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

Release 7.0

B31681-02

ORACLE

TIMESTEN

April 2007

Printed in the United States of America

Contents

5 JDBC Reference

6 JMS/XLA Reference

Index

About this Guide

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

This guide is for application developers who use and administer Oracle TimesTen In-Memory Database JDBC and for system administrators who configure and manage the Oracle TimesTen In-Memory Database daemon.

To work with this guide, you should understand how database systems work. You should also have knowledge of SQL (Structured Query Language) and JDBC (Java Database Connectivity). See "Background reading" on page 2 if you are not familiar with these interfaces.

TimesTen documentation

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

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

Book Titles	Description
Oracle TimesTen In-Memory Database Installation Guide	Contains information needed to install and configure Oracle TimesTen In-Memory Database on all supported platforms.
Oracle TimesTen In-Memory Database Introduction	Describes all the available features in the Oracle TimesTen In-Memory Database.
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.
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 Oracle TimesTen In-Memory Database to develop and implement applications that use Oracle TimesTen In-Memory Database.

Oracle TimesTen In-Memory Database API Reference Guide	Describes all TimesTen utilities, procedures, APIs and provides a reference to other features of TimesTen.
Oracle TimesTen In-Memory Database SQL Reference Guide	Contains a complete reference to all TimesTen SQL statements, expressions and functions, including TimesTen SQL extensions.
Oracle TimesTen In-Memory Database Error Messages and SNMP Traps	Contains a complete reference to the TimesTen error messages and information on using SNMP Traps with TimesTen.
Oracle TimesTen In-Memory Database TTClasses Guide	Describes how to use the TTClasses C++ API to use the features available in Oracle TimesTen In-Memory Database to develop and implement applications.
TimesTen to TimesTen Replication Guide	Provides information to help you understand how Oracle TimesTen In-Memory Database Replication works and step-by-step instructions and examples that show how to perform the most commonly needed tasks. This guide is for application developers who use and administer Oracle TimesTen In-Memory Database and for system administrators who configure and manage Oracle TimesTen In-Memory Database Replication.
TimesTen Cache Connect to Oracle Guide	Describes how to use Cache Connect to cache Oracle data in TimesTen data stores. This guide is for developers who use and administer TimesTen for caching Oracle data.
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.

Background reading

For a Java reference, see:

- Horstmann, Cay and Gary Cornell. *Core Java(TM) 2, Volume I--Fundamentals (7th Edition) (Core Java 2)*. Prentice Hall PTR; 7 edition (August 17, 2004).

A list of books about ODBC and SQL is in the Microsoft ODBC manual included in your developer's kit. Your developer's kit includes the appropriate ODBC manual for your platform:

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

- Kyle Geiger. *Inside ODBC*. Redmond, WA: Microsoft Press. 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, Second Edition.* McGraw-Hill Osborne Media. 2002.

For information about Unicode, see:

- The Unicode Consortium, *The Unicode Standard, Version 5.0*, Addison-Wesley Professional, 2006.
- The Unicode Consortium Home Page at http://www.unicode.org

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.` or `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 Oracle TimesTen In-Memory Database 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 with an appropriate value.

`[ ]`	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 Oracle TimesTen In-Memory Database 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 Oracle TimesTen In-Memory Database release number, with or without a dot. For example, 60 or 7.0 represents Oracle TimesTen In-Memory Database Release 7.0.
`jdk_version`	Two digits that represent the version number of the major JDK release. Specifically, 14 represent JDK 1.4; 5 represents JDK 5.

`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.

Technical Support

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

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

1

Configuring the Java Development Environment

This chapter describes how to install, configure, and test your TimesTen application development environment. It includes the following topics:

- Installing TimesTen and the JDK
- Setting the Java environment variables
- Compiling and executing Java applications
- About the TimesTen Java demos

Installing TimesTen and the JDK

Install and configure TimesTen for your environment, as described in the *Oracle TimesTen In-Memory Database Installation Guide*, and the Java JDK, as described in your Java installation guide. The topics of particular interest in the *Oracle TimesTen In-Memory Database Installation Guide* when setting up a Java development environment include:

- Access Control
- JDK support
- Client/Server configurations
- Environment modifications

After you have installed and configured TimesTen, create a data store DSN as described in the *Oracle TimesTen In-Memory Database Operations Guide*. The topics of particular interest include:

- TimesTen JDBC driver
- User and system DSNs
- Data Manager and Client DSNs
- Thread programming with TimesTen
- Creating a DSN on UNIX or Creating a DSN on Windows

Setting the Java environment variables

The environment variable settings for TimesTen are explained in "Environment modifications" in the *Oracle TimesTen In-Memory Database Installation Guide*. This section provides more detail on those that impact the environment for TimesTen Java applications.

On UNIX platforms, you can set all of the environment variables described in this section by sourcing one of the following scripts:

install_dir/bin/ttSetEnv.sh

install_dir/bin/ttSetEnv.csh

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

install_dir\bin\ttenv.bat

The rest of this section describes values the environment variables are set to, as well as how to set them manually, if necessary.

Set CLASSPATH

Java classes and class libraries are found on CLASSPATH. Before executing a Java program that loads any of the TimesTen JDBC drivers, the CLASSPATH environment variable must contain the class library file:

install_dir/lib/classes*jdk_ver*.jar

jdk_ver indicates the version of the JDK that you are using. For example, for JDK 1.4, *jdk_ver* is 14. For JDK 5.0, *jdk_ver* is 5.

Note: If more than one jar file is listed in the CLASSPATH, make sure the TimesTen jar file is listed first.

On UNIX, CLASSPATH elements are separated by colon. For example:

```
set CLASSPATH .:/opt/TimesTen/tt70/lib/ttjdbc14.jar
```

or

```
setenv CLASSPATH .:/opt/TimesTen/tt70/lib/ttjdbc14.jar
```

On Windows, CLASSPATH elements are separated by semicolons. Also, on Windows, do not use quotes when setting the CLASSPATH environment variable even if a directory pathname contains spaces.

For example, this is correct:

```
set CLASSPATH=.;C:/TimesTen/tt70/lib/ttjdbc14.jar
```

This is incorrect:

```
set CLASSPATH=.;"C:/TimesTen/tt70/lib/ttjdbc14.jar"
```

If in doubt about the JDK version you have installed on your system, enter:

```
> java -version
```

If you are going to use the JMS/XLA interface described in Chapter 3, "Using JMS/XLA for Event Management", then you also need to add the following to your CLASSPATH:

install_dir`/lib/timestenjmsxla.jar`
install_dir`/3rdparty/jms1.1/lib/jms.jar`

For example, your CLASSPATH would look like:

```
.:C:/TimesTen/tt70/lib/ttjdbc14.jar:C:/TimesTen/tt70/lib/
timestenjmsxla.jar:C:/TimesTen/tt70/3rdparty/jms1.1/lib/jms.jar
```

By default, JMS/XLA looks for a configuration file called `jmsxla.xml` in the current working directory. If you want to use another name or location for the file, you need to specify it as part of the environment variable in the **InitialContext** class and add the location to CLASSPATH. See "XLA configuration file and topics" on page 52 for more information about the `jmsxla.xml` configuration file.

Set the shared library path variable

Before running a java program that loads the TimesTen JDBC driver, the `shared library path for your system` environment variable must be set to include the TimesTen *`install_dir`*`/lib directory`. The name of the variable used for the shared library path depends on the system used:

System	Name of Variable
Linux	LD_LIBRARY_PATH
Solaris	LD_LIBRARY_PATH
HPUX	SHLIB_PATH or LD_LIBRARY_PATH
AIX	LIBPATH
Windows	PATH

See "Shared library path environment variable" in the *Oracle TimesTen In-Memory Database Installation Guide* for details on setting the shared library path.

Set the THREADS_FLAG variable (UNIX only)

The TimesTen JDBC driver uses native threads; green threads are not supported.

On some UNIX platforms, in order to use the native threads package, you must set the environment variable THREADS_FLAG to native. How you set the flag depends on your shell.

In csh, the syntax is:

```
setenv THREADS_FLAG native
```

In sh, the syntax is:

```
THREADS_FLAG=native
export THREADS_FLAG
```

Set the PATH variable

Make sure the executables **javac** and **java** are both on your executable search path, or will need to invoke them using absolute paths.

Compiling and executing Java applications

To compile a Java program, at your shell or command prompt use the command:

`javac` *`SourceFile`*`.java`

The command generates the bytecode file *`SourceFile`*`.class` if the `.java` file contains a public class. A `.class` file is generated for all classes defined in *`SourceFile`*`.java`. By default the `.class` files reside in the same directory as the `.java` source files. To specify a different target directory for the `.class` files, use the command:

`javac -d` *`Directory SourceFile`*`.java`

The class name is the same as the filename prefix of its corresponding `.class` file. To execute a Java program, at your shell or command prompt use the command:

`java` *`ClassName`*

`ClassName` is the name of a class that contains a `main` method. This command starts the Java Virtual Machine (JVM) that will interpret and execute the Java bytecode in the `.class` file and any other bytecode files that it is dependent upon.

Example 1.1 To compile the `level1.java` demo and execute it using the *demo* data store, enter:

```
> cd install_dir/demo/tutorial/java
> javac level1.java
> ttIsql -f ../datfiles/input0.dat demo
> java level1 demo
```

About the TimesTen Java demos

Once you have configured your Java environment, you can confirm that everything is set up correctly by compiling and running the TimesTen Java demo applications in the *install_dir*/demo/tutorial/java directory.

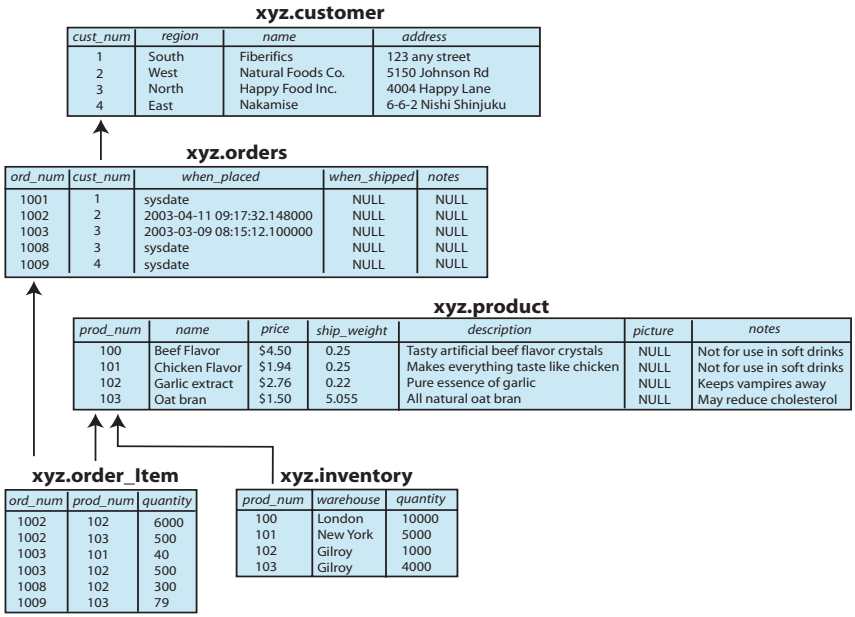
About the TimesTen demo schema

The TimesTen Java demos are designed to work with the TimesTen demo schema, which simulates a simple order-processing database. You can populate a data store with the TimesTen demo schema by running the *install_dir*/demo/tutorial/datfiles/input0.dat, as described in "Executing the TimesTen Java demos" on page 15. The input0.dat file creates the following tables:

- xyz.product
- xyz.inventory
- xyz.customer
- xyz.orders
- xyz.order_item

The tables in the demo schema are organized and populated with data, as shown in Figure 1.1.

Figure 1.1 TimesTen Demo Schema



What the TimesTen demos do

The TimesTen Java demos are named `level1.java`, `level2.java`, `level3.java`, `level4.java`, `XlaLevel1.java`, `XlaLevel2.java`, and `XlaLevel3.java`. All of the `level` demos support both direct and client connections to the data store.

The `level1` demo uses the **DriverManager** interface to connect to a data store and forms a prepared INSERT and SELECT statement to insert new customer data into the `xyz.customer` table and then view the contents of the table. It executes the INSERT until all of the data in the `input1.dat` file is loaded into the table, executes the SELECT, fetches and prints the result set to `stdout` and disconnects from the data store.

The `level2` demo uses the **DataSource** interface to connect to a data store and forms prepared INSERT, UPDATE, DELETE, and SELECT statements to insert, update, delete, and view product data in the xyz.product table. It executes the INSERT until all of the data in the `input2.dat` file is loaded into the table. It executes the DELETE to delete any duplicate product data and then the UPDATE to increase the price of the products in the table by 10%. It executes a **ttCkpt** procedure to checkpoint the data store to disk, executes the SELECT, fetches and prints the result set to `stdout` and then disconnects from the data store.

The `level3` demo uses the **DataSource** interface to connect to a data store and forms prepared statements to perform order processing operations on the order data in the `input3.dat` file. For each order item in the data file, the demo performs the following transaction:

- INSERT the new order into the xyz.orders and `xyz.order_item` tables.
- SELECT from the `xyz.inventory` table to check the available quantity of the ordered item in the inventory.
- UPDATE the `xyz.inventory` table to debit the ordered item from the inventory.

If there are not enough items in the inventory, the demo rolls back the entire transaction and reports that there is insufficient inventory for the order. Finally, the demo checkpoints the data store to disk and disconnects.

The `level4` demo processes the same orders as `level3`, only it uses multiple threads and multiple connections to increase throughput.

Both the `level1` and `level2` demos call the TimesTen **ttOptUpdateStats** built-in procedure to update the statistics for the `customer` and `product` tables. The **ttOptUpdateStats** procedure stores the statistics in the SYS.COL_STATS and SYS.TBL_STATS tables for use by the TimesTen query optimizer to enable more efficient query execution.

The `XlaLevel1.java`, `XlaLevel2.java`, and `XlaLevel3.java` demos use the JMS/XLA API described in Chapter 3, "Using JMS/XLA for Event Management" to monitor and report on specific updates to the data store. The `XlaLevel1.java` and `XlaLevel2.java` demos monitor updates to the `xyz.customer` table. The `XlaLevel3.java` demo monitors updates to a user-specified table.

Compiling the TimesTen Java demos

To compile the Java demos, go to the Java demo directory and run ANT on the `build.xml` file. If you do not want to use ANT, use the `javac` command to compile each demo. For example:

```
> cd /TimesTen/tt70/demo/tutorial/java
> javac *.java
```

Executing the TimesTen Java demos

Prior to executing any of the Java demos, you must execute the SQL statements in the `input0.dat` file, as shown below, to build or rebuild the demo schema. Each demo requires that you specify a DSN name. The DSN can be for either a direct connection or client connection to the data store.

Executing the level demos

All of the level demos have the following command syntax:

```
demoname [-t] [-d | -c] {DSN}
demoname -h | -help

  -h | -help    print usage and exit
  -d            connect using direct driver (default)
  -c            connect using client driver
  -t            enable JDBC tracing
  DSN           name of the data store
```

Example 1.2

In this example, we execute the `level1` and `level2` demos using a *direct driver* connection to the *DMdemo* data store. We enable JDBC tracing when executing the `level2` demo. Last, we execute the `level3` demo using a client connection to the *CSdemo* data store.

Note: Before executing each demo, you must execute the `input0.dat` file to rebuild the demo schema on the data store.

```
> ttIsql -f ../datfiles/input0.dat DMdemo
...... output
> java level1 DMdemo
...... output
> ttIsql -f ../datfiles/input0.dat DMdemo
...... output
> java level2 -t DMdemo
...... output
> ttIsqlCS -f ../datfiles/input0.dat CSdemo
...... output
> java level3 -c CSdemo
...... output
```

If you cannot connect to the data store, you may not have configured the DSN name that you are specifying. See "Data source names" in the *Oracle TimesTen In-Memory Database Operations Guide*.

Executing the XlaLevel demos

Note: You only need to look at the XlaLevel demos if you are going to be using the JMS/XLA API described in Chapter 3, "Using JMS/XLA for Event Management."

In JMS/XLA, the name of the data store and other parameters used by XLA applications are specified in the form of *XLA topics*, as described in "XLA configuration file and topics" on page 52. The XlaLevel demos obtain their XLA topics from the `jmsxla.xml` file located in the *`install_dir`*`/demo/tutorial/java` directory. The topics in the `jmsxla.xml` file are configured so that the XlaLevel demos use the predefined **RunData_*TTinstance*** data store and other default parameters. You can edit the `jmsxla.xml` file to specify other topics or change the settings in the exiting topics.

Note: The demos in this document use the predefined data store **RunData_tt70**.

All of the XlaLevel demos accept an optional topic name.

The syntax for executing `XlaLevel1` is:

```
XlaLevel1 [topic]
```

where *topic* defaults to the *Level1Demo* topic that is prespecified in the `jmsxla.xml` file.

The syntax for executing `XlaLevel2` is:

```
XlaLevel2 [topic [bookmark]]
```

where *topic* defaults to *Level2Demo* and *bookmark* defaults to *bookmark*.

The syntax for executing `XlaLevel3` is:

```
XlaLevel3 [topic [bookmark [table]]]
```

where *topic* defaults to *Level3Demo*, *bookmark* defaults to *bookmark*, and *table* defaults to *tbl*.

Prior to executing any of the `XlaLevel1` and `XlaLevel2` demos, you must execute the SQL statements in the `input0.dat` file to build or rebuild the demo schema. Both the `XlaLevel1` and `XlaLevel2` demos monitor changes to the `xyz.customer` table.

When running the `XlaLevel1` demo, you can make updates to the `xyz.customer` table by running the `level1.java` demo in a separate shell.

Example 1.3 To run the `XlaLevel1.java` demo with its default topic, in one shell enter:

```
> ttIsql -f ../datfiles/input0.dat RunData_tt70
...... output
> java XlaLevel1
...... detected changes to the xyz.customer table
```

In another shell, enter:

```
> java level1 RunData_tt70
...... output
```

The output from the `XlaLevel1` demo shows the detected changes to the `xyz.customer` table.

If you create a new topic in the `jmsxla.xml` file, you can specify that when you enter run the `XlaLevel1.java` demo. For example, if you created a new topic, named *MyTopic*, you would start the `XlaLevel1.java` demo with:

```
> java XlaLevel1 MyTopic
```

The `XlaLevel2` demo prompts you to enter changes to the `xyz.customer` table in the form of SQL from the command line. It then displays the detected changes to the table after you commit the transaction.

Example 1.4 To run the `XlaLevel2.java` demo with its default topic and bookmark, enter:

```
> ttIsql -f ../datfiles/input0.dat RunData_tt70
...... output
> java XlaLevel2
+++ Using default topic Level2Demo and default bookmark bookmark
+++ create session
+++ create topic
+++ createDurableSubscriber
+++ using connection string
'DSN=RunData_tt70;ExclAccess=0;Logging=1;LogPurge=0;Overwrite=0'
+++ connecting to
jdbc:timesten:direct:DSN=RunData_tt70;ExclAccess=0;Logging=1;LogPurge=0;Overwrit
e=0
+++ turning off autocommit
You can now enter SQL commands.  You should enter
either DML (such as inserts, updates, or deletes),
or DDL (such as CREATE SEQUENCE).
```

```
For instance, try:
  create sequence s minvalue 1000
  insert into xyz.customer values(s.nextVal,'us','Bob','nowhere')
  commit
After each SQL command you enter, the demo tries
to get and display any JMS/XLA updates.
Type "quit" to exit the demo, or "help"
to see this message again.
NOTE: autocommit is turned off, so you will
have to enter "commit" to see your updates.
Enter SQL: create sequence s minvalue 1000
+++ create sequence s minvalue 1000
Enter SQL: insert into xyz.customer values(s.nextVal,'us','Bob','nowhere')
+++ insert into xyz.customer values(s.nextVal,'us','Bob','nowhere')
Enter SQL: insert into xyz.customer values(s.nextVal,'us','Bob','nowhere')
+++ insert into xyz.customer values(s.nextVal,'us','Bob','nowhere')
Enter SQL: commit
+++ commit
>>> got a CREATE SEQUENCE message
  CYCLE=true  INCREMENT=1  MAX_VALUE=9223372036854775807  MIN_VALUE=1000
  NAME=S  OWNER=ASPIN  __COMMIT=false  __CONTEXT=(null)  __FIRST=true
  __REPL=false  __TYPE=16  __mtyp=null  __mver=1144080
>>> got a INSERT message
  ADDRESS=nowhere  CUST_NUM=1000  NAME=Bob  REGION=us     __NULLS=
  __TYPE=10  __mtyp=null  __mver=1146360
>>> got a INSERT message
  ADDRESS=nowhere  CUST_NUM=1001  NAME=Bob  REGION=us     __COMMIT=true
  __NULLS=  __TYPE=10  __mtyp=null  __mver=1147248
Enter SQL: quit
+++ cleaning up
+++ Subscriber close
+++ Producer.close
+++ done
+++ shutting down...
```

The `XlaLevel3` demo prompts you to specify the table you wish to make changes to and to monitor. If the table doesn't exist in the *RunData_tt70* data store, it is created.

Example 1.5 To run the `XlaLevel3.java` demo with its default topic, *Level3Demo*; a bookmark named *bkmk*, and to create a new table, named *tbl*, enter:

```
> java XlaLevel3 Level3Demo bkmk tbl
+++ topic=Level3Demo, bookmark=bkmk, table=tbl
May 11, 2005 3:32:26 PM
com.timesten.dataserver.jmsxla.SimpleInitialContextFactory getInitialContext
INFO: Using configuration file jmsxla.xml
+++ create session
+++ create topic
May 11, 2005 3:32:27 PM com.timesten.dataserver.jmsxla.DestinationImpl <init>
FINE: Properties for topic Level3Demo:{xlaPrefetch=100, name=Level3Demo,
connectionString=DSN=RunData_tt70}
+++ createDurableSubscriber
May 11, 2005 3:32:27 PM com.timesten.dataserver.jmsxla.XlaSubscriber <init>
FINE: Making XLA subscription, connstr=DSN=RunData_tt70, bookmark=bkmk,
prefetch=100, ackMode=1 May 11, 2005 3:32:27 PM
com.timesten.dataserver.jmsxla.MessageConsumerImpl createXlaSubscriber
FINE: Creating MessageConsumer with connection string=DSN=RunData_tt70,
bookmark=bkmk May 11, 2005 3:32:27 PM
com.timesten.dataserver.jmsxla.XlaSubscriber start
FINE: Starting XLA subscription
+++ using connection string 'DSN=RunData_tt70'
+++ connecting to jdbc:timesten:direct:DSN=RunData_tt70
+++ turning off autocommit
table tbl already exists
+++ {call ttXlaSubscribe('tbl', 'bkmk')}
You can now enter SQL commands.  You should enter either DML (such as inserts,
updates, or deletes), or DDL (such as CREATE SEQUENCE).
For instance, try:
  create sequence s minvalue 1000
  insert into tbl values(s.nextVal)
  call ttApplicationContext('inserted something')
  commit
After each SQL command you enter, the demo tries to get and display any JMS/XLA
updates.
Type "quit" to exit the demo, or "help"
to see this message again.
NOTE: autocommit is turned off, so you will have to enter "commit" to see your
updates.
Enter SQL: create sequence s minvalue 1000
+++ create sequence s minvalue 1000
Enter SQL: insert into tbl values(s.nextVal)
+++ insert into tbl values(s.nextVal)
```

```
Enter SQL: call ttApplicationContext('inserted something')
+++ call ttApplicationContext('inserted something')
Enter SQL: commit
+++ commit
>>> got a CREATE TABLE message
  NAME=TBL  OWNER=ASPIN  _A_INPRIMARYKEY=null  _A_NULLABLE=null
  _A_OUTOFLINE=null  _A_PRECISION=null  _A_SCALE=null  _A_SIZE=null
  __COMMIT=null  __FIRST=null  __TYPE=null  __mtyp=null  __mver=null
>>> got a CREATE SEQUENCE message
  CYCLE=true  INCREMENT=1  MAX_VALUE=9223372036854775807  MIN_VALUE=1000
  NAME=S  OWNER=ASPIN  __COMMIT=false  __CONTEXT=(null)  __FIRST=true
  __REPL=false  __TYPE=16  __mtyp=null  __mver=385368
>>> got a INSERT message
  A=1000  __NULLS=B  __TYPE=10  __mtyp=null  __mver=386576
>>> got a COMMIT ONLY message
  __COMMIT=true  __CONTEXT=inserted something  __FIRST=false
  __REPL=false  __TYPE=13  __mtyp=null  __mver=386880 Enter SQL: create index ix
on tbl(a)
+++ create index ix on tbl(a)
Enter SQL: commit
+++ commit
>>> got a CREATE INDEX message
  COLUMNS=A  HASH_PAGES=0  INDEX_METHOD=T  INDEX_TYPE=R  IXNAME=IX
  TBLNAME=TBL  TBLOWNER=ASPIN  UNIQUE=false  __COMMIT=true
  __CONTEXT=(null)  __FIRST=true  __REPL=false  __TYPE=3  __mtyp=null
  __mver=392904
Enter SQL: drop index ix
+++ drop index ix
Enter SQL: insert into tbl values(s.nextVal)
+++ insert into tbl values(s.nextVal)
Enter SQL: insert into tbl values(s.nextVal)
+++ insert into tbl values(s.nextVal)
Enter SQL: update tbl set a=a+10
+++ update tbl set a=a+10
Enter SQL: commit
+++ commit
>>> got a DROP INDEX message
  INDEX_NAME=IX  OWNER=ASPIN  TABLE_NAME=TBL  __COMMIT=false
  __CONTEXT=(null)  __FIRST=true  __REPL=false  __TYPE=4  __mtyp=null
  __mver=398776
>>> got a INSERT message
  A=1001  __NULLS=B  __TYPE=10  __mtyp=null  __mver=399472
>>> got a INSERT message
  A=1002  __NULLS=B  __TYPE=10  __mtyp=null  __mver=400112
>>> got a UPDATE message
  A=1010  _A=1000  __NULLS=_B;B  __TYPE=11  __UPDCOLS=A  __mtyp=null
  __mver=400712
```

```
>>> got a UPDATE message
  A=1011  _A=1001  __NULLS=_B;B  __TYPE=11  __UPDCOLS=A  __mtyp=null
  __mver=401256
>>> got a UPDATE message
  A=1012  _A=1002  __COMMIT=true  __NULLS=_B;B  __TYPE=11  __UPDCOLS=A
  __mtyp=null  __mver=401800
Enter SQL: quit
+++ cleaning up
+++ Subscriber close
May 11, 2005 3:35:04 PM com.timesten.dataserver.jmsxla.XlaSubscriber
tableUnsubscribe
FINE: Unsubscribing from table TBL
+++ Producer.close
+++ done
+++ shutting down...
```

Problems executing the TimesTen Java demo programs

If you receive an error message like:

```
java.lang.UnsatisfiedLinkError:no ttJdbcCS
```

or

```
java.lang.UnsatisfiedLinkError: no ttJdbc in java.library.path
```

then you do not have LD_LIBRARY_PATH set properly. Find `libttJdbc.so` and put that directory on the LD_LIBRARY_PATH:

```
setenv LD_LIBRARY_PATH install_dir/lib
```

Problems compiling the TimesTen Java demo program

If you receive the error message:

```
java.lang.ClassNotFoundException:com.timesten.jdbc.TimesTenDriver
```

CLASSPATH is not set properly. Find the classes archive file and make sure that it is on the CLASSPATH, for example:

```
setenv CLASSPATH install_dir/lib/ttjdbc14.jar
```

If you get a ClassNotFoundException for a class defined in one of the demos (such as ‘level1’), make sure the current directory is included in your CLASSPATH. For example:

```
setenv CLASSPATH install_dir/lib/ttjdbc14.jar:.
```

2

Working with TimesTen Data Stores

This chapter describes the basic procedures for writing a Java application to access data in a TimesTen data store. Before attempting to write a TimesTen application, be sure you have completed the following prerequisite tasks:

Prerequisite Task	What you do
Create a TimesTen data store	Follow the procedures described in Chapter 1, "Creating TimesTen Data Stores" in the *Oracle TimesTen In-Memory Database Operations Guide*.
Configure Java environment	Follow the procedures described in "Setting the Java environment variables" on page 8.
Compile and execute the TimesTen Java demos	Follow the procedures described in "About the TimesTen Java demos" on page 12.

After you have successfully executed the TimesTen Java demos, your development environment is set up correctly and ready for you to create applications that accesses a TimesTen data store.

Topics in this chapter are:

Java classes

Most TimesTen applications can be written using the supported Java classes and interfaces listed in Chapter 5, "JDBC Reference."

You must import the standard JDBC packages in any java program that use JDBC:

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

If you are going to make use of the **DataSource** interface, you must also import the optional JDBC packages:

```
import javax.sql.*;
```

Though you can accomplish most operations with the standard Java interfaces, TimesTen provides the following extensions to the Java standard.

To use the TimesTen implementation of the **javax.sql.DataSource** interface, import:

```
import com.timesten.jdbc.DataSource;
```

To use the TimesTen implementation of the **javax.sql.XADataSource** interface, import:

```
import com.timesten.jdbc.xa.TimesTenVendorCode;
```

To use the TimesTen connection-based prefetch feature described in "Fetching multiple rows of data" on page 35, import:

```
import com.timesten.sql.TimesTenConnection;
```

See "TimesTen extensions to JDBC" on page 80 for more information on these TimesTen extentions.

Connecting to a TimesTen data store

The *Oracle TimesTen In-Memory Database Operations Guide* describes how to create a DSN to define a connection to a TimesTen data store. The type of DSN you create depends on whether your application connects directly to the data store or connects by a client. 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*.

After you have created a DSN, the application can connect to the data store. This section describes how to create a JDBC connection to a data store using either the JDBC direct driver or the JDBC client driver.

The operations described in this section are based on the `level1.java` demo.

The procedures for connecting to a TimesTen data store are:

- Load the TimesTen driver
- Create a connection URL for the data store
- Connect to the data store
- Disconnect from the data store
- Putting it all together: preparing and executing SQL

Load the TimesTen driver

The TimesTen JDBC driver must be loaded before it is available for making connections with a TimesTen data store. The TimesTen JDBC driver is:

```
com.timesten.jdbc.TimesTenDriver
```

If you are using the **DriverManager** interface to connect to TimesTen, call the **Class.forName()** method to load the TimesTen JDBC driver. This method creates an instance of the TimesTen Driver and registers it with the driver manager.

If you are using the **TimesTenDataSource** interface, you do not need to call **Class.forName()**.

Example 2.1 To identify and load the TimesTen driver:

```
Class.forName("com.timesten.jdbc.TimesTenDriver");
```

Note: If the TimesTen JDBC driver is not loaded, an error is returned when the application attempts to connect to a TimesTen data store.

Create a connection URL for the data store

To create a JDBC connection, you need to specify a TimesTen connection URL for the data store. The format of a TimesTen connection URL is:

```
jdbc:timesten:{direct | client}:dsn=DSNname;[DSNattributes;]
```

For example, to create a direct connection to the *demo* data store, the URL looks similar to the following:

```
String URL = "jdbc:timesten:direct:dsn=demo";
```

Specifying data store attributes in the connection URL

You can programmatically set or override the connection attributes in the DSN description by specifying attributes in the connection URL.

For example, to set the **LockLevel** DSN attribute to '1', you could create a URL like:

```
String URL = "jdbc:timesten:direct:dsn=demo;LockLevel=1";
```

Connect to the data store

After you have defined a URL, you can use either the **DriverManager.getConnection()** or **TimesTenDataSource.getConnection()** method to connect to the TimesTen data store.

If you use the **DriverManager.getConnection()** method, specify the driver URL to connect to the data store:

```
import java.sql.*;
Connection con = DriverManager.getConnection(URL);
```

To use the **DataSource.getConnection()** method, first create a **DataSource**. Then use the **DataSource.setUrl()** method to set the URL and **DataSource.getConnection()** to connect:

```
import com.timesten.jdbc.TimesTenDataSource;

TimesTenDataSource ds = new TimesTenDataSource();
ds.setUrl(URL);
con = ds.getConnection();
```

Either method returns a **Connection** object (*con* in this example) that you can use as a handle to the data store. See the `level1` demo for an example on how to use **DriverManager.getConnection()** and the `level2` and `level3` demos for examples of using **DataSource.getConnection()**.

Disconnect from the data store

When you are finished accessing the TimesTen data store, call the **Connection.close()** method to close the connection to the data store.

If an error has occurred, you may want to roll back the transaction before disconnecting from the data store. See "Handling non-fatal errors" on page 43 and "Rolling back failed transactions" on page 47 for more information.

Opening and closing a direct driver connection

Example 2.2 shows the general framework for an application that uses the **DriverManager** to create a direct driver connection to the *demo* data store; execute some SQL, and then close the connection. See the `level1.java` demo for a working example.

Example 2.2

```
String URL = "jdbc:timesten:dsn=demo";
Connection con = null;

try {
     Class.forName("com.timesten.jdbc.TimesTenDriver");
} catch (ClassNotFoundException ex) {
       // See "Handling errors" on page 42
}

try {
     // Open a connection to TimesTen
     con = DriverManager.getConnection(URL);

     // Report any SQLWarnings on the connection
     // See "Reporting errors and warnings" on page 44

     // Do SQL operations
     // See "Managing TimesTen data" on page 28

// Close the connection to TimesTen
con.close();

// Handle any errors
} catch (SQLException ex) {
     // See "Handling errors" on page 42
}
```

Managing TimesTen data

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

- Calling SQL statements within Java applications
- Fetching multiple rows of data
- Executing multiple SQL statements in a batch
- Working with result sets

Calling SQL statements within Java applications

This section includes the following topics:

- Setting autocommit
- Specifying multibyte characters in SQL functions
- Preparing SQL statements
- Executing SQL statements
- Setting a timeout value for executing SQL statements
- Putting it all together: preparing and executing SQL

Setting autocommit

A TimesTen Connection has autocommit enabled by default. You can use **Connection.setAutoCommit()** to enable or disable autocommit. If autocommit is disabled (set to false), you must use **Connection.commit()** to manually commit transactions.

For example, to set autocommit to off:

```
con.setAutoCommit(false);

    // Report any SQLWarnings on the connection
    // See "Reporting errors and warnings" on page 44
```

Specifying multibyte characters in SQL functions

When using SQL in JDBC, pay special care to Java escape syntax. SQL functions such as UNISTR use the backslash (\) character. You should escape the backslash character. For example, using the following SQL syntax in a Java application may not produce the intended results:

```
INSERT INTO table1 SELECT UNISTR('\00E4') FROM dual;
```

Escape the backslash character as follows:

```
INSERT INTO table1 SELECT UNISTR('\\00E4') FROM dual;
```

Preparing SQL statements

SQL statements that are to be executed more than once should be prepared in advance by calling the **Connection.prepareStatement()** method.

For maximum performance, prepare parameterized statements. In TimesTen, SQL statements containing duplicate parameters are parsed such that only distinct parameter names are considered as separate parameters. Binding is based on the position of the first occurrence of a parameter name. Each duplicate parameter occurrence is bound to the same value.

Example 2.3 shows how four separate INSERT statements can be substituted with a single parameterized statement.

Example 2.3 Rather than execute a similar INSERT statement with different values:

```
Statement.execute("insert into t1 values (1, 2)");
Statement.execute("insert into t1 values (3, 4)");
Statement.execute("insert into t1 values (5, 6)");
Statement.execute("insert into t1 values (7, 8)");
```

It is much more efficient to prepare a single parameterized INSERT statement and use **PreparedStatement.set...()** methods to set the row values before each execute:

```
PreparedStatement pIns =
       con.PreparedStatement("insert into t1 values (?,?)");
con.commit();

pIns.setInt(1, Integer.parseInt(1));
pIns.setInt(2, Integer.parseInt(2));
pIns.executeUpdate();

pIns.setInt(1, Integer.parseInt(3));
pIns.setInt(2, Integer.parseInt(4));
pIns.executeUpdate();

pIns.setInt(1, Integer.parseInt(5));
pIns.setInt(2, Integer.parseInt(6));
pIns.executeUpdate();

pIns.setInt(1, Integer.parseInt(7));
pIns.setInt(2, Integer.parseInt(8));
pIns.executeUpdate();

con.commit();
pIns.close();
```

Note: After preparing SQL statements, call **Connection.commit()** in order to release the locks held by the prepare and to allow the query plan to persist. When you have finished executing a prepared statement, call the **PreparedStatement.close()** method to release the resources associated with the statement.

TimesTen shares prepared statements automatically once they have been committed. For example, if two or more separate connections to the data store each prepare the same statement, then the second, third, and *n*th prepare returns very quickly because TimesTen remembers the first prepared statement.

Example 2.4 In this example, we prepare three identical parameterized INSERT statements for three separate connections. The first prepared INSERT for connection *con1* is shared with the *con2* and *con3* connections and speeds up the *pIns2* and *pIns3* prepare operations:

```
Connection con1;
Connection con2;
Connection con3;
.....
PreparedStatement pIns1 = con1.prepareStatement
                  ("insert into t1 values (?,?)");
con1.commit();

PreparedStatement pIns2 = con2.prepareStatement
                  ("insert into t1 values (?,?)");
con2.commit();

PreparedStatement pIns3 = con3.prepareStatement
                  ("insert into t1 values (?,?)");
con3.commit();
```

Note: All tuning options, such as join ordering, indexes and locks must match for the statement to be shared. Also, if the prepared statement references a temp table, it will only be shared within a single connection.

Note: TimesTen also supports prepared statement pooling for **PooledConnection**s, as specified in the JDBC 3.0 specification. You can configure the maximum size of the pool by setting **ObservableConnectionDS.setMaxStatements()**. Once set, this value should not be changed.

See "Prepare statements in advance" on page 69 for a general discussion of the performance benefits of preparing SQL statements in advance.

Executing SQL statements

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 how to use the **Connection.createStatement()**, **Statement.executeUpdate()**, and **Statement.executeQuery()** methods to execute a SQL statement within a Java application.

Unless statements are prepared in advance, as described in "Preparing SQL statements", use the **Statement** execute methods, such as **Statement.execute(), Statement.executeUpdate()**, or **Statement.executeQuery()**, depending on the nature of your SQL statement and any returned result set.

For SQL statements that are a prepared in advance, use the **PreparedStatement** execute methods, such as **PreparedStatement.execute()**, **PreparedStatement.executeUpdate()**, or **PreparedStatement.executeQuery()**.

The **execute()** method returns True if there is a result set (for example, on. a SELECT) or False if there is no result set (for example, on an INSERT, UPDATE, or DELETE). The **executeUpdate()** method returns the number of rows affected. For example, when executing an INSERT statement, the **executeUpdate()** method returns the number of rows inserted. The **executeQuery()** method returns a result set, so it should only be called when a result set is expected (for example, when executing a SELECT).

Note: See "Working with result sets" on page 38 for details about what you need to know when working with result sets generated by TimesTen.

Example 2.5 to use the **Statement.executeUpdate()** method to execute an INSERT into the *xyz.customer* table, enter:

```
Connection con;
Statement stmt;
. . . . . .
try {
    stmt = con.createStatement();
    int numRows = stmt.executeUpdate("insert into xyz.customer
       values" + "(40, 'West', 'Big Dish', '123 Signal St.');");
}
catch (SQLException ex) {
    .....
}
```

Example 2.6 In this example, we use a **Statement.executeQuery()** method to execute a SELECT on the *xyz.customer* table and display the returned **ResultSet**:

```
Statement stmt;
. . . . . .
try {
  ResultSet rs = stmt.executeQuery("select cust_num, region, " +
                       "name, address from xyz.customer;");

  System.out.println("Fetching result set...");
  while (rs.next()) {
     System.out.println("\n  Customer number: " + rs.getInt(1));
     System.out.println("  Region: " + rs.getString(2));
     System.out.println("  Name: " + rs.getString(3));
     System.out.println("  Address: " + rs.getString(4));
     }
  }
catch (SQLException ex) {
    ex.printStackTrace();
}
```

Example 2.7 In this example, we use a **PreparedStatement.executeQuery()** method to execute a prepared SELECT statement and display the returned **ResultSet**:

```
PreparedStatement pSel = con.prepareStatement("select cust_num, " +
                          "region, name, address " +
                          "from xyz.customer;");

con.commit();

try {
  ResultSet rs = pSel.executeQuery();

  while (rs.next()) {
      System.out.println("\n  Customer number: " + rs.getInt(1));
      System.out.println("  Region: " + rs.getString(2));
      System.out.println("  Name: " + rs.getString(3));
      System.out.println("  Address: " + rs.getString(4));
    }
}
catch (SQLException ex) {
    ex.printStackTrace();
}
```

Setting a timeout value for executing SQL statements

In TimesTen you can set the DSN attribute **SqlQueryTimeout** to specify the query timeout period for all connections. If you set **SqlQueryTimeout** in the DSN specification, its value becomes the default value for all subsequent connections to the data store.

You can override the **SqlQueryTimeout** value for the current connection by calling the **Statement.setQueryTimeout()** method to set 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 **Statement.setQueryTimeout()** method 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 **LockWait** and **SqlQueryTimeout** settings in TimesTen 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 SQL query timeout, and indicates the appropriate **SqlQueryTimeout** value to any executing thread. If both a **LockWait** timeout value and a **SqlQueryTimeout** value are specified, the lesser of the two values causes a timeout first.

Putting it all together: preparing and executing SQL

In this example, we prepare INSERT and SELECT statements; execute the INSERT twice; execute the SELECT, and print the returned result set. For a working example, see the `level1.java` demo.

```
Connection con;
Statement stmt;

// Disable auto-commit
con.setAutoCommit(false);

     // Report any SQLWarnings on the connection
     // See "Reporting errors and warnings" on page 44

// Prepare a parameterized INSERT and a SELECT Statement
PreparedStatement pIns = con.prepareStatement("insert into
xyz.customer values (?,?,?,?)");

PreparedStatement pSel = con.prepareStatement
   ("select cust_num, region, name, " +
    "address from xyz.customer");

// Prepare is a transaction; must commit to release locks
con.commit();

// Data for first INSERT statement
pIns.setInt(1, Integer.parseInt(100));
pIns.setString(2, 'N');
pIns.setString(3, 'Fiberifics');
pIns.setString(4,'123 any street');

// Execute the INSERT statement
pIns.executeUpdate();

// Data for second INSERT statement
pIns.setInt(1, Integer.parseInt(101));
pIns.setString(2, 'N');
pIns.setString(3,'Natural Foods Co.');
pIns.setString(4,'5150 Johnson Rd');

// Execute the INSERT statement
pIns.executeUpdate();

// Commit the inserts
con.commit();

// Done with INSERTs, so close the prepared statement
pIns.close();
```

```
// Report any SQLWarnings on the connection
reportSQLWarnings(con.getWarnings());
CheckIfStopIsRequested();

// Execute the prepared SELECT statement
ResultSet rs = pSel.executeQuery();

System.out.println("Fetching result set...");
while (rs.next()) {
    System.out.println("\n Customer number: " + rs.getInt(1));
    System.out.println("  Region: " + rs.getString(2));
    System.out.println("  Name: " + rs.getString(3));
    System.out.println("  Address: " + rs.getString(4));
}

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

// Commit the select - yes selects need to be committed too
con.commit();

// Close the select statement - we're done with it
pSel.close();
```

Fetching multiple rows of data

Fetching multiple rows of data from a TimesTen data store can increase the performance of an application that connects to a data store set with read committed isolation.

You can specify the number of rows to be prefetched by:

- Calling the **Statement.setFetchSize()** and **ResultSet.setFetchSize** methods. These are the standard JDBC calls, but the limitation is that they only affect one statement at a time.
- Calling the **TimesTenConnection.setTtPrefetchCount()** method or by using the **ttIsql** `prefetchcount` command. These enable a TimesTen extension that establishes prefetch on a connection level so that all of the statements on the connection use the same prefetch setting.

This section describes the connection-level prefetch implemented in TimesTen.

Note: You can use the TimesTen prefetch count extension only with direct-linked applications.

When the prefetch count is set to 0, TimesTen uses a default value, depending on the **Isolation** level you have set for the data store. In read committed isolation mode, the default prefetch value is 5. In serializable isolation mode, the default is 128. The default prefetch value is the optimum setting for most applications. Generally, a higher value may result in better performance for larger result sets, at the expense of slightly higher resource use.

To disable prefetch, set the prefetch count to 1.

Call **TimesTenConnection.getTtPrefetchCount()** to check the current prefetch value.

Example 2.8

In this example, we use the **ttIsql** `prefetchcount` command to set the prefetch count for the connection to 6:

```
> ttIsql RunData_tt51
Command > prefetchcount 6;
```

Example 2.9

In this example, we use **setTtPrefetchCount()** to set the prefetch count to 10 and then use **getTtPrefetchCount()** to return the prefetch count in the count variable.

```
TimesTenConnection con =
     (TimesTenConnection) DriverManager.getConnection(url);

// set prefech count to 10 for this connection
con.setTtPrefetchCount(10);

// Return the prefetch count to the 'count' variable.
int count = con.getTtPrefetchCount();
```

Executing multiple SQL statements in a batch

You can improve performance by calling the **addBatch()** and **executeBatch()** methods for the **Statement** and **PreparedStatement** objects.

For **Statement** objects, a batch typically consists of a set of INSERT or UPDATE statements. Statements that return result sets are not allowed in a batch. A SQL statement is added to a batch by calling the **addBatch()** method. The set of SQL statements associated with a batch are executed through the **executeBatch()** method. For example:

```
// turn off autocommit
conn.setAutoCommit(false);

Statement stmt = conn.createStatement();
stmt.addBatch("INSERT INTO employees VALUES (1000, 'Joe
Jones')");
stmt.addBatch("INSERT INTO departments VALUES (260, 'Shoe')");
stmt.addBatch("INSERT INTO emp_dept VALUES (1000, 260)");

// submit a batch of update commands for execution
int[] updateCounts = stmt.executeBatch();
conn.commit ();
```

For **PreparedStatement** objects, a batch consists of a set of prepared statement input parameters. Prepared statement parameters are added to the batch by executing **set** calls followed by the **addBatch()** call. The batch is executed via the **executeBatch()** method. For example:

```
// turn off autocommit
conn.setAutoCommit(false);

PreparedStatement stmt = conn.prepareStatement(
"INSERT INTO employees VALUES (?, ?)");

// first set of parameters
stmt.setInt(1, 2000);
stmt.setString(2, "Kelly Kaufmann");
stmt.addBatch();

// second set of parameters
stmt.setInt(1, 3000);
stmt.setString(2, "Bill Barnes");
stmt.addBatch();

// submit the batch for execution. Check update counts
int[] updateCounts = stmt.executeBatch();
conn.commit ();
```

Working with result sets

In addition to queries, some methods and built-in procedures return TimesTen data in the form of a **ResultSet** object. This section describes what you need to know when using **ResultSet** objects from TimesTen.

- TimesTen does not support multiple open **ResultSet** objects per statement. TimesTen cannot return multiple **ResultSet** objects from a single **Statement** object without first closing the current result set.
- TimesTen does not support holdable cursors. You cannot specify the holdability of a result set, i.e. whether a cursor can remain open after it has been committed.
- **ResultSet** objects are not scrollable or updatable, so you cannot specify **ResultSet.TYPE_SCROLL_SENSITIVE** or **ResultSet.CONCUR_UPDATABLE**.
- Use the **ResultSet.close** method to close **ResultSet** objects as soon as you are done with them.
- Calling **ResultSet.getString** is more costly with regard to performance if the underlying data type is not a string. Because Java strings are immutable, **ResultSet.getString** must allocate space for a new string each time it is called. This makes **ResultSet.getString** one of the costlier calls in JDBC. Do not use **ResultSet.getString** to retrieve primitive numeric types, like Byte or Integer, unless it is absolutely necessary. For example, it is much faster to call **ResultSet.getInt** on an integer column.
- JDBC ignores the setting for the **ConnectionCharacterSet** attribute. It returns data in UTF-16 encoding.

Calling TimesTen built-in procedures

Chapter 2, "Built-In Procedures" in the *Oracle TimesTen In-Memory Database API Reference Guide* describes the TimesTen built-in procedures that extend standard ODBC functionality. You can execute a TimesTen built-in procedure using the **CallableStatement** interface.

To execute the built-in procedure, use the format:

```
CallableStatement.execute("{ Call Procedure }")
```

To prepare and execute a built-in procedure, use the format:

```
CallableStatement cStmt;
cStmt = con.prepareCall("{ Call Procedure }");
cStmt.execute();
```

For built-in procedures that return results, you can use the **ResultSet get*()** methods to retrieve the data from the returned **ResultSet**, as demonstrated in Example 2.11.

Note: See "Working with result sets" on page 38 for details about what you need to know when working with result sets generated by TimesTen.

Example 2.10 To call the **ttCkpt** procedure to initiate a fuzzy checkpoint, enter:

```
Connection con;
CallableStatement cStmt;
.......
cStmt = con.prepareCall("{ Call ttCkpt }");
cStmt.execute();
con.commit();          // commit the transaction
```

Example 2.11 This example calls the **ttDataStoreStatus** procedure and prints out the returned result set.

Contrary to the advice given in "Working with result sets" on page 38, we use **ResultSet.getString** in this example to retrieve the *Context* field, which is a binary. This is because the data is output is printed, rather than used for processing. If you were not to print the *Context* value, you could achieve better performance using the **ResultSet.getBytes** method.

```
ResultSet rs;

cStmt = con.prepareCall("{ Call ttDataStoreStatus }");

if (cStmt.execute() == true) {
     rs = cStmt.getResultSet();
     System.out.println("Fetching result set...");
```

```
     while (rs.next()) {
        System.out.println("\n Data store: " + rs.getString(1));
        System.out.println("  PID: " + rs.getInt(2));
        System.out.println("  Context: " + rs.getString(3));
        System.out.println("  ConType: " + rs.getString(4));
        System.out.println("  memoryID: " + rs.getString(5));
        }
    rs.close();
  }
cStmt.close();
```

Note: You cannot pass parameters to **CallableStatement** by name. You must set parameters by ordinal numbers. You cannot use the SQL escape syntax.

Managing multiple threads

Note: On some UNIX platforms, it is necessary to set the THREADS_FLAG variable, as described in "Set the THREADS_FLAG variable (UNIX only)" on page 9.

The `level4.java` demo demonstrates the use of multiple threads.

When your application has a direct driver connection to the data store, TimesTen functions share stack space with your application. In multithreaded environments, it is important to avoid overrunning the stack allocated to each thread because consequences can result that are unpredictable and difficult to debug. The amount of stack space consumed by TimesTen calls varies depending on the SQL functionality used. Most applications should set thread stack space to at least 16 KB on 32-bit systems and between 34 KB to 72 KB on 64-bit systems.

The amount of stack space allocated for each thread is specified by the operating system when threads are created. On Windows, you can use the TimesTen debug driver and link your application against the Visual C++ debug C library to enable "stack probes" that raise an identifiable exception if a thread attempts to grow its stack beyond the amount allocated.

Note: In multithreaded applications, a thread that issues requests on different connection handles to the same data store may encounter lock conflict with itself. TimesTen resolves these conflicts with lock timeouts.

Handling errors

This section discusses how to check for, identify and handle errors in you TimesTen Java application.

For a list of the errors that TimesTen returns and what to do if the error is encountered, see Chapter 1, "Warnings and Errors" in the *Oracle TimesTen In-Memory Database Error Messages and SNMP Traps*.

The topics are:

- About fatal errors, non-fatal errors, and warnings
- Reporting errors and warnings
- Detecting and responding to specific errors
- Rolling back failed transactions

About fatal errors, non-fatal errors, and warnings

TimesTen can return a fatal error, a non-fatal error, or a warning.

Handling fatal errors and recovery

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 rollback the transaction and disconnect from the data store.

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.

Handling 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 in the case of a process crash, an error cannot be returned, so 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, as described in "Rolling back failed transactions" on page 47.

Note: If a **ResultSet**, **Statement**, **PreparedStatement**, **CallableStatement** or **Connection** operation results in a data store error, it is a good practice to call the **close** method for that object.

About 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

You should always include code that checks for warnings, as they can indicate application problems.

Reporting errors and warnings

You should check for and report all errors and warnings that can be returned on every call. This saves considerable time and effort during development and debugging. A **SQLException** object is generated in case of one or more data store access errors and a **SQLWarning** object is generated in the case of one or more warning messages. A single call may return multiple errors and/or warnings, so your application should report all errors or warnings in the returned **SQLException** or **SQLWarning** objects.

Multiple errors or warnings are returned in linked chains of **SQLException** or **SQLWarning** objects. Example 2.12 and Example 2.13 demonstrate how you might iterate through the lists of returned **SQLException** and **SQLWarning** objects to report all of the errors and warnings, respectively.

Example 2.12 This method prints out the content of all exceptions in the linked **SQLException** objects.

```
static int reportSQLExceptions(SQLException ex)
  {
    int errCount = 0;

    if (ex != null) {
      errStream.println("\n--- SQLException caught ---");
      ex.printStackTrace();

      while (ex != null) {
        errStream.println("SQL State:   " + ex.getSQLState());
        errStream.println("Message:     " + ex.getMessage());
        errStream.println("Error Code: " + ex.getErrorCode());
        errCount ++;
        ex = ex.getNextException();
        errStream.println();
      }
    }

    return errCount;
  }
```

Example 2.13 This method prints out the content of all warning in the linked **SQLWarning** objects.

```
static int reportSQLWarnings(SQLWarning wn)
{
   int warnCount = 0;

   while (wn != null) {
      errStream.println("\n--- SQL Warning ---");
```

```
      errStream.println("SQL State:   " + wn.getSQLState());
      errStream.println("Message:     " + wn.getMessage());
      errStream.println("Error Code: " + wn.getErrorCode());

      // is this a SQLWarning object or a DataTruncation object?
      if (wn instanceof DataTruncation) {
         DataTruncation trn = (DataTruncation) wn;
         errStream.println("Truncation error in column: " +
            trn.getIndex());
      }

      warnCount++;
      wn = wn.getNextWarning();
      errStream.println();
   }

   return warnCount;
}
```

Detecting and responding to specific errors

In some situations it may be desirable to respond to a specific SQL state or TimesTen error code. You can use **SQLException.getSQLState** to return the SQL99 SQL state error string, and **SQLException.getErrorCode** to return TimesTen error codes, as shown in Example 2.14.

Example 2.14 The TimesTen demos require that you load the demo schema before they are executed. The following **catch** statement alerts the user that the demo schema has not been loaded or has not been refreshed by detecting ODBC error S0002 and TimesTen error 907:

```
catch (SQLException ex) {
  if (ex.getSQLState().equalsIgnoreCase("S0002")) {
     errStream.println("\nError: The table xyz.customer
         does not exist.\n\t Please run ttIsql -f input0.dat
         to initialize the database.");
  } else if (ex.getErrorCode() == 907) {
     errStream.println("\nError: Attempting to insert a row
         with a duplicate primary key.\n\tPlease rerun ttIsql -f
         input0.dat to reinitialize the database.");
  }
```

You can use the **TimesTenVendorCode** interface to detect the errors by their name, rather than their number.

For example:

```
ex.getErrorCode() ==
com.timesten.jdbc.TimesTenVendorCode.TT_ERR_KEYEXISTS
```

is the same as:

```
ex.getErrorCode() == 907
```

See "TimesTenVendorCode" on page 82 for the complete list of error name-to-number mappings.

Rolling back failed transactions

In some situations, such as recovering from a deadlock or time-out condition, you may want to explicitly roll back the transaction using the **Connection.rollback()** method.

For example:

```
try {
  if (con != null && !con.isClosed()) {
   // Rollback any transactions in case of errors
     if (retcode != 0) {
       try {
         System.out.println("\nEncountered error.
                             Rolling back transaction");
         con.rollback();
       } catch (SQLException ex) {
           reportSQLExceptions(ex);
         }
      }

  System.out.println("\nClosing the connection\n");
  con.close();
}
```

The XACT_ROLLBACKS column of the SYS.MONITOR table indicates the number of transactions that were rolled back.

A transaction rollback consumes resources and the entire transaction is in effect wasted. To avoid unnecessary rollbacks, design your application to avoid contention (see "Choose the best method of locking" on page 67) and check application or input data for potential errors before submitting it, whenever possible.

Note: Should your application crash in the middle of an active transaction, TimesTen automatically rolls back the transaction.

3

Using JMS/XLA for Event Management

You can use the TimesTen JMS/XLA API (JMS/XLA) to 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 JMS/XLA is to provide a high-performance, asynchronous alternative to triggers.

You can also use JMS/XLA to build a custom data replication solution, if the TimesTen replication solutions described in the *TimesTen to TimesTen Replication Guide* do not meet your needs.

JMS/XLA implements Sun Microsystems' Java Message Service (JMS) interfaces to make the functionality of the TimesTen Transaction Log API (XLA) available to Java applications. For detailed information about the JMS API, refer to the documentation published by Sun Microsystems at http://java.sun.com/products/jms/docs.html. The Sun JMS documentation is installed with the Oracle TimesTen In-Memory Database.

For an introduction to TimesTen event management, see "Event Notification" in the *Oracle TimesTen In-Memory Database Introduction*.

For information about tuning TimesTen JMS/XLA applications for improved performance, see "Tuning JMS/XLA applications" on page 74.

This chapter includes the following topics:

JMS/XLA concepts

Java applications can use the JMS/XLA API to receive event notifications from TimesTen. JMS/XLA uses the JMS publish-subscribe interface to provide access to XLA updates.

You subscribe to updates by establishing a JMS **Session** that provides a connection to XLA and creating a durable subscriber (**TopicSubscriber**). You can receive and process messages synchronously through the subscriber, or you can implement a listener (**MessageListener**) to process the updates asynchronously.

JMS/XLA is designed for applications that want to monitor a local data store. TimesTen and the application receiving the notifications must reside on the same machine.

Note: The JMS/XLA API supports persistent-mode XLA. In this mode, XLA obtains update records directly from the transaction log buffer or log files, so the records are available until they are read. Persistent-mode XLA also allows multiple readers to access transaction log updates simultaneously.

This section includes the following topics:

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 disk-based logging is enabled for the data store, log records are periodically flushed in batches from the log buffer in memory to log files 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.

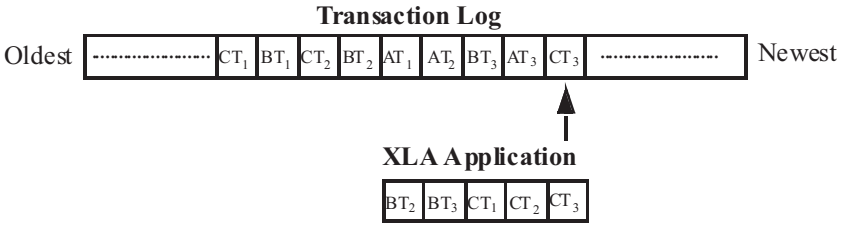
Only the records for committed transactions are returned. They are returned in the order in which their final commit record appears in the transaction log. XLA filters out records associated with changes to the data store that have not yet committed.

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's* 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).

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.

XLA configuration file and topics

To connect to XLA, you establish a connection to a JMS *`Topic`* that corresponds to a particular data store. The JMS/XLA configuration file provides the mapping between topic names and data stores.

By default, JMS/XLA looks for a configuration file called `jmsxla.xml` in the current working directory. If you want to use another name or location for the file, you need to specify it as part of the environment variable in the **InitialContext** class and add the location to CLASSPATH. To specify it as part of the environment variable in the **InitialContext** class, use code similar to Example 3.2.

Example 3.2

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
   "com.timesten.dataserver.jmsxla.SimpleInitialContextFactory");
env.put(XlaConstants.CONFIG_FILE_NAME, "/newlocation.xml");
InitialContext ic = new InitialContext(env);
```

The JMS/XLA API usually uses the class loader to locate the JMS/XLA configuration file if `XlaConstants.CONFIG_FILE_NAME` is set. In Example 3.2, the JMS/XLA API searches for the `newlocation.xml` file in the top directory in

both the location specified in the CLASSPATH environment variable and in the jar files specified in the CLASSPATH environment variable.

The JMS/XLA configuration file can also be located in subdirectories, as shown in Example 3.3.

Example 3.3

```
env.put(XlaConstants.CONFIG_FILE_NAME,
     "/com/mycompany/myapplication/deepinside.xml");
```

In Example 3.3, the JMS/XLA API searches for the `deepinside.xml` file in the `com/mycompany/myapplication` subdirectory in both the location specified in the CLASSPATH environment variable and in the jar files specified in the CLASSPATH environment variable.

The JMS/XLA API uses the first configuration file that it finds.

A topic definition in the configuration file consists of a name, a JDBC connection string, and a prefetch value that specifies how many updates to retrieve at a time.

For example, the configuration file shown in Example 3.4 maps the `DemoDataStore` topic to the `TestDB` DSN.

Example 3.4

```
<xlaconfig>
  <topics>
    <topic name="DemoDataStore"
      connectionString="jdbc:timesten:direct:DSN=TestDB"
      xlaPrefetch="100"
  />
  </topics>
</xlaconfig>
```

XLA updates

Applications receive XLA updates as JMS **MapMessage** objects. The `MapMessage` contains a set of typed name/value pairs that correspond to the fields in an XLA update header.

You can access the message fields using the `MapMessage` *`get`* methods. The `getMapNames` method returns an `Enumeration` that contains the names of all of the fields in the message. You can retrieve individual fields from the message by name. All reserved field names begin with two underscores, for example `__TYPE`.

All update messages have a `__TYPE` field that indicates what type of update the message contains. The types are specified as integer values. As a convenience, you can use the constants defined in

`com.timesten.dataserver.jmsxla.XlaConstants` to compare against the integer types. The supported types are described in Table 3.1.

Table 3.1 XLA Update Types

Update Type	Description
`INSERT`	A row has been added.
`UPDATE`	A row has been modified.
`DELETE`	A row has been removed.
`COMMIT_ONLY`	A transaction has been committed.
`CREATE_TABLE`	A table has been created.
`DROP_TABLE`	A table has been dropped.
`CREATE_INDEX`	An index has been created.
`DROP_INDEX`	An index has been dropped.
`ADD_COLUMNS`	New columns have been added to the table.
`DROP_COLUMNS`	Columns have been removed from the table.
`CREATE_VIEW`	A materialized view has been created.
`DROP_VIEW`	A materialized view has been dropped.
`CREATE_SEQ`	A SEQUENCE has been created.
`DROP_SEQ`	A SEQUENCE has been dropped.
`TRUNCATE`	The table has been truncated and all rows in the table have been deleted.

For more information about the contents of an XLA update message see "XLA MapMessage contents" on page 83.

XLA bookmarks

An XLA bookmark marks the read position of an XLA subscriber application in the transaction log. Bookmarks facilitate durable subscriptions, enabling an application to disconnect from a topic and then reconnect to continue receiving updates where it left off.

When you create a message consumer for XLA, you always use a durable **TopicSubscriber**. The subscription identifier you specify when you create the

subscriber is used as the XLA bookmark name. When you use the built-in procedures **ttXlaSubscribe** and **ttXlaUnsubscribe** via JDBC to start and stop XLA publishing for a table, you explicitly specify the name of the bookmark to be used.

Bookmarks are reset to the last read position whenever an acknowledgement is received. For more information about how update messages are acknowledged, see "XLA acknowledgement modes" on page 55.

You can remove a durable subscription by calling `unsubscribe` on the JMS **`Session`**. This deletes the corresponding XLA bookmark and forces a new subscription to be created when you reconnect. For more information see "Deleting bookmarks" on page 61.

XLA acknowledgement modes

XLA's acknowledgement mechanism is designed to ensure that an application has not only received a message, but has successfully processed it. Acknowledging an update permanently resets the application's XLA bookmark to the last record that was read. This prevents previously returned records from being re-read, ensuring that an application does not receive previously acknowledged records if the bookmark is reused when an application reconnects to XLA.

JMS/XLA can automatically acknowledge XLA update messages, or applications can choose to acknowledge messages explicitly. You specify how updates are to be acknowledged when you create the `Session`.

JMS/XLA supports three acknowledgement modes:

- AUTO_ACKNOWLEDGE—In this mode, updates are automatically acknowledged as you receive them. Each message is delivered only once--duplicate messages will not be sent and in the event of an application failure, messages might be lost. In AUTO_ACKNOWLEDGE mode, messages are always delivered and acknowledged individually, so JMS/XLA does not prefetch multiple records. (The `xlaprefetch` attribute in the topic is ignored.)
- DUPS_OK_ACKNOWLEDGE—In this mode, updates are automatically acknowledged, but duplicate messages might be delivered in the event of an application failure. In DUPS_OK_ACKNOWLEDGE mode, JMS/XLA prefetches records according to the `xlaprefetch` attribute specified for the topic and sends an acknowledgement when the last record in a prefetched block is read. If the application fails before reading all of the prefetched records, all of the records in the block are presented to the application it restarts.
- CLIENT_ACKNOWLEDGE—in this mode, applications are responsible for acknowledging receipt of update messages by calling `acknowledge` on the `MapMessage`. In CLIENT_ACKNOWLEDGE mode, JMS/XLA prefetches records according to the `xlaprefetch` attribute specified for the topic.

Prefetching updates

Prefetching multiple update records at a time is more efficient than obtaining each update record from XLA individually. Because updates are not prefetched when you use AUTO_ACKNOWLEDGE mode, it can be slower than the other modes. If possible, you should design the application to tolerate duplicate updates so you can use DUPS_OK_ACKNOWLEDGE, or explicitly acknowledge updates. Explicitly acknowledging updates usually yields the best performance, as long as you can avoid acknowledging each message individually.

Acknowledging updates

To explicitly acknowledge an XLA update, you call `acknowledge` on the update message. Acknowledging a message implicitly acknowledges all previous messages. Typically, you receive and process multiple update messages between acknowledgements. If you are using the CLIENT_ACKNOWLEDGE mode and intend to reuse a durable subscription in the future, you should call `acknowledge` to reset the bookmark to the last-read position before exiting.

XLA demos

The TimesTen JMS/XLA demos demonstrate how to use the JMS/XLA API to subscribe to updates to a TimesTen table. The demo applications are located in the *`install_dir`*`/demo/tutorial/java` directory.

The demos use the TimesTen demo schema, which simulates a simple order-processing database. For information about compiling the TimesTen Java demos and building the demo schema, see "About the TimesTen Java demos" on page 12.

XlaLevel1 demo

The procedures demonstrated in this section are based on the `XlaLevel1.java` demo application. This application reports on updates to the `xyz.customer` table.

JMS/XLA and Oracle GDK dependency

The JMS/XLA API uses `orai18n.jar`, part of the Oracle Globalization Development Kit (GDK) for translating from the database character set specified by the **DatabaseCharacterSet** attribute to UTF-16 encoding. JMS/XLA API supports a specific version of GDK with each TimesTen release. If JMS/XLA API finds other versions of GDK already loaded in the JVM, it displays a severe warning and continues processing. You can find out the version of GDK supported by JMS/XLA API by entering the following commands:

```
cd install_dir/lib
$ java -cp ./orai18n.jar oracle.i18n.util.GDKOracleMetaData
-version
```

Connecting to XLA

To connect to XLA so you can receive updates, you use the JMS **ConnectionFactory** to create a `Connection`. You then use the `Connection` to establish a **Session**. When you are ready to start processing updates, you call `start` on the **Connection** to enable message dispatching.

For example, in the XlaLevel1 demo, the JMS `Session` is set up by the `XLALevel1 subscribe` method, as shown in Example 3.5. Note that the acknowledgement mode is set when the session is created.

Example 3.5

```
ConnectionFactory connectionFactory;

Context messaging = new InitialContext();
connectionFactory = (ConnectionFactory)
      messaging.lookup("ConnectionFactory");
Connection connection = connectionFactory.createConnection();

Session session = connection.createSession(false,
      Session.AUTO_ACKNOWLEDGE);

connection.start();
```

Monitoring tables for updates

Before you can start receiving updates, you need to tell XLA which tables you want to monitor for changes.

To subscribe to changes and turn on XLA publishing for a table, you call the **ttXlaSubscribe** built-in procedure via JDBC.

When you use **ttXlaSubscribe** to enable XLA publishing for a table, you need to specify two parameters, the name of the table and the name of the bookmark that will be used to track the table:

```
ttXlaSubscribe(user.table, bookmark)
```

For example, to call **ttXlaSubscribe** via the JDBC `CallableStatement` interface:

```
Connection con;
CallableStatement cStmt;
...
cStmt = con.prepareCall("{call ttXlaSubscribe(user.table,
bookmark)}");
cStmt.execute();
```

Use **ttXlaUnsubscribe** to unsubscribe from the table during shutdown. For more information, see "Unsubscribing from a table" on page 61.

For more information about using TimesTen built-in procedures in a Java application, see "Calling TimesTen built-in procedures" in Chapter 2. See *Oracle TimesTen In-Memory Database API Reference Guide* for more information about built-in procedures.

Receiving and processing updates

You can receive XLA updates either synchronously or asynchronously.

To receive and process update for a topic synchronously, perform the following tasks:

1. Create a durable **TopicSubscriber** to subscribe to a topic.
2. Call `receive` or `receiveNoWait` on your subscriber to get the next available update.
3. Process the returned **MapMessage**.

To receive and process updates for a topic asynchronously, perform the following tasks:

1. Create a **MessageListener** to process the updates.
2. Create a durable `TopicSubscriber` to subscribe to a topic.
3. Register the `MessageListener` with the `TopicSubscriber`.
4. Start the **Connection**.

Note: You must register the `MessageListener` before you start the `Connection`. Otherwise, you can miss messages. If the `Connection` is already started, stop the `Connection`, register `MessageListener` and then start the `Connection`.

---

5. Wait for messages to arrive. You can call `Object.wait` to wait for messages if your application does not need to do anything else in its main thread.

   When an update is published, the `MessageListener onMessage` method is called and the message is passed in as a `MapMessage`.

   The application can verify table subscriptions by checking the SYS.XLASUBSCRIPTIONS system table.

The XlaLevel1 demo uses a listener to process updates asynchronously, as shown in Example 3.6.

Example 3.6

```
MyListener myListener = new MyListener(outStream);

outStream.println("Creating consumer for topic " + topic);
Topic xlaTopic = session.createTopic(topic);
TopicSubscriber subscriber =
      session.createDurableSubscriber(xlaTopic, "mybookmark");

subscriber.setMessageListener(myListener);
```

Note that `mybookmark` must already exist. You can use JDBC and the **ttXlaBookmarkCreate** built-in procedure to create a bookmark. Also, the `TopicSubscriber` must be a durable subscriber. XLA connections are designed to be durable. XLA bookmarks make it possible to disconnect from a topic and then reconnect to start receiving updates where you left off. The `String` you pass in as the subscription identifier when you create a durable subscriber is used as the XLA bookmark name.

You can call `unsubscribe` on the JMS **TopicSession** to delete the XLA bookmark used by the subscriber when the application shuts down. This causes a new bookmark to be created when the application is restarted.

Processing updates

When you receive an update, you can use the **MapMessage** *`get`* methods to extract information from the message and then perform whatever processing your application requires. The TimesTen `XlaConstants.java` class defines constants for the update types and special message fields to make it easier to process XLA update messages.

The first step is typically to determine what type of update the message contains. You can use the `MapMessage.getInt` method to get the contents of the `__TYPE` field, and compare the value against the numeric constants defined in `the XlaConstants` class.

In the `XlaLevel1` demo, MySubscriber's `onMessage` method extracts the update type from the `MapMessage` and displays the action that the update signifies. This is shown in Example 3.7.

Example 3.7

```
public void onMessage(Message message) {
      MapMessage mapMessage = (MapMessage) message;
      String messageType = null;
      if (message == null) {
        errStream.println("MyListener: update message is null");
        return;
```

```
}

try {
  // Get the update type(insert, update, delete, etc.).
  int type = mapMessage.getInt(XlaConstants.TYPE_FIELD);
  if (type == XlaConstants.INSERT) {
    System.out.println("A row was inserted.");
  } else if (type == XlaConstants.UPDATE) {
    System.out.println("A row was updated.");
  } else if (type == XlaConstants.DELETE) {
    System.out.println("A row was deleted.");
  } else {
    return;
  }
```

When you know what type of message you have received, you can process the message according to the application's needs. To get a list of all of the fields in a message, you can call `MapMessage.getMapNames`. You can retrieve individual fields from the message by name.

For example, the XlaLevel1 demo extracts the column values from insert, update, and delete messages using the column names, as shown in Example 3.8.

Example 3.8

```
if (type == XlaConstants.INSERT
      || type == XlaConstants.UPDATE
      || type == XlaConstants.DELETE) {
         // Get the column values from the message.
         int cust_num = mapMessage.getInt("cust_num");
         String region = mapMessage.getString("region");
         String name = mapMessage.getString("name");
         String address = mapMessage.getString("address");

         System.out.println("New Column Values:");
         System.out.println("cust_num=" + cust_num);
         System.out.println("region=" + region);
         System.out.println("name=" + name);
         System.out.println("address=" + address);
       }
```

For detailed information about the contents of XLA update messages, see "XLA MapMessage contents" on page 83. For information about how TimesTen column types map to JMS data types and the *get* methods used to retrieve the column values, see "Data type mapping" on page 95.

Terminating an XLA application

When the XLA application has finished reading from the transaction log, it should gracefully exit by closing the XLA connection, deleting any unneeded bookmarks, and unsubscribing from any tables to which you explicitly subscribed.

Closing the connection

To close the connection to XLA, you call `close` on the **Connection** object.

After a connection has been closed, any attempt to use it, its sessions, or its subscribers will throw an `IllegalStateException`. You can continue to use messages received through the connection, but you cannot call a received message's `acknowledge` method after the connection is closed.

Deleting bookmarks

Deleting XLA bookmarks during shutdown is optional. Deleting a bookmark enables the disk space associated with any unread update records in the transaction log to be freed.

If you *do not* delete the bookmark, it can be reused by a durable subscriber. As long as the bookmark is available when a durable subscriber reconnects, the subscriber will receive all unacknowledged updates published since the previous connection was terminated. Keep in mind that as long as a bookmark exists with no application reading from it, the transaction log will continue to grow and the amount of disk space consumed by your database will increase.

To delete a bookmark, you can simply call `unsubscribe` on the JMS Session, which invokes the **ttXlaBookmarkDelete** built-in procedure to remove the XLA bookmark.

Unsubscribing from a table

To turn off XLA publishing for a table, you use the **ttXlaUnsubscribe** built-in procedure. If you use **ttXlaSubscribe** to enable XLA publishing for a table, you should use **ttXlaUnsubscribe** to unsubscribe from the table when shutting down your application.

When you unsubscribe from a table, you specify the name of the table and the name of the bookmark used to track the table:

```
ttXlaUnsubscribe(user.table, bookmark)
```

For example, to call **ttXlaSubscribe** via the JDBC `CallableStatement` interface:

```
Connection con;
CallableStatement cStmt;
...
cStmt = con.prepareCall("{call ttXlaUnSubscribe(user.table,
bookmark)}");
cStmt.execute();
```

For more information about using TimesTen built-in procedures in a Java application, see "Calling TimesTen built-in procedures" in Chapter 2. See *Oracle TimesTen In-Memory Database API Reference Guide* for more information about the built-in procedures themselves.

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 JMS/XLA to replicate updates from a source data store to a target data store. The source data store generates JMS/XLA messages. To apply the messages to a target data store, you must extract the XLA descriptor from them. Use the MapMessage interface to extract the update descriptor:

```
MapMessage message;
/**
*...other code
*/
try {
    byte[]updateMessage=
       mapMessage.getBytes(XlaConstants.UPDATE_DESCRIPTOR_FIELD);
}
catch (JMSException jex){
/**
*...other code
*/
}
```

The target data store may reside on a different machine from the source data store. The update descriptor is returned as a byte array and can be serialized for network transmission.

You must create a target data store object that represents the target data store so you can apply the objects from the source data store. You can create a target data store object called `myTargetDataStore` by using the **TargetDataStore** interface:

```
TargetDataStore myTargetDataStore=
new TargetDataStoreImpl("DSN=sampleDSN");
```

Apply messages to `myTargetDataStore` by using the **TargetDataStore**::apply method:

```
myTargetDataStore.apply(updateDescriptor);
```

By default, TimesTen checks for conflicts on the target data store before applying the update. If the target data store has information that is later than the update, **TargetDataStore** throws an exception. If you do not want TimesTen to check for conflicts, use the **TargetDataStore**::setUpdateConflictCheckFlag method to change the behavior.

By default, TimesTen commits the update to the data store based on commit flags and transaction boundaries contained in the update descriptor. If you want the application to perform manual commits instead, use the **setAutoCommitFlag** method to change the autocommit flag. To perform a manual commit on `myTargetDataStore`, use the following command:

```
myTargetDataStore.commit();
```

You can perform a rollback if errors occur during the application of the update. Use the following command for `myTargetDataStore`:

```
myTargetDataStore.rollback();
```

Close `myTargetDataStore` by using the following command:

```
myTargetDataStore.close;
```

See "JMS/XLA replication API" on page 99 for more information about **TargetDataStore**.

TargetDataStore error recovery

Invoking **TargetDataStore** can yield transient and permanent errors.

TargetDataStore methods return a nonzero value when transient errors occur. The application can retry the operation and is responsible for monitoring update descriptors that need to be reapplied. For more information about transient XLA errors, see "Handling XLA errors" in *Oracle TimesTen In-Memory Database C Developer's and Reference Guide*.

TargetDataStore methods return a JMSException for permanent errors. If the application receives a permanent error, it should verify that the data store is valid. If the data store is invalid, the target data store object should be closed and a new one should be created. Other types of permanent errors may require manual intervention.

The following sample code shows how to recover errors from **TargetDataStore**:

```
TargetDataStore theTargetDataStore;
byte[] updateDescriptor;
int rc;

// Other code
try {
 …
```

```
 if ( (rc = theTargetDataStore.apply(updateDescriptor) ) == 0 ) {
     // apply successful
 }
 else {
     // Transient error.  Retry later.
  }
 }
catch (JMSException jex) {
 if (theTargetDataStore.isDataStoreValid()  ) {
    // Data store is valid; permanent error that may need
Administrator intervention.
 }
 else {
  try {
     theTargetDataStore.close();
 }
 catch (JMSException closeEx) {
   // Close errors are not usual. This may need Administrator
intervention.
 }
}
```

4

Application Tuning

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

This chapter includes general principles to consider when tuning Java applications for the Oracle TimesTen In-Memory Database and specific performance tuning tips for applications that utilize the JMS/XLA API.

Tuning Java applications

This section describes general principles to consider when tuning Java applications for TimesTen. It includes the following topics:

- Turn off autocommit mode
- Choose a timeout interval
- Choose the best method of locking
- Reduce contention
- Choose the appropriate logging options
- Prepare statements in advance
- Avoid unnecessary prepare operations
- Use the batch update facility for executing multiple statements
- Bulk fetch rows of TimesTen data
- Size transactions appropriately
- Use durable commits appropriately
- Use the ResultSet.getString method sparingly
- Avoid data type conversions
- Avoid transaction rollback
- Avoid frequent checkpoints

Turn off autocommit mode

By default, autocommit is enabled, which forces a commit after each statement. 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, as described in "Setting autocommit" on page 28.

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

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

Choose a timeout interval

By default, connections wait 10 seconds to acquire a lock. To change the timeout interval for locks, use the **ttLockWait** built-in procedure.

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 67.
- 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.
- LOCK_DENIALS_COND counts how often a lock request was not granted because the requestor did not want 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 JDBC 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.

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.

TimesTen uses table locks only with serializable isolation. If your application specifies table locks with any other isolation levels, TimesTen overrides table-level locking and uses row locks. However, the optimizer plan may still display table-level locking hints.

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 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.

Choose the appropriate logging options

The TimesTen Data Manager makes data store transactions durable by maintaining logs of transactions on disk. Log records are written when a transaction commits. Each commit incurs a disk write unless you specify nondurable commits, as discussed in "Use durable commits appropriately" on page 72.

When **DurableCommits**=**1**, the log I/O is amortized over other concurrent connections using a technique called "group commit". Thus response times may be long. Log I/O also affects throughput if there is not sufficient concurrency to hide the disk write. The LOG_FORCES column in the SYS.MONITOR table keeps track of the number of times the log was flushed to disk. Flushing the log to disk too frequently can result in I/O contention between the writes to the log and the checkpoint files. You can use the **LogDir** connection attribute to specify a different I/O path for log files.

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 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 logging to

disk. To use non-durable commits, see “Use durable commits appropriately” on page 72.

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, you should include periodic nondurable commits in your application (see “Size transactions appropriately” on page 71). Durable commits are not permitted. (Because there is no log to write to disk, durability must be achieved through checkpoints.) If logging is disabled, 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 is required to cache Oracle tables.

Prepare statements in advance

As described in “Preparing SQL statements” on page 29, your application should 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.

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 after they have 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 ***Statement.execute*** 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 statements to be shared, it is still a good practice, for performance reasons, to use parameterized statements. Using parameterized statements can reduce prepare overhead beyond what is made available through statement sharing.

Avoid unnecessary prepare operations

Because preparing SQL statements is an expensive operation, your application should minimize the number of calls to the **Connection.prepareStatement**

method. 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. 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 the batch update facility for executing multiple statements

The TimesTen JDBC driver supports the **addBatch**, **clearBatch**, and **executeBatch** methods for the **Statement** and **PreparedStatement** JDBC objects. These APIs can be used by applications to improve performance when multiple SQL update operations are executed through a **Statement** object or when multiple sets of parameters associated with a prepared statement are executed through a **PreparedStatement** object.

For **Statement** objects, a batch consists of a set of SQL write operation statements. Statements that return result sets are not allowed in a batch. A SQL write operation statement is added to a batch by calling the **addBatch** method. The set of SQL statements associated with a batch are executed through the **executeBatch** method. For example:

```
// turn off autocommit
conn.setAutoCommit(false);

Statement stmt = conn.createStatement();
stmt.addBatch("INSERT INTO employees VALUES (1000, 'Joe
Jones')");
stmt.addBatch("INSERT INTO departments VALUES (260, 'Shoe')");
stmt.addBatch("INSERT INTO emp_dept VALUES (1000, 260)");

// submit a batch of update commands for execution
int[] updateCounts = stmt.executeBatch();
conn.commit ();
```

For **PreparedStatement** objects, a batch consists of a set of prepared statement input parameters. Prepared statement parameters are added to the batch by executing **setXXX** calls followed by the **addBatch** call. The batch is executed via the **executeBatch** method. For example:

```
// turn off autocommit
conn.setAutoCommit(false);

PreparedStatement stmt = conn.prepareStatement(
"INSERT INTO employees VALUES (?, ?)");

// first set of parameters
stmt.setInt(1, 2000);
stmt.setString(2, "Kelly Kaufmann");
```

```
stmt.addBatch();

// second set of parameters
stmt.setInt(1, 3000);
stmt.setString(2, "Bill Barnes");
stmt.addBatch();

// submit the batch for execution
int[] updateCounts = stmt.executeBatch();
conn.commit ();
```

For more information on using the JDBC batch update facility see the Java Platform API specification for the **Statement** and objects.

Bulk fetch rows of TimesTen data

TimesTen provides an extension that allows an application to fetch multiple rows of data. For applications that retrieve large amounts of TimesTen data, fetching multiple rows can increase performance greatly. However, when using read committed isolation, locks are held on all rows being retrieved until the application has received all the data, decreasing concurrency. For more information on this feature, see "Fetching multiple rows of data" on page 35.

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 68.) 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 72.

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

By default, each TimesTen transaction results in a disk write at commit time. 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 nondurable commits. Nondurable 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.

The advantage of nondurable 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 nondurable 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.

Use the ResultSet.getString method sparingly

Because Java strings are immutable, **ResultSet.getString** must allocate space for a new string (in addition to translating the underlying C-string to a Unicode string) each time it is called. Therefore, this is one of the costliest calls in JDBC.

In addition, you should not call **ResultSet.getString** on primitive numeric types, like Byte or Integer, unless it is absolutely necessary. It is much faster to call **ResultSet.getInt** on an integer column, for example.

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.

Use the default **ResultSet.get***XXX* method for the data type of the data in the underlying database. For example, if the data type of the data is DOUBLE, to avoid data conversion in the JDBC driver you should call **getDouble**. For information on the default **get***XXX* method for a particular data type, refer to the "JDBC Quick Reference" in the book *JDBC Database Access with Java* by Hamilton, Cattell, Fisher. Similarly, use the default **PreparedStatement.set***XXX* method for the input parameter in an SQL statement. For example, if you are inserting data into a CHAR column using a **PreparedStatement**, you should use **setString**.

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 **Connection.rollback()** 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 67) 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 call the **ttCkpt** procedure to checkpoint the data store 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.

It is generally best to call **ttCkpt** to perform a non-blocking (or "fuzzy") checkpoint than **ttCkptBlocking** to perform a blocking checkpoint. Non-blocking checkpoints may take longer, but they permit other transactions to operate against the data store at the same time and thus impose less overall 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 column of the SYS.MONITOR table indicates how often checkpoints have successfully completed.

Tuning JMS/XLA applications

This section contains specific performance tuning tips for applications that use the JMS/XLA API. JMS/XLA has some overhead that makes it slower than using the C XLA API. In the C API, records are returned to the user in a batch. In the JMS model an object is instantiated and each record is presented one at a time in a callback to the **MessageListener onMessage** method. High performance applications can use some tuning to overcome some of this overhead.

Configure xlaPrefetch parameter

The code underlying the JMS layer that reads the transaction log is more efficient if it can fetch as many rows as possible before presenting the object/rows to the user. The amount of prefetching is controlled in the jmsxla.xml configuration file with the “xlaPrefetch” parameter. Set the prefetch count to a large value like 100 or 1000.

Batch calls to ttXlaAcknowledge

Calls to **ttXlaAcknowledge** move the bookmark and involve updates to system tables, so one way to increase throughput is to wait until several transactions have been seen before issuing the call. This means that the reader application must have some tolerance for seeing the same set of records more than once. Moving the bookmark can be done manually using the **Session CLIENT_ACKNOWLEDGE** mode when instantiating a session:

```
Session session = connection.createSession
                  (false, Session.CLIENT_ACKNOWLEDGE);
```

For many applications, setting this value to 100 is a reasonable choice.

Increase log buffer size

A larger log buffer size is called for when using XLA. When XLA is turned on, additional log records are generated to store additional information for XLA. To ensure the log buffer is properly sized, one can watch for changes in the SYS.MONITOR table entries LOG_FS_READS and LOG_BUFFER_WAITS. For optimal performance, both of these values should remain 0. Increasing the log buffer size may be necessary to ensure the values remain 0.

Handling high event rates

The synchronous interface is suitable only for applications with low event rates and for which AUTO_ACKNOWLEDGE or DUPS_OK_ACKNOWLEDGE acknowledgement modes are acceptable. Applications that require

CLIENT_ACKNOWLEDGE acknowledgement mode and applications with high event rates should use the asynchronous interface for receiving updates. They should acknowledge the messages on the call back thread itself if they are using CLIENT_ACKNOWLEDGEMENT as acknowledgement mode. See "Receiving and processing updates" on page 58.

5

JDBC Reference

This appendix lists the JDBC interfaces supported by TimesTen and the TimesTen extensions to JDBC. The main topics are:

- Supported JDBC interfaces
- TimesTen extensions to JDBC

Supported JDBC interfaces

This section lists all interfaces supported by the TimesTen implementation of JDBC. This section includes the following topics:

- Support for interfaces in java.sql package
- Support for interfaces in javax.sql package

For complete reference information, see the Java documentation:

http://java.sun.com/j2se/1.4.2/docs/api/index.html

http://java.sun.com/j2se/1.5.0/docs/api/index.html

Support for interfaces in java.sql package

The supported java.sql interfaces are:

- **CallableStatement**
- **Connection**
- **DatabaseMetaData**
- **Driver**
- **ParameterMetaData**
- **PreparedStatement**
- **ResultSet**
- **ResultSetMetaData**
- **Statement**

The supported java.sql classes are:

- **Date**
- **DriverManager**
- **DriverPropertyInfo**
- **Time**
- **Timestamp**
- **Types**
- **DataTruncation**
- **SQLException**
- **SQLWarning**

In general, TimesTen does not support:

- CLOB, BLOB, Array, Struct and Ref data types.
- Scrollable and updatable result sets.
- Result set holdability.
- Calendar for set/get Date.
- Calendar for set/get Time.

Restrictions for the specific interfaces are described below.

CallableStatement

TimesTen implements the **java.sql.CallableStatement** interface

Restrictions:

- You cannot pass parameters to CallableStatement objects by name. You must set parameters by ordinal numbers.
- You cannot use the SQL escape syntax.

Connection

TimesTen implements the **java.sql.Connection** interface.

Restriction:

- No support for savepoints.

DatabaseMetaData

TimesTen implements the **java.sql.DatabaseMetaData** interface.

No restrictions.

Driver

TimesTen implements the **java.sql.Driver** interface from the classes:

`com.timesten.jdbc.TimesTenDriver`

`com.timesten.jdbc.TimesTenClientDriver`

No restrictions.

ParameterMetaData

TimesTen implements the **java.sql.ParameterMetaData** interface.

Restrictions:

- No support for **ResultSetMetaData getMetaData()**.
- The JDBC driver cannot determine whether or not a column is nullable and will always return *parameterNullableUnknown* from calls to **ParameterMetaData. isNullable(int param)**.
- The **ParameterMetaData.getScale()** returns 1 for VARCHAR, NVARCHAR and VARBINARY data types if they are INLINE.

Note: Scale is of no significance to these data types.

PreparedStatement

TimesTen fully implements the **java.sql.PreparedStatement** interface.

No restrictions.

ResultSet

TimesTen implements the **java.sql.ResultSet** interface.

Restrictions:

- You cannot have multiple open ResultSet objects per statement.
- You cannot specify the holdability of a result set, so a cursor cannot remain open after it has been committed.

See "Working with result sets" on page 38 for more information.

ResultSetMetaData

TimesTen implements the **java.sql.ResultSetMetaData** interface

No restrictions.

Statement

TimesTen implements the **java.sql.Statement** interface

No restrictions.

Support for interfaces in javax.sql package

- **DataSource**
- **ConnectionPoolDataSource**
- **PooledConnection**
- **XADataSource**

DataSource

TimesTen implements **javax.sql.DataSource** using:

`com.timesten.jdbc.TimesTenDataSource`

ConnectionPoolDataSource

TimesTen implements the JDBC **javax.sql.ConnectionPoolDataSource** factory for **javax.sql.PooledConnection** using:

`com.timesten.jdbc.ObservableConnectionDS`

PooledConnection

TimesTen implements the **javax.sql.PooledConnection** interface using:

`com.timesten.jdbc.ObservableConnection`

XADataSource

TimesTen implements **javax.sql.XADataSource** using:

`com.timesten.jdbc.xa.TimesTenXADataSource`

TimesTen extensions to JDBC

The TimesTen extensions to JDBC are:

- TimesTenConnection
- TimesTenVendorCode

TimesTenConnection

TimesTen implements the connection-level prefetch feature described in "Fetching multiple rows of data" on page 35 with the class:

`com.timesten.sql.TimesTenConnection`

The methods implemented by **TimesTenConnection** are:

- **getTtPrefetchClose**
- **getTtPrefetchCount**
- **isDataStoreValid**
- **setTtPrefetchClose**
- **setTtPrefetchCount**

getTtPrefetchClose

```
public boolean getTtPrefetchClose()
    throws SQLException;
```

Returns the current state of TT_PREFETCH_CLOSE.

Returns:	True if TT_PREFETCH_CLOSE is enabled, or false if disabled.
Throws:	SQLException - if a database access error occurs.

getTtPrefetchCount

```
public int getTtPrefetchCount()
    throws SQLException;
```

Returns the current prefetch count set for the TimesTen connection.

Returns:	The current prefetch count.
Throws:	SQLException - if a database access error occurs.

isDataStoreValid

```
public boolean isDataStoreValid()
    throws SQLException;
```

Detect whether or not the data store is valid.

Returns:	True if the data store is valid, or false if invalid.
Throws:	SQLException - if a database access error occurs.

setTtPrefetchClose

```
public void setTtPrefetchClose(boolean enable)
    throws SQLException;
```

Sets the state of TT_PREFETCH_CLOSE to true or false.

Parameters:	enable - set to true to enable TT_PREFETCH_CLOSE or false to disable.
Throws:	SQLException - if a database access error occurs.

setTtPrefetchCount

```
public void setTtPrefetchCount(int count)
    throws SQLException;
```

Establishes the number of rows to be prefetched for all of the statements on the TimesTen connection. Your application must have a direct driver connection to the data store to use this method. See "Fetching multiple rows of data" on page 35 for details.

Parameters:	count - The number of prefetches to set for the connection. The count can be any integer from 0 to 128, inclusive.
Throws:	SQLException - if a database access error occurs.

TimesTenVendorCode

The **TimesTenVendorCode** interface defines error names for TimesTen error numbers:

`com.timesten.jdbc.TimesTenVendorCode`

See the *Oracle TimesTen In-Memory Database JDBC API Extensions* for the complete list of errors. It is located in *`install_dir`*`/doc/ttjava.zip`.

6

JMS/XLA Reference

This chapter provides reference information for the JMS/XLA API.

The topics in this section are:

XLA MapMessage contents

A **`MapMessage`** contains a set of typed name/value pairs that correspond to the fields in an XLA update header, which is published as the C structure **ttXlaUpdateDesc_t**. The fields contained in a `MapMessage` depend on what type of update it is.

Update type

Each `MapMessage` returned by the JMS/XLA API contains at least one name/value pair called `__TYPE` (with 2 underscores) that identifies the type of update described in the message as an integer value. The types are specified as integer values. As a convenience, you can use the constants defined in `com.timesten.dataserver.jmsxla.XlaConstants` to compare against the integer types. The following table shows the supported types:

Type	Description
`ADD_COLUMNS`	Indicates that columns have been added.
`COMMIT_FIELD`	The name of the field in a message that contains a commit.

Type	Description
COMMIT_ONLY	Indicates that a commit has occurred.
CONTEXT_FIELD	The name of the field in a message that contains the context value passed to **ttApplicationContext** as a byte array.
CREATE_INDEX	Indicates that an index has been created.
CREATE_SEQ	Indicates that a sequence has been created.
CREATE_TABLE	Indicates that a table has been created.
CREATE_VIEW	Indicates that a view has been created.
DELETE	Indicates that a row has been deleted.
DROP_COLUMNS	Indicates that columns have been dropped.
DROP_INDEX	Indicates that an index has been dropped.
DROP_SEQ	Indicates that a sequence has been dropped.
DROP_TABLE	Indicates that a table has been dropped.
DROP_VIEW	Indicates that a view has been dropped.
FIRST_FIELD	The name of the field that contains the flag that indicates the first record in a transaction.
INSERT	Indicates that a row has been inserted.

Type	Description
MTYP_FIELD	The name of the field in a message that contains type information.
MVER_FIELD	The name of the field in a message that contains the log file number of the XLA record.
NULLS_FIELD	The name of the field in a message that contains the list of fields that have null values.
REPL_FIELD	The name of the field in a message that contains the flag that indicates that the update was applied by replication.
TBLNAME_FIELD	The name of the field in a message that contains the table name.
TBLOWNER_FIELD	The name of the field in a message that specifies the table owner.
TRUNCATE	Indicates that a table has been truncated.
TYPE_FIELD	The name of the field in a message that specifies the message type.
UPDATE	Indicates that a row has been updated.
UPDATE_DESCRIPTOR_FIELD	The name of the field that returns **ttXlaUpdateDesc_t** as a byte array.
UPDATED_COLUMNS_FIELD	The name of the field in a message that contains the list of updated columns.

XLA flags

For all update types, the `MapMessage` contains name/value pairs that indicate:

- Whether this is the first record of a transaction
- Whether this is the last record of a transaction
- Whether the update was performed by replication
- Which table was updated
- The owner of the updated table

The name/value pairs that contain these XLA flags are described in Table 6.1. Each name is preceded by two underscores:

Table 6.1 XLA Flags

Name	Description	Corresponding ttXlaTblDesc_t Field
__COMMIT	Indicates that this is the last record in a transaction and that a commit was performed after this operation. Only included in the MapMessage if UPDCOMMIT is on.	TT_UPDCOMMIT
__FIRST	Indicates that this is the first record in a new transaction. Only included in the MapMessage if UPDFIRST is on.	TT_UPDFIRST
__REPL	Indicates that this change was applied to the database via replication. Only included in the MapMessage if UPDREPL is on.	TT_UPDREPL
__UPDCOLS	Only used for UPDATETUP records, this flag indicates that the XLA update descriptor contains a list of columns that were actually modified by the operation. Specified as a String that contains a semicolon delimited list of column names. Only included in the MapMessage if UPDCOLS is on.	TT_UPDCOLS

DML event data formats

Many Data Manipulation Language (DML) operations generate XLA updates that can be monitored by XLA event handlers. This section describes the contents of the MapMessage objects that are generated for these operations.

Table data

For INSERT, UPDATE and DELETE operations MapMessages contain two name/value pairs called __TBLOWNER and __TBLNAME. These fields

describe the name and owner of the table that is being updated. For example, for a table owned by user SCOTT, called EMPLOYEES, which could be referred to in a SQL statement as SCOTT.EMPLOYEES, MapMessages related to this table contain a field called __TBLOWNER whose value is the string SCOTT, and a field called __TBLNAME whose value is the string EMPLOYEES.

Row data

For INSERT and DELETE operations, a complete image of the inserted or deleted row is included in the message and all column values are available.

For UPDATE operations, the complete before and after image of the row is available, as well as a list of column numbers indicating which columns were actually modified. You access the column values using the names of the columns. The column names in the "before" image all begin with a single underscore. For example, `<columnname>` contains the new value and `_<columnname>` contains the old value.

If the value of a column is NULL, it is omitted from the column list. The `__NULLS` name/value pair contains a semicolon-delimited list of the columns that contain NULL values.

Context information

If the **ttApplicationContext** built-in procedure was used to encode context information in an XLA record, that information is included in the `__CONTEXT` name/value pair in the `MapMessage`. If no context information is provided, the `__CONTEXT` value is not included in the `MapMessage`.

DDL event data formats

Many Data Definition Language (DDL) operations generate XLA updates that can be monitored by XLA event handlers. This section describes the contents of the `MapMessage` objects that are generated for these operations.

CREATE_TABLE

Messages with `__TYPE=1 (XlaConstants.CREATE_TABLE)` indicate that a table has been created. Table 6.2 shows the name/value pairs that are included in a `MapMessage` generated for a CREATE_TABLE operation.

Table 6.2 Create Table Data Provided in Update Messages

Name	Value
`OWNER`	String value of the owner of the created table.
`NAME`	String value of the name of the created table.
`PK_COLUMNS`	String value containing the names of the columns in the primary key for this table. If the table has no primary key, the PK_COLUMNS value is not specified. Format: `<col1name>[;<col2name>[;<col3name>[;…]]]`
`COLUMNS`	String value containing the names of the columns in the table. Format: `<col1name>[;<col2name>[;<col3name>[;…]]]` **Note:** For each column in the table, additional name/value pairs that describes the column are included in the `MapMessage`.
`_column_name``_TYPE`	Integer value representing the datatype of this column. From `java.sql.types`.
`_column_name``_PRECISION`	Integer value containing the precision of this column (for NUMERIC / DECIMAL).
`_column_name``_SCALE`	Integer value containing the scale of this column (for NUMERIC / DECIMAL).
`_column_name``_SIZE`	Integer value indicating the maximum size of this column (for CHAR / VARCHAR / BINARY / VARBINARY).

Table 6.2 Create Table Data Provided in Update Messages

Name	Value
*_column_name*_NULLABLE	Boolean value indicating whether this column can have a NULL value.
*_column_name*_OUTOFLINE	Boolean value indicating whether this column is stored in the “inline” or “out of line” part of the tuple.
*_column_name*_INPRIMARYKEY	Boolean value indicating whether this column is part of the primary key of the table.

DROP_TABLE

Messages with __TYPE=2 (XlaConstants.DROP_TABLE) indicate that a table has been dropped. Table 6.3 shows the name/value pairs that are included in a MapMessage generated for a DROP_TABLE operation.

Table 6.3 Drop Table Data Provided in Update Messages

Name	Value
OWNER	String value of the owner of the sequence.
NAME	String value of the name of the dropped sequence.

CREATE_INDEX

Messages with __TYPE=3 (XlaConstants.CREATE_INDEX) indicate that an index has been created. Table 6.4 shows the name/value pairs that are included in a MapMessage generated for a CREATE_INDEX operation.

Table 6.4 Create Index Data Provided in Update Messages

Name	Value
TBLOWNER	String value of the owner of the table on which the index was created.
TBLNAME	String value of the name of the table on which the index was created.

Table 6.4 Create Index Data Provided in Update Messages

Name	Value
IXNAME	String value of the name of the created index.
INDEX_TYPE	String value representing the index type: “P” (Primary Key), “F” (Foreign Key), or “R” (Regular).
INDEX_METHOD	String value representing the index method: “H” (Hash) or “T” (T-tree).
UNIQUE	Boolean value indicating whether or not the index is UNIQUE.
HASH_PAGES	Integer value representing the number of PAGES in a hash index. (Not specified for T-Tree indexes).
COLUMNS	String value describing the columns in the index. Format: `<col1name>[;<col2name>[;<col3name>[;…]]]`

DROP_INDEX

Messages with `__TYPE=4 (XlaConstants.DROP_INDEX)` indicate that an index has been dropped. Table 6.5 shows the name/value pairs that are included in a `MapMessage` generated for a DROP_INDEX operation.

Table 6.5 Drop Index Data Provided in Update Messages

Name	Value
OWNER	String value of the owner of the table on which the index was dropped.
TABLE_NAME	String value of the name of the table on which the index was dropped.
INDEX_NAME	String value of the name of the dropped index.

ADD_COLUMNS

Messages with `__TYPE=5 (XlaConstants.ADD_COLUMNS)` indicate that a table has been altered by adding new columns. Table 6.6 shows the name/value pairs that are included in a `MapMessage` generated for a ADD_COLUMNS operation.

Table 6.6 Add Columns Data Provided in Update Messages

Name	Value
`OWNER`	String value of the owner of the altered table.
`NAME`	String value of the name of the altered table.
`PK_COLUMNS`	String value containing the names of the columns in the primary key for this table. If the table has no primary key, the PK_COLUMNS value is not specified. Format: `<col1name>[;<col2name>[;<col3name>[;…]]]`
`COLUMNS`	String value containing the names of the columns added to the table. Format: `<col1name>[;<col2name>[;<col3name>[;…]]]` **Note:** For each added column, additional name/value pairs that describe the column are included in the `MapMessage`.
`_column_name``_TYPE`	Integer value representing the datatype of this column. From `java.sql.types`.
`_column_name``_PRECISION`	Integer value containing the precision of this column (for NUMERIC / DECIMAL).
`_column_name``_SCALE`	Integer value containing the scale of this column (for NUMERIC / DECIMAL).
`_column_name``_SIZE`	Integer value indicating the maximum size of this column (for CHAR / VARCHAR / BINARY / VARBINARY).

Table 6.6 Add Columns Data Provided in Update Messages

Name	Value
*_column_name*_NULLABLE	Boolean value indicating whether this column can have a NULL value.
*_column_name*_OUTOFLINE	Boolean value indicating whether this column is stored in the “inline” or “out of line” part of the tuple.
*_column_name*_INPRIMARYKEY	Boolean value indicating whether this column is part of the primary key of the table.

DROP_COLUMNS

Messages with __TYPE=6 (XlaConstants.DROP_COLUMNS) indicate that a table has been altered by dropping existing columns. Table 6.7 shows the name/value pairs that are included in a MapMessage generated for a DROP_COLUMNS operation.

Table 6.7 Drop Columns Data Provided in Update Messages

Name	Value
OWNER	String value of the owner of the altered table.
NAME	String value of the name of the altered table.
COLUMNS	String value containing the names of the columns dropped from the table. Format: <col1name>[;<col2name>[;<col3name>[;…]]] **Note:** For each dropped column, additional name/value pairs that describe the column are included in the MapMessage.
*_column_name*_TYPE	Integer value representing the datatype of this column. From java.sql.types.

Table 6.7 Drop Columns Data Provided in Update Messages

Name	Value
*_column_name*_PRECISION	Integer value containing the precision of this column (for NUMERIC / DECIMAL).
*_column_name*_SCALE	Integer value containing the scale of this column (for NUMERIC / DECIMAL).
*_column_name*_SIZE	Integer value indicating the maximum size of this column (for CHAR / VARCHAR / BINARY / VARBINARY).
*_column_name*_NULLABLE	Boolean value indicating whether this column can have a NULL value.
*_column_name*_OUTOFLINE	Boolean value indicating whether this column is stored in the “inline” or “out of line” part of the tuple.
*_column_name*_INPRIMARYKEY	Boolean value indicating whether this column is part of the primary key of the table.

CREATE_VIEW

Messages with `__TYPE=14 (XlaConstants.CREATE_VIEW)` indicate that a materialized view has been created. Table 6.8 shows the name/value pairs that are included in a `MapMessage` generated for a CREATE_VIEW operation.

Table 6.8 Create View Data Provided in Update Messages

Name	Value
OWNER	String value of the owner of the created view.
NAME	String value of the name of the created view.

DROP_VIEW

Messages with __TYPE=15 (XlaConstants.DROP_VIEW) indicate that a materialized view has been dropped. Table 6.8 shows the name/value pairs that are included in a MapMessage generated for a DROP_VIEW operation

Table 6.9 Drop View Data Provided in Update Messages

Name	Value
OWNER	String value of the owner of the dropped view.
NAME	String value of the name of the dropped view.

CREATE_SEQ

Messages with __TYPE=16 (XlaConstants.CREATE_SEQ) indicate that a SEQUENCE has been created. Table 6.10 shows the name/value pairs that are included in a MapMessage generated for a CREATE_SEQ operation

Table 6.10 Create Sequence Data Provided in Update Messages

Name	Value
OWNER	String value of the owner of the created sequence.
NAME	String value of the name of the created sequence.
CYCLE	Boolean value indicating whether the CYCLE option was specified on the new sequence.
INCREMENT	Long value indicating the INCREMENT BY option specified for the new sequence.
MIN_VALUE	Long value indicating the MINVALUE option specified for the new sequence.
MAX_VALUE	Long value indicating the MAXVALUE option specified for the new sequence.

DROP_SEQ

Messages with __TYPE=17 (XlaConstants.DROP_SEQ) indicate that a sequence has been dropped. Table 6.11 shows the name/value pairs that are included in a

MapMessage generated for a DROP_SEQ operation

Table 6.11 Drop Sequence Data Provided in Update Messages

Name	Value
OWNER	String value of the owner of the dropped table.
NAME	String value of the name of the dropped table.

TRUNCATE

Messages with __TYPE=18(XlaConstants.TRUNCATE) indicate that a table has been truncated. All rows in the table have been deleted. Table 6.12 shows the name/value pairs that are included in a MapMessage generated for a TRUNCATE operation

Table 6.12 Truncate Data Provided in Update Messages

Name	Value
OWNER	String value of the owner of the truncated table.
NAME	String value of the name of the truncated table.

Data type mapping

The following table lists access methods for the data types supported by TimesTen. For more information about data types, see Chapter 1, "Data Types"in *Oracle TimesTen In-Memory Database SQL Reference Guide*.

TimesTen Column Type	Read With
CHAR(*n*)	MapMessage.getString
VARCHAR(*n*)	MapMessage.getString
NCHAR(*n*)	MapMessage.getString
NVARCHAR(*n*)	MapMessage.getString
NVARCHAR2(*n*)	MapMessage.getString
DOUBLE	MapMessage.getDouble

TimesTen Column Type	Read With
FLOAT	MapMessage.getFloat
DECIMAL(*p*,*s*)	MapMessage.getString. Can be converted to BigDecimal or to Double by the application.
NUMERIC(*p*,*s*)	MapMessage.getString. Can be converted to BigDecimal or to Double by the application.
INTEGER	MapMessage.getInt
SMALLINT	MapMessage.getShort
TINYINT	MapMessage.getShort
BIGINT	MapMessage.getLong
BINARY(*n*)	MapMessage.getBytes
VARBINARY(*n*)	MapMessage.getBytes
DATE	MapMessage.getLong, MapMessage.getString MapMessage.getLong returns microseconds since epoch (00:00:00 UTC, January 1, 1970). Can be converted to Date or Calendar by the application.
TIME	MapMessage.getString. Can be converted to Date or Calendar by the application.
TIMESTAMP	MapMessage.getLong, MapMessage.getString MapMessage.getLong returns microseconds since epoch (00:00:00 UTC, January 1, 1970). It truncates nanoseconds. Use MapMessage.getString if you require nanosecond precision. Can be converted to Date or Calendar by the application.
TT_CHAR	MapMessage.getString
TT_VARCHAR	MapMessage.getString

TimesTen Column Type	Read With
TT_NCHAR	MapMessage.getString
TT_NVARCHAR	MapMessage.getString
ORA_CHAR	MapMessage.getString
ORA_VARCHAR2	MapMessage.getString
ORA_NCHAR	MapMessage.getString
ORA_NVARCHAR2	MapMessage.getString
VARCHAR2	MapMessage.getString
TT_TINYINT	MapMessage.getShort
TT_SMALLINT	MapMessage.getShort
TT_INTEGER	MapMessage.getInt
TT_BIGINT	MapMessage.getLong
BINARY_FLOAT	MapMessage.getFloat
BINARY_DOUBLE	MapMessage.getDouble
REAL	MapMessage.getFloat
NUMBER	MapMessage.getString
ORA_NUMBER	MapMessage.getString
TT_DECIMAL	MapMessage.getString
TT_TIME	MapMessage.getString
TT_DATE	MapMessage.getLong, MapMessage.getString MapMessage.getLong returns microseconds since epoch (00:00:00 UTC, January 1, 1970).
TT_TIMESTAMP	MapMessage.getLong, MapMessage.getString MapMessage.getLong returns microseconds since epoch (00:00:00 UTC, January 1, 1970).
ORA_DATE	MapMessage.getLong, MapMessage.getString MapMessage.getLong returns microseconds since epoch (00:00:00 UTC, January 1, 1970).

TimesTen Column Type	Read With
ORA_TIMESTAMP	MapMessage.getLong, MapMessage.getString MapMessage.getLong returns microseconds since epoch (00:00:00 UTC, January 1, 1970). It truncates nanoseconds. Use MapMessage.getString if you require nanosecond precision.
TT_BINARY	MapMessage.getBytes
TT_VARBINARY	MapMessage.getBytes

Internationalization support

JMS/XLA uses a UTF-16 character set for the following data types:

TT_CHAR
TT_VARCHAR
ORA_CHAR
ORA_VARCHAR2
TT_NCHAR
TT_NVARCHAR
ORA_NCHAR
ORA_NVARCHAR2
NCHAR
NVARCHAR
NVARCHAR2

JMS classes for event handling

You can use JMS classes when programming to the JMS/XLA API. The JMS/XLA API supports only publish/subscribe messaging. JMS classes include:

Message (parent class only)
TopicConnectionFactory
Topic
TopicSubscriber
Connection
Session
ConnectionMetaData
MapMessage
TopicConnection
TopicSession
ConnectionFactory
Destination

MessageConsumer
ExceptionListener

See http://java.sun.com/j2ee/1.4/docs/api/javax/jms/package-frame.html for documentation for these classes.

JMS/XLA replication API

The TimesTen `com.timesten.dataserver.jmsxla` package includes the **TargetDataStore** interface and the **TargetDataStoreImpl** class.

See the *Oracle TimesTen In-Memory Database JMS/XLA API* for complete documentation. It is located in *install_dir*`/doc/ttjava.zip`.

TargetDataStore

This interface is used to apply XLA update records from a source data store to a target data store. The source and target data store schema must be identical for the affected tables.

This interface includes the following methods:

Method	Description
`apply`	Applies XLA update descriptor to the target data store
`close`	Closes the connections to the data store and releases the resources
`commit`	Performs a manual commit
`getAutoCommitFlag`	Returns the value of the autocommit flag
`getConnectString`	Returns the data store connection string
`getUpdateConflictCheckFlag`	Returns the value of the flag for checking update conflicts
`isClosed`	Checks whether the object is closed
`isDataStoreValid`	Checks whether the data store is valid
`rollback`	Rolls back the last transaction

`setAutoCommitFlag`	Sets the flag for autocommit during apply
`setUpdateConflictCheckFlag`	Sets the flag for checking update conflicts during apply

TargetDataStoreImpl

This class creates connections and XLA handles for a target data store.It implements the **TargetDataStore** interface.

JMS message header fields

Table 6.13 shows the JMS message header fields provided by JMS/XLA.

Table 6.13 JMS/XLA Header Fields

Header	Contents
`JMSMessageId`	The log file number of the XLA record.
`JMSType`	The string representation of the `__TYPE` field

Index

U

X