Oracle TimesTen In-Memory Database Architectural Overview

Release 6.0

B25267-03

For the latest updates, refer to the TimesTen release notes.

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

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

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

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

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

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

March 2006

Printed in the United States of America

Contents

About this Guide

The purpose of this guide is to provide an overview of the TimesTen® system. Readers of this guide will benefit most if they have a basic understanding of database systems.

Conventions used in this guide

TimesTen supports multiple platforms. Unless otherwise indicated, the information in this guide applies to all supported platforms. The term Windows refers to Windows 2000, Windows XP and Windows Server 2003. The term UNIX refers to Solaris, Linux, HP-UX, Tru64 and AIX.

TimesTen documentation uses these typographical conventions:

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

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

If you see...	It means...
`fixed width italics`	Variable; must be replaced
`[ ]`	Square brackets indicate that an item in a command line is optional.
`{ }`	Curly braces indicated that you must choose one of the items separated by a vertical bar (|) in a command line.
`|`	A vertical bar (or pipe) separates arguments that you may use more than one argument on a single command line.
`. . .`	An ellipsis (. . .) after an argument indicates that you may use more than one argument on a single command line.

`%`	The percent sign indicates the UNIX shell prompt.
`#`	The number (or pound) sign indicates the UNIX root prompt.

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

If you see...	It means...
`install_dir`	The path that represents the directory where the current release of TimesTen is installed.
`TTinstance`	The instance name for your specific installation of TimesTen. Each installation of TimesTen must be identified at install time with a unique alphanumeric instance name. This name appears in the install path. The instance name "giraffe" is used in examples in this guide.
`bits` or *`bb`*	Two digits, either 32 or 64, that represent either the 32-bit or 64-bit operating system.
`release` or *`rr`*	Two digits that represent the first two digits of the current TimesTen release number, with or without a dot. For example, 50 or 5.0 represents TimesTen Release 5.0.
`jdk_version`	Two digits that represent the version number of the major JDK release. For example 14 for versions of jdk1.4.
`timesten`	A sample name for the TimesTen instance administrator. You can use any legal user name as the TimesTen administrator. On Windows, the TimesTen instance administrator must be a member of the Administrators group. Each TimesTen instance can have a unique instance administrator name.
`DSN`	The data source name.

TimesTen documentation

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

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

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

Background reading

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

- Hamilton, Cattell, Fisher. *JDBC Database Access with Java*. Reading, MA: Addison Wesley. 1998.

For a Java reference, see:

- Horstmann, Cornell. *Core Java*. Palo Alto, CA: Sun Microsystems Press. 1999.
- For the JDBC API specification, refer to java.sql package in the appropriate Java Platform API Specification.
- If you are working with JDK 1.2, refer to the Java 2 Platform API specification at: `http://java.sun.com/products/jdk/1.2/docs/api/index.html`
- If you are working with JDK 1.3, refer to the Java 2 Platform API specification at: `http://java.sun.com/j2se/1.3/docs/api/index.html`

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

- ***SQL***—*Oracle TimesTen In-Memory Database API and SQL Reference Guide* is a complete reference to TimesTen SQL.

UNIX

For a review of SQL, see:

- Jim Melton and Alan R. Simon. *Understanding the New SQL: A Complete Guide*. San Francisco, CA: Morgan Kaufmann Publishers. 1993.

For information on Unicode, see:

- The Unicode Consortium, *The Unicode Standard, Version 4.0*, Addison-Wesley, 2003.
- The Unicode Consortium Home Page at `http://www.unicode.org`

Technical Support

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

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

Email: timesten-support_us@oracle.com

1 What is TimesTen?

TimesTen is a high performance event-processing software component that enables applications to capture, store, use, and distribute information in real-time, while preserving transactional integrity and continuous availability.

Applications that incorporate TimesTen can process massive transaction volumes and respond instantly to requests using less expensive hardware configurations than would be required by conventional software architectures. TimesTen has been successfully integrated into many applications in telecom and networking, financial services, travel and logistics, and real-time enterprises.

TimesTen is designed to operate most efficiently in an application’s address space. Using standard interfaces, TimesTen can be integrated into an application to serve as either a stand-alone relational database management system (RDBMS) or an application-tier cache that works in conjunction with a traditional disk-based RDBMS, such as the Oracle database. TimesTen can be configured to operate entirely in memory, or it can be configured for disk-based environments to log and checkpoint data to disk.

Why is TimesTen Faster Than a Conventional Database?

In a conventional RDBMS, client applications communicate with a database server process over some type of IPC connection, which adds substantial performance overhead to all SQL operations. An application can link TimesTen directly into its address space to eliminate the IPC overhead and streamline query processing. This is accomplished through a *direct connection* to TimesTen. Client/server connections are also available to applications. From an application’s perspective, the TimesTen API is identical whether it is a direct connection or a client/server connection.

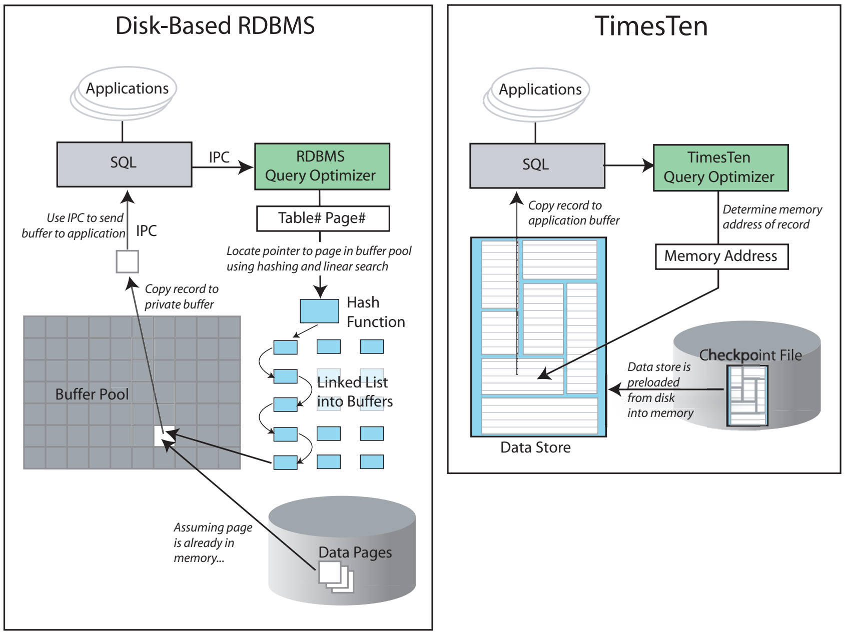
Furthermore, much of the work that is done by a conventional, disk-optimized RDBMS is done under the assumption that data is primarily disk resident. Optimization algorithms, buffer pool management, and indexed retrieval techniques are designed based on this fundamental assumption.

Even when an RDBMS has been configured to hold all of its data in main memory, its performance is hobbled by deeply interwoven assumptions of disk-based data residency. These assumptions cannot be easily reversed because they are hard coded—spanning decades of research and development—within the deepest recesses of RDBMS processing logic, indexing schemes, data access mechanisms, etc.

TimesTen, on the other hand, is designed with the knowledge that data resides in main memory and can therefore take more direct routes to data, reducing codepath length and simplifying both algorithm and structure.

When the assumption of disk-residency is removed, complexity is dramatically reduced. The number of machine instructions drops by at least a factor of ten, buffer pool management disappears, extra data copies aren't needed, index pages shrink, and their structure is simplified. When memory-residency for data is the bedrock assumption, the design gets simpler, more elegant, more compact, and requests are executed faster.

Figure 1.1 Comparing a disk-based RDBMS to TimesTen


Disk-Based RDBMS
Applications
SQL
IPC
RDBMS Query Optimizer
Table# Page#
Use IPC to send buffer to application
IPC
Locate pointer to page in buffer pool using hashing and linear search
Copy record to private buffer
Hash Function
Buffer Pool
Linked List into Buffers
Assuming page is already in memory...
Data Pages
TimesTen
Applications
SQL
TimesTen Query Optimizer
Copy record to application buffer
Determine memory address of record
Memory Address
Checkpoint File
Data store is preloaded from disk into memory
Data Store

Comparing TimesTen to a Conventional Database

TimesTen looks in many ways like other RDBMS systems, so much of its interface and administration should be familiar. This section describes many familiar database features, and indicates where TimesTen is similar to a typical RDBMS and where it differs. In comparing TimesTen to conventional RDBMS systems, there are a number of key differences that clearly illustrate how TimesTen attains a many-fold performance improvement over its disk-based counterparts.

Standard ODBC/JDBC interfaces

TimesTen supports version 2.5 of ODBC and version 1.2 of JDBC. Unlike many other database systems, where ODBC and/or JDBC API support may be much slower than the proprietary interface, ODBC and JDBC are native TimesTen interfaces that operate directly with the database engine. TimesTen supports versions of these APIs that are both fully compliant with the standards and tuned for maximum performance in the TimesTen environment.

See “TimesTen ODBC and JDBC APIs” on page 39 for more information on TimesTen ODBC and JDBC support.

SQL

TimesTen supports a wide range of SQL-92 functionality, as well as SQL extensions to simplify the configuration and management of special features, such as replication, Oracle data caching, and materialized views.

See “SQL-92 standard” on page 40 for more information on SQL support and “SQL Administration” on page 108 for more information on how SQL is used for administrative activities.

Access Control

TimesTen can be installed with an additional layer of user-access control by enabling the *Access Control* feature. TimesTen Access Control uses standard SQL operations to establish TimesTen user accounts with specific privilege levels.

See “TimesTen Access Control” on page 106 for more information on TimesTen Access Control.

Distributed transactions

TimesTen supports distributed transactions through the XA and JTA interfaces. These standard interfaces allow TimesTen to interoperate with transaction managers in distributed transaction processing (DTP) environments.

See “Distributed Transaction Processing (DTP) APIs” on page 40 for more information.

Database connectivity

Like most other database systems, TimesTen supports client/server connections. TimesTen also supports direct driver connections for higher performance, as well as connections through a driver manager for applications that may want to access several types of database systems at the same time.

This multiplicity of connection options allows users to choose the best performance/functionality tradeoff for their applications. Direct driver connections are fastest; client/server connections may provide more flexibility; and driver manager connections can provide support for ODBC applications not written for ODBC version 2.5, as well as support multiple DBMS databases from different vendors simultaneously.

See Chapter 4, “How Applications Connect to Data Stores” for more information on the various ways applications connect to TimesTen.

Logging

TimesTen keeps a log of changes and can optionally write them to disk. The log is used to:

- Redo transactions if the application or data store crashes and recovery is needed
- Undo transactions that are rolled back
- Replicate changes to other TimesTen data stores
- Replicate changes to an Oracle database
- Enable applications to detect changes to tables (using the XLA API)

Unlike many other database systems, applications can specify whether or not logging is enabled. If logging is enabled, then recovery, persistent XLA, and caching Oracle data is available. Applications can specify exactly how often the log is written to disk.

By providing logging options, TimesTen enables applications to fine-tune logging operations to obtain the optimum balance between transaction durability and response time.

See “Logging” on page 70 for more information on logging options and “Durable and non-durable commits” on page 72 for more information on the commit options.

Transaction log monitoring and materialized views

Like several other database systems, TimesTen has an API that allows applications to monitor update activities in order to generate actions outside the database. In TimesTen, this capability is provided by the Transaction Log API (or XLA), which allows applications to monitor update records as they are written to the transaction log and take various actions based on the detected updates. For example, an XLA application may apply the detected updates to another database, which could be TimesTen or a disk-based RDBMS. Another type of XLA application may simply notify subscribers that an update of interest has taken place.

TimesTen provides materialized views that can be used in conjunction with the XLA logging API to enable notification of "events" described by SQL queries.

See Chapter 8, "Event Notification" for more information on the logging API and materialized views.

Checkpoints

Like most database systems, TimesTen has a checkpoint operation that is designed to take place in the background and has very little impact on the database applications. This is called a "fuzzy" checkpoint. TimesTen also has a blocking checkpoint that does not require log files for recovery. Checkpoints are automatic.

As is typical of many database systems, TimesTen maintains two checkpoint files in case a crash should occur in mid-checkpoint. Checkpoints may reside on disks separate from the transaction logs to further minimize the impact of checkpointing on the regular application activity.

See "Checkpointing" on page 73 for more information on checkpointing.

Replication

TimesTen provides a replication subsystem for transmitting transactions between TimesTen systems. While replication is a common component of many database systems, TimesTen replication is highly evolved to enable maximum throughput given application constraints on topology, consistency, and recovery. A very high-speed asynchronous replication mechanism is provided, along with a more synchronous, return service mechanisms that outperform asynchronous replication on many RDBMS systems.

Several replication options are provided for resolving conflicts, recovering failed data stores, and doing online upgrades. This range of replication options allows users to determine the optimum balance between runtime performance, consistency, and failover complexity.

See "Replication" on page 75 for information on replication and "Upgrading TimesTen" on page 110 for information regarding on-line upgrades.

Oracle data caching

Cache Connect enables TimesTen to be used as a cache for an Oracle database. This feature makes it easier for TimesTen applications to store frequently used or more important data in one or more copies of TimesTen, while the remaining bulk of the data is stored in a slower disk-based database. Cache Connect can be configured to automatically propagate transactions from an Oracle database to TimesTen and from TimesTen to an Oracle database. Cached data can be automatically aged out when the caching capacity in TimesTen is exceeded.

Cache Connect is described in more detail in Chapter 9.

Query optimization

Similar to many other database systems, TimesTen has a cost-based query optimizer that chooses the best query plan based on factors such as the ordering of tables and choice of access method.

Optimizer cost sensitivity is somewhat higher in TimesTen than in disk-based systems, as the cost structure of a main-memory system differs from that of disk-based systems in which disk access is a dominant cost factor. Because disk access is not a factor in TimesTen, the optimization cost model includes factors not considered by optimizers for disk-based systems, such as the cost of evaluating predicates.

TimesTen provides two types of indexes (hash and t-tree) and supports two types of join methods (nested-loop and merge-join). The optimizer may create temporary indexes on the fly, as needed.

The TimesTen optimizer also accepts hints that give applications the flexibility to make tradeoffs between such factors as temporary-space usage and performance.

See Chapter 6, “Query Optimization” for more information on TimesTen's query optimizer and indexing techniques.

Concurrency

TimesTen provides full support for shared data stores, as do all database systems. Unlike most systems, however, TimesTen provides several options to allow users to determine the optimum balance among response time, throughput, and transaction semantics for their system. For example, access to a data store, table, or row can be serialized to improve performance.

For data stores with extremely strict transaction semantics, Read committed isolation isolation is available. The default Read committed isolation isolation provides non-blocking operations. These isolation levels conform to the ODBC standards and are implemented with optimal performance in mind, so they avoid copying data whenever possible. As defined by the ODBC standard, a default isolation level can be set for a TimesTen data store, which can be dynamically modified for each connection at runtime.

Finally, TimesTen runs well on machines with one or more processors. Depending on the number of processors available, TimesTen can be configured to trade off response-time for throughput.

For more information on managing concurrent operations in TimesTen, see Chapter 5, "Concurrent Operations."

Administration and utilities

TimesTen supports typical database utilities such as interactive SQL, backup and restore, copy (copies data between different database systems), and migrate (a higher speed copy for moving data between different versions of TimesTen).

Similar to most other database systems, SQL configuration is available for many other administrative activities, such as creating indexes and altering tables. TimesTen also uses SQL configuration to set up replication, Cache Connect, and materialized views. A web-based administrator is also available for setting up Cache Connect.

TimesTen built-in procedures and C language functions enable programmatic control over TimesTen operations and settings. TimesTen command-line utilities allow users to monitor the status of connections, locks, replication, and so on. Status can also be obtained using SQL SELECT queries on the system tables in the TimesTen schema. For example, the TimesTen MONITOR table records many statistics that are of use in analyzing or debugging a TimesTen application.

For more information on TimesTen administration, see Chapter 10, "TimesTen Administration."

2

How is TimesTen Used?

As described in Chapter 1, TimesTen provides many of the capabilities of a general-purpose database, but with a bias toward delivering real-time performance. This chapter describes how TimesTen might be used to enable applications that require real-time access to data.

The main sections in this chapter are:

General Uses for TimesTen

In general, TimesTen can be used as:

- The *primary database* for real-time applications. In this case all data needed by the application(s) resides in TimesTen.
- A *real-time data manager* for specific tasks in an overall workflow in collaboration with disk-based RDBMSs. For example, a phone billing application may capture and store recent call records in TimesTen while storing information about customers, their billing addresses, credit information, and so on in a disk-based RDBMS. It may also age and keep archives of all call records in the disk-based database. Thus, the information that requires real-time access is stored in TimesTen while the information needed for longer-term analysis, auditing, and archival is stored in the disk-based RDBMS.
- A *data utility* for accelerating performance-critical points in an architecture. For example, providing persistence and transactional capabilities to a message queuing system might be achieved by using TimesTen as the repository for the messages.
- A *data integration point* for multiple data sources on top of which new applications can be built. For example, an organization may have large amounts of information stored in several data sources, but only subsets of this information may be relevant to running its daily business. A suitable architecture would be to pull the relevant information from the different data sources into one TimesTen operational data store to provide a central repository for the data of immediate interest to the applications.

Within the context of these different roles, TimesTen might be used to store the following types of data:

- *Reference data*, which is relatively static and used only for validation and enrichment purposes. This data exists in persistent storage.
- *Derived data*, which is constructed from other data and can be re-constructed in case of failure. This type of data is transitory, but differs from the next category in that it is directly accessed by end users.
- *Intermediate/transitory data* that a business or system process creates and needs to manipulate in order to complete its processing. It has no intrinsic value after the process has completed.
- *Data of record*, which describes the business state required for audit-trail, regulatory and MIS purposes. This type of data is traditionally managed by disk-based databases. TimesTen is typically integrated with a back-end database when managing such data.

TimesTen Application Scenarios

This section describes some application scenarios to illustrate how TimesTen might be integrated as part of an overall data management solution.

The application scenarios are:

- "Scenario 1: Caller usage metering application": Uses TimesTen to store metering data on the activities of cellular callers. The metering data is collected from multiple TimesTen nodes distributed throughout a service area and archived in a central disk-based database for use by a central billing application.
- "Scenario 2: Real-time quote service application": Uses TimesTen to store stock quotes from a data feed for access by program trading applications. Quote data is collected from the data feed and published on a real-time message bus. The data is read from the message bus and stored in TimesTen, where it is accessed by the program trading applications.
- "Scenario 3: Online travel agent application": Uses TimesTen as an application-tier cache to hold user profiles and reservation data maintained in a central Oracle database.

In the following discussions, the term *data store* describes the memory segment that contains the TimesTen "database." Data stores are discussed in "TimesTen Data Stores" on page 34.

Note: The company names used in the following application scenarios are fictitious. Each application is not a specific actual customer application but an amalgam of similar applications in a given market segment.

Scenario 1: Caller usage metering application

NoWires Communications, a cellular service provider, has a *usage metering* application that keeps track of the duration of each cellular call and the services used. For example, if a caller makes a regular call, a base rate is applied for the duration of the call. If a caller uses special features, such as roaming, Web browsing, or 3-way calling, extra charges are applied.

The usage metering application must efficiently monitor up to 100,000 concurrent calls, gather usage data on each call, and store the data in a central database for use by other applications that generate bills, reports, audits, and so on.

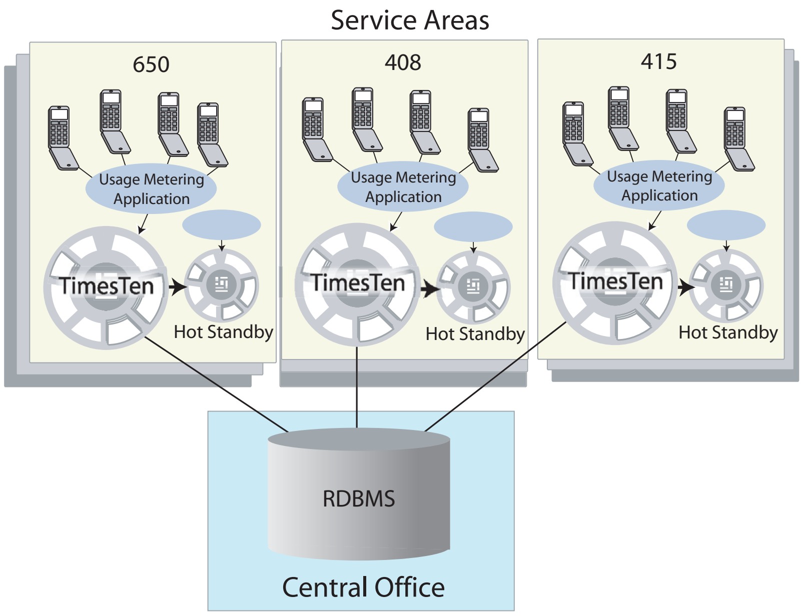
In the current configuration, all of the application data is stored in a *Calls* table in a central disk-based RDBMS database. However, volume has been increasing and the company's infrastructure needs modernization. The company has determined that scaling the central RDBMS database would result in too many connections and too many records in the *Calls* table for acceptable performance.

In addition, the RDBMS database isn't capable of real time updates, which are needed by the usage metering application for maximum performance.

The company determines that performance and scaling could be improved by using TimesTen to store the caller data that is of immediate interest to the usage metering application and to warehouse all of the other data in the central RDBMS database. The company's solution is to decentralize the data gathering operation by distributing multiple instances of the usage metering application and TimesTen on individual nodes throughout its service areas. For maximum performance, each usage metering application connects to its local TimesTen data store by means of an ODBC direct driver connection.

As shown in Figure 2.1, a separate usage metering application and TimesTen node combination is deployed to handle the real-time processing for calls beginning and terminating at different geographical locations delineated by area code. For each call, the local node stores a separate record for the beginning and the termination of a call. This is because the beginning of a cellular call might be detected by one node and its termination by another node.

Figure 2.1 Distributed collection of usage data



In this scenario, TimesTen gains a performance edge over the central RDBMS database because it can manage millions of rows per table with relatively small performance degradation. So each TimesTen node scales well "vertically," and the overall environment scales "horizontally" by adding more TimesTen nodes.

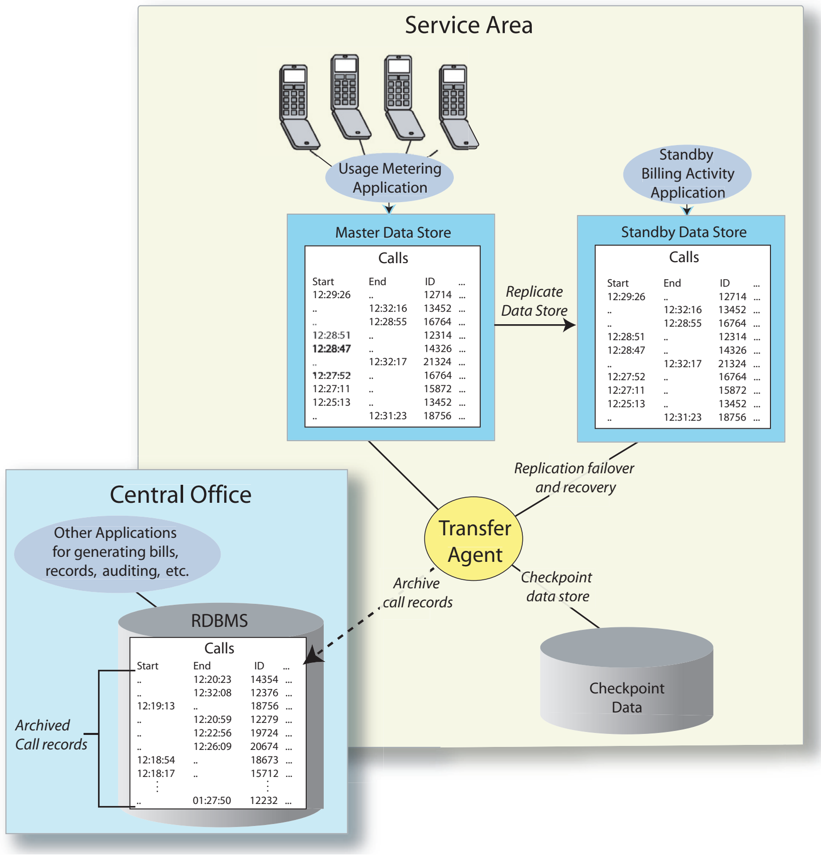
Revenue impacting transactions (inserts/updates) against TimesTen must be durable, so disk-based logging is used. To ensure data availability, each TimesTen node is replicated to a hot standby node.

The company implements a special background component, called a *Transfer Agent*, that runs on each TimesTen node. As shown in Figure 2.2, the Transfer Agent connects locally to TimesTen by means of the ODBC direct driver and to the remote RDBMS database by means of IPC. The purpose of the Transfer Agent is to periodically archive the TimesTen records to the central RDBMS database, as well as to handle replication failover and recovery, manage checkpoints, and update statistical information. In the event of a failure of the main node, the Transfer Agent shifts the callers to a redundant usage metering application on the TimesTen standby node.

Each time a customer makes, receives, or terminates a cellular call, the application inserts a record of the activity into its TimesTen *Calls* table. Each call record includes a timestamp, unique identifier, originating host's IP address, and information on the services used.

The Transfer Agent periodically selects the first 5000 rows in the *Calls* table and uses fast batch operations to archive them to the central RDBMS. (The SQL SELECT's result set is limited to a 'first N' rows value to ensure low impact on the real-time processing of calls and to keep the data being archived to a manageable size.) After the Transfer Agent confirms successful archival of the call records in the central RDBMS database, it deletes them from TimesTen. The interval at which the Transfer Agent archives calls changes dynamically, based on the call load and the current size of the data store.

Figure 2.2 Transfer Agent archiving call records



Master Data Store — Calls

Start	End	ID	...
12:29:26	..	12714	...
..	12:32:16	13452	...
..	12:28:55	16764	...
12:28:51	..	12314	...
12:28:47	..	14326	...
..	12:32:17	21324	...
12:27:52	..	16764	...
12:27:11	..	15872	...
12:25:13	..	13452	...
..	12:31:23	18756	...

Standby Data Store — Calls

Start	End	ID	...
12:29:26	..	12714	...
..	12:32:16	13452	...
..	12:28:55	16764	...
12:28:51	..	12314	...
12:28:47	..	14326	...
..	12:32:17	21324	...
12:27:52	..	16764	...
12:27:11	..	15872	...
12:25:13	..	13452	...
..	12:31:23	18756	...

RDBMS — Calls

Start	End	ID	...
..	12:20:23	14354	...
..	12:32:08	12376	...
12:19:13	..	18756	...
..	12:20:59	12279	...
..	12:22:56	19724	...
..	12:26:09	20674	...
12:18:54	..	18673	...
12:18:17	..	15712	...
⋮		⋮	
..	01:27:50	12232	...

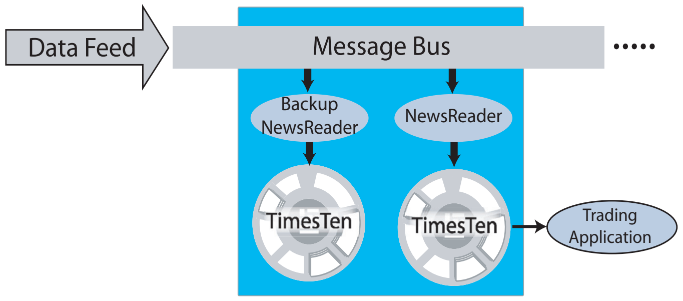
Scenario 2: Real-time quote service application

Beeman&Shuster, a financial services company, is adding a real-time quote and news service to their online trading facility. The immediate requirement of the real-time quote service is to read an incoming news wire from one of the major market data vendors and to make a select subset of the data available to trading applications that manage the automated trading operations for the company. The company's plan is to build an infrastructure that is extensible and flexible enough to accommodate future expansion to provide real-time quotes, news, and other trading services to retail subscribers.

The real-time quote service includes a *NewsReader* process that reads incoming data from a real-time message bus, such as TIBCO SmartSockets® or Rendezvous™, that is constantly fed data from a news wire. Each NewsReader is paired with a backup NewsReader that independently reads the data from the bus and inserts it into a separate TimesTen data store. In this way, the message bus is used to fork incoming data to two TimesTen data stores for redundancy. In this scenario, forking the data from the message bus is more efficient than using TimesTen replication.

Of each pair, one NewsReader makes the stock data available to a trading application, while the other serves as a hot standby backup to be used by the application in the event of a failure. The current load requires four NewsReader pairs, but more NewsReader pairs can be added in the future to scale the service to deliver real-time quotes to other types of clients over the Web, pager, cellular phone, and so on.

Figure 2.3 Capturing feed data from a message bus

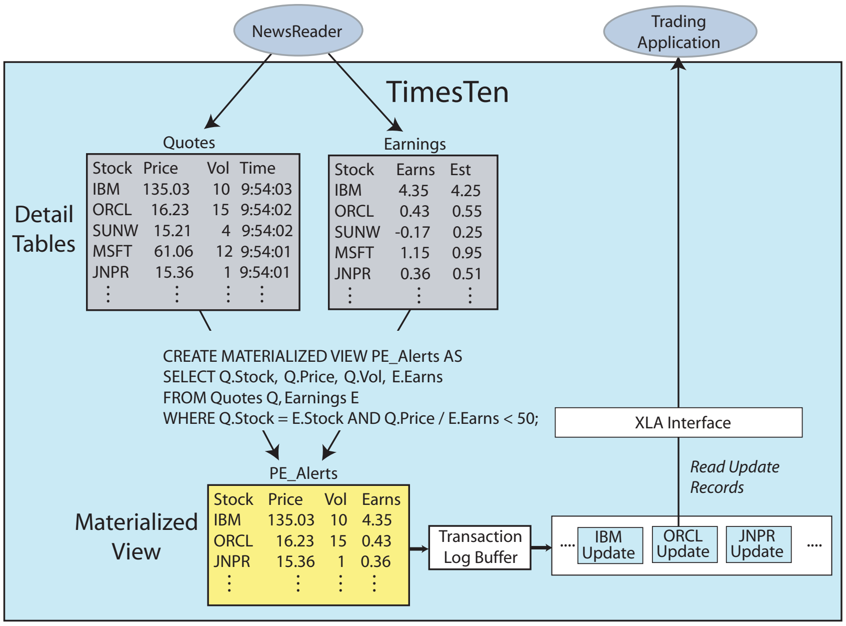


As shown in Figure 2.4, the NewsReader updates stock price data in a *Quotes* table in the TimesTen data store. Less dynamic earnings data is updated in an *Earnings* table. The *Stock* columns in the *Quotes* and *Earnings* tables are linked through a foreign key relationship.

The purpose of the trading application is to track only those stocks with PE ratios below 50, then use some internal logic to analyze the current stock price and trading volume to determine whether to place a trade using another part of the trading facility. For maximum performance, the trading application implements an event facility that uses the TimesTen Transaction Log API (XLA) to monitor the TimesTen transaction log for updates to the stocks of interest.

To provided the fastest possible access to such updates, the company creates a Materialized View, named *PE_Alerts*, with a WHERE clause that calculates the PE ratio from the *Price* column in the *Quotes* table and the *Earns* column in the *Earnings* table. By using the XLA event facility to monitor the transaction log for price updates in the Materialized View, the trading application only receives alerts for those stocks that meet its trading criteria.

Figure 2.4 Use of Materialized Views and XLA



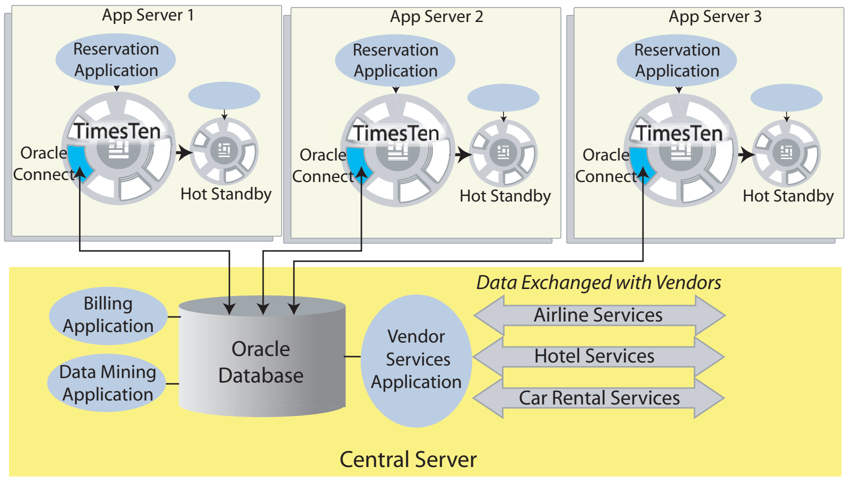
Scenario 3: Online travel agent application

CustomTravel.com, an on-line travel agency, provides Web-based interactive travel services. The company maintains extensive profiles for each customer to track characteristics that include travel history and personal preferences regarding airlines, destinations, rental cars, hotels, and so on. These profiles are used by a *reservation application* to provide each customer with a personalized interactive travel-planning session.

As shown in the lower portion of Figure 2.5, the company's system includes a central server, which hosts back-end applications and an Oracle database that stores the customer profiles and reservation data. The reservation data includes flight schedules, available seats, hotel and car rental availability, and similar information from travel-industry vendors. This reservation data is constantly updated in the Oracle database by a *vendor services application* that manages the communication between the company and its travel vendors.

In order to efficiently manage an expected volume of up to 1000 concurrent customer sessions, the company has distributed a number of application servers throughout its service region, as shown in the upper portion of Figure 2.5. Each application server includes a separate reservation application and caches the customer session data in a local TimesTen data store. To ensure availability, each TimesTen data store is replicated to a hot standby data store.

Figure 2.5 Using Cache Connect to cache Oracle data



The session data in each TimesTen data store includes a copy of the reservation data and selected customer profiles stored in the central Oracle database. This Oracle data is cached in TimesTen *cache groups* and is propagated between TimesTen and Oracle using the TimesTen "Cache Connect to Oracle" feature.

TimesTen "Cache Connect to Oracle" periodically refreshes the reservation data from the central Oracle database to a cache group in the TimesTen data store. Customer profiles are cached in another cache group. When a customer logs on, the reservation application loads that customer's profile from the central Oracle database as a cache instance in the customer profiles cache group. After querying the customer for initial information, such as destination and date, the reservation application uses the customer's profile in the TimesTen cache to formulate SELECT queries that "filter" the reservation data in a manner that provides the customer with the data that is of most interest.

When a customer completes a session, the final selections of the flight, hotel, and so on are flushed from TimesTen to Oracle. "Cache Connect to Oracle" ages out the cache instances for non-active customer profiles from TimesTen to Oracle, as space is needed.

All of the final reservation and scheduling data is stored in the Oracle database. The Oracle database is much larger than each TimesTen data store and is best accessed by applications that do not require the real-time performance of TimesTen but do require access to large amounts of data. Such applications include the company's *vendor services application*, which updates the reservation data supplied by the travel vendors and confirms the final reservations supplied by TimesTen with the vendors, and a *billing application*, which charges the customer's account for the selected services and generates an invoice. Another Oracle application is the company's *data mining application,* which analyzes the historical data to determine reservation patterns as they relate to time of year, time of week, duration of travel, and number of family members traveling together. The results of this application are used to improve the overall revenue of the company by offering travel discounts during periods of infrequent travel.

Application Design Considerations

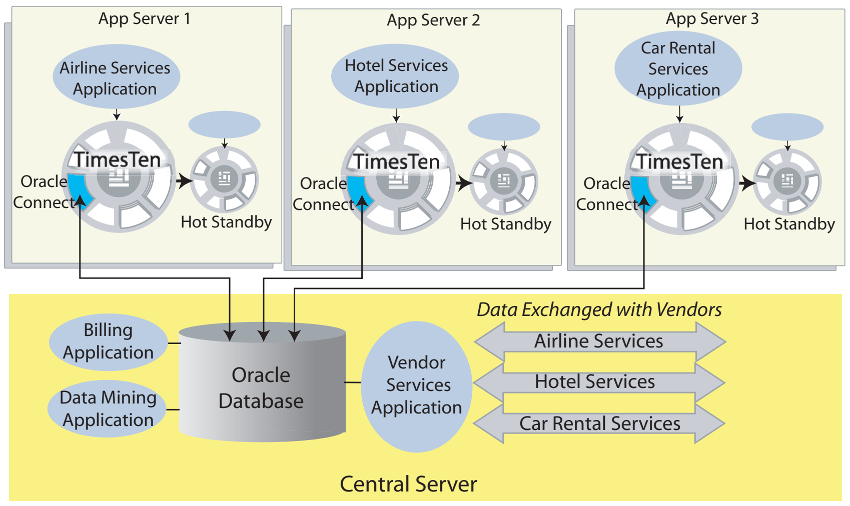
TimesTen is designed to offer maximum flexibility to accommodate a wide range of application architectures. For example, consider the following alternatives to the online travel agent application described in "Scenario 3: Online travel agent application" on page 25.

Alternative 1

To efficiently manage the large number of concurrent customer sessions, *CustomTravel.com* could have chosen an alternate, more centralized architecture for its online travel agent application, such as the one shown in Figure 2.6.

In this alternative architecture, the company implements separate applications to provide the airline services, hotel services, and car rental services. Each application runs on a separate server at a central location and caches only the data relevant to the application. This architecture has the same benefits as the one described in the original scenario, but uses a more centralized approach to data management.

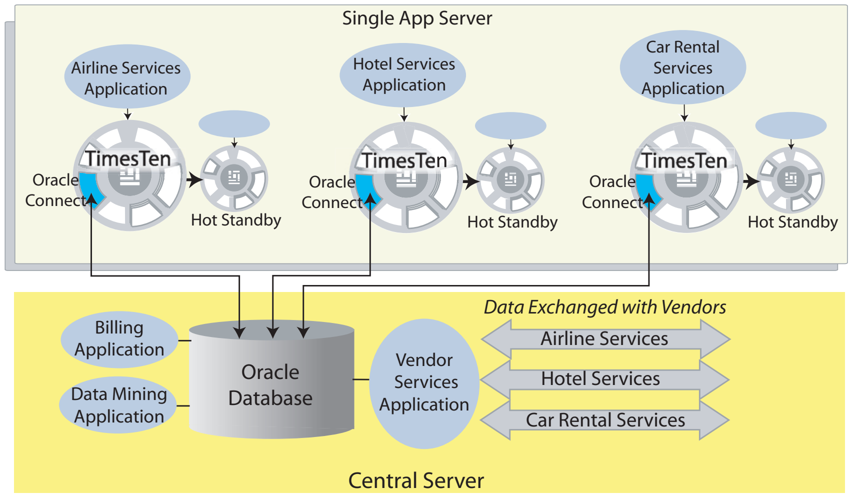
Figure 2.6 Centralized TimesTen caches on multiple co-located servers



Alternative 2

The three applications in Scenario 3 could run on the same server, as shown in Figure 2.7, if the server has enough processing power to handle all three applications.

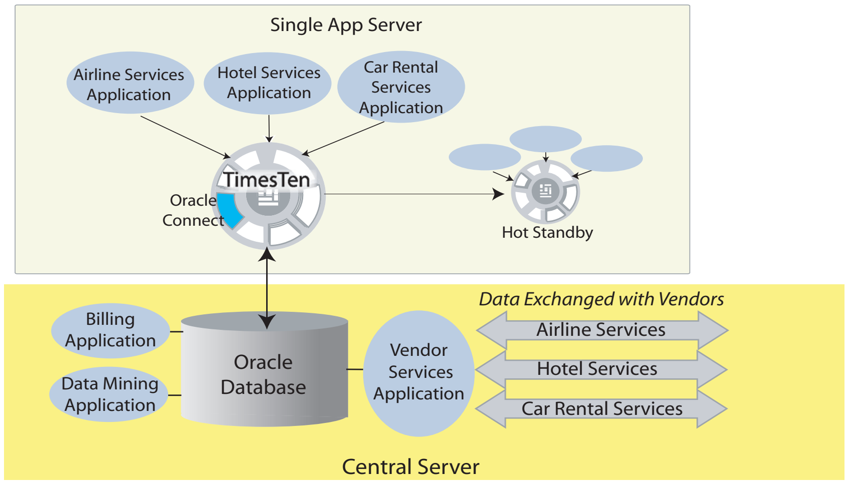
Figure 2.7 Centralized TimesTen caches on the same server



Alternative 3

If all three applications in Scenario 3 were to run on the same server, they could also access the same TimesTen data store, as shown in Figure 2.8. A single data store could contain the union of data needed by the three applications.

Figure 2.8 Centralized, shared TimesTen cache on the same server



Architectural tradeoffs

The original Scenario 3 design and the alternative designs described above illustrate the following architectural tradeoffs:

- □Geographically distributed servers, as described in the original discussion of Scenario 3, can provide better response time for local applications and end users. They can also provide better fault tolerance because multiple servers in a centralized location can all suffer from the same failure, such as a power outage or a fire.
- □Multiple servers at the same site, as shown in Alternative 1, may provide more fault tolerance than one large centralized server, but they also require more administrative overhead and more coordination.
- □Multiple specialized applications can provide better fault tolerance to software failures than a single more general-purpose application. For example, in Alternative 1, a bug in the Airline Services Application will not affect the Car Rental Services Application, while in the original scenario, since all reservation services are combined into one application, bugs in one of the applications can affect all of the services on that node.
- □Multiple specialized applications with different, but potentially overlapping data stores (Alternative 2), can provide better performance than multiple applications sharing one data store (Alternative 3) as each application may have different access patterns and may require different indexes on the data.
- □A single, shared data store described in Alternative 3 has some advantages and some disadvantages. On the positive side, it simplifies data management as all policies for when/how to transfer data to/from TimesTen and Oracle can be handled in one place. It can also save on main memory consumption since multiple caches may contain redundant data. However, performance and scalability may be better with multiple data stores. Furthermore, failure of the one-shared data store may affect all applications. On the other hand, a standby data store can provide rapid failover should such a failure occur.

Summary of design options

Connection to TimesTen

For best performance, use a direct connection to the TimesTen data manager, as described in "Direct Driver Connection" on page 44. Direct connections avoid the high cost of context switches and IPC that are associated with Client/Server connections.

Operation with a disk-based RDBMS

TimesTen is appropriate for managing relatively small to moderate amounts of data (from single-digit to the tens of gigabyte range), while a disk-based RDBMS like the Oracle database is more appropriate for managing large amounts of data (in the terabyte range).

TimesTen can work in partnership with the Oracle database, where TimesTen manages operational data that requires high performance while the Oracle database manages historical data. This was shown in Scenario 1: Caller usage metering application and Scenario 3: Online travel agent application. Several options are available when TimesTen is used with the Oracle database:

- The transfer of data between TimesTen and the Oracle database may either be handled by the application (as shown in Scenario 1) or by the TimesTen "Cache Connect to Oracle" feature (as shown in Scenario 3).
- TimesTen may be used to collect rapidly arriving new data that is later transferred in batches to a historical Oracle database (as shown in Scenario 1).
- TimesTen may be used to manage frequently accessed subsets of a much larger database (as shown in Scenario 3).
- The operational data may be managed in several TimesTen data stores or in a single data store. In the case of multiple data stores, the data stores may be overlapping or may be completely disjoint. Multiple data stores provide the advantage of scalability, fault-tolerance and potentially better tuning for specific application needs, as discussed in the alternatives to Scenario 3.

Data partitioning

Data partitioning may be used to scale applications for higher throughput. This is shown in Scenario 1: Caller usage metering application, where the caller information is captured in different nodes. This partitioning allows the application to scale infinitely as more nodes may be added when the capacity of existing nodes is saturated. Such partitioning also provides geographic flexibility, where different nodes can be used to capture caller data as the caller changes location between the beginning and the end of the call.

Data availability

In all three scenarios described in "TimesTen Application Scenarios" on page 19, continuous availability to data managed by TimesTen is essential. This is achieved by always maintaining a standby copy of the data in another TimesTen data store. In Scenario 1: Caller usage metering application and Scenario 3: Online travel agent application, the standby copy is maintained by TimesTen's replication feature, while in Scenario 2: Real-time quote service application, the standby copy of the data store is maintained by the application through redundant messaging and through the redundant processing of messages on both the master and the standby.

TimesTen's replication provides an asynchronous and more synchronous-like *return service* options. In the event of a failure of the master data store, TimesTen replication enables rapid failover to a standby data store. In Scenarios 1 and 3, applications run against the master data store. TimesTen automatically replicates all updates from the master data store to the standby data store. If the master data store fails, an OS-specific cluster manager can detect the failure and automatically switch all activities to the applications running on the standby data store.

In environments where updates are delivered to the application by messaging middleware, the application may choose to implement high availability through redundant messaging. In this case, the messaging middleware delivers messages to both the primary and the standby. The applications on both the primary and standby apply updates to their respective data stores. Read requests may be handled by either the primary only or by both the primary and the standby. Again, in case of failure, a cluster manager can detect the failure and can automatically route requests to the surviving data store.

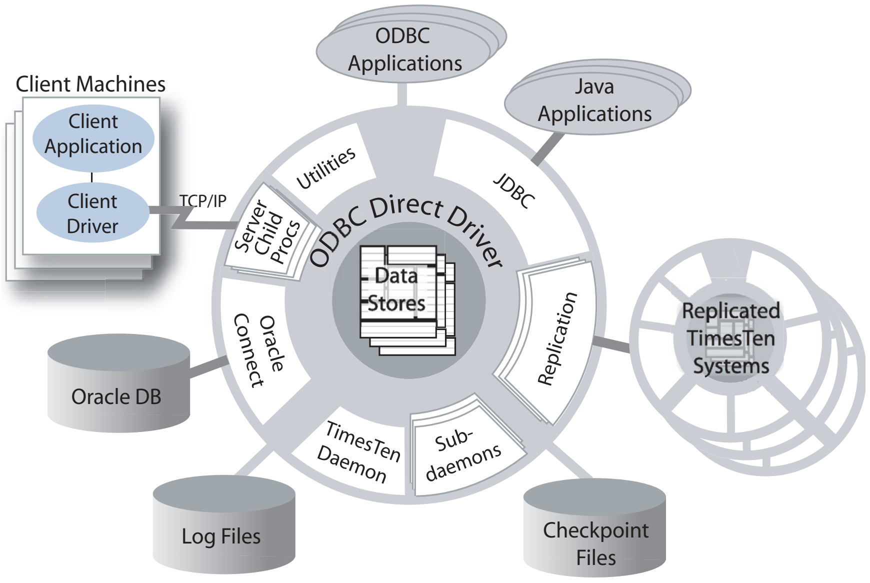
3

Anatomy of a TimesTen System

As shown in the middle of Figure 3.1, a TimesTen "database" (or *data store*) is maintained at runtime in a segment of shared memory that contains all of the tables, indexes and related data structures needed to manage the data in the TimesTen system.

The TimesTen *ODBC direct driver* serves as the "database engine" that manages the interaction between applications and the data stores. As shown in Figure 3.1, ODBC applications running on the same machine as TimesTen can obtain maximum performance by connecting directly to the ODBC direct driver. Java applications access the direct driver through the JDBC library. Client applications running on remote machines communicate over TCP/IP connections with TimesTen *server child processes* that, in turn, access the direct driver.

Figure 3.1 TimesTen Components



As shown in Figure 3.1, TimesTen provides disk-based logging and checkpointing facilities. Applications can also fine-tune logging and checkpointing to make tradeoffs between performance and durability by various degrees.

Other TimesTen features include *replication* support for enhanced data availability and *Cache Connect* to enable TimesTen to operate as a cache for Oracle data.

The remainder of this chapter describes the main TimesTen components:

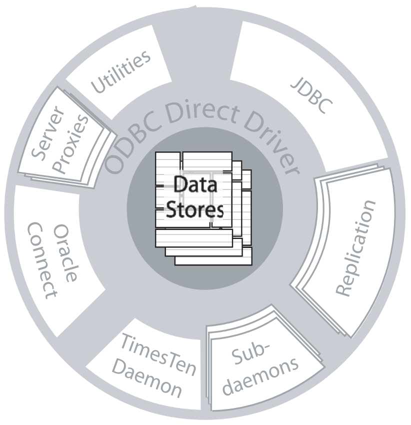
- TimesTen Data Stores
- TimesTen Data Manager
- TimesTen ODBC and JDBC APIs

The other components are described in subsequent chapters.

TimesTen Data Stores

A TimesTen data store is managed at runtime in a segment of memory that contains a collection of TimesTen tables and indexes. The TimesTen Data Manager can manage multiple data stores.

Figure 3.2 TimesTen Data Stores



Each data store is identified by a logical name and a set of attributes that define its configuration. Under most circumstances, the name and attributes of a data store are defined in an ODBC *data source name* (DSN).

User and system DSNs

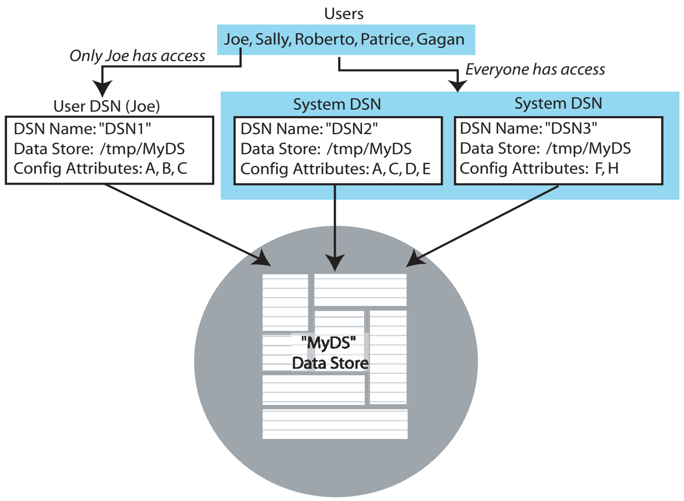
Each DSN uniquely identifies a data store. However, a data store can be referenced by multiple DSNs to define different connection configurations to that data store. This allows users to give convenient names to different connection configurations for the same data store. Figure 3.3 shows a TimesTen data store identified by three DSNs, each with unique configuration attributes.

DSNs are resolved using a two-tiered naming system, consisting of user DSNs and system DSNs:

- A **user DSN** can be used only by select users. Although a user DSN is private to select users, it is only the DSN (the character-string name and its attributes) that is private. The underlying data store can be referenced by other user or system DSNs.
- A **system DSN** can be used by any user on the machine on which the system DSN is defined.

For example, Figure 3.3 shows a data store, named *MyDS*, that is referenced by three DSNs, each with its own unique configuration attributes for the data store. *DSN1* is a user DSN belonging to the user *Joe*; while *DSN2* and *DSN3* are system DSNs that can be accessed by any of the users, including *Joe*.

Figure 3.3 Multiple DSNs for a TimesTen Data Store

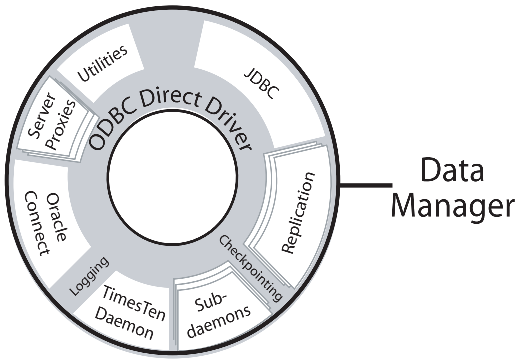


When looking for a specific DSN, TimesTen first looks for a user DSN with the specified name. If no matching user DSN is found, TimesTen looks for a system DSN with the specified name. If a user DSN and a system DSN with the same name exist, TimesTen selects the user DSN.

TimesTen Data Manager

As shown in Figure 3.4, the TimesTen *Data Manager* describes all of the TimesTen processes and libraries and is responsible for processing ODBC and JDBC function calls and SQL statements issued by applications on data stores.

Figure 3.4 TimesTen Data Manager



At the core of the data manager is the ODBC *direct driver*, which is a library of standard ODBC routines and core TimesTen routines that implement SQL, logging, checkpointing, locking, failure recovery, and so on. As described in "Direct Driver Connection" on page 44, ODBC applications can gain a significant performance edge by communicating directly with a data store through the direct driver.

TimesTen processes

The TimesTen data manager includes a number of processes that access the ODBC direct driver to provide specific TimesTen services.

The TimesTen processes are:

- TimesTen Daemon
- TimesTen Subdaemons
- TimesTen Server Daemon and Server Child Processes
- Cache agents
- Replication Agents

TimesTen Daemon

There is one *TimesTen daemon* for each TimesTen installation. The TimesTen daemon is a single multithreaded process that is automatically started when the machine boots. One thread is created for each application connection to TimesTen to implement communication and failure detection.

In order for an application or server process to establish a connection to a TimesTen data store, it first contacts the ODBC or JDBC driver to establish a TCP socket connection with the TimesTen daemon running on that machine.

This socket is used for two purposes:

- The TimesTen ODBC/JDBC driver, running as part of the application's process (packaged as a shared library / DLL), communicates with the TimesTen daemon using the socket. This connection is used to implement much of the logic involved in connecting to and disconnecting from a data store. For example, applications contact the TimesTen daemon via the socket connection to obtain the id of a data store memory segment.
- The TimesTen daemon uses the socket as a failure detection mechanism. If an application that is connected to one or more data stores exits (either voluntarily or involuntarily), the socket connecting that application to the TimesTen daemon is automatically closed by the operating system. This alerts the TimesTen daemon that the application has exited, so it can initiate the necessary recovery processes.

Note: The TimesTen daemon simply manages application connections to the data store and plays no role in processing queries.

TimesTen Subdaemons

The TimesTen daemon does not directly access the individual data stores. Instead, the TimesTen daemon assigns a separate *subdaemon* to each shared data store to coordinate the access of multiple applications to the data store. An application that has an exclusive connection to a data store manages the data store directly, without any intervention from a subdaemon.

When the system starts, the TimesTen daemon starts a predetermined configurable number of subdaemons. When a data store is loaded into memory, the TimesTen daemon allocates one of its free subdaemons to manage that data store.

The subdaemon assigned to manage a shared data store performs a number of operations that include:

- Loading the data store into memory from a checkpoint file on disk (if present).
- Periodically checking for deadlocks and canceling SQL operations on behalf of other applications in order to break any deadlocks.
- Performing the final checkpoint operation when the data store is unloaded from RAM.
- Recovering the data store from checkpoint files and log files after failures, as described in "Recovery from log and checkpoint files" on page 75.
- Performing periodic checkpoints

TimesTen Server Daemon and Server Child Processes

The *TimesTen server daemon* manages incoming connections from remote client applications. Its behavior is much like the TimesTen daemon for local applications, only it allocates a *server child* process to access the data store on behalf of the client.

See "Client/Server Connection" on page 46 for more information.

Cache agents

If Cache Connect is enabled, a *cache agent* process is used to automatically propagate updates from the Oracle database to the TimesTen data store and to "age out" unused data from the TimesTen cache.

See Chapter 9, "Cache Connect to Oracle" for more information.

Replication Agents

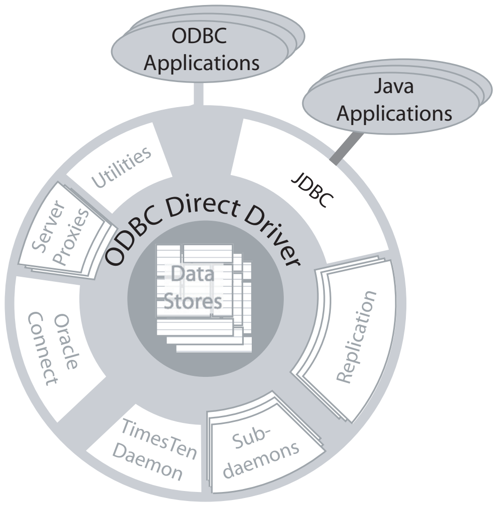
If TimesTen data stores are configured for replication, *replication agents* are used to copy updates between the replicated data stores. When an application updates the master data store, its replication agent detects the update and forwards it to another replication agent on the subscriber data store. There is one replication agent for each data store configured in a replication scheme.

See "Replication" on page 75 for more information.

TimesTen ODBC and JDBC APIs

The runtime architecture of TimesTen supports connectivity through the ODBC and JDBC APIs, which allow applications to access TimesTen data using SQL-92 as the standard data access language. ODBC and JDBC are supported by most DBMS vendors and are widely adopted by DBMS application developers because they provide a standard, vendor-independent interface to the most commonly used database systems.

Figure 3.5 TimesTen ODBC and JDBC APIs



TimesTen provides built-in procedures that extend the standard ODBC and JDBC functionality for TimesTen-specific operations. TimesTen also provides specialized APIs to support applications that manage distributed transactions or monitor the transaction log.

The remainder of this section describes the:

- SQL-92 standard
- TimesTen built-in procedures
- Distributed Transaction Processing (DTP) APIs
- Transaction Log API

SQL-92 standard

TimesTen supports the SQL-92 standard, through which the TimesTen Data Manager provides full relational semantics. Applications pass SQL statements to TimesTen through the ODBC or JDBC interface.

TimesTen SQL support includes standard Data Definition Language (DDL) statements (such as CREATE TABLE, ALTER TABLE, and DROP TABLE) to define database objects, and Data Manipulation Language (DML) statements (such as SELECT, INSERT, UPDATE and DELETE) to query and manipulate the data within the database.

TimesTen SQL expressions include arithmetic and string functions (MOD, CONCAT, UPPER), conversion functions (TO_DATE, TO_CHAR), and aggregates (AVG, COUNT, MAX, MIN, SUM). Subqueries may be used in predicates in SELECT, UPDATE, and DELETE. TimesTen has support for ROWID and SEQUENCEs. A MATERIALIZED VIEW feature propagates changes synchronously to view tables. A TimesTen-specific FIRST '*n*' rows extension to the SQL syntax allows halting SELECT/UPDATE/DELETE after the first few rows. SELECT support includes GROUP BY, HAVING, and ORDER BY.

TimesTen also provides extended SQL to support TimesTen-specific functionality, such as cache group and replication configuration, as described in "Replication" on page 75 and "Cache Groups" on page 94.

TimesTen built-in procedures

TimesTen provides built-in procedures that extend standard ODBC and JDBC functionality to provide applications with programmatic control over TimesTen-specific operations such as initiating checkpoints, setting lock levels, obtaining status information, and so on. The TimesTen built-in procedures are called through the ODBC or JDBC API. TimesTen does not support user-defined stored procedures.

Distributed Transaction Processing (DTP) APIs

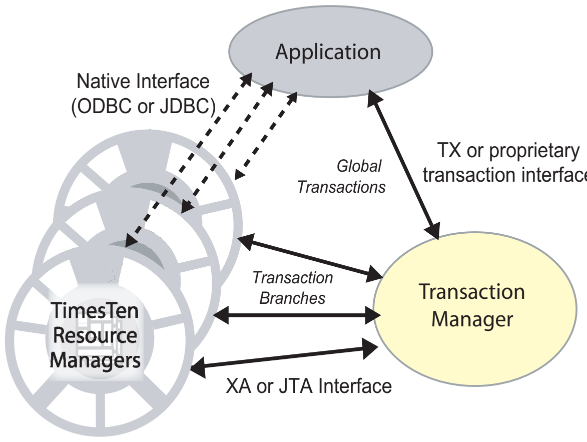
TimesTen implements the X/Open XA Specification and its Java derivative, the Java Transaction API (JTA).

The TimesTen implementation of the XA interfaces is intended for use by transaction managers in distributed transaction processing (DTP) environments. These interfaces can be used to write a new transaction manager or to adapt an existing transaction manager to operate with TimesTen resource managers.

The TimesTen implementation of the Java JTA interfaces is intended to enable Java applications, application servers, and transaction managers to use TimesTen resource managers in DTP environments.

Figure 3.6 illustrates the interfaces defined by the X/Open DTP model.

Figure 3.6 Distributed Transaction Processing Model


Application
Native Interface (ODBC or JDBC)
Global Transactions
TX or proprietary transaction interface
Transaction Branches
TimesTen Resource Managers
Transaction Manager
XA or JTA Interface

In the DTP model, a *transaction manager* breaks each *global transaction* down into multiple *branches* and distributes them to separate *resource managers* for service. In the context of TimesTen XA, the resource managers can be a collection of TimesTen data stores, or data stores in combination with other commercial databases that support XA.

As shown in Figure 3.6, applications use the *TX* interface to communicate global transactions to the transaction manager. The transaction manager breaks the global transaction down into branches and uses the *XA* interface to coordinate each transaction branch with the appropriate resource manager.

Global transaction control provided by the TX and XA interfaces is distinct from local transaction control provided by the native ODBC and JDBC interfaces. An application can maintain separate connections for local and global transactions or initiate both local and global transactions over the same connection.

Note: The TimesTen implementation of the X/Open XA Specification is fairly generic and should be used only as needed, as the semantics of XA restrict the performance of distributed transactions.

Transaction Log API

The Transaction Log API (XLA) allows applications to detect changes made to specified tables in a local data store. XLA also provides functions that can be used by applications to apply changes detected in one data store to another data store. See "Transaction Log API" on page 85 for more information.

For More Information

For more information on TimesTen data stores, see Chapter 1, “Working with TimesTen Data Stores” and “Chapter 6, “Working with Data in a TimesTen Data Store” in the *Oracle TimesTen In-Memory Database Operations Guide*.

For more information on the TimesTen Data Manager, see “Chapter 4, “Working with the Oracle TimesTen Data Manager Daemon” in the *Oracle TimesTen In-Memory Database Operations Guide*.

For more information on the TimesTen APIs, see the *TimesTen C* and *Java Developer Guides* and the *Oracle TimesTen In-Memory Database API and SQL Reference Guide*.

4

How Applications Connect to Data Stores

Applications can connect to a data store in one of three ways:

- Direct Driver Connection
- Client/Server Connection
- Driver Manager Connection

Once connected, applications can communicate with the TimesTen data store by means of the JDBC or ODBC APIs, as described in "TimesTen ODBC and JDBC APIs" on page 39.

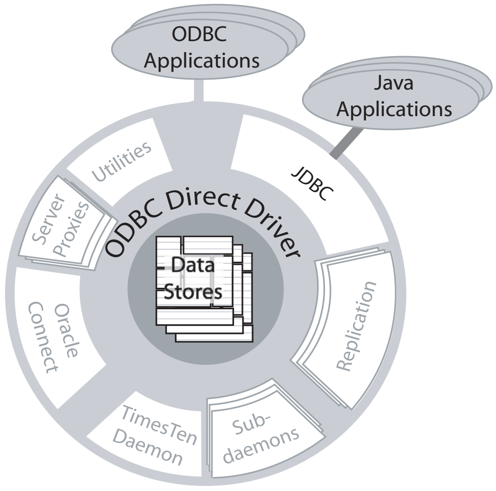
The TimesTen Data Manager supports multithreaded applications, so that multiple application threads can connect to the same or to different TimesTen data stores simultaneously. Each connection to a data store is represented by an *ODBC connection handle or* a *JDBC connection object*.

Note: Client/server and driver manager connections add performance overhead to TimesTen. The performance overhead for client/server connections can be significant.

Direct Driver Connection

In a traditional database system, TCP/IP or another IPC mechanism is used by client applications to communicate with a database server process. All exchanges between client and server are sent over a TCP/IP connection. This IPC overhead adds substantial cost to all SQL operations and can be avoided in TimesTen by connecting the application directly to the TimesTen ODBC *direct driver*.

Figure 4.1 Direct driver connection



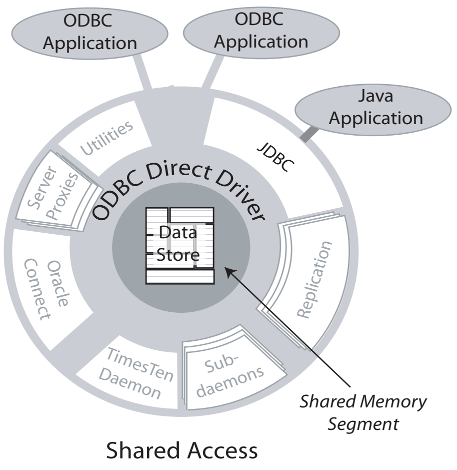
As described in "TimesTen Data Manager" on page 36, the ODBC direct driver is a library of ODBC and TimesTen routines that implement the database engine used to manage the data stores. Java applications access the ODBC direct driver through the JDBC library, as shown in Figure 4.1.

An application can create a direct driver connection when it runs on the same machine as the TimesTen data store. In a direct driver connection, the ODBC driver directly loads the TimesTen data store into either the application's heap space or a shared memory segment, as described in "Shared data stores" on page 45. The application then uses the direct driver to access the memory image of the data store. Because no inter-process communication (IPC) of any kind is required, a direct-driver connection provides extremely fast performance and is the preferred way for applications to access TimesTen data stores.

Shared data stores

When creating a direct driver connection to a data store, an application can accesses the data store in *shared mode*. In shared mode, the data store is loaded into a shared memory segment, where it can be shared by multiple applications and accept multiple connections from each.

Figure 4.2 Shared mode access

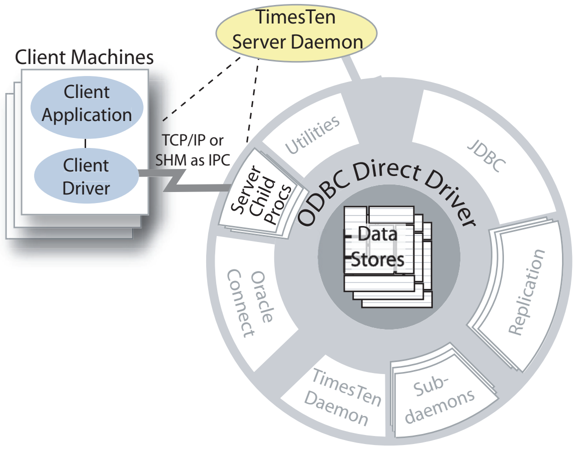


All applications connected to a shared data store use the same shared memory segment. As a result, changes made from one application are instantly visible to all other applications once the first application commits the transaction.

Client/Server Connection

The TimesTen *client driver* and *server daemon* processes accommodate connections from remote client machines to data stores across a network. The server daemon spawns a separate *server child process* for each client connection to the data store, as shown in Figure 4.3

Figure 4.3 Client/server connection to a data store



Applications on a client machine issue ODBC calls to a local ODBC *client driver* that communicates with a server child process on the TimesTen server machine. The server child process, in turn, issues native ODBC requests to the ODBC direct driver to access the TimesTen data store.

If the client and server reside on separate nodes in a network, they communicate by means of sockets and TCP/IP. If both the client and server reside on the same machine, they can communicate more efficiently by means of a shared memory segment (SHM) as IPC.

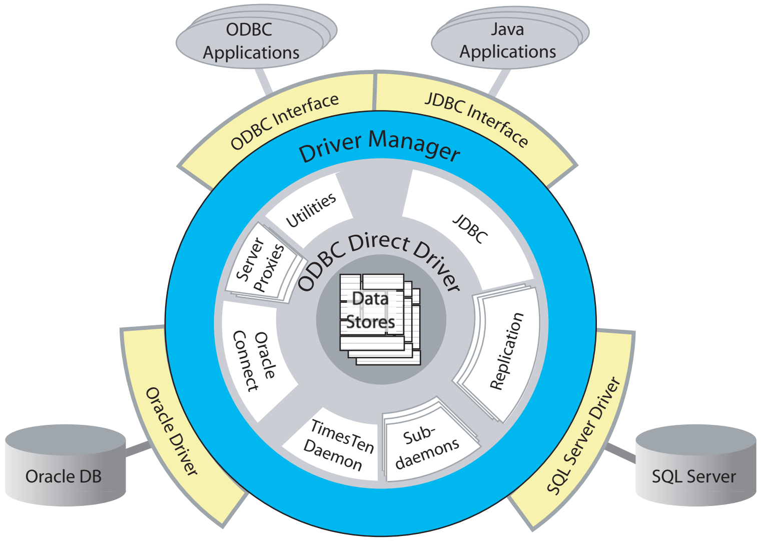
Traditional database systems are typically structured in this client/server model, even when the application and the database are on the same system. Client/server communication adds extra cost to all database operations, so it should only be used in TimesTen for applications that are not performance sensitive.

Driver Manager Connection

Applications can connect to TimesTen through an ODBC *driver manager*, which is a database-independent interface that adds a layer of abstraction between the applications and the TimesTen data store. In this way, the driver manager allows applications to be written to operate independently of the data store and to use interfaces that are not directly supported by TimesTen. The use of a driver manager also enables a single process to have both direct and client connections to the data store.

On Microsoft Windows systems, applications can connect to the MS ODBC driver manager to make use of a TimesTen data store along with data sources from other vendors, such as Oracle. Driver managers for UNIX systems are available as open-source software, such as unixODBC, as well as from third-party vendors.

Figure 4.4 Driver manager connections



For More Information

For more information on connecting to data stores, see Chapter 2, "Creating TimesTen Data Stores" and Chapter 3, "Working with the TimesTen Client and Server" in the *Oracle TimesTen In-Memory Database Operations Guide*.

5

Concurrent Operations

As described in “Shared data stores” on page 45, a data store can be accessed in either exclusive mode or shared mode. When a shared data store is accessed by multiple transactions, there must be a means to coordinate concurrent changes to and scans of the same data in the data store.

TimesTen uses two tools to coordinate concurrent access to data:

- “Locks”, which are reservations placed on data store objects, such as rows, tables, or the entire data store. Locks give a precise specification of the reservation (different levels of sharing); provide first-come, first-served ordering; and provide deadlock detection and timeout. Locks are designed to be held for a relatively long time, such as across application interactions.
- “Latches”, which are simple low-level concurrency tools that operate quickly. In contrast to locks, latches are not designed to be held across application interactions and should be invisible to all customer applications.

Applications can choose one of two transaction isolation levels for a connection to control how locks are placed on fetched rows. Isolation levels are described in “Transaction Isolation” on page 53.

TimesTen also provides an SMP scaling attribute that can be used to adjust latches to gain maximum performance when running applications on multi-processor machines. SMP scaling is described in “For More Information” on page 55.

Locks

Locks are high-level constructs used to serialize access to resources in order to prevent one user from changing an element that is being read or changed by another user. TimesTen automatically performs locking if a data store is accessed in shared mode.

Applications can select from three lock levels:

- Data store-level locking
- Table-level locking
- Row-level locking

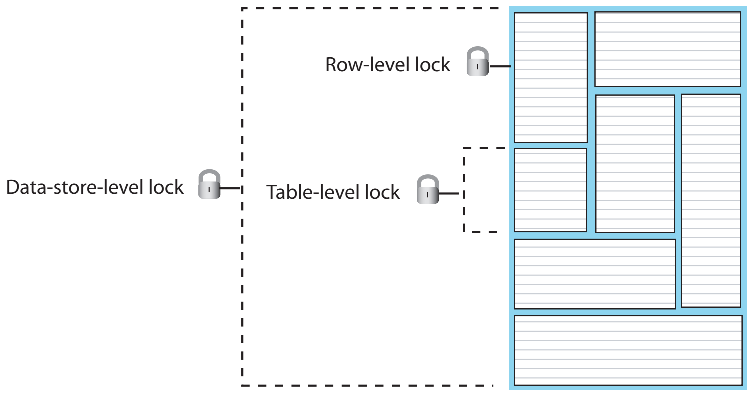
TimesTen provides a **LockLevel** configuration attribute and a **ttLockLevel**() procedure to either lock the entire data store or to direct the optimizer to choose whether to place locks on a row or on the entire data store. Applications can optionally call the **ttOptSetFlag**() procedure to suggest that the query optimizer place locks on tables. Row- and table-level locking require that logging be enabled. Only data store-level locking is possible if logging is disabled.

Serializable transactions acquire share locks on the items the read and exclusive locks on the items they write. These locks are held until the transaction commits or rolls back. Read-committed transactions acquire exclusive locks on the items they write and hold these locks until the transactions are committed. Read-committed transactions do not acquire locks on the items they read. Committing or rolling back a transaction closes all cursors and releases all locks held by the transaction. (See "Transaction Isolation" on page 53 for a discussion of serializable and read committed isolation levels.)

TimesTen performs deadlock detection to report and eliminate deadlock situations. If an application is denied a lock because of a deadlock error, it should roll back the entire transaction and retry the transaction.

An application can use the **ttLockWait**() procedure to control how many seconds it is willing to wait for a lock that is in use by another connection.

Figure 5.1 Lock Levels



Data store-level locking

Locking at the data store level locks the entire data store when accessed by a transaction, thus prohibiting access by other transactions. All data store-level locks are exclusive. A transaction that requires a data store-level lock cannot start until there are no active transactions on the data store. Once a transaction has obtained a data store-level lock, all other transactions are blocked until the transaction commits.

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

Different transactions can coexist with different levels of locking, but the presence of even one transaction using data store-level locking leads to reduced concurrency.

Table-level locking

Table-level locking locks a table when it is accessed by a transaction and is useful when a statement accesses most of the rows in a table. Such statements can be queries, updates, deletes or multiple inserts done in a single transaction. For any SQL statement, the optimizer determines when a table lock should be used. To optimize transactions that contain multiple statements that, when combined, access most of the rows in the table, applications can call the **ttOptSetFlag**() procedure to request that the optimizer use table locks.

As with all higher-granularity locks, table locks may reduce throughput, so they should only be used where a substantial portion of the table needs to be locked, or where high concurrency is not needed. In read-committed isolation, TimesTen does not use table-level locking for read operations unless it was explicitly requested by the application by calling **ttOptSetFlag**().

Row-level locking

Row-level locking only locks the rows that are accessed by a transaction. It provides the best concurrency by allowing concurrent transactions to access rows in the same table. Row-level locking is generally preferable when there are many concurrent transactions, each operating on different rows of the same tables.

Latches

Unlike locks, which are designed to be held across application interactions, *latches* are low-level concurrency tools that are designed to operate fast and are not held across application interactions. Latches are used internally by TimesTen to protect internal data structures from concurrent access and are transparent to applications.

Transaction Isolation

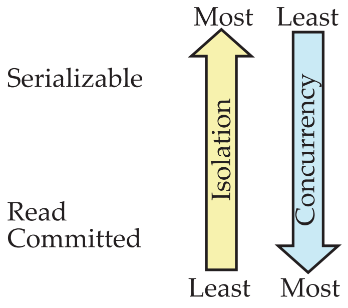
Transaction isolation provides an application with the appearance that the system performs one transaction at a time, even though there are concurrent connections to the data store. Applications can use the **Isolation** connection attribute to set the isolation level for a connection. Concurrent connections can use different isolation levels.

The isolation levels are:

- Read committed isolation
- Serializable isolation

As shown in Figure 5.2, isolation level and concurrency are inversely related. A lower isolation level enables greater concurrency, but with greater risk of data inconsistencies. A higher isolation level provides a higher degree of data consistency, but at the expense of concurrency.

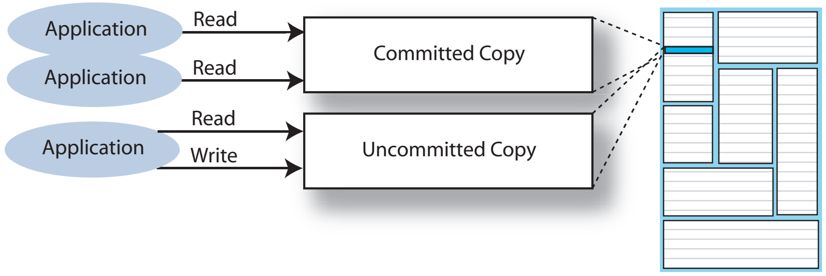
Figure 5.2 Isolation levels and degree of concurrency



Read committed isolation

When an application uses read committed isolation, readers have a separate copy of the data from writers, so locks are not needed. Read committed isolation is non-blocking for queries and can work together with Read committed isolation isolation. Writers block only other writers and Read committed isolation readers. Read committed isolation is the default isolation level.

Figure 5.3 Read Committed Isolation



Read committed isolation provides increased concurrency because readers do not block writers and writers do not block readers. This isolation level is useful for applications that have long-running scans that may conflict with other operations needing access to a scanned row. However, the disadvantage when using this isolation level is that non-repeatable reads are possible within a transaction or even a single statement (for example, the inner loop of a nested join).

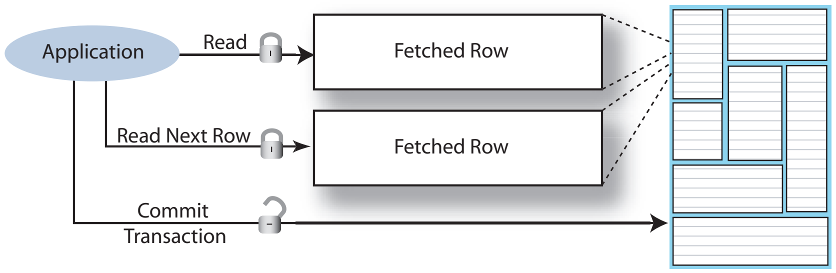
When using this isolation level, DDL statements that operate on a table can block readers and writers of that table. For example, an application cannot read a row from a table if another application has an uncommitted DROP TABLE, DROP INDEX, or ALTER TABLE operation on that table. In addition, blocking checkpoints will block readers and writers.

Read committed isolation does acquire read locks as needed during materialized view maintenance to ensure that views are consistent with their detail tables. These locks are not held until the end of the transaction but are instead released when maintenance has been completed (for example, when an update to the detail tables returns success or failure).

Serializable isolation

When an application uses serializable isolation, locks are acquired within a transaction and are held until the transaction commits or rolls back. This level of isolation provides for repeatable reads and increased isolation within a transaction at the expense of decreased concurrency. Transactions use serializable isolation when data store-level locking is chosen.

Figure 5.4 Serializable Isolation



Serializable isolation level is useful for transactions that require the strongest level of isolation. Serializable isolation also reduces deadlocks. However, concurrent applications that need to modify the data read by the transaction may encounter lock timeouts because read locks are held until the transaction commits.

For More Information

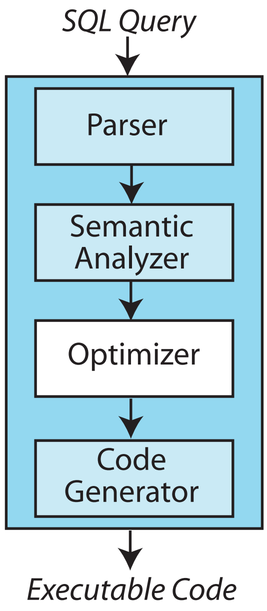
For more information on locks and transaction isolation, see Chapter 7, "Transaction Management and Recovery" in the *Oracle TimesTen In-Memory Database Operations Guide*.

6

Query Optimization

TimesTen has a cost-based query optimizer that speeds up data access by automatically choosing the optimal way to answer queries. Optimization is performed in the third stage of the compilation process. The four stages of compilation are shown in Figure 6.1.

Figure 6.1 Compilation stages



The role of the optimizer is to determine the lowest cost plan for executing queries. By “lowest cost plan” we mean an access path to the data that will take the least amount of time. The optimizer determines the cost of a plan based on:

- Table and column statistics
- Metadata information (referential integrity, primary key, etc.)
- Index choices (including automatic creation of temporary indexes)
- Scan methods (full table scan, Rowid lookup, hash or T-tree index scan)
- Join algorithm choice (nested loop joins, nested loop joins with indexes, or merge join)

The main topics in this chapter are:

- Optimization Time and Memory Usage
- Statistics
- Optimizer Hints
- Indexing Techniques
- Scan Methods
- Join Methods
- Optimizer Plan

Optimization Time and Memory Usage

The TimesTen optimizer is designed to generate the best possible plan within reasonable time and memory constraints. However, unlike many other components of TimesTen, the speed of optimization is not an overriding factor in the optimizer's design. We assume that most commands will be pre-compiled and that compilation will not be done during time-critical periods. On the other hand, in order to provide TimesTen applications with optimum throughput, the quality of the optimized plans is critical. For these reasons, the optimizer is designed to give precedence to execution time over optimization time.

Furthermore, the plans generated by the optimizer emphasize performance over memory usage. The optimizer may designate the use of significant amounts of temporary memory space in order to speed up execution time. In memory constrained environments, applications can use the optimizer hints described in "Optimizer Hints" on page 59 to disable the use of temporary indexes and tables in order to create plans that trade maximum performance for less memory usage.

Statistics

When determining the optimum execution path for a query, the optimizer examines statistics about the data referenced by the query, such as the number of rows in the tables, the minimum and maximum values and the number of unique values in interval statistics of columns used in predicates, the existence of primary keys within a table, the size and configuration of any existing indexes. These statistics are stored in the "SYS.TBL_STATS" and "SYS.COL_STATS" tables, which are populated by TimesTen when an applications calls the **ttOptUpdateStats**() procedure.

The optimizer uses the statistics for each table to calculate the *selectivity* of predicates, such as T1.A = 4, or a combination of predicates, such as T1.A = 4 AND T1.B < 10. *Selectivity* is a measurement of the number of rows in a table, index, view, or the result from a join or GROUP BY operation that are expected to pass the predicate test. If a predicate selects a small percentage of rows, it is said to have *high* selectivity, while a predicate that selects a large percentage of rows has *low* selectivity.

Optimizer Hints

The optimizer allows applications to provide *hints* to adjust the way that plans are generated. For example, applications can use the **ttOptSetFlag**() procedure to provide the TimesTen optimizer with hints about how to best optimize any particular query. This takes the form of directives that restrict the use of particular join algorithms, use of temporary indexes and types of index (T-tree or hash), use of locks, whether to optimize for all the rows or only the first '*n*' number of rows in a table, and whether to materialize intermediate results. The existing hints set for a data store can be viewed using the **ttOptGetFlag**() procedure.

Applications can also use the **ttOptSetOrder**() procedure to specify the order in which tables are to be joined by the optimizer, as well as the **ttOptUseIndex**() procedure to specify which indexes should be considered for each correlation in a query.

Indexing Techniques

The TimesTen query optimizer uses indexes to speed up the execution of a query. The optimizer uses existing indexes or, if necessary, creates temporary indexes to generate an optimal execution plan when preparing a SELECT, INSERT SELECT, UPDATE, or DELETE statement.

An index is a map of keys to row locations in a table. Strategic use of indexes is essential in order to obtain maximum performance from a TimesTen system.

TimesTen uses two types of indexing technologies:

- Hash indexes
- T-tree indexes

Hash indexes are created automatically when a primary key is defined on a table to provide fast lookup for equality searches. T-tree indexes are created by the CREATE INDEX statement and are used for lookups involving equality and ranges (greater than/equal to, less than/equal to, and so on). Indexes are automatically created to enforce unique column constraints and foreign key constraints. In some situations, the optimizer may create temporary indexes for tables to enable them to be joined more efficiently.

In general, hash indexes are faster than T-tree indexes for exact match lookups and equijoins. However, hash indexes cannot be used for lookups involving ranges or the prefix of a key and require more space than T-tree indexes.

All hash indexes are *unique*, which means that each row in the table has a unique value for the indexed column or columns. T-tree indexes can be designated as unique. Unlike hash indexes, where all component columns cannot be nullable, unique T-tree indexes can be created over nullable columns. In conformance with the SQL standard, multiple NULL values are permitted in a unique T-tree index.

Hash indexes

A hash index is created automatically by TimesTen when a table is created by a CREATE TABLE statement with a PRIMARY KEY or UNIQUE HASH option specified on one or more columns. Primary key columns cannot be NULL and there can be only one hash index for each table.

The PAGES parameter in the CREATE TABLE statement specifies how many buckets are to be allocated for the table's hash index. Fewer buckets in the hash index result in more hash collisions. More buckets reduce collisions but can waste memory. Hash key comparison is a fast operation, so a small number of hash collisions does not significantly impact performance.

T-tree indexes

Main memory data management systems are designed to reduce memory requirements, as well as to shrink code path and footprint. A typical disk-based RDBMS uses B+-tree indexes to reduce the amount of disk I/O needed to accomplish indexed look-up of data files. TimesTen uses *T-tree* indexes, which are optimized for main memory access. T-tree indexes are more economical than B-trees in that they require less memory space and fewer CPU cycles.

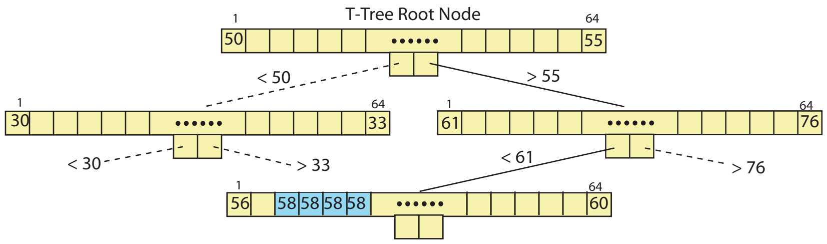
T-trees reduce space requirements by holding only the value of the left-most key in each index node. Other values can be accessed by accessing the data rows themselves. T-trees have no disk I/O because the index tree structures are all memory-resident. They shrink the complexity of code and reduce code path length by removing unnecessary computations and taking advantage of pointer traversal where possible.

Use the CREATE INDEX statement to create a T-tree index for a table. A T-tree index can be created either before or after the rows are inserted in the table. Each CREATE INDEX statement describes at least one column to be used as the index key. If multiple columns are described, the first column in the list has the highest precedence in the sort order of index keys.

TimesTen creates T-tree indexes to enforce unique column constraints for foreign key constraints.

Figure 6.2 shows the search path used to locate a range of rows that contain the value '58' in the column designated as the index key. (This column could be either the only index key or the first in a sort order.)

Figure 6.2 T-Tree Nodes



As shown in Figure 6.2, a T-tree consists of a group of *T-tree nodes*. Each node consists of a group of 64 index keys, each of which points to a separate table row. T-tree nodes start with a *root node*, which is the first node in the search path. The other nodes are distributed in a "balanced" manner from the root node, so that the higher half of the values are in nodes off the right branch and the lower half of the values in nodes off the left branch. In this way, the T-tree search algorithm knows with just two comparisons whether the value it is searching for is on the current node or elsewhere in memory. With every new index node pointer indirection, its search area is cut in half.

Scan Methods

The optimizer can select from multiple types of scan methods, the most common of which are:

- Full Table Scan
- Rowid Lookup
- Hash Index Scan (on either a permanent or temporary index)
- T-tree Index Scan (on either a permanent or temporary index)

TimesTen performs high-performance exact matches through hash indexes and rowid lookups, and performs range matches through in-memory optimized T-tree indexes.

A *full table scan* involves a complete scan of a table. When a full scan is executed, a cursor is opened on a table, each row is examined as the cursor is advanced, and at the end the cursor is closed. The full scan is considered only if no hash scan or range scan is applicable. If the **ttOptSetFlag** flag *scan* is false, other types of scans take precedence over a full scan.

TimesTen assigns a unique id, called ROWID, to each row stored in a table. A *Rowid lookup* is applicable if, for example, an application has previously selected a ROWID and then uses a 'WHERE ROWID =' clause to fetch that same row. RowID lookups are faster than either type of index scan. If the **ttOptSetFlag** flag *Rowid* is false, then Rowid lookups are not considered applicable.

A *hash index scan* involves using a hash index to access a table. Such a scan is applicable if the table has a hash index containing a key that can be completely matched by the predicate. The term "scan" implies that the optimizer looks at more than one row, so a hash index scan is really more of a "lookup," like Rowid lookup, because it returns at most one row. If the **ttOptSetFlag** flag *hash* is false, then hash scans are not considered applicable.

A *T-tree index scan* involves using a T-tree index to access a table. Such a scan is applicable as long as the table has one or more T-tree indexes. The optimizer attempts to match as long a prefix of the key columns as possible. If no match is found for a particular key column, then the index scan returns all of the values in that column. If the predicate specifies a single value (such as T1.A = 2) for each key column and if the T-tree index is unique, the optimizer locates the row by means of a lookup, rather than a scan. If the **ttOptSetFlag** flag *Ttree* is false, then T-tree scans are not considered applicable.

Join Methods

The optimizer can select from multiple types of join methods. When the rows from two tables are joined, one table is designated the *outer table* and the other the *inner table*. During a join, the optimizer scans the rows in the outer and inner tables to locate the rows that match the join condition.

The optimizer analyzes the statistics for each table and, for example, might identify the smallest table or the table with the best selectivity for the query as outer table. If indexes exist for one or more of the tables to be joined, the optimizer will take them into account when selecting the outer and inner tables.

If more than two tables are to be joined, the optimizer analyzes the various combinations of joins on table pairs to determine which pair to join first, which table to join with the result of the join, and so on for the optimum sequence of joins.

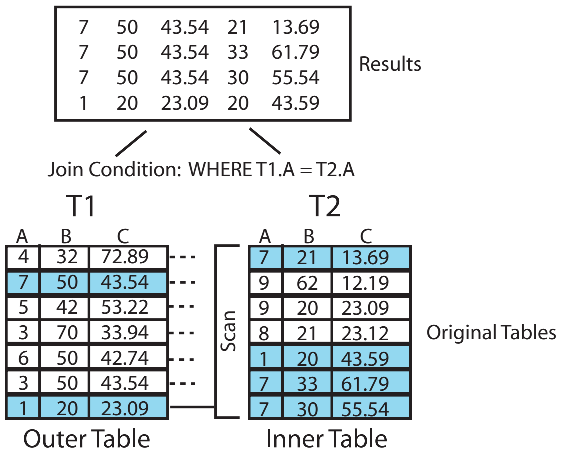
The cost of a join is largely influenced by the method in which the inner and outer tables are accessed to locate the rows that match the join condition. The optimizer can select from two join methods:

- Nested Loop Join
- Merge Join

Nested Loop Join

In a nested loop join with no indexes, a row in the outer table is selected one at a time and matched against every row in the inner table. All of the rows in the inner table are scanned as many times as the number of rows in the outer table. If the inner table has an index on the join column, that index is used to select the rows that meet the join condition. The rows from each table that satisfy the join condition are returned. Indexes may be created on the fly for inner tables in nested loops, and the results from inner scans may be materialized before the join.

Figure 6.3 Nested Loop Join



7	50	43.54	21	13.69
7	50	43.54	33	61.79
7	50	43.54	30	55.54
1	20	23.09	20	43.59

A	B	C
4	32	72.89
7	50	43.54
5	42	53.22
3	70	33.94
6	50	42.74
3	50	43.54
1	20	23.09

A	B	C
7	21	13.69
9	62	12.19
9	20	23.09
8	21	23.12
1	20	43.59
7	33	61.79
7	30	55.54

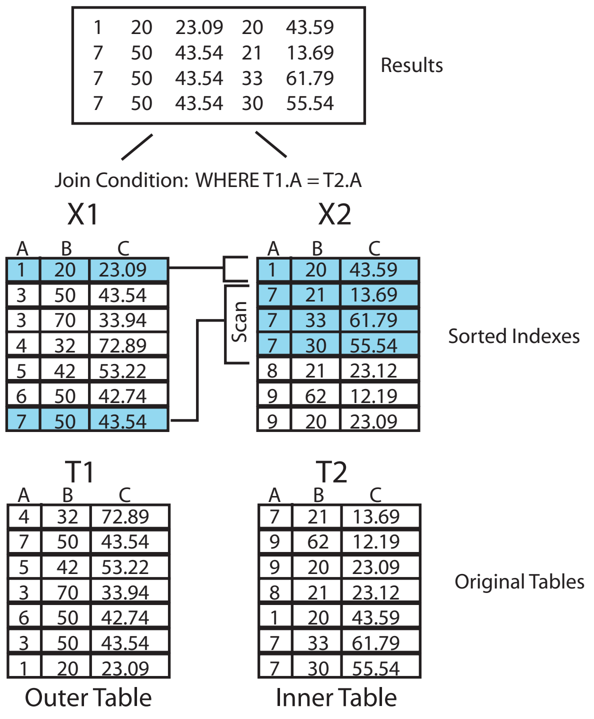
Merge Join

Merge-join is used only when the join columns are sorted by T-tree indexes. In a merge join, a cursor advances through each index one row at a time. Because the rows are already sorted on the join columns in each index, a simple formula is applied to efficiently advance the cursors through each row in a single scan. The formula looks something like:

- If Inner.JoinColumn < Outer.JoinColumn -- advance inner cursor
- If Inner.JoinColumn = Outer.JoinColumn -- read match
- If Inner.JoinColumn > Outer.JoinColumn -- advance outer cursor

Unlike a nested loop join, there is no need to scan the entire inner table for each row in the outer table. A merge join can be used when T-tree indexes have been created for the tables by means of CREATE INDEX statements prior to preparing the query. If no T-tree indexes exist for the tables being joined prior to preparing the query, the optimizer may in some situations create temporary T-tree indexes in order to use merge join.

Figure 6.4 Merge Join



Results

1	20	23.09	20	43.59
7	50	43.54	21	13.69
7	50	43.54	33	61.79
7	50	43.54	30	55.54

X1

A	B	C
1	20	23.09
3	50	43.54
3	70	33.94
4	32	72.89
5	42	53.22
6	50	42.74
7	50	43.54

X2

A	B	C
1	20	43.59
7	21	13.69
7	33	61.79
7	30	55.54
8	21	23.12
9	62	12.19
9	20	23.09

T1

A	B	C
4	32	72.89
7	50	43.54
5	42	53.22
3	70	33.94
6	50	42.74
3	50	43.54
1	20	23.09

T2

A	B	C
7	21	13.69
9	62	12.19
9	20	23.09
8	21	23.12
1	20	43.59
7	33	61.79
7	30	55.54

Optimizer Plan

Like most database optimizers, the TimesTen query optimizer stores the details on how to most efficiently perform SQL operations in an *execution plan*, which can be examined and customized by application developers and administrators.

The execution plan data is stored in the TimesTen "SYS.PLAN" table and includes information on which tables are to be accessed and in what order, which tables are to be joined, and which indexes are to be used. Users can direct the TimesTen query optimizer to enable or disable the creation of an execution plan in the "SYS.PLAN" table by setting the *GenPlan* optimizer flag in the **ttOptSetFlag**() procedure.

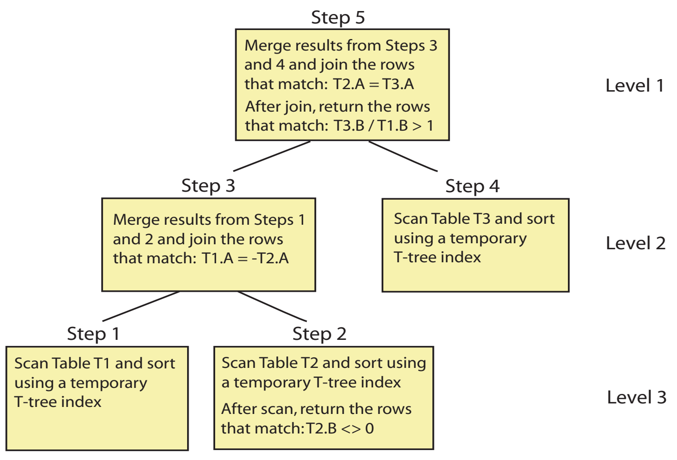
The execution plan designates a separate *step* for each database operation to be performed to execute the query. The steps in the plan are organized into *levels* that designate which steps must be completed to generate the results required by the step or steps at the next level.

For example, the optimizer prepares an execution plan for the following query:

```
SELECT COUNT(*)
FROM T1, T2, T3
WHERE T3.B/T1.B > 1
AND T2.B <> 0
AND T1.A = -T2.A
AND T2.A = T3.A
```

In this example, the optimizer dissects the query into its individual operations and generates a 5-step execution plan to be performed at three levels, as shown in Figure 6.5.

Figure 6.5 Example Execution Plan



Step 5
Merge results from Steps 3 and 4 and join the rows that match: T2.A = T3.A
After join, return the rows that match: T3.B / T1.B > 1
Level 1
Step 3
Merge results from Steps 1 and 2 and join the rows that match: T1.A = -T2.A
Step 4
Scan Table T3 and sort using a temporary T-tree index
Level 2
Step 1
Scan Table T1 and sort using a temporary T-tree index
Step 2
Scan Table T2 and sort using a temporary T-tree index
After scan, return the rows that match:T2.B <> 0
Level 3

For More Information

For more information on the query optimizer, see Chapter 9, "The TimesTen Query Optimizer" in the *Oracle TimesTen In-Memory Database Operations Guide*.

For more information on indexing, see the "Understanding indexes" and "Working with indexes" in Chapter 6, "Working with Data in a TimesTen Data Store" in the *Oracle TimesTen In-Memory Database Operations Guide*. Also see descriptions for the CREATE TABLE and CREATE INDEX statements in the *Oracle TimesTen In-Memory Database API and SQL Reference Guide*.

7

Data Availability and Integrity

Although TimesTen data stores are maintained in memory, they are not transient, temporary objects. TimesTen ensures the availability, durability, and integrity of data through the use of three mechanisms:

- Logging
- Checkpointing
- Replication

Consistency and concurrency are guaranteed through locking, which is described in "Locks" on page 50.

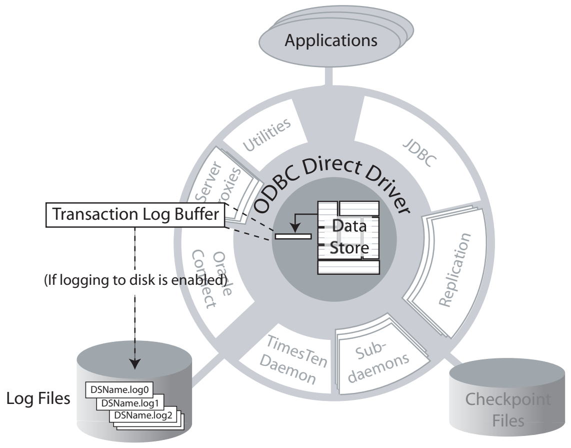
Logging

As described in “Logging” on page 13, the TimesTen log is used to redo transactions should a system failure occur, undo transactions that are rolled back, replicate changes to other TimesTen data stores, replicate changes to an Oracle database, and allow applications to monitor changes to tables (through the XLA interface described in “Transaction Log API” on page 85).

As shown in Figure 7.1, TimesTen enables logging at two levels:

- An in-memory *transaction log buffer*
- On-disk *log files*

Figure 7.1 Two-tiered logging



Logging options

TimesTen provides a **Logging** configuration attribute that allows users to fine-tune the logging behavior of a data store to obtain the best compromise between performance and durability.

The logging options are:

- Disk-based logging
- Logging disabled

Note: The logging option selected for a data store becomes a fixed property of that data store when it is loaded into memory. You cannot have different connections concurrently use different logging options.

Logging disabled

Disabling logging provides the best performance. However, with logging disabled, transactions cannot be rolled back or recovered, and only data store-level locking is allowed. Logging should be disabled only during operations that can be restarted from the beginning if a failure occurs, such as bulk-loading a data store.

Disk-based logging

Applications can obtain data durability by periodically writing the contents of the log buffer to disk-based log files. Users can configure TimesTen to do this write-to-disk operation automatically at commit time or the application can do it manually using the **ttDurableCommit**() built-in procedure call, as described in "Durable and non-durable commits" on page 72. When the log buffer becomes full, TimesTen automatically writes its contents to a log file on disk.

With disk-based logging enabled, TimesTen records a copy of the data store changes to disk as part of completing (committing) the transaction. Transactions saved in log files can be rolled back. In addition, you can use the contents of the log files to recover data stores to the most recently committed state. However, due to the relative slowness of disk access, disk-based logging can impact the response time of the transaction and the throughput of the total system.

Log files are either generated in the same directory as the data store files or in the directory specified by the **LogDir** attribute in the DSN. Users can use the **LogBuffSize** attribute to configure the size of the in-memory log buffer and the **LogFileSize** attribute to configure the size of log files. When a log file becomes full, TimesTen creates a new log file. Log files are named *DataStoreNam*e.log*n* where *n* is an integer starting at 0 and incremented each time a new log file is created.

Logging disabled

Disabling logging provides better performance. However, when logging is disabled, transactions cannot be rolled back or recovered, and only data store-level locking is allowed. Logging should be disabled only during operations that can be restarted from the beginning if a failure occurs, such as bulk-loading a data store.

Autocommits

ODBC provides an autocommit mode that forces a commit after each statement. By default, autocommit is enabled so that an implicit commit is issued immediately after a statement executes successfully. Also, an implicit rollback is issued immediately after a statement fails to execute. Regardless of whether a statement has been executed successfully or unsuccessfully, if the connection has open cursors, the automatic commit or rollback is issued only after all its cursors have been closed. If applications require transactions that contain more than one statement, autocommit must be disabled and commits must be explicitly issued.

Different commit frequencies need to be tested to determine optimal performance. For example, if **DurableCommits** is enabled, the cost of performing frequent commits is decreased performance due to increased disk writes. On the other hand, the cost of performing infrequent commits is decreased concurrency because locks are held for a longer period of time and it takes longer to rollback transactions.

Applications can call the ODBC **SQLSetConnectOption** function or the JDBC **Connection.setAutoCommit** method to enable or disable autocommits.

Durable and non-durable commits

As described in "Disk-based logging" on page 71, when disk-based logging is enabled, applications can control when the data in the transaction log buffer is written to disk. TimesTen provides a **DurableCommits** configuration attribute that specifies whether the log buffer is automatically written to disk at transaction commit or is deferred until the application calls the **ttDurableCommit**() procedure.

Setting **DurableCommits** so that the log is not automatically posted to disk at transaction commit reduces the transaction execution time at the risk of losing the results of some committed transactions in the event of a failure. However, regardless of the setting of **DurableCommits**, the log is saved to disk when the log buffer in memory fills up.

If logging to disk is disabled, durable commits are not possible. In this situation, durability of data can be achieved through checkpointing, which the application can explicitly initiate. TimesTen automatically initiates checkpointing when the last connection to the data store exits successfully. See "Checkpointing" on page 73 for more information.

When are log files deleted?

Log files are deleted from disk when a checkpoint operation is done on the data store (see "Checkpointing"). Log files are kept until TimesTen declares them to be *purgeable*. A log file cannot be purged until:

- All transactions writing log records to the log file (or a previous log file) have committed or rolled back
- All changes recorded in the log file have been written to the checkpoint files on disk
- All changes recorded in the log file have been replicated (if replication is used)
- All changes recorded in the log file have been propagated to Oracle (if Cache Connect is used)
- All changes recorded in log files have been reported to the XLA applications (if XLA is used)

Checkpointing

Checkpoints are used to post a snapshot of the data store to files on disk. In the event of a system failure, TimesTen can use a checkpoint file together with log files to restore a data store to its last transactionally consistent state prior to a crash. An application can initiate a checkpoint operation by calling the **ttCkpt**() or **ttCkptBlocking**() procedure.

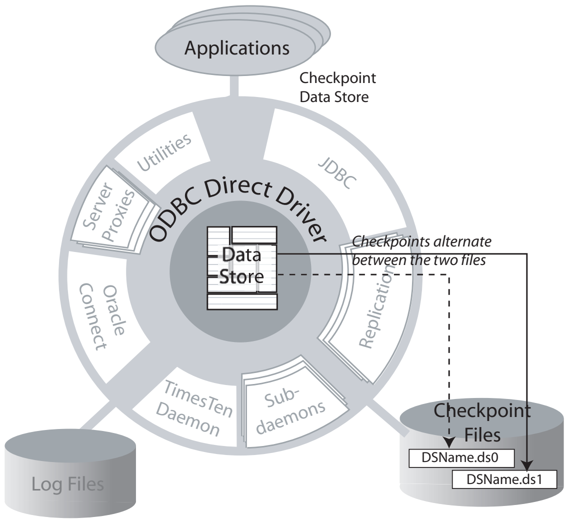
TimesTen maintains two checkpoint files for each data store, named *DataStoreName*.ds0 and *DataStoreName*.ds1. When a data store is checkpointed to disk, TimesTen overwrites the oldest of the two checkpoint files.

Only the data that has changed since the last checkpoint operation is rewritten to the checkpoint file. The checkpoint operation scans the data store for blocks that have changed since the last checkpoint. It then updates the checkpoint file with the changes and removes any log files that are no longer needed.

If checkpoints are not performed frequently enough, log files build up on disk. An accumulation of log files may cause the disk to run out of space, resulting in a lengthy recovery operation should the system crash. On the other hand, checkpointing too frequently may incur extra overhead that may impact the performance of other applications using the data store.

In the event of a system failure, TimesTen can recover a data store by using the latest consistent checkpoint file and applying the log files to create a new checkpoint file. If disk-based logging is enabled, this new checkpoint file will include transactions that were written to disk prior to the failure. See "Recovery from log and checkpoint files" on page 75 for more information.

Figure 7.2   Checkpoint operation



Blocking and non-blocking (“fuzzy”) checkpoints

Checkpoints can be blocking or non-blocking.

An application can call the **ttCkptBlocking**() procedure to initiate a blocking checkpoint, which briefly requires exclusive access to the data store in order to construct a transaction-consistent checkpoint. While a blocking checkpoint operation is in progress, any other new transactions are put in a queue behind the checkpointing transaction. As a result, if any transaction is long-running, it may cause many other transactions to be held up. No log is needed to recover from a blocking checkpoint, so this type of checkpoint does not require disk-based logging.

When disk-based logging is enabled, an application can call the **ttCkpt**() procedure to initiate a non-blocking checkpoint, which is also known as a *fuzzy* checkpoint. Non-blocking checkpoints do not require any locks on the data store, so multiple applications can asynchronously commit or roll back transactions on the same data store while the checkpoint operation is in progress. When using non-blocking checkpoints, should the system crash, TimesTen will use the log to recover from the checkpoints. For this reason, non-blocking checkpoints requires disk-based logging.

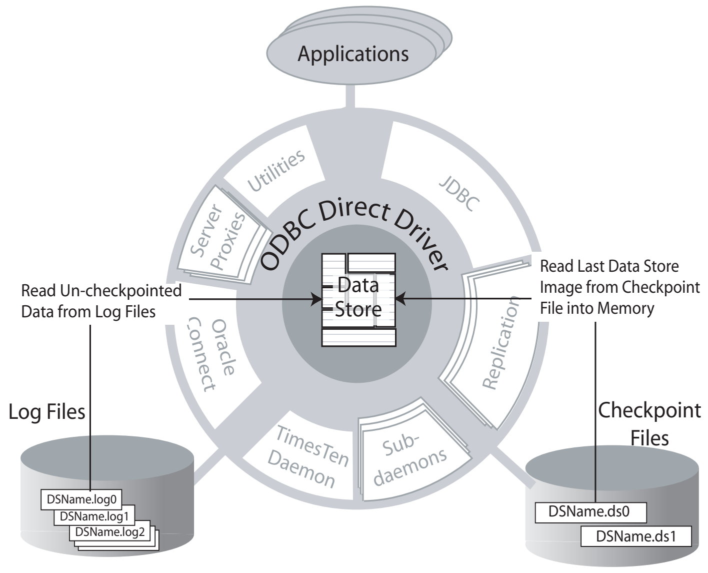
Recovery from log and checkpoint files

Should a data store become invalid or corrupted by a system crash or process failure, every connection to the data store is invalidated. When an application reconnects to a failed data store, the subdaemon allocates a new memory segment for the data store and recovers its data from the checkpoint and log files.

During recovery, the latest checkpoint file is read into memory and all durably committed transactions are rolled forward from the appropriate log file(s). Uncommitted or rolled back transactions are not recovered.

For mission-critical applications that require uninterrupted access to TimesTen data in the event of failures, see "Replication" on page 75.

Figure 7.3 Recovering from log and checkpoint files



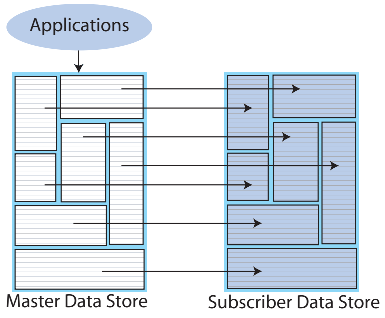
Replication

Replication is the process of copying data between data stores. The fundamental motivation behind TimesTen replication is to make data highly available to mission critical applications with minimal performance impact.

In addition to its role in failure recovery, replication is also useful for distributing user loads across multiple data stores for maximum performance and for facilitating online upgrades and maintenance, as described in "Upgrading TimesTen" on page 110.

As shown in Figure 7.4, TimesTen replication copies updates made in a *master* data store to a *subscriber* data store.

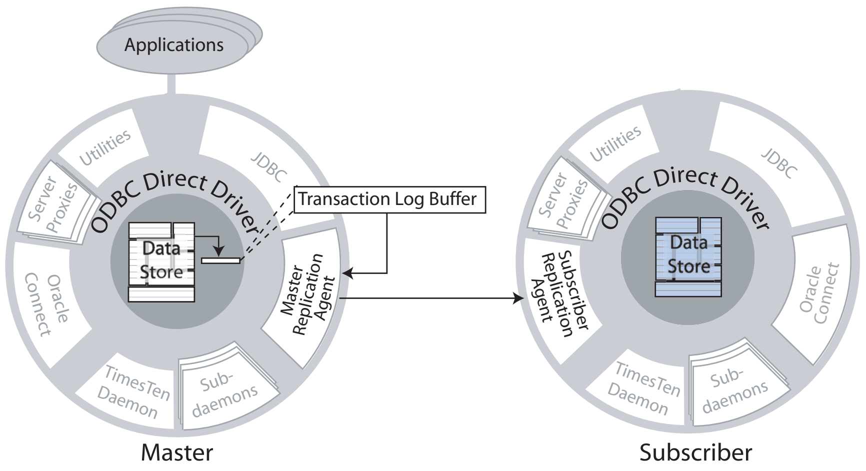
Figure 7.4 Master and subscriber data stores



To enable high efficiency and low overhead, TimesTen uses a transaction-log based replication scheme.

As shown in Figure 7.5, replication at each master and subscriber data store is controlled by *replication agents* that communicate through TCP/IP stream sockets. The replication agent on the master data store reads the records from its transaction log and forwards any detected changes to replicated elements to the replication agent on the subscriber data store. The replication agent on the subscriber then applies the updates to its data store. If the subscriber agent is not running when the updates are forwarded by the master, the master retains the updates in its log until they can be applied at the subscriber.

Figure 7.5 Master and Subscriber Replication Agents



Replication configurations

The focus for TimesTen's data replication is high availability. To achieve this goal, TimesTen replication can be configured in a variety of ways for the best balance between performance and availability.

Replication is configured through SQL statements. In general, replication can be configured to be unidirectional ("one way") from a master to one or more subscribers, or bidirectional ("two way") between two or more data stores that serve as both master and subscriber. More than two data stores configured for bidirectional replication is often referred to as "N-way" or "update anywhere" replication.

Figure 7.6 Example replication configuration schemes

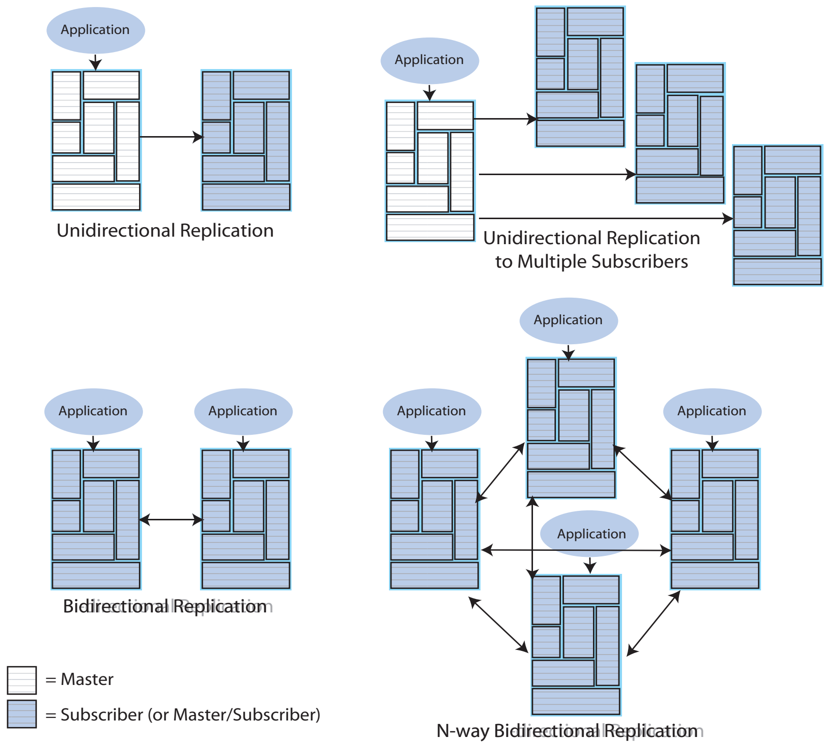

Application
Unidirectional Replication
Application
Unidirectional Replication
to Multiple Subscribers
Application
Application
Bidirectional Replication
Application
Application
Application
Application
N-way Bidirectional Replication
= Master
= Subscriber (or Master/Subscriber)

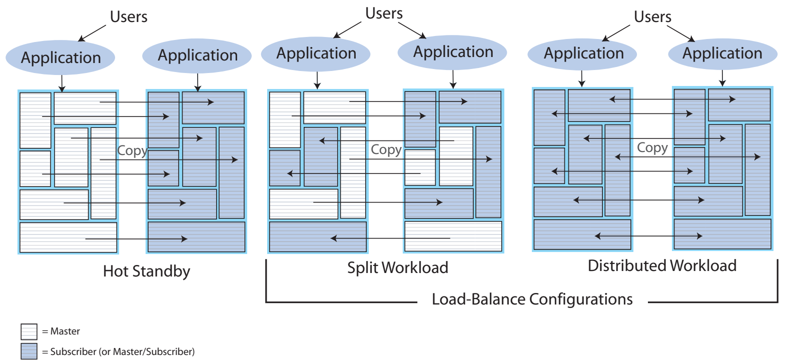
Figure 7.7 shows how replication can be configured to provide a hot standby backup data store or to more evenly balance user loads.

In the hot standby configuration, a single master data store is replicated to a subscriber data store with appropriate fail-over mechanisms built into the application itself. See "Scenario 1: Caller usage metering application" on page 19 and "Scenario 3: Online travel agent application" on page 25 for example applications that use hot standby data stores.

In the load-balanced pair configuration, the workload can be split between two bidirectionally replicated data stores. As shown in Figure 7.7, there are two basic types of load-balance configurations:

- *Split workload* where each data store bidirectionally replicates a portion of its data to the other data store.
- *Distributed workload* where user access is distributed across duplicate application/data store combinations that bidirectionally replicate updates.

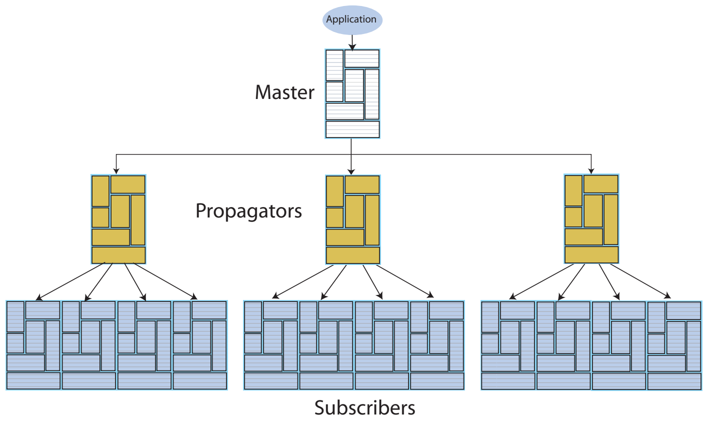
Figure 7.7 Hot standby and load balancing configurations



In a distributed workload configuration, the application has the responsibility to divide the work between the two systems so that replication collisions do not occur. In the event that collisions do occur, TimesTen has a timestamp-based collision detection and resolution capability.

Replication can also be used to propagate updates from one or more data stores to many data stores. This can be useful when maintaining duplicate data stores over lower-bandwidth connections and for distributing (or *fanning out*) replication loads in configurations in which a master data store must replicate to a large number of subscribers.

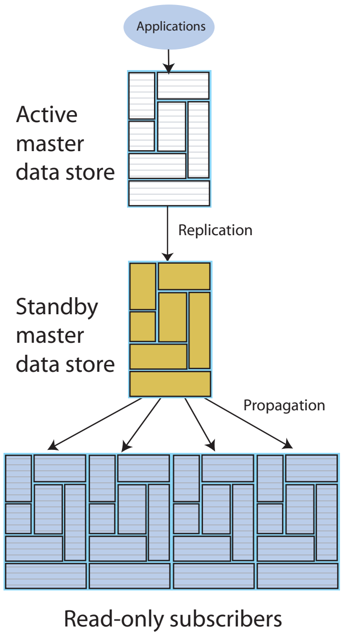
Figure 7.8 Propagation configuration



Active standby pair

Figure 7.9 shows an active standby pair.

Figure 7.9 Active standby pair



An active standby pair is created by the CREATE ACTIVE STANDBY PAIR SQL statement. The CREATE ACTIVE STANDBY PAIR specifies a active master data store, a standby master data store, subscriber data stores, and the tables and cache groups that comprise the data stores.

Note: The other replication schemes in this chapter are created by using the CREATE REPLICATION statement.

In an active standby pair, two data stores are defined as masters. One is an active master data store, and the other is a standby master data store. The active master data store is updated directly. The standby master data store cannot be updated directly. It receives the updates from the active master data store and propagates the changes to as many as 62 read-only subscriber data stores. This arrangement ensures that the standby master data store is always ahead of the subscriber data stores and enables rapid failover to the standby data store if the active master data store fails.

Only one of the master data stores can function as an active master data store at a specific time. The active role is specified through the **ttRepStateSet** procedure. If the active master data store fails, then the user can use the **ttRepStateSet** procedure to change the role of the standby master data store to active before recovering the failed data store as a standby data store. The replication agent must be started on the new standby master data store.

If the standby master data store fails, the active master data store can replicate changes directly to the subscribers. After the standby master data store has recovered, it contacts the active standby data store to receive any updates that have been sent to the subscribers while the standby was down or was recovering. When the active and the standby master data stores have been synchronized, then the standby resumes propagating changes to the subscribers.

Asynchronous and return service replication

TimesTen replication is by default an asynchronous mechanism. When using asynchronous replication, an application updates a master data store and continues working without waiting for the updates to be received by the subscribers. The master and subscriber data stores have internal mechanisms to confirm that the updates have been successfully received and committed by the subscriber. These mechanisms ensure that updates are applied at a subscriber only once, but they are completely invisible to the application.

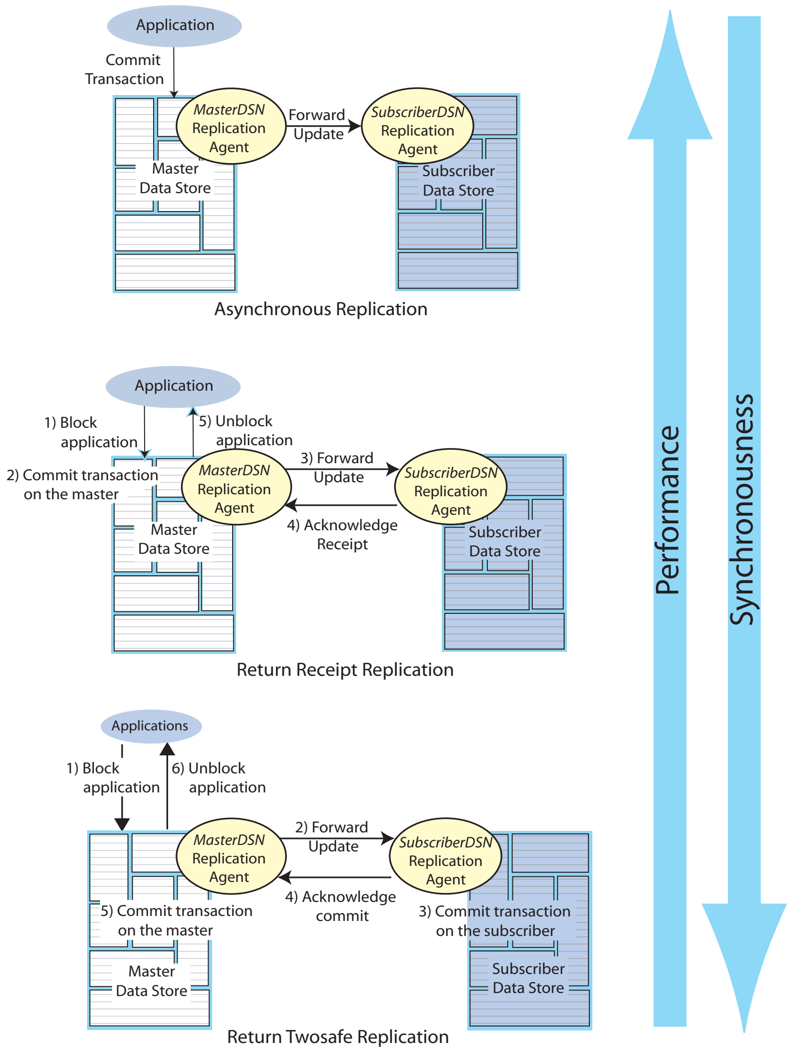
Asynchronous replication provides maximum performance, but the application is completely decoupled from the receipt process of the replicated elements on the subscriber. TimesTen also provides two *return service* options for more "pessimistic" applications that need higher levels of confidence that the replicated data is consistent between the master and subscriber data stores:

- The *return receipt service* loosely couples or "synchronizes" the application with the replication mechanism by blocking the application until replication confirms that the update has been received by the subscriber replication agent.

- The *return twosafe service* enables fully synchronous replication by blocking the application until replication confirms that the update has been both received *and committed* on the subscriber.

Applications that make use of the return services trade some performance to ensure higher levels of data integrity and consistency between the master and subscriber data stores. In the event of a master failure, the application has a higher degree of confidence that a transaction committed at the master persists in the subscribing data store. Return receipt replication has less performance impact than return twosafe at the expense of less synchronization.

Figure 7.10 Asynchronous and return service replication



Replication failover and recovery

In order for replication to make data highly available to mission critical applications with minimal performance impact, there must be a means to shift users from the failed data store to its surviving backup in as seamless a manner as possible.

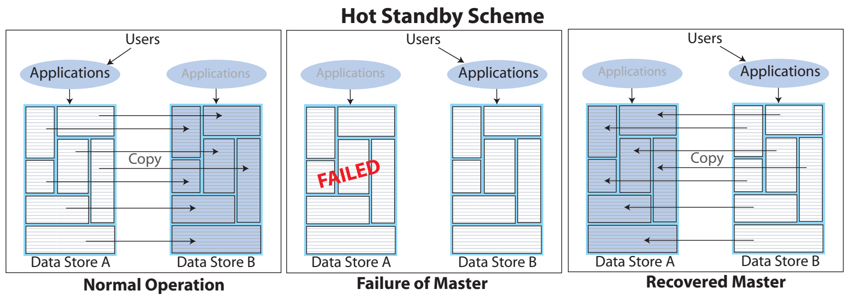
Management of failover and recovery operations are outside the scope of TimesTen. Instead, TimesTen replication has been designed and tested to operate in conjunction with both custom and third-party cluster managers that detect failures, redirect users or applications from the failed data store to one of its subscribers, and manage recovery of the failed data store. The cluster manager or administrator can use the TimesTen **ttRepAdmin** -duplicate utility or **ttRepDuplicateEx**() C function to duplicate the surviving data store and recover the failed data store.

Subscriber failures generally have no impact on the applications connected to the master data stores and can be recovered without disrupting user service. On the other hand, should a failure occur on a master data store, the cluster manager must redirect the application load to a subscriber in order to continue service with no or minimal interruption.

Failover and recovery are more efficient when the data stores are configured in a bi-directional general-workload scheme, such as the hot standby scheme shown in Figure 7.11 and the distributed workload scheme shown in Figure 7.12.

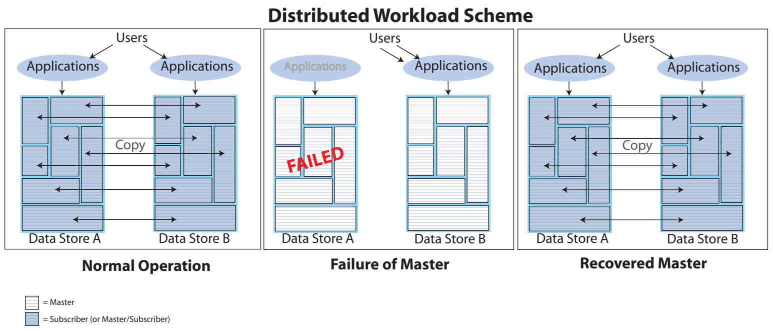
In the hot-standby scheme, should the master data store fail, the cluster manager need only shift the user load to the "hot standby" application on the subscriber data store. Upon recovering the failed data store, replication can be resumed with minimal interruption to service by reversing the roles of the master and subscriber data stores.

Figure 7.11 Failover scenario for hot standby scheme



The failover procedure for data stores configured using a distributed workload scheme is similar to that used for the hot standby, only failover involves shifting the users affected by the failed data store to join the other users of an application on a surviving data store. Upon recovery, the workload can be redistributed to the application on the recovered data store.

Figure 7.12 Failover scenario for distributed workload scheme



For More Information

For more information on logging and checkpointing, see Chapter 7, “Transaction Management and Recovery” in the *Oracle TimesTen In-Memory Database Operations Guide*.

For more information on replication, see the *TimesTen to TimesTen Replication Guide*.

8

Event Notification

TimesTen event notification is done through the “Transaction Log API” (*XLA*), which provides functions to detect updates to data stores. TimesTen provides XLA as an alternative to triggers.

The primary purpose of XLA is to monitor log records, each of which describes an update to a row in a table or a DDL operation. XLA can be used in conjunction with “Materialized Views” to better focus the scope of notification to detect updates made only to select rows across multiple tables. See “Materialized Views and XLA” on page 90 for more information.

TimesTen can also use “SNMP Traps” to send asynchronous alerts of critical events. See “SNMP Traps” on page 91 for more information.

Transaction Log API

TimesTen provides a Transaction Log API (XLA) that enables applications to monitor the transaction log of a local data store to detect updates made by other applications. XLA also provides functions that enable XLA applications to apply the detected changes to another data store. XLA is a C language API. However, TimesTen provides a C++ wrapper interface for XLA as part of the TTClasses product, as well as a separate Java wrapper interface.

Applications primarily use XLA to implement a change notification scheme. In this scheme, XLA applications can monitor a data store for changes and then possibly take actions based on those changes. For example, a TimesTen data store in a stock trading environment might be constantly updated from a data stream of stock quotes. Automated trading applications might use XLA to “watch” the data store for updates on certain stock prices and use that information to determine whether to execute orders. See “Scenario 2: Real-time quote service application” on page 23 for a complete example.

XLA might also be used to build a custom data replication solution in place of the TimesTen replication service described in the *TimesTen Replication Guide*. Such XLA-enabled replication solutions might include replication with a non-TimesTen database or *pushing* updates to another TimesTen data store.

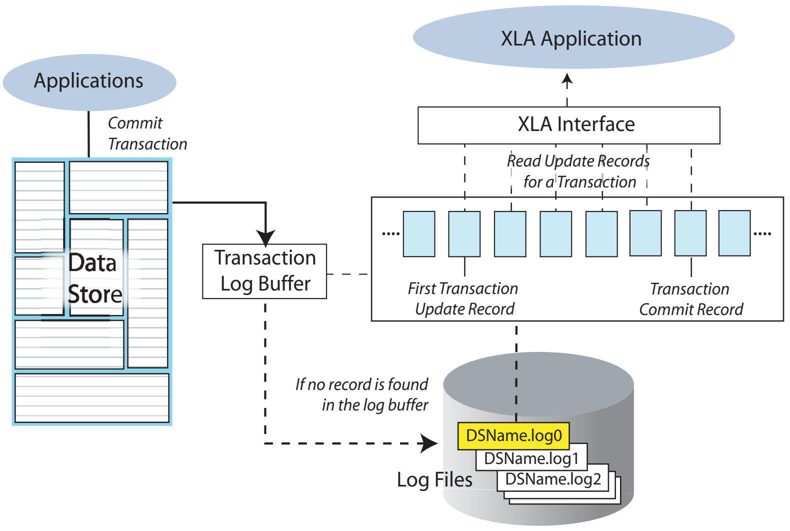
How XLA works

XLA operates in one of two modes:

- Persistent Mode
- Non-persistent Mode

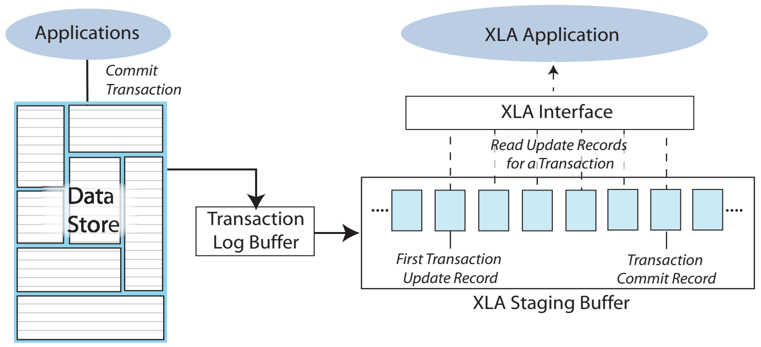
In persistent mode, XLA obtains update records for transactions directly from the transaction log buffer. If the records are not present in the buffer, XLA obtains the update records from the log files on disk, as shown in Figure 8.1.

Figure 8.1 XLA in Persistent Mode


XLA Application
Applications
XLA Interface
Commit Transaction
Read Update Records for a Transaction
Data Store
Transaction Log Buffer
First Transaction Update Record
Transaction Commit Record
If no record is found in the log buffer
DSName.log0
DSName.log1
DSName.log2
Log Files

In non-persistent mode, XLA obtains update records from the transaction log and stages them in an *XLA staging buffer*, as shown in Figure 8.2.

Figure 8.2 XLA in Non-persistent Mode



Each mode has its tradeoffs. For example, non-persistent mode is generally faster than persistent mode. However, the staging buffer can only be accessed by one reader at a time and, once records are read from the buffer, they are removed and no longer available. In addition, all of the update records in the buffer are lost when the data store is shut down. In contrast, when using persistent mode, multiple readers can simultaneously read transaction log updates and the log records are available for as long as the log files are available.

When working in persistent mode, readers make use of *bookmarks* to maintain their position in the log update stream. Bookmarks are stored in the data store, so they are persistent across data store connections, shutdowns, and crashes. An XLA application can use the **ttXlaGetLSN** function to obtain the location of its bookmark and the **ttXlaSetLSN** function to reset the bookmark location to reread a set of update records.

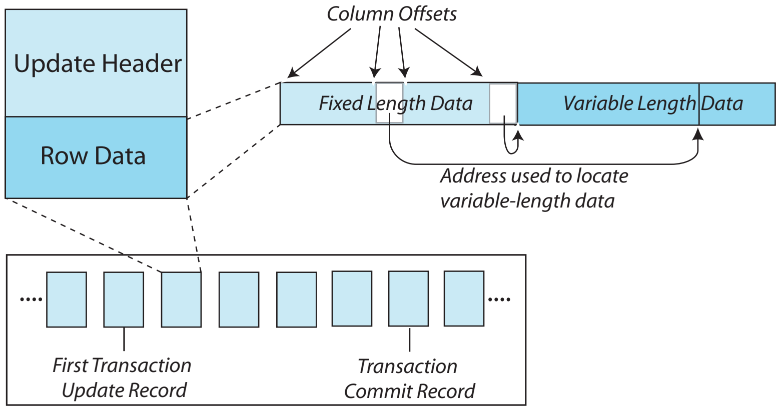
Log update records

Update records are available to be read from the log as soon as the transaction that created them commits. A "log sniffer" application can then use the XLA **ttXlaNextUpdate** function to obtain groups of update records written to the log.

Each returned record contains a fixed-length update header (**ttXlaUpdateDesc_t**) and one or two rows of data stored in an internal format. The update header describes the table to which the updated row applies; whether the record is the first or last (commit) record in the transaction; the type of transaction it represents; the length of the returned row data; and which columns in the row were updated, if any.

For records reflecting data updates, two rows are returned to report the row data before and after the update. For each row, the first portion of the data is the fixed-length data, which is followed by any variable-length data. Applications can call the **ttXlaGetColumnInfo** function to obtain the offset and size of the columns for a particular table. When XLA returns a record for that table, the columns in the fixed length portion of the returned row are located at those offsets returned by **ttXlaGetColumnInfo**. In the case of variable length data, the data located at the offsets in the fixed-length portion of the row is an address used to calculate the location of actual data in the variable-length portion of the row, as shown in Figure 8.3.

Figure 8.3 Contents of an update record



Column Offsets
Update Header
Row Data
Fixed Length Data
Variable Length Data
Address used to locate variable-length data
....
....
First Transaction Update Record
Transaction Commit Record

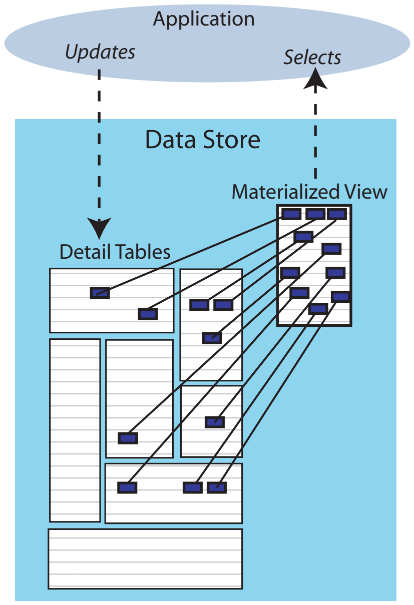
The row data in a record is in an internal TimesTen format. Once an application locates the address of a column, it can directly cast the value at that address to a corresponding ODBC C type. Complex data types, such as date, time, and decimal values, can be converted by applications to their corresponding ODBC C value using the XLA conversion functions described in """Converting complex data types""" in Chapter 3, "XLA and TimesTen Event Management" of the *Oracle TimesTen In-Memory Database C Developer's and Reference Guide*.

Materialized Views

A materialized view is a read-only table that maintains a summary of data selected from one or more TimesTen tables. The summary data in a materialized view is called a *result set* and the TimesTen tables that are queried to make up the result set are called *detail tables*.

By maintaining a result set extracted from the detail tables, materialized views provide applications with faster access to frequently used data. This is because the expensive joins and aggregate operations needed to gather the data from the separate detail tables are done once when the materialized view is created and do not need to be repeated for subsequent application queries.

Figure 8.4 Materialized View



After a materialized view is created, changes made to the data in the detail tables are immediately reflected in the materialized view. The only way to update a materialized view is by changing the underlying detail tables. A materialized view is a read-only table that cannot be updated directly. This means a materialized view cannot be updated by an INSERT, DELETE, or UPDATE statement, by replication, by XLA (using the **ttXlaApply** function), or by the Cache Connect agent (it cannot be part of a cache group).

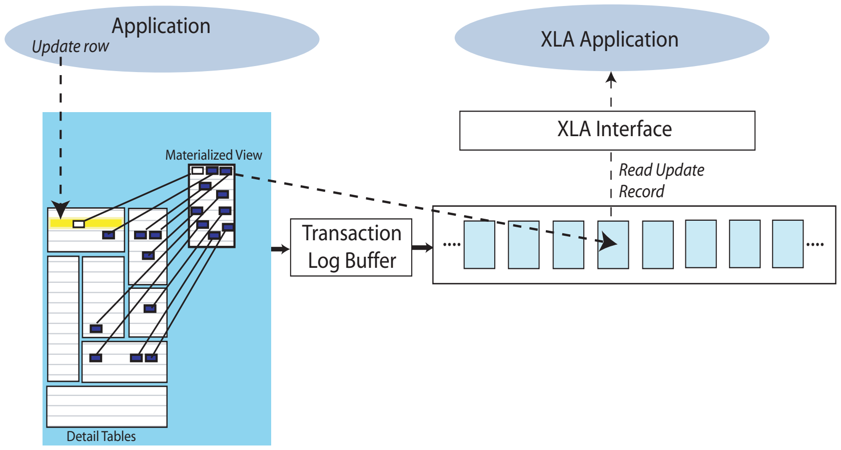
Materialized Views and XLA

In most database systems, materialized views are used to simplify and enhance the performance of SELECT queries that involve multiple tables. Though they offer this same capability in TimesTen, another purpose of materialized views in TimesTen is their role in conjunction with XLA applications as a means to keep track of select rows and columns in multiple tables.

When a materialized view is present, an XLA application only needs to monitor update records that are of interest from a single materialized view table. Without a materialized view, the XLA application would have to monitor all of the update records from all of the detail tables, including records reflecting updates to rows and columns of no interest to the application.

For example, Figure 8.5 shows an update made to a column in a detail table that is part of the materialized view result set. The XLA application monitoring updates to the materialized view captures the updated record. Updates to other columns and rows in the same detail table that are not part of the materialized view result set are not seen by the XLA application.

Figure 8.5 Using XLA to detect updates on a materialized view table



See "Scenario 2: Real-time quote service application" on page 23 for an example of a trading application that makes use of XLA and a materialized view to detect updates to select stocks.

The TimesTen implementation of materialized views emphasizes performance and their function in detecting updates across multiple tables. Readers familiar with other implementations of materialized views should note that the following tradeoffs were made:

- Materialized views must be explicitly created by the application. The TimesTen query optimizer has no facility to automatically create materialized views.
- The TimesTen query optimizer does not rewrite queries on the detail tables to reference materialized views. Application queries must directly reference views, if they are to be used.
- There is no deferred maintenance option for materialized views. A materialized view is refreshed automatically when changes are made to its detail tables and there is no facility for manually refreshing a view.
- There are some restrictions to the SQL used to create materialized views. See CREATE MATERIALIZED VIEW in the *TimesTen SQL Reference Guide* for details.

SNMP Traps

Simple Network Management Protocol (SNMP) is a protocol for network management services. Network management software typically uses SNMP to query or control the state of network devices like routers and switches. These devices sometimes also generate asynchronous alerts in the form of UDP/IP packets, called *SNMP traps*, to inform the management systems of problems.

TimesTen cannot be queried or controlled through SNMP. However, TimesTen sends SNMP traps for certain critical events to facilitate user recovery mechanisms. TimesTen sends traps for the following events:

- Cache Connect to Oracle autorefresh transaction failure
- Data store out of space
- Replicated transaction failure
- Death of daemons
- Data store invalidation
- Assertion failure

These events also cause log entries to be written by the TimesTen daemon, but exposing them through SNMP traps allows properly configured network management software to take immediate action.

For More Information

For more information on XLA, see Chapter 7, “XLA Reference” in the *Oracle TimesTen In-Memory Database API and SQL Reference Guide*.

For more information on Materialized Views, see ““Understanding materialized views”” in Chapter 6, “Working with Data in a TimesTen Data Store of the *Oracle TimesTen In-Memory Database Operations Guide*. Also see CREATE MATERIALIZED VIEW in Chapter 13, “SQL Statements” of the *Oracle TimesTen In-Memory Database API and SQL Reference Guide*.

For more information on SNMP traps, see “Chapter 7, “Diagnostics through SNMP Traps” in the *Oracle TimesTen In-Memory Database API and SQL Reference Guide*.

9

Cache Connect to Oracle

The Cache Connect to Oracle feature allows TimesTen to cache data stored in one or more tables in an Oracle database.

The main topics in this chapter are:

Cache Groups

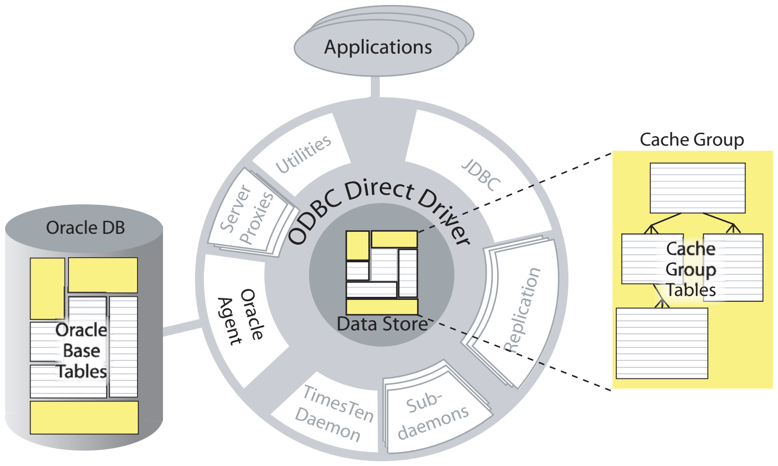
Cache Connect allows you to cache Oracle data by creating a *cache group* in a TimesTen data store. A cache group can be created to cache a single Oracle table or a set of related Oracle tables. The Oracle data cached on TimesTen can consist of all of the rows and columns or a subset of the rows and columns in the Oracle tables.

Cache Connect supports the following features:

- Applications can both read from and write to cache groups
- Cache groups can be refreshed (bring Oracle data into the cache group) automatically or manually
- Cache updates can be sent to the Oracle database automatically or manually

The Oracle tables cached in a TimesTen cache group are referred to as *base tables*. The TimesTen Data Manager interacts with Oracle to perform all of the synchronous cache group operations, such as create a cache group and propagate updates between the cache group and the Oracle database. A TimesTen process called the *cache agent* performs asynchronous cache operations, such as loading data into the cache group, manual refreshing of the data from the Oracle database to the cache group, cache aging, and autorefreshing of the data from the Oracle database to the cache group.

Figure 9.1 Cache Connect cache group

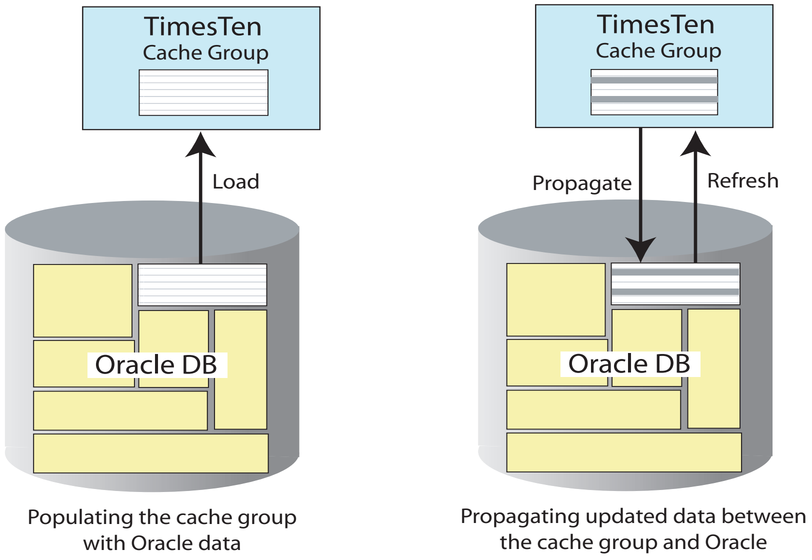


Cache groups can be created and modified by the browser-based *Cache Administrator* or by SQL statements, as described in the *TimesTen Cache Connect to Oracle Guide*.

Loading and Updating Cache Groups

The data from Oracle is initially *loaded* into TimesTen to populate the cache group. After loading the cache group, the cached data can be updated in either the TimesTen cache group or the Oracle database. Cache Connect can automatically *propagate* updates from the cache group to Oracle, as well as *refresh* from Oracle to the cache group.

Figure 9.2 Loading data and propagating/refreshing updates


TimesTen
Cache Group
Load
Oracle DB
Populating the cache group with Oracle data
TimesTen
Cache Group
Propagate
Refresh
Oracle DB
Propagating updated data between the cache group and Oracle

Oracle-to-TimesTen updates (refresh)

The following mechanisms are available to keep the TimesTen cache group up to date with the Oracle base tables:

- *Full Autorefresh* – This is specified by the AUTOREFRESH MODE FULL clause in the CREATE CACHE GROUP statement, which includes an optional INTERVAL value that indicates how frequently refreshes ought to take place. With autorefresh enabled, TimesTen automatically refreshes the entire cache group or cache instances at the specified time intervals.
- *Incremental Autorefresh* – This is specified by the AUTOREFRESH MODE INCREMENTAL clause in the CREATE CACHE GROUP statement. Unlike full autorefresh, an incremental autorefresh updates only the records that have been modified in Oracle since the last refresh. As with full autorefresh, the application can specify an INTERVAL to indicate the frequency in which TimesTen automatically performs the incremental refresh. (Incremental autorefresh is the default behavior for READONLY cache groups, as described in "System-Managed Cache Groups" on page 98.)

- *Manual Refresh* – This is specified by an application issuing an explicit REFRESH CACHE GROUP statement to refresh either an entire cache group or specific cache instances. It is equivalent to an UNLOAD CACHE GROUP operation followed by a LOAD CACHE GROUP operation.

Each refresh mechanism has its advantages and tradeoffs. *Incremental Autorefresh* refreshes only changed rows, but requires the use of triggers on Oracle to keep track of the updates. This adds overhead and slows down updates. *Full Autorefresh* does not require Oracle to keep track of the updates, but updates everything in the cache at once. *Manual Refresh* is a refresh controlled by the application and the logic needed to determine when to refresh adds overhead to the application.

These three refresh mechanisms are useful under different circumstances. For example, a full autorefresh may be the best choice if the Oracle table is updated only once a day and many rows are changed. An incremental autorefresh would be the best choice if the Oracle table is updated often, but only a few rows are changed with each update. A manual refresh would be the best choice if the application logic knows when the refresh should happen.

TimesTen-to-Oracle updates (propagate)

For data that is updated in a TimesTen cache group, the following mechanisms are available to keep the Oracle database up to date with the cache group:

- *Propagate* – This is specified by one of the following methods:
  - Specifying the PROPAGATE option in the CREATE USERMANAGED CACHE GROUP statement.
  - Creating a synchronous writethrough (SWT) cache group with the CREATE SYNCHRONOUS WRITETHROUGH CACHE GROUP statement.
  - Creating an asynchronous writethrough (AWT) cache group with the CREATE ASYNCHRONOUS WRITETHROUGH CACHE GROUP statement.

  With the PROPAGATE option enabled, all modifications to a cache group are automatically propagated to the Oracle database. When the application completes a transaction that has modified one or more cache groups that have the PROPAGATE option enabled or are SWT cache groups, TimesTen first commits the transaction in Oracle and then in TimesTen. This technique allows Oracle to apply any required logic related to the data before it is committed in TimesTen, so that Oracle always reflects the latest image of the data. Use the PROPAGATE option or SWT cache groups when the cache and Oracle must be synchronized at all times.

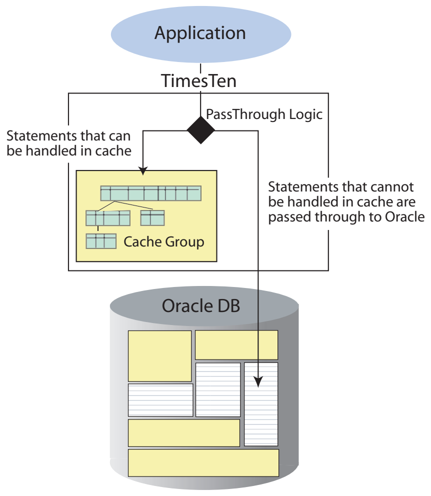
  Modifications to an AWT cache group are committed without waiting for the changes to be applied to Oracle. AWT cache groups provide better response times and performance, but the cache and Oracle do not always contain the same data because changes are applied to Oracle asynchronously.

- *Flush* – This is specified by an application issuing an explicit FLUSH CACHE GROUP statement. A flush operation can be used to manually propagate updates from the TimesTen cache groups to Oracle. Flush operations are useful when frequent updates occur for a limited period of time over a set of records. Rather than propagate the updates of each transaction individually, the updates are batched all at once with the flush operation. Flush operations require that the PROPAGATE option in the CREATE CACHE GROUP statement be disabled (NOT PROPAGATE). Flush operations do not propagate deletes.

Passthrough Feature

TimesTen applications can send SQL statements to either a TimesTen cache group or to the Oracle database through a single connection to a TimesTen data store. This single-connection capability is enabled by a *passthrough* feature that checks if the SQL statement can be handled locally by the cached tables in TimesTen or if it must be redirected to Oracle, as shown in Figure 9.3. The passthrough feature provides settings that specify what types of statements are to be passed through and under what circumstances.

Figure 9.3 TimesTen Passthrough feature


Application
TimesTen
PassThrough Logic
Statements that can be handled in cache
Cache Group
Statements that cannot be handled in cache are passed through to Oracle
Oracle DB

The specific behavior of the passthrough feature is controlled by the data store attribute, **PassThrough**, and can be reset programmatically by calling the **ttOptSetFlag** procedure with the *PassThrough* flag.

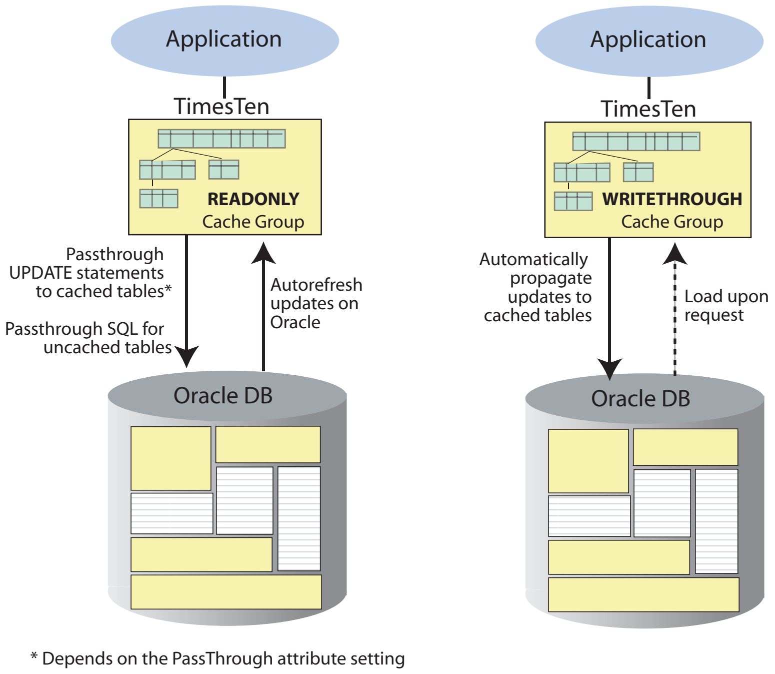
System-Managed Cache Groups

System-managed cache groups are predefined frameworks for caching Oracle data in TimesTen. When using a system-managed cache group, operations such as propagating data are managed automatically by the cache agent.

As shown in Figure 9.4, there are two basic types of system-managed cache groups:

- READONLY cache groups hold read-only data, so updates on the tables in the cache group are not allowed. Updates must be done on Oracle. These updates can either be done directly in Oracle or in TimesTen using the passthrough feature described in "Passthrough Feature" on page 97. By default, a READONLY cache group is automatically refreshed from Oracle.
- WRITETHROUGH cache groups load the cached table data from Oracle once when created. Thereafter, all updates to the cache group are automatically propagated to Oracle. Writethrough cache groups can be either asynchronous (AWT) or synchronous (SWT). SWT cache groups wait for a commit on Oracle before committing in the cache. AWT cache groups commit changes in the cache without waiting for a commit on Oracle.

Figure 9.4 Read-only and writethrough cache groups

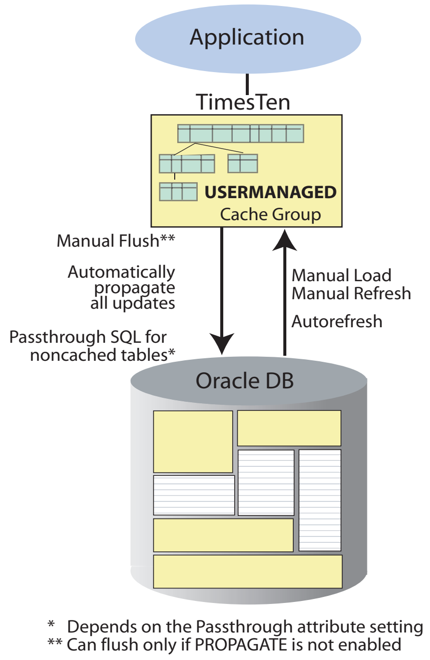


Application
TimesTen
READONLY
Cache Group
Passthrough UPDATE statements to cached tables*
Passthrough SQL for uncached tables
Autorefresh updates on Oracle
Oracle DB
Application
TimesTen
WRITETHROUGH
Cache Group
Automatically propagate updates to cached tables
Load upon request
Oracle DB
* Depends on the PassThrough attribute setting

User-Managed Cache Groups

User-managed cache groups allow users to select from a full range of attributes and SQL statements to define customized caching behaviors. Users have full control over how and when the cached data is loaded, propagated, and removed from the cache group.

Figure 9.5 User-managed cache group


Application
TimesTen
USERMANAGED
Cache Group
Manual Flush**
Automatically propagate all updates
Passthrough SQL for noncached tables*
Manual Load
Manual Refresh
Autorefresh
Oracle DB
* Depends on the Passthrough attribute setting
** Can flush only if PROPAGATE is not enabled

Cache Instances

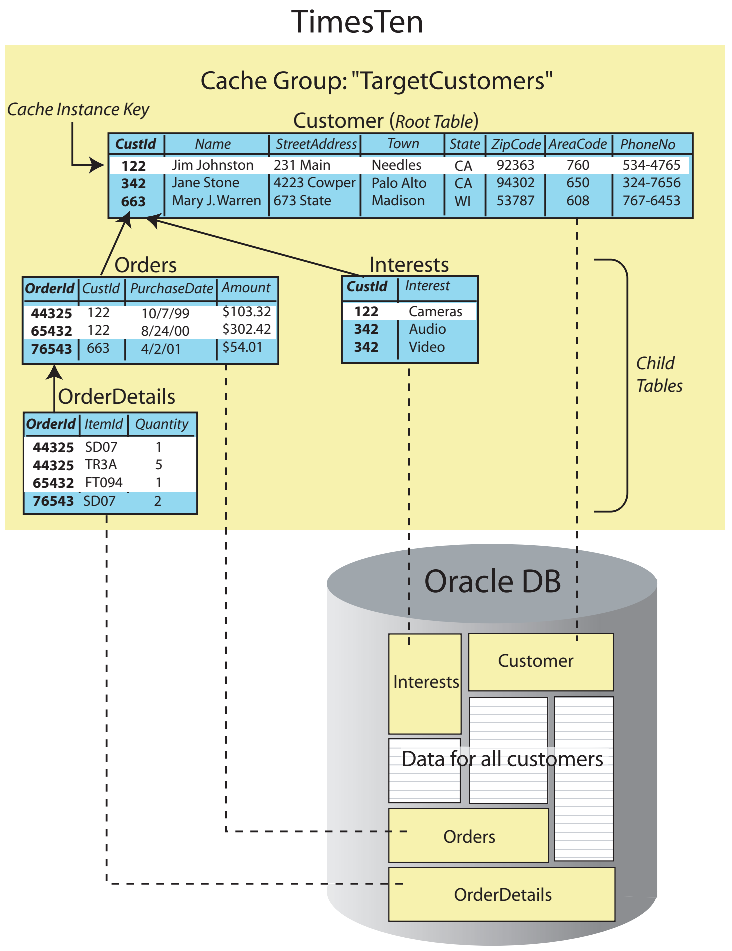
In contrast to system-managed cache groups, in which changes are propagated between Oracle and the cache group at table-level granularity, user-managed cache groups can propagate load, refresh, and flush operations for rows of data collectively referred to as *cache instances*.

As shown in Figure 9.6, each cache group has a *root table* that contains the primary key for the cache group. Rows in the root table may have one-to-many relationships with rows in *child tables*, each of which may have one-to-many relationships with rows in other child tables.

A *cache instance* is the set of rows that are associated by foreign key relationships with a particular row in the cache group root table. Each primary key value in the root table specifies a cache instance. Cache instances form the unit of cache loading and cache aging, as described in "Cache Loading Techniques" on page 101. Exactly one of the rows in a cache instance must be from the root table. No table in the cache group can be a child to more than one parent in the cache group. These constraints ensure that each TimesTen record belongs to only one cache instance and has only one parent in its cache group. This disambiguates the aging of cache instances. The foreign key constraints prevent the creation of dangling child records through updates or deletes to the parent or through inserts of child records that have no parent.

In the example shown in Figure 9.6, all records in the *Customer*, *Orders*, and *Interests* tables that belong to a given customer id (*CustId*) are related to each other through foreign key constraints, so they belong to the same cache instance. Because *CustId* uniquely identifies the cache instance, it is referred to as the *cache instance key*.

Figure 9.6 TargetCustomers cache group



CustId	Name	StreetAddress	Town	State	ZipCode	AreaCode	PhoneNo
122	Jim Johnston	231 Main	Needles	CA	92363	760	534-4765
342	Jane Stone	4223 Cowper	Palo Alto	CA	94302	650	324-7656
663	Mary J.Warren	673 State	Madison	WI	53787	608	767-6453

OrderId	CustId	PurchaseDate	Amount
44325	122	10/7/99	$103.32
65432	122	8/24/00	$302.42
76543	663	4/2/01	$54.01

CustId	Interest
122	Cameras
342	Audio
342	Video

OrderId	ItemId	Quantity
44325	SD07	1
44325	TR3A	5
65432	FT094	1
76543	SD07	2

Cache Loading Techniques

When using a user-managed cache group, an application can select from the following techniques to load the cache group data from the Oracle database into TimesTen:

- Load the entire cache group at once. This is a suitable technique to use if the content of the entire cache group can fit in memory. TimesTen also provides the ability to unload an entire cache group.
- Load cache instances by their cache instance key. In this case, the records that make up a cache instance are brought into TimesTen on demand. If the cache instance is already loaded in TimesTen, then the request is ignored. This technique is useful if the capacity of the cache is not large enough to hold all the data described by the cache group. TimesTen also provides the ability to unload cache instances by their cache instance key.
- Load cache instances by WHERE clause. This technique is similar to loading a cache instance by its cache instance key, but in this case, the cache instances are described by a WHERE clause rather than by ID. TimesTen also provides the ability to unload cache instances by WHERE clause.

Note: You can also load and refresh data with or without logging enabled. See “Logging options” on page 70.

Cache Instance Aging

When cache instances are loaded into TimesTen, they can be automatically aged out of TimesTen when its caching capacity reaches a specified threshold. TimesTen provides applications with a number of cache-aging options to set up different aging durations for different cache instances, as well as to specify that certain cache instances should never be aged out. For example, the application may want to keep catalog information in the cache all the time, but may want to load user profiles only when users connect to the application and to automatically age out profiles sometime after users disconnect.

Replicating Cache Groups

TimesTen replication can be configured to replicate a cache group from one data store to a cache group in another data store. Only cache groups of the same type can be replicated. For example, a SYNCHRONOUS WRITETHROUGH cache group can be bidirectionally replicated with other SYNCHRONOUS WRITETHROUGH cache groups, and a READONLY cache group can be bidirectionally replicated with other READONLY cache groups.

You can also replicate a table in a cache group to a regular TimesTen table.

When TimesTen cache groups are replicated to another TimesTen data store, only updates to the cache group from the application connected to the data store are propagated to Oracle. Updates to a cache group that are replicated from a cache group on another data store are not propagated to Oracle.

For More Information

For more information on Cache Connect, see the *TimesTen Cache Connect to Oracle Guide*.

10

TimesTen Administration

TimesTen provides tools to perform a variety of administrative tasks, such as:

- Installing TimesTen
- TimesTen Access Control
- Command Line Administration
- SQL Administration
- Browser-Based Administration
- Upgrading TimesTen

Installing TimesTen

TimesTen software is easily installed in minutes. On UNIX systems, TimesTen is installed by a simple set-up script. On Windows, TimesTen is installed by running InstallShield®. The installation can include the TimesTen client, server, and data manager components, or just the TimesTen client or data manager by themselves.

TimesTen can be installed with Access Control enabled or disabled, as described next in "TimesTen Access Control".

See the *Oracle TimesTen In-Memory Database Installation Guide* for details.

TimesTen Access Control

TimesTen can be installed with *Access Control* enabled to allow only users with specific privileges to access particular TimesTen features.

TimesTen Access Control uses standard SQL operations, such as CREATE USER, DROP USER, GRANT, and REVOKE, to establish TimesTen user accounts with specific privilege levels. However, in TimesTen, privileges are granted for the entire TimesTen instance, rather than for specific tables. Each user's privileges apply to all data stores in a TimesTen instance or installation.

If Access Control is to be enabled, installation must be performed by the instance administrator for the data store. The instance administrator owns all files in the installation directory tree and is the only user with the privileges to start and stop the TimesTen daemon and other TimesTen processes, such as the replication and cache agents.

To enable client connections to an access-controlled data store, you must enable the **Authenticate** attribute and the client must specify the correct user name and password in the client DSN or when connecting to TimesTen.

See Chapter 1, "Access Control and non-root installations" in the *Oracle TimesTen In-Memory Database Installation Guide* for more information.

The Access Control feature of TimesTen provides an environment of basic control for applications that use the defined privileges. Access Control does not provide definitive security for all processes that might be able to access the data store. For example, this feature does not protect the data store from user processes that may have sufficient privileges to attach to the data store when in memory or that can access files on disk that are associated with the data store, such as log files and checkpoint files.

Command Line Administration

Most TimesTen administration tasks are done through command line utilities, which include:

Name	Description
ttAdmin	A general utility for managing TimesTen data stores. Used to specify policies to automatically or manually load and unload data stores from RAM, as well as to start and stop TimesTen cache agents and replication agents.
ttBackup and **ttRestore**	Used to create a backup copy of a data store and restore it at a later time.
ttBulkCp	Used to transfer data between TimesTen tables and ASCII files.
ttIsql	Used to run SQL interactively from the command line. Also provides a number of administrative commands to reconfigure and monitor data stores.
ttMigrate	Used to save and restore TimesTen tables and cache group definitions to and from a binary data file.
ttRepAdmin	Used to monitor replication status. Previous versions of TimesTen used this utility to configure replication schemes and initiate many replication maintenance operations. However, **ttRepAdmin** capabilities other than those related to monitoring replication are being deprecated in favor of the SQL interface described in the *TimesTen to TimesTen Replication Guide*.
ttSize	Used to estimate the amount of space to allocate for a table in the data store.
ttStatus	Used to display information that describes the current state of TimesTen.
ttTail	Used to obtain the TimesTen internal trace information from a data store and display to `stdout`.
ttTraceMon	Used to enable and disable the TimesTen internal tracing facilities.
ttXactAdmin	Used to list ownership, status, log and lock information for each outstanding transaction. The **ttXactAdmin** utility also allows users to heuristically commit, abort or forget an XA transaction branch.

SQL Administration

TimesTen provides SQL statements for administrative activities, such as:

- creating and dropping indexes
- creating and managing tables
- creating and managing replication schemes
- creating and managing cache groups
- creating and managing materialized views

The metadata for each TimesTen data store is stored in a group of system tables. Applications can use SQL SELECT queries on these tables to monitor the current state of a data store. See Chapter 4, "System and Replication Tables" in the *TimesTen Reference Guide* for details on the TimesTen tables.

Administrators can use the **ttIsql** utility for casual SQL interaction with a data store. For example, there are several built-in **ttIsql** commands that display information on data store structures. These commands include:

- **monitor** to display a summary of the current state of the data store (contents of the SYS.MONITOR table).
- **describe** to display information on tables, prepared statements, and procedures.
- **tables** to display information on tables.
- **indexes** to display information on table indexes.
- **cachegroup** to display the attributes of cache groups.
- **dssize** to report the current sizes of the permanent and temporary data store partitions.

Browser-Based Administration

TimesTen provides a Web browser-based tool, called the *Cache Administrator*, that can be used by any machine within the intranet to create and manage Cache Connect cache groups. The Cache Administrator can be used to create a cache group directly or to generate a SQL file that defines a cache group.

The Cache Administrator enables users to set up one or more cache groups by navigating through the Oracle schema with a Web browser. Users can use the administrator to selectively load and unload tables and indexes to and from the Cache Connect data store.

See the *TimesTen Cache Connect to Oracle Guide* for more information.

Upgrading TimesTen

TimesTen provides the facilities to perform three types of upgrades:

- In-place upgrades
- Offline upgrades
- Online upgrades

These upgrade options are summarized below. The detailed procedures are described in Chapter 3, “Data Store Upgrades” of the *Oracle TimesTen In-Memory Database Installation Guide*.

In-place upgrades

In-place upgrades are typically used to move to a new patch release (or “dot-dot release”) of TimesTen.

In-place upgrades can be done without destroying the existing data stores. However, all applications must first disconnect from the data stores and the data stores must be unloaded from shared memory. After uninstalling the old release of TimesTen and installing the new release, applications can reconnect to the data stores and resume operation.

Offline upgrades

Offline upgrades involve using the **ttMigrate** utility to export the data store into an external file and to restore the data store with the desired changes.

Offline upgrades are used to:

- move to a new major TimesTen release or a dot release
- move to a different directory or machine
- reduce data store size

During an offline upgrade, the data store is not available to applications. Offline upgrades usually require enough disk space for an extra copy of the upgraded data store.

Online upgrades

TimesTen replication enables online upgrades, which can be performed online by the **ttMigrate** and **ttRepAdmin** utilities while the data store and its applications remain operational and available to users. Online upgrades are useful for applications where continuous availability of the data store is critical.

Online upgrades are used to:

- move to a new major release (or "dot release") of TimesTen and retain continuous availability to the data store
- increase or reduce the data store size
- move the data store to a new location or machine

Updates made to the data store during the upgrade are transmitted to the upgraded data store at the end of the upgrade process. Because an online upgrade requires that the data store be replicated to another data store, it can require more memory and disk space than offline upgrades.

For More Information

For more information on installing and upgrading TimesTen, see the *Oracle TimesTen In-Memory Database Installation Guide*.

For more information on general administration of TimesTen, see Chapter 2, "Creating TimesTen Data Stores" and Chapter 6, "Working with Data in a TimesTen Data Store" in the *Oracle TimesTen In-Memory Database Operations Guide*.

For more information on administration of Cache Connect, see the *TimesTen Cache Connect to Oracle Guide*.

For more information on administration of TimesTen replication, see the *TimesTen Replication Guide*.

For a complete list of SQL statements, see Chapter 13, "SQL Statements" in the *Oracle TimesTen In-Memory Database API and SQL Reference Guide*.

For a complete list of TimesTen command-line utilities, see Chapter 2, "Utilities" in the *Oracle TimesTen In-Memory Database API and SQL Reference Guide*.

Index