Oracle® TimesTen In-Memory Database

Java Developer's Guide

Release 18.1

E61200-08

July 2020

ORACLE®

Oracle TimesTen In-Memory Database Java Developer's Guide, Release 18.1

E61200-08

Contents

3 Using JMS/XLA for Event Management

4 Distributed Transaction Processing: JTA

5 Java Application Tuning

6 JMS/XLA Reference

Index

Preface

Oracle TimesTen In-Memory Database (TimesTen) is a relational database that is memory-optimized for fast response and throughput. The database resides entirely in memory at runtime and is persisted to the file system.

- Oracle TimesTen In-Memory Database in classic mode, or TimesTen Classic, refers to single-instance and replicated databases (as in previous releases).
- Oracle TimesTen In-Memory Database in grid mode, or TimesTen Scaleout, refers to a multiple-instance distributed database. TimesTen Scaleout is a grid of interconnected hosts running instances that work together to provide fast access, fault tolerance, and high availability for in-memory data.
- TimesTen alone refers to both classic and grid modes (such as in references to TimesTen utilities, releases, distributions, installations, actions taken by the database, and functionality within the database).
- TimesTen Application-Tier Database Cache, or TimesTen Cache, is an Oracle Database Enterprise Edition option. TimesTen Cache is ideal for caching performance-critical subsets of an Oracle database into cache tables within a TimesTen database for improved response time in the application tier. Cache tables can be read-only or updatable. Applications read and update the cache tables using standard Structured Query Language (SQL) while data synchronization between the TimesTen database and the Oracle database is performed automatically. TimesTen Cache offers all of the functionality and performance of TimesTen Classic, plus the additional functionality for caching Oracle Database tables.
- TimesTen Replication features, available with TimesTen Classic or TimesTen Cache, enable high availability.

TimesTen supports standard application interfaces JDBC, ODBC, and ODP.NET; Oracle interfaces PL/SQL, OCI, and Pro*C/C++; and the TimesTen TTClasses library for C++.

This document covers TimesTen support for JDBC.

The following topics are discussed in the preface:

Audience

This guide is for anyone developing or supporting applications that use TimesTen through JDBC.

In addition to familiarity with JDBC, you should be familiar with TimesTen, SQL (Structured Query Language), and database operations.

Related documents

TimesTen documentation is available at
https://docs.oracle.com/database/timesten-18.1.

This includes Javadoc for TimesTen JDBC and JMS/XLA support.

For reference information on standard JDBC, see the following for information about the `java.sql` and `javax.sql` packages:

https://docs.oracle.com/javase/8/docs/api/java/sql/package-summary.html
https://docs.oracle.com/javase/8/docs/api/javax/sql/package-summary.html

Javadoc for standard Java EE classes and interfaces is available at this location:

https://javaee.github.io/javaee-spec/javadocs/

Oracle Database documentation is also available on the Oracle documentation website. This may be especially useful for Oracle Database features that TimesTen supports but does not attempt to fully document.

In particular, the following Oracle Database documents may be of interest.

- *Oracle Database SQL Language Reference*
- *Oracle Database JDBC Developer's Guide*

Conventions

TimesTen supports multiple platforms. Unless otherwise indicated, the information in this guide applies to all supported platforms. The term Windows applies to all supported Windows platforms. The term UNIX applies to all supported UNIX platforms. The term Linux is used separately. Refer to "Platforms and compilers" in *Oracle TimesTen In-Memory Database Release Notes* (`README.html`) in your installation directory for specific platform versions supported by TimesTen.

Note: In TimesTen documentation, the terms "data store" and "database" are equivalent. Both terms refer to the TimesTen database.

This document uses the following text conventions:

Convention	Meaning
italic	Italic type indicates terms defined in text, book titles, or emphasis.
`monospace`	Monospace type indicates code, commands, URLs, class names, interface names, method names, function names, attribute names, directory names, file names, text that appears on the screen, or text that you enter.

Convention	Meaning
italic monospace	Italic monospace type indicates a placeholder or a variable in a code example for which you specify or use a particular value. For example: `LIBS = -Ltimesten_home/install/lib -ltten` Replace *timesten_home* with the path to the TimesTen instance home directory.
[ ]	Square brackets indicate that an item in a command line is optional.
{ }	Curly braces indicated that you must choose one of the items separated by a vertical bar (|) in a command line.
|	A vertical bar (or pipe) separates alternative arguments.
. . .	An ellipsis (. . .) after an argument indicates that you may use more than one argument on a single command line. An ellipsis in a code example indicates that what is shown is only a partial example.
% or $	The percent sign or dollar sign indicates the Linux or UNIX shell prompt, depending on the shell that is used.
#	The number (or pound) sign indicates the Linux or UNIX root prompt.

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

Convention	Meaning
installation_dir	The path that represents the directory where TimesTen is installed.
timesten_home	The path that represents the home directory of a TimesTen instance.
release or *rr*	The first two parts in a release number, with or without the dot. The first two parts of a release number represent a major TimesTen release. For example, 181 or 18.1 represents TimesTen Release 18.1.
DSN	TimesTen data source name (for the TimesTen database).

Note: TimesTen release numbers are reflected in items such as TimesTen utility output, file names, and directory names, all of which are subject to change with every minor or patch release. The documentation cannot always be up to date. It seeks primarily to show the basic form of output, file names, directory names, and other code that may include release numbers. You can confirm the current release number by looking at *Oracle TimesTen In-Memory Database Release Notes* or executing the `ttVersion` utility.

Documentation Accessibility

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

Access to Oracle Support

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

What's New

This section summarizes new features and functionality of TimesTen Release 18.1 that are documented in this guide, providing links into the guide for more information.

New features in Release 18.1.2.1.0

- The TimesTen JDBC driver implements the JDBC 4.2 API (Java 8) and is certified to work with Java 8, 9, and 10 runtime environments (JRE).

  This includes support for standard REF CURSORs, large update counts, `SQLType`, and `DatabaseMetaData` enhancements. (See "Working with REF CURSORs" on page 2-25.)

New features in Release 18.1.1.2.0

- TimesTen Scaleout includes a new client routing API that enables Java client applications to route connections to a grid element based on the key value for a hash distribution key. This feature enables the client application to connect to the element that stores the row with the specified key value, avoiding unnecessary communication between the element storing the row and the element connected to your application. For more information, see "Client routing in TimesTen Scaleout" on page 2-55.

New features in Release 18.1.1.1.0

- The TimesTen JDBC driver implements the JDBC 4.1 API (Java 7) and is certified to work with Java 7 and Java 8 runtime environments (JRE).
- There are new error codes for manual retry after transient errors. TimesTen automatically resolves most transient errors (which is particularly important for TimesTen Scaleout), but if your application detects certain error codes, it is suggested to retry the current transaction or most recent API call, as applicable. Refer to "Retrying after transient errors (JDBC)" on page 2-45 for details.

1

Java Development Environment

This chapter provides information about the Java development environment and related considerations. It includes the following topics:

- Installing TimesTen and the JDK
- Setting the environment for Java development
- Compiling Java applications
- TimesTen Quick Start and sample applications

Installing TimesTen and the JDK

Install and configure TimesTen for your environment, as described in *Oracle TimesTen In-Memory Database Installation, Migration, and Upgrade Guide* for TimesTen Classic or "Prerequisites and Installation of TimesTen Scaleout" in *Oracle TimesTen In-Memory Database Scaleout User's Guide* for TimesTen Scaleout.

Install the Java JDK as described in your Java installation documentation.

> **Note:** The TimesTen JDBC driver implements the JDBC 4.2 API (Java 8) and is certified to work with Java 8, 9, and 10 runtime environments (JRE).

For TimesTen Scaleout, DSNs are automatically available for all connectables defined in the grid. Refer to "Connecting to a database" in *Oracle TimesTen In-Memory Database Scaleout User's Guide*.

For TimesTen Classic, after you have installed and configured TimesTen, create a database DSN (data source name) as described in "Managing TimesTen Databases" in *Oracle TimesTen In-Memory Database Operations Guide*. (A DSN is a logical name that identifies a TimesTen database and the set of connection attributes user for connecting to the database.) Topics of particular interest include the following:

- "Connecting using the TimesTen JDBC driver and driver manager"
- "Overview of user and system DSNs"
- "Defining DSNs for direct or client/server connections"
- "Thread programming with TimesTen"
- "Creating a DSN on Linux and UNIX for TimesTen Classic"

Setting the environment for Java development

Before you begin developing Java applications for TimesTen, you must set your environment appropriately. This includes setting environment variables.

Environment variables and runtime access to the Instant Client are configured through the appropriate `ttenv` script in the *`timesten_home`*`/bin` directory, where *`timesten_home`* is the TimesTen instance home directory: `ttenv.sh` and `ttenv.csh` for Linux and UNIX platforms (where which you use depends on your shell) and `ttenv.bat` for Windows platforms.

See "Environment variables" and "Java environment variables" in *Oracle TimesTen In-Memory Database Installation, Migration, and Upgrade Guide* for information about environment variables, including how to set them using `ttenv` and discussion of the `PATH` and `CLASSPATH` environment variables.

Note: TimesTen includes Oracle Instant Client, which is required for certain JDBC features and operations.

Compiling Java applications

"Java environment variables" in *Oracle TimesTen In-Memory Database Installation, Migration, and Upgrade Guide* discusses the `CLASSPATH` setting for compiling Java applications in TimesTen.

Compiling any Java application requires the JAR file appropriate for your JDK to be in your classpath. In TimesTen, the following is for JDK 8:

`timesten_home``/install/lib/ttjdbc8.jar`

In addition, compiling any JMS/XLA application requires the following to be in your classpath:

`timesten_home``/install/lib/timestenjmsxla.jar`
`timesten_home``/install/3rdparty/jms1.1/lib/jms.jar`
`timesten_home``/install/lib/orai18n.jar`

Use the appropriate `ttenv` script to set your environment, as discussed in the preceding section.

Notes:

- For each TimesTen instance, the *`timesten_home`*`/install` path is a symbolic link to *`installation_dir`*, where TimesTen is installed.
- On Windows, there is only one TimesTen instance per installation, and *`timesten_home`* refers to *`installation_dir`*`\instance`.

TimesTen Quick Start and sample applications

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

After you have configured your environment, you can confirm that everything is set up correctly by compiling and running the sample applications. For TimesTen Classic, applications are located under the Quick Start `sample_code` directory. For instructions

on compiling and running them, see the instructions in the subdirectories. For TimesTen Scaleout, clone the `oracle-timesten-examples` GitHub repository and follow the instructions in the README files.

For TimesTen Classic, the following are included:

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

2

Working with TimesTen Databases in JDBC

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

Prerequisite task	What you do
Create a database.	For TimesTen Classic, follow the procedures described in "Managing TimesTen Databases" in *Oracle TimesTen In-Memory Database Operations Guide*. For TimesTen Scaleout, refer to "Creating a database" in *Oracle TimesTen In-Memory Database Scaleout User's Guide*.
Configure the Java environment.	Follow the procedures described in "Setting the environment for Java development" on page 1-2.
Compile and execute the TimesTen Java sample applications.	Refer to "TimesTen Quick Start and sample applications" on page 1-2.

After you have successfully executed the TimesTen Java sample applications, your development environment is set up correctly and ready for you to create applications that access a database.

The following topics are covered in this chapter:

Key JDBC classes and interfaces

This section discusses important standard and TimesTen-specific JDBC packages, classes, and interfaces. The following topics are covered:

- Support for interfaces in the java.sql package
- Support for classes in the java.sql package
- Support for interfaces and classes in the javax.sql package
- TimesTen JDBC extensions
- Additional TimesTen classes and interfaces

For reference information on standard JDBC, see the following for information about the `java.sql` and `javax.sql` packages:

https://docs.oracle.com/javase/8/docs/api/java/sql/package-summary.html
https://docs.oracle.com/javase/8/docs/api/javax/sql/package-summary.html

For reference information on TimesTen JDBC extensions, refer to *Oracle TimesTen In-Memory Database JDBC Extensions Java API Reference*.

Note: TimesTen supports Java 8 APIs (JDBC 4.2).

Package imports

Import the standard JDBC package in any Java program that uses JDBC:

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

If you are going to use data sources or pooled connections, also import the standard extended JDBC package:

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

Import the TimesTen JDBC package:

```
import com.timesten.jdbc.*;
```

To use XA data sources for JTA, also import this TimesTen package:

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

Support for interfaces in the java.sql package

TimesTen supports the `java.sql` interfaces as indicated in Table 2–1, with TimesTen-specific support and restrictions noted.

Also see "TimesTen JDBC extensions" on page 2-5.

Table 2–1 Supported java.sql interfaces

Interface in java.sql	Remarks on TimesTen support
`Blob`	▪ The `position()` method, which returns the position where a specified byte pattern or BLOB pattern begins, is not supported.
`CallableStatement`	▪ You must pass parameters to `CallableStatement` by position, not by name. ▪ You cannot use SQL escape syntax. ▪ There is no support for `Array`, `Struct`, or `Ref`. ▪ There is no support for `Calendar` for `setDate()`, `getDate()`, `setTime()`, or `getTime()`.

Table 2–1 (Cont.) Supported java.sql interfaces

Interface in java.sql	Remarks on TimesTen support
`Clob`	■ The `position()` method, which returns the position where a specified character pattern or CLOB pattern begins, is not supported.

`Connection`	■ There is no support for savepoints.
	■ TimesTen supports Read Committed and Serializable isolation levels:
	`setTransactionIsolation(TRANSACTION_READ_COMMITTED);`
	`setTransactionIsolation(TRANSACTION_SERIALIZABLE);`
	See "Fetching multiple rows of data" on page 2-13 for information about the relationship between prefetching and isolation level. Also see "Concurrency control through isolation and locking" in *Oracle TimesTen In-Memory Database Operations Guide* and "Isolation" in *Oracle TimesTen In-Memory Database Reference*.

`DatabaseMetaData`	■ There are no restrictions. ■ The `supportsRefCursors()` method returns `TRUE`. ■ The `getMaxLogicalLobSize()` method returns the maximum number of bytes that TimesTen allows for the logical size of a LOB.
`Driver`	■ TimesTen does not use `java.util.logging`, so the `getParentLogger()` method returns `SQLFeatureNotSupportedException`. (The TimesTen driver preceded `java.util.logging` functionality and uses its own logging mechanism.)
`NClob`	■ The `position()` method, which returns the position where a specified character pattern or NCLOB pattern begins, is not supported.
`ParameterMetaData`	■ The JDBC driver cannot determine whether a column is nullable and always returns `parameterNullableUnknown` from calls to `isNullable()`. ■ The `getScale()` method returns 1 for `VARCHAR`, `NVARCHAR`, and `VARBINARY` data types if they are `INLINE`. (Scale is of no significance to these data types.)
`PreparedStatement`	■ There is no support for `getMetaData()` in `PreparedStatement`. ■ There is no support for `Array`, `Struct`, or `Ref`. ■ Settings using `setObject(java.util.Calendar)` and `setDate(java.util.Date)` are mapped to `TIMESTAMP`. There is no support for the `Calendar` type in `setDate()`, `getDate()`, `setTime()`, or `getTime()`.
`ResultSet`	■ There *is* support for `getMetaData()` in `ResultSet`. ■ 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. ■ There is no support for scrollable or updatable result sets. ■ There is no support for `Array`, `Struct`, or `Ref`. ■ There is no support for the `Calendar` type in `setDate()`, `getDate()`, `setTime()`, or `getTime()`. ■ See "Working with TimesTen result sets: hints and restrictions" on page 2-12.

Table 2–1 (Cont.) Supported java.sql interfaces

Interface in java.sql	Remarks on TimesTen support
`ResultSetMetaData`	▪ The `getPrecision()` method returns 0 for undefined precision. ▪ The `getScale()` method returns -127 for undefined scale.
`RowId`	▪ The `ROWID` data type can be accessed using the `RowId` interface. ▪ Output and input/output rowids can be registered as `Types.ROWID`. ▪ Metadata methods return `Types.ROWID` and `RowId` as applicable.
`Statement`	▪ TimesTen does not support auto-generated keys. ▪ In TimesTen, the `cancel()` method delegates to the ODBC function `SQLCancel`. For details about the TimesTen implementation of this function, see "ODBC 2.5 function support" in *Oracle TimesTen In-Memory Database C Developer's Guide*. The function is also supported for ODBC 3.5. ▪ See "Managing cache groups" on page 2-40 for special TimesTen functionality of the `getUpdateCount()` method with cache groups.
`Wrapper`	▪ TimesTen exposes `TimesTenConnection`, `TimesTenCallableStatement`, `TimesTenPreparedStatement`, and `TimesTenStatement` through `Wrapper`.

Support for classes in the java.io package

An `InputStream` object returned by TimesTen does not support mark or reset operations (specifically, the `mark()`, `markSupported()`, and `reset()` methods).

Support for classes in the java.sql package

TimesTen supports these `java.sql` classes (and some additional subclasses of `SQLException`):

Table 2–2 Supported java.sql classes

Interface in java.sql	Remarks on TimesTen support
`DataTruncation`	No remarks
`Date`	No remarks
`DriverManager`	No remarks
`DriverPropertyInfo`	No remarks
`Time`	Because TimesTen does not support `TIMEZONE` in the `TIME` data type, Java client/server applications should run in the same time zone as the server.
`Timestamp`	Same consideration for `TIMESTAMP` as for `TIME`.
`Types`	No remarks
`SQLException`	No remarks
`SQLFeatureNotSupportedException`	No remarks
`SQLWarning`	No remarks

Support for interfaces and classes in the javax.sql package

TimesTen supports these `javax.sql` interfaces:

- `CommonDataSource` and `DataSource` are implemented by `TimesTenDataSource`.

  TimesTen does not use `java.util.logging`, so the `getParentLogger()` method, specified in `CommonDataSource`, returns `SQLFeatureNotSupportedException`. (The TimesTen driver preceded `java.util.logging` functionality and uses its own logging mechanism.)

- `PooledConnection` is implemented by `ObservableConnection`.
- `ConnectionPoolDataSource` is implemented by `ObservableConnectionDS`.
- `XADataSource` is implemented by `TimesTenXADataSource` (in package `com.timesten.jdbc.xa`).

> **Important:** The TimesTen JDBC driver itself does not implement a database connection pool. The `ObservableConnection` and `ObservableConnectionDS` classes simply implement standard Java EE interfaces, facilitating the creation and management of database connection pools according to the Java EE standard.

TimesTen supports this `javax.sql` event listener:

- When using a `PooledConnection` instance, you can register a `ConnectionEventListener` instance to listen for `ConnectionEvent` occurrences.

> **Note:** You can register a `StatementEventListener` instance in TimesTen; however, `StatementEvent` instances are not supported.

TimesTen JDBC extensions

For most scenarios, you can use standard JDBC functionality as supported by TimesTen.

TimesTen also provides extensions in the `com.timesten.jdbc` package for TimesTen-specific features, as shown in Table 2–3.

Table 2–3 TimesTen JDBC extensions

Interface	Extends	Remarks
`TimesTenBlob`	`Blob`	You can cast `Blob` instances to `TimesTenBlob`. This includes features to indicate whether a LOB is an Oracle Database passthrough LOB, free LOB resources, and get a binary stream with position and length specifications. See "Working with LOBs" on page 2-29.

Table 2–3 (Cont.) TimesTen JDBC extensions

Interface	Extends	Remarks
`TimesTenCallableStatement`	`CallableStatement`	Exposed through `java.sql.Wrapper`. Supports PL/SQL REF CURSORs. See "Working with REF CURSORs" on page 2-25. Supports associative array binds with methods to set input parameters and to register and get output parameters. See "Binding associative arrays" on page 2-21.
`TimesTenClob`	`Clob`	You can cast `Clob` instances to `TimesTenClob`. This includes features to indicate whether a LOB is an Oracle Database passthrough LOB, free LOB resources, and get a character stream with position and length specifications. See "Working with LOBs" on page 2-29.
`TimesTenConnection`	`Connection`	Exposed through `java.sql.Wrapper`. Provides capabilities such as prefetching rows to improve performance, optimizing query performance, listening to events for automatic client failover, setting the track number for parallel replication schemes where you specify replication tracks, and checking database validity. See "Fetching multiple rows of data" on page 2-13, "Optimizing query performance" on page 2-14, "General Client Failover Features" on page 2-48, "Features for use with replication" on page 2-40, and "Check database validity" on page 2-10.
`TimesTenNClob`	`NClob`	You can cast `NClob` instances to `TimesTenNClob`. This includes features to indicate whether a LOB is an Oracle Database passthrough LOB. See "Working with LOBs" on page 2-29.

Table 2–3 (Cont.) TimesTen JDBC extensions

Interface	Extends	Remarks
`TimesTenPreparedStatement`	`PreparedStatement`	Exposed through `java.sql.Wrapper`. Supports DML returning. See "Working with DML returning (RETURNING INTO clause)" on page 2-27. Supports associative array binds with a method to set input parameters. See "Binding associative arrays" on page 2-21.
`TimesTenStatement`	`Statement`	Exposed through `java.sql.Wrapper`. Provides capabilities for specifying a query threshold. See "Setting a threshold duration for SQL statements" on page 2-38.

Additional TimesTen classes and interfaces

In addition to implementations discussed previously, TimesTen provides interfaces and classes in the `com.timesten.jdbc` package. Features supported by these interfaces and classes are discussed later in this chapter.

Additional TimesTen interfaces

- Use `TimesTenTypes` for TimesTen type extensions (such as for TimesTen REF CURSORs).
- Use `ClientFailoverEventListener` (and also the `ClientFailoverEvent` class below) for automatic client failover features. See "JDBC support for automatic client failover" on page 2-47.
- Use `TimesTenVendorCode` for vendor codes used in SQL exceptions.
- Use `TimesTenDistributionKey` and `TimesTenDistributionKeyBuilder` for client routing in TimesTen Scaleout. See "Client routing in TimesTen Scaleout" on page 2-55.
- Use `TimesTenConnectionBuilder` to connect to an optimal element based on a distribution key, element ID, or replica set ID. See "Connecting to an element based on a distribution key" on page 2-57.

Additional TimesTen classes

- Use `ClientFailoverEvent` (and also the `ClientFailoverEventListener` interface above) for automatic client failover features.

Managing TimesTen database connections

The type of DSN you create depends on whether your application connects directly to the database or connects by a client.

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

For TimesTen Classic, if you intend to connect directly to the database, create a DSN as described in "Creating a DSN on Linux and UNIX for TimesTen Classic" in *Oracle TimesTen In-Memory Database Operations Guide*. If you intend to create a client connection to the database, create a DSN as described in "Creating and configuring Client DSNs on Windows" or "Creating and configuring Client DSNs on Linux and UNIX" in *Oracle TimesTen In-Memory Database Operations Guide*.

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

The operations described in this section are based on the `level1` sample application. Refer to "TimesTen Quick Start and sample applications" on page 1-2.

Note: TimesTen exposes TimesTen-specific implementations through standard `java.sql.Wrapper` functionality. You can use `Wrapper` to retrieve connection objects that implement the `TimesTenConnection` interface and provide access to TimesTen-specific features. The following example returns a `TimesTenConnection` object then calls its TimesTen extension `setReplicationTrack()` method.

```
String databaseUrl = null;
...
Connection conn = DriverManager.getConnection(databaseUrl);
If (conn.isWrapperFor(TimesTenConnection.class) ) {
  TimesTenConnection tconn = conn.unwrap(TimesTenConnection.class);
  tconn.setReplicationTrack(4);
}
...
```

This following topics are covered:

- Create a connection URL for the database and specify connection attributes
- Connect to the database
- Disconnect from the database
- Open and close a direct connection

Create a connection URL for the database and specify connection attributes

To create a JDBC connection, specify a TimesTen connection URL for the database. The format of a TimesTen connection URL is:

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

The default is `direct`.

For example, the following creates a direct connection to the sample database:

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

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

Refer to "Connection attributes for Data Manager DSNs or Server DSNs" in *Oracle TimesTen In-Memory Database Operations Guide* for introductory information about connection attributes. General connection attributes require no special privilege. First connection attributes are set when the database is first loaded, and persist for all

connections. Only the instance administrator can load a database with changes to first connection attribute settings. Refer to "Connection Attributes" in *Oracle TimesTen In-Memory Database Reference* for specific information about any particular connection attribute, including required privilege.

For example, to set the `LockLevel` general connection attribute to 1, create a URL as follows:

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

Connect to the database

After you have defined a URL, you can use the `getConnection()` method of either `DriverManager` or `TimesTenDataSource` to connect to the database.

If you use the `DriverManager.getConnection()` method, specify the driver URL to connect to the database.

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

To use the `TimesTenDataSource` method `getConnection()`, first create a data source. Then use the `TimesTenDataSource` method `setUrl()` to set the URL and `getConnection()` to connect:

```
import com.timesten.jdbc.TimesTenDataSource;
import java.sql.*;
...

TimesTenDataSource ds = new TimesTenDataSource();
ds.setUrl("jdbc:timesten:direct:<dsn>");
Connection conn = ds.getConnection();
```

The TimesTen user name and password can be set in the DSN within the URL in the `setUrl()` call, but there are also `TimesTenDataSource` methods to set them separately, as well as to set the Oracle Database password (as applicable):

```
TimesTenDataSource ds = new TimesTenDataSource();
ds.setUser(myttusername);                    // User name to log in to TimesTen
ds.setPassword(ttpassword);                  // Password to log in to TimesTen
ds.setUrl("jdbc:timesten:direct:<dsn>");
ds.setOraclePassword(oraclepassword);        // Password to log in to Oracle DB
Connection conn = ds.getConnection();
```

Either the `DriverManager.getConnection()` method or the `ds.getConnection()` method returns a `Connection` object (`conn` in this example) that you can use as a handle to the database. See the `level1` sample application for an example on how to use the `DriverManager` method `getConnection()`, and the `level2` and `level3` sample applications for examples of using the `TimesTenDataSource` method `getConnection()`. Refer to "TimesTen Quick Start and sample applications" on page 1-2.

Disconnect from the database

When you are finished accessing the database, typically call the `Connection` method `close()` to close the connection to the database.

TimesTen connections also support the standard `abort()` method, as well as standard try-with-resource functionality using `java.lang.AutoCloseable`.

If an error has occurred, you may want to roll back the transaction before disconnecting from the database. See "Handling non-fatal errors" on page 2-41 and "Rolling back failed transactions" on page 2-44 for more information.

Open and close a direct connection

Example 2–1 shows the general framework for an application that uses the `DriverManager` class to create a direct connection to the sample database, execute some SQL, and then close the connection. See the `level1` sample application for a working example. (See "TimesTen Quick Start and sample applications" on page 1-2 regarding the sample applications.)

Example 2–1 Connecting, executing SQL, and disconnecting

```
String URL = "jdbc:timesten:dsn=sampledb";
Connection conn = null;

try {
     Class.forName("com.timesten.jdbc.TimesTenDriver");
} catch (ClassNotFoundException ex) {
      // See "Handling errors" on page 2-41
}

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

    // Report any SQLWarnings on the connection
    // See "Reporting errors and warnings" on page 2-42

    // Do SQL operations
    // See "Managing TimesTen data" below

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

// Handle any errors
} catch (SQLException ex) {
    // See "Handling errors" on page 2-41
}
```

Check database validity

Applications can call the following `TimesTenConnection` method to detect whether the database is valid:

```
boolean isDataStoreValid() throws java.sql.SQLException
```

It returns `true` if the database is valid, or `false` if the database is in an invalid state, such as due to system or application failure.

Managing TimesTen data

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

Note: TimesTen exposes TimesTen-specific implementations through standard `java.sql.Wrapper` functionality. You can use `Wrapper` to retrieve statement objects that implement the `TimesTenStatement`, `TimesTenPreparedStatement`, and `TimesTenCallableStatement` interfaces and provide access to TimesTen-specific features. See "Managing TimesTen database connections" on page 2-7 for similar discussion and an example regarding connection objects.

The following topics are discussed here:

- Executing simple SQL statements
- Working with TimesTen result sets: hints and restrictions
- Fetching multiple rows of data
- Optimizing query performance
- Binding parameters and executing statements
- Working with REF CURSORs
- Working with DML returning (RETURNING INTO clause)
- Working with rowids
- Working with LOBs
- Committing or rolling back changes to the database
- Managing multiple threads
- Java escape syntax and SQL functions

Executing simple SQL statements

"Working with Data in a TimesTen Database" in *Oracle TimesTen In-Memory Database Operations Guide* describes how to use SQL to manage data. This section describes how to use the `createStatement()` method of a `Connection` instance, and the `executeUpdate()` or `executeQuery()` method of a `Statement` instance, to execute a SQL statement within a Java application.

Unless statements are prepared in advance, use the execution methods of a `Statement` object, such as `execute()`, `executeUpdate()` or `executeQuery()`, depending on the nature of the SQL statement and any returned result set.

For SQL statements that are prepared in advance, use the same execution methods of a `PreparedStatement` object.

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` statement).

Note: See "Working with TimesTen result sets: hints and restrictions" on page 2-12 for details about what you should know when working with result sets generated by TimesTen.

Example 2–2 Executing an update

This example uses the `executeUpdate()` method on the `Statement` object to execute an `INSERT` statement to insert data into the `customer` table in the current schema. The connection must have been opened, which is not shown.

```
Connection conn = null;
Statement stmt = null;
...
// [Code to open connection. See "Connect to the database" on page 2-9...]
...
try {
    stmt = conn.createStatement();
    int numRows = stmt.executeUpdate("insert into customer values"
                  + "(40, 'West', 'Big Dish', '123 Signal St.')");
}
catch (SQLException ex) {
    ...
}
```

Example 2–3 Executing a query

This example uses an `executeQuery()` call on the `Statement` object to execute a `SELECT` statement on the `customer` table in the current schema and display the returned `java.sql.ResultSet` instance:

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

Working with TimesTen result sets: hints and restrictions

Use `ResultSet` objects to process query results. In addition, some methods and built-in procedures return TimesTen data in the form of a `ResultSet` object. This section describes what you should know when using `ResultSet` objects from TimesTen.

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

- 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, essentially 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`.
- Typically, use the `ResultSet` method `close()` to close a result set as soon as you are done with it. For performance reasons, this is especially important for result sets used for both read and update operations and for result sets used in pooled connections.

  TimesTen result sets also support standard try-with-resource functionality using `java.lang.AutoCloseable`.
- Calling the `ResultSet` method `getString()` is more costly in terms of performance if the underlying data type is not a string. Because Java strings are immutable, `getString()` must allocate space for a new string each time it is called. Do not use `getString()` to retrieve primitive numeric types, like `byte` or `int`, unless it is absolutely necessary. For example, it is much faster to call `getInt()` on an integer column. Also see "Use the ResultSet method getString() sparingly" on page 5-4.

  In addition, for dates and timestamps, the `ResultSet` native methods `getDate()` and `getTimestamp()` have better performance than `getString()`.
- Application performance is affected by the choice of `get`*`XXX`*`()` calls and by any required data transformations after invocation.
- JDBC ignores the setting for the `ConnectionCharacterSet` attribute. It returns data in UTF-16 encoding.

Fetching multiple rows of data

Fetching multiple rows of data can increase the performance of a client/server application that connects to a database set with Read Committed isolation level.

You can specify the number of rows to be prefetched as follows.

- Call the `Statement` or `ResultSet` method `setFetchSize()`. These are the standard JDBC calls, but the limitation is that they only affect one statement at a time.
- Call the `TimesTenConnection` method `setTtPrefetchCount()`. This enables a TimesTen extension that establishes prefetch at the 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:** The TimesTen prefetch count extension provides no benefit for an application using a direct connection to the database.

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

To disable prefetch, set the prefetch count to 1.

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

See the Connection interface entry in "Support for interfaces in the java.sql package" on page 2-2 for information about setting transaction isolation level. Refer to *Oracle TimesTen In-Memory Database JDBC Extensions Java API Reference* for additional information.

Example 2–4 Setting a prefetch count

The following code uses a setTtPrefetchCount() call to set the prefetch count to 10, then uses a getTtPrefetchCount() call to return the prefetch count in the count variable.

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

// set prefetch count to 10 for this connection
conn.setTtPrefetchCount(10);

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

Optimizing query performance

A TimesTen extension enables applications to optimize read-only query performance in client/server applications by calling the TimesTenConnection method setTtPrefetchClose() with a setting of true.

All transactions should be committed when executed, including read-only transactions. With a setTtPrefetchClose(true) call, the server automatically closes the cursor and commits the transaction after the server has prefetched all rows of the result set for a read-only query. This enhances performance by reducing the number of network round-trips between client and server.

The client should still close the result set and commit the transaction, but those calls are executed in the client and do not require a network round trip between the client and server.

Notes:

- Do not use multiple statement handles for the same connection with a setTtPrefetchClose(true) call. The server may fetch all rows from the result set, commit the transaction, and close the statement handle before the client is finished, resulting in the closing of all statement handles.
- A true setting is ignored for TimesTen direct connections and for SELECT FOR UPDATE statements.
- Use getTtPrefetchClose() to get the current setting (true or false).

The following example shows usage of setTtPrefetchClose(true).

```
import com.timesten.sql;
...
con = DriverManager.getConnection ("jdbc:timesten:client:" + DSN);
stmt = con.createStatement();
```

```
...
con.setTtPrefetchClose(true);
rs = stmt.executeQuery("select * from t");
while(rs.next())
{
// do the processing
}
rs.close();
con.commit();
```

Binding parameters and executing statements

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

- Preparing SQL statements and setting input parameters
- Working with output and input/output parameters
- Binding duplicate parameters in SQL statements
- Binding duplicate parameters in PL/SQL
- Binding associative arrays

Notes:

- Typically, use the `Statement`, `PreparedStatement`, or `CallableStatement` method `close()` to close a statement you have finished using it. TimesTen statements also support standard try-with-resource functionality using `java.lang.AutoCloseable`.
- The term "bind parameter" as used in TimesTen developer guides (in keeping with ODBC terminology) is equivalent to the term "bind variable" as used in TimesTen PL/SQL documents (in keeping with Oracle Database PL/SQL terminology).

Preparing SQL statements and setting input parameters

SQL statements that are to be executed more than once should be prepared in advance by calling the `Connection` method `prepareStatement()`. For maximum performance, prepare parameterized statements.

Be aware of the following:

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

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

  You could address the issue as follows, for example.

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

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

- Application performance is influenced by the choice of `setXXX()` calls and by any required data transformations before invocation. For example, for time, dates, and timestamps, the `PreparedStatement` native methods `setTime()`, `setDate()` and `setTimestamp()` have better performance than `setString()`.
- For `TT_TINYINT` columns, use `setShort()` or `setInt()` instead of `setByte()` to use the full range of `TT_TINYINT` (0-255).
- Settings using `setObject(java.util.Calendar)` and `setDate(java.util.Date)` are mapped to `TIMESTAMP`.

Example 2–5 Prepared statement for querying

This example shows the basics of an `executeQuery()` call on a `PreparedStatement` object. It executes a prepared `SELECT` statement and displays the returned result set.

```
PreparedStatement pSel = conn.prepareStatement("select cust_num, " +
                         "region, name, address " +
                         "from customer" +
                         "where region = ?");
pSel.setInt(1,1);

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();
}
```

Example 2–6 Prepared statement for updating

This example shows how a single parameterized statement can be substituted for four separate statements.

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` methods `setXXX()` to set the row values before each execute.

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

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

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

pIns.setInt(1, 5);
```

```
pIns.setInt(2, 6);
pIns.executeUpdate();

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

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

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

Example 2–7 Prepared statements for updating and querying

This example prepares `INSERT` and `SELECT` statements, executes the `INSERT` twice, executes the `SELECT`, and prints the returned result set. For a working example, see the `level1` sample application. (Refer to "TimesTen Quick Start and sample applications" on page 1-2 regarding the sample applications.)

```
Connection conn = null;
...
// [Code to open connection. See "Connect to the database" on page 2-9...]
...

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

    // Report any SQLWarnings on the connection
    // See "Reporting errors and warnings" on page 2-42

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

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

// Data for first INSERT statement
pIns.setInt(1, 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, 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
```

```
conn.commit();

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

// Report any SQLWarnings on the connection.
reportSQLWarnings(conn.getWarnings());

// 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 must be committed too
conn.commit();

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

Example 2–8 Prepared statements for multiple connections

This example prepares three identical parameterized `INSERT` statements for three separate connections. The first prepared `INSERT` for connection `conn1` is shared (inside the TimesTen internal prepared statement cache) with the `conn2` and `conn3` connections, speeding up the prepare operations for `pIns2` and `pIns3`:

```
Connection conn1 = null;
Connection conn2 = null;
Connection conn3 = null;
.....
PreparedStatement pIns1 = conn1.prepareStatement
                  ("insert into t1 values (?,?)");

PreparedStatement pIns2 = conn2.prepareStatement
                  ("insert into t1 values (?,?)");

PreparedStatement pIns3 = conn3.prepareStatement
                  ("insert into t1 values (?,?)");
```

> **Note:** All optimizer hints, such as join ordering, indexes and locks, must match for the statement to be shared in the internal TimesTen prepared statement cache. Also, if the prepared statement references a temp table, it is only shared within a single connection.

Working with output and input/output parameters

"Preparing SQL statements and setting input parameters" on page 2-15 shows how to prepare a statement and set input parameters using `PreparedStatement` methods. TimesTen also supports output and input/output parameters, for which you use `java.sql.CallableStatement` instead of `PreparedStatement`, as follows.

1. Use the method `registerOutParameter()` to register an output or input/output parameter, specifying the parameter position (position in the statement) and data type.

   This is the standard method as specified in the `CallableStatement` interface:

   ```
   void registerOutParameter(int parameterIndex, int sqlType, int scale)
   ```

   Be aware, however, that if you use this standard version for `CHAR`, `VARCHAR`, `NCHAR`, `NVARCHAR`, `BINARY`, or `VARBINARY` data, TimesTen allocates memory to hold the largest possible value. In many cases this is wasteful.

   Instead, you can use the TimesTen extended interface `TimesTenCallableStatement`, which has a `registerOutParameter()` signature that enables you to specify the maximum data length. For `CHAR`, `VARCHAR`, `NCHAR`, and `NVARCHAR`, the unit of length is number of characters. For `BINARY` and `VARBINARY`, it is bytes.

   ```
   void registerOutParameter(int paramIndex,
                             int sqlType,
                             int ignore, //This parameter is ignored by TimesTen.
                             int maxLength)
   ```

2. Use the appropriate `CallableStatement` method `set`*`XXX`*`()`, where *`XXX`* indicates the data type, to set the input value of an input/output parameter. Specify the parameter position and data value.
3. Use the appropriate `CallableStatement` method `get`*`XXX`*`()` to get the output value of an output or input/output parameter, specifying the parameter position.

Important: Check for SQL warnings before processing output parameters. In the event of a warning, output parameters are undefined. See "Handling errors" on page 2-41 for general information about errors and warnings.

Notes: In TimesTen:

- You cannot pass parameters to a `CallableStatement` object by name. You must set parameters by position. You cannot use the SQL escape syntax.
- The `registerOutParameter()` signatures specifying the parameter by name are not supported. You must specify the parameter by position.
- SQL structured types are not supported.

Example 2–9 Using an output parameter in a callable statement

This example shows how to use a callable statement with an output parameter. In the `TimesTenCallableStatement` instance, a PL/SQL block calls a function `RAISE_SALARY` that calculates a new salary and returns it as an integer. Assume a `Connection` instance `conn`. (Refer to *Oracle TimesTen In-Memory Database PL/SQL Developer's Guide* for information about PL/SQL.)

```
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.Types;
import com.timesten.jdbc.TimesTenCallableStatement;
```

```
...
// Prepare to call a PL/SQL stored procedure RAISE_SALARY
CallableStatement cstmt = conn.prepareCall
                          ("BEGIN :newSalary := RAISE_SALARY(:name, :inc); end;");

// Declare that the first param (newSalary) is a return (output) value of type int
cstmt.registerOutParameter(1, Types.INTEGER);

// Raise Leslie's salary by $2000 (she wanted $3000 but we held firm)
cstmt.setString(2, "LESLIE"); // name argument (type String) is the second param
cstmt.setInt(3, 2000); // raise argument (type int) is the third param

// Do the raise
cstmt.execute();

// Check warnings. If there are warnings, output parameter values are undefined.
SQLWarning wn;
boolean warningFlag = false;
if ((wn = cstmt.getWarnings() ) != null) {
   do {
        warningFlag = true;
        System.out.println(wn);
        wn = wn.getNextWarning();
   } while(wn != null);
}

// Get the new salary back
if (!warningFlag) {
   int new_salary = cstmt.getInt(1);
   System.out.println("The new salary is: " + new_salary);
}

// Close the statement and connection
cstmt.close();
conn.close();
...
```

Binding duplicate parameters in SQL statements

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

Notes:

- This discussion applies only to SQL statements issued directly from ODBC, not through PL/SQL, for example.
- "TimesTen mode" for binding duplicate parameters, and the `DuplicateBindMode` connection attribute, are deprecated.

Consider this query:

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

When parameter position numbers are assigned, a number is given to each parameter occurrence without regard to name duplication. The application must, at a minimum,

bind a value for the first occurrence of each parameter name. For any subsequent occurrence of a given parameter name, the application has the following choices.

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

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

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

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

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

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

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

Binding duplicate parameters in PL/SQL

The preceding discussion does not apply to PL/SQL, which has its own semantics. In PL/SQL, you bind a value for each unique parameter name. An application executing the following block, for example, would bind only one parameter, corresponding to `:a`.

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

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

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

And the same for the following `CALL` statement:

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

An application executing the following block would bind two parameters, with `:a` as parameter #1 and `:b` as parameter #2. The second parameter in each `INSERT` statement would take the same value as the first parameter in the first `INSERT` statement, as follows.

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

Binding associative arrays

TimesTen JDBC supports associative arrays, formerly known as index-by tables or PL/SQL tables, as `IN`, `OUT`, or `IN OUT` bind parameters to TimesTen PL/SQL. Associative arrays enable arrays of data to be passed efficiently between a JDBC application and the database.

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

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

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

Also see "Using associative arrays from applications" in *Oracle TimesTen In-Memory Database PL/SQL Developer's Guide*.

When you bind an associative array in Java, match the Java type as closely as possible with the array type for optimal performance. TimesTen does, however, support some simple input conversions:

- Strings can be converted to integers or floating point numbers.
- Strings can be converted to `DATE` data if the strings are in TimesTen `DATE` format (*`YYYY-MM-DD HH:MI:SS`*).

Notes: Note the following restrictions in TimesTen:

- The following types are not supported in binding associative arrays: LOBs, REF CURSORs, `TIMESTAMP`, `ROWID`.
- Associative array binding is not allowed in passthrough statements.
- General bulk binding of arrays is not supported in TimesTen JDBC. Varrays and nested tables are not supported as bind parameters.
- Associative array parameters are not supported with JDBC batch execution. (See "Use arrays of parameters for batch execution" on page 5-2.)

TimesTen provides extensions, described below, through the interfaces `TimesTenPreparedStatement` and `TimesTenCallableStatement` to support associative array binds. Refer to *Oracle TimesTen In-Memory Database JDBC Extensions Java API Reference* for additional information about any of the methods described here.

For an associative array that is a PL/SQL `IN` or `IN OUT` parameter, TimesTen provides the `setPlsqlIndexTable()` method in the `TimesTenPreparedStatement` interface (for an `IN` parameter) and in the `TimesTenCallableStatement` interface (for an `IN OUT` parameter) to set the input associative array.

- `void setPlsqlIndexTable(int` *`paramIndex`*`, java.lang.Object` *`arrayData`*`, int` *`maxLen`*`, int` *`curLen`*`, int` *`elemSqlType`*`, int` *`elemMaxLen`*`)`

  Specify the following:

  – *`paramIndex`*: Parameter position within the PL/SQL statement (starting with 1)

- *arrayData*: Array of values to be bound (which can be an array of primitive types such as `int[]` or an array of object types such as `BigDecimal[]`)
- *maxLen*: Maximum number of elements in the associative array (in TimesTen must be same as *curLen*)
- *curLen*: Actual current number of elements in the associative array (in TimesTen must be same as *maxLen*)
- *elemSqlType*: Type of the associative array elements according to `java.sql.Types` (such as `Types.DOUBLE`)
- *elemMaxLen*: For `CHAR`, `VARCHAR`, `BINARY`, or `VARBINARY` associative arrays, the maximum length of each element (in characters for `CHAR` or `VARCHAR` associative arrays, or in bytes for `BINARY` or `VARBINARY` associative arrays)

For example (assuming a `TimesTenPreparedStatement` instance `pstmt`):

```
int maxLen = 3;
int curLen = 3;
// Numeric field can be set with int, float, double types.
// elemMaxLen is set to 0 for numeric types and is ignored.
// elemMaxLen is specified for VARCHAR types.
pstmt.setPlsqlIndexTable
      (1, new int[]{4, 5, 6}, maxLen, curLen, Types.NUMERIC, 0);
pstmt.setPlsqlIndexTable
      (2, new String[]{"Batch1234567890", "2", "3"}, maxLen, curLen,
       Types.VARCHAR, 15);
pstmt.execute();
```

Notes:

- The *elemMaxLen* parameter is ignored for types other than `CHAR`, `VARCHAR`, `BINARY`, or `VARBINARY`. For any of those types, you can use a value of 0 to instruct the driver to set the maximum length of each element based on the actual length of data that is bound. If *elemMaxLen* is set to a positive value, then wherever the actual data length is greater than *elemMaxLen*, the data is truncated to a length of *elemMaxLen*.
- If *curLen* is smaller than the actual number of elements in the associative array, only *curLen* elements are bound.

For an associative array that is a PL/SQL `OUT` or `IN OUT` parameter, TimesTen provides two methods in the `TimesTenCallableStatement` interface: `registerIndexTableOutParameter()` to register an output associative array, and `getPlsqlIndexTable()` to retrieve an output associative array. There are two signatures for `getPlsqlIndexTable()`, one to use the JDBC default Java object type given the associative array element SQL type, and one to specify the type.

- `void registerIndexTableOutParameter(int` *paramIndex*`, int` *maxLen*`, int` *elemSqlType*`, int` *elemMaxLen*`)`

  Specify the following:

  - *paramIndex*: Parameter position within the PL/SQL statement (starting with 1)
  - *maxLen*: Maximum possible number of elements in the associative array
  - *elemSqlType*: Type of the associative array elements according to `java.sql.Types` (such as `Types.DOUBLE`)

  - *elemMaxLen*: For CHAR, VARCHAR, BINARY, or VARBINARY associative arrays, the maximum length of each element (in characters for CHAR or VARCHAR associative arrays, or in bytes for BINARY or VARBINARY associative arrays)

    Note: If *elemMaxLen* has a value of 0 or less, the maximum length for the data type is used.

- `java.lang.Object getPlsqlIndexTable(int paramIndex)`

  With this method signature, the type of the returned associative array is the JDBC default mapping for the SQL type of the data retrieved. Specify the parameter position within the PL/SQL statement (starting with 1). See Table 2–4 for the default mappings.

- `java.lang.Object getPlsqlIndexTable(int paramIndex, java.lang.Class primitiveType)`

  With this method signature, in addition to specifying the parameter position, specify the desired type of the returned associative array according to `java.sql.Types` (such as `Types.DOUBLE`). It must be a primitive type.

  Note: For TimesTen extensions for associative array binds, you must use instances of `TimesTenPreparedStatement` and `TimesTenCallableStatement` rather than `java.sql.PreparedStatement` and `CallableStatement`.

Table 2–4 JDBC default mappings for associative array elements

Return type	SQL type
Integer[]	TINYINT, SMALLINT, TT_INTEGER
Long[]	BIGINT
BigDecimal[]	NUMBER
Float[]	BINARY_FLOAT
Double[]	BINARY_DOUBLE
String[]	CHAR, VARCHAR, NCHAR, NVARCHAR
Timestamp[]	DATE

The following code fragment illustrates how to set, register, and retrieve the contents of an `IN OUT` parameter (assuming a connection `conn` and `TimesTenCallableStatement` instance `cstmt`):

```
int maxLen = 3;
int curLen = 3;
anonBlock = "begin AssocArrayEx_inoutproc(:o1); end;";
cstmt = (TimesTenCallableStatement) conn.prepareCall(anonBlock);
cstmt.setPlsqlIndexTable
     (1, new Integer[] {1,2,3}, maxLen, curLen, Types.NUMERIC, 0);
cstmt.registerIndexTableOutParameter(1, maxLen, Types.NUMERIC, 0);
cstmt.execute();

int[]  ret = (int [])cstmt.getPlsqlIndexTable(1, Integer.TYPE);
cstmt.execute();
```

Example 2–10 Binding an associative array

This is a more complete example showing the mechanism for binding an associative array.

```
TimesTenCallableStatement cstmt = null;
try {
  // Prepare procedure with associative array in parameter
  cstmt = (TimesTenCallableStatement)
           conn.prepareCall("begin AssociativeArray_proc(:name, :inc); end;");

  // Set up input array and length
  String[] name = {"George", "John", "Thomas", "James", "Bill"};
  Integer[] salaryInc = {10000, null, 5000, 8000, 9007};
  int currentLen = name.length;
  int maxLen = currentLen;


  // Use elemMaxLen for variable length data types such as
  // Types.VARCHAR, Types.CHAR.
  int elemMaxLen = 32;

  // set input parameter, name as a VARCHAR
  cstmt.setPlsqlIndexTable
        (1, name, maxLen, currentLen, Types.VARCHAR, elemMaxLen);
  // set input parameter, salaryInc as a number
  cstmt.setPlsqlIndexTable
        (2, salaryInc, maxLen, currentLen, Types.NUMERIC, 0);
```

Working with REF CURSORs

REF CURSOR is a PL/SQL concept, a handle to a cursor over a SQL result set that can be passed between PL/SQL and an application. In TimesTen, the cursor can be opened in PL/SQL, then the REF CURSOR can be passed to the application for processing of the result set.

TimesTen supports standard REF CURSORs as well as TimesTen REF CURSORs. (TimesTen-specific support was implemented prior to standard support.)

An application can receive a REF CURSOR `OUT` parameter as follows:

1. Using the `CallableStatement` method `registerOutParameter()`, register the REF CURSOR `OUT` parameter as type `java.sql.Types.REF_CURSOR` (for standard REF CURSORs) or as type `TimesTenTypes.CURSOR` (for TimesTen REF CURSORs). In the `registerOutParameter()` call, specify the parameter position of the REF CURSOR (position in the statement).
2. Retrieve the REF CURSOR using the `CallableStatement` method `getObject()` (for standard REF CURSORs) or the `TimesTenCallableStatement` method `getCursor()` (for TimesTen REF CURSORs), casting the return as `ResultSet`. Specify the parameter position of the REF CURSOR.

Refer to *Oracle TimesTen In-Memory Database JDBC Extensions Java API Reference* for additional information about TimesTen JDBC APIs. See "PL/SQL REF CURSORs" in *Oracle TimesTen In-Memory Database PL/SQL Developer's Guide* for additional information about REF CURSORs.

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

The following example demonstrates this usage.

Example 2–11 Using a REF CURSOR

This example shows how to use a callable statement with a TimesTen REF CURSOR, then with a standard REF CURSOR.

```
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.ResultSet;
import com.timesten.jdbc.TimesTenCallableStatement;
import com.timesten.jdbc.TimesTenTypes;
...
Connection conn = null;
CallableStatement cstmt = null;
ResultSet cursor;
...
// Use a PL/SQL block to open the cursor.
cstmt = conn.prepareCall
             (" begin open :x for select tblname,tblowner from tables; end;");
cstmt.registerOutParameter(1, TimesTenTypes.CURSOR);
cstmt.execute();
cursor = ((TimesTenCallableStatement)cstmt).getCursor(1);

// Use the cursor as you would any other ResultSet object.
while(cursor.next()){
  System.out.println(cursor.getString(1));
}

// Close the cursor, statement, and connection.
cursor.close();
cstmt.close();
conn.close();
...
```

For a standard REF CURSOR:

```
...
Connection conn = null;
CallableStatement cstmt = null;
ResultSet rs;
...
cstmt = conn.prepareCall
             (" begin open :x for select tblname,tblowner from tables; end;");
cstmt.registerOutParameter(1, Types.REF_CURSOR);
cstmt.execute();
rs = cstmt.getObject(1, ResultSet.class);
while(rs.next()){
  System.out.println(rs.getString(1));
}

// Close the result set, statement, and connection.
rs.close();
cstmt.close();
```

```
conn.close();
...
```

Note: If you are evaluating the callable statement with different parameter values in a loop, close the cursor each time at the end of the loop. The typical use case is to prepare the statement, then, in the loop, set parameters, execute the statement, process the cursor, and close the cursor.

Working with DML returning (RETURNING INTO clause)

You can use a RETURNING INTO clause, referred to as *DML returning*, with an `INSERT`, `UPDATE`, or `DELETE` statement to return specified items from a row that was affected by the action. This eliminates the need for a subsequent `SELECT` statement and separate round trip, in case, for example, you want to confirm what was affected by the action.

With TimesTen, DML returning is limited to returning items from a single-row operation. The clause returns the items into a list of output parameters.

`TimesTenPreparedStatement`, an extension of the standard `PreparedStatement` interface, supports DML returning. Use the `TimesTenPreparedStatement` method `registerReturnParameter()` to register the return parameters.

```
void registerReturnParameter(int paramIndex, int sqlType)
```

As with the `registerOutParameter()` method discussed in "Working with output and input/output parameters" on page 2-18, this method has a signature that enables you to optionally specify a maximum size for `CHAR`, `VARCHAR`, `NCHAR`, `NVARCHAR`, `BINARY`, or `VARBINARY` data. This avoids possible inefficiency where TimesTen would otherwise allocate memory to hold the largest possible value. For `CHAR`, `VARCHAR`, `NCHAR`, and `NVARCHAR`, the unit of size is number of characters. For `BINARY` and `VARBINARY`, it is bytes.

```
void registerReturnParameter(int paramIndex, int sqlType, int maxSize)
```

Use the `TimesTenPreparedStatement` method `getReturnResultSet()` to retrieve the return parameters, returning a `ResultSet` instance.

Be aware of the following restrictions when using `RETURNING INTO` in TimesTen JDBC.

- The `getReturnResultSet()` method must not be invoked more than once. Otherwise, the behavior is indeterminate.
- `ResultSetMetaData` is not supported for the result set returned by `getReturnResultSet()`.
- Streaming methods such as `getCharacterStream()` are not supported for the result set returned by `getReturnResultSet()`.
- There is no batch support for DML returning.

Refer to *Oracle TimesTen In-Memory Database JDBC Extensions Java API Reference* for additional information about the TimesTen JDBC classes, interfaces, and methods discussed here.

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

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

> **Important:** Check for SQL warnings after executing the TimesTen prepared statement. In the event of a warning, output parameters are undefined. See "Handling errors" on page 2-41 for general information about errors and warnings.

Example 2–12 DML returning

This example shows how to use DML returning with a `TimesTenPreparedStatement` instance, returning the name and age for a row that is inserted.

```
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.SQLWarning;
import java.sql.Types;
import com.timesten.jdbc.TimesTenPreparedStatement;

Connection conn = null;
...

// Insert into a table and return results
TimesTenPreparedStatement pstmt =
  (TimesTenPreparedStatement)conn.prepareStatement
  ("insert into tab1 values(?,?) returning name, age into ?,?");

// Populate table
pstmt.setString(1,"John Doe");
pstmt.setInt(2, 65);

/* register returned parameter
 * in this case the maximum size of name is 100 chars
 */
pstmt.registerReturnParameter(3, Types.VARCHAR, 100);
pstmt.registerReturnParameter(4, Types.INTEGER);

// process the DML returning statement
int count = pstmt.executeUpdate();

/* Check warnings; if there are warnings, values of DML RETURNING INTO
   parameters are undefined. */
SQLWarning wn;
boolean warningFlag = false;
if ((wn = pstmt.getWarnings() ) != null) {
  do {
    warningFlag = true;
    System.out.println(wn);
    wn = wn.getNextWarning();
  } while(wn != null);
}

if (!warningFlag) {
  if (count>0)
  {
    ResultSet rset = pstmt.getReturnResultSet(); //rset not null, not empty
    while(rset.next())
    {
      String name = rset.getString(1);
```

```
            int age = rset.getInt(2);
            System.out.println("Name " + name + " age " + age);
          }
        }
      }
```

Working with rowids

Each row in a table has a unique identifier known as its *rowid*. An application can retrieve the rowid of a row from the `ROWID` pseudocolumn. A rowid value can be represented in either binary or character format, with the binary format taking 12 bytes and the character format 18 bytes.

TimesTen supports the `java.sql.RowId` interface and `Types.ROWID` type.

You can use the following `ResultSet` methods to retrieve a rowid:

- `RowId getRowId(int columnIndex)`
- `RowId getRowId(String columnLabel)`

You can use the following `PreparedStatement` method to set a rowid:

- `setRowId(int parameterIndex, RowId x)`

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

Refer to "ROWID data type" and "ROWID pseudocolumn" in *Oracle TimesTen In-Memory Database SQL Reference* for additional information about rowids and the `ROWID` data type, including usage and lifecycle.

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

Working with LOBs

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

This section provides a brief overview of LOBs and discusses their use in JDBC, covering the following topics:

- About LOBs
- LOB objects in JDBC
- Differences between TimesTen LOBs and Oracle Database LOBs
- LOB factory methods
- LOB getter and setter methods
- TimesTen LOB interface methods
- LOB prefetching
- Passthrough LOBs

You can also refer to the following.

- "LOB data types" in *Oracle TimesTen In-Memory Database SQL Reference* for additional information about LOBs in TimesTen

- *Oracle Database SecureFiles and Large Objects Developer's Guide* for general information about programming with LOBs (but not specific to TimesTen functionality)

About LOBs

A LOB is a large binary object (BLOB) or character object (CLOB or NCLOB). In TimesTen, a BLOB can be up to 16 MB and a CLOB or NCLOB up to 4 MB. LOBs in TimesTen have essentially the same functionality as in Oracle Database, except as noted otherwise. (See "Differences between TimesTen LOBs and Oracle Database LOBs" on page 2-31.)

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

LOB objects in JDBC

In JDBC, a LOB object—`Blob`, `Clob`, or `NClob` instance—is implemented using a SQL LOB locator (`BLOB`, `CLOB`, or `NCLOB`), which means that a LOB object contains a logical pointer to the LOB data rather than the data itself.

Important:

- Because LOB objects do not remain valid past the end of the transaction in TimesTen, it is not feasible to use them with autocommit enabled. You would receive errors about LOBs being invalidated.
- LOB manipulations through APIs that use LOB locators result in usage of TimesTen temporary space. Any significant number of such manipulations may necessitate a size increase for the TimesTen temporary data region. See "TempSize" in *Oracle TimesTen In-Memory Database Reference*.

An application can use the JDBC API to instantiate a temporary LOB explicitly, for use within the application, then to free the LOB when done with it. Temporary LOBs are stored in the TimesTen temporary data region.

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

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

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

1. The LOB column contains "XY".

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

Differences between TimesTen LOBs and Oracle Database LOBs

Be aware of the following:

- A key difference between the TimesTen LOB implementation and the Oracle Database implementation is that in TimesTen, LOB objects do not remain valid past the end of the transaction. All LOB objects are invalidated after a commit or rollback, whether explicit or implicit. This includes after any autocommit (making it infeasible to use LOBs with autocommit enabled), or after any DDL statement.
- TimesTen does not support BFILEs, SecureFiles, reads and writes for arrays of LOBs, or callback functions for LOBs.
- TimesTen does not support binding associative arrays of LOBs.
- TimesTen does not support batch processing of LOBs.
- Relevant to BLOBs, there are differences in the usage of hexadecimal literals in TimesTen. see the description of *`HexadecimalLiteral`* in "Constants" in *Oracle TimesTen In-Memory Database SQL Reference*.

LOB factory methods

TimesTen supports the standard `Connection` methods `createBlob()`, `createClob()`, and `createNClob()`.

Refer to *Oracle TimesTen In-Memory Database JDBC Extensions Java API Reference* for additional information.

> **Important:** In TimesTen, creation of a LOB object results in creation of a database transaction if one is not already in progress. You must execute a commit or rollback to close the transaction.

LOB getter and setter methods

You can access LOBs through getter and setter methods that are defined by the standard `java.sql.ResultSet`, `PreparedStatement`, and `CallableStatement` interfaces, just as they are for other data types. Use the appropriate `get`*`XXX`*`()` method to retrieve a LOB result or output parameter or `set`*`XXX`*`()` method to bind a LOB input parameter:

- `ResultSet` getter methods: There are `getBlob()` methods, `getClob()` methods, and `getNClob()` methods where you can specify the LOB to retrieve according to either column name or column index.

  You can also use `getObject()` to retrieve a `Blob`, `Clob`, or `NClob` object.

- `PreparedStatement` setter methods: There is a `setBlob()` method, `setClob()` method, and `setNClob()` method where you can input the `Blob`, `Clob`, or `NClob` instance and the parameter index to bind an input parameter.

  You can also use `setObject()` to bind a `Blob`, `Clob`, or `NClob` input parameter.

  There are also `setBlob()` methods where instead of a `Blob` instance, you specify an `InputStream` instance, or an `InputStream` instance and length.

  There are `setClob()` and `setNClob()` methods where instead of a `Clob` or `NClob` instance, you specify a `Reader` instance, or a `Reader` instance and length.

- `CallableStatement` getter methods: There are `getBlob()` methods, `getClob()` methods, and `getNClob()` methods where you can retrieve the LOB output parameter according to either parameter name or parameter index.

  You can also use `getObject()` to retrieve a `Blob`, `Clob`, or `NClob` output parameter.

  You must also register an output parameter from a `CallableStatement` object. The `registerOutParameter()` method takes the parameter index along with the SQL type: `Types.BLOB`, `Types.CLOB`, or `Types.NCLOB`.

- `CallableStatement` setter methods: These are identical to (inherited from) `PreparedStatement` setter methods.

Refer to *Oracle TimesTen In-Memory Database JDBC Extensions Java API Reference* for additional information.

TimesTen LOB interface methods

You can cast a `Blob` instance to `com.timesten.jdbc.TimesTenBlob`, a `Clob` instance to `com.timesten.jdbc.TimesTenClob`, and an `NClob` instance to `com.timesten.jdbc.TimesTenNClob`. These interfaces support methods specified by the `java.sql.Blob`, `Clob`, and `NClob` interfaces.

The following list summarizes `Blob` features.

- The `isPassthrough()` method, a TimesTen extension, indicates whether the BLOB is a passthrough LOB from Oracle Database.
- Free `Blob` resources when the application is done with it.
- Retrieve the BLOB value as a binary stream. There are methods to retrieve it in whole or in part.
- Retrieve all or part of the BLOB value as a byte array.
- Return the number of bytes in the BLOB.
- Retrieve a stream to be used to write binary data to the BLOB, beginning at the specified position. This overwrites existing data.
- Specify an array of bytes to write to the BLOB, beginning at the specified position, and return the number of bytes written. This overwrites existing data. There are methods to write either all or part of the array.
- Truncate the BLOB to the specified length.

The following list summarizes `Clob` and `NClob` features.

- The `isPassthrough()` method, a TimesTen extension, indicates whether the CLOB or NCLOB is a passthrough LOB from Oracle Database.
- Free `Clob` or `NClob` resources when the application is done with it.
- Retrieve the CLOB or NCLOB as an ASCII stream.

- Retrieve the CLOB or NCLOB as a `java.io.Reader` object (or as a stream of characters). There are methods to retrieve it in whole or in part.
- Retrieve a copy of the specified substring in the CLOB or NCLOB, beginning at the specified position for up to the specified length.
- Return the number of characters in the CLOB or NCLOB.
- Retrieve a stream to be used to write ASCII characters to the CLOB or NCLOB, beginning at the specified position. This overwrites existing data.
- Specify a Java `String` value to write to the CLOB or NCLOB, beginning at the specified position. This overwrites existing data. There are methods to write either all or part of the `String` value.
- Truncate the CLOB or NCLOB to the specified length.

Notes:

- For methods that write data to a LOB, the size of the LOB does not change other than in the circumstance where from the specified position there is less space available in the LOB than there is data to write. In that case, the LOB size increases enough to accommodate the data.
- If the value specified for the position at which to write to a LOB is greater than LOB length + 1, the behavior is undefined.
- The `read()` method of an `InputStream` or `Reader` object returns 0 (zero) if the length of the buffer used in the method call is 0, regardless of the amount of data in the `InputStream` or `Reader` object. Therefore, usage such as the following is problematic if the CLOB length may be 0, such as if it were populated using the SQL `EMPTY_CLOB()` function:

  ```
  java.io.Reader r = myclob.getCharacterStream();
  char[] buf = new char[myclob.length()]; //buf for r.read() call
  ```

  Normally when you call `read()`, -1 is returned if the end of the stream is reached. But in the preceding case, -1 is never returned. Be aware of this when you use streams returned by the BLOB `getBinaryStream()` method, which returns `InputStream`, the CLOB `getAsciiStream()` method, which returns `InputStream`, or the CLOB `getCharacterStream()` method, which returns `Reader`.

Refer to *Oracle TimesTen In-Memory Database JDBC Extensions Java API Reference* for additional information.

LOB prefetching

To reduce round trips to the server in client/server connections, LOB prefetching is enabled by default when you fetch a LOB from the database. The default prefetch size is 4000 bytes for BLOBs or 4000 characters for CLOBs or NCLOBs.

You can use the `TimesTenConnection` property `CONNECTION_PROPERTY_DEFAULT_LOB_PREFETCH_SIZE` to set a different default value that applies to any statement in the connection. Use a value of -1 to disable LOB prefetching by default for the connection, 0 (zero) to enable LOB prefetching for only metadata by default, or any value greater than 0 to specify the number of bytes for BLOBs or characters for CLOBs and NCLOBs to be prefetched by default along with the LOB locator during fetch operations.

At the statement level, you can use the following `TimesTenStatement` methods to manipulate the prefetch size and override the default value from the connection:

- `setLobPrefetchSize(int)`: Set a new LOB prefetch value for the statement.
- `int getLobPrefetchSize()`: Return the current LOB prefetch value that applies to the statement (either a value set in the statement itself or the default value from the connection, as applicable).

Refer to *Oracle TimesTen In-Memory Database JDBC Extensions Java API Reference* for additional information.

Passthrough LOBs

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

- As noted in "TimesTen LOB interface methods" on page 2-32, the `TimesTenBlob`, `TimesTenClob`, and `TimesTenNClob` interfaces specify the following method to indicate whether the LOB is a passthrough LOB:

  ```
  boolean isPassthrough()
  ```

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

Refer to *Oracle TimesTen In-Memory Database JDBC Extensions Java API Reference* for additional information.

Committing or rolling back changes to the database

This section discusses autocommit and manual commits or rollbacks, assuming a JDBC `Connection` object `myconn` and `Statement` object `mystmt`.

> **Note:** All open cursors on the connection are closed upon transaction commit or rollback in TimesTen.

You can refer to "Transaction overview" in *Oracle TimesTen In-Memory Database Operations Guide* for additional information about transactions.

Setting autocommit

A TimesTen connection has autocommit enabled by default, but for performance reasons it is recommended that you disable it. You can use the `Connection` method `setAutoCommit()` to enable or disable autocommit.

Disable autocommit as follows:

```
myconn.setAutoCommit(false);
// Report any SQLWarnings on the connection.
// See "Reporting errors and warnings" on page 2-42.
```

Note: Beginning in TimesTen 12*c* releases, a `setAutoCommit()` call results in a commit only when the call actually changes the autocommit setting. In previous releases, any `setAutoCommit()` call resulted in a commit.

Manually committing or rolling back changes

If autocommit is disabled, you must use the `Connection` method `commit()` to manually commit transactions, or the `rollback()` method to roll back changes. Consider the following example.

```
myconn.commit();
```

Or:

```
myconn.rollback();
```

Using COMMIT and ROLLBACK SQL statements

You can prepare and execute `COMMIT` and `ROLLBACK` SQL statements the same way as other SQL statements. Using `COMMIT` and `ROLLBACK` statements has the same effect as using the `Connection` methods `commit()` and `rollback()`. For example:

```
mystmt.execute("COMMIT");
```

Managing multiple threads

The `level4` sample application demonstrates the use of multiple threads. Refer to "TimesTen Quick Start and sample applications" on page 1-2.

When your application has a direct connection to the database, TimesTen functions share stack space with your application. In multithreaded environments it is important to avoid overrunning the stack allocated to each thread, as this can cause a program to fail in unpredictable ways that are 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 between 34 KB and 72 KB.

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 database may encounter lock conflicts with itself. TimesTen returns lock timeout and deadlock errors in this situation.

Java escape syntax and SQL functions

When using SQL in JDBC, pay special attention 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;
```

Using additional TimesTen data management features

Preceding sections discussed key features for managing TimesTen data. This section covers the following additional features:

- Using CALL to execute procedures and functions
- Setting a timeout or threshold for executing SQL statements
- Features for use with TimesTen Cache
- Features for use with replication

Using CALL to execute procedures and functions

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

```
CALL procname[(argumentlist)]

CALL funcname[(argumentlist)] INTO :returnparam

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

TimesTen JDBC also supports each of the following syntax formats:

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

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

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

You can execute procedures and functions through the `CallableStatement` interface, with a prepare step first when appropriate (such as when a result set is returned).

The following example calls the TimesTen built-in procedure `ttCkpt`. (Also see Example 2–13 below for a more complete example with JDBC syntax.)

```
CallableStatement.execute("call ttCkpt")
```

The following example calls the TimesTen built-in procedure `ttDataStoreStatus`. A prepare call is used because this procedure produces a result set. (Also see Example 2–14 below for a more complete example with JDBC syntax.)

```
CallableStatement cStmt = null;
cStmt = conn.prepareCall("call ttDataStoreStatus");
cStmt.execute();
```

The following examples call a PL/SQL procedure `myproc` with two parameters.

```
cStmt.execute("{ call myproc(:param1, :param2) }");

cStmt.execute("{ call myproc(?, ?) }");
```

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

```
cStmt.execute("CALL myfunc() INTO :retparam");

cStmt.execute("CALL myfunc() INTO ?");
```

```
cStmt.execute("{ :retparam = myfunc() }");

cStmt.execute("{ ? = myfunc() }");
```

See "CALL" in *Oracle TimesTen In-Memory Database SQL Reference* for details about `CALL` syntax.

> **Note:** A user's own procedure takes precedence over a TimesTen built-in procedure with the same name, but it is best to avoid such naming conflicts.

Example 2–13 Executing a ttCkpt call

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

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

Be aware that the `ttCkpt` built-in procedure requires `ADMIN` privilege. Refer to "ttCkpt" in *Oracle TimesTen In-Memory Database Reference* for additional information.

Example 2–14 Executing a ttDataStoreStatus call

This example calls the `ttDataStoreStatus` procedure and prints out the returned result set.

For built-in procedures that return results, you can use the `get`*`XXX`*`()` methods of the `ResultSet` interface to retrieve the data, as shown.

Contrary to the advice given in "Working with TimesTen result sets: hints and restrictions" on page 2-12, this example uses a `getString()` call on the `ResultSet` object to retrieve the `Context` field, which is a binary. This is because the output is printed, rather than used for processing. If you do not want to print the `Context` value, you can achieve better performance by using the `getBytes()` method instead.

```
ResultSet rs;

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

if (cStmt.execute() == true) {
    rs = cStmt.getResultSet();
    System.out.println("Fetching result set...");
    while (rs.next()) {
      System.out.println("\n Database: " + 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();
```

Setting a timeout or threshold for executing SQL statements

TimesTen offers two ways to limit the time for SQL statements to execute, applying to any `execute()`, `executeBatch()`, `executeQuery()`, `executeUpdate()`, or `next()` call.

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

The former is to set a timeout, where if the timeout duration is reached, the statement stops executing and an error is thrown. The latter is to set a threshold, where if the threshold is reached, a warning is written to the support log but execution continues.

Setting a timeout duration for SQL statements

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

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

For a particular statement, you can override the `SQLQueryTimeout` setting by calling the `Statement` method `setQueryTimeout()`.

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

See "Choose SQL and PL/SQL timeout values" in *Oracle TimesTen In-Memory Database Operations Guide* for considerations regarding the SQL query timeout with respect to other timeout settings.

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

Setting a threshold duration for SQL statements

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

Despite the name, this threshold applies to any JDBC call executing a SQL statement, not just queries.

By default, the application obtains the threshold value from the `QueryThreshold` general connection attribute setting, for which the default is 0 (no warnings). You can override the threshold for a JDBC `Connection` object by including the `QueryThreshold` attribute in the connection URL for the database. For example, to set `QueryThreshold` to a value of 5 seconds for the `myDSN` database:

```
jdbc:timesten:direct:dsn=myDSN;QueryThreshold=5
```

You can also use the `setQueryTimeThreshold()` method of a `TimesTenStatement` object to set the threshold. This overrides the connection attribute setting and the `Connection` object setting.

You can retrieve the current threshold value by using the `getQueryTimeThreshold()` method of the `TimesTenStatement` object.

Refer to *Oracle TimesTen In-Memory Database JDBC Extensions Java API Reference* for additional information.

Features for use with TimesTen Cache

This section discusses features related to the use of TimesTen Application-Tier Database Cache (TimesTen Cache) in TimesTen Classic.

- Setting temporary passthrough level with the ttOptSetFlag built-in procedure
- Determining passthrough status
- Managing cache groups

> **Note:** The `OraclePassword` attribute maps to the Oracle Database password. You can use the `TimesTenDataSource` method `setOraclePassword()` to set the Oracle Database password. See "Connect to the database" on page 2-9 for an example.

Setting temporary passthrough level with the ttOptSetFlag built-in procedure

TimesTen provides the `ttOptSetFlag` built-in procedure for setting various flags, including the `PassThrough` flag to temporarily set the passthrough level. You can use `ttOptSetFlag` to set `PassThrough` in a JDBC application as in the following sample statement, which sets the passthrough level to 1. The setting affects all statements that are prepared until the end of the transaction.

```
pstmt = conn.prepareStatement("call ttoptsetflag('PassThrough', 1)");
```

The example that follows has samples of code that accomplish these steps:

1. Create a prepared statement (a `PreparedStatement` instance `thePassThroughStatement`) that calls `ttOptSetFlag` using a bind parameter for passthrough level.
2. Define a method `setPassthrough()` that takes a specified passthrough setting, binds it to the prepared statement, then executes the prepared statement to call `ttOptSetFlag` to set the passthrough level.

```
thePassThroughStatement =
       theConnection.prepareStatement("call ttoptsetflag('PassThrough', ?)");
...
private void setPassthrough(int level) throws SQLException{
  thePassThroughStatement.setInt(1, level);
  thePassThroughStatement.execute();
}
```

See "ttOptSetFlag" in *Oracle TimesTen In-Memory Database Reference* for more information about this built-in procedure.

See "PassThrough" in *Oracle TimesTen In-Memory Database Reference* for information about that general connection attribute. See "Setting a passthrough level" in *Oracle*

TimesTen Application-Tier Database Cache User's Guide for information about passthrough settings.

Determining passthrough status

You can call the `TimesTenPreparedStatement` method `getPassThroughType()` to determine whether a SQL statement is to be executed in the TimesTen database or passed through to the Oracle database for execution:

```
PassThroughType getPassThroughType()
```

The return type, `TimesTenPreparedStatement.PassThroughType`, is an enumeration type for values of the TimesTen `PassThrough` connection attribute.

You can make this call after preparing the SQL statement. It is useful with PassThrough settings of 1 or 2, where the determination of whether a statement is actually passed through is not made until compilation time.

See "Setting a passthrough level" in *Oracle TimesTen Application-Tier Database Cache User's Guide* for information about `PassThrough` settings.

Managing cache groups

In TimesTen, following the execution of a `FLUSH CACHE GROUP`, `LOAD CACHE GROUP`, `REFRESH CACHE GROUP`, or `UNLOAD CACHE GROUP` statement, the `Statement` method `getUpdateCount()` returns the number of cache instances that were flushed, loaded, refreshed, or unloaded.

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

Features for use with replication

For TimesTen Classic applications that employ replication, you can improve performance by using *parallel replication*, which uses multiple threads acting in parallel to replicate and apply transactional changes to databases in a replication scheme. TimesTen supports the following types of parallel replication:

- Automatic parallel replication (`ReplicationApplyOrdering=0`): Parallel replication over multiple threads that automatically enforces transactional dependencies and all changes applied in commit order. This is the default.
- Automatic parallel replication with disabled commit dependencies (`ReplicationApplyOrdering=2`): Parallel replication over multiple threads that automatically enforces transactional dependencies, but does not enforce transactions to be committed in the same order on the subscriber database as on the master database. In this mode, you can optionally specify replication tracks.

See "Configuring parallel replication" in *Oracle TimesTen In-Memory Database Replication Guide* for additional information and usage scenarios.

For JDBC applications that use parallel replication and specify replication tracks, you can specify the track number for transactions on a connection through the following `TimesTenConnection` method. (Alternatively, use the general connection attribute `ReplicationTrack` or the `ALTER SESSION` parameter `REPLICATION_TRACK`.)

- `void setReplicationTrack(int` *`track`*`)`

`TimesTenConnection` also has the corresponding getter method:

- `int getReplicationTrack()`

Refer to *Oracle TimesTen In-Memory Database JDBC Extensions Java API Reference* for additional information.

Handling errors

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

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

This section includes the following topics.

About fatal errors, non-fatal errors, and warnings

When operations are not completely successful, TimesTen can return a fatal error, a non-fatal error, or a warning.

Handling fatal errors

Fatal errors make the database inaccessible until it can be recovered. When a fatal error occurs, all database connections are required to disconnect. No further operations may complete. Fatal errors are indicated by TimesTen error codes 846 and 994. Error handling for these errors should be different from standard error handling. In particular, the code should roll back the current transaction and, to avoid out-of-memory conditions in the server, disconnect from the database. Shared memory from the old TimesTen instance is not freed until all connections that were active at the time of the error have disconnected. Inactive applications still connected to the old TimesTen instance may have to be manually terminated.

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

- Every connection to the database is invalidated, a new memory segment is allocated and applications are required to disconnect.
- The database is recovered from the checkpoint and transaction log files upon the first subsequent initial connection.
  - The recovered database reflects the state of all durably committed transactions and possibly some transactions that were committed non-durably.
  - No uncommitted or rolled back transactions are reflected.

Handling non-fatal errors

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

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

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

An application can handle non-fatal errors by modifying its actions or, in some cases, by rolling back one or more offending transactions, as described in "Rolling back failed transactions" on page 2-44.

Also see "Reporting errors and warnings", which follows shortly.

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

About warnings

TimesTen returns warnings when something unexpected occurs. Here are some examples of events that cause TimesTen to issue a warning:

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

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

Also see "Reporting errors and warnings" immediately below.

Abnormal termination

In some cases, such as with a process failure, an error cannot be returned, so TimesTen automatically rolls back the transactions of the failed process.

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 if there are one or more database access errors and a `SQLWarning` object is generated if there are one or more warning messages. A single call may return multiple errors or warnings or both, 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 2–15 and Example 2–16 demonstrate how you might iterate through the lists of returned `SQLException` and `SQLWarning` objects to report all of the errors and warnings, respectively.

Example 2–15 Printing exceptions

The following 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 2–16 Printing warnings

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;
}
```

Catching 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 the `SQLException` method `getSQLState()` to return the SQL state and the `getErrorCode()` method to return TimesTen error codes, as shown in Example 2–17.

Also refer to the entry for `TimesTenVendorCode` in *Oracle TimesTen In-Memory Database JDBC Extensions Java API Reference* for error information.

Example 2–17 Catching an error

The TimesTen Classic s Quick Start ample applications require you to load the their schema before they are executed. The following `catch` statement alerts the user that `appuser` 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 appuser.customer " +
      "does not exist.\n\t Please reinitialize the database.");
  } else if (ex.getErrorCode() == 907) {
    errStream.println("\nError: Attempting to insert a row " +
      "with a duplicate primary key.\n\tPlease reinitialize the database.");
}
```

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

Consider this example:

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

The following is equivalent:

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

Rolling back failed transactions

In some situations, such as recovering from a deadlock or lock timeout, you should explicitly roll back the transaction using the `Connection` method `rollback()`, as in the following example.

Example 2–18 Rolling back a transaction

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

    System.out.println("\nClosing the connection\n");
    conn.close();
} catch (SQLException ex) {

  reportSQLExceptions(ex);
}
```

A transaction rollback consumes resources and the entire transaction is in effect wasted. To avoid unnecessary rollbacks, design your application to avoid contention and check the application or input data for potential errors before submitting it.

> **Note:** If your application aborts, crashes, or disconnects in the middle of an active transaction, TimesTen automatically rolls back the transaction.

Retrying after transient errors (JDBC)

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

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

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

This is returned by the `getSQLState()` method of the `SQLException` class and may be encountered by method calls from any of the following JDBC types:

- `Connection`
- `Statement`
- `PreparedStatement`
- `CallableStatement`
- `ResultSet`
- `Connection`
- `Statement`
- `PreparedStatement`
- `CallableStatement`
- `ResultSet`

Here is an example:

```
// Database connection object
Connection        dbConn;

// Open the connection  to the database
...

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

...

// Prepre the SQL statements
PreparedStatement stmtQuery = dbConn.prepare("SELECT ...");
PreparedStatement stmtUpdate = dbConn.prepare("UPDATE ...");

...

// Set max retries for transaction to 5
int retriesLeft = 5;

// Records outcome
boolean success = false;

// Excute transaction with retries until success or retries exhausted
while (  retriesLeft > 0  )
{
```

```
    try {

        // First execute the query

        // Set input values
        stmtQuery.setInt(1, ...);
        stmtQuery.setString(2, ...);

        // Execute and process results
        ResultSet rs = stmtQuery.executeQuery();
        while (  rs.next()  )
        {
            int val1 = rs.getInt(1);
            String val2 = rs.getString(2);
            ...
        }
        rs.close();
        rs = null;

        // Now excute the update

        // Set input values
        stmtUpdate.setInt(1,...);
        stmtUpdate.setString(2,...);

        // Execute and check number of rows affected
        int updCount = stmtUpdate.executeUpdate();
        if (  updCount < 1  )
        {
            ...
        }

        // And finally commit
        dbConn.commit();

        // We are done
        success = true;
        break;

    } catch ( SQLException sqe ) {

        if (  sqe.getSQLState().equals("TT005")  ) // grid transient error
        {
            // decrement retry count
            retriesLeft--;
            // and rollback the transaction ready for retry
            try {
                dbConn.rollback();
            } catch ( SQLException sqer ) {
                // This is a fatal error so handle accordingly
            }
        }
        else
        {
            // handle other kinds of error
            ...
        }
    }
} // end of retry loop
```

```
if (  ! success  )
{
    // Handle the failure
    ...
}
```

Note: Example 2–22 in "Failover delay and retry settings" on page 2-53 also shows how to retry for transient errors.

JDBC support for automatic client failover

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

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

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

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

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

Any of the following error conditions indicates automatic client failover.

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

This section discusses TimesTen JDBC extensions related to automatic client failover, covering the following topics:

- Features and functionality of JDBC support for automatic client failover
- Configuration of automatic client failover
- Synchronous detection of automatic client failover
- Asynchronous detection of automatic client failover
- Application action in the event of failover

For TimesTen Scaleout, see "Client connection failover" in *Oracle TimesTen In-Memory Database Scaleout User's Guide* for additional information. For TimesTen Classic, see "Using automatic client failover" in *Oracle TimesTen In-Memory Database Operations*

Guide. For related information for developers, see "Using automatic client failover in your application" in *Oracle TimesTen In-Memory Database C Developer's Guide*.

Notes:

- Automatic client failover applies only to client/server connections. The functionality described here does not apply to a direct connection.
- Automatic client failover is complementary to Oracle Clusterware in situations where Oracle Clusterware is used, though the two features are not dependent on each other. You can also refer to "Using Oracle Clusterware to Manage Active Standby Pairs" in *Oracle TimesTen In-Memory Database Replication Guide* for information about Oracle Clusterware.

Features and functionality of JDBC support for automatic client failover

This section discusses general TimesTen JDBC features related to client failover, and functionality relating specifically to pooled connections.

Refer to *Oracle TimesTen In-Memory Database JDBC Extensions Java API Reference* for additional information about the TimesTen JDBC classes, interfaces, and methods discussed here.

General Client Failover Features

TimesTen JDBC support for automatic client failover provides two mechanisms for detecting a failover:

- *Synchronous detection*, through a SQL exception: After an automatic client failover, JDBC objects created on the failed connection—such as statements, prepared statements, callable statements, and result sets—can no longer be used. A Java SQL exception is thrown if an application attempts to access any such object. By examining the SQL state and error code of the exception, you can determine whether the exception is the result of a failover situation.
- *Asynchronous detection*, through an event listener: An application can register a user-defined client failover event listener, which is notified of each event that occurs during the process of a failover.

TimesTen JDBC provides the following features, in package `com.timesten.jdbc`, to support automatic client failover.

- `ClientFailoverEvent` class

  This class is used to represent events that occur during a client failover: begin, end, abort, or retry.

- `ClientFailoverEventListener` interface

  An application interested in client failover events must have a class that implements this interface, which is the mechanism to listen for client failover events. At runtime, the application must register `ClientFailoverEventListener` instances through the TimesTen connection (see immediately below).

  You can use a listener to proactively react to failure detection, such as by refreshing connection pool statement caches, for example.

- Methods in the `TimesTenConnection` interface

This interface specifies the methods `addConnectionEventListener()` and `removeConnectionEventListener()` to register or remove, respectively, a client failover event listener.

- A constant, `TT_ERR_FAILOVERINVALIDATION`, in the `TimesTenVendorCode` interface

  This enables you to identify an event as a failover event.

Client failover features for pooled connections

TimesTen recommends that applications using pooled connections (`javax.sql.PooledConnection`) or connection pool data sources (`javax.sql.ConnectionPoolDataSource`) use the synchronous mechanism noted previously to handle stale objects on the failed connection. Java EE application servers manage pooled connections, so applications are not able to listen for events on pooled connections. And application servers do not implement and register an instance of `ClientFailoverEventListener`, because this is a TimesTen extension.

Configuration of automatic client failover

Refer to "Configuring automatic client failover for TimesTen Classic" in *Oracle TimesTen In-Memory Database Operations Guide* or "Client connection failover" in the *Oracle TimesTen In-Memory Database Scaleout User's Guide* for complete details on managing client connection failover in TimesTen.

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

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

Be aware of these TimesTen connection attributes:

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

Note: If you set any of these in `odbc.ini` or the connection string, the settings are applied to the failover connection. They cannot be set in your application (including by `ALTER SESSION`).

Synchronous detection of automatic client failover

If, in a failover situation, an application attempts to use objects created on the failed connection, then JDBC throws a SQL exception. The vendor-specific exception code is set to `TimesTenVendorCode.TT_ERR_FAILOVERINVALIDATION`.

Detecting a failover through this mechanism is referred to as synchronous detection. The following example demonstrates this.

Example 2–19 Synchronous detection of automatic client failover

```
try {
   // ...
   // Execute a query on a previously prepared statement.
   ResultSet theResultSet = theStatement.executeQuery("select * from dual");
   // ...

} catch (SQLException sqlex) {
   sqlex.printStackTrace();
   if (sqlex.getErrorCode() == TimesTenVendorCode.TT_ERR_FAILOVERINVALIDATION) {
   // Automatic client failover has taken place; discontinue use of this object.
   }
}
```

Asynchronous detection of automatic client failover

Asynchronous failover detection requires an application to implement a client failover event listener and register an instance of it on the TimesTen connection. This section describes the steps involved:

1. Implement a client failover event listener.
2. Register the client failover listener instance.
3. Remove the client failover listener instance.

Implement a client failover event listener

TimesTen JDBC provides the `com.timesten.jdbc.ClientFailoverEventListener` interface for use in listening for events, highlighted by the following method:

- `void notify(ClientFailoverEvent` *`event`*`)`

To use asynchronous failover detection, you must create a class that implements this interface, then register an instance of the class at runtime on the TimesTen connection (discussed shortly).

When a failover event occurs, TimesTen calls the `notify()` method of the listener instance you registered, providing a `ClientFailoverEvent` instance that you can then examine for information about the event.

The following example shows the basic form of a `ClientFailoverEventListener` implementation.

Example 2–20 Asynchronous detection of automatic client failover

```
private class MyCFListener implements ClientFailoverEventListener {
   /* Applications can build state system to track states during failover.
      You may want to add methods that talks about readiness of this Connection
      for processing.
   */
   public void notify(ClientFailoverEvent event) {

      /* Process connection failover type */
      switch(event.getTheFailoverType()) {
      case TT_FO_CONNECTION:
         /* Process session fail over */
         System.out.println("This should be a connection failover type " +
                            event.getTheFailoverType());
         break;

      default:
         break;
      }
      /* Process connection failover events */
      switch(event.getTheFailoverEvent()) {
      case BEGIN:
         System.out.println("This should be a BEGIN event " +
                            event.getTheFailoverEvent());
         /* Applications cannot use Statement, PreparedStatement, ResultSet,
            etc. created on the failed Connection any longer.
         */
         break;

      case END:
         System.out.println("This should be an END event " +
                            event.getTheFailoverEvent());

         /* Applications may want to re-create Statement and PreparedStatement
            objects at this point as needed.
         */
         break;

      case ABORT:
         System.out.println("This should be an ABORT event " +
                            event.getTheFailoverEvent());
         break;

      case ERROR:
         System.out.println("This should be an ERROR event " +
                            event.getTheFailoverEvent());
         break;

      default:
         break;
      }
   }
}
```

The `event.getTheFailoverType()` call returns an instance of the nested class `ClientFailoverEvent.FailoverType`, which is an enumeration type. In TimesTen, the only supported value is `TT_FO_CONNECTION`, indicating a connection failover.

The `event.getTheFailoverEvent()` call returns an instance of the nested class `ClientFailoverEvent.FailoverEvent`, which is an enumeration type where the value can be one of the following:

- `BEGIN`, if the client failover has begun
- `END`, if the client failover has completed successfully
- `ERROR`, if the client failover failed but will be retried
- `ABORT`, if the client failover has aborted

Register the client failover listener instance

At runtime you must register an instance of your failover event listener class with the TimesTen connection object, so that TimesTen can call the `notify()` method of the listener class as needed for failover events.

`TimesTenConnection` provides the following method for this.

- `void addConnectionEventListener (ClientFailoverEventListener` *`listener`*`)`

Create an instance of your listener class, then register it using this method. The following example establishes the connection and registers the listener. Assume `theDsn` is the JDBC URL for a TimesTen Client/Server database and `theCFListener` is an instance of your failover event listener class.

Example 2–21 Registering the client failover listener

```
try {
   /* Assume this is a client/server conn; register for conn failover. */
   Class.forName("com.timesten.jdbc.TimesTenClientDriver");
   String url = "jdbc:timesten:client:" + theDsn;
   theConnection = (TimesTenConnection)DriverManager.getConnection(url);
   theConnection.addConnectionEventListener(theCFListener);
   /* Additional logic goes here; connection failover listener is
      called if there is a fail over.
   */
}
catch (ClassNotFoundException cnfex) {
   cnfex.printStackTrace();
}
catch (SQLException sqlex) {
   sqlex.printStackTrace();
}
```

Remove the client failover listener instance

The `TimesTenConnection` interface defines the following method to deregister a failover event listener:

- `void removeConnectionEventListener (ClientFailoverEventListener` *`listener`*`)`

Use this method to deregister a listener instance.

Application action in the event of failover

This section discusses these topics:

- Application steps for failover

- Failover delay and retry settings

Application steps for failover

If you receive any of the error conditions noted at the beginning of automatic client failover discussion in "JDBC support for automatic client failover" on page 2-47 in response to an operation in your application, then application failover is in progress. Perform these recovery actions:

1. Roll back all transactions on the connection.
2. Clean up all objects from the previous connection. None of the state or objects associated with the previous connection are preserved.
3. Assuming `TTC_NoReconnectOnFailover=0` (the default), sleep briefly, as discussed in the next section, "Failover delay and retry settings". If `TTC_NoReconnectOnFailover=1`, then you must instead manually reconnect the application to an alternate database or database element.
4. Recreate and reprepare all objects related to your connection.
5. Restart any in-progress transactions from the beginning.

Failover delay and retry settings

The reconnection to another database or database element during automatic client failover may take some time. If your application attempts recovery actions before TimesTen has completed its client failover process, you may receive another failover error condition as listed at the beginning of automatic client failover discussion in "JDBC support for automatic client failover" on page 2-47.

Therefore, your application should place all recovery actions within a loop with a short delay before each subsequent attempt, where the total number of attempts is limited. If you do not limit the number of attempts, the application may appear to hang if the client failover process does not complete successfully. For example, your recovery loop could use a retry delay of 100 milliseconds with a maximum number of retries limited to 100 attempts. The ideal values depend on your particular application and configuration.

Example 2–22 illustrates this point.

Example 2–22 Client failover retry in JDBC

This code snippet, using the synchronous detection method, illustrates how you might handle the retrying of connection failover errors in a JDBC application. Code not directly relevant is omitted (...).

```
// Database connection object
Connection        dbConn;

// Open the connection  to the database
...

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

...

// Prepre the SQL statements
PreparedStatement stmtQuery = dbConn.prepare("SELECT ...");
PreparedStatement stmtUpdate = dbConn.prepare("UPDATE ...");
```

```
...

// Set max retries to 100
int retriesLeft = 100;
// and retry delay to 100 ms
int retryDelay = 100;

// Records outcome
boolean success = false;
Boolean needReprepare = false;

// Execute transaction with retries until success or retries exhausted
while (  retriesLeft > 0  )
{
    try {

        // Do we need to re-prepare
        if ( needReprepare )
        {
            Thread.sleep( retryDelay ); // delay before proceeding
            stmtQuery = dbConn.prepare("SELECT ...");
            stmtUpdate = dbConn.prepare("UPDATE ...");
            needReprepare = false;
        }

        // First execute the query

        // Set input values
        stmtQuery.setInt(1, ...);
        stmtQuery.setString(2, ...);

        // Execute and process results
        ResultSet rs = stmtQuery.executeQuery();
        while (  rs.next()  )
        {
            int val1 = rs.getInt(1);
            String val2 = rs.getString(2);
            ...
        }
        rs.close();
        rs = null;

        // Now execute the update

        // Set input values
        stmtUpdate.setInt(1,...);
        stmtUpdate.setString(2,...);

        // Execute and check number of rows affected
        int updCount = stmtUpdate.executeUpdate();
        if (  updCount < 1  )
        {
            ...
        }

        // And finally commit
        dbConn.commit();

        // We are done
        success = true;
```

```
        break;

    } catch ( SQLException sqe ) {

       if (  (sqe.getErrorCode() == 47137) ||
             ( (sqe.getErrorCode() == 30105) &&
(sqe.getSQLState().equals("08006")) )  )
        // connection failover error
        {
            // decrement retry count
            retriesLeft--;
            // rollback the transaction ready for retry
            dbConn.rollback();
            // and indicate that we need to re-prepare
            needReprepare = true;
        }
        else
        {
            // handle other kinds of error
            ...
        }

    }

} // end of retry loop

if (  ! success  )
{
    // Handle the failure
    ...
}
```

Client routing in TimesTen Scaleout

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

This section includes these topics:

- Building a distribution key
- Getting the element location given a set of key values
- Supported data types
- Restrictions

Building a distribution key

The client application must identify and build a distribution key, which is required to determine the elements (or replica set) that allocates a specific set of key values. The `TimesTenDistributionKey` and `TimesTenDistributionKeyBuilder` interfaces specify functionality for building a distribution key.

Note: The application has to maintain a client connection to the database to build the distribution key, compute the element IDs or replica set ID, and build a connection to an element of the database based on either of the three, as shown in Example 2–23 and Example 2–24.

The `TimesTenDistributionKeyBuilder` interface specifies the following builder method to support compound keys with different data types.

```
subkey(Object subkey, java.sql.Types subkeyType)
```

For a compound distribution key, invoke the `subkey` method once for every column in the hash distribution key of the table. Invoke each `subkey` in the same order as the key values and types of the distribution key columns of the table.

Getting the element location given a set of key values

Once you build a distribution key, use the `getElementIDs` or `getReplicaSetID` method of the `TimesTenDistributionKey` interface to get the element IDs or replica set ID that stores the key values specified in the distribution key.

Note: For TimesTen Scaleout, the `TimesTenDataSource` class implements factory methods for connection and distribution key.

Example 2–23 Getting the element IDs and replica set ID

The example computes and prints the element IDs and replica set ID for a key value in a single column distribution key.

```
import java.sql.SQLException;
import java.sql.Types;

import com.timesten.jdbc.TimesTenDataSource;
import com.timesten.jdbc.TimesTenDistributionKey;
import com.timesten.jdbc.TimesTenDistributionKeyBuilder;

public class ClientRouting {

   public static void main(String[] args) {

      try {

         /* Establish a connection to an element of the database. Maintain this
          * connection for the duration of the application for computing element
          * IDs and creating connections. */
         TimesTenDataSource ds = new TimesTenDataSource();
         ds.setUrl("jdbc:timesten:client:database1");
         ds.setUser("terry");
         ds.setPassword("password");

         /* Build a distribution key. The distribution key is composed of a single
          * TT_INTEGER column. */
         TimesTenDistributionKey dk = ds.createTimesTenDistributionKeyBuilder()
               .subkey(3, Types.INTEGER)
               .build();

         // Get the element IDs for the distribution key.
```

```
         short[] elementIDs = dk.getElementIDs();

         for (short id : elementIDs) {
            System.out.println("Distribution key(3), element ID: " + id);
         }

         // Get the replica set ID for the disitribution key.
         System.out.println("Distribution key(3), replica set ID: " +
 dk.getReplicaSetID());

      } catch (SQLException ex) {
         ...
      }
   }
}
```

This code snippet computes and prints the element IDs and replica set ID for a set of key values in a distribution key composed of more than one column.

```
...
/* Build a distribution key. The distribuion key is composed of two columns --
 * one TT_INTEGER and one VARCHAR2. */
dk = ds.createTimesTenDistributionKeyBuilder()
      .subkey(1, Types.INTEGER)
      .subkey("john.doe", Types.VARCHAR)
      .build();

// Get the element IDs for the distribution key
elementIDs = dk.getElementIDs();
for (short id : elementIDs) {
   System.out.println("Distribution key(1, john.doe), element ID: " + id);
}
// Get the replica set ID for the distribution key.
System.out.println("Distribution key(1, john.doe), replica set ID: " +
 dk.getReplicaSetID());
...
```

Connecting to an element based on a distribution key

Your client application may use any custom method to connect to a specific element of a database in TimesTen Scaleout. However, the features specified in the `TimesTenConnectionBuilder` interface enable your application to connect to an optimal element based on a distribution key, element ID, or replica set ID.

Example 2–24 Connecting to an element based on a distribution key

This example builds a distribution key and then builds a connection with it.

```
import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.Types;

import com.timesten.jdbc.TimesTenDataSource;
import com.timesten.jdbc.TimesTenDistributionKey;
import com.timesten.jdbc.TimesTenDistributionKeyBuilder;

public class ClientRouting {

   public static void main(String[] args) {
```

```
      try {

         // Create and mantain connection to database.
         TimesTenDataSource ds = new TimesTenDataSource();
         ds.setUrl("jdbc:timesten:client:database1");
         ds.setUser("terry");
         ds.setPassword("password");

         // Build a distribution key.
         TimesTenDistributionKey dk = ds.createTimesTenDistributionKeyBuilder()
               .subkey(1, Types.INTEGER)
               .subkey("john.doe", Types.VARCHAR)
               .build();

         // Connect to optimal element based on a distribuion key.
         Connection conn;
         conn = ds.createTimesTenConnectionBuilder()
               .user("terry")
               .password("password")
               .distributionKey(dk)
               .build();
         Statement stmt = conn.createStatement();
         stmt.execute("... SQL statement here ...");
         stmt.close();
         conn.close();

      } catch (SQLException ex) {
         ...
      }
   }
}
```

This code snippet builds a connection based on an element ID.

```
...
// Connect to optimal element based on an element ID.
short[] elementIDs = dk.getElementIDs();
conn = ds.createTimesTenConnectionBuilder()
      .user("terry")
      .password("password")
      .elementID(elementIDs[0])
      .build();
Statement stmt = conn.createStatement();
stmt.execute("... SQL statement here ...");
stmt.close();
conn.close();
...
```

This code snippet builds a connection based on a replica set ID.

```
...
// Connect to optimal element based on a replica set ID.
short repSetID = dk.getReplicaSetID();
conn = ds.createTimesTenConnectionBuilder()
      .user("terry")
      .password("password")
      .replicaSetID(repSetID)
      .build();
Statement stmt = conn.createStatement();
stmt.execute("... SQL statement here ...");
```

```
stmt.close();
conn.close();
...
```

Supported data types

Table 2–5 describes the supported data types and acceptable object types. For best performance, use the recommended object types to avoid type conversion.

Table 2–5 Supported data types and acceptable object types

SQL type	java.sql.Types	Recommended Object Class	Acceptable Object Classes
TT_TINYINT	Types.TINYINT	Short	Byte, Short
TT_SMALLINT	Types.SMALLINT	Short	Byte, Short
TT_INTEGER	Types.INTEGER	Integer	Byte, Short, Integer
TT_BIGINT	Types.BIGINT	Long	Byte, Short, Integer, Long
CHAR	Types.CHAR	String	String
NCHAR	Types.NCHAR	String	String
VARCHAR2	Types.VARCHAR	String	String
NVARCHAR	Types.NCHAR	String	String
NUMBER	Types.DECIMAL TYPES.NUMERIC	BigDecimal	BigDecimal toString() method will be invoked for other classes.

Restrictions

The JDBC extensions for client routing in TimesTen Scaleout share the same restrictions as the ones listed in "Client routing API for TimesTen Scaleout" in the *Oracle TimesTen In-Memory Database C Developer's Guide*.

3

Using JMS/XLA for Event Management

You can use the TimesTen JMS/XLA API, supported by TimesTen Classic, to monitor TimesTen for changes to specified tables in a local database and receive real-time notification of these changes. The primary purpose of JMS/XLA is as a high-performance, asynchronous alternative to triggers.

JMS/XLA implements Java Message Service (JMS) interfaces to make the functionality of the TimesTen Transaction Log API (XLA) available to Java applications. JMS information and resources are available at the following location:

`http://www.oracle.com/technetwork/java/jms/index.html`

In addition, the standard JMS API documentation is included with the TimesTen installation at the following location:

installation_dir`/3rdparty/jms1.1/doc/api/index.html`

For information about tuning TimesTen JMS/XLA applications for improved performance, see "Tuning JMS/XLA applications" on page 5-5.

> **Note:** In the unlikely event that the TimesTen replication solutions described in *Oracle TimesTen In-Memory Database Replication Guide* do not meet your needs, it is possible to use JMS/XLA to build a custom data replication solution.

This chapter includes the following topics:

JMS/XLA concepts

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

Subscribe to updates by establishing a JMS `Session` instance that provides a connection to XLA and then 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 database. TimesTen and the application receiving the notifications must reside on the same system.

Note: The JMS/XLA API is a wrapper for XLA. XLA obtains update records directly from the transaction log buffer or transaction log files, so the records are available until they are read. XLA also allows multiple readers to access transaction log updates simultaneously.

See "XLA and TimesTen Event Management" in *Oracle TimesTen In-Memory Database C Developer's Guide* for information about XLA.

This section includes the following topics:

How XLA reads records from the transaction log

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

New transaction log records are always written to the end of the transaction log buffer as they are generated. Transaction log records are periodically flushed in batches from the log buffer in memory to transaction log files on the file system.

Applications can use XLA to monitor the transaction log for changes to the database. XLA reads through the transaction log, filters the log records, and delivers 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 database simultaneously, transaction log records from the different applications are interleaved in the transaction log.

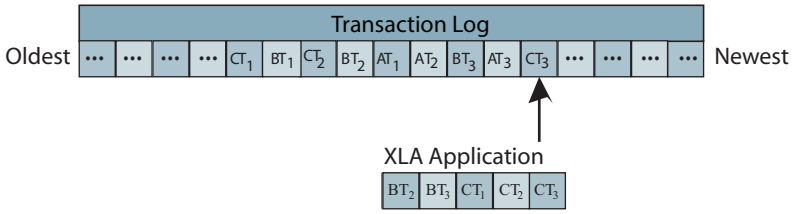
XLA transparently extracts all transaction log records associated with a particular transaction and delivers them in a contiguous list to the application.

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

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

Consider the example transaction log illustrated in Figure 3–1 and Example 3–1 that follow, which illustrate most of these basic XLA concepts.

Figure 3–1 Records extracted from the transaction log



Example 3–1 Reading transaction log records

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

CT1 - Application C updates row 1 of table W with value 7.7.
BT1 - Application B updates row 3 of table X with value 2.
CT2 - Application C updates row 9 of table W with value 5.6.
BT2 - Application B updates row 2 of table Y with value "XYZ".
AT1 - Application A updates row 1 of table Z with value 3.
AT2 - Application A updates row 3 of table Z with value 4.
BT3 - Application B commits its transaction.
AT3 - Application A rolls back its transaction.
CT3 - Application C commits its transaction.

An XLA application that is set up to detect changes to tables W, Y, and Z would see the following:

BT2 and BT3 - Update row 2 of table Y with value "XYZ" and commit.
CT1 - Update row 1 of table W with value 7.7.
CT2 and CT3 - Update row 9 of table W with value 5.6 and commit.

This example demonstrates the following:

- Transaction records for application B and application C all appear.
- Though the records for application C begin to appear in the transaction log before those for application B, the commit for application B (BT3) appears in the transaction log before the commit for application C (CT3). As a result, the records for application B are returned to the XLA application ahead of those for application C.
- The application B update to table X (BT1) is not presented because XLA is not set up to detect changes to table X.
- The application A updates to table Z (AT1 and AT2) are never presented because it did not commit and was rolled back (AT3).

XLA and materialized views

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

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

For more information about materialized views, see the following:

- "CREATE MATERIALIZED VIEW" in *Oracle TimesTen In-Memory Database SQL Reference*
- "Understanding materialized views" in *Oracle TimesTen In-Memory Database Operations Guide*

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.

The rest of this section covers the following:

- How bookmarks work
- Replicated bookmarks
- XLA bookmarks and transaction log holds

How bookmarks work

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 `ttXlaSubscribe` and `ttXlaUnsubscribe` built-in procedures through JDBC to start and stop the XLA subscription for a table, you explicitly specify the name of the bookmark to be used.

Bookmarks are reset to the last read position whenever an acknowledgment is received. For more information about how update messages are acknowledged, see the "XLA acknowledgment modes" on page 3-8.

You can remove a durable subscription by calling `unsubscribe()` on the JMS `Session` object. 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 3-14.

A bookmark subscription cannot be altered when it is in use. To alter a subscription, you must close the message consumer, alter the subscription using `ttXlaSubscribe` and `ttXlaUnsubscribe`, and open the message consumer.

Note: You can also use the `ttXlaBookmarkCreate` TimesTen built-in procedure to create bookmarks. See "ttXlaBookmarkCreate" in *Oracle TimesTen In-Memory Database Reference* for information about that function.

Replicated bookmarks

If you are using an active standby pair replication scheme, you have the option of using *replicated bookmarks*, according to the `replicatedBookmark` attribute of the `<topic>` element in the `jmsxla.xml` file as discussed in "JMS/XLA configuration file and topics" on page 3-6. For a replicated bookmark, operations on the bookmark are replicated to the standby database as appropriate, assuming there is suitable write

privilege for the standby. This enables more efficient recovery of your bookmark positions if a failover occurs.

When you use replicated bookmarks, steps must be taken in the following order:

1. Create the active standby pair replication scheme. (This is accomplished by the `create active standby pair` operation, or by the `ttCWAdmin -create` command in a Clusterware-managed environment.)
2. Create the bookmarks.
3. Subscribe the bookmarks.
4. Start the active standby pair, at which time duplication to the standby occurs and replication begins. (This is accomplished by the `ttRepAdmin -duplicate` command, or by the `ttCWAdmin -start` command in a Clusterware-managed environment.)

Notes:

- Alternatively, if you use `ttXlaBookmarkCreate` to create a bookmark, that function has a parameter for specifying a replicated bookmark.
- If you specify replicated bookmarks in the JMS/XLA configuration file, JMS/XLA will create and subscribe to the bookmarks when the application is started. (Also see "JMS/XLA configuration file and topics" on page 3-6.)

Be aware of the following usage notes:

- The position of the bookmark in the standby database is very close to that of the bookmark in the active database; however, because the replication of acknowledge operations is asynchronous, you may see a small window of duplicate updates when there is a failover, depending on how often acknowledge operations are performed.
- It is permissible to drop the active standby pair scheme while replicated bookmarks exist. The bookmarks of course cease to be replicated at that point, but are not deleted. If you subsequently re-enable the active standby pair scheme, these bookmarks are automatically added to the scheme.
- You cannot delete replicated bookmarks while the replication agent is running.
- You can only read and acknowledge a replicated bookmark in the active database. Each time you acknowledge a replicated bookmark, the acknowledge operation is asynchronously replicated to the standby database.

XLA bookmarks and transaction log holds

You should be aware that when XLA is in use, there is a hold on TimesTen transaction log files until the XLA bookmark advances. The hold prevents transaction log files from being purged until XLA can confirm it no longer needs them. If a bookmark becomes stuck, which can occur if an XLA application terminates unexpectedly or disconnects without first deleting its bookmark or disabling change tracking, the log hold persists and there may be an excessive accumulation of transaction log files. This accumulation may result in file system space being filled.

For information about monitoring and addressing this situation, see "Monitoring accumulation of transaction log files" in *Oracle TimesTen In-Memory Database Operations Guide*.

JMS/XLA configuration file and topics

To connect to XLA, establish a connection to a JMS `Topic` object that corresponds to a particular database. The JMS/XLA configuration file provides the mapping between topic names and databases.

You can specify a replicated bookmark by setting `replicatedBookmark="yes"` in the `<topic>` element when you specify the topic. The default setting is `"no"`. Also see "XLA bookmarks" on page 3-4.

By default, JMS/XLA looks for a configuration file named `jmsxla.xml` in the current working directory. If you want to use another name or location for the file, you must specify it as part of the environment variable in the `InitialContext` class and add the location to the classpath.

Example 3–2 Specifying the JMS/XLA configuration file

The following code specifies the configuration file as part of the environment variable in the `InitialContext` class.

```
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 uses the class loader to locate the JMS/XLA configuration file if `XlaConstants.CONFIG_FILE_NAME` is set. In this example, 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` variable.

The JMS/XLA configuration file can also be located in subdirectories, as follows:

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

In this case, 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` variable.

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

Example 3–3 Defining a topic in the configuration file

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

For example, this configuration maps the `DemoDataStore` topic to the `TestDB` DSN:

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

```
</xlaconfig>
```

Example 3–4 Defining a topic to use replicated bookmarks

A topic definition can also specify whether a replicated bookmark should be used. The following repeats the preceding example, but with a replicated bookmark.

```
<xlaconfig>
  <topics>
    <topic name="DemoDataStore"
      connectionString="DSN=TestDB"
      xlaPrefetch="100" replicatedBookmark="yes" />
  </topics>
</xlaconfig>
```

XLA updates

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

You can access the message fields using the `MapMessage` getter methods. The `getMapNames()` method returns an `Enumeration` object 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.
`CREATE_SYNONYM`	A synonym has been created.
`DROP_SYNONYM`	A synonym has been dropped.
`TRUNCATE`	All rows in the table have been deleted.

For more information about the contents of an XLA update message, see "JMS/XLA MapMessage contents" on page 6-1.

XLA acknowledgment modes

The XLA acknowledgment 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 XLA bookmark to the last record that was read. This prevents previously returned records from being reread, ensuring that an application receives only new batches of 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` object.

JMS/XLA supports three acknowledgment modes:

- `AUTO_ACKNOWLEDGE`: In this mode, updates are automatically acknowledged as you receive them. Each message is delivered only once. Duplicate messages are not sent, so messages might be lost if there is an application failure. 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 when there is an application failure. JMS/XLA prefetches records according to the `xlaprefetch` attribute specified for the topic and sends an acknowledgment 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.

  See "JMS/XLA configuration file and topics" on page 3-6 for examples setting `xlaprefetch`.
- `CLIENT_ACKNOWLEDGE`: In this mode, applications are responsible for acknowledging receipt of update messages by calling `acknowledge()` on the `MapMessage` instance. JMS/XLA prefetches records according to the `xlaprefetch` attribute specified for the topic.

The following example sets the acknowledgment mode:

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

Also see "Reduce frequency of update acknowledgments" on page 5-5.

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 your application to tolerate duplicate updates so you can use `DUPS_OK_ACKNOWLEDGE`, or explicitly acknowledge updates. Explicitly acknowledging updates usually yields the best performance, if you can avoid acknowledging each message individually.

Acknowledging updates

To explicitly acknowledge an XLA update, call `acknowledge()` on the update message. Acknowledging a message implicitly acknowledges all previous messages. Typically, you receive and process multiple update messages between acknowledgments. 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.

Access control impact on XLA

Access control impacts XLA as follows:

- Any XLA functionality requires the system privilege XLA. This includes connecting to TimesTen (which also requires the CREATE SESSION privilege) as an XLA reader and executing the TimesTen XLA built-in procedures ttXlaBookmarkCreate, ttXlaBookmarkDelete, ttXlaSubscribe, and ttXlaUnsubscribe, all of which are documented in "Built-In Procedures" in *Oracle TimesTen In-Memory Database Reference*.
- A user with the XLA privilege has capabilities equivalent to the SELECT ANY TABLE, SELECT ANY VIEW, and SELECT ANY SEQUENCE system privileges.

XLA limitations

Be aware of the following XLA limitations when you use TimesTen JMS/XLA:

- JMS/XLA is available on all platforms supported by TimesTen. However, XLA does not support data transfer between different platforms.
- JMS/XLA support for LOBs is limited. See "Monitoring tables for updates" on page 3-10 for information.
- JMS/XLA does not support applications linked with a driver manager library or the client/server library.
- An XLA reader cannot subscribe to a table that uses in-memory column-based compression.
- For autorefresh cache groups, the change-tracking trigger on Oracle Database does not have column-level resolution. (To have that would be very expensive.) Therefore the autorefresh feature updates all the columns in the row, and XLA can only report that all the columns have changed, even if data did not actually change in all columns.

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. The JMS/XLA API supports a specific version of the GDK with each TimesTen release. If JMS/XLA finds other versions of the GDK loaded in the JVM, it displays a severe warning and continues processing. You can find out the GDK version supported by JMS/XLA by entering the following commands:

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

Also see "Compiling Java applications" on page 1-2.

Note: The path *timesten_home*/install is a symbolic link to *installation_dir*.

Connecting to XLA

To connect to XLA so you can receive updates, use a JMS connection factory to create a connection. Then use the connection to establish a session. When you are ready to start processing updates, call `start()` on the connection to enable message dispatching. This is shown in Example 3–5 that follows, from the `syncJMS` TimesTen Classic Quick Start sample application. See "TimesTen Quick Start and sample applications" on page 1-2.

Example 3–5 Connecting to XLA

```
/* JMS connection */
private javax.jms.TopicConnection connection;
/* JMS session */
private TopicSession session;
...
// get Connection
Context messaging = new InitialContext();
TopicConnectionFactory connectionFactory =
        (TopicConnectionFactory)messaging.lookup("TopicConnectionFactory");
connection = connectionFactory.createTopicConnection();
connection.start();
...
// get Session
session = connection.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);
```

Monitoring tables for updates

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

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

When you use `ttXlaSubscribe` to enable XLA publishing for a table, you must specify parameters for the name of the table and the name of the bookmark that are used to track the table:

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

For example, call `ttXlaSubscribe` by the JDBC `CallableStatement` interface:

```
Connection con;

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

Use `ttXlaUnsubscribe` to unsubscribe from the table during shutdown. For more information, see "Unsubscribing from a table" on page 3-14.

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

For more information about using TimesTen built-in procedures in a Java application, see "Using CALL to execute procedures and functions" on page 2-36.

Note: LOB support in JMS/XLA is limited, as follows:

- You can subscribe to tables containing LOB columns, but information about the LOB value itself is unavailable.
- Columns containing LOBs are reported as empty (zero length) or null (if the value is actually `NULL`). In this way, you can tell the difference between a null column and a non-null column.

See the next section, "Receiving and processing updates", for additional notes.

Receiving and processing updates

You can receive XLA updates either synchronously or asynchronously.

To receive and process updates for a topic synchronously, perform the following tasks.

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

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

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

Note: You may miss messages if you do not register the `MessageListener` before you start the connection. If the connection is already started, stop the connection, register the `MessageListener`, then start the connection.

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

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

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

Note: LOB support in XLA is limited. You can access LOB fields in update messages using the `MapMessage` method `getBytes()` for BLOB fields or `getString()` for CLOB or NCLOB fields; however, these fields contain zero-length data (or null data if the value is actually `NULL`).

Example 3–6, from the `asyncJMS` TimesTen Classic Quick Start sample application, uses a listener to process updates asynchronously.

Example 3–6 Using a listener to process updates asynchronously

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

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

// After setMessageListener() has been called, myListener's onMessage
// method is called for each message received.
subscriber.setMessageListener(myListener);
```

Note that `bookmark` 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.

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

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

In Example 3–7, from the `asyncJMS` TimesTen Classic Quick Start sample application, the method `onMessage()` extracts the update type from the `MapMessage` object and displays the action that the update signifies.

Example 3–7 Determining the update type

```
public void onMessage(Message message)
{
  MapMessage mapMessage = (MapMessage)message;
  String messageType = null;
  /* Standard output stream */
  private static PrintStream outStream = System.out;

  if (message == null)
  {
    errStream.println("MyListener: update message is null");
    return ;
  }

  try
  {
    outStream.println();
    outStream.println("onMessage: got a " + mapMessage.getJMSType() + " message");

    // Get the type of event (insert, update, delete, drop table, etc.).
    int type = mapMessage.getInt(XlaConstants.TYPE_FIELD);
    if (type == XlaConstants.INSERT)
```

```
    {
      outStream.println("A row was inserted.");
    }
    else if (type == XlaConstants.UPDATE)
    {
      outStream.println("A row was updated.");
    }
    else if (type == XlaConstants.DELETE)
    {
      outStream.println("A row was deleted.");
    }
    else
    {

      // Messages are also received for DDL events such as CREATE TABLE.
      // This program processes INSERT, UPDATE, and DELETE events,
      // and ignores the DDL events.
      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 the `MapMessage` method `getMapNames()`. You can retrieve individual fields from the message by name.

Example 3–8, from the `asyncJMS` TimesTen Classic Quick Start sample application, extracts the column values from insert, update, and delete messages using the column names.

Example 3–8 Extracting column values

```
/* Standard output stream */
private static PrintStream outStream = System.out;
...
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");

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

For detailed information about the contents of XLA update messages, see "JMS/XLA MapMessage contents" on page 6-1. For information about how TimesTen column types map to JMS data types and the getter methods used to retrieve the column values, see "Data type support" on page 6-10.

Terminating a JMS/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, call `close()` on the `Connection` object.

After a connection has been closed, any attempt to use it, its sessions, or its subscribers results in an `IllegalStateException` error. You can continue to use messages received through the connection, but you cannot call the `acknowledge()` method on the received message after the connection is closed.

Deleting bookmarks

Deleting XLA bookmarks during shutdown is optional. Deleting a bookmark enables the file system 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. If the bookmark is available when a durable subscriber reconnects, the subscriber receives all unacknowledged updates published since the previous connection was terminated. Keep in mind that when a bookmark exists with no application reading from it, the transaction log continues to grow and the amount of file system space consumed by your database increases.

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.

Note: You cannot delete replicated bookmarks while the replication agent is running.

Unsubscribing from a table

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

Note: If you want to drop a table, you must unsubscribe from it first.

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

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

The following example calls `ttXlaUnSubscribe` through a `CallableStatement` object.

Example 3–9 Unsubscribing from a table

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

For more information about using TimesTen built-in procedures in a Java application, see "Using CALL to execute procedures and functions" on page 2-36.

Using JMS/XLA as a replication mechanism

TimesTen replication as described in *Oracle TimesTen In-Memory Database Replication Guide* is sufficient for most customer needs; however, it is also possible to use JMS/XLA to replicate updates from one database to another. Implementing your own replication scheme on top of JMS/XLA in this way is fairly complicated, but can be considered if TimesTen replication is not feasible for some reason.

Applying JMS/XLA messages to a target database

The source database generates JMS/XLA messages. To apply the messages to a target database, 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 database may reside on a different system from the source database. The update descriptor is returned as a byte array and can be serialized for network transmission.

You must create a target database object that represents the target database so you can apply the objects from the source database. You can create a target database object named `myTargetDataStore` as an instance of the `TargetDataStoreImpl` class. For example:

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

Apply messages to `myTargetDataStore` by using the `TargetDataStore` method `apply()`. For example:

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

By default, TimesTen checks for conflicts on the target database before applying the update. If the target database 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` method `setUpdateConflictCheckFlag()` to change the behavior.

By default, TimesTen commits the update to the database 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 6-13 for more information about the `TargetDataStore` interface.

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 must be reapplied. For more information about transient XLA errors, see "Handling XLA errors" in *Oracle TimesTen In-Memory Database C Developer's Guide*.

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

The following example shows how to recover errors from a `TargetDataStore` object.

Example 3–10 Recovering errors

```
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() ) {
    // Database 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

Distributed Transaction Processing: JTA

This chapter describes the implementation of the Java Transaction API (JTA) for TimesTen Classic.

This implementation of the JTA interfaces is intended to enable Java applications, application servers, and transaction managers to use TimesTen resource managers in distributed transaction processing (DTP) environments. The TimesTen implementation is supported for use with the Oracle WebLogic Server.

Refer to the following locations for additional information.

- General information about JTA:

  `https://www.oracle.com/java/technologies/jta.html`

- Oracle WebLogic information and documentation:

  `https://www.oracle.com/middleware/technologies/weblogic.html`

As TimesTen JTA is built on top of the TimesTen implementation of the X/Open XA standard, much of the discussion here is in terms of underlying XA features. You can also refer to "Distributed Transaction Processing: XA" in *Oracle TimesTen In-Memory Database C Developer's Guide*.

This chapter includes the following topics:

Important:

- The TimesTen XA implementation does not work with the TimesTen Application-Tier Database Cache (TimesTen Cache). The start of any XA transaction fails if the cache agent is running.
- You cannot execute an XA transaction if replication is enabled.
- Do not execute DDL statements within an XA transaction.

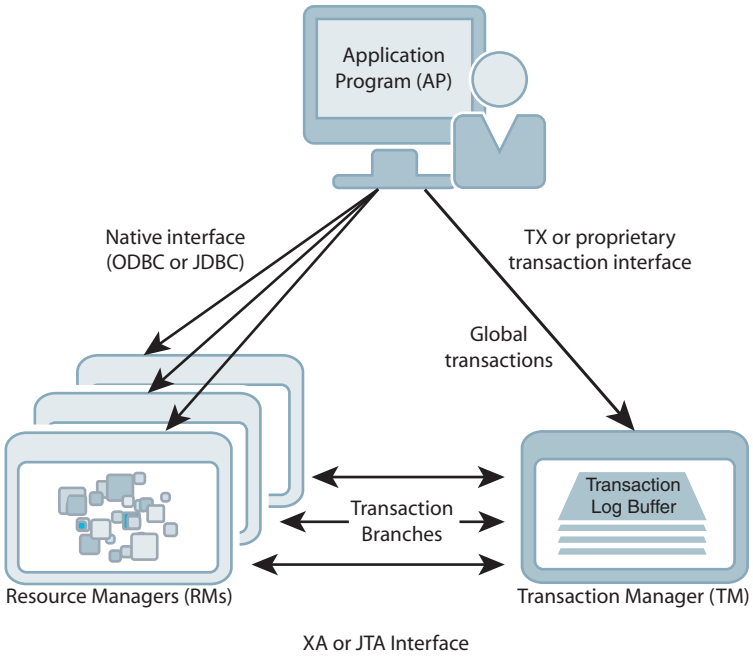
Overview of JTA

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

X/Open DTP model

Figure 4–1 illustrates the interfaces defined by the X/Open DTP model.

Figure 4–1 Distributed transaction processing model



The TX interface is what applications use to communicate with a transaction manager. The figure shows an application communicating global transactions to the transaction manager. In the DTP model, the transaction manager breaks each global transaction down into multiple branches and distributes them to separate resource managers for service. It uses the JTA interface to coordinate each transaction branch with the appropriate resource manager.

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

Global transaction control provided by the TX and JTA interfaces is distinct from local transaction control provided by the native JDBC interface. It is generally best to maintain separate connections for local and global transactions. Applications can obtain a connection handle to a TimesTen resource manager to initiate both local and global transactions over the same connection.

Two-phase commit

In a JTA implementation, the transaction manager commits the distributed branches of a global transaction by using a two-phase commit protocol.

1. In phase 1, the transaction manager directs each resource manager to prepare to commit, which is to verify and guarantee it can commit its respective branch of the global transaction. If a resource manager cannot commit its branch, the transaction manager rolls back the entire transaction in phase 2.
2. In phase 2, the transaction manager either directs each resource manager to commit its branch or, if a resource manager reported it was unable to commit in phase 1, rolls back the global transaction.

Note the following optimizations.

- If a global transaction is determined by the transaction manager to have involved only one branch, it skips phase 1 and commits the transaction in phase 2.
- If a global transaction branch is read-only, where it does not generate any transaction log records, the transaction manager commits the branch in phase 1 and skips phase 2 for that branch.

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

Using JTA in TimesTen

This section discusses the following considerations for using JTA in TimesTen:

- TimesTen database requirements for XA
- Global transaction recovery in TimesTen
- XA error handling in TimesTen

TimesTen database requirements for XA

To guarantee global transaction consistency, TimesTen XA transaction branches must be durable. The TimesTen implementation of the `xa_prepare()`, `xa_rollback()`, and `xa_commit()` functions log their actions to the file system, regardless of the value set in the `DurableCommits` general connection attribute or by the `ttDurableCommit` built-in procedure. If you must recover from a failure, both the resource manager and the TimesTen transaction manager have a consistent view of which transaction branches were active in a prepared state at the time of failure.

Global transaction recovery in TimesTen

When a database is loaded from the file system to recover after a failure or unexpected termination, any global transactions that were prepared but not committed are left pending, or in doubt. Normal processing is not enabled until the disposition of all in-doubt transactions has been resolved.

After connection and recovery are complete, TimesTen checks for in-doubt transactions. If there are no in-doubt transactions, operation proceeds as normal. If there are in-doubt transactions, other connections may be created, but virtually all operations are prohibited on those connections until the in-doubt transactions are resolved. Any other JDBC calls result in the following error:

```
Error 11035 - "In-doubt transactions awaiting resolution in recovery must be
resolved first"
```

The list of in-doubt transactions can be retrieved through the XA implementation of `xa_recover()`, then dealt with through the XA call `xa_commit()`, `xa_rollback()`, or `xa_forget()`, as appropriate. After all the in-doubt transactions are cleared, operations proceed normally.

This scheme should be adequate for systems that operate strictly under control of the transaction manager, since the first thing the transaction manager should do after connect is to call `xa_recover()`.

If the transaction manager is unavailable or cannot resolve an in-doubt transaction, you can use the `ttXactAdmin` utility `-HCommit` or `-HAbort` option to independently

commit or abort the individual transaction branches. Be aware, however, that these `ttXactAdmin` options require `ADMIN` privilege. See "ttXactAdmin" in *Oracle TimesTen In-Memory Database Reference*.

XA error handling in TimesTen

The XA specification has a limited, strictly defined set of errors that can be returned from XA interface calls. The ODBC `SQLError` mechanism returns XA defined errors, along with any additional information.

The TimesTen XA related errors begin at number 11000. Errors 11002 through 11020 correspond to the errors defined by the XA standard.

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

Using the JTA API

The TimesTen implementation of JTA provides an API consistent with that specified in the JTA specification.

This section covers the following topics for using the JTA API:

- Required packages
- Creating a TimesTen XAConnection object
- Creating XAResource and Connection objects

Regarding how to register a TimesTen DSN with WebLogic, information on configuring TimesTen for application servers and object-relational mapping frameworks is available in the TimesTen Classic Quick Start. See "TimesTen Quick Start and sample applications" on page 1-2.

Required packages

The TimesTen JDBC and XA implementations are available in the following packages:

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

Your application should also import these standard packages:

```
import java.sql.*;
import javax.sql.*;
import javax.transaction.xa.*;
```

Creating a TimesTen XAConnection object

Connections to XA data sources are established through `XADataSource` objects. You can create an `XAConnection` object for your database by using the `TimesTenXADataSource` instance as a connection factory. `TimesTenXADataSource` implements the `javax.sql.XADataSource` interface.

After creating a new `TimesTenXADataSource` instance, use the `setUrl()` method to specify a database.

The URL should look similar to the following.

- For a direct connection: `jdbc:timesten:direct:DSNname`
- For a client connection: `jdbc:timesten:client:DSNname`

You can also optionally use the `setUser()` and `setPassword()` methods to set the ID and password for a specific user.

Example 4–1 Creating a TimesTen XA data source object

In this example, the `TimesTenXADataSource` object is used as a factory to create a TimesTen XA data source object. Then the URL that identifies the TimesTen DSN (`dsn1`), the user name (`myName`), and the password (*`password`*) are set for this `TimesTenXADataSource` instance. Then the `getXAConnection()` method is used to return a connection to the object, `xaConn`.

```
TimesTenXADataSource xads = new TimesTenXADataSource();

xads.setUrl("jdbc:timesten:direct:dsn1");
xads.setUser("myName");
xads.setPassword("password");

XAConnection xaConn = null;
try {
    xaConn = xads.getXAConnection();
}
catch (SQLException e){
    e.printStackTrace();
    return;
}
```

You can create multiple connections to an XA data source object. This example creates a second connection, `xaConn2`:

```
XAConnection xaConn  = null;
XAConnection xaConn2 = null;

try {
    xaConn  = xads.getXAConnection();
    xaConn2 = xads.getXAConnection();
}
```

Example 4–2 Creating multiple TimesTen XA data source objects

This example creates two instances of `TimesTenXADataSource` for the databases named `dsn1` and `dsn2`. It then creates a connection for `dsn1` and two connections for `dsn2`.

```
TimesTenXADataSource xads = new TimesTenXADataSource();

xads.setUrl("jdbc:timesten:direct:dsn1");
xads.setUser("myName");
xads.setPassword("password");

XAConnection xaConn1 = null;
XAConnection xaConn2 = null;
XAConnection xaConn3 = null;

try {
   xaConn1 = xads.getXAConnection(); // connect to dsn1
}
catch (SQLException e){
   e.printStackTrace();
   return;
}

xads.setUrl("jdbc:timesten:direct:dsn2");
```

```
xads.setUser("myName");
xads.setPassword("password");

try {
   xaConn2 = xads.getXAConnection(); // connect to dsn2
   xaConn3 = xads.getXAConnection(); // connect to dsn2
}
catch (SQLException e){
   e.printStackTrace();
   return;
}
```

Note: Once an `XAConnection` is established, autocommit is turned off.

Creating XAResource and Connection objects

After using `getXAConnection()` to obtain an `XAConnection` object, you can use the `XAConnection` method `getXAResource()` to obtain an `XAResource` object, then the `XAConnection` method `getConnection()` to obtain a `Connection` object for the underlying connection.

Example 4–3 Getting an XA resource object and a connection

```
//get an XAResource
XAResource xaRes = null;
try {
   xaRes = xaConn.getXAResource();
}
catch (SQLException e){
   e.printStackTrace();
   return;
}

//get an underlying physical Connection
Connection conn = null;
try {
   conn = xaConn.getConnection();
}
catch (SQLException e){
   e.printStackTrace();
   return;
}
```

From this point, you can use the same connection, `conn`, for both local and global transactions. Be aware of the following, however.

- You must commit or roll back an active local transaction before starting a global transaction. Otherwise you get the `XAException` exception `XAER_OUTSIDE`.
- You must end an active global transaction before initiating a local transaction, otherwise you get a `SQLException`, "Illegal combination of local transaction and global (XA) transaction."

5

Java Application Tuning

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

This chapter is organized as follows:

Tuning JDBC applications

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

> **Note:** Also see "Working with TimesTen result sets: hints and restrictions" on page 2-12 and the notes in "Binding parameters and executing statements" on page 2-15.

Use prepared statement pooling

TimesTen supports prepared statement pooling for pooled connections through the TimesTen `ObservableConnectionDS` class. This is the TimesTen implementation of `ConnectionPoolDataSource`. Note that statement pooling is transparent to an application. Use of the `PreparedStatement` object, including preparing and closing the statement, is no different.

Enable prepared statement pooling and specify the maximum number of statements in the pool by calling the `ObservableConnectionDS` method `setMaxStatements()`. A value of 0, the default, disables prepared statement pooling. Any integer value greater than 0 enables prepared statement pooling with the value taken as the maximum number of statements. Once set, this value should not be changed.

Prepared statements or callable statements are pooled at the time of creation if the pool has not reached its capacity. You can remove a prepared statement or callable statement from the pool by calling `setPoolable(false)` on the statement object. After the statement is closed, it is removed from the pool.

Important: With prepared statement pooling, JDBC considers two statements to be identical if their SQL (including comments) is identical, regardless of other considerations such as optimizer settings. Do not use prepared statement pooling in a scenario where different optimizer hints may be applied to statements that are otherwise identical. In this scenario, a statement execution may result in the use of an identical statement from the pool with an unpredictable optimizer setting.

Use arrays of parameters for batch execution

You can improve performance by using groups, referred to as *batches*, of statement executions, calling the `addBatch()` and `executeBatch()` methods for `Statement` or `PreparedStatement` objects.

A batch can consist of a set of `INSERT`, `UPDATE`, `DELETE`, or `MERGE` statements. Statements that return result sets, such as `SELECT` statements, are not allowed in a batch. A SQL statement is added to a batch by calling `addBatch()` on the statement object. The set of SQL statements associated with a batch are executed through the `executeBatch()` method.

For `PreparedStatement` objects, the batch consists of repeated executions of a statement using different input parameter values. For each set of input values, create the batch by using appropriate `set`*`XXX`*`()` calls followed by the `addBatch()` call. The batch is executed by the `executeBatch()` method.

TimesTen recommends the following batch sizes for TimesTen Release 18.1.

In TimesTen Classic:

- 256 for `INSERT` statements
- 32 for `UPDATE` statements
- 32 for `DELETE` statements
- 32 for `MERGE` statements

In TimesTen Scaleout:

- 1024 x (number of elements in the grid) for `INSERT` statements
- 32 x (number of elements in the grid) for `UPDATE` statements
- 32 x (number of elements in the grid) for `DELETE` statements

(TimesTen Scaleout does not support `MERGE` statements.)

Example 5–1 Batching statements

```
// 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 ();
```

Example 5–2 Batching prepared statements

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

For either a `Statement` or `PreparedStatement` object, the `executeBatch()` method returns an array of update counts (`updateCounts[]` in Example 5–1 and Example 5–2 above), with one element in the array for each statement execution. The value of each element can be any of the following:

- A number indicating how many rows in the database were affected by the corresponding statement execution
- `SUCCESS_NO_INFO`, indicating the corresponding statement execution was successful, but the number of affected rows is unknown
- `EXECUTE_FAILED`, indicating the corresponding statement execution failed

  Once there is a statement execution with `EXECUTE_FAILED` status, no further statement executions are attempted.

For more information about using the JDBC batch update facility, refer to Javadoc for the `java.sql.Statement` interface, particularly information about the `executeBatch()` method, at the following location:

`https://docs.oracle.com/javase/8/docs/api/java/sql/package-summary.html`

> **Note:** Associative array parameters are not supported with JDBC batch execution. (See "Binding associative arrays" on page 2-21.)

Bulk fetch rows of TimesTen data

TimesTen provides an extension that enables an application to fetch multiple rows of data. For applications that retrieve large amounts of data, fetching multiple rows can increase performance greatly. However, when using Read Committed isolation level, 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 2-13.

Use the ResultSet method getString() sparingly

Because Java strings are immutable, the `ResultSet` method `getString()` must allocate space for a new string in addition to translating the underlying C string to a Unicode string, making it a costly call.

In addition, you should not call `getString()` on primitive numeric types, like `byte` or `int`, unless it is absolutely necessary. It is much faster to call `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 appropriate `get`*`XXX`*`()` method on a `ResultSet` object 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()`. Similarly, use the appropriate `set`*`XXX`*`()` method on the `PreparedStatement` object 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()`.

Close connections, statements, and result sets

For better performance, always close JDBC objects such as connection, statement, and result set instances when finished using them. Example 5–3 shows typical usage.

Example 5–3 Closing connection, statement, and result set

```
Connection conn = null;
Statement stmt = null;
ResultSet rs = null;
try {
  // create connections, execute statements, etc.
  // Handle any errors
} catch (SQLException ex) {
  // See "Handling errors" on page 2-41.
}
finally {
  // Close JDBC objects such as connections, statements, result sets, etc.
  if (rs != null) {
    try {
      rs.close();
    }
    catch(SQLException finalex) {
      // See "Handling errors" on page 2-41.
    }
  }
  if (stmt != null) {
    try {
      stmt.close();
    }
    catch(SQLException finalex) {
      // See "Handling errors" on page 2-41.
    }
  }
  // Always, close the connection to TimesTen
  if (conn != null) {
    try {
      conn.close();
```

```
  }
  catch(SQLException finalex) {
 // See "Handling errors" on page 2-41.
  }
}
```

Optimize queries

TimesTen provides the `TimesTenConnection` method `setTtPrefetchClose()` to optimize query performance with a `true` setting. For information, refer to "Optimizing query performance" on page 2-14.

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` method `onMessage()`. High performance applications can use some tuning to overcome some of this overhead.

This section includes the following topics:

- Configure xlaPrefetch parameter
- Reduce frequency of update acknowledgments
- Handling high event rates

> **Note:** See "Access control impact on XLA" on page 3-9 for access control considerations relevant to JMS/XLA.

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.

Reduce frequency of update acknowledgments

In JMS/XLA, acknowledging updates moves the bookmark and results in updates to system tables. You can typically improve application performance by waiting until several updates have been detected before issuing the acknowledgment. You can control the acknowledgment frequency in either of the following modes. (See "XLA acknowledgment modes" on page 3-8 for related information.)

- `DUPS_OK_ACKNOWLEDGE`, where JMS/XLA prefetches records according to the `xlaprefetch` setting, and an acknowledgment is automatically sent when the last record in the prefetched block is read.
- `CLIENT_ACKNOWLEDGE`, where you manually call the `acknowledge()` method on the `MapMessage` instance as desired.

The appropriate choice for acknowledgment frequency depends on your application logic. Acknowledging after every 100 updates, for example, has been used successfully. Be aware, however, that there is a trade-off. Acknowledgments affect XLA

log holds, and acknowledging too infrequently may result in undesirable log file accumulation. (Also see "XLA bookmarks and transaction log holds" on page 3-5.)

Note: In `DUPS_OK_ACKNOWLEDGE` or `CLIENT_ACKNOWLEDGE` mode, the reader application must have some tolerance for seeing the same set of records more than once.

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` acknowledgment modes are acceptable. Applications that require `CLIENT_ACKNOWLEDGE` acknowledgment mode and applications with high event rates should use the asynchronous interface for receiving updates. They should acknowledge the messages on the callback thread itself if they are using `CLIENT_ACKNOWLEDGE` as acknowledgment mode. See "Receiving and processing updates" on page 3-11.

6

JMS/XLA Reference

This chapter provides reference information for the JMS/XLA API, supported by TimesTen Classic. It includes the following topics:

- JMS/XLA MapMessage contents
- DML event data formats
- DDL event data formats
- Data type support
- JMS interfaces for event handling
- JMS/XLA replication API
- JMS message header fields

> **Note:** "Access control impact on XLA" on page 3-9 introduces the effects of TimesTen access control features on XLA functionality.

JMS/XLA MapMessage contents

A `javax.jms.MapMessage` contains a set of typed name and 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` instance depend on what type of update it is.

XLA update types

Each `MapMessage` returned by the JMS/XLA API contains at least one name and value pair, `__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. Table 6–1 shows the supported types.

Table 6–1 XLA update types

Type	Description
`ADD_COLUMNS`	Indicates that columns have been added.
`COMMIT_FIELD`	This is the name of the field in a message that contains a commit.
`COMMIT_ONLY`	Indicates that a commit has occurred.

Table 6–1 (Cont.) XLA update types

Type	Description
CONTEXT_FIELD	This is the name of the field in a message that contains the context value passed to the ttApplicationContext procedure as a byte array.
CREATE_INDEX	Indicates that an index has been created.
CREATE_SEQ	Indicates that a sequence has been created.
CREATE_SYNONYM	Indicates that a synonym 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_SYNONYM	Indicates that a synonym has been dropped.
DROP_TABLE	Indicates that a table has been dropped.
DROP_VIEW	Indicates that a view has been dropped.
FIRST_FIELD	This is 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.
MTYP_FIELD	This is the name of the field in a message that contains type information.
MVER_FIELD	This is the name of the field in a message that contains the transaction log file number of the XLA record.
NULLS_FIELD	This is the name of the field in a message that contains the list of fields that have null values.
REPL_FIELD	This is the name of the field in a message that contains the flag that indicates that the update was applied by replication.
TBLNAME_FIELD	This is the name of the field in a message that contains the table name.
TBLOWNER_FIELD	This is the name of the field in a message that specifies the table owner.
TRUNCATE	Indicates that a table has been truncated.
TYPE_FIELD	This is the name of the field in a message that specifies the message type.
UPDATE	Indicates that a row has been updated.
UPDATE_DESCRIPTOR_FIELD	This is the name of the field that returns a ttXlaUpdateDesc_t structure as a byte array.
UPDATED_COLUMNS_FIELD	This is 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 and value pairs that indicate the following.

- 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 and value pairs that contain these XLA flags are described in Table 6–2. Each name is preceded by two underscores.

Table 6–2 JMS/XLA flags

Name	Description	Corresponding ttXlaUpdateDesc_t flag
__AGING_DELETE	Indicates that a delete was due to aging. The flag is present only if the XLA update record is due to an aging delete. The XlaConstants constant AGING_DELETE_FIELD represents this flag.	TT_AGING
__CASCADING_DELETE	Indicates that a delete was due to a cascading delete. The flag is present only if the XLA update record is due to a cascading delete. The XlaConstants constant CASCADING_DELETE_FIELD represents this flag.	TT_CASCDEL
__COMMIT	Indicates that this is the last record in a transaction and that a commit was performed after this operation. This is in the MapMessage if TT_UPDCOMMIT is on. The XlaConstants constant COMMIT_FIELD represents this flag.	TT_UPDCOMMIT
__FIRST	Indicates that this is the first record in a new transaction. This is in the MapMessage if TT_UPDFIRST is on. The XlaConstants constant FIRST_FIELD represents this flag.	TT_UPDFIRST
__REPL	Indicates that this change was applied to the database through replication. This is in the MapMessage if TT_UPDREPL is on. The XlaConstants constant REPL_FIELD represents this flag.	TT_UPDREPL
__UPDCOLS	This is only used for UPDATETUP records, indicating that the XLA update descriptor contains a list of columns that were actually modified by the operation. It is specified as a string that contains a semicolon-delimited list of column names and is in the MapMessage only if TT_UPDCOLS is on. The XlaConstants constant UPDATE_COLUMNS_FIELD represents this flag.	TT_UPDCOLS

Note: The XlaConstants interface is in the com.timesten.dataserver.jmsxla package.

Applications can use the MapMessage method itemExists() to determine whether a flag is present, and getBoolean() to determine whether a flag is set. As input, specify the XlaConstants constant that corresponds to the flag, such as XlaConstants.AGING_DELETE_FIELD.

Example 6–1 Check for commit

Equivalent to using TT_UPDCOMMIT in XLA, you can use the following test in JMS/XLA to see whether this is the last record in a transaction and that a commit was performed after the operation.

```
if (MapMessage.getBoolean(XlaConstants.COMMIT_FIELD) ) { // Field is set
   ...
}
```

DML event data formats

Many 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, MapMessage contains two name and value pairs, __TBLOWNER and __TBLNAME. These fields describe the name and owner of the table that is being updated. For example, for a table SCOTT.EMPLOYEES, any related MapMessage contains a field __TBLOWNER with the string value "SCOTT" and a field __TBLNAME with the string value "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" images of the row are available, along with a list of column numbers indicating which columns were modified. 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 and 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 in the __CONTEXT name and value pair in the MapMessage. If no context information is provided, the __CONTEXT value is not 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 6–3 shows the name and value pairs that are in a MapMessage generated for a CREATE_TABLE operation.

Table 6–3 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 and value pairs that describe the column are in the MapMessage.
*_column_name*_TYPE	Integer value representing the data type of this column (from java.sql.Types)
*_column_name*_PRECISION	Integer value containing the precision of this column (for NUMERIC or DECIMAL)
*_column_name*_SCALE	Integer value containing the scale of this column (for NUMERIC or DECIMAL)
*_column_name*_SIZE	Integer value indicating the maximum size of this column (for CHAR, VARCHAR, BINARY, or 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

DROP_TABLE

Messages with __TYPE=2 (XlaConstants.DROP_TABLE) indicate that a table has been dropped. Table 6–4 shows the name and value pairs that are in a MapMessage generated for a DROP_TABLE operation.

Table 6–4 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 6–5 shows the name and value pairs that are in a MapMessage generated for a CREATE_INDEX operation.

Table 6–5 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
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), "T" (range), or "B" (bit map)
UNIQUE	Boolean value indicating whether the index is unique
HASH_PAGES	Integer value representing the number of pages in a hash index (not specified for range 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 6–6 shows the name and value pairs that are in a MapMessage generated for a DROP_INDEX operation.

Table 6–6 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 6–7 shows the name and value pairs that are in a MapMessage generated for a ADD_COLUMNS operation.

Table 6–7 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

Table 6–7 (Cont.) ADD_COLUMNS data provided in update messages

Name	Value
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 and value pairs that describe the column are in the MapMessage.
*_column_name*_TYPE	Integer value representing the data type of this column (from java.sql.Types)
*_column_name*_PRECISION	Integer value containing the precision of this column (for NUMERIC or DECIMAL)
*_column_name*_SCALE	Integer value containing the scale of this column (for NUMERIC or DECIMAL)
*_column_name*_SIZE	Integer value indicating the maximum size of this column (for CHAR, VARCHAR, BINARY, or 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

DROP_COLUMNS

Messages with __TYPE=6 (XlaConstants.DROP_COLUMNS) indicate that a table has been altered by dropping existing columns. Table 6–8 shows the name and value pairs that are in a MapMessage generated for a DROP_COLUMNS operation.

Table 6–8 DROP_COLUMNS data provided in update message

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 and value pairs that describe the column are in the MapMessage.
*_column_name*_TYPE	Integer value representing the data type of this column (from java.sql.Types)

Table 6–8 (Cont.) DROP_COLUMNS data provided in update message

Name	Value
*_column_name*_PRECISION	Integer value containing the precision of this column (for NUMERIC or DECIMAL)
*_column_name*_SCALE	Integer value containing the scale of this column (for NUMERIC or DECIMAL)
*_column_name*_SIZE	Integer value indicating the maximum size of this column (for CHAR, VARCHAR, BINARY, or 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 6–9 shows the name and value pairs that are in a MapMessage generated for a CREATE_VIEW operation.

Table 6–9 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 6–10 shows the name and value pairs that are in a MapMessage generated for a DROP_VIEW operation.

Table 6–10 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 6–11 shows the name and value pairs that are in a MapMessage generated for a CREATE_SEQ operation.

Table 6–11 CREATE_SEQ 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

Table 6–11 (Cont.) CREATE_SEQ data provided in update messages

Name	Value
INCREMENT	A long value indicating the INCREMENT BY option specified for the new sequence
MIN_VALUE	A long value indicating the MINVALUE option specified for the new sequence
MAX_VALUE	A 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 6–12 shows the name and value pairs that are in a MapMessage generated for a DROP_SEQ operation.

Table 6–12 DROP_SEQ 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

CREATE_SYNONYM

Messages with __TYPE=19 (XlaConstants.CREATE_SYNONYM) indicate that a synonym has been created. Table 6–13 shows the name and value pairs that are in a MapMessage generated for a CREATE_SYNONYM operation.

Table 6–13 CREATE_SYNONYM data provided in update messages

Name	Value
OWNER	String value of the owner of the created synonym
NAME	String value of the name of the created synonym
OBJECT_OWNER	String value of the schema of the object for which you are creating a synonym
OBJECT_NAME	String value of the name of the object for which you are creating a synonym
IS_PUBLIC	Boolean value indicating whether the synonym is public
IS_REPLACE	Boolean value indicating whether the synonym was created using CREATE OR REPLACE

DROP_SYNONYM

Messages with __TYPE=20 (XlaConstants.DROP_SYNONYM) indicate that a synonym has been dropped. Table 6–14 shows the name and value pairs that are in a MapMessage generated for a DROP_SYNONYM operation.

Table 6–14 DROP_SYNONYM data provided in update messages

Name	Value
OWNER	String value of the owner of the dropped synonym
NAME	String value of the name of the dropped synonym
IS_PUBLIC	Boolean value indicating whether the synonym was public

TRUNCATE

Messages with __TYPE=18 (XlaConstants.TRUNCATE) indicate that a table has been truncated. All rows in the table have been deleted. Table 6–15 shows the name and value pairs that are in a MapMessage generated for a TRUNCATE operation.

Table 6–15 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 support

This section covers data type considerations for JMS/XLA.

Data type mapping

Table 6–16 lists access methods for the data types supported by TimesTen. For more information about data types, see "Data Types" in *Oracle TimesTen In-Memory Database SQL Reference*.

Table 6–16 Data type mapping

TimesTen column type	Read with MapMessage method...
CHAR(*n*)	getString()
VARCHAR(*n*)	getString()
NCHAR(*n*)	getString()
NVARCHAR(*n*)	getString()
NVARCHAR2(*n*)	getString()
DOUBLE	getString() Can be converted to BigDecimal or to Double by the application.
FLOAT	getString() Can be converted to BigDecimal or to Double by the application.
DECIMAL(*p*,*s*)	getString() Can be converted to BigDecimal or to Double by the application.
NUMERIC(*p*,*s*)	getString() Can be converted to BigDecimal or to Double by the application.
INTEGER	getInt()
SMALLINT	getShort()
TINYINT	getShort()
BINARY(*n*)	getBytes()
VARBINARY(*n*)	getBytes()
DATE	getLong(),getString() The getLong() method returns microseconds since epoch (00:00:00 UTC, January 1, 1970). Can be converted to Date or Calendar by the application.

Table 6–16 (Cont.) Data type mapping

TimesTen column type	Read with MapMessage method...
TIME	getString() Can be converted to Date or Calendar by the application.
TIMESTAMP	getLong(), getString() The getLong() method returns microseconds since epoch (00:00:00 UTC, January 1, 1970). It truncates nanoseconds. Use getString() if you require nanosecond precision. Can be converted to Date or Calendar by the application.
TT_CHAR	getString()
TT_VARCHAR	getString()
TT_NCHAR	getString()
TT_NVARCHAR	getString()
ORA_CHAR	getString()
ORA_VARCHAR	getString()
ORA_NCHAR	getString()
ORA_NVARCHAR2	getString()
VARCHAR2	getString()
TT_TINYINT	getShort()
TT_SMALLINT	getShort()
TT_INTEGER	getInt()
TT_BIGINT	getLong()
BINARY_FLOAT	getFloat()
BINARY_DOUBLE	getDouble()
REAL	getFloat()
NUMBER	getString()
ORA_NUMBER	getString()
TT_TIME	getString()
TT_DATE	getLong(), getString() The getLong() method returns microseconds since epoch (00:00:00 UTC, January 1, 1970).
TT_TIMESTAMP	getLong(), getString() The getLong() method returns microseconds since epoch (00:00:00 UTC, January 1, 1970).
ORA_DATE	getLong(), getString() The getLong() method returns microseconds since epoch (00:00:00 UTC, January 1, 1970).
ORA_TIMESTAMP	getLong(), getString() The getLong() method returns microseconds since epoch (00:00:00 UTC, January 1, 1970). It truncates nanoseconds. Use getString() if you require nanosecond precision.
TT_BINARY	getBytes()
TT_VARBINARY	getBytes()

Table 6–16 (Cont.) Data type mapping

TimesTen column type	Read with MapMessage method...
ROWID	getBytes(), getString()
BLOB	getBytes() **Note**: Information about the LOB value itself is unavailable. LOB fields contain zero-length data or null data (if the value is actually NULL).
CLOB, NCLOB	getString() **Note**: Information about the LOB value itself is unavailable. LOB fields contain zero-length data or null data (if the value is actually NULL).

Data types character set

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 interfaces for event handling

The following standard JMS interfaces are available for JMS/XLA applications. Note that the JMS/XLA API supports only publish/subscribe messaging.

- Connection
- ConnectionFactory
- ConnectionMetaData
- Destination
- ExceptionListener
- MapMessage
- Message
- MessageConsumer
- Session
- Topic
- TopicConnection

- TopicConnectionFactory
- TopicSession
- TopicSubscriber

At the following Java EE location, refer to the javax.jms package documentation for information about these interfaces:

https://javaee.github.io/javaee-spec/javadocs/

JMS/XLA replication API

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

See *Oracle TimesTen In-Memory Database JMS/XLA Java API Reference* for information.

TargetDataStore interface

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

This interface defines the methods shown in Table 6–17.

Table 6–17 TargetDataStore methods

Method	Description
apply()	Applies XLA update descriptor to the target database.
close()	Closes the connections to the database and releases the resources.
commit()	Performs a manual commit.
getAutoCommitFlag()	Returns the value of the autocommit flag.
getConnectString()	Returns the database connection string.
getUpdateConflictCheckFlag()	Returns the value of the flag for checking update conflicts.
isClosed()	Checks whether the object is closed.
isDataStoreValid()	Checks whether the database 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 class

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

JMS message header fields

Table 6–18 shows the JMS message header fields provided by JMS/XLA.

Table 6–18 JMS/XLA header fields

Header	Contents
`JMSMessageId`	Transaction log file number of the XLA record
`JMSType`	String representation of the `__TYPE` field

Index

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X