will be overridden with 0 (return all available data).
SQL_MAX_ROWS	Yes
SQL_NOSCAN	Yes
SQL_QUERY_TIMEOUT	Yes. See "Setting a timeout or threshold for executing SQL statements" on page 1-23.
SQL_RETRIEVE_DATA	No
SQL_ROWSET_SIZE	No
SQL_SIMULATE_CURSOR	No
SQL_USE_BOOKMARKS	No

Table 10–5 TimesTen options: SQLSetStmtOption, SQLGetStmtOption

Option	Comment
TT_PREFETCH_COUNT	See "Prefetching multiple rows of data" on page 1-20.

Table 10–5 (Cont.) TimesTen options: SQLSetStmtOption, SQLGetStmtOption

Option	Comment
TT_QUERY_THRESHOLD	See "Setting a timeout or threshold for executing SQL statements" on page 1-23.
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 will be passed through to Oracle by the passthrough feature of IMDB Cache. 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 the passthrough setting for a statement" in *Oracle In-Memory Database Cache User's Guide*.

11 ODBC for UNIX

ODBC programming for UNIX is fundamentally the same as ODBC programming on Windows. There is a difference, however, regarding driver managers:

- UNIX platforms usually do not supply an ODBC driver manager. On UNIX, it is typical for programs to be linked directly with the TimesTen Data Manager ODBC driver or the TimesTen Client ODBC driver.
- Windows platforms do supply an ODBC driver manager, and it is typical for programs on Windows to use it.

TimesTen supplies a driver manager for either Windows or UNIX with the Quick Start sample applications.

The TimesTen Data Manager ODBC driver and the TimesTen Client ODBC driver are ODBC 2.5 drivers closer in functionality to ODBC 2.0 than to ODBC 3.0.

This chapter covers the following topics:

- Inapplicable chapters in the Microsoft ODBC manual
- Minor inconsistencies
- Microsoft ODBC 2.5

Inapplicable chapters in the Microsoft ODBC manual

A few chapters in the *Microsoft ODBC Programmer's Reference and SDK Guide*, version 2.0, are not relevant for UNIX developers working with TimesTen, in case you use that document:

- Chapter 18, "Constructing an ODBC Driver", and Chapter 24, "Installer DLL Function Reference", are relevant only for Windows developers.
- Chapter 19 and Chapter 20 are platform-specific and are also not relevant for TimesTen applications programmers. They are only relevant for ODBC driver developers.

Minor inconsistencies

In addition to the inapplicable chapters, consider the following issues if you work with the *Microsoft ODBC Programmer's Reference and SDK Guide*, version 2.0, as a UNIX user:

- **Inapplicable information.** Some information in the Microsoft ODBC 2.0 manual is inapplicable for UNIX users. This includes information on Microsoft products, such as comparisons with Microsoft Excel® or caveats that apply to the Windows environment only.

- **DLL/Shared library.** The equivalent of a Windows DLL (dynamically linked library) is a UNIX shared library.
- **File names.** Under Windows, all file names are uppercase. Under UNIX, all filenames are lowercase. Otherwise the names are equivalent except for the `odbc.ini` file.
- **odbc.ini file or registry.** The Microsoft ODBC 2.0 manual states that certain information is stored in the registry. On UNIX platforms, this information is stored in the `$HOME/.odbc.ini` file.
- **Character restrictions.** The Microsoft ODBC 2.0 manual states character restrictions for certain functions, such as `ConfigDSN`, that differ from the restrictions in UNIX, as follows:
  - ODBC on Windows: The keywords and their values should not contain `[]{}(),;?*=!@` characters, and the value of the DSN keyword cannot consist only of blanks. Because of the registry grammar, keywords and data store names cannot contain the backslash (\) character.
  - ODBC on UNIX: The keywords and their values should not contain the `[]{}(),;?*=!@ \` characters, and the value of the DSN keyword cannot consist only of blanks.

Microsoft ODBC 2.5

The information in this section was taken, with permission, directly from the `README.TXT` file for Microsoft ODBC 2.5.

© Copyright Microsoft® Corporation, 1995.

Note: Information that is not applicable for TimesTen developers was cut from the material distributed by Microsoft Corporation.

The standard and extended header files, `SQL.H` and `SQLEXT.H`, have been modified in ODBC 2.5 to align with changes in the X/Open CAE specification, as follows:

- All material in `SQL.H` that was specific to Microsoft has been moved to `SQLEXT.H`. The format of the file was changed so that the data types and return types conform to the X/Open CAE specification.
- All material in `SQLEXT.H` that has been adopted by the standard has been moved to `SQL.H`.

`SQLTYPES.H` has been added to provide type definition for program types in ODBC 2.5. `SQLTYPES.H` defines the handle environment; SQL portable types for C; transfer types for DATE, TIME and TIMESTAMP; and bookmarks.

Index