Oracle TimesTen In-Memory Database C Developer's and Reference Guide

Release 6.0

B25265-03

For last-minute updates, see the TimesTen release notes.

Copyright ©1996, 2006, Oracle. All rights reserved.

ALL SOFTWARE AND DOCUMENTATION (WHETHER IN HARD COPY OR ELECTRONIC FORM) ENCLOSED AND ON THE COMPACT DISC(S) ARE SUBJECT TO THE LICENSE AGREEMENT.

The documentation stored on the compact disc(s) may be printed by licensee for licensee's internal use only. Except for the foregoing, no part of this documentation (whether in hard copy or electronic form) may be reproduced or transmitted in any form by any means, electronic or mechanical, including photocopying, recording, or any information storage and retrieval system, without the prior written permission of TimesTen Inc.

Oracle, JD Edwards, PeopleSoft, Retek, TimesTen, the TimesTen icon, MicroLogging and Direct Data Access are trademarks or registered trademarks of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

The Programs (which include both the software and documentation) contain proprietary information; they are provided under a license agreement containing restrictions on use and disclosure and are also protected by copyright, patent, and other intellectual and industrial property laws. Reverse engineering, disassembly, or decompilation of the Programs, except to the extent required to obtain interoperability with other independently created software or as specified by law, is prohibited.

The information contained in this document is subject to change without notice. If you find any problems in the documentation, please report them to us in writing. This document is not warranted to be error-free. Except as may be expressly permitted in your license agreement for these Programs, no part of these Programs may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose.

March 2006

Printed in the United States of America

Contents

About this Guide

TimesTen documentation . . . 2
Background reading . . . 3
Installing TimesTen . . . 3
Conventions used in this guide . . . 4
Finding the information you need . . . 6
Technical Support . . . 7

Part I: TimesTen C Developers' Guide

1 Working with TimesTen Data Stores

Connecting to and disconnecting from a data store . . . 11
  Setting data store attributes programmatically . . . 17
Managing TimesTen data . . . 18
  Calling SQL statements within C applications . . . 18
  Calling TimesTen built-in procedures within C applications . . . 19
  Fetching multiple rows of data . . . 19
  Setting a timeout value for executing SQL statements . . . 20
  Working with cursors . . . 21
Handling errors and recovery . . . 23
  Checking for errors . . . 23
  Fatal and non-fatal errors . . . 24
    Fatal errors . . . 24
    Non-fatal errors . . . 24
    Warnings . . . 24
  Recovery . . . 25

2 Compiling and Linking TimesTen Applications

Linking options . . . 27
  Linking directly with a TimesTen ODBC Driver . . . 27
  Linking with a driver manager . . . 28
Compiling and linking applications on Windows . . . 29
Testing link options . . . 30
Compiling and linking UNIX applications . . . 30

3 XLA and TimesTen Event Management

XLA Concepts . . . 34
  XLA modes: persistent vs. non-persistent . . . 34
  How XLA reads records from the transaction log . . . 35

About XLA and materialized views . . . 37
About XLA bookmarks . . . 37
XLA Demos . . . 39
The xlaSimple demo . . . 40
The xlaPersistent demo . . . 41
Compiling and Linking an XLA Application . . . 41
Writing an XLA Event-Handler Application . . . 42
TimesTen #include files . . . 42
Obtaining a data store connection handle . . . 44
Initializing XLA and obtaining an XLA handle . . . 45
Specifying which tables to monitor for updates . . . 46
Retrieving update records from the transaction log . . . 47
Inspecting record headers and locating row addresses . . . 51
Inspecting column data . . . 53
Obtaining column descriptions . . . 53
Reading fixed-length column data . . . 55
Reading NOT INLINE variable-length column data . . . 56
Null-terminating returned strings . . . 58
Converting complex data types . . . 60
Detecting NULL values . . . 61
Putting it all together — a PrintColValues() function . . . 62
Handling XLA errors . . . 67
Terminating an XLA application . . . 70
Deleting bookmarks . . . 72
Using XLA in Non-Persistent Mode . . . 73
Initializing XLA in non-persistent mode . . . 74
Configuring the staging buffer . . . 74
Retrieving and resetting buffer status . . . 75
Using XLA as a Replication Mechanism . . . 76
Checking table compatibility between data stores . . . 77
Replicating updates between data stores . . . 78
Handling timeout and deadlock errors . . . 79
Checking for update conflicts . . . 80
Replicating updates to a non-TimesTen data store . . . 80
Other XLA Features . . . 81
Changing the location of a bookmark . . . 81
Passing application context . . . 82

4 Distributed Transaction Processing: XA

X/Open DTP Model . . . 84
Two-phase commit . . . 85
One-phase commit optimization . . . 85
Read-only optimization . . . 85
Setting TimesTen data store attributes for XA . . . 85
DurableCommits . . . 85
Logging . . . 86
Recovering Global Transactions . . . 86
Heuristic recovery . . . 86
Using the XA API . . . 87
Primary Documents . . . 87
Using XA functions . . . 88
xa_open() . . . 88
xa_close() . . . 88
Transaction id (XID) parameter . . . 88
Obtaining an ODBC handle from an XA connection . . . 89
tt_xa_context() . . . 89
Calling ODBC functions over an XA connection . . . 91
AUTOCOMMIT . . . 91
Local transaction COMMIT and ROLLBACK . . . 91
Closing open cursors . . . 91
XA resource manager switch . . . 92
xa_switch_t . . . 92
tt_xa_switch . . . 93
XA Support through the ODBC driver manager . . . 94
Linking to the TimesTen ODBC XA driver manager extension library 94
Configuring Tuxedo to use TimesTen XA . . . 95
Update the $TUXDIR/udataobj/RM file . . . 95
Build the Tuxedo transaction manager server . . . 96
Update the GROUPS section in the UBBCONFIG file . . . 96
Compile the servers . . . 98
Error Handling . . . 98

5 Application Tuning

Bypass driver manager if appropriate . . . 100
Prepare statements in advance . . . 101
Avoid unnecessary prepare operations . . . 101
Use arrays of parameters to batch operations . . . 102
Avoid excessive binds . . . 102
Avoid SQLGetData . . . 102
Avoid data type conversions . . . 102
Reduce contention . . . 103

Bulk fetch rows of TimesTen data 103
Choose the best method of locking 104
Choose an appropriate lock level 104
Choose an appropriate isolation level 105
Turn off autocommit mode 105
Size transactions appropriately 106
Use durable commits appropriately 107
Choose the appropriate logging options 107
Choose a timeout interval 108
Avoid transaction rollback 108
Avoid frequent checkpoints 109

Part II: TimesTen C Reference Guide

6 TimesTen Utility API

Utility API list 114
ttBackup 115
ttDestroyDataStore 119
ttDestroyDataStoreForce 121
ttRamGrace 123
ttRamLoad 124
ttRamPolicy 125
ttRamUnload 127
ttRepDuplicate 128
ttRepDuplicateAcc 128
ttRepDuplicateEx 129
ttRestore 133
ttUtilAllocEnv 135
ttUtilFreeEnv 137
ttUtilGetError 139
ttUtilGetErrorCount 142
ttXactIdRollback 144

7 XLA Reference

About XLA Functions . . . 145
About return codes . . . 145
About parameter types (input, out, inout) . . . 145
About results output by functions . . . 146
Summary of XLA Functions by Category . . . 146
XLA persistent mode functions . . . 147
XLA core functions . . . 147
XLA non-persistent mode functions . . . 149
XLA replication functions . . . 150
XLA Function Reference . . . 150
ttXlaAcknowledge . . . 151
ttXlaApply . . . 152
ttXlaClose . . . 154
ttXlaCommit . . . 155
ttXlaConfigBuffer . . . 156
ttXlaDateToODBCCType . . . 158
ttXlaDecimalToCString . . . 159
ttXlaDeleteBookmark . . . 161
ttXlaError . . . 162
ttXlaErrorRestart . . . 164
ttXlaGenerateSQL . . . 165
ttXlaGetColumnInfo . . . 167
ttXlaGetLSN . . . 169
ttXlaGetTableInfo . . . 170
ttXlaGetVersion . . . 171
ttXlaLookup . . . 172
ttXlaNextUpdate . . . 174
ttXlaNextUpdateWait . . . 176
ttXlaOpenTimesTen . . . 178
ttXlaPersistOpen . . . 180
ttXlaResetStatus . . . 183
ttXlaRollback . . . 184
ttXlaSetLSN . . . 185
ttXlaSetVersion . . . 186
ttXlaStatus . . . 187
ttXlaTableByName . . . 188
ttXlaTableCheck . . . 189
ttXlaTableStatus . . . 191
ttXlaTimeToODBCCType . . . 193

ttXlaTimeStampToODBCCType 194
ttXlaVersionCompare 195
XLA Data Types 197
Summary of XLA Data Types 197
ttXlaNodeHdr_t 198
ttXlaUpdateDesc_t 199
Special Update Data Formats 203
Locating the row data following a ttXlaUpdateDesc_t header . . . 207
ttXlaStatus_t 208
ttXlaVersion_t 209
ttXlaTblDesc_t 210
ttXlaColDesc_t 211
tt_LSN_t 213
tt_XlaLsn_t 213

8 Working with Unicode

ODBC 215
SQL data types 215
C data type 216
SQL_C_DEFAULT 216
Include File Definitions 217
Data Conversions 218
SQL to C Data Conversions 218
SQL to C: Character 218
C to SQL Data Conversions 218
C to SQL: Character 219
C to SQL: Binary 219
Function Support 220
ODBC Driver Manager 221
Unicode Escapes Embedded in C Strings 221
Encoding of Bound Data 221

9 TimesTen ODBC Functions

Supported ODBC functions 223
Supported and unsupported options 225

10 ODBC for UNIX

Irrelevant chapters 227
Minor inconsistencies 228
Microsoft ODBC 2.5 229
Header files 229
ODBC function changes 229
Rebinding with SQLBindCol 229
Binding a LONG on 64-bit platforms 229
pcbValue in SQLBindParameter 230
SQLSTATE S1C00 returned by SQLPrepare 230
SQLSTATE 22005 returned by SQLFetch 230
SQLSTATE 22008 returned by SQLFetch 230
SQLSTATE 22012 returned by SQLGetData 230
ODBC functions 230
ODBC data types 231

Glossary

Index

About this Guide

TimesTen® is a high-performance, in-memory data manager that supports the ODBC and JDBC interfaces. The examples and procedures in this guide use the ODBC interface.

This guide is for application developers who use and administer TimesTen ODBC and for system administrators who configure and manage the TimesTen Daemon. It provides:

- Background information to help you understand how TimesTen works.
- Step-by-step instruction and examples that show how to perform the most commonly needed tasks.

To work with this guide, you should understand how database systems work. You should also have knowledge of SQL (Structured Query Language) and ODBC (Open DataBase Connectivity). See “Background reading” on page 3. if you are not familiar with these interfaces.

TimesTen documentation

Including this guide, the TimesTen documentation set consists of these documents:

- The *Oracle TimesTen In-Memory Database Installation Guide* provides information needed to install and configure TimesTen on all supported platforms.
- The *Oracle TimesTen In-Memory Database Architectural Overview* provides a description of all the available features in TimesTen.
- The *Oracle TimesTen In-Memory Database Operations Guide* provides information on configuring TimesTen and using the ttIsql utility to manage a data store. This guide also provides a basic tutorial for TimesTen.
- The *Oracle TimesTen In-Memory Database C Developer's and Reference Guide* and the *Oracle TimesTen In-Memory Database Java Developer's and Reference Guide* provide information on how to use the full set of available features in TimesTen to develop and implement applications that use TimesTen.
- The *Oracle TimesTen In-Memory Database Recommended Programming Practices* provides information that will assist developers who are writing applications to work with TimesTen.
- The *Oracle TimesTen In-Memory Database API and SQL Reference Guide* contains a complete reference to all TimesTen utilities, procedures, APIs and other features of TimesTen.
- The *Oracle TimesTen In-Memory Database TTClasses Guide* describes how to use the TTClasses C++ API to use the features available features in TimesTen to develop and implement applications that use TimesTen.
- The *TimesTen to TimesTen Replication Guide*. This guide is for application developers who use and administer TimesTen and for system administrators who configure and manage TimesTen Replication. This guide provides background information to help you understand how TimesTen Replication works and step-by-step instructions and examples that show how to perform the most commonly needed tasks.
- The *TimesTen Cache Connect to Oracle Guide* describes how to use Cache Connect to cache Oracle data in TimesTen. This guide is for developers who use and administer TimesTen for caching Oracle data. It provides information on caching Oracle data in TimesTen data stores. It also describes how to use the Cache Administrator, a web-based interface for creating cache groups.
- The *Oracle TimesTen In-Memory Database Troubleshooting Procedures Guide* provides information and solutions for handling problems that may arise while developing applications that work with TimesTen, or while configuring or managing TimesTen.

TimesTen documentation is available on the product CD-ROM and on the Oracle Technology Network: http://www.oracle.com/technology/documentation/timesten_doc.html.

Background reading

An extensive list of books about ODBC and SQL is in the Microsoft ODBC manual included in your developer's kit. In addition to this guide, your developer's kit includes the appropriate ODBC manual for your platform:

WINDOWS

– *Microsoft ODBC 3.0 Programmer's Reference and SDK Guide* provides all relevant information on ODBC for Windows developers.

UNIX

– *Microsoft ODBC 2.0 Programmer's Reference and SDK Guide*, included online in PDF format, provides information on ODBC for UNIX developers.

For a conceptual overview and programming how-to of ODBC, see:

- Sanders, Roger E. *ODBC 3.5 Developer's Guide* (McGraw-Hill Series On Data Warehousing and Data Management); McGraw-Hill. 1999
- Signore, Robert / Stegman, Michael O. / et al. *ODBC Solution: Open Database Connectivity in Distributed Environments*: *Mcgraw-hill Series On Computer Communications*; McGraw Hill. 1995

For a review of SQL, see:

- Melton, Jim and Simon, Alan R. *Understanding the New SQL: A Complete Guide*. San Francisco, CA: Morgan Kaufmann Publishers. 1993.
- Groff, James R. / Weinberg, Paul N. *SQL: The Complete Reference.* McGraw-Hill. /1999

For information on Unicode, see:

- The Unicode Consortium, *The Unicode Standard, Version 3.0*, Addison-Wesley, 2000.
- The Unicode Consortium Home Page at `http://www.unicode.org`

Installing TimesTen

TimesTen Release 6.0 includes the TimesTen Data Manager for 32-bit and 64-bit platforms. See the *Oracle TimesTen In-Memory Database Installation Guide* for a description of supported platforms.

In addition to the Data Manager, TimesTen Release 6.0 also includes TimesTen Client and Server components. You can install the TimesTen Data Manager stand-alone or in a client/server environment.

For a list of the The TimesTen default installation directories, see the *Oracle TimesTen In-Memory Database Installation Guide*.

Conventions used in this guide

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, Tru64 and AIX.

TimesTen documentation uses these typographical conventions:

If you see...	It means...
`code font`	Code examples, filenames, and pathnames. For example, the `.odbc.ini`.`ttconnect.ini` file.
`italic code font`	A variable in a code example that you must replace. For example: `Driver=`*`install_dir`*`/lib/libtten.sl` Replace *`install_dir`* with the path of your TimesTen installation directory.

TimesTen documentation uses these conventions in command line examples and descriptions:

If you see...	It means...
`fixed width italics`	Variable; must be replaced
`[ ]`	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 arguments that you may use more than one argument on a single command line.
`...`	An ellipsis (. . .) after an argument indicates that you may use more than one argument on a single command line.
`%`	The percent sign indicates the UNIX shell prompt.
`#`	The number (or pound) sign indicates the UNIX root prompt.

TimesTen documentation uses these variables to identify path, file and user names:

If you see...	It means...
install_dir	The path that represents the directory where the current release of TimesTen is installed.
TTinstance	The instance name for your specific installation of TimesTen. Each installation of TimesTen must be identified at install time with a unique alphanumeric instance name. This name appears in the install path. The instance name “giraffe” is used in examples in this guide.
bits or *bb*	Two digits, either 32 or 64, that represent either the 32-bit or 64-bit operating system.
release or *rr*	Two digits that represent the first two digits of the current TimesTen release number, with or without a dot. For example, 60 or 5.0 represents TimesTen Release 5.0.
jdk_version	Two digits that represent the version number of the major JDK release. For example 14 for versions of jdk1.4.
timesten	A sample name for the TimesTen instance administrator. You can use any legal user name as the TimesTen administrator. On Windows, the TimesTen instance administrator must be a member of the Administrators group. Each TimesTen instance can have a unique instance administrator name.
DSN	The data source name.

Finding the information you need

The table below provides a brief overview of the TimesTen development process. It will help you get started with your application and find relevant information as you progress.

To learn how to	See
Install TimesTen.	*Oracle TimesTen In-Memory Database Installation Guide*
Create a data store and establish and close a connection.	Chapter 2, "Creating TimesTen Data Stores" and Chapter 3, "Working with the TimesTen Client and Server." in the *Oracle TimesTen In-Memory Database Operations Guide*.
Resize a data store.	"Specifying the size of a data store" in the *Oracle TimesTen In-Memory Database Operations Guide*.
Compile and run a TimesTen application.	Chapter 2, "Compiling and Linking TimesTen Applications"
Set up the TimesTen Client and Server.	Chapter 3, "Working with the TimesTen Client and Server" in the *Oracle TimesTen In-Memory Database Operations Guide*.
Work with TimesTen tables, indexes and rows.	Chapter 6, "Working with Data in a TimesTen Data Store" in the *Oracle TimesTen In-Memory Database Operations Guide*.
Understand options regarding transaction semantics.	Chapter 7, "Transaction Management and Recovery" in the *Oracle TimesTen In-Memory Database Operations Guide*.
View and change the query optimizer plan.	Chapter 9, "The TimesTen Query Optimizer" in the *Oracle TimesTen In-Memory Database Operations Guide*.
Use XLA to monitor changes to a data store	Chapter 3, "XLA and TimesTen Event Management"
Replicate your system.	The *TimesTen to TimesTen Replication Guide*.

To learn how to	See
Achieve optimum performance.	Chapter 5, "Application Tuning" and Chapter 8, "Data Store Performance Tuning" in the *Oracle TimesTen In-Memory Database Operations Guide*.
Find troubleshooting information.	The *Oracle TimesTen In-Memory Database Troubleshooting Procedures Guide*.
Use TimesTen procedures.	The *Oracle TimesTen In-Memory Database API and SQL Reference Guide*.
Use TimesTen utilities.	The *Oracle TimesTen In-Memory Database API and SQL Reference Guide*.

Technical Support

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

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

Email: timesten-support_us@oracle.com

Part I: TimesTen C Developers' Guide

1

Working with TimesTen Data Stores

See Chapter 2, "Creating TimesTen Data Stores in the *Oracle TimesTen In-Memory Database Operations Guide* for details on how to create a data store. This chapter describes how to use ODBC to connect to and use a TimesTen data store.

Topics in this chapter are:

- Connecting to and disconnecting from a data store
- Managing TimesTen data
- Handling errors and recovery

Connecting to and disconnecting from a data store

How to create a DSN for a TimesTen data store is described in the *Oracle TimesTen In-Memory Database Operations Guide*. The type of DSN you create depends on whether your application is to connect directly to the data store, or by means of client connection. If you intend to connect directly to the data store, create a DSN as described in "Creating a DSN on UNIX" or "Creating a DSN on Windows" in the *Oracle TimesTen In-Memory Database Operations Guide*. If you intend to create a client connection to the data store, create a DSN as described in "Creating and configuring Client DSNs on Windows" or "Creating and configuring Client DSNs on UNIX" in the *Oracle TimesTen In-Memory Database Operations Guide*.

To connect to a data store, call one of the ODBC functions **SQLConnect** or **SQLDriverConnect**. To disconnect from a data store, call the ODBC function **SQLDisconnect**. For a full description of these functions, see the *Microsoft ODBC Programmer's Reference and SDK Guide.*

The code fragment in Example 1.1 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 data store `FixedDs`. Subsequent invocations of **SQLConnect** connect to the existing data store.

Example 1.1

```
#include <sql.h>

SQLRETURN   retcode;
SQLHDBC     hdbc;

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

Example 1.2

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. It also shows how to get error messages.

```
#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>

void CheckReturnCode(SQLRETURN rc, SQLHENV henv,
                SQLHDBC hdbc, SQLHSTMT hstmt,
                char* msg, char *filename,
                int lineno);
```

```
void main( void )
{

  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     *ConnString = (SQLCHAR *)
                "DSN=OperationalDS;PermSize=1;"
                     /* Connection attributes */

  rc = SQLAllocEnv(&hEnv);
  if (rc != SQL_SUCCESS) {
    fprintf(stderr,"Unable to allocate an "
                   "environment handle\n");
    exit(1);
  }

  rc = SQLAllocConnect(hEnv, &hDbc);
  CheckReturnCode(rc, hEnv, SQL_NULL_HDBC,
                  SQL_NULL_HSTMT,
                  "Unable to allocate a "
                  "connection handle\n",
                  __FILE__, __LINE__);

  rc = SQLDriverConnect(hDbc, NULL,
                        ConnString, SQL_NTS,
                        ConnOut, 255,
                        &connOutLen,
                        SQL_DRIVER_NOPROMPT);
  CheckReturnCode(rc, hEnv, hDbc, SQL_NULL_HSTMT,
                  "Error in connecting to the"
                  " driver\n",
                  __FILE__, __LINE__);
```

```
  rc = SQLAllocStmt(hDbc, &hStmt);
  CheckReturnCode(rc, hEnv, hDbc, SQL_NULL_HSTMT,
                  "Unable to allocate a "
                  "statement handle\n",
                  __FILE__,__LINE__);

    /*  Your application code here */

  if (hStmt != SQL_NULL_HSTMT) {
    rc = SQLFreeStmt(hStmt, SQL_DROP);
    CheckReturnCode(rc, hEnv, hDbc, hStmt,
                    "Unable to free the "
                    "statement handle\n",
                    __FILE__, __LINE__);
  }

  if (hDbc != SQL_NULL_HDBC) {
    rc = SQLDisconnect(hDbc);
    CheckReturnCode(rc, hEnv, hDbc,
                    SQL_NULL_HSTMT,
                    "Unable to close the "
                    "connection\n",
                    __FILE__, __LINE__);

    rc = SQLFreeConnect(hDbc);
    CheckReturnCode(rc, hEnv, hDbc,
                    SQL_NULL_HSTMT,
                    "Unable to free the "
                    "connection handle\n",
                    __FILE__, __LINE__);
  }

  if (hEnv != SQL_NULL_HENV) {
    rc = SQLFreeEnv(hEnv);
    CheckReturnCode(rc, hEnv, SQL_NULL_HDBC,
                         SQL_NULL_HSTMT,
                         "Unable to free the "
                         "environment handle\n",
                         __FILE__, __LINE__);
  }
}
```

```
void CheckReturnCode(SQLRETURN rc, SQLHENV henv,
                SQLHDBC hdbc, SQLHSTMT hstmt,
                char* msg, char *filename,
                int lineno)
{
#define MSG_LNG 512

  SQLCHAR     szSqlState[MSG_LNG];
             /* SQL state string */
  SQLINTEGER pfNativeError;
             /* Native error code */
  SQLCHAR     szErrorMsg[MSG_LNG];
             /* Error msg text buffer pointer */
  SQLSMALLINT pcbErrorMsg;
             /* 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,
                     szSqlState, &pfNativeError,
                     szErrorMsg, MSG_LNG,
                     &pcbErrorMsg);
```

```
      switch (ret) {
        case SQL_SUCCESS:
          fprintf(stderr, "*** %s\n"
                  "*** ODBC Error/Warning = %s, "
                  "TimesTen Error/Warning "
                  " = %d\n",
                  szErrorMsg, szSqlState,
                  pfNativeError);
          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 */
};
```

Setting data store attributes programmatically

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

Note: When caching Oracle data in TimesTen, the OracleID attribute must be set in the TimesTen ODBC Setup dialog or the ODBC.INI file. To prevent confusion between connected clients, you should set the OracleID programmatically. For more information on caching Oracle data in TimesTen, see the *TimesTen Cache Connect to Oracle Guide*.

To connect to a data store called `BulkInsData` and indicate that the application should use data store-level locking rather than the default row-level locking, the application uses the **SQLDriverConnect** `routine`, as shown in the following code fragment:

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

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

Note: Each connection to a TimesTen data store opens a number of files. This means that an application with many threads, each with a separate connection, will have 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 the *Oracle TimesTen In-Memory Database API and SQL Reference Guide* for additional information.

Managing TimesTen data

This section provides detailed information on working with data in a TimesTen data store.

The topics are:

- Calling SQL statements within C applications
- Calling TimesTen built-in procedures within C applications
- Fetching multiple rows of data
- Setting a timeout value for executing SQL statements
- Working with cursors

Calling SQL statements within C applications

Chapter 6, "Working with Data in a TimesTen Data Store in the *Oracle TimesTen In-Memory Database Operations Guide* describes how to use SQL to manage data in a TimesTen data store. This section describes the general format used to call a SQL statement within a C application.

Call SQL statements with the **SQLExecDirect** function using the format:

```
#include <sql.h>
SQLRETURN rc;
SQLHSTMT hstmt;
...
rc = SQLExecDirect(hstmt, (SQLCHAR*) "SQL Statement", SQL_NTS);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
... /* handle error */
```

Example 1.3 For example, the following C code fragment 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 */
```

For more information about SQL, see the *Oracle TimesTen In-Memory Database API and SQL Reference Guide*.

Calling TimesTen built-in procedures within C applications

Chapter 3, "Built-In Procedures in the *Oracle TimesTen In-Memory Database API and SQL Reference Guide* describes the TimesTen built-in procedures that extend standard ODBC functionality. You can invoke these procedures using the ODBC call interface. The format is:

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

Example 1.4 The following ODBC example calls the **ttCkpt** procedure to initiate a fuzzy checkpoint:

The following JDBC example initiates a fuzzy checkpoint:

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

Fetching multiple rows of data

A TimesTen extension to ODBC allows applications to prefetch multiple rows of data from a TimesTen data store in one call. Fetching multiple rows of data from TimesTen can increase the performance of applications that are using the Read_Committed isolation level.

Note: The TT_PREFETCH_COUNT attribute is only available for direct-linked applications.

The TT_PREFETCH_COUNT attribute can be used in a call to either **SQLSetConnectOption** or **SQLSetStmtOption**. The value can be any integer from 0 to 128, inclusive. To have **SQLFetch** fetch only one row, set prefetch=1. This in effect turns off prefetch. To fetch more than one row at **SQLFetch**, set prefetch to the number of rows you want to fetch. For example to set the prefetch count to 100:

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

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 isolation mode, the default prefetch value is 128; in Read committed isolation isolation mode, the default is 5. The default prefetch value is the optimum setting for most applications.

You can reset the prefetch value to any integer from 0 to 128, inclusive. When resetting the prefetch count, be aware that, when using Read committed isolation isolation, locks will be acquired until commit, so lock duration may increase if there is some delay between the fetch and execution. In Read committed isolation isolation, copies of the rows read will be retained until fetch so if those rows are deleted or updated, multiple copies may exist that consume space.

To disable prefetch, set the prefetch count to 1.

Setting a timeout value for executing SQL statements

To control how long SQL statements should execute before timing out, you can use the ODBC functions **SQL_QUERY_TIMEOUT (SQLSetStmtOption)** or **SQL_QUERY_TIMEOUT (SQLSetConnectOption)**.

In TimesTen you can specify the query timeout value for all connections by using the DSN attribute sqlQueryTimeout. If you set sqlQueryTimeout in the DSN specification, its value becomes the default value for all subsequent connections in TimesTen. A call to SQLSetConnectOption with the option SQL_QUERY_TIMEOUT will override any default value that a connection may have inherited. Similarly, a call to SQLSetStmtOption with the option SQL_QUERY_TIMEOUT will override any default value inherited from the connection.

SQL_QUERY_TIMEOUT specifies the time limit in seconds for which the data store should execute SQL queries. In TimesTen, once the timeout trigger fires it indicates to the executing query that it must timeout. Since there can be a lag in the time that it takes the timeout message to get to the query, the actual time it takes for the query to end is approximately the time it takes for the time out message to get to the query plus the timeout value specified.

The SQL_QUERY_TIMEOUT feature works only when the SQL statement is actively executing. A timeout does not occur during the commit or rollback phase of the operation. For those transactions that do a large number of UPDATEs DELETEs OR INSERTs, the commit or rollback phases may take a long time to complete. During that time the timeout value is ignored.

Note: The lock timeout used in TimesTen and the SQLQueryTimeout are separate features and can have separate values. TimesTen checks for both lock timeout and SQLQueryTimeout values. TimesTen examines all threads and wakes up any sleeping process that has either a lock timeout or a SQLQueryTimeout, and indicates the appropriate SQLQueryTimeout value to any executing thread. If both a lock timeout value and a SQLQueryTimeout value are specified the lesser of the two values causes a timeout first.

Working with cursors

Applications use cursors to scroll through the results of a query, examining one result row at a time.

Cursors are discussed in more detail in the *Microsoft ODBC 2.0 Programmer's Reference and SDK Guide* and in the *Microsoft ODBC 3.0 Software Development Kit and Programmer's Reference* in the section "Scrollable Cursors."

In the ODBC setting, a cursor is always associated with a result set. This association is made by the ODBC driver, but the application can control the cursor's characteristics (such as number of rows to fetch at one time) using calls such as **SQLSetStmtOption**. The steps involved in retrieving results of a query include:

- Use **SQLPrepare** to prepare the statement for execution**SELECT**
- Use **SQLBindParam**, if the statement has parameters, to bind each parameter to an application address.
- Use **SQLBindCol** to bind each result to an application address.
- Call **SQLExecute** to initiate the **SELECT**.

The following example illustrates how this is done in ODBC. In this example, error checking has been omitted to simplify the example.

Note: All open cursors are closed at transaction commit or rollback.

Example 1.5

```
#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] = (UCHAR*) 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);
}
```

Handling errors and recovery

This section discusses how to check for, identify and handle errors in TimesTen applications.

For information on how to write a function to handle standard ODBC errors, see the section "Retrieving errors and warnings" in Chapter 6, "Warnings and Errors" of the *Oracle TimesTen In-Memory Database API and SQL Reference Guide*.

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 on the TimesTen installation media show examples of error checking. A single call may return multiple errors. Your application should be written to return all errors.

Errors can be checked using either their number or *tt_Err* string. For the complete list of possible TimesTen errors (both *tt_Err* string and associated number), see the *install_dir*/include/tt_errCode.h file. For a description of each message, see "List of errors and warnings" in Chapter 6, "Warnings and Errors" of the *Oracle TimesTen In-Memory Database API and SQL Reference Guide*.

After calling standard ODBC functions, check the return code. If the return code is not SQL_SUCCESS, call an ODBC error-handling function that uses the **SQLError** function to retrieve the errors on the ODBC environment handle.

Example 1.6 For example, after calling a standard ODBC function such as **SQLAllocConnect**, you can check to see if the return code is SQL_SUCCESS. If not, then you can call an error-handling function and terminate the application:

```
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);
}
```

In your error-handling function, always check for multiple errors by repeatedly calling the **SQLError** function until all errors are read from the error stack (return code from **SQLError** is SQL_NO_DATA_FOUND). For information on how to write a function to handle standard ODBC errors, see the section "Retrieving errors and warnings" in Chapter 6, "Warnings and Errors" of the *Oracle TimesTen In-Memory Database API and SQL Reference Guide*.

Fatal and non-fatal errors

TimesTen can return either fatal or non-fatal errors.

Fatal errors

Fatal errors are those that make the data store inaccessible until it can be recovered. 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 code should include a disconnect from the data store.

Non-fatal errors

Non-fatal errors include simple errors such as an INSERT 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 failed process' transactions.

An application can handle non-fatal errors by modifying its actions or, in some cases, by 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:

- A checkpoint failure
- The use of a deprecated TimesTen feature
- The truncation of some data
- The execution of a recovery process upon connect

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

Recovery

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

- Every connection to the data store is invalidated, a new memory segment is allocated and applications are required to disconnect.
- The data store is recovered from the checkpoint and 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 log files exist and **AutoCreate** is set, TimesTen creates an empty data store.

2

Compiling and Linking TimesTen Applications

This chapter describes how to compile and link TimesTen applications for UNIX and Windows. TimesTen includes source code and makefiles for the demo applications. For more information, see "Building and running the demo applications" in the *Oracle TimesTen In-Memory Database Installation Guide*.

Linking options

There are three options for linking TimesTen applications. An application can link:

- Directly with the TimesTen ODBC driver.
- Directly with the TimesTen Client ODBC driver.
- With an ODBC driver manager.

Linking directly with a TimesTen ODBC Driver

Applications that only need to 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 of 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 to multiple DSNs that use various ODBC drivers at the same time. 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 can only be used in applications that are linked directly to the TimesTen Data Manager ODBC driver. It cannot be used with applications that link to a driver manager. For more information on using TT_PREFETCH_COUNT option, see "Fetching multiple rows of data" on page 19.
- Applications that use Unicode values must take special steps when calling SQLBindCol and SQLBindParameter for NCHAR and NVARCHAR parameters and results. For more information on using Unicode, see the chapter "Working with Unicode" in the *Oracle TimesTen In-Memory Database API and SQL Reference Guide*.
- Applications cannot set or reset the TimesTen-specific TT_PREFETCH_CLOSE connection option. This is useful in certain SELECT queries for applications that intend to use the TimesTen Client ODBC driver. For more information on using the TT_PREFETCH_CLOSE connection option, see "Fetching multiple rows of data" on page 19.
- Driver managers are not available by default on most non-Windows platforms. UNIX developers who wish to use a driver manager must obtain one from a third party.
- 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 do not need to specify the location of the ODBC include files. These files are included with Microsoft Visual C++. However, if your application uses any of the TimesTen include files, you must indicate the location of those 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

```
CFLAGS = "/Iinstall_dir\include"
LIBSDM = ODBC32.LIB
LIBS = TTEN60.LIB TTDV60.LIB
LIBSDEBUG = TTEN60D.LIB TTDV60D.LIB
LIBSCS = TTCL60.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)
```

Testing link options

To test whether an application was directly linked, the application can call **SQLGetInfo** to determine the driver release of an SQLHDBC (data store connection handle) and compare it with the SQLHDBC returned from **SQLAllocConnect**. For example:

Example 2.2

```
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 */
}
```

For direct-linked applications, the call to SQLGetInfo returns the unchanged SQLHDBC. For applications that use the driver manager, the returned SQLHDBC will be different from the passed-in SQLHDBC.

Compiling and linking UNIX applications

UNIX

On UNIX platforms:

- Compile TimesTen applications using the TimesTen header files from the TimesTen installation directory, and
- 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 ODBC types of ULONG, SLONG, USHORT or SSHORT must specify the TT_USE_ALL_TYPES pre-processor option while compiling. This is typically done using the C compiler option -DTT_USE_ALL_TYPES.

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

-I*install_dir*/include

To link with the TimesTen Data Manager ODBC driver, use the following command (where *install_dir* is the installation directory path):

-L*install_dir*/lib -ltten

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

To link with the TimesTen Client ODBC driver, use the following command (where *install_dir* is the installation directory path):

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

On Solaris, only WorkShop 5 and 6 are supported.

The default TimesTen Client ODBC driver was compiled with and supports Workshop 5. An additional library (`libttclientSW6.so`) is provided for the TimesTen Client. The library allows you to link an application (compiled with the Sun Workshop 6
C/C++ compiler) directly with TimesTen Client:

```
-Linstall_dir/lib -lttclientSW6
```

Users of Sun Workshop 5 instead should use:

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

You can use the makefile in the *install_dir*`/demo` file or the following code sample as an example for creating your own makefile.

Example 2.3

```
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)
```

Note: To directly link your application to the TimesTen Data Manager debug ODBC driver, substitute `-lttenD` for `-ltten` on the link line.

Note: On Solaris, when compiling with Sun Workshop C/C++ compilers, TimesTen applications must be compiled and linked with the `-mt` option.

3

XLA and TimesTen Event Management

The Transaction Log API (XLA) is a set C-language functions that allow you to implement applications that monitor TimesTen for changes to specified tables in a local data store and 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 additional functions that enable you to build your own custom data replication solution, if the TimesTen replication solutions described in the *TimesTen to TimesTen Replication Guide* *do not meet your needs.*

Note: 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.

The main topics in this chapter are:

- XLA Concepts
- XLA Demos
- Compiling and Linking an XLA Application
- Writing an XLA Event-Handler Application
- Using XLA in Non-Persistent Mode
- Using XLA as a Replication Mechanism
- Other XLA Features

For a complete description of each XLA function, see Chapter 7, "XLA Reference."

XLA Concepts

The basic concepts of XLA are described in Chapter 8, "Event Notification" in the *Oracle TimesTen In-Memory Database Architectural Overview*. This chapter provides a more in-depth overview of the key XLA concepts. The topics described in this section are:

- XLA modes: persistent vs. non-persistent
- How XLA reads records from the transaction log
- About XLA and materialized views
- About XLA bookmarks

XLA modes: persistent vs. non-persistent

TimesTen XLA can be initialized in one of two modes:

- *Persistent mode.* In this mode, XLA obtains update records directly from the transaction log buffer or 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.
- *Non-persistent mode*. In this mode, 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 only be accessed by one reader at a time and all of the buffered data is lost when the computer or data store is shut down.

Whether you access the transaction log in *persistent* or *non-persistent* mode is determined by the function you use to open the TimesTen data store:

- **ttXlaPersistOpen** -- Opens a connection to a TimesTen data store in persistent mode.
- **ttXlaOpenTimesTen** -- Opens a connection to a TimesTen data store in non-persistent mode.

You should use **ttXlaPersistOpen** under most circumstances. The **ttXlaOpenTimesTen** function is provided primarily to ensure backwards compatibility. However, persistent mode is only supported with disk-based logging. If you require diskless logging, open your XLA connection in non-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** functions to interoperate with earlier XLA versions.

How XLA reads records from the transaction log

As applications modify a TimesTen data store, TimesTen generates log records that describe the changes made to the data and other events such as transaction commits.

New log records are always written to the end of the log buffer as they are generated.

When TimesTen has disk-based logging enabled, log records are periodically flushed in batches from the log buffer in memory to log files on disk. When XLA is initialized in persistent mode, the XLA application does not need to be concerned with which portions of the log are on disk or in memory, so 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 XLA applications with 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, log records from the different applications will be interleaved in the log.

XLA transparently extracts all 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 committed.

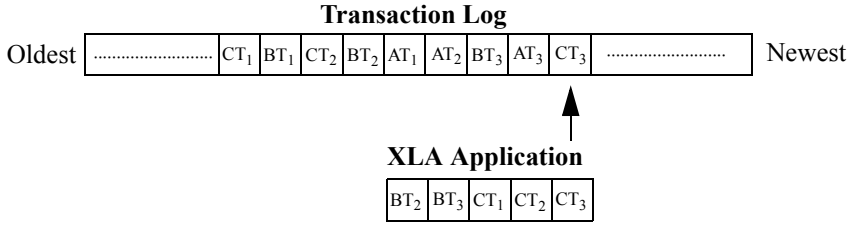
If a change is made but then rolled back, XLA does not deliver the records for the aborted transaction to the application.

Most of these basic XLA concepts are demonstrated in Example 3.1 and summarized in the bulleted list on the bottom.

Example 3.1 Consider the example transaction log illustrated in Figure 3.1.

Figure 3.1

Records extracted from the transaction log



In this example, the transaction log contains the following records:

CT_1 - Application C updates row 1 of table W with value 7.7

BT_1 - Application B updates row 3 of table X with value 2

CT_2 - Application C updates row 9 of table W with value 5.6

BT_2 - Application B updates row 2 of table Y with value XYZ

AT_1 - Application A updates row 1 of table Z with value 3

AT_2 - Application A updates row 3 of table Z with value 4

BT_3 - Application B commits its transaction

AT_3 - Application A rolls back its transaction

CT_3 - Application C commits its transaction

An XLA application that is set up to detect changes to *Tables W, Y,* and *Z* would see:

BT_2 and BT_3 - Update row 2 of table Y with value XYZ and commit

CT_1 - Update row 1 of table W with value 7.7

CT_2 and CT_3 - Update row 9 of table W with value 5.6 and commit

What this example demonstrates:

- *Application B* and *C's* transaction records all appear together.
- Though the records for *Application C* begin to appear in the transaction log before those for *Application B*, the commit for *Application B* (BT_3) appears in the log before the commit for *Application C* (CT_3). As a result, the records for *Application B* are returned to the XLA application ahead of those for *Application C*.
- *Application B's* update to *Table X* (BT_1) aren't presented because XLA is not set up to detect changes to *Table X*.
- *Application A's* updates to *Table Z* (AT_1 and AT_2) are never presented, since it didn't commit and was rolled back (AT_3).

About XLA and materialized views

You can use XLA to track changes to both tables and materialized views. A materialized view provides a single source from which you can track changes to selected rows and columns in multiple *detail* tables. Without a materialized view, the XLA application would have to monitor and filter the update records from all of the detail tables, including records reflecting updates to rows and columns of no interest to the application.

In general, there are no operational differences between the XLA mechanisms used to track changes to a table or a materialized view.

For an overview of how materialized views work with XLA, see Chapter 8, "Event Notification" of the *Oracle TimesTen In-Memory Database Architectural Overview*. For more information on materialized views in general, see "CREATE MATERIALIZED VIEW"in Chapter 13, "SQL Statements" of the *Oracle TimesTen In-Memory Database API and SQL Reference Guide*, and "Understanding materialized views" in the *Oracle TimesTen In-Memory Database Operations Guide*.

About XLA bookmarks

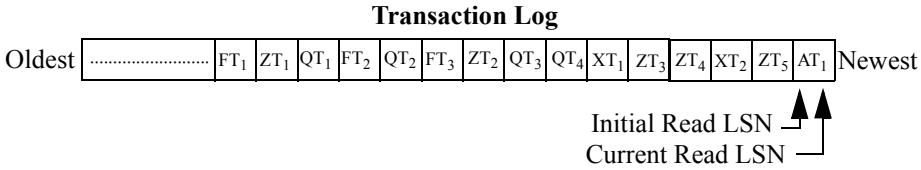
Each reader of a persistent transaction log makes use of a *bookmark* to maintain its position in the log update stream. Each bookmark consists of two pointers that track update records in the transaction log by means of their log sequence numbers (LSNs):

- An *Initial Read LSN* pointer identifies the most recently acknowledged log record (more on "acknowledging" later). Initial Read LSNs are stored in the data store, so they are persistent across data store connections, shutdowns, and crashes.
- A *Current Read LSN* pointer identifies the record currently being read from the log.

As described later in "Initializing XLA and obtaining an XLA handle" on page 45, 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. At this time, the Initial Read LSN associated with the bookmark is read from the data store, cached in the persistent XLA handle (*ttXlaHandle_h*), and designates the reader's "start" position in the transaction log.

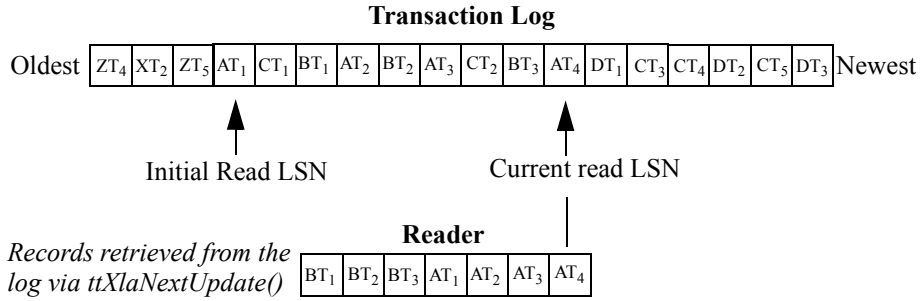
When an application first initializes XLA and obtains an XLA handle, its Current Read LSN pointer is the same as the Initial Read LSN pointer, and they both point to the last record written to the data store, as shown in Figure 3.2.

Figure 3.2 LSN positions upon initializing a persistent XLA handle



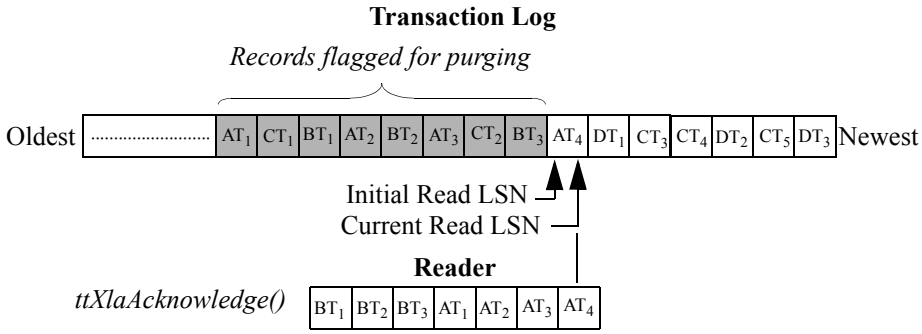
As described later in "Retrieving update records from the transaction log" on page 47, you use the **ttXlaNextUpdate** or **ttXlaNextUpdateWait** function to return a batch of records for committed transactions from the log in the order in which they were committed. Each call to **ttXlaNextUpdate** resets the bookmark's Current Read LSN pointer to the last record read, as shown in Figure 3.3. The Current Read LSN pointer marks the start position for the next call to **ttXlaNextUpdate**.

Figure 3.3 Records retrieved by ttXlaNextUpdate()



You can use the **ttXlaGetLSN** and **ttXlaSetLSN** functions to re-read records, as described in "Changing the location of a bookmark" on page 81. However, calling the **ttXlaAcknowledge** function *permanently* resets the bookmark's Initial Read LSN to its Current Read LSN, as shown in Figure 3.4. Once you've called the **ttXlaAcknowledge** function to reset the Initial Read LSN, all previously read transaction records are flagged for purging by TimesTen. Once the Initial Read LSN is reset, you cannot use **ttXlaSetLSN** to go back and re-read any of the previously read transactions.

Figure 3.4 ttXlaAcknowledge resets bookmark



There is no limit to the number of bookmarks created in a TimesTen data store. Each bookmark can only be associated with 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 the connection, (even 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.

XLA Demos

Your TimesTen install directory contains three demos that demonstrate how to use many of the XLA functions described in this chapter:

- *install_dir*`/demo/xlaSimple.c`
- *install_dir*`/demo/xlaNonPersistent.c`
- *install_dir*`/demo/xlaPersistent/*`

Read the *install_dir*`/demo/README` and `install_dir/demo/xlaPersistent/README` files for instructions on how to build and run the demos.

Most of this chapter describes the procedures demonstrated in the `xlaSimple.c` demo, which is the C equivalent of the C++ `xlaSimple.cpp` demo for TTClasses. The purpose of the `xlaSimple.c` demo is to provide an easy-to-understand, hello-world-type introduction to XLA, while the purpose of the xlaPersistent demo is to provide a deeper, more comprehensive understanding of XLA. The purpose of the xlaNonPersistent.c demo is to demonstrate how to use XLA in non-persistent mode and introduce some of the procedures for implementing an XLA-based replication mechanism.

The xlaSimple demo

The example code described in "Writing an XLA Event-Handler Application" on page 42 is based on the `install_dir/demo/xlaSimple.c` demo application, which creates a table, named SCOTT.MYDATA, and reports on updates to the table.

The SCOTT.MYDATA table looks like:

```
Table SCOTT.MYDATA:
  Columns:
   *NAME                          CHAR (30) NOT NULL
    ADDRESS                       VARCHAR (50)
    CUSTNO                        INTEGER
    SERVICE                       NCHAR (20)
    TSTAMP                        TIMESTAMP
    PRICE                         DECIMAL (10,2)
```

See the README file and "Compiling and Linking an XLA Application" on page 41 for information on compiling the `xlaSimple.c` file. Before running the `xlaSimple` executable, create a data store called 'sample' with the following (non-default) DSN settings:

```
PermSize=16 (or bigger)
TempSize=16 (or bigger)
DurableCommits=0
DataStore=SomeDirectory/sample
```

Open a shell (or command prompt) window and execute `xlaSimple` with the command:

```
xlaSimple sample
```

In a separate shell window, start a **ttIsql** session on the 'sample' data store by entering:

```
ttIsql sample
```

At the **ttIsql** command> prompt, enter a few INSERTs to populate the table and then view the XLA output in the `xlaSimple` window. For example:

```
INSERT INTO scott.mydata VALUES ('John C Durant', '21 Chopping Blvd.
Homeville CA  94032', 12341, n'Buy XYZ', sysdate, 67.23);
INSERT INTO scott.mydata VALUES ('Carol Shelly', '56 Franklin St.
Crawling WA 85002', 34256, n'Quick Search', sysdate, .57);
INSERT INTO scott.mydata VALUES ('Stan T Mann', '4332 Crenshaw
Av. Blue Mountain WI 45322', 23417, n'Sell FDC', sysdate, 92.46);
```

Now try a few UPDATEs to change the rows and view the XLA output. For example:

```
UPDATE scott.mydata SET Address = '24 Westpoint Av. Palo Alto
CA  94022' WHERE Name = 'John C Durant';
UPDATE scott.mydata SET Service = n'Sell WQD' WHERE Name = 'Carol
Shelly';
UPDATE scott.mydata SET Price = 1204.32 WHERE Name = 'Stan T
Mann';
```

The xlaPersistent demo

The more complex xlaPersistent demo consists of a number of files. Read the `install_dir/demo/xlaPersistent/README` file for instructions on how to build and run the demo.

The source files for the xlaPersistent demo are:

`initialize.c`	Initializes ODBC and creates a table in a TimesTen data store.
`subscriber.c`	This is the main file of interest. It demonstrates the various uses of XLA functions to observe changes made to a TimesTen data store populated by `initialize.c` and updated by `publisher.c`.
`publisher.c`	This program makes continuous changes to the data store observed by `subscriber.c`.
`common.c` `tprintf.c`	Provide support functions for the demo, but show no XLA techniques.

Compiling and Linking an XLA Application

The general procedures for compiling and linking TimesTen applications are described in Chapter 2, "Compiling and Linking TimesTen Applications." In releases of TimesTen, previous to release 5.1, you needed to link and compile XLA applications separately. This is no longer the case.

If you have a previously developed XLA application, you will need to recompile the application following the general procedures for compiling TimesTen applications.

If you have migrated TimesTen data stores and restored them using the **ttMigrate** -r -rename option, you may need to update your application, since tables which had been subscribed to previously now have different owners.

Note: XLA does not work with driver manager or client/server applications. Do not link XLA applications to a driver manager library or a client/server library.

TimesTen includes the demo XLA source file `xlaSimple.c` directly with the TimesTen Data Manager ODBC production driver. Review the Makefiles in `install_dir/demo` for details on how to build XLA applications on various platforms.

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," the procedures described in this section are applicable to most XLA applications.

The procedures described in this section are:

- TimesTen #include files
- 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
- Terminating an XLA application

The example code in this section is based on the *install_dir*/demo/`xlaSimple.c` demo application. See the README file and "Compiling and Linking an XLA Application" on page 41 for details on how to compile this demo. See "The xlaSimple demo" on page 40 for operational details, such as how to set up a TimesTen data store compatible with this demo and for some example input.

Note: To simplify the code examples, the routine error checking code for each function call has been omitted. See "Handling XLA errors" on page 67 for a complete discussion.

TimesTen #include files

In addition to the standard C libraries, your XLA application must include the following TimesTen-specific files:

Include File	Description
`timesten.h`	TimesTen ODBC include file
`tt_errCode.h`	TimesTen native error codes
`tt_xla.h`	TimesTen XLA include file

Obtaining a data store connection handle

This section reviews the generic TimesTen operation described in "Connecting to and disconnecting from a data store" on page 11.

As with every ODBC application, an XLA application must first initialize ODBC and obtain an environment handle (*henv*). Then 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()** function to allocate an environment handle:

```
rc = SQLAllocEnv(&henv);
```

Pass the address of *hdbc* to the **SQLAllocConnect()** function to allocate a connection handle for the TimesTen data store:

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

Call the **SQLDriverConnect()** function to connect to the data store specified by the connection string (*connStr*), which in this example is passed from the command line:

```
char connStr[256];
char conn_str_out[1024];

rc = SQLDriverConnect(hdbc,
                      NULL,
                     (SQLCHAR*)connStr,
                      SQL_NTS,
                     (SQLCHAR*)conn_str_out,
                      sizeof(conn_str_out),
                      NULL,
                      SQL_DRIVER_NOPROMPT);
```

Note: Once 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()** function to turn auto-commit OFF (it is set to ON by default):

```
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 44, you can initialize XLA and obtain an XLA handle (*xla_handle*) to access the transaction log. Create only one XLA handle per ODBC connection. If your application makes use of multiple XLA reader threads, create a separate XLA handle and ODBC connection for each thread.

As described in "XLA modes: persistent vs. non-persistent" on page 34, TimesTen can be initialized in either persistent or non-persistent mode. This section describes how to initialize XLA in persistent mode, which is the mode used by most XLA applications. If you have special needs that require you to use XLA in non-persistent mode, see "Initializing XLA in non-persistent mode" on page 74.

Before initializing XLA, you need to initialize a bookmark (see "About XLA bookmarks" on page 37) and an XLA handle as type **ttXlaHandle_h**:

```
char bookmarkName [32] ;
strcpy(bookmarkName, "xlaSimple");
ttXlaHandle_h xla_handle = NULL;
```

Pass the *bookmarkName* and address of *xla_handle* to the **ttXlaPersistOpen** function to obtain an XLA handle:

```
rc = ttXlaPersistOpen(hdbc, bookmarkName, XLACREAT, &xla_handle);
```

Note: 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.

The *XLACREAT* option indicates that you wish to create a new bookmark. If a bookmark by that name already exists, then you can call **ttXlaPersistOpen** with the *XLAREUSE* option to reuse the existing bookmark:

```
#include <tt_errCode.h> /* TimesTen error file */

if ( native_error == tt_ErrKeyExists ) {
      rc = ttXlaPersistOpen(hdbc, bookmarkName,
                            XLAREUSE, &xla_handle);
}
```

Note: When initially created, TimesTen configures an *xla_handle* for the same version as the TimesTen release to which the application is linked. If you need to 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 45, you can specify which tables or materialized views you wish to monitor for update events.

The first step is to 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 for changes to the table or materialized view.

Example 3.2 In this example, we track changes to the SCOTT.MYDATA table:

```
#define TABLE_OWNER "SCOTT"
#define TABLE_NAME "MYDATA"
SQLUBIGINT SYSTEM_TABLE_ID = 0;
SQLUBIGINT userID;

rc = ttXlaTableByName(xla_handle, TABLE_OWNER, TABLE_NAME,
                      &SYSTEM_TABLE_ID, &userID);
```

Once you have the table identifiers, you can use the **ttXlaTableStatus** function to enable XLA update tracking to detect changes to the SCOTT.MYDATA table. Setting the *newstatus* parameter to a non-zero 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 and 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 log. These functions impact the position of the application's bookmark in the transaction log, as described in "About XLA bookmarks" on page 37.

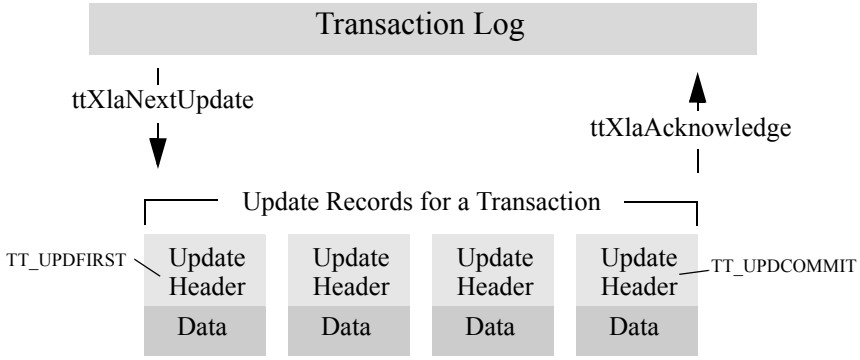
Note: **ttXlaAcknowledge** is an expensive operation and should be used sparingly. See description of **ttXlaAcknowledge** in Chapter 7 for details.

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 3.5 shows a call to **ttXlaNextUpdate** that returns a transaction consisting of four update records from the 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 3.5 Transaction log records



Example 3.3 In the `xlaSimple.c` demo, we continue to monitor our table for updates until stopped by the user.

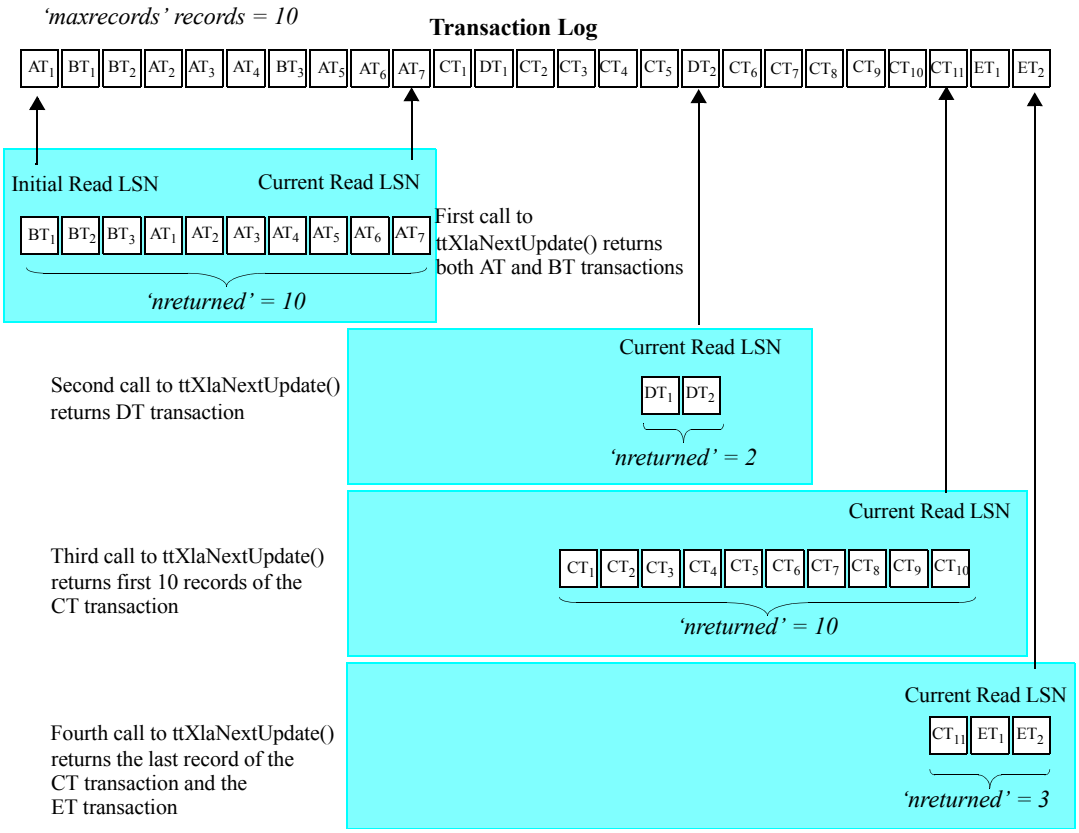
Prior to calling **ttXlaNextUpdateWait**, we initialize a pointer to the buffer to hold the returned **ttXlaUpdateDesc_t** records (*arry*), as well as specify the maximum number of records to be returned from the transaction log (*MAX_RECORDS*) and a variable to hold the actual number of returned records (*records*). Since we're calling **ttXlaNextUpdateWait** we also specify the number of seconds to wait (*FETCH_WAIT_SECS*) if no records are found in the transaction log buffer.

Next we call **ttXlaNextUpdateWait**, passing these values to obtain a batch of **ttXlaUpdateDesc_t** records in *arry.* We then "process" each record in *arry* by passing it to the **HandleChange()** function described in Example 3.4. Once all records are processed, we call **ttXlaAcknowledge** to reset the bookmark position.

```
#define FETCH_WAIT_SECS 5
#define MAX_RECORDS 100

SQLINTEGER records;
ttXlaUpdateDesc_t ** arry;
int j;

while (!StopRequested()) {
    /* Get a batch of update records */
    rc = ttXlaNextUpdateWait(xla_handle, &arry, MAX_RECORDS,
                             &records, FETCH_WAIT_SECS);
      if (rc != SQL_SUCCESS {
        /* See "Handling XLA errors" on page 67 */
      }
    /* 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 67 */
     }
} /* end while !StopRequested() */
```

The actual number of records returned by **ttXlaNextUpdate** or **ttXlaNextUpdateWait**, as indicated by the *nreturned* output parameter, may be less than the value of the *maxrecords* parameter. Figure 3.6 shows an example where *maxrecords* is 10 and the log contains transaction *AT* that is made up of 7 records and transaction *BT* that is made up of 3 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 2 records, and *ET* with 2 records. Because the commit record for the *DT* transaction appears before the *CT* commit record, the next call to **ttXlaNextUpdate** returns the 2 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) and the second batch contains the last *CT* record and the 2 records for the *ET* transaction (assuming no commit record for a transaction following *ET* is detected within the next 7 records).

Figure 3.6 Records retrieved when 'maxrecords=10'



XLA reads records from either a memory buffer or log files on disk, as described in "How XLA reads records from the transaction log" on page 35. 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.

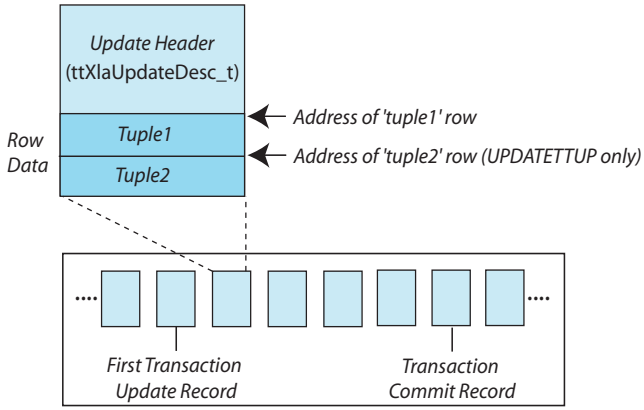
Note: Some XLA applications may improve performance by making the "fetch" and "process record" procedures asynchronous. For example, you can create one thread to fetch and store the records and one or more other threads to process the stored records.

Inspecting record headers and locating row addresses

Now that we have an array of update records and know what type of operation each record represents, let's inspect the returned row data.

Each record returned by the **ttXlaNextUpdate** or **ttXlaNextUpdateWait** function begins with an **ttXlaUpdateDesc_t** header that describes 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); and which columns in the row were updated (if any).

Figure 3.7 Address of row data returned in an XLA update record



The **ttXlaUpdateDesc_t** header has a fixed length and, depending on the type of operation, is followed by zero to two rows (or *tuples*) from the data store. We can locate the address of the first returned row by obtaining the address of the **ttXlaUpdateDesc_t** and adding it to the **sizeof ttXlaUpdateDesc_t**:

```
tup1 = (void*) ((char*) ttXlaUpdateDesc_t +
        sizeof(ttXlaUpdateDesc_t));
```

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. We can locate the address of the second returned row that holds the *after* value 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);
}
```

Example 3.4 In Example 3.3, we pass 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).

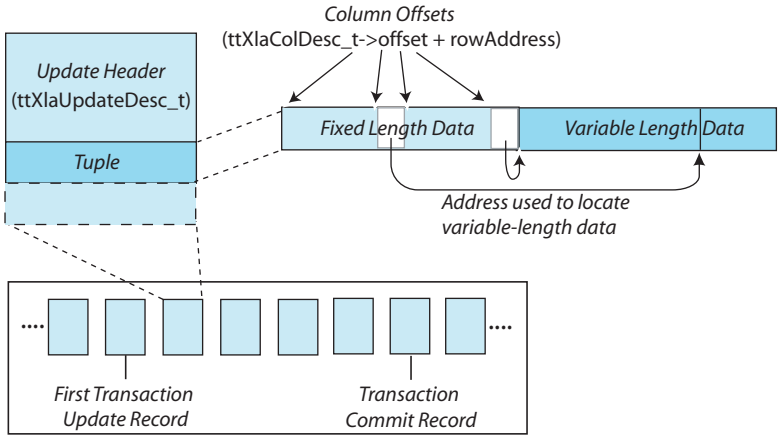
The **HandleChange()** function handles each type of SQL operation differently before calling the **PrintColValues()** function described in Example 3.16.

```
void HandleChange(ttXlaUpdateDesc_t* xlaP)
{
  void * tup1;
  void * tup2;

  tup1 = (void*) ((char*) xlaP + sizeof(ttXlaUpdateDesc_t));

  switch(xlaP->type) {

    case INSERTTUP: /* Handle INSERT operations */
      printf("Inserted new row:\n");
      PrintColValues(tup1);
    break;

    case UPDATETUP: /* Handle UPDATE operations */
      tup2 = (void*) ((char*) tup1 + xlaP->tuple1);
      printf("Updated row:\n");
      PrintColValues(tup1);
      printf("To:\n");
      PrintColValues(tup2);
    break;

    case DELETETUP: /* Handle DELETE operations */
      printf("Deleted row:\n");
      PrintColValues(tup1);
    break;

    default: /* Ignore all other operations */
    break;

  }
}
```

Inspecting column data

As described in "Inspecting record headers and locating row addresses" on page 51, 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 3.8.

Figure 3.8 Column offsets in a row returned in an XLA update 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. (We omitted this call from the discussion in "Initializing XLA and obtaining an XLA handle" on page 45 in order to focus on the details in this section.)

When calling **ttXlaGetColumnInfo**, you 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.

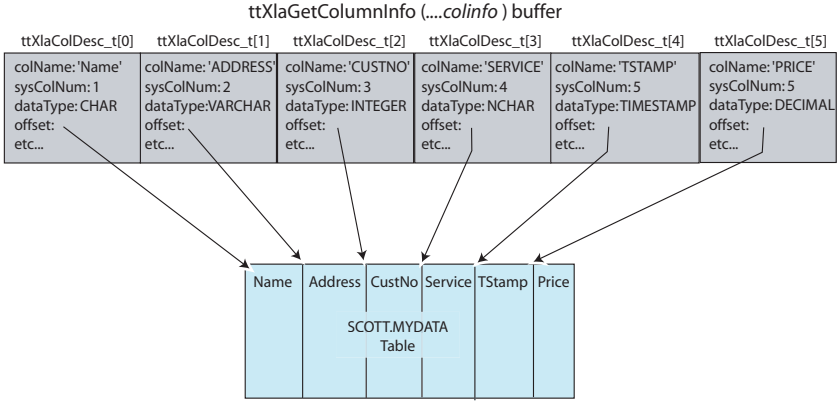
Example 3.5 In the sample code from the `xlaSimple.c` demo below, we guess at the maximum number of returned columns (*MAX_XLA_COLUMNS*), which sets the size of the buffer (*xla_column_defs*) to hold the returned **ttXlaColDesc_t** structures. An alternative and more precise way to set the *maxcols* parameter would be to first call the **ttXlaGetTableInfo** function and use the value returned in **ttXlaTblDesc_t** -> *columns*. (See the **initialize** function in *install_dir*/demo/xlaPersistent/subscriber.c.)

```
#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 67 */
  }
```

As shown in Figure 3.9, the **ttXlaGetColumnInfo** function outputs:

- A list of **ttXlaColDesc_t** structures into the buffer pointed to by the **ttXlaGetColumnInfo** *colinfo* parameter (named *xla_column_defs*).
- An *nreturned* value (named *ncols*) that holds the actual number of columns returned in the *xla_column_defs* buffer.

Figure 3.9 ttXlaColDesc_t structs returned by ttXlaGetColumnInfo()

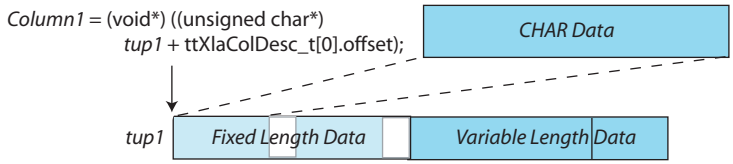


As shown in Figure 3.9, each **ttXlaColDesc_t** structure returned by **ttXlaGetColumnInfo** includes an *offset* value that describes the offset location of that column. How you use this *offset* value to read the column data depends on whether the column contains fixed-length data (such as CHAR, NCHAR, 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 3.10 Locating fixed-length data in row



Example 3.6 The first column in the SCOTT.MYDATA table is of type CHAR. If you use the address of the *tup1* row obtained earlier in the **HandleChange()** function (Example 3.4) and the *offset* from the first **ttXlaColDesc_t** structure returned by the **ttXlaGetColumnInfo** function (Example 3.5), you can obtain the value of the first column with:

```
char * Column1;
Column1 = ((unsigned char*) tup1 + xla_column_defs[0].offset);
```

Note: Strings returned by XLA are not null-terminated. See "Null-terminating returned strings" on page 58 for information on how to null terminate strings.

Example 3.7 The third column in the SCOTT.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 (the data is guaranteed to be aligned properly):

```
int Column3;

Column3 = *((int*) ((unsigned char*) tup +
           xla_column_defs[2].offset));
```

Example 3.8 The fourth column in the SCOTT.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:

```
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 3.16.

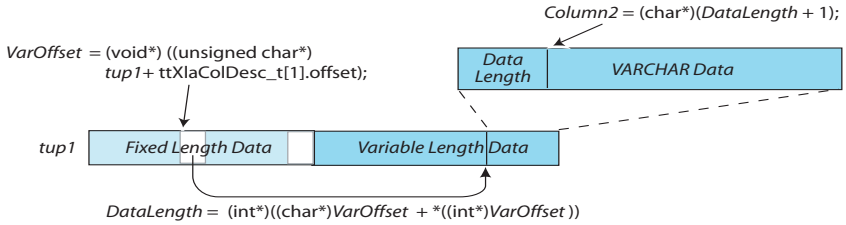
See Chapter 8, "Working with Unicode" for more information on the SQL_WCHAR data type and accessing Unicode data.

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 60.

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 3.11 Locating NOT INLINE variable-length data in row



Example 3.9 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 3.11. 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 done on a 32-bit platform, the first 4 bytes at the *DataLength* location is the length of the data. The next byte after *DataLength* is the beginning of the actual data (*Column2*).

Note: This example assumes the operation is done 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 VARCHARs. 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 3.16.

Null-terminating returned strings

Strings returned from record row data are not null terminated. You can null-terminate a string by copying it into a buffer and adding a NULL character 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 below in Example 3.10. Example 3.11 describes the procedure for null-terminating variable-length strings of a known size, and Example 3.12 describes the procedure for strings of an unknown size.

Example 3.10 To null-terminate the fixed-length CHAR(10) *Column1* string returned in Example 3.6, establish a buffer large enough to hold the string + 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. You can now use the contents of the buffer (here, we print the string):

```
char buffer[10+1];
int size;

size = xla_column_defs[0].size;
memcpy(buffer, Column1, size);
buffer[size] = (char)NULL;

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 56.

Example 3.11 If the *Column2* string obtained in Example 3.9 is a VARCHAR(32), establish a buffer large enough to hold the string + NULL character. Use the value located at the *DataLength* offset to determine the size of the string:

```
char buffer[32+1];

memcpy(buffer, Column2, *DataLength);
buffer[*DataLength] = (char)NULL;

printf("  Row %s is %s\n",
      ((unsigned char*) xla_column_defs[1].colName),
       buffer);
```

If you're writing general-purpose code to read all data types, you can't 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 3.12.

Example 3.12 If the *Column2* string obtained in Example 3.9 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] = (char)NULL;

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

Complex data types, such as DATE, TIME, and DECIMAL values, are stored in an internal TimesTen format that can be converted to their corresponding ODBC C value using the XLA conversion functions:

Conversion Function	Converts
ttXlaDateToODBCCType	Internal DATE value to an ODBC C value
ttXlaTimeToODBCCType	Internal TIME value to an ODBC C value
ttXlaTimeStampToODBCCType	Internal TIMESTAMP value to an ODBC C value
ttXlaDecimalToCString	Internal DECIMAL value to a string value

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

Example 3.13 If you use the address of the *tup1* row obtained earlier in the **HandleChange()** function (Example 3.4) and the *offset* from the fifth **ttXlaColDesc_t** structure returned by the **ttXlaGetColumnInfo** function (Example 3.5), 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. In this example, we 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 67 */
  }
printf("  %s: %02d-%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);
```

Example 3.14 If you use the address of the *tup1* row obtained earlier in the **HandleChange()** function (Example 3.4) and the *offset* from the sixth **ttXlaColDesc_t** structure returned by the **ttXlaGetColumnInfo** function (Example 3.5), 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. In this example, we 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 67 */
  }

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 3.15 To check to see if *Column6* is NULL:

```
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

Let's put all of our column inspection code together into a **PrintColValues()** function, as shown in Example 3.16. In this example, we check the **ttXlaColDesc_t** -> *dataType* of each column to locate columns with a datatype of CHAR, NCHAR, INTEGER, TIMESTAMP, DECIMAL, and VARCHAR and print the value. This is just the approach we use in this example. How you identify which columns to read and what you do in response to the results is entirely up to you. For example, another option might be to check the **ttXlaColDesc_t** -> *ColName* values to locate specific columns by name.

Example 3.16

First we check **ttXlaColDesc_t** -> *NullOffset* to see if the column is NULL. Next we check the **ttXlaColDesc_t** -> *dataType* field to determine the data type for the column. For simple fixed-length data (CHAR, NCHAR, and INTEGER), we simply cast the value located at the **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 XLA conversion functions, **ttXlaTimeStampToODBCCType** and **ttXlaDecimalToCString**.

For variable-length data (VARCHAR), we locate the data in the variable-length portion of the row, as described in "Reading NOT INLINE variable-length column data" on page 56.

Note: This function handles CHAR and VARCHAR strings up to 50 bytes in length. NCHAR characters must belong to the ASCII character set.

```
void PrintColValues(void * tup)
{
  SQLRETURN rc;
  SQLINTEGER native_error;

  void * pColVal;
  char buffer[50+1]; /* No strings over 50 bytes */
  int i, j, size;

  for (i = 0; i < ncols; i++) /* ncols from ttXlaGetColumnInfo */
  {
     if (xla_column_defs[i].nullOffset != 0) { /* Is col NULL? */
        if (*((unsigned char*) tup +
            xla_column_defs[i].nullOffset) == 1) { /* If so... */
               printf("  %s: NULL\n",
               ((unsigned char*) xla_column_defs[i].colName));
               continue; /* Skip rest and re-loop */
        }
     }
```

Handle fixed-length data types:

```
/* For INTEGER, recast as int */
if (xla_column_defs[i].dataType == SQL_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 == SQL_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] = (char)NULL;
      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 == SQL_WCHAR) {
      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");
}
```

Handle variable-length data types:

```
/* For VARCHAR, locate value at its varaible-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 == SQL_VARCHAR) {
     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);
}
```

Convert complex data types using 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 == SQL_TIMESTAMP) {
      TIMESTAMP_STRUCT timestamp;
      pColVal = (void*) ((unsigned char*) tup +
                xla_column_defs[i].offset);

      rc = ttXlaTimeStampToODBCCType(pColVal, &timestamp);
         if (rc != SQL_SUCCESS) {
            /* See "Handling XLA errors" on page 67
*/
```

```
        }
      printf("  %s: %02d-%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 == SQL_DECIMAL) {
            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) {
                /* See "Handling XLA errors" on page 67 */
            }
            printf("  %s: %s\n",
                  ((unsigned char*) xla_column_defs[i].colName),
                   decimalData);
      }
   } /* 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, then terminate the program as described in "Terminating an XLA application" on page 70.

Errors can be checked using either their number or *tt_Err* string. For the complete list of possible TimesTen errors (both *tt_Err* string and associated number), see the *install_dir*/include/tt_errCode.h file. For a description of each message, see "List of errors and warnings" in Chapter 6, "Warnings and Errors" of the *Oracle TimesTen In-Memory Database API and SQL Reference Guide*.

This section describes how to handle XLA errors. For information on how to handle ODBC errors, see "Checking for errors" on page 23.

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.

Example 3.17 For example, after calling an XLA function such as **ttXlaTableByName**, you can check to see if the return code is SQL_SUCCESS. If not, then you can call an XLA error-handling function and terminate the application:

```
rc = ttXlaTableByName(xla_handle, TABLE_OWNER, TABLE_NAME,
                      &SYSTEM_TABLE_ID, &userID);
if (rc != SQL_SUCCESS) {
  handleXLAerror (rc, xla_handle, err_buf, &native_error);
  fprintf(stderr,
    "ttXlaTableByName() returns an error <%d>: %s", rc, err_buf);
  TerminateGracefully(1);
}
```

Your XLA error-handling function should repeatedly call **ttXlaError** until all XLA errors are read from the error stack (return code from **ttXlaError** is SQL_NO_DATA_FOUND). If you need to re-read 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 **ttXlaOpenTimesTen** and **ttXlaPersistOpen** are unable to create an XLA handle, they return 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 need to take action on specific XLA errors. These include:

tt_ErrDbAllocFailed	802	T
tt_ErrCacheXlaNotRead	5023	
tt_ErrCacheXLARollbackFailed	5031	
tt_ErrCondLockConflict	6001	T
tt_ErrDeadlockVictim	6002	T
tt_ErrTimeoutVictim	6003	T
tt_ErrPermSpaceExhausted	6220	T
tt_ErrTempSpaceExhausted	6221	T
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	

Note: The errors marked with a 'T' are the transient errors that you might want to respond to by retrying the operation or taking some other specific action.

For a complete description of these and other errors, see the section "List of errors and warnings" in Chapter 6, "Warnings and Errors" of the *Oracle TimesTen In-Memory Database API and SQL Reference Guide*.

Example 3.18

In the `xlaSimple.c` demo, after calling an XLA function such as **ttXlaTableByName**, if the return code is not SQL_SUCCESS, we call the **handleXLAerror** error handling function to retrieve the errors and the **TerminateGracefully** function to terminate the application. (See "Terminating an XLA application" on page 70.)

```
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);
  }
```

The **handleXLAerror** function for the `xlaSimple.c` demo looks like:

```
void handleXLAerror(SQLRETURN rc, ttXlaHandle_h xlaHandle,
                    SQLCHAR * err_msg, SQLINTEGER * native_error)
{
#ifdef _WIN32
  long retLen;
#else
  int retLen;
#endif /* _WIN32 */
  SQLCHAR message[1024];
  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[0];

  while (1)
  {
    int rc = ttXlaError(xlaHandle, &code, message,
                        sizeof(message), &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 = (char*) &err_msg[0] + strlen((char*)err_msg);
  } */ end while loop */
}
```

Terminating an XLA application

When your XLA application has finished reading from the transaction log, you should gracefully exit by "unsubscribing" the tables and/or materialized views being monitored, rolling back uncommitted transactions, and freeing all handles. You may or may not want to delete the XLA bookmark when the program terminates, as described in "Deleting bookmarks" on page 72.

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 (*xla_handle*).

Call the **SQLTransact** function with SQL_ROLLBACK to rollback any uncommitted transaction. Next, call **SQLDisconnect** to close the connection to TimesTen. Finally call **SQLFreeConnect** and **SQLFreeEnv** functions to release the connection (*hdbc*) and environment (*henv*) handles and to free all of the associated memory.

Example 3.19 For example, the **TerminateGracefully** function in the `xlaSimple.c` demo looks like:

```
void TerminateGracefully(int status)
{
  SQLRETURN     rc;
  SQLINTEGER    native_error ;
  SQLINTEGER    oldstatus;
  SQLINTEGER    newstatus = 0;

/* If the table has been subscribed to via XLA, unsubscribe it */

  if (SYSTEM_TABLE_ID != 0) {
    rc = ttXlaTableStatus(xla_handle, SYSTEM_TABLE_ID, 0,
                          &oldstatus, &newstatus);
    if (rc != SQL_SUCCESS) {
      /* See "Handling XLA errors" on page 67 */
    }
    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;
  }

  /* Disconnect from TimesTen entirely. */

  if (hdbc != SQL_NULL_HDBC) {
    rc = SQLTransact(henv, hdbc, SQL_ROLLBACK);
    if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO) {
      /* See "Handling XLA errors" on page 67 */
    }

    rc = SQLDisconnect(hdbc);
    if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO) {
      /* See "Handling XLA errors" on page 67 */
    }

    rc = SQLFreeConnect(hdbc);
    if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO) {
      /* See "Checking for errors" on page 23 */
    }
    hdbc = SQL_NULL_HDBC;
  }

  if (henv != SQL_NULL_HENV) {
    rc = SQLFreeEnv(henv);
    if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO) {
      /* See "Checking for errors" on page 23 */
    }
    henv = SQL_NULL_HENV;
  }

  exit(status);
}
```

Deleting bookmarks

Before terminating your XLA application, you may want to use the **ttXlaDeleteBookmark** function to delete the XLA bookmark. As described in "About XLA bookmarks" on page 37, a bookmark may be reused by a new connection after its previous connection has closed. In this way, the new connection can resume reading from the transaction log from where the previous connection stopped. The pros and cons of deleting the bookmark are:

- If you delete the bookmark, subsequent checkpoint (**ttCkpt** or **ttCkptBlocking**) operations 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 log files and consume a lot of disk space.

Note: When you reuse a bookmark, you start with the "Initial Read LSN" in the log file. To ensure a connection that reuses a bookmark will begin 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.

Example 3.20 The **InitHandler** function in the `xlaSimple.c` demo deletes the XLA bookmark upon exit:

```
if (deleteBookmark) {
    ttXlaDeleteBookmark(xla_handle);
    if (rc != SQL_SUCCESS) {
       /* See "Handling XLA errors" on page 67 */
    }
    xla_handle = NULL; /* Deleting the bookmark has the */
                       /* effect of disconnecting from XLA. */
}
/* Close the XLA connection, as described in "Terminating an
XLA application" on page 70. */
```

Using XLA in Non-Persistent Mode

As discussed in "XLA modes: persistent vs. non-persistent" on page 34, XLA can be initialized in non-persistent mode. For most purposes, you should use XLA in persistent mode. However, you must use XLA in non-persistent mode if you are reading records from a data store with diskless logging enabled (**Logging** = 2).

The main differences between non-persistent mode and persistent mode are that in non-persistent mode:

- you initialize XLA by calling the **ttXlaOpenTimesTen** function.
- transaction update records are maintained in a transient staging buffer, rather than obtained directly from transaction log buffer or log file on disk.
- if the staging buffer becomes full, transactions cannot complete until you empty the buffer.
- you don't use XLA bookmarks.
- you must configure the size of the staging buffer by using **ttXlaConfigBuffer**.
- you can check the status of the staging buffer using the **ttXlaStatus** function and reset the staging buffer status by calling the **ttXlaResetStatus** function.
- only one XLA application can read from the staging buffer at any point in time.

The procedures for operating XLA in non-persistent mode are described in the sections:

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

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 42.

Note: The example code in this section is based on the *install_dir*/demo/xlaNonPersistent.c demo application.

Initializing XLA in non-persistent mode

After initializing ODBC and obtaining an environment, connection, and statement handle as described in "Obtaining a data store connection handle" on page 44, you can initialize XLA in non-persistent mode and obtain an XLA handle (*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 point in 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 45, only you don't need 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 choosing the size of your staging buffer, consider that, if the buffer is set too small, TimesTen updates will exhaust the buffer, causing further updates to be rejected. Conversely, over allocating space for the buffer wastes memory.

To set the size of the staging buffer, specify a value for the **ttXlaConfigBuffer** *newSize* parameter and NULL for the *oldSize* parameter. Upon return, *oldSize* contains the previous size of the staging buffer.

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

```
SQLUBIGINT  oldsz, newsz
newsz = 400000;

rc = ttXlaConfigBuffer(xla_handle, &oldsz, &newsz);

printf("The old size of the XLA buffer is %d\n",(int)oldsz);
printf("The new size of the XLA buffer is %d\n",(int)newsz);
```

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. 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.

Note: 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.

You can also use the **ttXlaConfigBuffer** function to retrieve the current size of the staging buffer. To obtain the current buffer size, specify NULL for *newSize*. The current size of the staging buffer is returned in *oldSize*. For example:

```
rc = ttXlaConfigBuffer(xla_handle, &oldsz, NULL);
printf("The old size of the XLA buffer is %d\n",(int)oldsz);
```

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

```
newsz = 0;
rc = ttXlaConfigBuffer(xla_handle, NULL &newsz);
```

Note: If the XLA staging buffer is set to a non-zero 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.

Retrieving and resetting 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. This information is encoded in the **ttXlaStatus_t** data type, which 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.

For example, to get the current XLA status:

```
ttXlaStatus_t s;
rc = ttXlaStatus(xla_handle, &s);
```

You can use the **ttXlaResetStatus** function to reset the value of the **ttXlaStatus_t** -> *xlabufminfree* field. For example:

```
rc = ttXlaResetStatus(xla_handle);
```

The **ttXlaStatus_t** -> *xlabufminfree* value is the minimum number of free bytes in the transaction log buffer and is a useful statistic should 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** to null *xlabufminfree* and, at some later time, call **ttXlaStatus** to obtain a new minimum value before calculating the optimum *newSize* value to pass to the **ttXlaConfigBuffer** function.

Using XLA as a Replication Mechanism

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

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** or **ttXlaOpenTimesTen** function to initialize the XLA handles, as described in "Initializing XLA and obtaining an XLA handle" on page 45 and "Initializing XLA in non-persistent mode" on page 74.

Once the XLA handles have been initialized for either data store, follow the procedures:

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

Note: The example code in this section is based on the *install_dir*/demo/xlaNonPersistent.c demo application. Be aware, however, that the xlaNonPersistent.c demo is very basic and does not include the error handling code that would be necessary in a production application of this type.

Checking table compatibility between data stores

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

Use the **ttXlaTableByName**, **ttXlaGetTableInfo**, and **ttXlaGetColumnInfo** functions to return a **ttXlaTblDesc_t** and **ttXlaColDesc_t** descriptions for each table you wish to replicate (these operations are described in "Specifying which tables to monitor for updates" on page 46 and "Obtaining column descriptions" on page 53). You can then pass these descriptions to the **ttXlaTableCheck** function. The output parameter, *compat*, specifies whether or not the tables are compatible (a value of 1 indicates compatibility and 0 indicates non-compatibility). You can then take the desired action, based on the results.

For example:

```
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 */
}
```

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.

For example:

```
int j;
SQLSMALLINT cbConnStrOut;
int records;
ttXlaUpdateDesc_t ** arry;

do {
    /* get up to 15 updates */
    rc = ttXlaNextUpdate(xla_handle,&arry,15,&records);
    if (rc != SQL_SUCCESS) {
      /* See "Handling XLA errors" on page 67 */
    }

    /* 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(xla_handle, p, 0);
      if (rc != SQL_SUCCESS){
     /* See "Handling XLA errors" on page 67 and */
     /* "Handling timeout and deadlock errors" on page
79 */
      }
    }

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

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

Handling timeout and deadlock errors

The return code from **ttXlaApply** indicates whether or not the update was successful. If the return code is != SQL_SUCCESS, then the update may have encountered a transient problem (deadlock or timeout) or a persistent problem. In this case, 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 re-executed.

In the event that **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 either use the **ttXlaRollback** function to roll the transaction back or **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 looking for commit records in the record data stream (**ttXlaUpdateDesc_t** -> *type* = COMMITONLY). Should you encounter an error that requires you to roll back a transaction, call **ttXlaRollback** to rollback the records already applied to the subscriber data store, then call **ttXlaApply** to reapply all the rolled back records stored in your buffer.

Note: If operating XLA in persistent mode, an alternative to buffering the transaction records in a user buffer is to call **ttXlaGetLSN** to get the log sequence number of each commit record in the transaction log, as described in "Changing the location of a bookmark" on page 81. Should 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 Chapter 8, "Conflict Resolution and Failure Recovery" of the *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** accepts an **ttXlaUpdateDesc_t** record as a parameter and output its SQL equivalent into a buffer.

For example, to translate a record (*record*) and store the resulting SQL output in a 200-character buffer (*buffer*):

```
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");
    }
}
```

The actual size of the buffer is returned in the *actualLength* parameter.

Note: **ttXlaGenerateSQL** cannot generate a SQL statement for a table that has been dropped or altered since the record was generated.

Other XLA Features

XLA provides other functions to accomplish tasks, such as:

- Changing the location of a bookmark
- Passing application context

Changing the location of a bookmark

At any point during the connection, you can call the **ttXlaGetLSN** function to query the system for the current read LSN. Should you need to replay a set of updates, you can use the **ttXlaSetLSN** function to reset the read LSN to any valid value "larger" than the initial read LSN set by the last **ttXlaAcknowledge**. In this context, "larger" only applies if the LSNs being compared are from records in the same transaction. If the LSNs being compared are for records belonging to different transactions, then any LSN from a transaction that committed before another transaction is the "smaller" LSN, even if the numerical value of the LSN is larger. The only way to enable the initial read LSN to move forward to the current read LSN is by calling the **ttXlaAcknowledge** function, which indicates that you have received and processed all log records up to the current read LSN. Once you have called **ttXlaAcknowledge** on a particular bookmark, you can no longer access log records with an LSN "smaller" than the current read LSN.

Passing application context

Though it is not an XLA function, writers to the log can call **ttApplicationContext** to pass binary data associated with an application to XLA readers. The **ttApplicationContext** 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 56.

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 re-established before each update.

To set the context:

1. Declare two program variables for invoking the **ttApplicationContext** function. 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 function:

```
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 needs to 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 lost. Should this happen, the application can call **ttApplicationContext** again before the next SQL operation.

4

Distributed Transaction Processing: XA

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

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 BEA's 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 X/Open Company Ltd.
- BEA documentation available from the BEA Systems web site:

  `http://edocs.bea.com/`

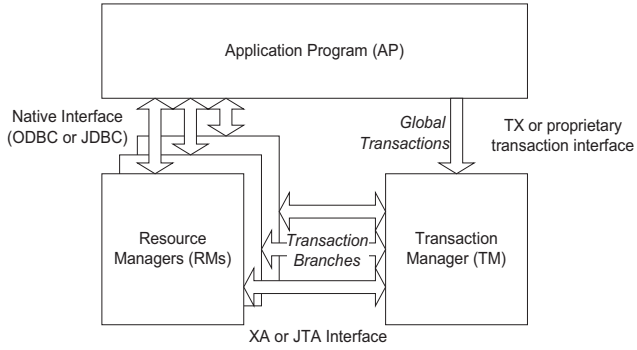
The main topics in this chapter are:

- "X/Open DTP Model" on page 84
- "Setting TimesTen data store attributes for XA" on page 85
- "Recovering Global Transactions" on page 86
- "Using the XA API" on page 87
- "Error Handling" on page 98

X/Open DTP Model

Figure 4.1 illustrates the interfaces defined by the X/Open DTP model.

Figure 4.1 Distributed transaction processing model



In the DTP model, a *transaction manager* breaks each *global transaction* down into multiple *branches* and distributes them to separate *resource managers* for service. 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.

As shown in Figure 4.1Figure 4.1, applications use the *TX* interface to communicate global transactions to the transaction manager. The transaction manager breaks the global transaction down into branches and uses the *XA* interface to coordinate each transaction branch with the appropriate resource manager.

Global transaction control provided by the TX and XA interfaces is distinct from local transaction control provided by the native ODBC and JDBC interfaces. 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 "Obtaining an ODBC handle from an XA connection" on page 89 for more information.

Two-phase commit

In an XA implementation, the transaction manager commits the distributed branches of a global transaction by means of a *two-phase commit* protocol. The two phases are:

- In *phase one* of the commit, 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 is unable to commit its branch, the transaction manager *rolls back* the entire transaction in phase two.
- In *phase two* of the commit, the transaction manager either directs each resource manager to commit its branch or, if a transaction manager reported it was unable to commit in phase one, rolls back the global transaction.

See the optimizations described below for exceptions.

Note: The transaction manager considers the global transaction committed if and only if all branches successfully commit.

One-phase commit optimization

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.

Read-only optimization

If a global transaction branch is read-only, the transaction manager commits the branch in phase one and skips phase two for that branch. A “read-only” transaction is one that does not generate any log records.

Setting TimesTen data store attributes for XA

This section describes how to set your TimesTen data store attributes for TimesTen XA.

DurableCommits

In order 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. Should you need to recover from a failure, both the resource manager and the TimesTen transaction manager will have a consistent view of which transaction branches were active (in a prepared state) at the time of failure.

See “Recovering Global Transactions” on page 86 for more information.

Logging

In order to allow for rollback of transactions, you must have disk-based logging enabled (**Logging**=1).

Operation with diskless or no logging is not allowed with XA.

Logging=0 with XA results in the error:

```
Error 11033 - "XA support requires that logging be enabled"
```

Logging=2 with XA results in the error:

```
Error 11033 "XA support requires that logging to disk be enabled"
```

Recovering Global Transactions

When a TimesTen data store is loaded from disk after a crash or unexpected termination, and recovery takes place, 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. The resolution procedure is accomplished as follows:

Once connect 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 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 or JTA implementation of **xa_recover()**, and then dealt with through XA or JTA **xa_commit()**, **xa_rollback()**, or **xa_forget()**, as appropriate. Once all 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()**.

Note: TimesTen does not allow you to replicate TimesTen resource managers, as failovers to subscribers would result in inconsistent global transactions.

Heuristic recovery

If the transaction manager is unavailable or is unable to resolve an in-doubt transaction, you can use the **ttXactAdmin** utility to independently commit or abort the individual transaction branches. See the section on "ttXactAdmin" in the *TimesTen Reference Guide* for more information.

Using the XA API

The TimesTen implementation of XA provides an API consistent with that specified in *Distributed Transaction Processing: The XA Specification*. This section describes what you need to know when using the TimesTen implementation of XA.

What you need to know	Where to find out
How to set the TimesTen data store attributes when using XA	"Setting TimesTen data store attributes for XA" on page 85
Issues related to using some XA functions with TimesTen XA	"Using XA functions" on page 88
How to make ODBC calls over an XA connection	"Obtaining an ODBC handle from an XA connection" on page 89 and "Calling ODBC functions over an XA connection" on page 91
Configuration of the TimesTen XA resource manager switch	"XA resource manager switch" on page 92
How to link to the TimesTen driver manager extension library	"XA Support through the ODBC driver manager" on page 94
How to recover in-doubt global transactions	"Recovering Global Transactions" on page 86
How to configure XA and TimesTen for use with Tuxedo	"Configuring Tuxedo to use TimesTen XA" on page 95
TimesTen XA errors	"Error Handling" on page 98

Primary Documents

This section provides the TimesTen information to be used with the following documents:

- X/Open CAE Specification, *Distributed Transaction Processing: The XA Specification* published by the X/Open Company Ltd.
- BEA Tuxedo documentation available from the BEA Systems web site:

  `http://edocs.bea.com/`

Using XA functions

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. Tuxedo users can skip this section and refer directly to the procedures described in "Configuring Tuxedo to use TimesTen XA" on page 95.

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()**.

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 (*XID*). The XID is a required parameter for XA functions that manipulate a transaction. Internally, TimesTen maps XIDs to its own transaction identifiers.

Note: The XID defined by XA standard has some of its members (such as, `formatID`, `gtrid_length`, and `bqual_length`) defined as type 'long'. This causes problem when 32-bit client applications connect to a 64-bit server or vice-versa because 'long' is a 32-bit integer on 32-bit platforms and 64-bit integer on 64-bit platforms (except 64-bit Windows). Hence, TimesTen internally uses only the 32 least significant bits of those XID members irrespective 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.

Obtaining an ODBC handle from an XA connection

TimesTen provides a function that allows you to acquire the ODBC connection handle associated with an XA connection opened by **xa_open()**.

tt_xa_context()

Description

This function is used to acquire the ODBC connection handles associated with a 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	int (input)	If the resource manager ID is NULL, return the handles associated with the first connection on this thread. If *rmid* is non-NULL, return the handles associated with the specified *rmid* value.
henv	SQLHENV (output)	The SQLHENV associated with the current **xa_open()** context.
hdbc	SQLHDBC (output)	The SQLHDBC associated with the current **xa_open()** context.

Return Values

0 Success

1 rmid Not Found

-1 Invalid Parameter

Comments

Use this function with a NULL *rmid* to establish context in the application environment where the connection has been opened outside of the scope of the user-written code (the *rmid* is unknown).

Example

In this 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, we use the **tt_xa_context()** function to locate the SQLHENV and SQLHDBC handles allocated by **xa_open()**.

```
do_insert()
{
   SQLHENV henv;
   SQLHDBC hdbc;
   SQLHSTMT hstmt;

   /* retrieve the handles for the current connection */
   tt_xa_context(NULL, &henv, &hdbc);

   /* now we can do our ODBC programming as usual */
   SQLAllocStmt(hdbc, &hstmt);

   SQLExecDirect(hstmt, "insert into t1 values (1)", SQL_NTS);

   SQLFreeStmt(hstmt, SQL_DROP);
}
```

Calling ODBC functions over an XA connection

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()**, as described in "Obtaining an ODBC handle from an XA connection" on page 89.

AUTOCOMMIT

To simplify operation and prevent any 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 error:

```
Error 11030 - "Autocommit must be turned off when working on global
(XA) transactions"
```

Once work on a global transaction branch has begun (**xa_start()** has been called), AUTOCOMMIT may not be turned ON until such work has been completed (through a call to **xa_end()**). Any attempt at turning AUTOCOMMIT ON in this case will result in the error:

```
Error 11030 - "Autocommit must be turned off when working on global
(XA) transactions"
```

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 error:

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

Closing open cursors

Any open statement cursors must be closed using **SQLFreeStmt**(hstmt, SQL_CLOSE) before calling **xa_end()** to end work on a global transaction branch or 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 need 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, named tt_xa_switch.

xa_switch_t

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

```
/*
 * 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, and by omission that 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 Support through the ODBC driver manager

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 to load at connect time (for example, 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, named `ttxadm60.dll` for the XA functions. Applications can make XA calls directly, but must link in the extension library.

To link with the `ttxadm60.dll` library, applications need to include `ttxadm60.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 ttxadm60.lib odbc32.lib
```

Note: The XA driver manager extension is only implemented on NT.

Configuring Tuxedo to use TimesTen XA

Note: Though TimesTen XA has been demonstrated to work with the BEA Tuxedo transaction manager on the Sun Solaris and Windows platforms, TimesTen cannot guarantee the operation of DTP software beyond the TimesTen implementation of XA.

To configure Tuxedo to use the TimesTen resource managers, you need to:

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

The following sections describe each of these steps.

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

Update the $TUXDIR/udataobj/RM file

To integrate the TimesTen XA resource manager into the BEA 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.

UNIX

On UNIX platforms, add:

```
# TimesTen 6.0
TimesTen:tt_xa_switch:-Linstall_dir/TTinstance/lib -ltten
```

WINDOWS

On NT, add:

```
# TimesTen 6.0
TimesTen;tt_xa_switch;install_dir\lib\ttdv60.lib
```

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

Example 4.1

```
# For Unix:
TimesTen:tt_xa_switch:-L/opt/TimesTen/giraffe/lib -ltten
```

Example 4.2

```
# For NT:
TimesTen;tt_xa_switch;C:\TimesTen\giraffe\lib\ttdv6.0d.lib
```

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 NT, 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 the TMSNAME, specify the TMS_TT file created by the **buildtms** command described in "Build the Tuxedo transaction manager server":

```
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:

```
OPENINFO="TimesTen:DSN=DSNname"
```

where *DSNname* is the name of your TimesTen data store.

Note that on NT, 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 4.3 shows the portions of a UBBCONFIG file used to configure two TimesTen resource managers, named GROUP1 and GROUP2.

Example 4.3

```
*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 a BEA Tuxedo ATMI server load module.

On UNIX platforms, enter:

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

On NT, enter:

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

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

Example 4.4 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 4.3.

Example 4.4

```
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
```

Error Handling

The XA specification has a limited, strictly defined set of errors that can be returned from XA interface calls. The ODBC **SQLError()** mechanism 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 Chapter 6, “Warnings and Errors for the complete list of errors.

5

Application Tuning

This chapter describes how to tune your C application to run optimally on a TimesTen data store. For information on data store and SQL tuning, see Chapter 8, "Data Store Performance Tuning in the *Oracle TimesTen In-Memory Database Operations Guide*.

The topics described in this section are:

- Bypass driver manager if appropriate
- Prepare statements in advance
- Avoid unnecessary prepare operations
- Use arrays of parameters to batch operations
- Avoid excessive binds
- Avoid SQLGetData
- Avoid data type conversions
- Reduce contention
- Bulk fetch rows of TimesTen data
- Choose the best method of locking
- Turn off autocommit mode
- Size transactions appropriately
- Choose the appropriate logging options
- Choose a timeout interval
- Use durable commits appropriately
- Avoid transaction rollback
- Avoid frequent checkpoints

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 27. The performance improvement will be approximately 20 percent.

"Testing link options" on page 30 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.

Prepare statements in advance

Use **SQLPrepare** to prepare a statement in advance if it will be executed more than a few times. Make sure that the prepared statements have been committed in order to release locks held by the prepare and to allow the query plan to persist. Your ODBC documentation explains how to prepare commands in advance.

If you have applications that generate a statement multiple times searching for different values each time, use a parameterized statement to reduce compile time. For example, if your application generates statements like:

```
SELECT A FROM B WHERE C = 10
SELECT A FROM B WHERE C = 15
```

You can replace these statements with the single statement:

SELECT A FROM B WHERE C = ?

TimesTen shares prepared commands automatically, once it has been committed. As a result, an application's request to prepare a command for execution may be completed very quickly if a prepared version of the command already exists in the system. Also, repeated requests for **SQLExecDirect** of the same command may be able to avoid the prepare overhead by sharing a previously prepared version of the command.

Even though TimesTen allows prepared commands to be shared, it is still a good practice, for performance reasons, to use parameterized commands. Using parameterized commands can reduce prepare overhead above and beyond what is made available through command sharing.

Avoid unnecessary prepare operations

Because preparing SQL statements is an expensive operation, your application should minimize the number of calls to the **SQLPrepare** function. Most applications prepare a set of commands at the beginning of a connection and use that set for the duration of the connection. This is a good strategy when connections are long, consisting of hundreds or thousands of transactions. But if connections are relatively short, a better strategy is to establish a long-duration connection that prepares the commands and executes them on behalf of all threads or processes. The trade-off here is between communication overhead and prepare overhead, and can be examined for each application. Prepared statements are invalidated when a connection is closed.

Use arrays of parameters to batch operations

The ODBC **SQLParamOptions** function allows an application to specify multiple values for the set of parameters assigned by **SQLBindParameter**. This is useful for bulk inserts and other work that requires the data store to process the same SQL statement multiple times with various parameter values. For example, your application can specify three sets of values for the set of parameters associated with an INSERT statement, and then execute the INSERT statement once to perform the three insert operations.

TimesTen supports the use of **SQLParamOptions** with INSERT, UPDATE and DELETE statements. TimesTen does not support batching of SELECT statements.

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

Avoid excessive binds

The purpose of an **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 and program buffer for a given data column or parameter are 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 since applications have to issue an **SQLGetData** ODBC call for every column of *every* row that is fetched. In contrast, using bound columns requires only one ODBC call per 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. The avoid data type conversions, match input argument types to expression types and also match the types of output buffers to the types of the fetched values.

Reduce contention

Data store contention can substantially impede application performance.

To reduce contention in your application:

- Choose the appropriate locking method. See "Choose the best method of locking" on page 104.
- Distribute data strategically in multiple tables and/or data stores.

If your application suffers a decrease in performance because of lock contention and a lack of concurrency, reducing contention is an important first step in improving performance.

The LOCK_GRANTS_IMMED, LOCK_GRANTS_WAIT and LOCK_DENIALS_COND columns in the SYS.MONITOR table provide some information on lock contention:

- LOCK_GRANTS_IMMED counts how often a lock was available and was immediately granted at lock request time.
- LOCK_GRANTS_WAIT counts how often a lock request was granted after the requestor had to wait for the lock to become available.

If limited concurrency results in a lack of throughput, or if response time is an issue, an application can serialize ODBC calls to avoid contention. This can be achieved by having a single thread issue all those calls. Using a single thread requires some queuing and scheduling on the part of the application, which has to trade off some CPU time for a decrease in contention and wait time. The result is higher performance for low-concurrency applications that spend the bulk of their time in the data store.

Bulk fetch rows of TimesTen data

TimesTen provides an option, named TT_PREFETCH_COUNT, that 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 "Fetching multiple rows of data" on page 19.

Choose the best method of locking

When multiple connections access a data store simultaneously,TimesTen uses locks to ensure that the various transactions operate in apparent isolation. TimesTen supports the isolation levels described in Chapter 7, "Transaction Management and Recovery" in the *Oracle TimesTen In-Memory Database Operations Guide*. It also supports the locking levels: data store-level locking, table-level locking and row-level locking. You can use the **LockLevel** connection attribute to indicate whether data store-level locking or row-level locking should be used. Use the **ttOptSetFlag** procedure to set optimizer hints that indicate whether table locks should be used. The default lock granularity is row locks.

Choose an appropriate lock level

If there is very little contention on the data store, use table-level locking. It provides better performance and deadlocks are less likely. There is generally little contention on the data store when transactions are short and/or there are few connections. In that case, transactions are not likely to overlap.

Table-level locking is also useful when a statement accesses nearly all the rows on a table. Such statements can be queries, updates, deletes or multiple inserts done in a single transaction.

Data store-level locking restricts concurrency more than table-level locking, and is generally useful only for initialization operations, such as bulk-loading, when no concurrency is necessary. It has better response-time than row-level or table-level locking, at the cost of diminished throughput.

Row-level locking is generally preferable when there are many concurrent transactions that are not likely to need access to the same row.

Choose an appropriate isolation level

When using row-level locking, applications can use the ODBC function **SQLSetConnectOption** to choose to run transactions at the SERIALIZABLE or READ_COMMITTED isolation level. The default isolation level is READ_COMMITTED. You can use the Isolation connection attribute to specify one of these isolation levels for new connections.

When running at SERIALIZABLE transaction isolation level, TimesTen holds all locks for the duration of the transaction, so:

- Any transaction updating a row blocks writers until the transaction commits.
- Any transaction reading a row blocks out writers until the transaction commits.

When running at READ_COMMITTED transaction isolation level, TimesTen only holds update locks for the duration of the transaction, so:

- Any transaction updating a row blocks out readers and writers of that row until the transaction commits.
- Phantoms are possible. A phantom is a row that appears during one read but not during another read, or appears in modified form in two different reads, in the same transaction, due to early release of read locks during the transaction.

You can determine if there is an undue amount of contention on your system by checking for time-out and deadlock errors (errors # 6001, 6002 and 6003). Information is also available in the LOCK_TIMEOUTS and DEADLOCKS columns of the SYS.MONITOR table.

Turn off autocommit mode

The AUTOCOMMIT mode forces a commit after each statement. AUTOCOMMIT mode is enabled by default. Committing each statement after execution can have a significant negative impact on performance. For performance-sensitive applications, you may want to set AUTOCOMMIT to off (see "Choose a timeout interval" on page 108). Your ODBC documentation describes how to disable this feature using the ODBC **SQLSetConnectOption** function.

The XACT_COMMITS column of the SYS.MONITOR table indicates the number of transaction commits.

Note: If you do not include any explicit commits in your application, the application can use up important resources unnecessarily, including memory and locks. All applications should do periodic commits.

Size transactions appropriately

Each transaction that generates log records (for example a transaction that does an INSERT, DELETE or UPDATE) by default incurs a disk write when the transaction commits. (See "Choose the appropriate logging options" on page 107.) Disk I/O affects response time and may affect throughput, depending on how effective group commit is.

Performance-sensitive applications should avoid unnecessary disk writes at commit. Use a performance analysis tool to measure the amount of time your application spends in disk writes (versus CPU time). If there seems to be an excessive amount of I/O, there are two steps you can take to avoid writes at commit:

- Adjust the transaction size.
- Adjust whether disk writes are performed at transaction commit. See "Use durable commits appropriately" on page 107.

Long transactions perform fewer disk writes per unit time than short transactions. However, long transactions also can reduce concurrency, as discussed in Chapter 7, "Transaction Management and Recovery" in the *Oracle TimesTen In-Memory Database Operations Guide*.

- If only one connection is active on a data store (for example, if it is an exclusive connection), longer transactions could improve performance. However, long transactions may have some disadvantages, such as longer rollbacks.
- If there are multiple connections, there is a trade-off between log I/O delays and locking delays. In this case transactions are best kept to their natural length, as determined by their requirements for atomicity and durability.

Use durable commits appropriately

This practice ensures that no committed transactions are lost because of system or application failures. Applications can avoid some or all of these disk writes by performing non-durable commits. Non-durable commits do everything that a durable commit does except write the transaction log to disk. Locks are released and cursors are closed, but no disk write is performed.

Note: Some controllers or drivers only write data into cache memory in the controller or write to disk some time after the operating system is told that the write is completed. In these cases, a power failure may cause some information that you thought was durably committed to be lost. To avoid this loss of data, configure your disk to write to the recording media before reporting completion or use an Uninterruptable Power Supply (UPS).

The advantage of non-durable commits is a potential reduction in response time and increase in throughput. The disadvantage is that some transactions may be lost in the event of system failure. An application can force the log to disk by performing an occasional durable commit or checkpoint, thereby decreasing the amount of potentially lost data. In addition, TimesTen itself periodically flushes the log to disk when internal buffers fill up, limiting the amount of data that will be lost.

Transactions can be made durable or can be made to have delayed durability on a connection-by-connection basis. Applications can force a durable commit of a specific transaction by calling the **ttDurableCommit** procedure.

Applications that do not use non-durable commits can benefit from using synchronous writes in place of write and flush. To turn on synchronous writes set **LogFlushMethod**=**2**.

The XACT_D_COMMITS column of the SYS.MONITOR table indicates the number of transactions that were durably committed.

Choose the appropriate logging options

If you set diskless logging, no log files are generated entire transactions may be rolled back, if the log buffer is not big enough. Hence transaction durability is not guaranteed by TimesTen, when diskless logging is used. Your application can guarantee durability by using checkpoints. In general, use diskless logging with caution as it may cause an excessive number of transaction aborts.

For some applications, lost transactions can be tolerated. For example, it may be possible to regenerate the data from another source in the event of a system or application failure, or the application may take checkpoints at strategic intervals to save data where needed. In these cases, it may be advantageous to turn off logging when connecting to the data store, as described in Chapter 1, “Data Store

Attributes in the *Oracle TimesTen In-Memory Database API and SQL Reference Guide*. However, this can result in a lack of atomicity, as described in Chapter 7, "Transaction Management and Recovery" in the *Oracle TimesTen In-Memory Database Operations Guide*.

Logs are also used to roll back transactions during error handling, as well as to redo transactions in case of application or system failure. If logging is turned off, most statements execute atomically, but entire transactions cannot be rolled back. For statements that do not execute atomically, TimesTen returns an error. If an application is in development or if it is susceptible to frequent rollbacks for other reasons, turning off logging may generate these errors, which may cause the data store to become inconsistent. In this case, it may be preferable to use diskless logging or logging to disk. To use non-durable commits, see "Use durable commits appropriately" on page 107.

Additional facts about logging are:

- Logging can be set to different values on a connection basis, but all concurrent connections must agree on the **Logging** attribute setting.
- Logging is required to use row-level locking.
- If logging is turned off or if diskless logging is enabled, you should include periodic non-durable commits in your application (see "Choose a timeout interval" on page 108). Durable commits are not permitted. (Because there is no log to write to disk, durability must be achieved through checkpoints.) If Logging is disabled, durable commits are not possible. If Logging is turned off, operations that are not atomic return an error or warning when the TimesTen Data Manager cannot restore the data store to its state prior to a failed operation.
- Logging to disk or diskless logging is required to cache Oracle tables.

Choose a timeout interval

To change the time-out interval for locks, use the built-in procedure **ttLockWait**. By default connections wait 10 seconds to acquire a lock. For some applications this interval may be either too long or too short.

Avoid transaction rollback

When transactions fail due to erroneous data or application failure, they are rolled back by TimesTen automatically. In addition, applications often explicitly rollback transactions using the **SQLTransact** function to recover from deadlock or time-out conditions. This is not desirable from a performance point of view: a rollback consumes resources and the entire transaction is in effect wasted.

Applications should avoid unnecessary rollbacks. This may mean designing the application to avoid contention (see "Choose the best method of locking" on

page 104) and checking application or input data for potential errors before submitting it, if possible. The XACT_ROLLBACKS column of the SYS.MONITOR table indicates the number of transactions that were rolled back.

Avoid frequent checkpoints

Applications that are connected to a data store for a long period of time occasionally need to checkpoint the data store explicitly so that log files do not fill up the disk. Transaction-consistent checkpoints can have a significant performance impact because they require exclusive access to the data store.

Occasional fuzzy checkpoints may be used instead of transaction consistent (blocking) checkpoints. They may take longer, but they permit other transactions to operate against the data store at the same time and thus have less overhead. You can increase the interval between successive checkpoints by increasing the amount of disk space available for accumulating log files.

As the log increases in size (if the interval between checkpoints is large), recovery time increases accordingly. If reducing recovery time after a system crash or application failure is important, frequent checkpoints may be preferable. The DS_CHECKPOINTS and DS_CHECKPOINTS_FUZZY columns of the SYS.MONITOR table indicates how often checkpointing has successfully been completed.

Part II: TimesTen C Reference Guide

6

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 the TimesTen Utilities chapter.

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 `ttdv60.lib` and `tten60.lib` on Windows) and the TimesTen utility library (`libttutil` on UNIX and `ttut60.lib` on Windows platforms).

Note: Applications must call **ttUtilAllocEnv** prior to calling any other TimesTen utility library function. In addition, applications must call **ttUtilFreeEnv** 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 **ttUtilGetError** to retrieve all actual error or warning information.

Utility API list

Function	See
ttBackup	page 115
ttDestroyDataStore	page 119
ttDestroyDataStoreForce	page 121
ttRamGrace	page 123
ttRamLoad	page 124
ttRamPolicy	page 125
ttRamUnload	page 127
ttRepDuplicate	page 128
ttRepDuplicateAcc	page 128
ttRepDuplicateEx	page 129
ttRestore	page 133
ttUtilAllocEnv	page 135
ttUtilFreeEnv	page 137
ttUtilGetError	page 139
ttUtilGetErrorCount	page 142
ttXactIdRollback	page 144

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 "To rename the owner of tables in a data store, use the ttMigrate utility. When restoring tables, you can use the -rename option to rename the owner of tables. Backing up and restoring a data store" in the *Oracle TimesTen In-Memory Database Operations Guide*.

Access Control

If Access Control is enabled, requires ADMIN privileges or data store object ownership.

Syntax

```
ttBackup (ttUtilHandle handle, const char *connStr,
          ttBackUpType type, ttBooleanType atomic,
          const char *backupDir, const char *baseName,
          ttUtFileHandle stream);
```

Parameters

ttBackup has these 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.

type	ttBackUpType	Specifies 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 if 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 log files from accumulating for an incremental backup.
atomic	ttBooleanType	Specifies whether an existing backup in the specified directory should be destroyed before creating a new backup. **TT_FALSE -** The existing backup in the specified directory is destroyed before the new backup begins. Otherwise, an attempt to create a full backup of a data store into a directory that already contains a backup of the data store is allowed to proceed and does not destroy the existing backup until the new backup is complete. This parameter only has an effect on full file backups. It is ignored for incremental backups and stream backups.

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, specifies the directory portion of the backup path that is to be disabled. For TT_BACKUP_INCREMENTAL_STOP or a file backup, if NULL is specified an error is returned.
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 filename portion of the DataStore attribute of the data store's ODBC definition. For TT_BACKUP_INCREMENTAL_STOP, 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 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 backup the data store for a DSN, “payroll” into `C:\backup`, use:

```
ttUtilHandleutilHandle;
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.

When ttBackup is called by an application to back up a data store that uses diskless logging or has logging turned off, the backup file reflects only those transactions that committed before the most recent checkpoint. Transactions that committed after the most recent checkpoint are not reflected in the backup.

You cannot backup temporary and diskless data stores.

See also “ttRestore” on page 133

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. But, it does not delete the DSN itself defined in the odbc.ini file on the supported UNIX platforms or, in Window's registry on the supported Windows platforms.

Access Control

If Access Control is enabled, requires ADMIN privileges or data store object ownership.

Syntax

```
ttDestroyDataStore (ttUtilHandle handle, const char *connStr,
                    unsigned int timeout);
```

Parameters

ttDestroyDataStore has these 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 DSN, “payroll”, consisting of files `C:\dsns\payroll.ds0`, `C:\dsns\payroll.ds1`, and several log files `C:\dsns\payroll.log`*n*, call:

```
char   errBuff [256];
int    rc;
unsigned intretCode;
ttUtilErrTyperetType;
ttUtilHandleutilHandle;

……————————

……————————

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. But, it does not delete the DSN itself defined in the odbc.ini file on the supported UNIX platforms or, in Window's registry on the supported Windows platforms.

Access Control

If Access Control is enabled, requires ADMIN privileges or data store object ownership.

Syntax

```
ttDestroyDataStoreForce (ttUtilHandle handle, const char
                    *connStr, unsigned int timeout);
```

Parameters

ttDestroyDataStoreForce has these 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 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 DSN, “payroll”, consisting of files `C:\dsns\payroll.ds0`, `C:\dsns\payroll.ds1`, and several log files `C:\dsns\payroll.log`*n*, call:

```
char   errBuff [256];
int    rc;
unsigned intretCode;
ttUtilErrTyperetType;
ttUtilHandleutilHandle;

……————————

……————————

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.

Access Control

If Access Control is enabled, requires ADMIN privileges or data store object ownership.

Syntax

```
ttRamGrace (ttUtilHandle handle, const char *connStr,
            unsigned int seconds)
```

Parameters

ttRamGrace has these 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.

Examples

To set the RAM Grace of 10 seconds for a DSN, “payroll,” use:

```
ttUtilHandleutilHandle;
int    rc;

rc = ttRamGrace (utilHandle, "DSN=payroll",  10);
```

Note

Using this function, the RAM grace period is not honored with an exclusive data store.

See Also

“ttRamLoad” on page 124
“ttRamPolicy” on page 125
“ttRamUnload” on page 127

ttRamLoad

Description

Causes TimesTen to load the data store specified by the connection string into the system's 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.

Access Control

If Access Control is enabled, requires ADMIN privileges or data store object ownership.

Syntax

```
ttRamLoad (ttUtilHandle handle, const char *connStr)
```

Parameters

ttRamLoad has these 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

Examples

To load the data store for the DSN payroll, use:

```
ttUtilHandleutilHandle;
int    rc;
rc = ttRamLoad (utilHandle, "DSN=payroll");
```

Note

You cannot load an exclusive data store into RAM using this function.

See Also

"ttRamGrace" on page 123
"ttRamPolicy" on page 125
"ttRamUnload" on page 127

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.

Access Control

If Access Control is enabled, requires ADMIN privileges or data store object ownership.

Syntax

```
ttRamPolicy (ttUtilHandle handle, const char *connStr,
             ttRamPolicyType policy)
```

Parameters

ttRamPolicy has these 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: **TT_RAMPOL_ALWAYS** - specifies that the data store should remain in RAM all the time. **TT_RAMPOL_MANUAL** - specifies that the data store can be loaded into RAM explicitly using either **ttRamLoad** utility API or the **ttAdmin** `-ramLoad` command. Similarly, the data store can be unloaded from RAM explicitly by using **ttRamUnload** utility API or using **ttAdmin** `-ramUnload` command. **TT_RAMPOL_INUSE** - specifies that the data store is to be loaded into RAM when in an application wants to connect to the data store. This Ram Policy may be further modified using the **ttRamGrace** utility API 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.

Examples

To set the RAM Policy to manual for a DSN, “payroll.”

```
ttUtilHandleutilHandle;
int    rc;

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

Note

The policy cannot be set to always or manual for an exclusive data store. The policy cannot be set for a temporary data store.

See Also

“ttRamGrace” on page 123
“ttRamLoad” on page 124
“ttRamUnload” on page 127

ttRamUnload

Description

Causes TimesTen to unload the data store specified by the connection string from the system's RAM if the RamPolicy is "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.

Access Control

If Access Control is enabled, requires ADMIN privileges or data store object ownership.

Syntax

```
ttRamUnload (ttUtilHandle handle, const char *connStr)
```

Parameters

ttRamUnload has these 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.

Examples

To unload the data store from RAM for a DSN, "payroll."

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

Notes

You cannot unload an exclusive data store from RAM using this function.

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

See Also

"ttRamGrace" on page 123
"ttRamLoad" on page 124
"ttRamPolicy" on page 125

ttRepDuplicate

Description In this release, this function is supported for backward compatibility only. It has been replaced by "ttRepDuplicateEx" on page 129.

ttRepDuplicateAcc

Description In this release, this function is supported for backward compatibility only. It has been replaced by "ttRepDuplicateEx" on page 129.

ttRepDuplicateEx

Description

Provides a way to create a replica of a remote data store on the local machine.

If the remote data store is diskless, this operation obtains a data store-level lock for the entire period of the memory transfer on the remote source data store. It prevents any other connection from accessing the remote data store during the duplicate operation.

You may optionally use a TT_TRUE for the *memCopy* parameter value to create a replica of a remote replication-enabled temporary or permanent data store. With this option, ttRepDuplicateEx in diskless mode creates an intermediate copy of the remote data store in memory prior to copying the data store across the network, so that the remote data store is locked only for the duration of the memory-to-memory copy. This reduces the data store lock out time for this operation in diskless mode. Adequate RAM must be available for the remote extra copy.

Access Control

If Access Control is enabled, the user specified must have ADMIN privileges.

Syntax

```
ttRepDuplicateEx (ttUtilHandle handle,
                  const char *destConnStr,
                  const char *srcDatastore,
                  const char *remoteHost,
                  ttRepDuplicateExArg *arg
                  );
typedef struct
{
      unsigned int size; /*set to sizeof(ttRepDuplicateExArg) */
      unsigned int flags;
      const char *uid;
      const char *pwd;
      const char *pwdcrypt;
      const char *oracleuide;
      const char *oraclepwd;
      const char *localHost;
      int truncListLen;
      const char **truncList;
      int dropListLen;
      const char **dropList;
      int maxkbytesPerSec
      /*new struct elements can only be added here at the end */
} ttRepDuplicateExArg;
```

Parameters **ttRepDuplicateEx** has these 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 datastore path name.
remoteHost	const char *	A NULL-terminated string specifying the TCP/IP host name of the machine where 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 struct 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 below.
uid	const char *	The User ID of the user performing the duplicate operation.
pwd	const char *	The password associated with the User ID.
pwdcrypt	const char *	The encrypted password associated with the User ID.
oracleuid	const char *	The Oracle User ID

oraclepwd	const char *	The Oracle password associated with *oracleuid*.
localHost	const char *	A NULL-terminated string specifying the TCP/IP host name of the local machine. *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.
dropListLen	int	The number of elements in dropList.
dropList	const char**	A list of non-replicated tables to drop after duplicate.
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.

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_MEMCOPY	If the remove source data store is diskless, ttRepDuplicateEx makes an intermediate copy of the remote data store to memory before copying the data store across the network.

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 ramPolicy 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.

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:

```
ttUtilHandleutilHandle;
int    rc;

ttRepDuplicateExArg arg;

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

See Also

The following built-in procedures are described in the *Oracle TimesTen In-Memory Database API and SQL Reference Guide*.

"ttReplicationStatus"
"ttRepPolicy"
"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** utility API 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 "To rename the owner of tables in a data store, use the ttMigrate utility. When restoring tables, you can use the -rename option to rename the owner of tables. Backing up and restoring a data store" in the *Oracle TimesTen In-Memory Database Operations Guide*.

Access Control

If Access Control is enabled, requires ADMIN privileges or data store object ownership.

Syntax

```
ttRestore (ttUtilHandle handle, const char *connStr,
            ttRestoreType type, const char *backupDir,
            const char *baseName, ttUtFileHandle stream,
             unsigned intflags)
```

Parameters

ttRestore has these 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. The valid values are: **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 filename portion of the DataStore attribute of the data store's ODBC definition. For TT_RESTORE_STREAM, this parameter is ignored.
stream	ttUtFileHandle	For TT_RESTORE_STREAM, specifies the stream from which the backup is to be read. On 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 in this release.

Examples

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

```
ttUtilHandleutilHandle;
int    rc;

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

See Also

"ttBackup" on page 115

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 prior to calling any other TimesTen utility library function. In addition, an application must call **ttUtilFreeEnv** when it is done with the TimesTen utility library interface.

Syntax

```
ttUtilAllocEnv (ttUtilHandle *handle_ptr, char *errBuff,
                unsigned int buffLen, unsigned int *errLen);
```

Parameters

ttUtilAllocEnv has these 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 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

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

To allocate and initialize 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" on page 137
"ttUtilGetErrorCount" on page 142
"ttUtilGetError" on page 139

ttUtilFreeEnv

Description

Frees memory associated with the TimesTen utility library handle.

An application must call **ttUtilAllocEnv** prior to calling any other TimesTen utility library function. In addition, an application must call ttUtilFreeEnv when it is done with the TimesTen utility library interface.

Syntax

```
ttUtilFreeEnv (ttUtilHandle handle, char *errBuff,
               unsigned int buffLen, unsigned int *errLen);
```

Parameters

ttUtilFreeEnv has these 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 (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, 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" on page 135
"ttUtilGetErrorCount" on page 142
"ttUtilGetError" on page 139

ttUtilGetError

Description

Retrieves the errors and/or warnings generated by the last call to the TimesTen utility library functions (excluding **ttUtilAllocEnv** and **ttUtilFreeEnv**).

Syntax

```
ttUtilGetError (ttUtilHandle handle, unsigned int errIndex,
                unsigned int *retCode, ttUtilErrType *retType,
                char *errbuff, unsigned int buffLen,
                 unsigned int *errLen);
```

Parameters

ttUtilGetError has these parameters:

Parameter	Type	Description
handle	ttUtilHandle	Specifies the TimesTen utility library environment handle allocated using **ttUtilAllocEnv**.
errIndex	unsigned int	Indicates error/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 **l..n** - Retrieve the specified record from the utility library error array. `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 intretCode;
ttUtilErrTyperetType;
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 utility library functions can potentially generate multiple errors and/or warnings for a single call from an application. To retrieve all of these errors and/or warnings, the application must make repeated calls to ttUtilGetError until it returns TTUTIL_NODATA.

See also

"ttUtilAllocEnv" on page 135
"ttUtilFreeEnv" on page 137
"ttUtilGetErrorCount" on page 142

ttUtilGetErrorCount

Description

Retrieves the number of errors and/or warnings generated by the last call to the TimesTen utility library functions (excluding **ttUtilAllocEnv** and **ttUtilFreeEnv**). Each of these functions can potentially generate multiple errors and/or warnings for a single call from an application. To retrieve all of these errors and/or warnings, the application must make repeated calls to ttUtilGetError until it returns TTUTIL_NODATA.

Syntax

```
ttUtilGetErrorCount (ttUtilHandle handle,
       unsigned int *errCount);
```

Parameters

ttUtilGetErrorCount has these 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/or warnings generated by the last call (excluding ttUtilAllocEnv and ttUtilFreeEnv) to the TimesTen utility library.

Return codes

ttUtilGetErrorCount 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 interrCount;
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/or warnings for a single call from an application. To retrieve all of these errors and/or warnings, the application must make repeated calls to ttUtilGetError until it returns TTUTIL_NODATA.

See also

"ttUtilAllocEnv" on page 135
"ttUtilFreeEnv" on page 137
"ttUtilGetError" on page 139

ttXactIdRollback

Description

Rolls back the transaction indicated by the Transaction ID specified. The intended user of ttXactIdRollback is the **ttXactAdmin** utility. However, programs that want to have a thread with the power to rollback the work of other threads must make sure that those threads call the built-in procedure **ttXactIdGet** before beginning work and put the results into a location known to the rolling back thread.

Syntax

```
ttXactIdRollback (ttUtilHandle handle, const char* connStr,
       const char* xactId);
```

Parameters

ttXactIdRollback has these 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 rollback 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)
   {
   ……————
}
```

7

XLA Reference

This appendix provides reference information for the Transaction Log API (XLA) described in Chapter 3, "XLA and TimesTen Event Management."

The topics in this appendix are:

- About XLA Functions
- Summary of XLA Functions by Category
- XLA Function Reference
- XLA Data Types

About XLA Functions

About return codes

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

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_NO_DATA_FOUND
- SQL_ERROR

See "Handling XLA errors" on page 67 for information on handling XLA errors.

About parameter types (input, out, inout)

In the function descriptions:

- All parameters are input-only parameters, unless otherwise indicated.
- Output parameters are prefixed with the word **out.**
- Input-output parameters are prefixed with the word **inout**.

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:

- 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 needs to 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 `typedef`'s 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 the *Oracle TimesTen In-Memory Database Operations Guide*.

Summary of XLA Functions by Category

As described in Chapter 3, "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. The XLA function categories are.

- XLA persistent mode functions
- XLA core functions (these apply to all of the other categories)
- XLA non-persistent mode functions
- XLA replication functions

XLA persistent mode functions

In addition to many of the core XLA functions listed in "XLA core functions" on page 147, the functions that are exclusive to operating XLA in persistent mode are:

Function	Description	See
ttXlaPersistOpen	Initializes a handle to a TimesTen data store to access the transaction log in persistent mode.	page 180
ttXlaAcknowledge	Acknowledges receipt of one or more transaction update records from the log.	page 151
ttXlaDeleteBookmark	Deletes a transaction log bookmark.	page 161
ttXlaGetLSN	Retrieves the current bookmark read *LSN* for a data store.	page 169
ttXlaSetLSN	Sets the current bookmark read *LSN* for a data store.	page 185

See "Writing an XLA Event-Handler Application" on page 42 for a discussion on how to use these functions.

XLA core functions

The core XLA functions that can be used by any XLA application in either persistent or non-persistent mode are:

Function	Description	See
ttXlaNextUpdate	Retrieves a batch of updates from TimesTen.	page 174
ttXlaNextUpdateWait	Retrieves a batch of updates from TimesTen. Will wait for a specified time if no updates are available in the log.	page 176
ttXlaClose	Closes the XLA handle opened by **ttXlaPersistOpen** or **ttXlaOpenTimesTen**.	page 154

ttXlaError	Retrieves error information.	page 162
ttXlaErrorRestart	Resets error stack information.	page 164
ttXlaGetColumnInfo	Retrieves information about all the columns in the table.	page 167
ttXlaGetTableInfo	Retrieves information about a table.	page 170
ttXlaTableByName	Finds the system and user table identifiers for a table given the table's owner and name.	page 188
ttXlaTableStatus	Sets and retrieves XLA status for a table.	page 191
ttXlaDateToODBCCType	Converts an XLA date value to an ODBC C value usable by applications.	page 158
ttXlaDecimalToCString	Converts an XLA decimal value to an ODBC C value usable by applications.	page 159
ttXlaTimeToODBCCType	Converts a transaction log time value to an ODBC C value usable by applications.	page 193
ttXlaTimeStampToODBCCType	Converts a transaction log timestamp value to an ODBC C value usable by applications.	page 194
ttXlaGetVersion	Retrieves the current version of XLA.	page 171
ttXlaSetVersion	Sets the XLA version to be used.	page 186
ttXlaVersionCompare	Compares two XLA versions.	page 195

See "Writing an XLA Event-Handler Application" on page 42 for a discussion on how to use most of these functions.

XLA non-persistent mode functions

Note: TimesTen recommends you use XLA in persistent mode.

In addition to many of the core XLA functions listed in "XLA core functions" on page 147, the functions that are exclusive to operating XLA in non-persistent mode are:

Function	Description	See
ttXlaOpenTimesTen	Initializes a handle to a TimesTen data store to access the transaction log in non-persistent mode.	page 178
ttXlaConfigBuffer	Sets the size of the XLA staging buffer.	page 156
ttXlaStatus	Retrieves the current XLA status.	page 187
ttXlaResetStatus	Resets all the XLA statistics counters.	page 183

See "Using XLA in Non-Persistent Mode" on page 73 for a discussion on how to use these functions.

XLA replication functions

In addition to many of the core XLA functions listed in "XLA core functions" on page 147, the functions that are exclusive to using XLA as a replication mechanism include:

Function	Description	See
ttXlaApply	Applies the update to the data store or database associated with the XLA handle.	page 152
ttXlaTableCheck	Verifies that the named table in the table description received from the sending data store is compatible with the receiving data store.	page 189
ttXlaLookup	Looks for an update record for a table with a specific key value.	page 172
ttXlaRollback	Rolls back a transaction.	page 184
ttXlaCommit	Commits a transaction.	page 155
ttXlaGenerateSQL	Generates a SQL statement that expresses the effect of an update record.	page 165

See "Using XLA as a Replication Mechanism" on page 76 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 log by the **ttXlaNextUpdate** or **ttXlaNextUpdateWait** function.

Once you make this call, the bookmark is reset so that you cannot re-read any of the previously returned records. So call **ttXlaAcknowledge** only when messages have been completely processed.

Note: The bookmark is only reset for the specified *handle*; other handles in the system may still be able to access those earlier transactions.

---

ttXlaAcknowledge is an expensive operation that should be used sparingly. Calling **ttXlaAcknowledge** more than once per log file read does not reduce the volume of the log since XLA only purges logs a file at a time. You can use the *LogFile* number from the LSN returned in the **ttXlaUpdateDesc_t** header to detect when a new log file is generated. You can then call **ttXlaAcknowledge** to purge the old log file(s).

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 47 for a discussion on the use of this function.

Syntax

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

Parameters

Name	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 = 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 transient problem (deadlock or timeout) or a persistent problem. 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 76 for a discussion on the use of this function.

Note: ttXlaApply cannot be used if the table definition was updated since it was originally written to the log. Unique key and foreign key constraints are checked at the row level rather than at the statement level.

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);
```

Also see the **main()** function in the `install_dir/demo/xlaNonPersistent.c` file.

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** or **ttXlaOpenTimesTen**. See "Terminating an XLA application" on page 70 for a discussion on the use of this function.

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

ttXlaOpenTimesTen

ttXlaCommit

Description

Commits the current transaction being applied on the *handle*. This routine commits the transaction whether or not the transaction has completed. You can call this routine to respond to transient errors (timeout or deadlock) reported by **ttXlaApply**.

See "Handling timeout and deadlock errors" on page 79 for a discussion on the use of this function.

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 only valid when operating XLA 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.

If you specify a value for *newSize*, the new size of the staging buffer is retrieved from **newSize*. A size of zero indicates no staging buffer is or should be allocated. Upon return, **oldSize* contains the previous size of the staging buffer, or 0 if the size was not previously set.

If you do not specify a *newSize,* the current size of the staging buffer is returned in **oldSize*.

Changes to the staging buffer size are carried out immediately. Only one buffer may be configured for a data store. When the buffer is resized, values returned by previous calls on **ttXlaNextUpdate** become invalid.

See "Configuring the staging buffer" on page 74 for a discussion on the use of this function.

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, use the call:

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

To set the size of the staging buffer to a new size of 400,000 bytes, use the call:

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

You can combine the two types of call if you wish and simultaneously set a new size while recording the previous size.

Finally, to delete the staging buffer altogether, use the call:

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

Also see the **main()** function in the `install_dir/demo/xla.c` file.

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

ttXlaDateToODBCCType

Description

Converts an internal date value to an ODBC C value usable by applications. See "Converting complex data types" on page 60 for a discussion on the use of this function.

Syntax

```
SQLRETURN ttXlaDateToODBCCType(void * fromData,
                               out DATE_STRUCT * returnData)
```

Parameters

Name	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.

Example

This example assumes you have used the *offset* value returned in a **ttXlaColDesc_t** structure to obtain a date value, *pColVal*, from a row returned in a transaction log record.

```
DATE_STRUCT date;

rc = ttXlaDateToODBCCType(pColVal, &date);
```

See Also

ttXlaDecimalToCString

ttXlaTimeToODBCCType

ttXlaTimeStampToODBCCType

ttXlaDecimalToCString

Description

Converts an internal decimal 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*.

See "Converting complex data types" on page 60 for a discussion on the use of this function.

Syntax

```
SQLRETURN ttXlaDecimalToCString(void *fromData,
                                out char *returnData,
                                SQLSMALLINT precision,
                                SQLSMALLINT scale)
```

Parameters

Name	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);
```

See Also **ttXlaDateToODBCCType**

ttXlaTimeToODBCCType

ttXlaTimeStampToODBCCType

ttXlaDeleteBookmark

Description

Deletes the bookmark associated with the specified *handle*. Once the bookmark has been deleted, it will no longer be accessible and its identifier may be reused for another bookmark. The deleted bookmark will no longer be associated with the data store handle and the effect will be the same as having opened the persistent connection with the XLANONE option.

See "Deleting bookmarks" on page 72 for a discussion on the use of this function

Access Control

If Access Control is enabled, requires ADMIN privileges or data store object ownership.

Syntax

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

Parameters

Name	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

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 67 for a discussion on the use of this function.

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 re-read the errors. See "Handling XLA errors" on page 67 for a discussion on the use of this function.

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 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.

See "Replicating updates to a non-TimesTen data store" on page 80 for a discussion on the use of this function.

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*.
actualLen	out SQLINTEGER *	The actual length of the *buffer*.

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 **ttXlaGenerateSQL** 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 **ttXlaColDesc_t** data type is defined in "ttXlaColDesc_t" on page 211. *SystemTableID* or *userTableID* describe the desired table (see "ttXlaGetTableInfo" on page 170 for details). This call is serialized with respect to changes in the table's definition.

See "Obtaining column descriptions" on page 53 for a discussion on the use of this function.

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 211 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 sequence number (LSN) for the connection specified by *handle*. See "About XLA bookmarks" on page 37 for a discussion on the use of this function.

Syntax

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

Parameters

Name	Type	Description
handle	ttXlaHandle_h	The transaction log handle for the data store.
LSN	out tt_XlaLsn_t *	The current read LSN for the *handle*.

Returns

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

Example

This example returns the current read LSN pointer, *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 non-zero, 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's definition.

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 modes: persistent vs. non-persistent" on page 34 for a discussion on the use of this function.

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.
actual version	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.

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. Prior to 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 call 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.

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 47 for a discussion on the use of this function.

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]);
}
```

Also see the **inspectTransactions()** function in the `install_dir/demo/persistent_xla/subscriber.c` file and the **main()** function in the `install_dir/demo/xla.c` file.

Notes Updates are generated for all data definition statements, regardless of tracking status. Updates are also generated for data update operations on *all* XLA tracked tables. This means that each XLA reader process will need to filter out updates for unwanted table IDs.

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 log. The actual number of seconds of wait time can be up to two seconds more than the specified *seconds* value.

See "Retrieving update records from the transaction log" on page 47 for a discussion on the use of this function.

Syntax

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

Parameters

Name	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 records returned. 0 is returned if no update data is available within the *seconds* wait period.
seconds	SQLINTEGER	The number of seconds to wait if 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 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

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 log at any point in time. See "Initializing XLA in non-persistent mode" on page 74 for a discussion on the use of this function.

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

Access Control

If Access Control is enabled, requires WRITE privileges or data store object ownership.

Note: Once a session has been opened, no further privilege checking is done for XLA calls on the handle.

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.

To open a transaction log in non-persistent mode and return a handle, named *xlahandle*, for the ODBC connection, call:

```
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

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 per ODBC connection. Once 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 37). The *tag* can identify either a new bookmark 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 log without a bookmark. The actions that *cannot* be done without a bookmark are:

- **ttXlaAcknowledge**
- **ttXlaGetLSN**
- **ttXlaSetLSN**
- **ttXlaNextUpdate**
- **ttXlaNextUpdateWait**

In persistent mode, multiple applications can concurrently read from the log. See "Initializing XLA and obtaining an XLA handle" on page 45 for a discussion on the use of this function.

Note: Persistent mode is only supported with disk-based logging.

Access Control

If Access Control is enabled, requires WRITE privileges or data store object ownership.

Note: Once a session has been opened, no further privilege checking is done for XLA calls on the handle.

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.
options	SQLUINTEGER	Bookmark options: • XLANONE - Connect without a bookmark. The *tag* field is ignored. • XLACREAT - Create a new bookmark. Fails if a bookmark already exists. • XLAREUSE - Associate with an existing bookmark. Fails if the bookmark doesn't 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. This example opens a transaction log in persistent mode, returns a handle, named *xlahandle*, and creates a new bookmark named *mybookmark*:

```
SQLHDBC hdbc;
ttXlaHandle_h xlahandle;

rc = ttXlaPersistOpen(hdbc, ( SQLCHAR*)mybookmark,
                      XLACREAT, &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

Resets all the XLA status counters reported in the **ttXlaStatus_t** structure returned by **ttXlaStatus**. Currently, only the *xlabufminfree* value is reset. See "Retrieving and resetting buffer status" on page 75 for a discussion on the use of this function.

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 79 for a discussion on the use of this function.

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

ttXlaSetLSN

Description

Sets the current read log sequence number (LSN) 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 initial read LSN).

See "About XLA bookmarks" on page 37 for a discussion on the use of this function.

Syntax

```
SQLRETURN ttXlaSetLSN(ttXlaHandle_h handle,
                      tt_XlaLsn_t *LSN)
```

Parameters

Name	Type	Description
handle	ttXlaHandle_h	The transaction log handle for the data store.
LSN	tt_XlaLsn_t *	The new read LSN for the *handle*.

Returns

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

Example

This example sets the current read LSN pointer 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 modes: persistent vs. non-persistent" on page 34 for a discussion on the use of this function.

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 only valid when operating XLA in non-persistent mode.

Retrieves the current XLA status and stores in the **status* parameter. See "Retrieving and resetting buffer status" on page 75 for a discussion on the use of this function.

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 46 for a discussion on the use of this function.

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** received from a master data store is compatible with a subscriber data store or database associated with the *handle*. The *compat* parameter indicates whether or not the tables are compatible.

See "Checking table compatibility between data stores" on page 77 for a discussion on the use of this function.

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 *	Table's column description.
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 non-zero, 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 non-zero 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 a table's status.

Note: Subscriptions made by **ttXlaTableStatus** to monitor a table or view are global. So that the tables or views subscribed to by one XLA reader are also seen by any other XLA reader of the transaction log.

See "Specifying which tables to monitor for updates" on page 46 for a discussion on the use of this function.

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: On = 1 or Off = 0.
newstatus	SQLINTEGER *	XLA status: On = 1 or Off = 0.

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 non-zero if update tracking for the table is enabled and 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, for 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 an internal time value to an ODBC C value usable by applications. See "Converting complex data types" on page 60 for a discussion on the use of this function.

Syntax

```
SQLRETURN ttXlaTimeToODBCCType (void *fromData,
                                out TIME_STRUCT *returnData)
```

Parameters

Name	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);
```

See Also

ttXlaDateToODBCCType

ttXlaDecimalToCString

ttXlaTimeStampToODBCCType

ttXlaTimeStampToODBCCType

Description

Converts an internal timestamp value to an ODBC C value usable by applications. See "Converting complex data types" on page 60 for a discussion on the use of this function.

Syntax

```
SQLRETURN ttXlaTimeStampToODBCCType(void *fromData,
                                    out TIMESTAMP_STRUCT *returnData)
```

Parameters

Name	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 call is successful. Otherwise, use **ttXlaError** to report the error.

Example

This example assumes you have used the *offset* value returned in a **ttXlaColDesc_t** structure to obtain a timestamp value, *pColVal*, from a row returned in a transaction log record.

```
TIMESTAMP_STRUCT timestamp;

rc = ttXlaTimeStampToODBCCType(pColVal, &timestamp);
```

See Also

ttXlaDateToODBCCType

ttXlaDecimalToCString

ttXlaTimeToODBCCType

ttXlaVersionCompare

Description

Compares two XLA versions and returns the result.

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 = *version1* and *version2* match -1 = *version1* is earlier than *version2* +1 = *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 the 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 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

XLA Data Types

This section describes the data types used by the XLA functions described in this appendix. These types are defined in the file:

```
install_dir/bits/include/tt_xla.h
```

You must include this file when building your XLA application.

Summary of XLA Data Types

Data Type	Description	See
ttXlaNodeHdr_t	Describes the record type. Used at the beginning of records returned by XLA.	page 198
ttXlaUpdateDesc_t	Describes an update record.	page 199
ttXlaStatus_t	Describes XLA status information (returned by **ttXlaStatus**).	page 208
ttXlaVersion_t	Describes XLA version information (returned by **ttXlaGetVersion**).	page 209
ttXlaTblDesc_t	Describes a table information (returned by **ttXlaGetTableInfo**).	page 210
ttXlaColDesc_t	Describes a table column information (returned by **ttXlaGetColumnInfo**).	page 211
tt_XlaLsn_t	Describes an LSN pointer used by an XLA bookmark.	page 213

ttXlaNodeHdr_t

Most data types in XLA 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

The **ttXlaUpdateDesc_t** 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 203.

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 **ttApplicationContext** is not followed by an operation that generates an XLA record. See "Passing application context" on page 82 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; or of type CREAIND or DROPIND when a unique attribute of a column is modified.

The fields of the update header defined by **ttXlaUpdateDesc_t** are:

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

flags	SQLUSMALLINT	Special options on record update: • TT_UPDFIRST - indicates that the update record is the first record for the transaction. • TT_UPDCOMMIT - indicates that the update record is the last record for the transaction. (Implied commit.) • 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 would be (2, 1, 3) or (2, 3, 1), depending on the UPDATE statement used. This array is included with all UPDATETUP records. • TT_UPDREPL - indicates that this update was the result of a non-XLA TimesTen replicated update from another data store.

flags (continued)	SQLUSMALLINT	• 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. If the value of a specific column is 0, it indicates that column does not have a default value. The defaults for all non-zero values are concatenated in a string and are presented in order, with the array value indicating the length of the default value. For example, 3 columns with defaults “1” of type integer, no default and “apple” of type varchar(10) would be (1,0,5)1,0,5)“apple”.
contextOffset	SQLUINTEGER	Offset to application-provided context value. This value is 0 if there is no context. A non-zero value indicates the location of the context relative to the beginning of the XLA record.
connID	SQLUBIGINT	Connection ID owning the transaction.
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	Log sequence number of this operation; used for diagnostics.
tuple1	SQLUINTEGER	Length of first row (*tuple*) or zero.
tuple2	SQLUINTEGER	Length of second row (*tuple*) or zero.

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 operation.

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 **ttXlaDropColumnTup_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:

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:

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:

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 (1)	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' = T-tree '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:

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:

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:

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:

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 starts from maxval to minval, and vice versa for ascending sequence.

ttXlaDropSeqTup_t

For a DROP SEQUENCE operation, the row value is:

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:

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:

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 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 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 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 47 and "Inspecting record headers and locating row addresses" on page 51 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 211 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 data type is returned by the **ttXlaStatus** function when operating XLA in non-persistent mode.

Field	Type	Description
header	ttXlaNodeHdr_t	Standard data type 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 data type **ttXlaVersion_t** describes the current XLA version and data type byte order. This data type is returned by the **ttXlaGetVersion** function.

This data type includes the 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** data type. This data type is return by the **ttXlaGetTableInfo** function.

This data type includes the 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	In-line 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 in-line row size includes space for all fixed-width columns, null column flags, and pointer information for varying-length columns. Each varying-length column occupies 4 bytes of in-line row space.

If the table has a declared primary key:

- *nprimcols* will be larger than 0.
- *primcolsSys* array will contain the column numbers of the primary key, in the same order they were originally declared with the create table statement.
- *primcolsUser* array will contain the corresponding application-specified column identifiers.

ttXlaColDesc_t

Column information is given through the **ttXlaColDesc_t** data type. This data type is return by the **ttXlaGetColumnInfo** function.

This data type includes the fields:

Field	Type	Description
header	ttXlaNodeHdr_t	Standard data header.
colName	char (31)	Name of the column.
sysColNum	SQLUINTEGER	Column number, numbered from 1.
userColNum	SQLUINTEGER	User-assigned column number.
dataType	SQLUINTEGER	Data type in ODBC SQL_* code.
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	SQLUSMALLINT	Numeric precision for decimal types.
scale	SQLUSMALLINT	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 detail in "Inspecting column data" on page 53. 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 **ttXlaUpdateDesc_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 60.

TimesTen row data consists of fixed-length data followed by any variable-length data.

- For fixed length column data, **ttXlaColDesc_t** returns the *offset* and *size* of the column data. The *offset* is relative to the beginning of the fixed part of the record. See the example below.
- For variable-length column data (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 the example below.

For columns that can have NULL values, *nullOffset* points to a "null-byte" in the record. This value is 1 if the column is NULL, or 0 if it is not NULL. See "Detecting NULL values" on page 61 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

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:

```
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:

```
*((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 done 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, we copy and print a VARCHAR string.

```
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 an LSN pointer used by bookmarks. This data type is used by the **ttXlaUpdateDesc_t** structure.

Field	Type	Description
LogFile	SQLUBIGINT	Log file number
LogOffset	SQLUBIGINT	Log file offset

tt_XlaLsn_t

Description of an LSN pointer used by bookmarks. This data type is return by the **ttXlaGetLSN** function and used by the **ttXlaSetLSN** function.

The *checksum* is specific to an XLA handle to ensure that every LSN pointer is related to a known XLA connection:

Field	Type	Description
checksum	SQLUINTEGER	Checksum used to ensure that it is a valid LSN handle
xid	SQLUSMALLINT	Transaction ID
LogFile	SQLUBIGINT	Log file number
LogOffset	SQLUBIGINT	Log file offset

8

Working with Unicode

ODBC

The ODBC types SQL_WCHAR and SQL_WVARCHAR provide support for the NCHAR and NVARCHAR data types. These data types will be accessible as bound data through function calls such as SQLBindCol and SQLBindParameter, etc. Functions such as SQLGetTypeInfo and SQLDescribeCol will return appropriate information for the SQL_WCHAR and SQL_WVARCHAR types.

SQL data types

The following sections describe the support for the types.

SQL type identifier	SQL data type	Description
SQL_WCHAR	NCHAR(n)	Unicode character string of fixed string length n
SQL_WVARCHAR	NVARCHAR(n)	Unicode variable-length character string with a maximum string length n

SQLGetTypeInfo results for SQL_WCHAR and SQL_WVARCHAR:

TYPE_NAME	DATA_TYPE	PRECISION	LITERAL_ PREFIX	LITERAL _SUFFIX
“NCHAR”	SQL_WCHAR	4150	“N’”	“‘”
“NVARCHAR”	SQL_WVARCHAR	2097152	“N’”	“‘”

CREATE_PARAMS	NULLABLE	CASE_SENSITIVE	SEARCHABLE	UNSIGNED_ATTRIBUTE
"length"	SQL_NULLABLE	SQL_TRUE	SQL_LIKE_ONLY[a]	NULL
"max length"	SQL_NULLABLE	SQL_TRUE	SQL_LIKE_ONLY[†]	NULL

a. Although certain comparison operators besides LIKE are supported, we cannot return SQL_SEARCHABLE here because not all comparison operators are supported with these types.

MONEY	AUTO_INCREMENT	LOCAL_TYPE_NAME	MINIMUM_SCALE	MAXIMUM_SCALE
SQL_FALSE	NULL	NULL	NULL	NULL
SQL_FALSE	NULL	NULL	NULL	NULL

C data type

C type identifier	ODBC C typedef	C type
SQL_C_WCHAR	SQLWCHAR *	unsigned short *

SQL_C_DEFAULT

When calling SQLBindCol, SQLGetData, or SQLBindParameter with SQL_C_DEFAULT as the type specifier, SQL_WCHAR maps to SQL_C_WCHAR.

Include File Definitions

typedef in include file `sqltypes.h`:

```
typedef unsigned short SQLWCHAR;
```

Note: This definition of SQLWCHAR does not make use of the system defined `wchar_t` type. The `wchar_t` type is implementation specific and thus cannot be guaranteed to provide a consistent interface between different platforms.

Type definitions in include file `timesten.h`:

```
#define SQL_WCHAR(-8)
#define SQL_WVARCHAR(-9)
#define SQL_WLONGVARCHAR(-10)

#define SQL_C_WCHARSQL_WCHAR

/* SQLGetInfo() InfoTypes */
#define SQL_CONVERT_WCHAR122
#define SQL_CONVERT_WLONGVARCHAR125
#define SQL_CONVERT_WVARCHAR126

/* SQLGetInfo() return value bitmasks */
#ifndef SQL_CVT_WCHAR
/*
** These definitions differ from Microsoft in that
** they are not specified as long (e.g. 0x00200000L),
** hence they are protected
** by the ifndef above.
*/
#define SQL_CVT_WCHAR0x00200000
#define SQL_CVT_WLONGVARCHAR0x00400000
#define SQL_CVT_WVARCHAR0x00800000
#endif

/*
** The Microsoft Driver Manager SQLBindParameter()
** will not pass SQL_WCHAR through.  Use this code
** to get around it.
*/
#define SQL_WCHAR_DM_SQLBINDPARAMETER_BYPASS -888
```

These definitions match those defined by Microsoft in ODBC 3.51, as found in the files `sqlucode.h` and `sqlext.h` distributed with Visual C++ and the Platform SDK.

Data Conversions

TimesTen provides limited support for conversions to and from the NCHAR types. The supported conversions (bindings) are any combination of the following types:

SQL Type	C Type
SQL_WCHAR	SQL_C_WCHAR
SQL_WVARCHAR	SQL_C_DEFAULT SQL_C_BINARY

SQL to C Data Conversions

SQL_WCHAR, SQL_WVARCHAR to SQL_C_WCHAR, SQL_C_BINARY

SQL to C: Character

C type identifier (TargetType)	Test	TargetValuePtr[a]	*StrLen_or_IndPtr[b]	SQLSTATE
SQL_C_WCHAR	Byte length of data < *BufferLength.*	Data	Length of data in bytes	N/A
	Byte length of data >= *BufferLength.*	Truncated data†	Length of data in bytes	01004
SQL_C_BINARY	Byte length of data <= *BufferLength.*	Data	Length of data in bytes	N/A
	Byte length of data > *BufferLength.*	Truncated data‡	Length of data in bytes	01004

a.Data will be truncated on character (two-byte) boundaries.

b.Data will be truncated on byte boundaries. If *BufferLength* is odd, a partial character will be returned.

If the `fcType` argument in SQLBindCol or SQLGetData contains an identifier that specifies a conversion from SQL_WCHAR or SQL_WVARCHAR to an ODBC C data type other than those listed above, SQLFetch or SQLGetData return SQLSTATE S1C00 (Driver not capable).

Attempts to convert from other SQL types to SQL_C_WCHAR will return SQLSTATE 07006 (Restricted data type attribute violation).

C to SQL Data Conversions

SQL_C_WCHAR to SQL_WCHAR, SQL_WVARCHAR

C to SQL: Character

SQL Type Identifier	Test	SQLSTATE
SQL_WCHAR	Character length of data <= Column length.	N/A
SQL_WVARCHAR	Character length of data > Column length.	01004

SQL_C_BINARY to SQL_WCHAR, SQL_WVARCHAR

C to SQL: Binary

SQL Type Identifier	Test	SQLSTATE
SQL_WCHAR SQL_WVARCHAR	Character length of data <= Column character length†	N/A
	Character length of data > Column character length	01004

Note: Data must contain full two-byte characters. Trailing partial characters will be dropped with SQLSTATE 01004 (Data truncated) being returned.

If the `fcType` argument in SQLBindParameter contains an identifier other than SQL_C_WCHAR or SQL_C_BINARY for and fSQlType of SQL_WCHAR or SQL_WVARCHAR, either SQLBindParameter or SQLExecute returns SQLSTATE S1C00 (Driver not capable).

Attempts to convert from SQL_C_WCHAR to SQL types other than SQL_WCHAR or SQL_WVARCHAR return SQLSTATE 07006 (Restricted data type attribute violation).

Function Support

Support for the NCHAR and NVARCHAR types does not require changes to the ODBC function signatures. The changes to the ODBC interface consist solely of recognition of the type identifiers.

The following functions recognize the data type identifiers as input or output parameters:

SQLBindCol

SQLBindParameter

SQLColAttributes

SQLDescribeCol

SQLDescribeParam

SQLGetData

SQLGetTypeInfo

The following functions recognize these data types and behave in an appropriate manner, subject to the conversion rules above:

SQLColumns

SQLExecDirect

SQLExecute

SQLFetch

SQLPutData

The *StatementText* parameter for the following functions must be of type `SQLCHAR *` and consist solely of ASCII characters:

SQLPrepare

SQLExecDirect

Support has been added to SQLGetInfo for the **SQL_CONVERT_xxx** InfoTypes.

ODBC Driver Manager

The NCHAR and NVARCHAR types are generally not available through ODBC driver managers. However, limited functionality can be obtained through the use of the SQL_C_DEFAULT parameter, which is passed through the Driver Manager and then interpreted correctly as the appropriate NCHAR type by the TimesTen driver.

SQLBindParameter presents a special problem in that an fSqlType argument of SQL_WCHAR or SQL_WVARCHAR is intercepted by the Driver Manager, resulting in an S1004 (SQL data type out of range) error. Passing the special value SQL_WCHAR_DM_SQLBINDPARAMETER_BYPASS instead of SQL_WCHAR or SQL_WVARCHAR allows the SQLBindParameter call to succeed through the Driver Manager. Use of this special value will make the program compatible only with TimesTen drivers.

Unless specific Driver Manager capabilities are required, TimesTen recommends that Unicode applications be linked directly to the appropriate TimesTen driver. In this case, the application should pass the correct type values to SQLBindParameter, not the special values described above.

Unicode Escapes Embedded in C Strings

Unicode escapes embedded in C language strings must have the backslash character escaped with another backslash. For example:

SQLCHAR c* = “insert into t values (N’\\u0061’);”;

The C compiler interprets the double-backslash as a single backslash, producing the intended result. If a single backslash is used, the compiler sees it as an unrecognized escape sequence and discards it, producing an incorrect result.

Encoding of Bound Data

All data input into TimesTen through bound variables must be inUTF-16 format. Similarly, TimesTen outputs bound variable data inUTF-16 format. If necessary, the application developer must perform any necessary conversions betweenUTF-16 encoded data and the appropriate native data encoding.

Such conversions can be done using the iconv(3C) set of routines available on most UNIX platforms. On Windows, the MultiByteToWideChar and WideCharToMultiByte functions can be used to accomplish similar conversions. The demo program `install_dir/demos/unicode.c` illustrates how to use ODBC with Unicode characters. It also shows how to convert from native character sets into theUTF-16 format required by the ODBC calls.

9

TimesTen ODBC Functions

This appendix lists all ODBC functions supported by TimesTen. For complete function definitions, see the *Microsoft ODBC Programmer's Reference*.

Supported ODBC functions

SQLAllocConnect
SQLAllocEnv
SQLAllocStmt
SQLBindCol
SQLBindParameter
SQLCancel[1.]
SQLColAttributes
SQLColumns[6.]
SQLConnect
SQLDataSources[6. 2.]
SQLDescribeCol
SQLDescribeParam[6.]
SQLDisconnect
SQLDriverConnect
SQLDrivers[2.]
SQLError
SQLExecDirect
SQLExecute
SQLFetch
SQLForeignKeys
SQLFreeConnect
SQLFreeEnv
SQLFreeStmt
SQLGetConnectOption
SQLGetCursorName
SQLGetData
SQLGetFunctions
SQLGetInfo
SQLGetStmtOption
SQLGetTypeInfo
SQLNativeSql[6.]
SQLNumParams[6.]
SQLNumResultCols
SQLParamData
SQLParamOptions[3.]
SQLPrepare
SQLPrimary Keys
SQLProcedureColumns[6.]
SQLProcedures[6.]
SQLPutData
SQLRowCount
SQLSetConnectOption
SQLSetCursorName
SQLSetStmtOption

SQLSetParam[4.]

SQLSpecialColumns

SQLStatistics

SQLTables

SQLTransact

Notes

1. In TimesTen, SQLCancel cannot cancel a function running asynchronously on the `hstmt` or one running on the `hstmt` on another thread.
2. ODBC Level 2 function.
3. Available only to programs using the driver manager.
4. When using SQLParamOptions with array or variable length types make sure to set the `cbValueMax` parameter of SQLBindParameter appropriately.
5. ODBC 1.0 function, replaced by SQLBindParameter in ODBC 2.0. Retained for backward compatibility.
6. SQLColumns returns an INTEGER column called INLINE, which indicates whether the given column is INLINE or NOT INLINE (the value is 0 or 1). The column is valid for all column types, not just the varying-length types.

Supported and unsupported options

Table 9.1 Supported ODBC options

ODBC Function	Supported Option	Notes
SQLSetStmtOption	SQL_QUERY_TIMEOUT	-
	SQL_MAX_ROWS	-
	SQL_NOSCAN	-
	TT_PREFETCH_COUNT	-
	SQL_ROWSET_SIZE	Only if size > 0
	SQL_BIND_TYPE	Only if setting it to SQL_BIND_BY_COLUMN
	SQL_ASYNC_ENABLE	Only if setting it to SQL_ASYNC_ENABLE_OFF
	SQL_CURSOR_TYPE	Only if setting it to SQL_CURSOR_FORWARD_ONLY
	SQL_USE_BOOKMARKS	Only if setting it to SQL_UB_ON
	SQL_MAX_LENGTH	Only if setting it to 0

SQLSetConnectOption	SQL_AUTOCOMMIT	-
	SQL_TXN_ISOLATION	-
	SQL_MAX_ROWS	-
	SQL_NOSCAN	-
	SQL_QUERY_TIMEOUT	-
	TT_PREFETCH_CLOSE	-
	TT_SQL_GETDATA_REFETCH	-
	SQL_ACCESS_MODE	SQL_MODE_READ_ONLY is accepted but has no effect
	SQL_LOGIN_TIMEOUT	Ignored for compatibility with Microsoft applications
	SQL_OPT_TRACE	Accepted if set to SQL_OPT_TRACE_OFF
	SQL_QUIET_MODE	Accepted but hwnd is not used even if provided

Table 9.2 Unsupported ODBC options

ODBC Function	Unsupported Option
SQLSetStmtOption	SQL_CONCURRENCY
	SQL_RETRIEVE_DATA
	SQL_SIMULATE_CURSOR
	SQL_KEYSET_SIZE
SQLSetConnectOption	SQL_CURRENT_QUALIFIER
	SQL_ODBC_CURSORS
	SQL_OPT_TRACEFILE
	SQL_PACKET_SIZE
	SQL_TRANSLATE_DLL
	SQL_TRANSLATE_OPTION

10

ODBC for UNIX

ODBC programming for UNIX is fundamentally the same as ODBC programming on Windows. The primary difference between the two platforms is:

- UNIX programs must be linked directly with the TimesTen Data Manager ODBC driver or the TimesTen Client ODBC driver.
- NT programs have the option of using the ODBC driver manager.

The TimesTen Data Manager ODBC driver and the TimesTen Client ODBC driver are ODBC 2.5 drivers and are closer in functionality to ODBC 2.0 than to ODBC 3.0. Therefore, the ODBC 2.0 manual is included in an online format for UNIX users.

Irrelevant chapters

A few chapters in the Microsoft ODBC manual are not relevant for UNIX developers working with TimesTen:

- Chapter 18, “Constructing an ODBC Driver” and Chapter 24, “Installer DLL Function Reference” are only relevant for Windows NT 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 irrelevant chapters, you have to consider the following minor issues to successfully work with the Microsoft ODBC manual as a UNIX user:

- **Irrelevant information**. Some information in the Microsoft manual is irrelevant 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.
- **Filenames**. Under Windows, all filenames are upper case (e.g., SQL.H or SQLEXT.H). Under UNIX, all filenames are all lower case (e.g., sql.h or sqlext.h). Otherwise, the names are equivalent, with the exception of the ODBC.INI file.
- **ODBC.INI file or registry**. The Microsoft ODBC 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 manual states character restrictions for certain functions, such as ConfigDSN, that differ from the restrictions in UNIX as follows:
- **ODBC on Windows NT:** The keywords and their values should not contain the `[ ] { } ( ) , ; ? * = ! @` 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 the following sections was taken, with permission, directly from the README.TXT file for Microsoft ODBC 2.5.

© Copyright Microsoft® Corporation, 1995.

Note: Information that is not relevant for TimesTen developers was cut from the material distributed by Microsoft Corporation.

Header files

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.

- 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.

ODBC function changes

The following changes have been made to the ODBC functions documented in the *ODBC 2.5 Programmer's Reference and SDK Guide*.

Rebinding with SQLBindCol

An application can call **SQLBindCol** to bind a column to a new storage location, regardless of whether data has already been fetched. The new binding replaces the old binding. This is true for bookmark columns as well as other bound columns. The new binding does not apply to data already fetched—it takes effect the next time **SQLFetch** or **SQLSetPos** is called.

Binding a LONG on 64-bit platforms

When binding a long on a 64-bit platform, you should use the SQL_C_SBIGINT data type:

```
rc = SQLBindCol(hstmt, 1, SQL_C_SBIGINT, &precision_l, 0, &pLen);
```

The *ODBC 3.0 Programmer's Reference and SDK Guide* describes SQL_C_SLONG as corresponding to the C type "long int." However, the SQL_C_SLONG data type is 4 bytes, even in 64-bit mode, while SQL_C_SBIGINT is 8 bytes, regardless of platform size.

pcbValue in SQLBindParameter

When *pcbValue* in **SQLBindParameter** is SQL_DEFAULT_PARAM, the corresponding parameter can only be a parameter for an ODBC canonical procedure invocation.

SQLExecDirect, **SQLExecute** and **SQLPutData** return SQLSTATE 07S01 (Invalid use of default parameter) when a parameter value was set to SQL_DEFAULT_PARAM and the corresponding parameter was not a parameter for an ODBC canonical procedure invocation.

SQLSTATE S1C00 returned by SQLPrepare

SQLPrepare will return SQLSTATE S1C00 (Driver not capable) if the cursor/concurrency combination is invalid.

SQLSTATE 22005 returned by SQLFetch

SQLFetch will return SQLSTATE 22005 (Error in assignment) if a zero-length string was inserted into a string field and the field was bound to a numeric data type, resulting in the string being converted to zero.

SQLSTATE 22008 returned by SQLFetch

SQLFetch will return SQLSTATE 22008 (Datetime field overflow) if a SQL_C_TIME, SQL_C_DATE, or SQL_C_TIMESTAMP value was converted to a SQL_CHAR data type, and the value was, respectively, an invalid DATE, TIME or TIMESTAMP.

SQLSTATE 22012 returned by SQLGetData

SQLGetData will return SQLSTATE 22012 (Division by zero) if a value from an arithmetic expression was returned that resulted in division by zero.

ODBC functions

Applications can use all ODBC functions listed in Chapter 9, "TimesTen ODBC Functions." These functions are described in the *Microsoft ODBC 2.0 Programmer's Reference*.

ODBC data types

In addition to the materials in the ODBC 2.0 manual, ODBC 3.0 provides optional new data types that may be used by applications. The data types used in ODBC 2.0 and prior have been renamed in ODBC 3.0 for ISO 92 standards compliance. The ODBC 2.0 documentation shipped with TimesTen for UNIX documents all functions using the old data types. However, the sample programs shipped with TimesTen for UNIX have been written using the new data types. The following table maps old types to their new equivalents.

ODBC 2 Data Type	ODBC 3 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

Both the old and new data types may be used with TimesTen without restriction.

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

Glossary

.odbc.ini file

See ODBC.INI file.

ACID transaction semantics

An acronym referring to the four fundamental properties of a transaction: atomicity, consistency, isolation and durability.

API

See "application programming interface (API)".

application programming interface (API)

A set of functions that an application uses to request and carry out lower-level services. The ODBC API is composed of the ODBC functions.

atomicity

A property of a transaction whereby either all or none of the operations of a transaction are applied to the data store.

backup instance

A set of files containing backup information for a given data store, residing at a given "backup path". See also "backup point", "full backup" and "incremental backup".

backup path

The location of a data store, specified by a directory name and an optional basename.

backup point

The time at which a backup begins. See also "backup path", "full backup" and "incremental backup".

cache group

A set of cached tables related through foreign keys.

cache instance

A set of rows related through foreign keys. Each cache instance contains exactly one row from the root table of a cache group and zero or more rows from the other tables in the cache group.

client/server

An approach to application design and development in which application processing is divided between components running on an end user's machine (the client) and a network server. Generally, user interface elements are implemented in the client component, while the server controls database access.

client data source name

See "data source name, client".

concurrency

The ability to have multiple transactions access and manipulate the data store at the same time.

connection

A data path between an application and a particular ODBC data source.

connection attribute

A character string that defines a connection parameter to be used when connecting to an ODBC data source. Connection attributes have the form *name=value*, where *name* is the name of the parameter and *value* is the parameter value. See also connection string.

connection request

A message sent by an application through an ODBC driver to an ODBC data source to request a connection to that data source.

connection string

A character string that defines the connection parameters to be used when connecting to an ODBC data source. A connection string is expressed as one or more connection attributes separated by semicolons.

consistency

A property of transactions whereby each transaction transforms the database from one consistent state to another.

cursor

A control structure used by an application to iterate through the results of an SQL query.

data source definition

A named collection of connection attributes that defines the connection parameters to be used when connecting to an ODBC data source. See also "data source name".

data source name

A logical name by which an end user or application refers to an ODBC data source definition. Sometimes incorrectly used to mean "data source definition." See also "data source definition", ODBC.INI file.

data source name, client

A data source name defined on a TimesTen client machine that refers to a TimesTen server data source name on a server machine.

data source name, server

A system data source name (system DSN) defined on a server machine. Server Data Source Names become available to all TimesTen clients on a network when the TimesTen Server is running.

data source name, system

A data source name that is accessible by all users of a particular machine.

data source name, user

A data source name that is accessible only by the user who created the data source name.

driver

See "ODBC driver".

DSN

See "data source name".

DSN, client

See "data source name, client".

DSN, server

See "data source name, server".

DSN, system

See "data source name, system".

DSN, user

See "data source name, user".

durability

A property of transactions whereby the effects of a committed transaction survive system failures.

environment variable

A *name*, *value* pair maintained by the operating system that can be used to pass configuration parameters to an application.

event

An activity or occurrence that can be tracked by a logging mechanism in an application, service or operating system. See also "logging", "protocol message logging" and "event viewer".

event viewer

On Windows, a utility program used to view the contents of the operating system event log.

firewall

A combination of hardware and software that reliably separates one part of a network from another for security purposes.

full backup

A data store backup procedure in which a complete copy of a data store is created. Typically, the first backup of a data store must be a full backup. See also "incremental backup".

host

A computer. Typically used to refer to a computer on a network that provides services to other computers on the network.

host name

A character string name that uniquely identifies a particular computer on a network. Examples: athena, krakatoa.timesten.com. See also "host".

in-line column

A column whose values are physically stored together with the other column values of a row.

incremental backup

A data store backup procedure in which an existing backup is augmented with all the log records created since its last full or incremental backup. See also "backup instance" and "full backup".

initialization file

See ODBC.INI file.

IP address

A numeric address that uniquely identifies a computer on a network and consists of four numbers separated by dots. Abbreviation for Internet Protocol address. Example: `123.61.129.91`.

IPC

Inter Process Communication.

isolation

A property of transactions whereby each transaction runs as if it were the only transaction in the system.

listener thread

A thread that runs on the TimesTen Server that receives and processes connection requests from TimesTen Clients.

logging

The process by which an application, service or operating system records specific events that occur during processing.

multi-threading

A programming paradigm in which a process contains multiple threads of control.

network address

A host name, or IP address that uniquely identifies a particular computer on a network. Examples: `123.61.129.91`, `athena`, `thams.mycompany.com`.

ODBC

See "Open Database Connectivity (ODBC)".

ODBC Administrator

A utility program used on Windows to create, configure and delete data source definitions.

ODBC data source

See "data source name" (DSN).

ODBC data source name

See "data source name" (DSN).

ODBC driver

A library that implements the function calls defined in the ODBC API and enables applications to interact with ODBC data sources.

ODBC Driver Manager

A library that acts as an intermediary between an ODBC application and one or more ODBC drivers.

ODBC initialization file

See ".odbc.ini file".

Open Database Connectivity (ODBC)

A database-independent application programming interface that enables applications to access data stored in heterogeneous relational and non-relational databases. Based on the Call-Level Interface (CLI) specification developed by X/Open's SQL Access Group and first popularized by Microsoft on the Windows platform.

out-of-line column

A column whose values are physically stored separately from the other column values of a row.

phantom

A row that appears during one read but not during another read within the same transaction, due to the actions of other concurrently executing transactions.

ping

A utility that tests the connection between two computers on a network by sending a message from one computer to the other and measuring how long it takes for the receiving system to confirm that the message was received. Typically packaged with network software.

port number

See "TCP/IP port number".

procedure

See "stored procedure".

process

An instance of a program in execution.

propagate

When using Cache Connect to Oracle, to send table or row modifications from a Cache Connect to Oracle data store to an Oracle database. Compare with "replicate".

protocol message logging

The process that the TimesTen Server uses to record each message it receives through the TimesTen network protocol.

relational database

A type of database or DBMS that stores data in tables organized as rows and columns. A relational database can perform searches by using data in specified columns of one table to find data stored in another table.

replicate

The sending of table or row modifications from one data store to another. Compare with "propagate".

result set

A collection of zero or more rows of data that represent the result of an SQL query.

rollback

To undo the actions of a transaction, thereby returning all items modified by the transaction to their original state.

row buffering

A performance enhancement used by the TimesTen Client in which the client receives multiple result rows of an SQL query in each message from the TimesTen Server to reduce network communication.

RPC

Remote Procedure Call.

scalability

The degree to which a system or application can handle increasing demands on system resources without significant performance degradation.

server data source name

See "data source name, server".

server DSN

See "data source name, server".

system DSN

See "data source name, system".

shorthand name

A logical name used to refer to a particular TimesTen Server. Shorthand names relieve the end user of having to enter a host name and port number to connect to a TimesTen Server.

SMP

Symmetric multi-processing. A hardware configuration in which two or more similar processors are connected via a high-bandwidth link and managed by one operating system, where each processor has equal access to I/O devices.

SNMP

Simple Network Management Protocol. Used to manage nodes on a network.

SQL

Structured Query Language.

stack overflow condition

An error condition in which the stack usage of a thread or process exceeds the amount of space allocated for the stack.

stored procedure

An executable object or named entity stored in a data store that can be invoked with input and output parameters and which can return result sets similar to those returned by an SQL query.

system account

A special account on Windows used by the operating system and certain operating system services. The TimesTen Service and the TimesTen Server run under the system account.

system DSN

See “data source name, system”.

T-tree

An index structure similar in functionality to a B-tree but optimized for in-memory data management.

TCP/IP

The communications protocol used by computers on the Internet. Abbreviation for Transport Control Protocol/Internet Protocol.

TCP/IP port number

A number used by TCP/IP that identifies the end point for a connection to a host that supports multiple simultaneous connections.

telnet

A utility program and protocol that enables a user on one computer to open a virtual terminal, log in to a remote host and interact as a terminal user of that host.

thread

An independent sequence of execution of program code inside a process. See also "process".

thread-safe ODBC driver

An ODBC driver that supports multithreaded servers and clients. The TimesTen data manager driver and the TimesTen Client driver are thread-safe.

timeout error

An error condition indicating that the requested operation did not complete within the given amount of time. See also "timeout interval".

timeout interval

A configuration parameter that specifies the maximum amount of time that an operation should take to complete. See also "timeout error".

TimesTen Client

(1) An ODBC driver that enables end users to access data sources through a TimesTen Server. (2) A computer on which the TimesTen Client software has been installed. Using the TimesTen Client driver, an end user or application can access any data source managed by an available TimesTen Server.

TimesTen Client/Server network protocol

The protocol used by TimesTen Clients and TimesTen Servers to exchange data over a standard TCP/IP network connection.

TimesTen Server

(1) An application program that makes TimesTen data sources available to the TimesTen Clients on a network. (2) A computer on which the TimesTen Server software is running.

TimesTen Server address

The host name or IP address used during installation of the TimesTen Server to identify the computer on which the software is being installed.

transaction

An operation or set of operations performed against data in a data store. The operation(s) defined in a transaction must be completed as a whole; if any part of the transaction fails, the entire transaction fails. See also "ACID transaction semantics".

UTF-8

A variable-length encoding scheme in which each Unicode character is represented by a sequence of bytes ranging in length from 1 to 4 bytes. Among other properties, this encoding has the property that the Unicode ASCII range values are encoded as single-byte ASCII characters.

UTF-16

An encoding scheme defined by the ISO/IEC 10646 standard in which each Unicode character is represented by either a two-byte integer or a pair of two-byte integers. Characters from European scripts and most Asian scripts are represented in two bytes. Surrogate pairs are represented in four bytes. Surrogate pairs represent characters such as infrequently used Asian characters that were not included in the original range of two-byte characters.

user account

The combination of a user name, password and access permissions that gives an individual user access to an operating system.

user data source name

See "data source name, user".

user DSN

See "data source name, user".

User Manager

A Windows utility program used to create user accounts and assign access rights and group membership.

Windows sockets (Winsock)

An API that defines a standard binary interface for TCP/IP transports on Windows platforms. This API adds Windows-specific extensions to the Berkeley Sockets interface originally defined in Berkeley UNIX.

Index

A

applying data store updates 152
attributes
    setting programmatically 17
AUTOCOMMIT
    with XA 91
autocommit
    performance impact 105

B

binding
    performance impact 102
bound data 221
buildtms command 96
bulk fetch 19, 103
bulk fetch rows 103

C

C language functions *See* Utility Library.
character restrictions, UNIX and NT 228
checking for errors 23
checkpoints
    and performance 109
Client
    ODBC driver 227
closing a transaction log API handle 154
code font 4
comparing Transaction Log API versions 195
compiling applications
    UNIX 30
    Windows 29
concurrency
    and locking 104
configuring the update buffer 156
connections
    to data store 11
contention, lock
    data store 103
conversions
    and performance 102
    C to SQL 218
    NCHAR 218
    SQL to C 218
cursors 21
    example 21

D

Data Manager
    ODBC driver 227
data store
    connecting to 11
    disconnecting from 11
    lock contention 103
data transaction processing 84
data types
    conversions and performance 102
    SQL 215
    transaction log API 197
    Unicode 215
deadlock error 79
demos
    transaction log API 39
disconnecting from data store 11
diskless logging
    and rollbacks 107
distributed transaction processing 83
DLL and UNIX shared library 228
driver manager
    linking with 28
    performance impact 100
DurableCommits
    with XA 85

E

errors
    and recovery 23
    checking for 23
    fatal 24
    non-fatal 24
event management 33
examples
    cursors 21

F

fatal errors 24
fetching multiple rows 19, 103
file descriptors 17
filenames in UNIX 228

files
makefiles 27
sqlext.h 217
sqltypes.h 217
sqlucode.h 217
unicode.c 221

G

global transactions
defined 84
recovering 86
Groff, James R. 3

H

header files 229
heuristic recovery 86

I

-I flag 29
I/O
performance 106
initializing a data store handle 178, 180
installation
default directory 3
isolation modes
and performance 105
italic font 4

J

JTA 83

K

key not found error 79

L

-L flag 30
linking applications
options 27
UNIX 30
Windows 29
with driver manager 28
with TimesTen driver 27
locks
and concurrency 104
and performance 104
data store-level 104
row-level 104
table-level 104
timeout interval and performance 108
-lodbc flag 30
Logging
with XA 86
logging
and rollbacks 108
looking up a record 172

M

makefiles 27
Melton, Jim 3

N

NCHAR
conversions 218
non-durable commits
advantages and disadvantages 107
non-fatal errors 24
non-persistent transaction log 34
NVARCHAR type 57

O

ODBC
and XA 91
driver manager and performance 100
XA support with driver manager 94
ODBC 2.5
SQLBindCol 229
SQLBindParameter 230
ODBC driver
Client driver 227
Data Manager 227
ODBC for UNIX 227
ODBC functions
supported in TimesTen 223
.odbc.ini file 228

P

performance
and isolation modes 105
and SQLGetData 102
application tuning
autocommit mode 105
checkpoints 109
ODBC driver manager 100
transaction rollback 108
transaction size 106
batch SQL operations 102

bulk fetch rows 103
lock timeout interval 108
SQL tuning
avoid excessive binds 102
prepare operations 101
persistent transaction log 34
phantoms 105
processing rows
with cursors 21

R

reading transaction log errors 162
recovery 25
global transactions 86
heuristic 86
registry and .odbc.ini file 228
replication
and XA 86
resetting the transaction log error stack 164
resetting Transaction Log counters 183
resource manager, defined 84
result set
using cursors 21
retrieving column information 167
retrieving table information 170
retrieving the Transaction Log API version 171
retrieving Transaction Log API status 187
rollback
performance impact 108
rolling back a transaction 184
row-level locking
and logging 108

S

Sanders, Roger E 3
sb_ErrXlaTupleMismatch error 80, 152
setting the Transaction Log API version 186
Signore, Robert 3
Simon, Alan R 3
sizing
transactions 106
SQL
batching and performance 102
reference reading 3
SQL_C_DEFAULT 216
SQL_WCHAR 215
SQL_WVARCHAR 215
SQLBindCol 215, 229
performance impact 102
SQLBindParameter 215, 230
performance impact 102
SQLConnect 11
SQLDescribeCol 215
SQLDisconnect 11
SQLDriverConnect
example code usage 17
SQLEXT.H 229
sqlext.h 217
SQLGetData
and performance 102
returning SQLSTATE 22012 230
SQLGetTypeInfo 215
SQL.H 229
SQLPrepare
performance impact 101
returning SQLSTATE S1C00 230
SQLSetConnectOption 19, 103
SQLSetStatementOption 19, 103
SQLSetStmtOption 19
SQLSTATE 22008 230
SQLSTATE 22012 230
SQLSTATE S1C00 230
SQLTYPES.H 229
sqltypes.h 217
sqlucode.h 217
Stegman, Michael O. 3
Sun Workshop C/C++ 31

T

table status 191
timeout error 79
TimesTen
installing 3
transaction branch, defined 84
transaction log API
and views 37
bookmarks 37
building an application 41
data types 197
demos 39
for replication 76
handling errors 67
modes 34
overview 33
retrieving table information 188
retrieving update data 174
ttXlaAcknowledge 151

ttXlaApply 152
ttXlaClose 154
ttXlaConfigBuffer 156
ttXlaDateToODBCCType 158
ttXlaDecimalToCString 159
ttXlaDeleteBookmark 161
ttXlaError 162
ttXlaErrorRestart 164
ttXlaGenerateSQL 165
ttXlaGetColumnInfo 167
ttXlaGetLSN 169
ttXlaGetTableInfo 170
ttXlaGetVersion 171
ttXlaLookup 172
ttXlaNextUpdate 174
ttXlaNextUpdateWait 176
ttXlaOpenTimesTen 178
ttXlaPersistOpen 180
ttXlaResetStatus 183
ttXlaRollback 184
ttXlaSetLSN 185
ttXlaSetVersion 186
ttXlaStatus 187
ttXlaTableByName 188
ttXlaTableCheck 189
ttXlaTableStatus 191
ttXlaTimeStampToODBCCType 194
ttXlaTimeToODBCCType 193
ttXlaVersionCompare 195
transaction manager, defined 84
transaction rollback 184
performance impact 108
transaction size
performance impact 106
tt_ErrBadXlaRecord 68
tt_ErrCacheXlaNotRead 68
tt_ErrCacheXLARollbackFailed 68
tt_ErrCondLockConflict 68
tt_ErrDbAllocFailed 68
tt_ErrDeadlockVictim 68
tt_ErrDeadlockVictim error 79
tt_ErrPermSpaceExhausted 68
tt_ErrTempSpaceExhausted 68
tt_ErrTimeoutVictim 68
tt_ErrTimeoutVictim error 79
tt_ErrXlaBookmarkBad 68
tt_ErrXlaBookmarkUsed 68
tt_ErrXlaDedicatedConnection 68
tt_ErrXlaLsnBad 68
tt_ErrXlaNoLogging 68
tt_ErrXlaNoSQL 68
tt_ErrXlaParameter 68
tt_ErrXlaTableDiff 68
tt_ErrXlaTableSystem 68
tt_ErrXlaTupleMismatch 68
TT_PREFETCH_COUNT 19, 103
TT_UPDCOLS flag 201
TT_UPDCOMMIT flag 201
TT_UPDFIRST flag 201
TT_UPDREPL flag 201
tt_xa_context() function 89
tt_xa_switch 93
tt_XlaLsn_h 213
ttApplicationContext
using 82
ttDestroyDataStore 119, 121
ttDurableCommit
with XA 85
ttRamGrace 123
ttRamLoad 124
ttRamPolicy 125
ttRamUnload 127
ttRepDuplicate 128, 129
ttxadm43.dll library 94
ttXlaAcknowledge 151
using 47
ttXlaAddColumnTup_t 204
ttXlaApply 152
using 78
ttXlaClose 154
using 70
ttXlaColDesc_t 211
using 53
ttXlaCommit 155
using 79
ttXlaConfigBuffer 156
using 73, 74
ttXlaCreateIndexTup_t 203
ttXlaDateToODBCCType 158
using 60
ttXlaDecimalToCString 159
using 60
ttXlaDeleteBookmark 161
using 72
ttXlaDropColumnTup_t 205
ttXlaDropindexTup_t 204
ttXlaDropTableTup_t 203
ttXlaError 162

using 67
ttXlaErrorRestart 164
using 67
ttXlaGenerateSQL 165
using 80
ttXlaGetColumnInfo 53, 167
using 53
ttXlaGetLSN 169
using 79
ttXlaGetTableInfo 170
using 54
ttXlaGetVersion 171
using 45
ttXlaHandle_h XLA handle 45, 74
ttXlaLookup 172
ttXlaNextUpdate 174
using 47
ttXlaNextUpdateWait 176
using 47
ttXlaNodeHdr_t 198
ttXlaOpenTimesTen 178
using 74
ttXlaPersistOpen 180
using 45
ttXlaResetStatus 183
using 73, 76
ttXlaRollback 184
using 79
ttXlaSetColumnTup_t 207
ttXlaSetLSN 185
using 79
ttXlaSetStatusTup_t 207
ttXlaSetTableTup_t 206
ttXlaSetVersion 186
using 45
ttXlaStatus 187
using 73, 75
ttXlaStatus_t 208
using 75
ttXlaTableByName 188
using 46, 68
ttXlaTableCheck 189
using 77
ttXlaTableStatus 191
using 46
ttXlaTblDesc_t 210
ttXlaTimeStampToODBCCType 194
using 60
ttXlaTimeToODBCCType 193
using 60
ttXlaTruncateTableTup_t 203
ttXlaUpdateDesc_t 51, 199
TT_UPDCOLS flag 201
TT_UPDCOMMIT flag 201
TT_UPDFIRST flag 201
TT_UPDREPL flag 201
using 47, 51, 53
ttXlaUpdateDesc_t type 51, 53, 199
ttXlaVersion_t 209
ttXlaVersionCompare 195
Tuxedo, configuring for XA 95
two-phase commit protocol 85
typographical conventions 4

U

UBBCONFIG file 96
Unicode 215
C string escapes 221
reference reading 3
Unicode Consortium 3
unicode.c 221
UNIX
filenames 228
ODBC 227
update conflicts 80
Utility Library
described 113
ttDestroyDataStore 119, 121
ttRamGrace 123
ttRamLoad 124
ttRamPolicy 125
ttRamUnload 127
ttRepDuplicate 128, 129

V

VARBINARY type 57
VARCHAR type 57
verifying a data store handle 189

W

Weinberg, Paul N. 3
working with
cursors 21

X

X/Open DTP model 84
XA 83

XA and replication 86
xa_close() function 88
xa_open() function 88
xa_switch_t 92
XID parameter 88
XLA *See* transaction log API.