Oracle® In-Memory Database Cache

Introduction

11*g* Release 2 (11.2.2)

E21631-03

December 2011

ORACLE®

Oracle In-Memory Database Cache Introduction, 11*g* Release 2 (11.2.2)

E21631-03

Contents

2 Using TimesTen and IMDB Cache

3 Oracle In-Memory Database Cache Architecture and Components

4 Concurrent Operations

5 Query Optimization

6 Data Availability and Integrity

7 Event Notification

8 IMDB Cache

9 TimesTen and IMDB Cache Administration

Index

Preface

This guide provides an introduction to the Oracle In-Memory Database Cache.

Audience

This document is intended for readers with a basic understanding of database systems.

Related documents

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

http://www.oracle.com/technetwork/database/timesten/documentation

Conventions

TimesTen supports multiple platforms. Unless otherwise indicated, the information in this guide applies to all supported platforms. The term Windows refers to all supported Windows platforms. The term UNIX applies to all supported UNIX platforms and Linux. See "Platforms" in *Oracle TimesTen In-Memory Database Release Notes* for specific platform versions supported by TimesTen.

> **Note:** In TimesTen documentation, the terms "data store" and "database" are equivalent. Both terms refer to the TimesTen database unless otherwise noted.

This document uses the following text conventions:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

Convention	Meaning
italic monospace	Italic monospace type indicates 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.
[ ]	Square brackets indicate that an item in a command line is optional.
{ }	Curly braces indicated that you must choose one of the items separated by a vertical bar (|) in a command line.
|	A vertical bar (or pipe) separates alternative arguments.
. . .	An ellipsis (. . .) after an argument indicates that you may use more than one argument on a single command line.
%	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:

Convention	Meaning
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.
bits or *bb*	Two digits, either 32 or 64, that represent either the 32-bit or 64-bit operating system.
release or *rr*	Three numbers that represent the first three numbers of the TimesTen release number, with or without a dot. For example, 1121 or 11.2.1 represents TimesTen Release 11.2.1.
jdk_version	Two digits that represent the version number of the major JDK release. Specifically, 14 represent JDK 1.4; 5 represents JDK 5.
DSN	The data source name.

Documentation Accessibility

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

Access to Oracle Support

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

What's New

This section summarizes the new features of Oracle TimesTen In-Memory Database release 11.2.2 that are described in this guide. It provides links to more information.

New features in Release 11.2.2.2.0

- You can increase replication throughput by configuring parallel replication at database creation time. See "Replication" on page 6-3.

New features in Release 11.2.2.1.0

- You can increase throughput from asynchronous writethrough (AWT) cache groups to the Oracle database by configuring parallel propagation at database creation time. See "Updating Oracle tables from a cache group" on page 8-5.

New features in Release 11.2.2.0.0

- TimesTen provides the ability to compress tables at the column level, thus storing the data more efficiently. See "In-memory columnar compression" on page 1-7.
- TimesTen provides analytic SQL functions, aggregate SQL functions and OLAP operators. See "Business intelligence and online analytical processing" on page 1-7.
- TimesTen supports large objects (LOBs) in tables that are not cached in the IMDB Cache. See "Large objects" on page 1-7.

1

Oracle TimesTen In-Memory Database and Oracle In-Memory Database Cache

Oracle TimesTen In-Memory Database (TimesTen) is a memory-optimized relational database that empowers applications with the responsiveness and high throughput required by today's real-time enterprises and industries such as telecom, capital markets and defense. Oracle In-Memory Database Cache (IMDB Cache) uses the Oracle TimesTen In-Memory Database as its RDBMS engine. Deployed in the application tier as an embedded database, Oracle TimesTen In-Memory Database operates on databases that fit entirely in physical memory using standard SQL interfaces. High availability for the in-memory database is provided through real-time transactional replication.

Oracle In-Memory Database Cache (IMDB Cache) is an Oracle Database product option ideal for caching performance-critical subsets of an Oracle database for improved response time in the application tier. Cache tables can be read-only or updatable. Applications read and update the cache tables using standard SQL, and data synchronization between the cache and the Oracle Database is performed automatically. Oracle In-Memory Database Cache offers applications the full generality and functionality of a relational database, the transparent maintenance of cache consistency with the Oracle Database, and the real-time performance of an in-memory database.

Oracle TimesTen In-Memory Database and IMDB Cache deliver real-time performance by changing the assumptions about where data resides at run time. By managing data in memory and optimizing data structures and access algorithms accordingly, database operations execute with maximum efficiency, achieving dramatic gains in responsiveness and throughput, even compared with a fully cached, disk-based relational database management system (RDBMS). TimesTen and IMDB Cache libraries are also embedded within applications, eliminating context switching and unnecessary network operations, further improving performance.

Following the standard relational data model, you can use SQL, JDBC, ODBC, PL/SQL and Oracle Call Interface (OCI) to access TimesTen and IMDB Cache databases. Using SQL to shield applications from system internals allows databases to be altered or extended without impacting existing applications. New services can be quickly added into a production environment simply by adding application modules, tables and columns. As with any mainstream RDBMS, a cost-based optimizer automatically determines the fastest way to process queries and transactions. Any developer familiar with the Oracle Database or SQL interfaces can be immediately productive developing real-time applications with TimesTen and IMDB Cache.

Why is Oracle TimesTen In-Memory Database fast?

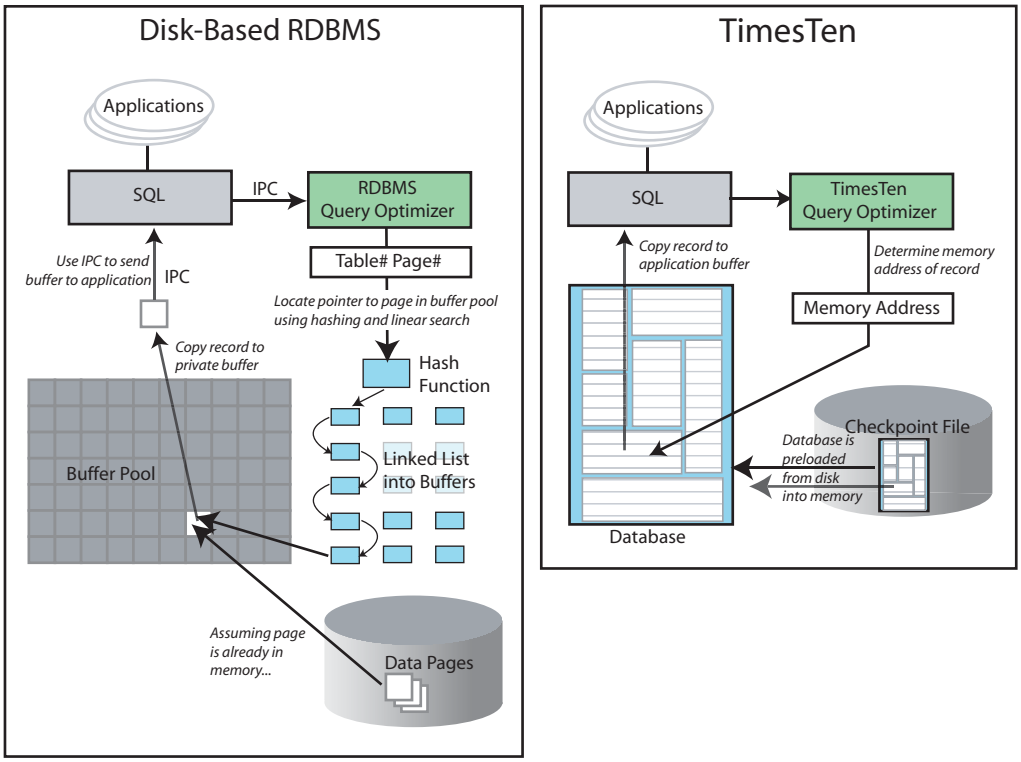
Much of the work that is done by a conventional, disk-optimized RDBMS is done under the assumption that data primarily resides on disk. Optimization algorithms, buffer pool management, and indexed retrieval techniques are designed based on this fundamental assumption.

Even when a disk-based RDBMS has been configured to hold all of its data in main memory, its performance is hobbled by assumptions of disk-based data residency. These assumptions cannot be easily reversed because they are hard-coded in processing logic, indexing schemes, and data access mechanisms.

TimesTen is designed with the knowledge that data resides in main memory and can take more direct routes to data, reducing the length of the code path and simplifying algorithms and structure.

When the assumption of disk-residency is removed, complexity is dramatically reduced. The number of machine instructions drops, buffer pool management disappears, extra data copies are not needed, index pages shrink, and their structure is simplified. The design becomes simple and more compact, and requests are executed faster. Figure 1–1 shows the simplicity of the TimesTen design.

Figure 1–1 Comparing a disk-based RDBMS to TimesTen



In a conventional disk-based RDBMS, client applications communicate with a database server process over some type of IPC connection, which adds 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. Traditional client/server access is also supported for functions such as reporting, or when a large number of application-tier platforms must share access to a common in-memory database. From

an application's perspective, the TimesTen API is identical whether it is a direct connection or a client/server connection.

TimesTen and IMDB Cache feature overview

TimesTen and IMDB Cache have many familiar database features as well as some unique features. This section includes the following topics:

TimesTen API support

The run time architecture of TimesTen supports connectivity through the ODBC, JDBC, OCI, Pro*C/C++ Precompiler and ODP.NET APIs. TimesTen also provides built-in procedures, utilities and the TTClasses API (C++) that extend ODBC, JDBC and OCI functionality for TimesTen-specific operations, as well as supporting PL/SQL. API support is described in subsequent sections.

ODBC and JDBC interfaces

TimesTen and IMDB Cache support ODBC and JDBC. Unlike many other database systems, where ODBC 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.

For more information, see *Oracle TimesTen In-Memory Database C Developer's Guide* and *Oracle TimesTen In-Memory Database Java Developer's Guide*.

SQL and PL/SQL functionality

TimesTen and IMDB Cache support extensive SQL functionality as well as SQL extensions to simplify the configuration and management of special features such as replication and IMDB Cache.

TimesTen and IMDB Cache support PL/SQL (Procedural Language Extension to SQL), a programming language that enables you to integrate procedural constructs

with SQL for a TimesTen or IMDB Cache database. You can execute PL/SQL from all supported APIs.

For more information, see *Oracle TimesTen In-Memory Database SQL Reference* and *Oracle TimesTen In-Memory Database PL/SQL Developer's Guide*

OCI and Pro*C/C++ Precompiler support

TimesTen and IMDB Cache support the Oracle Call Interface (OCI) and the Pro*C/C++ Precompiler for TimesTen functionality.

TimesTen OCI depends on the Oracle client library and the TimesTen ODBC libraries. TimesTen OCI support enables you to run many existing OCI applications with TimesTen in direct mode or client/server mode. TimesTen OCI also enables you to use other Oracle products, including the Pro*C/C++ Precompiler, that use OCI as a database interface. You can call PL/SQL from OCI applications.

For more information, see *Oracle TimesTen In-Memory Database C Developer's Guide*.

ODP.NET support

Oracle Data Provider for .NET (ODP.NET) is an implementation of the Microsoft ADO.NET interface. ODP.NET support for TimesTen and IMDB Cache provides fast and efficient ADO.NET data access from .NET and C# client applications to TimesTen databases.

For more information, see *Oracle Data Provider for .NET Oracle TimesTen In-Memory Database Support User's Guide*.

Transaction Log API

TimesTen and IMDB Cache have an API that allows applications to monitor update activities in order to generate actions outside the database. In TimesTen and IMDB Cache, this capability is provided by the Transaction Log API (XLA), which allows applications to monitor update records as they are committed and take various actions based on the detected updates. For example, an XLA application can apply the detected updates to another database, which could be TimesTen or a disk-based RDBMS. Another type of XLA application can notify subscribers that an update of interest has taken place. This API is supported for C, Java (JMS/XLA) and TTClasses.

TimesTen and IMDB Cache provide materialized views that can be used with XLA to enable notification of events described by SQL queries. The primary purpose of XLA is to be a high performance, asynchronous alternative to triggers.

For more information, see "Transaction Log API" on page 7-1 and *Oracle TimesTen In-Memory Database C Developer's Guide*.

TTClasses

TimesTen C++ Interface Classes (TTClasses) is more convenient than ODBC while maintaining fast performance. This C++ class library provides wrappers around the most common ODBC functionality. The TTClasses library is also intended to promote best practices when writing application software.

For more information, see *Oracle TimesTen In-Memory Database TTClasses Guide*.

Distributed Transaction Processing 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 JTA interfaces is intended to enable Java applications, application servers, and transaction managers to use TimesTen resource managers in DTP environments.

For more information, see *Oracle TimesTen In-Memory Database C Developer's Guide* and *Oracle TimesTen In-Memory Database Java Developer's Guide*.

Access Control

TimesTen and IMDB Cache are installed with access control to allow only users with specific privileges to access particular TimesTen features. TimesTen Access Control uses standard SQL operations to establish user accounts with specific privileges. TimesTen supports TimesTen offers object-level access control as well as database-level access control.

For more information, see *Oracle TimesTen In-Memory Database Operations Guide*.

Database connectivity

TimesTen and IMDB Cache support direct driver connections for higher performance, as well as connections through a driver manager. TimesTen also supports client/server connections.

These connection options allow users to choose the best tradeoff between performance and functionality for their applications. Direct driver connections are fastest. Client/server connections may provide more flexibility. Driver manager connections can provide support for ODBC applications written for a different ODBC version or for multiple RDBMS products with ODBC interfaces.

See "TimesTen connection options" on page 3-4.

Durability

TimesTen and IMDB Cache achieve durability through a combination of transaction logging and periodic refreshes of a disk-resident version of the database. Log records are written to disk asynchronously or synchronously to the completion of the transaction and controlled by the application at the transaction level. For systems where maximum throughput is paramount, such as non-monetary transactions within network systems, asynchronous logging allows extremely high throughput with minimal exposure. In cases where data integrity must be preserved, such as securities trading, TimesTen and IMDB Cache ensure complete durability, with no loss of data.

TimesTen uses the transaction log in the following situations:

- Recover transactions if the application or database fails
- Undo transactions that are rolled back
- Replicate changes to other TimesTen databases
- Replicate TimesTen changes to Oracle tables
- Enable applications to detect changes to tables (using the XLA API)

TimesTen and IMDB Cache maintain the disk-resident version of the database with a checkpoint operation that takes place in the background and has very little impact on

database applications. This operation is called a "fuzzy" checkpoint and is performed automatically. TimesTen and IMDB Cache also have a blocking checkpoint that does not require transaction log files for recovery. Blocking checkpoints must be initiated by the application. TimesTen and IMDB Cache maintain two checkpoint files in case a failure occurs mid-checkpoint. Checkpoint files should reside on disks separate from the transaction logs to minimize the impact of checkpointing on application activity.

See the following sections for more information:

- "Transaction logging" on page 6-1
- "When are transaction log files deleted?" on page 6-2
- "Checkpointing" on page 6-2

Query optimization

TimesTen and IMDB Cache have a cost-based query optimizer that chooses the best query plan based on factors such as the presence of indexes, the cardinality of tables and the presence of `ORDER BY` clauses in the query.

Optimizer cost sensitivity is somewhat higher in TimesTen and IMDB Cache than in disk-based systems because 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 and IMDB Cache, the optimization cost model includes factors not usually considered by optimizers for disk-based systems, such as the cost of evaluating predicates.

TimesTen and IMDB Cache provide range, hash and bitmap indexes and support two types of join methods (nested-loop and merge-join). The optimizer can create temporary indexes as needed.

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

See "Query Optimization" on page 5-1 for more information about the query optimizer and indexing techniques.

Concurrency

TimesTen and IMDB Cache provide full support for shared databases. Options are available so users can choose the optimum balance between response time, throughput and transaction semantics for an application.

Read-committed isolation provides nonblocking operations and is the default isolation level. For databases with extremely strict transaction semantics, serializable isolation is available. These isolation levels conform to the ODBC standard and are implemented with optimal performance in mind. As defined by the ODBC standard, a default isolation level can be set for a TimesTen or IMDB Cache database, which can be dynamically modified for each connection at run time.

For more information about managing concurrent operations in TimesTen and IMDB Cache, see "Concurrent Operations" on page 4-1.

Automatic data aging

Data aging is an operation to remove data that is no longer needed. There are two general types of data aging: removing old data based on some time value or removing data that has been least recently used (LRU). For example, you can remove yesterday's

price list, remove profiles and preferences of users who have logged out from the system, or remove detailed records that are more than 2 days old.

Two types of automatic data aging capability for TimesTen database tables and IMDB Cache data are available:

- Time-based data aging based on timestamp values
- Usage-based data aging based on the LRU algorithm

For more information, see "Implementing aging in your tables" in *Oracle TimesTen In-Memory Database Operations Guide* and "Implementing aging in a cache group" in *Oracle In-Memory Database Cache User's Guide*.

Globalization support

TimesTen and IMDB Cache provide globalization support for storing, retrieving, and processing data in native languages. Over 50 different national, multinational, and vendor-specific character sets including the most popular single-byte and multibyte encodings, plus Unicode, are supported as the database storage character set. The connection character set can be defined to enable an application running in a different encoding to communicate to the TimesTen or IMDB Cache database; character set conversion between the application and the database occurs automatically and transparently.

TimesTen and IMDB Cache offer linguistic sorting capabilities that handle the complex sorting requirements of different languages and cultures. More than 80 linguistic sorts are provided. They can be extended to enable the application to perform case-insensitive and accent-insensitive sorting and searches.

For more information, see "Globalization Support" in *Oracle TimesTen In-Memory Database Operations Guide*.

In-memory columnar compression

TimesTen provides the ability to compress tables at the column level, thus storing the data more efficiently. This mechanism provides space reduction for tables by eliminating the redundant storage of duplicate values within columns and improves the performance of SQL queries that perform full table scans.

For more information, see "CREATE TABLE" in *Oracle TimesTen In-Memory Database SQL Reference*.

Business intelligence and online analytical processing

TimesTen and IMDB Cache provide analytic SQL functions and aggregate SQL functions for business intelligence and similar applications. Analytic functions enable analysts and decision makers to make comparisons and identify trends. They include ranking, cumulative, moving, centered and reporting functions.

The `GROUP BY` SQL clause includes online analytical processing (OLAP) operators such as `GROUPING SETS`, `CUBE` and `ROLLUP`.

For more information, see "Functions" and "SQL Statements" in *Oracle TimesTen In-Memory Database SQL Reference*.

Large objects

TimesTen supports large objects (LOBs) in tables that are not cached in the IMDB Cache. TimesTen supports the `BLOB`, `CLOB` and `NCLOB` data types.

For more information, see "Data Types" in *Oracle TimesTen In-Memory Database SQL Reference*.

Administration and utilities

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

Many administrative activities are available by using SQL extensions. TimesTen and IMDB Cache also use SQL extensions to set up replication, caching from an Oracle database and materialized views.

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 `SYS.SYSTEMSTATS` table records many statistics that are useful for analyzing or debugging a TimesTen application.

For more information on TimesTen administration, see "TimesTen and IMDB Cache Administration" on page 9-1.

Replication

TimesTen and IMDB Cache replication enable real-time data replication between servers for high availability and load sharing. Data replication configurations can be active-standby or active-active, using asynchronous or synchronous transmission, with conflict detection and resolution and automatic resynchronization after a failed server is restored.

See "Replication" on page 6-3.

IMDB Cache

The Oracle In-Memory Database Cache creates a real-time, updatable cache for Oracle data. It offloads computing cycles from Oracle databases and enables responsive and scalable real-time applications. IMDB Cache loads a subset of Oracle tables into a cache database. It can be configured to propagate updates in both directions and to automate passthrough of SQL requests for uncached data. It automatically resynchronizes data after failures.

See "IMDB Cache" on page 8-1.

2

Using TimesTen and IMDB Cache

This chapter describes how TimesTen and IMDB Cache can be used to enable applications that require real-time access to data. It includes the following sections:

Uses for TimesTen

TimesTen can be used as:

- The *primary database* for real-time applications. All data needed by the applications resides in the TimesTen database.
- 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 database to provide a central repository for the data of immediate interest to the applications.

Uses for IMDB Cache

IMDB Cache can be used as:

- A *real-time data manager* for specific tasks in an overall workflow in collaboration with a disk-based RDBMS like the Oracle Database. For example, a phone billing application may capture and store recent call records in the IMDB Cache database while storing information about customers, their billing addresses and credit information in an Oracle database. It can also age and keep archives of all call records in the Oracle database. Thus the information that requires real-time access is stored in the IMDB Cache database while the information needed for longer-term analysis, auditing, and archival is stored in the Oracle database.

- A *read-only cache*. Oracle data can be cached in an IMDB Cache read-only cache group. Read-only cache groups are automatically refreshed when Oracle tables are updated. A read-only cache group provides fast access to reference data such as look-up tables and subscriber profiles.
- An *updatable cache*. Oracle data can be cached in IMDB Cache updatable cache groups. Transactions on the cache groups can be committed synchronously or asynchronously to the associated Oracle tables.
- A *distributed cache*. Oracle data can be cached in multiple installations of IMDB Cache running on different machines to provide scalability. You can configure a dynamic distributed cache in which records are loaded automatically and aged automatically.

TimesTen application scenario

This section describes an application scenario to illustrate how TimesTen can be integrated as part of a data management solution.

The application scenario is a Real-time quote service application. It 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.

Real-time quote service application

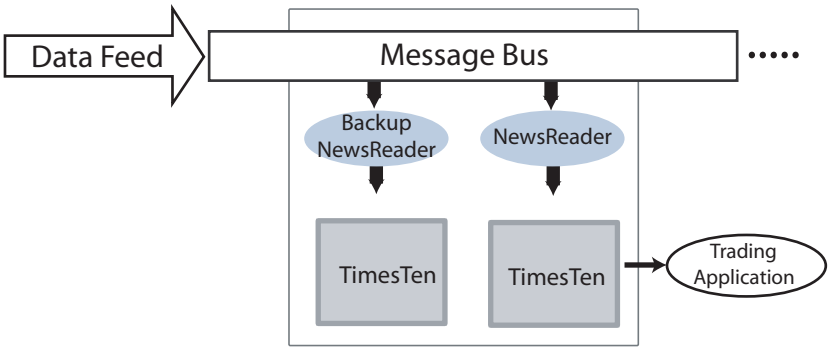
A financial services company is adding a real-time quote and news service to its online trading facility. The real-time quote service will read an incoming news wire from a major market data vendor and make a subset of the data available to trading applications that manage the automated trading operations for the company. The company plans to build an infrastructure that can 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 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 database. In this way, the message bus is used to fork incoming data to two TimesTen databases for redundancy. In this scenario, forking the data from the message bus is more efficient than using TimesTen replication.

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 if a failure occurs. 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 or cellular phone.

Figure 2–1 shows the configuration for capturing data from a message bus and feeding it to NewsReaders.

Figure 2–1 Capturing feed data from a message bus

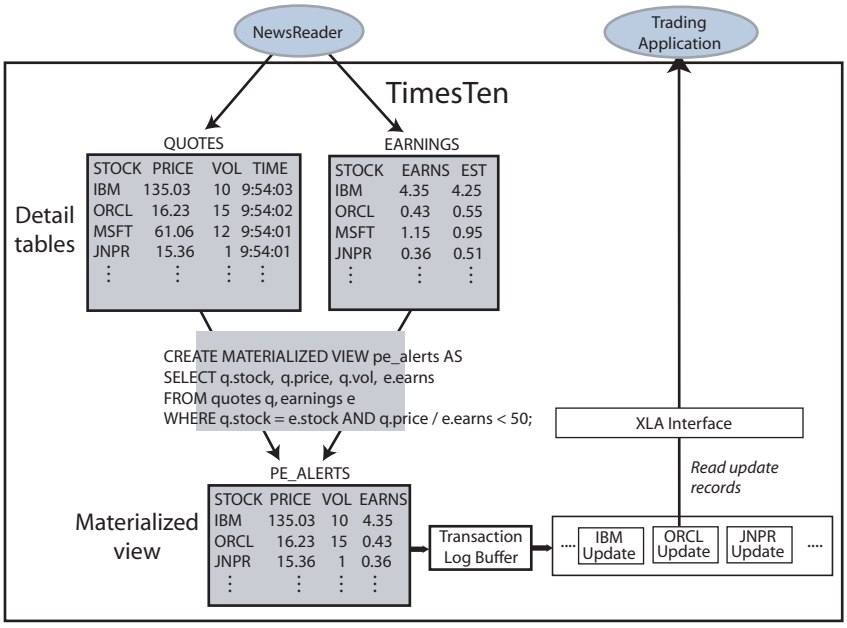


As shown in Figure 2–2, the NewsReader updates stock price data in a QUOTES table in the TimesTen database. Less dynamic earnings data is updated in an EARNINGS table. The STOCK columns in the QUOTES and EARNINGS tables are linked through a foreign key relationship.

The purpose of the trading application is to track only those stocks with PE ratios below 50, then use 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 provide 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 receives alerts only for those stocks that meet its trading criteria.

Figure 2–2 Using materialized views and XLA



IMDB Cache application scenarios

This section describes scenarios that illustrate how IMDB Cache can be used:

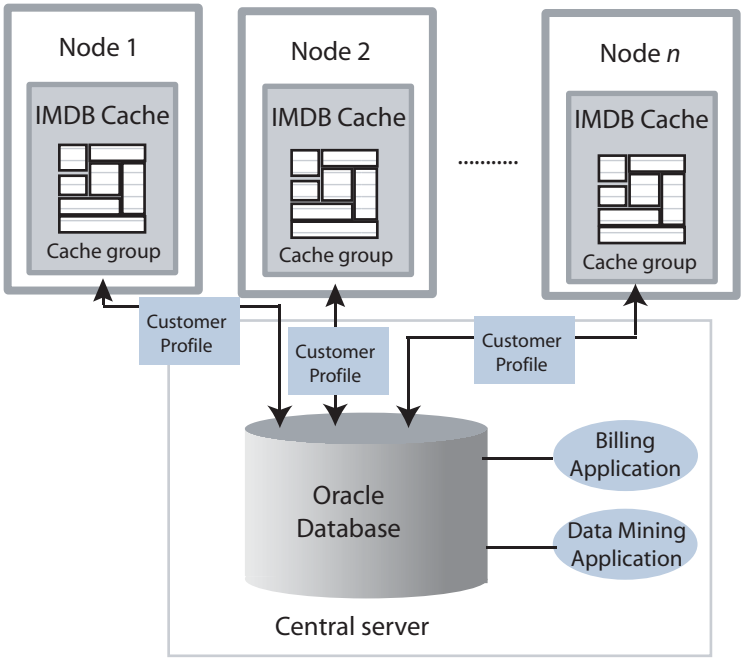
- Call center application: Uses IMDB Cache as an application-tier cache to hold customer profiles maintained in an Oracle database.
- Caller usage metering application: Uses IMDB Cache to store metering data on the activities of cellular callers. The metering data is collected from multiple TimesTen databases distributed throughout a service area and archived in a central Oracle database for use by a central billing application.

Call center application

Advance Call Center provides customer service for *Wireless Communications*.

Figure 2–3 shows that the call center system includes a central server that hosts back-end applications and an Oracle database that stores the customer profiles.

Figure 2–3 Dynamically loading Oracle data to IMDB Cache nodes



To manage a large volume of concurrent customer sessions, the call center has deployed several application server nodes and periodically deploys additional nodes as its customer base increases. Each node contains an IMDB Cache database. When a customer contacts the call center, the user is automatically routed to an available application server node and the appropriate customer profile is dynamically loaded from the Oracle database into the cache database.

When a customer completes a call, changes to the customer profile are flushed from IMDB Cache database to the Oracle database. Least recently used (LRU) aging is configured to remove inactive customer profiles from the IMDB Cache database.

If the same customer contacts the call center again shortly after the first call and is connected to a different application server node, the customer profile is dynamically loaded to the new node from either the Oracle database or from the first IMDB Cache node, depending on where the most recent copy resides. The IMDB Cache determines where the most recent copy resides and uses peer-to-peer communication to exchange

information with other IMDB Cache databases in its grid. It also manages concurrent updates to the same data within its grid.

All of the customer data is stored in the Oracle database. The Oracle database is much larger than the combined IMDB Cache databases and is best accessed by applications that do not require the real-time performance of IMDB Cache but do require access to large amounts of data. Such applications may include a billing application and a data mining application.

As the customer base increases and demands to serve more customers concurrently increases, the call center may decide to deploy additional application server nodes. New IMDB Cache members can join the IMDB Cache grid with no disruption to ongoing requests in the grid. Similarly, failures or removal of individual nodes do not disrupt operations in the rest of the grid.

Caller usage metering application

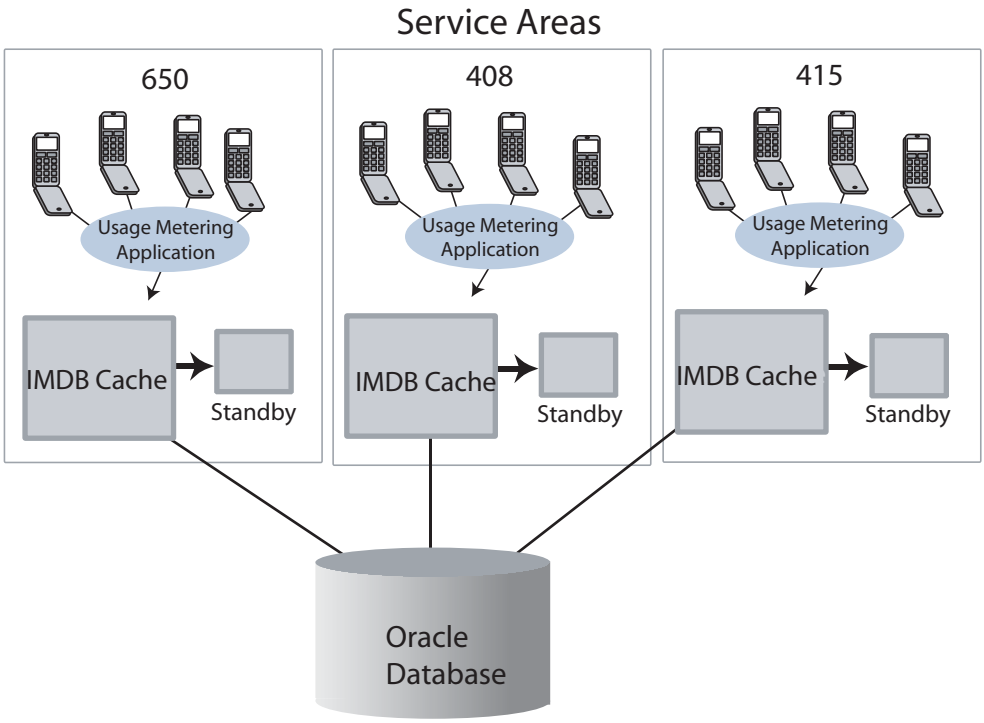
Wireless Communications has a *usage metering* application that keeps track of the duration of each cellular call and the services used. For example, if a caller makes a regular call, a base rate is applied for the duration of the call. If a caller uses special features like roaming, extra charges are 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.

The company uses an IMDB Cache database 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 Oracle database. The company distributes multiple installations of the usage metering application and IMDB Cache on individual nodes throughout its service areas. For maximum performance, each usage metering application connects to its local IMDB Cache database by an ODBC direct driver connection.

Figure 2–4 shows the configuration.

Figure 2–4 Distributed caching of usage data



A usage metering application and IMDB Cache are deployed on each node 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.

Transactions that impact revenue (inserts and updates) must be durable. To ensure data availability, each IMDB Cache database is replicated to a standby database.

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

An IMDB Cache process periodically archives the rows in the *Calls* table to the Oracle database. After the call records have been successfully archived in the Oracle database, they are deleted from the IMDB Cache database by a time-based aging process.

3

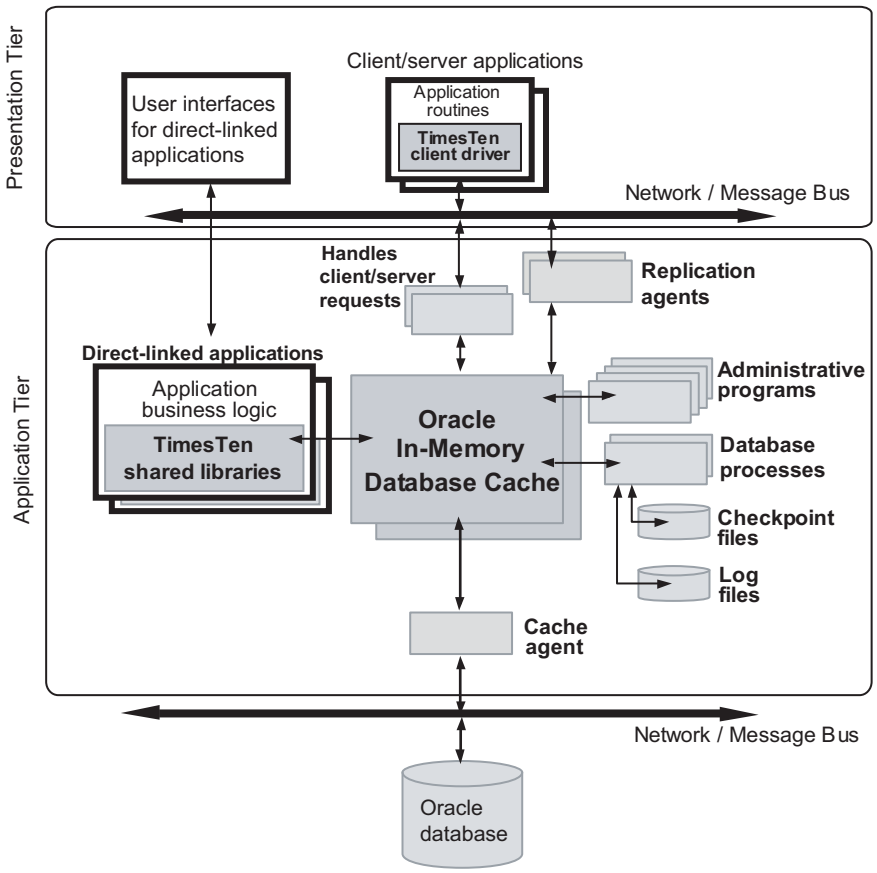
Oracle In-Memory Database Cache Architecture and Components

This chapter includes the following topics:

Architectural overview

This section describes the architecture of the Oracle In-Memory Database Cache. The architecture of the Oracle TimesTen In-Memory Database is the same as the architecture of the IMDB Cache except that the Oracle database and cache agent are not included. Figure 3–1 shows the architecture of the IMDB Cache.

Figure 3–1 Oracle In-Memory Database Cache architecture



The architectural components include shared libraries, memory-resident data structures, database processes, and administrative programs. Memory-resident data structures include tables, indexes, system tables, locks, cursors, compiled commands and temporary indexes. The application can connect to the IMDB Cache or TimesTen database by direct link and by client/server connections.

Replication agents receive information from master databases and send information to subscriber databases. The cache agent performs all asynchronous data transfers between cache groups in the IMDB Cache and the Oracle database.

These components are described in subsequent sections.

Shared libraries

The routines that implement the TimesTen functionality are embodied in a set of shared libraries that developers link with their applications and execute as a part of the application's process. This shared library approach is in contrast to a more conventional RDBMS, which is implemented as a collection of executable programs to which applications connect, typically over a client/server network. Applications can also use a client/server connection to access an IMDB Cache or TimesTen database, though in most cases the best performance will be realized with a directly linked application. See "TimesTen connection options" on page 3-4.

Memory-resident data structures

The IMDB Cache or TimesTen database resides entirely in main memory at run time. It is maintained in shared memory segments in the operating system and contains all user data, indexes, system catalogs, log buffers, lock tables and temp space. Multiple applications can share one database, and a single application can access multiple databases on the same system.

Database processes

TimesTen assigns a separate process to each database to perform operations including the following tasks:

- Loading the database into memory from a checkpoint file on disk
- Recovering the database if it needs to be recovered after loading it into memory
- Performing periodic checkpoints in the background against the active database
- Detecting and handling deadlocks
- Performing data aging
- Writing log records to files

Administrative programs

Utility programs are explicitly invoked by users, scripts, or applications to perform services such as interactive SQL, bulk copy, backup and restore, database migration and system monitoring.

Checkpoint and transaction log files

Checkpoint files contain an image of the database on disk. TimesTen uses dual checkpoint files for additional safety, in case the system fails while a checkpoint operation is in progress. Changes to databases are captured in transaction logs that are written to disk periodically. If a database needs to be recovered, TimesTen merges the database checkpoint on disk with the completed transactions that are still in the transaction log files. Normal disk file systems are used for checkpoints and transaction log files.

See "Data Availability and Integrity" on page 6-1 for more information.

Cached data

When the IMDB Cache is used to cache portions of an Oracle database in a TimesTen in-memory database, a *cache group* is created to hold the cached data.

A cache group is a collection of one or more tables arranged in a logical hierarchy by using primary key and foreign key relationships. Each table in a cache group is related to an Oracle database table. A cache table can contain all rows and columns or a subset of the rows and columns in the related Oracle table. You can create or modify cache groups by using SQL statements or by using Oracle SQL Developer. Cache groups support these features:

- Applications can read from and write to cache groups.
- Cache groups can be refreshed from Oracle data automatically or manually.

- Updates to cache groups can be propagated to Oracle tables automatically or manually.
- Changes to either Oracle tables or the cache group can be tracked automatically.

When rows in a cache group are updated by applications, the corresponding rows in Oracle tables can be updated synchronously as part of the same transaction or asynchronously immediately afterward depending on the type of cache group. The asynchronous configuration produces significantly higher throughput and much faster application response times.

Changes that originate in the Oracle tables are refreshed into the cache by the cache agent.

See "IMDB Cache" on page 8-1 for more information.

Replication

TimesTen replication enables you to achieve near-continuous availability or workload distribution by sending updates between two or more hosts. A master host is configured to send updates and a subscriber host is configured to receive them. A host can be both a master and a subscriber in a bidirectional replication scheme. Time-based conflict detection and resolution are used to establish precedence in case the same data is updated in multiple locations at the same time.

TimesTen recommends the active standby pair configuration for highest availability. It is the only replication configuration that you can use for replicating IMDB Cache. An active standby pair includes an active database, a standby database, optional read-only subscriber databases, and the tables and cache groups that comprise the databases.

When replication is configured, a *replication agent* is started for each database. If multiple databases on the same host are configured for replication, each database has a separate replication agent. Each replication agent can send updates to one or more subscribers and to receive updates from one or more masters. Each of these connections is implemented as a separate thread of execution inside the replication agent process. Replication agents communicate through TCP/IP stream sockets.

For maximum performance, the replication agent detects updates to a database by monitoring the existing transaction log. It sends updates to the subscribers in batches, if possible. Only committed transactions are replicated. On the subscriber host, the replication agent updates the database through an efficient low-level interface, avoiding the overhead of the SQL layer.

See "Replication" on page 6-3 for more information about replication configurations.

TimesTen connection options

Applications can connect to a TimesTen database in one of the following ways:

- Direct driver connection
- Client/server connection
- Driver manager connection

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*.

The ODBC direct driver is a library of ODBC and TimesTen routines that implement the database engine used to manage the databases. Java applications access the ODBC direct driver through the JDBC library. OCI applications access the ODBC direct driver through the OCI library.

An application can create a direct driver connection when it runs on the same machine as the IMDB Cache or TimesTen database. In a direct driver connection, the ODBC driver directly loads the IMDB Cache or TimesTen database into a shared memory segment. The application then uses the direct driver to access the memory image of the database. 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 the IMDB Cache or TimesTen database.

Client/server connection

The TimesTen *client driver* and *server daemon* processes accommodate connections from remote client machines to databases across a network. The server daemon spawns a separate *server child process* for each client connection to the database.

Applications on a client machine issue ODBC, JDBC or OCI calls. These calls access 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 IMDB Cache or TimesTen database.

If the client and server reside on separate hosts in a network, they communicate by using sockets and TCP/IP. If both the client and server reside on the same machine, they can communicate more efficiently by using a shared memory segment 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.

Driver manager connection

Applications can connect to the IMDB Cache or TimesTen database through an ODBC *driver manager*, which is a database-independent interface that adds a layer of abstraction between the applications and the TimesTen database. In this way, the driver manager allows applications to be written to operate independently of the database 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 database.

On Microsoft Windows systems, applications can connect to the MS ODBC driver manager to use an IMDB Cache or TimesTen database along with data sources from other vendors. Driver managers for UNIX systems are available as open-source software as well as from third-party vendors.

For more information

For more information about the TimesTen database, see "Working with TimesTen Databases" in *Oracle TimesTen In-Memory Database C Developer's Guide* and "Working with Data in a TimesTen Database" in *Oracle TimesTen In-Memory Database Operations Guide*.

For more information about connecting to databases, see "Managing TimesTen Databases" and "Working with the TimesTen Client and Server" in *Oracle TimesTen In-Memory Database Operations Guide*.

4

Concurrent Operations

A database can be accessed in shared mode. When a shared database is accessed by multiple transactions, there must be a way to coordinate concurrent changes to data with reads of the same data in the database. TimesTen and IMDB Cache use transaction isolation and locks to coordinate concurrent access to data.

This chapter includes the following topics:

- Transaction isolation
- Locks

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 database. Applications can use the `Isolation` general connection attribute to set the isolation level for a connection. Concurrent connections can use different isolation levels.

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.

TimesTen has two isolation levels:

- Read committed isolation
- Serializable isolation

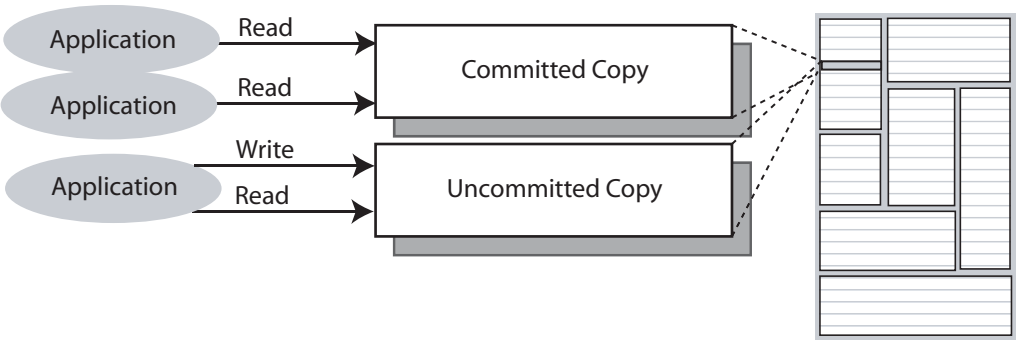
Read committed isolation

When an application uses read committed isolation, readers use a separate copy of the data from writers, so read locks are not needed. Read committed isolation is nonblocking for queries and can work with Serializable isolation or read committed isolation. Under read committed isolation, writers block only other writers and readers using serializable isolation; writers do not block readers using read committed isolation. Read committed isolation is the default isolation level.

TimesTen and IMDB Cache use *versioning* to implement read committed isolation. TimesTen and IMDB Cache update operations create new copies of the rows they update to allow nonserializable reads of those rows to proceed without waiting.

Figure 4–1 shows that some applications read a committed copy of the data while another application writes and reads on an uncommitted copy.

Figure 4–1 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`, `CREATE INDEX`, or `ALTER TABLE` operation on that table. In addition, blocking checkpoints will block readers and writers.

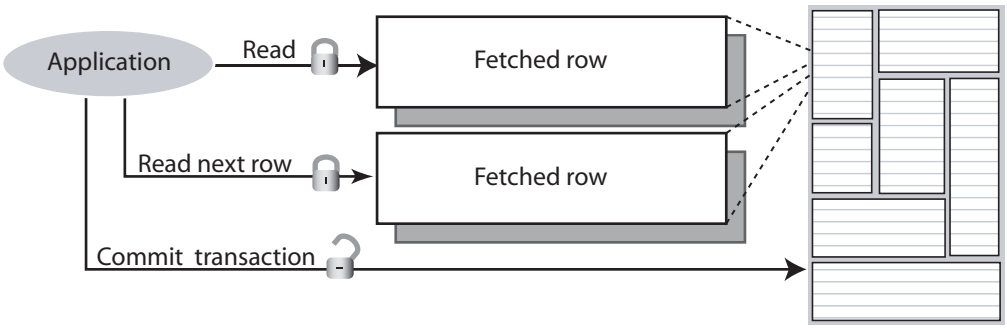
Read committed isolation does acquire read locks as needed during materialized view maintenance to ensure that views are consistent with their detail tables. These locks are not held until the end of the transaction but are instead released when maintenance has been completed.

Serializable isolation

When an application uses serializable isolation, locks are acquired within a transaction and are held until the transaction commits or rolls back for both reads and writes. 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 database-level locking is chosen.

Figure 4–2 shows that locks are held until the transaction is committed.

Figure 4–2 Serializable isolation



Serializable isolation level is useful for transactions that require the strongest level of isolation. Concurrent applications that must modify the data that is read by a

transaction may encounter lock timeouts because read locks are held until the transaction commits.

Locks

Locks are used to serialize access to resources to prevent one user from changing an element that is being read or changed by another user. TimesTen and IMDB Cache automatically perform locking if a database is accessed in shared mode.

Serializable transactions acquire share locks on the items they 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.

TimesTen and IMDB Cache perform 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 it.

Applications can select from three lock levels:

- Database-level locking
- Table-level locking
- Row-level locking

Database-level locking

Locking at the database level locks an entire database when it is accessed by a transaction. All database-level locks are exclusive. A transaction that requires a database-level lock cannot start until there are no active transactions on the database. After a transaction has obtained a database-level lock, all other transactions are blocked until the transaction commits or rolls back.

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

Different transactions can coexist with different levels of locking, but the presence of even one transaction that uses database-level locking leads to reduced concurrency.

Use the `LockLevel` general connection attribute or the `ttLockLevel` built-in procedure to implement database-level locking.

Table-level locking

Table-level locking locks a table when it is accessed by a transaction. It is useful when a statement accesses most of the rows in a table. Applications can call the `ttOptSetFlag` built-in procedure to request that the optimizer use table locks. The optimizer determines when a table lock should be used.

Table locks can reduce throughput, so they should be used only when a substantial portion of the table must be locked or when high concurrency is not needed. For example, tables can be locked for operations such as bulk updates. In read-committed isolation, TimesTen and IMDB Cache do not use table-level locking for read operations unless it is explicitly requested by the application.

Row-level locking

Row-level locking locks only 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 preferable when there are many concurrent transactions, each operating on different rows of the same tables.

Applications can use the `LockLevel` general connection attribute, the `ttLockLevel` built-in procedure and the `ttOptSetFlag` built-in procedure to manage row-level locking.

For more information

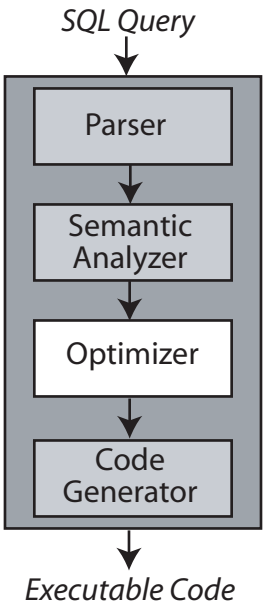
For more information about locks and transaction isolation, see "Transaction Management and Recovery" in *Oracle TimesTen In-Memory Database Operations Guide*.

5

Query Optimization

TimesTen and IMDB Cache have a cost-based query optimizer that ensures efficient data access by automatically searching for the best way to answer queries. Optimization is performed in the third stage of the compilation process. The stages of compilation are shown in Figure 5–1.

Figure 5–1 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 (such as referential integrity, primary key)
- Index choices (including automatic creation of temporary indexes)
- Scan methods (full table scan, rowid lookup, range index scan, bitmap index lookup, hash index lookup)
- Join algorithm choice (nested loop joins, nested loop joins with indexes, or merge join)

This chapter includes the following topics:

Optimization time and memory usage

The optimizer is designed to generate the best possible plan within reasonable time and memory constraints. No optimizer always chooses the optimal plan for every query. Instead, the goal of the optimizer is to choose the best plan from among a set of plans generated by using strategies for finding the most promising areas within the search-space of plans. Because optimization usually happens only once for each query while the query itself may be executed many times, the optimizer is designed to give precedence to execution time over optimization time.

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 5-2 to disable the use of temporary indexes and tables in order to create plans that trade maximum performance for reduced memory usage.

Statistics

When determining the 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 when an applications calls the `ttOptUpdateStats` built-in 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 an estimate of the number of rows in a table. 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` built-in procedure to provide the optimizer with hints about how to best optimize a particular query. This takes the form of directives that restrict the use of particular join algorithms, use of temporary indexes and types of index, 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. You can view the existing hints set for a database by using the `ttOptGetFlag` built-in procedure.

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

Indexes

The query optimizer uses indexes to speed up the execution of a query. The optimizer uses existing indexes or creates temporary indexes to generate an 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 to obtain maximum performance from a TimesTen system.

TimesTen uses these types of indexes:

- *Range index*: Range indexes are useful for finding rows with column values within a range specified as an equality or inequality. Range indexes can be created over one or more columns of a table. They can be designated as unique or not unique. Multiple `NULL` values are permitted in a unique range index. When sorting data values, TimesTen considers `NULL` values to be larger than all non-`NULL` values. When you create an index using the `CREATE INDEX` SQL statement and do not specify the index type, TimesTen creates a range index.
- *Bitmap index*: Bitmap indexes encode information about a unique value in a row in a bitmap. Each bit in the bitmap corresponds to a row in the table. Use a bitmap index for columns that do not have many unique values. An example of such a column is a column that records gender as one of two values. Bitmap indexes are widely used in data warehousing environments. The environments typically have large amounts of data and ad hoc queries, but a low level of concurrent DML transactions. Bitmap indexes are compressed and have smaller storage requirements than other indexing techniques.
- *Hash index*: Hash indexes are created for tables with a primary key when you specify the `UNIQUE HASH INDEX` clause in the `CREATE TABLE` statement. There can be only one hash index for each table. In general, hash indexes are faster than range indexes for exact match lookups and equijoins. However, hash indexes cannot be used for lookups involving ranges or the prefix of a key and can require more space than range indexes and bitmap indexes.

Scan methods

The optimizer can select from multiple types of scan methods. The most common scan methods are:

- Full table scan
- Rowid lookup
- Range index scan (on either a permanent or temporary index)
- Hash index lookup (on either a permanent or temporary index)
- Bitmap index lookup (on a permanent index)

TimesTen and IMDB Cache perform fast exact matches through hash indexes, bitmap indexes and rowid lookups. They perform range matches through range indexes. The `ttOptSetFlag` built-in procedure can be used to allow or disallow the optimizer from considering certain scan methods when choosing a query plan.

A *full table scan* examines every row in a table. Because it is the least efficient way to evaluate a query predicate, a full scan is only used when no other method is available.

TimesTen assigns a unique ID, called a 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 index lookups.

A *hash index lookup* uses a hash index to find rows based on their primary keys. Such lookups are applicable if the table has a primary key that has a hash index and the predicate specifies an exact match over the primary key columns.

A *bitmap index lookup* uses a bitmap index to find rows that satisfy an equality predicate such as `customer.gender='male'`. Bitmap indexes are appropriate for columns with few unique values. They are particularly useful in evaluating several predicates each of which can use a bitmap index lookup because the combined predicates can be efficiently evaluated through bit operations on the indexes themselves. For example, if table `customer` has a bitmap index on the column `gender` and if table `sweater` has a bitmap index on the column `color`, the predicates `customer.gender='male'` and `sweater.color ='pink'` could rapidly find all male customers who purchased pink sweaters by performing a logical `AND` operation on the two bitmap indexes.

A *range index scan* uses a range index to access a table. Such a scan is applicable to exact match predicates such as `t1.a=2` or to range predicates such as `t1.a>2` and `t1.a<10` as long as the column used in the predicate has a range index defined over it. If a range index is defined over multiple columns, it can be used for multiple column predicates. For example, the predicates `t1.b=100` and `t1.c>'ABC'` result in a range index scan if a range index is defined over columns `t1.b` and `t1.c`. The index can be used if it is defined over more columns. For example, if a range index is defined over `t1.b`, `t1.c` and `t1.d`, the optimizer uses the index prefix over columns `b` and `c` and returns all the values for column `d` that match the stated predicate over columns `b` and `c`.

Join methods

The optimizer can select from multiple 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 takes 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 5–2 shows an example of a nested loop join. The join condition is:

```
WHERE t1.a=t2.a
```

`t1` is the outer table and `t2` is the inner table. Values in column `a` in table `t1` that match values in column `a` in table `t2` are 1 and 7. The join results concatenate the rows from `t1` and `t2`. For example, the first join result is the following row:

7 50 43.54 21 13.69

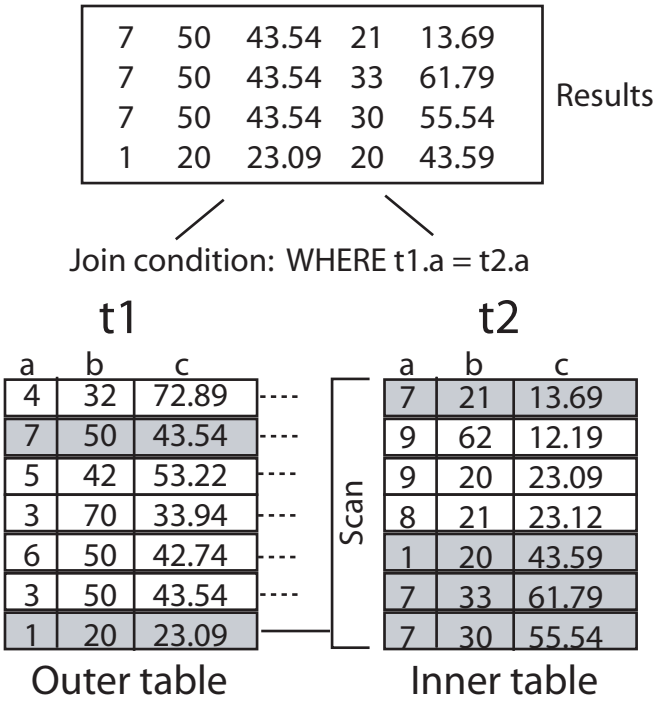
It concatenates a row from `t1`:

7 50 43.54

with the first row from `t2` in which the values in column `a` match:

7 21 13.69

Figure 5–2 Nested loop join



Merge join

A merge join is used only when the join columns are sorted by range 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, then advance inner cursor
- If Inner.JoinColumn = Outer.JoinColumn, then read match
- If Inner.JoinColumn > Outer.JoinColumn, then 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 range indexes have been created for the tables before preparing the query. If no range indexes exist for the tables being joined before preparing the query, the optimizer may in some situations create temporary range indexes in order to use a merge join.

Figure 5–3 shows an example of a merge join. The join condition is:

```
WHERE t1.a=t2.a
```

`x1` is the index on table `t1`, sorting on column `a`. `x2` is the index on table `t2`, sorting on column `a`. The merge join results concatenate the rows in `x1` with rows in `x2` in which the values in column `a` match. For example, the first merge join result is:

1 20 23.09 20 43.59

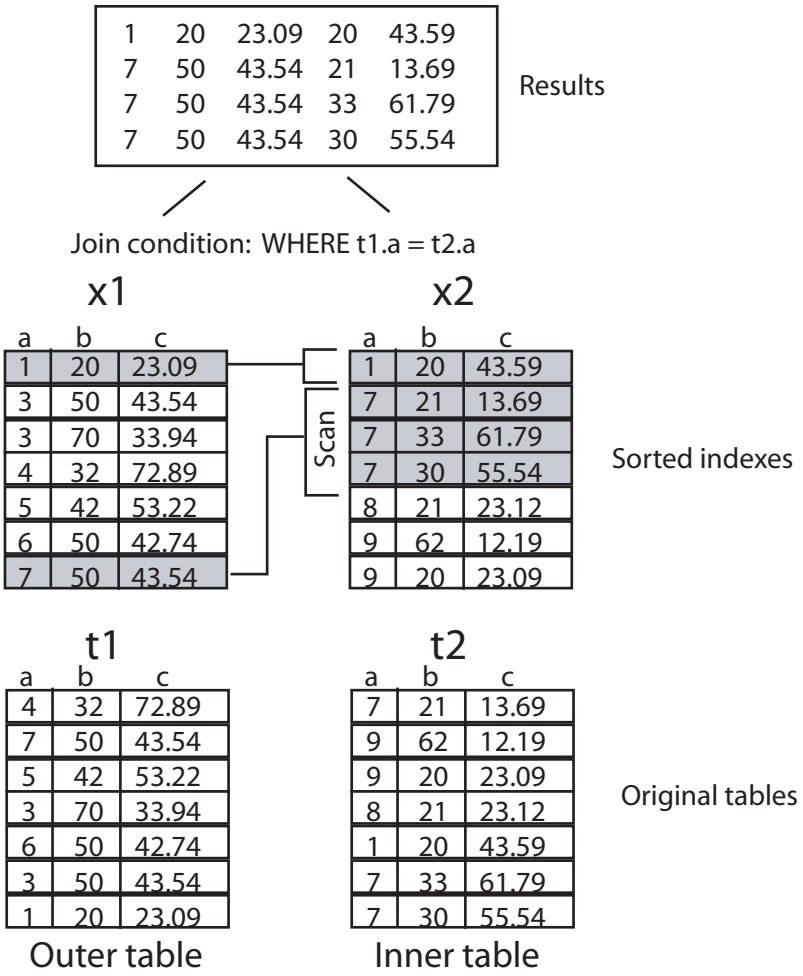
It concatenates a row in `x1`:

1 20 23.09

with the first row in `x2` in which the values in column `a` match:

1 20 43.59

Figure 5–3 Merge join



Optimizer plan

Like most database optimizers, the 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 about 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 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` built-in 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.

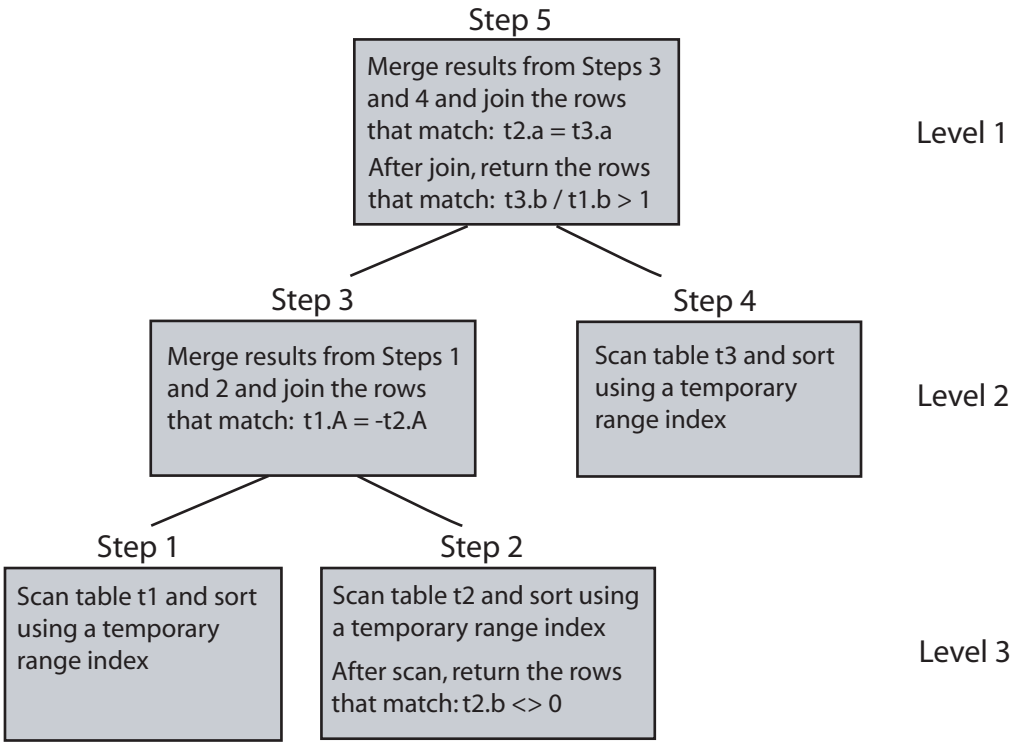
Consider this query:

```
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 breaks down the query into its individual operations and generates a 5-step execution plan to be performed at three levels, as shown in Figure 5–4.

Figure 5–4 Example execution plan



For more information

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

For more information about indexing, see "Understanding indexes" in *Oracle TimesTen In-Memory Database Operations Guide*.

Also see descriptions for the `CREATE TABLE` and `CREATE INDEX` statements in *Oracle TimesTen In-Memory Database SQL Reference*.

6

Data Availability and Integrity

TimesTen and IMDB Cache ensure the availability, durability, and integrity of data through the following mechanisms:

- Transaction logging
- Checkpointing
- Replication

Transaction logging

The TimesTen or IMDB Cache transaction log is used for the following purposes:

- Redo transactions if a system failure occurs
- Undo transactions that are rolled back
- Replicate changes to other TimesTen databases or IMDB Cache databases
- Replicate changes to an Oracle database
- Enable applications to monitor changes to tables through the XLA interface

The transaction log is stored in files on disk. The end of the transaction log resides in an in-memory buffer.

Writing the log buffer to disk

TimesTen and IMDB Cache write the contents of the in-memory log buffer to disk at every durable commit, at every checkpoint, and at other times defined by the implementation. Applications that cannot tolerate the loss of any committed transactions if a failure occurs should request a durable commit for every transaction that is not read-only. They can do so by setting the `DurableCommits` general connection attribute to 1 when they connect to the database.

Applications that can tolerate the loss of some recently committed transactions can significantly improve their performance by committing some or all of their transactions nondurably. To do so, they set the `DurableCommits` general connection attribute to 0 (the default) and typically request explicit durable commits either at regular time intervals or at specific points in their application logic. To request a durable commit, applications call the `ttDurableCommit` built-in procedure.

When are transaction log files deleted?

Transaction log files are kept until TimesTen or the IMDB Cache declares them to be purgeable. A transaction log file cannot be purged until all of the following actions have been completed:

- All transactions writing log records to the transaction log file (or a previous transaction log file) have committed or rolled back.
- All changes recorded in the transaction log file have been written to the checkpoint files on disk.
- All changes recorded in the transaction log file have been replicated (if replication is used).
- All changes recorded in the transaction log file have been propagated to the Oracle database if the IMDB Cache has been configured for that behavior.
- All changes recorded in transaction log files have been reported to the XLA applications (if XLA is used).

TimesTen commits

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. TimesTen recommends that you turn autocommit off so that commits are intentional.

TimesTen issues an implicit commit before and after any data definition language (DDL) statement by default. This behavior is controlled by the `DDLCommitBehavior` general connection attribute. You can use the attribute to specify instead that DDL statements be executed as part of the current transaction and committed or rolled back along with the rest of the transaction.

Checkpointing

Checkpoints are used to keep a snapshot of the database. If a system failure occurs, TimesTen and the IMDB Cache can use a checkpoint file with transaction log files to restore a database to its last transactionally consistent state.

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

TimesTen and IMDB Cache provide two kinds of checkpoints:

- Nonblocking checkpoints
- Blocking checkpoints

TimesTen and IMDB Cache create nonblocking checkpoints automatically.

Nonblocking checkpoints

TimesTen and IMDB Cache initiate nonblocking checkpoints in the background automatically. Nonblocking checkpoints are also known as *fuzzy* checkpoints. The frequency of these checkpoints can be adjusted by the application. Nonblocking checkpoints do not require any locks on the database, so multiple applications can asynchronously commit or roll back transactions on the same database while the checkpoint operation is in progress.

Blocking checkpoints

An application can call the `ttCkptBlocking` built-in procedure to initiate a blocking checkpoint 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. 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 because the checkpoint record contains the information needed to recover.

Recovery from log and checkpoint files

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

During recovery, the latest checkpoint file is read into memory. All transactions that have been committed since the last checkpoint and whose log records are on disk are rolled forward from the appropriate transaction log files. Note that such transactions include all transactions that were committed durably as well as all transactions whose log records aged out of the in-memory log buffer. Uncommitted or rolled-back transactions are not recovered.

For applications that require uninterrupted access to TimesTen data in the event of failures, see "Replication" on page 6-3.

Replication

The fundamental motivation behind replication is to make data highly available to applications with minimal performance impact. In addition to its role in failure recovery, replication is also useful for distributing application workloads across multiple databases for maximum performance and for facilitating online upgrades and maintenance.

Replication is the process of copying data from a *master* database to a *subscriber* database. Replication at each master and subscriber database is controlled by *replication agents* that communicate through TCP/IP stream sockets. The replication agent on the master database reads the records from the transaction log for the master database. It forwards changes to replicated elements to the replication agent on the subscriber database. The replication agent on the subscriber then applies the updates to its database. If the subscriber agent is not running when the updates are forwarded by the master, the master retains the updates in its transaction log until they can be applied at the subscriber.

You can increase replication throughput by configuring parallel replication at database creation time. Parallel replication is enabled by default. You configure the number of threads for applying updates to subscribers. The updates are transmitted in commit order.

TimesTen recommends the active standby pair configuration for highest availability. It is the only replication configuration that you can use for replicating IMDB Cache.

The rest of this section includes the following topics:

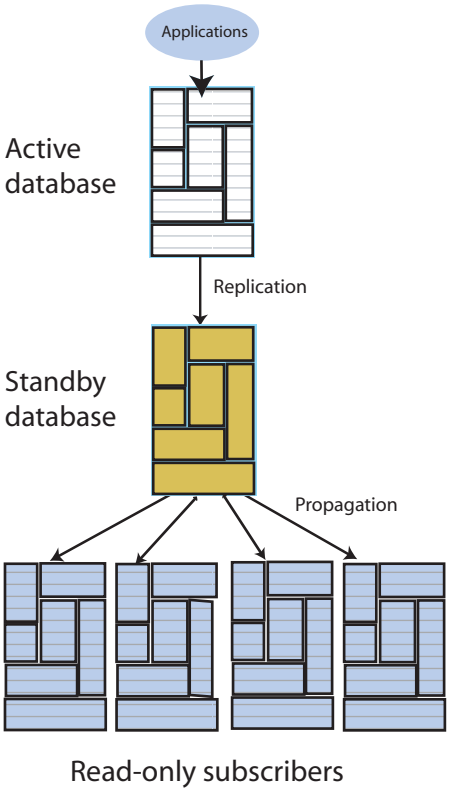
- Active standby pair
- Other replication configurations
- Asynchronous and return service replication

- Replication failover and recovery

Active standby pair

An active standby pair includes an active database, a standby database, optional read-only subscriber databases, and the tables and cache groups that comprise the databases. Figure 6–1 shows an active standby pair.

Figure 6–1 Active standby pair



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

Only one of the master databases can function as an active database at a specific time. If the active database fails, the role of the standby database must be changed to active before recovering the failed database as a standby database. The replication agent must be started on the new standby database.

If the standby database fails, the active database replicates changes directly to the read-only subscribers. After the standby database has recovered, it contacts the active database to receive any updates that have been sent to the read-only subscribers while the standby was down or was recovering. When the active and the standby databases have been synchronized, then the standby resumes propagating changes to the subscribers.

Active standby replication can be used with IMDB Cache to achieve cross-tier high availability. Active standby replication is available for both read-only and asynchronous writethrough cache groups. When used with read-only cache groups, updates are sent from the Oracle database to the active database. Thus the Oracle database plays the role of the application in this configuration. When used with asynchronous writethrough cache groups, the standby database propagates updates that it receives from the active database to the Oracle database. In this scenario, the Oracle database plays the role of one of the read-only subscribers.

An active standby pair that replicates one of these types of cache groups can perform failover and recovery automatically with minimal chance of data loss. See "Active standby pairs with cache groups" in *Oracle TimesTen In-Memory Database Replication Guide*.

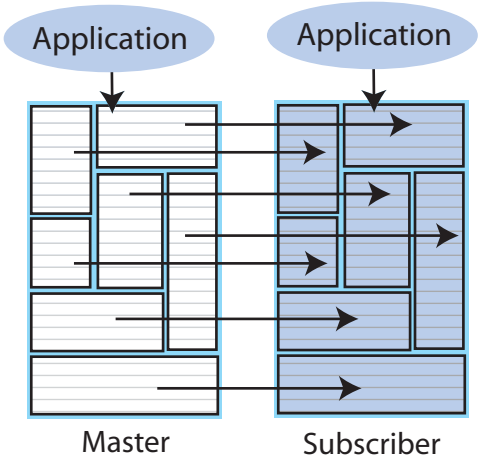
Other replication configurations

TimesTen replication architecture is flexible enough to achieve balance between performance and availability. In general, replication can be configured to be unidirectional from a master to one or more subscribers, or bidirectional between two or more databases that serve as both master and subscriber.

Unidirectional replication

Figure 6–2 shows a *unidirectional* replication scheme. The application is configured on both hosts so that the subscriber is ready to take over if the master host fails. While the master is up, updates from the application to the master database are replicated to the subscriber database. The application on the subscriber host does not execute any updates against the subscriber database, but may read from that database. If the master fails, the application on the subscriber host takes over the update function and starts updating the subscriber database.

Figure 6–2 Unidirectional replication scheme



Replication can also be used to copy updates from a master database to many subscriber databases. Figure 6–3 shows a replication scheme with multiple subscribers.

Figure 6–3 Unidirectional replication to multiple subscribers

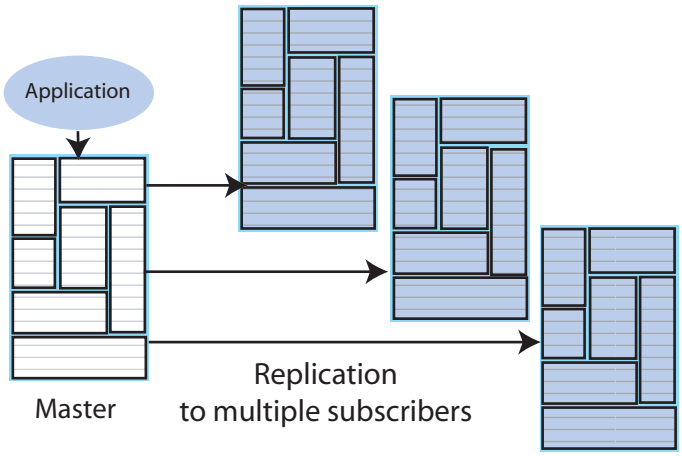
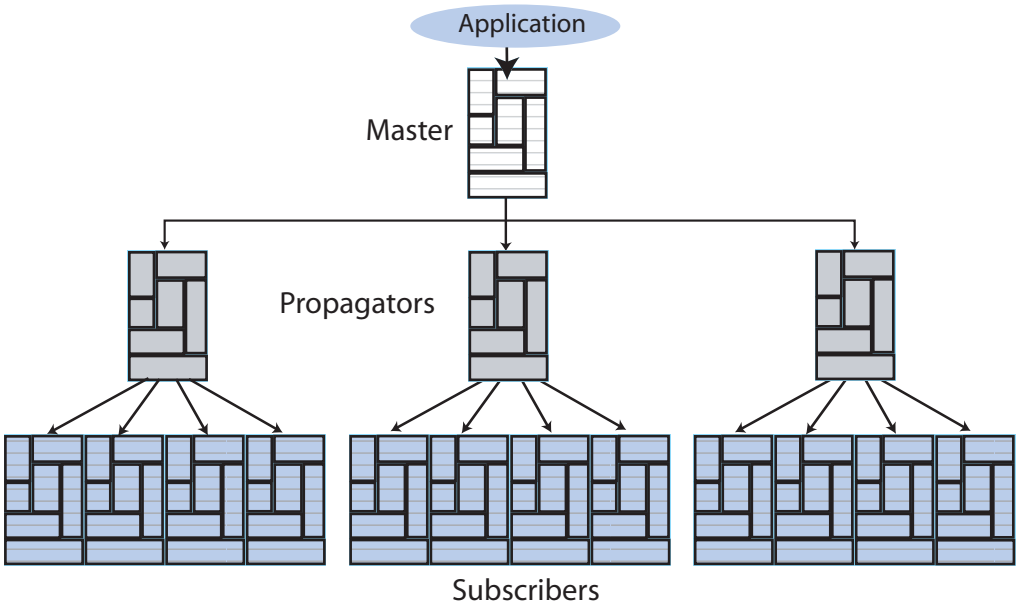


Figure 6–4 shows a *propagation* configuration. One master propagates updates to three subscribers. The subscribers are also masters that propagate updates to additional subscribers.

Figure 6–4 Propagation configuration



Bidirectional replication

Bidirectional replication schemes are used for load balancing. The workload can be split between two bidirectionally replicated databases. There are two basic types of load-balancing configurations:

- *Split workload* where each database bidirectionally replicates a portion of its data to the other database. Figure 6–5 shows a split workload configuration.
- *Distributed workload* where user access is distributed across duplicate application/database combinations that replicate updates to each other. 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. If collisions do occur, TimesTen has a timestamp-based collision detection and resolution capability. Figure 6–6 shows a distributed workload configuration.

Figure 6–5 Split workload replication

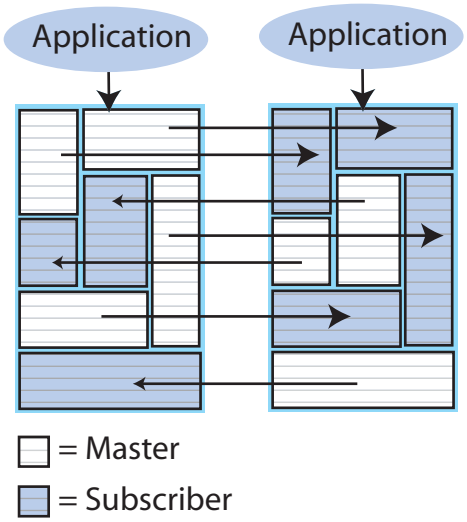
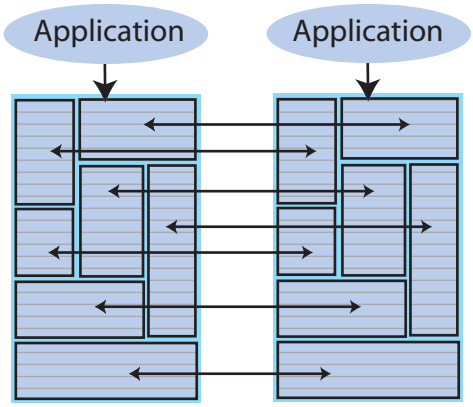


Figure 6–6 Distributed workload replication



Asynchronous and return service replication

TimesTen replication is by default an asynchronous mechanism. When using asynchronous replication, an application updates the master database and continues working without waiting for the updates to be received by the subscribers. The master and subscriber databases 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 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 applications that need higher levels of confidence that the replicated data is consistent between the master and subscriber databases:

- The *return receipt service* 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.

> **Note:** Do not use return twosafe service in a distributed workload configuration. This can produce deadlocks.

Applications that use the return services trade some performance to ensure higher levels of consistency and reduce the risk of transaction loss between the master and subscriber databases. 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 database. Return receipt replication has less performance impact than return twosafe at the expense of potential loss of transactions.

Replication failover and recovery

For replication to make data highly available to applications with minimal performance impact, there must be a way to shift applications from the failed database to its surviving backup as seamlessly as possible.

You can use Oracle Clusterware to manage failures automatically in systems with active standby pairs. Other kinds of replication schemes can be managed with custom and third-party cluster managers. They detect failures, redirect users or applications from the failed database to either a standby database or a subscriber, and manage recovery of the failed database. The cluster manager or administrator can use TimesTen-provided utilities and functions to duplicate the surviving database and recover the failed database.

Subscriber failures generally have no impact on the applications connected to the master databases and can be recovered without disrupting user service. If a failure occurs on a master database, the cluster manager must redirect the application load to a standby database or a subscriber in order to continue service with no or minimal interruption.

You can configure automatic client failover for databases that have active standby pairs with client/server connections. This enables the client to fail over automatically to the server on which the standby database resides.

For more information

For more information about logging and checkpointing, see "Transaction Management and Recovery" in *Oracle TimesTen In-Memory Database Operations Guide*.

For more information about replication, see *Oracle TimesTen In-Memory Database Replication Guide*.

For more information about automatic client failover, see *Oracle TimesTen In-Memory Database Operations Guide*.

7

Event Notification

TimesTen and IMDB Cache event notification is done through the Transaction Log API (XLA), which provides functions to detect changes to the database. XLA monitors log records. A log record describes an insert, update or delete on a row in a table. XLA can be used with materialized views to focus the scope of notification on changes made to specific rows across multiple tables.

TimesTen and IMDB Cache also use SNMP traps to send asynchronous alerts of events.

This chapter includes the following topics:

- Transaction Log API
- Materialized views and XLA
- SNMP traps

Transaction Log API

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

Applications use XLA to implement a change notification scheme. In this scheme, XLA applications can monitor a database for changes and then take actions based on those changes. For example, a TimesTen database 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 database for updates on certain stock prices and use that information to determine whether to execute orders. See "Real-time quote service application" on page 2-2 for a complete example.

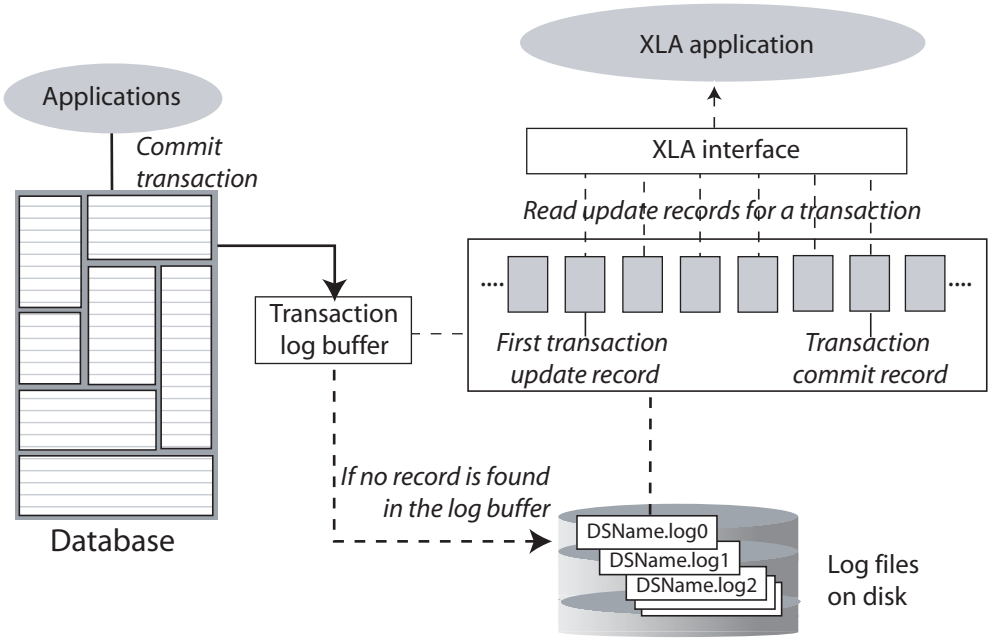
XLA can also be used to build a custom data replication solution in place of the TimesTen replication. Such XLA-enabled replication solutions might include replication with a non-TimesTen database or pushing updates to another TimesTen database.

How XLA works

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 transaction log files on disk, as shown in Figure 7–1. Records are available as long as the transaction log files are available. Readers use bookmarks to maintain their

position in the log update stream. Bookmarks are stored in the database, so they are persistent across database connections, shutdowns, and failures.

Figure 7–1 How XLA works



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 obtain groups of update records written to the log.

Each returned record contains a fixed-length update header 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
- Which columns in the row were updated

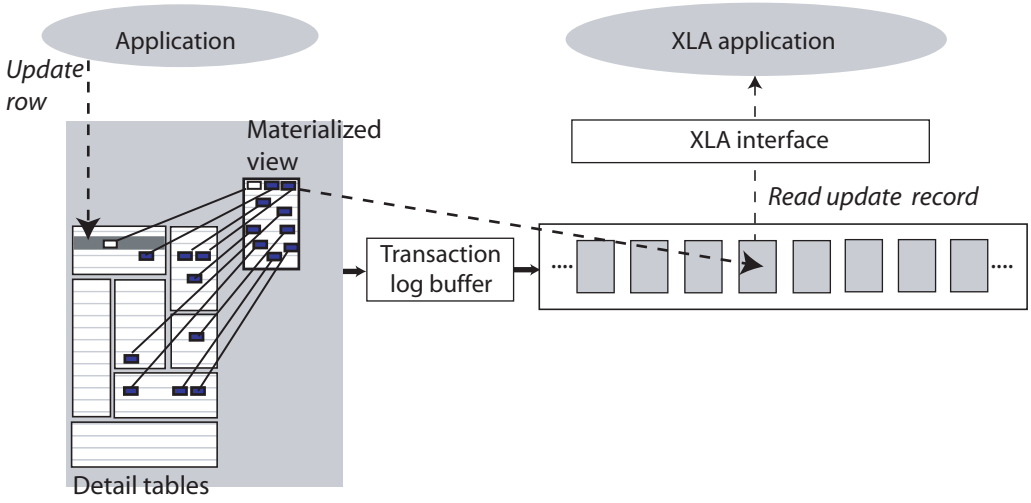
Materialized views and XLA

In most database systems, materialized views are used to simplify and enhance the performance of `SELECT` queries that involve multiple tables. Though materialized views offer this same capability in TimesTen and IMDB Cache, another purpose of materialized views in TimesTen and IMDB Cache is their role in working with XLA to keep track of specific rows and columns in multiple tables.

When a materialized view is present, an XLA application needs to monitor only update records that are of interest from a single materialized view. 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.

Figure 7–2 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 7–2 Using XLA to detect updates on a materialized view table



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

The TimesTen and IMDB Cache implementation of materialized views emphasizes performance as well as the ability to detect updates across multiple tables. Readers familiar with other implementations of materialized views should note that the following tradeoffs have been made:

- The application must explicitly create materialized views. The TimesTen query optimizer has no facility to create materialized views automatically.
- The query optimizer does not rewrite queries on the detail tables to reference materialized views. Application queries must directly reference views.
- There are some restrictions to the SQL used to create materialized views.

When creating a materialized view, the application must specify whether the maintenance of the view should be immediate or deferred. With immediate maintenance, a view is refreshed as soon as changes are made to its detail tables. With deferred maintenance, a view is refreshed only after the transaction that updated the detail tables is committed. A view with deferred maintenance is called an *asynchronous materialized view*. The refreshes may be automatic or may be initiated by the application, and they may be incremental or full. The application must specify the frequency of automatic refreshes. Note that the order of XLA notifications for an asynchronous materialized view is not necessarily the same as the order of transactions for the associated detail tables.

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

- IMDB Cache autorefresh failure
- Database out of space
- Replicated transaction failure
- Death of daemons
- Database 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 about XLA, see "XLA and TimesTen Event Management" in the *Oracle TimesTen In-Memory Database C Developer's Guide* and "Using JMS/XLA for Event Management" in *Oracle TimesTen In-Memory Database Java Developer's Guide*.

For more information about TTClasses, see *Oracle TimesTen In-Memory Database TTClasses Guide*.

For more information about materialized views, see "Understanding materialized views" in *Oracle TimesTen In-Memory Database Operations Guide*. Also see the `CREATE MATERIALIZED VIEW` statement in *Oracle TimesTen In-Memory Database SQL Reference*.

For more information about SNMP traps, see "Diagnostics through SNMP Traps" in *Oracle TimesTen In-Memory Database Error Messages and SNMP Traps*.

8

IMDB Cache

IMDB Cache provides the ability to transfer data between an Oracle database and an IMDB Cache database.

You can cache Oracle data in an IMDB Cache database by defining a cache grid and then creating cache groups in TimesTen where each cache group maps to a single table in the Oracle database or to a group of tables related by foreign key constraints.

This chapter includes the following topics:

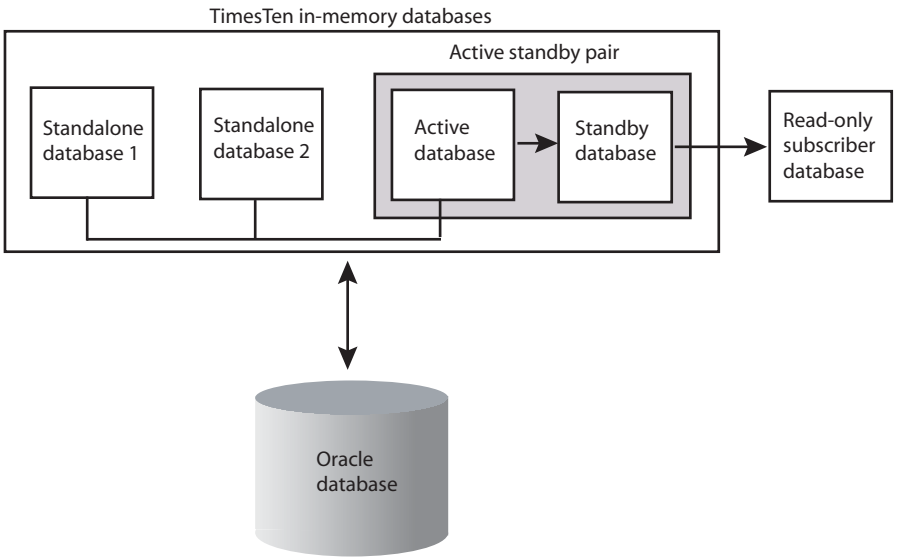
Cache grid

A cache grid is a collection of IMDB Cache databases that collectively manage the application data. A cache grid consists of one or more grid members. A grid member can be either a standalone TimesTen database or an active standby pair. Grid members cache tables from a central Oracle database or Real Application Cluster (Oracle RAC). Cached data is dynamically distributed across multiple grid members without shared storage. This architecture allows the capacity of the cache grid to scale based on the processing needs of the application. When the workload increases or decreases, new grid members attach to the grid or existing grid members detach from the grid without interrupting operations on other grid members.

An IMDB Cache database within a cache grid can contain explicitly loaded and dynamic cache groups as well as global and local cache groups of any cache group type. A cache grid ensures that data is consistent across nodes.

Figure 8–1 shows a cache grid. The grid has three members: two standalone IMDB Cache databases and an active standby pair with a read-only subscriber. The read-only subscriber is not part of the grid.

Figure 8–1 Cache grid



Cache groups

You can cache Oracle data by creating a cache group in an IMDB Cache database. A cache group can be created to cache a single Oracle table or a set of related Oracle tables. The cached Oracle data can consist of all the rows and columns or a subset of the rows and columns in the Oracle tables.

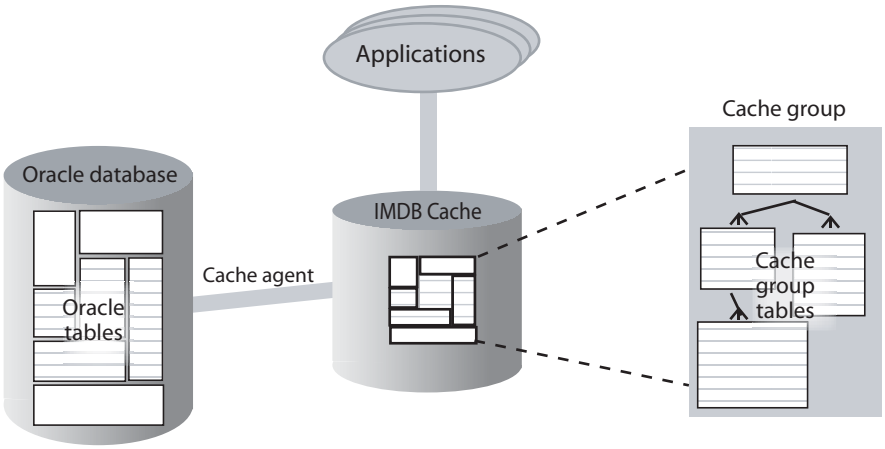
IMDB Cache supports the following features:

- 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 updates can be sent synchronously or asynchronously.

The IMDB Cache database interacts with the Oracle database to perform all of the synchronous cache group operations, such as creating a cache group and propagating updates between the cache group and the Oracle database. A process called the cache agent performs asynchronous cache operations, such as loading data into the cache group, manually refreshing the data from the Oracle database to the cache group, and automatically refreshing the data from the Oracle database to the cache group.

Figure 8–2 illustrates the IMDB Cache features and processes.

Figure 8–2 IMDB Cache



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 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. No table in the cache group can be a child to more than one parent in the cache group. Each IMDB Cache record belongs to only one cache instance and has only one parent in its cache group.

The most commonly used cache group types are:

- *Read-only cache group* - A read-only cache group enforces a caching behavior in which committed updates to Oracle tables are automatically refreshed to the corresponding cache tables in the IMDB Cache database.
- *Asynchronous writethrough (AWT) cache group* - An AWT cache group enforces a caching behavior in which committed updates to cache tables in the IMDB Cache database are automatically propagated to the corresponding Oracle tables asynchronously.

Other types of cache groups are:

- *Synchronous writethrough (SWT) cache group* - An SWT cache group enforces a caching behavior in which committed updates to cache tables in the IMDB Cache database are automatically propagated to the corresponding Oracle tables synchronously.
- *User managed cache group* - A user managed cache group defines customized caching behavior. For example, individual cache tables in a user managed cache are not constrained to be all of the same type. Some tables may be defined as read-only while others may be defined as updatable.

Dynamic cache groups and explicitly loaded cache groups

Cache groups can be either dynamically loaded or explicitly loaded.

In explicitly loaded cache groups, the application preloads data into the cache tables from the Oracle database using a load cache group operation. From that point on, all data needed by the application is available in the IMDB Cache database.

In dynamic cache groups, cache instances are automatically loaded into the IMDB Cache from the Oracle database when the application references cache instances that are not already in the IMDB Cache. The use of dynamic cache groups is typically coupled with least recently used (LRU) aging so that less recently used cache instances are aged out of the cache to free up space for recently used cache instances. Using dynamic cache groups is appropriate when the size of the data that qualifies for caching exceeds the size of the memory available for the IMDB Cache database.

All cache group types (read-only, AWT, SWT, user managed) can be defined as a explicitly loaded or dynamic.

Global and local cache groups

Cache groups can be defined as either local or global.

In local cache groups, data in the cached tables is not shared among IMDB Cache databases even if the databases are members of the same cache grid. Consequently, the content of the databases may overlap with no coordination from the IMDB Cache. Local cache groups are appropriate for applications that have logically partitioned their data between different nodes or for read-only cache groups. Any cache group type can be defined as a local cache group. Local cache groups can be explicitly loaded or dynamic.

In global cache groups, data in the cached tables is shared among IMDB Cache databases within the same cache grid. Updates to the same data by different grid members are coordinated by the grid to ensure read/write data consistency across the IMDB Caches.

A dynamic AWT cache group and an explicitly loaded AWT cache group can be defined as a global cache group. New cache instances are loaded into the cache tables of a global cache group on demand. Queries on a dynamic AWT global cache group can be satisfied by data from the local grid member on which the query is made, from remote grid members or from the Oracle database. Queries on an explicitly loaded AWT cache group can be satisfied by data from the local grid member or from remote grid members

Transmitting data between the IMDB Cache and Oracle Database

The IMDB Cache maintains consistency between cached data and the Oracle database by automatically propagating updates from cache groups to the Oracle database and automatically refreshing data in cache groups from the Oracle database.

The rest of this section includes the following topics:

- Updating a cache group from Oracle tables
- Updating Oracle tables from a cache group

Updating a cache group from Oracle tables

The following mechanisms are available to keep a cache group synchronized with the corresponding data in the Oracle tables:

- *Autorefresh* - An *incremental autorefresh* operation updates only records that have been modified in the Oracle database since the last refresh. The IMDB Cache automatically performs the incremental refresh at specified time intervals. You can also specify a *full autorefresh* operation, which automatically refreshes the entire cache group at specified time intervals.

- *Manual refresh* - An application issues a `REFRESH CACHE GROUP` statement to refresh either an entire cache group or a specific cache instance. It is equivalent to unloading and then loading the cache group or cache instance.

These mechanisms are useful under different circumstances. 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 is the best choice if the Oracle table is updated often, but only a few rows are changed with each update. A manual refresh is the best choice if the application logic knows when the refresh should happen.

Updating Oracle tables from a cache group

The *propagate* and *flush* mechanisms are available to keep the Oracle database up to date with the cache group:

- *Propagate* - The most common way to propagate cache group data to the Oracle database is by using an asynchronous writethrough (AWT) cache group. Other methods of updating the Oracle tables are using a synchronous writethrough (SWT) cache group or specifying the `PROPAGATE` option in a user managed cache group.

  Changes to an AWT cache group are committed without waiting for the changes to be applied to the Oracle tables. AWT cache groups provide better response times and performance than SWT cache groups and user managed cache groups with the `PROPAGATE` option, but the IMDB Cache database and the Oracle database do not always contain the same data because changes are applied to the Oracle tables asynchronously.

  You can increase throughput from AWT cache groups to Oracle tables by configuring parallel propagation at database creation time. You configure the number of threads for applying updates to the Oracle tables.

- *Flush* - A flush operation can be used to propagate updates manually from a user managed cache group to the Oracle database.An application initiates a flush operation by issuing a `FLUSH CACHE GROUP` statement. Flush operations are useful when frequent updates occur for a limited period of time over a set of records. Flush operations do not propagate deletes.

Aging feature

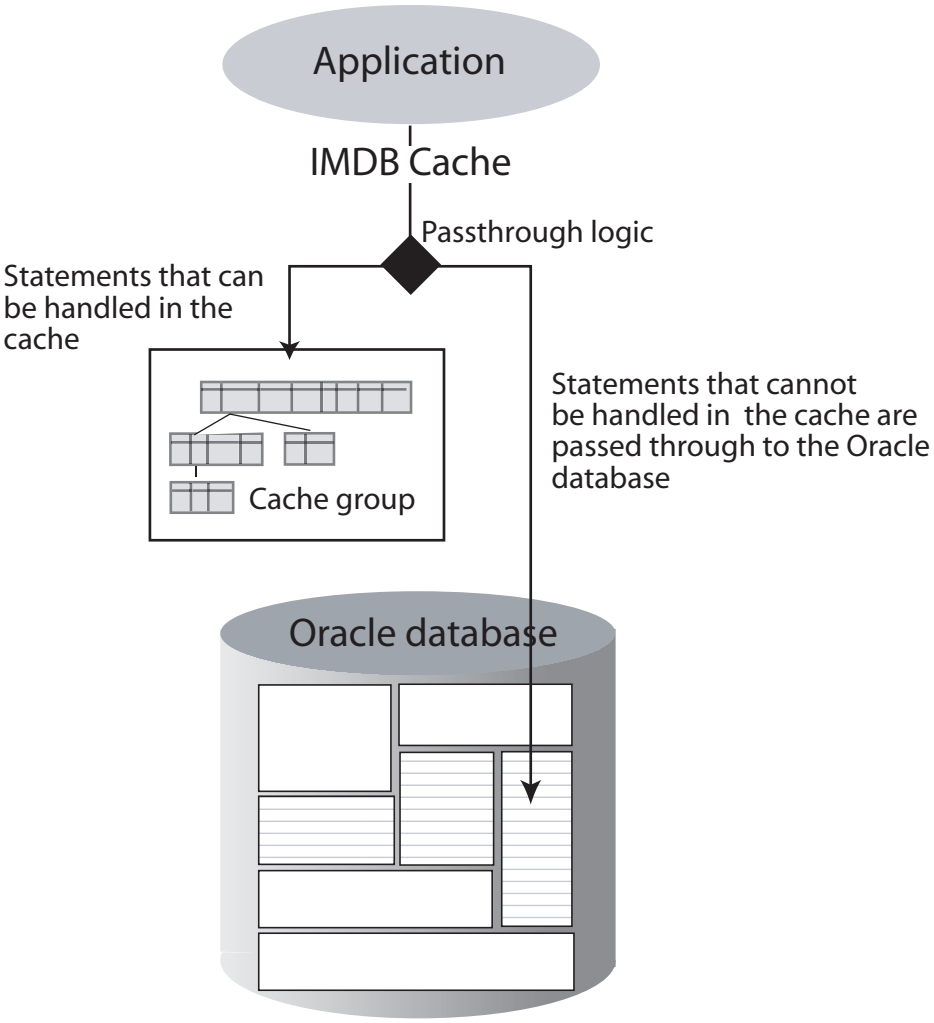
Records can be automatically aged out of a TimesTen database, and cache instances can be automatically aged out of an IMDB Cache database. Aging can be usage-based or time-based.You can configure both usage-based and time-based aging in the same system, but you can define only one type of aging on a specific cache group.

Dynamic load can be used to reload a requested cache instance that has been deleted by aging.

Passthrough feature

Applications can send SQL statements to either a cache group or to the Oracle database through a single connection to an IMDB Cache. This single-connection capability is enabled by a *passthrough* feature that checks whether the SQL statement can be handled locally by the cached tables in the IMDB Cache or if it must be redirected to the Oracle database, as shown in Figure 8–3. The passthrough feature provides settings that specify what types of statements are to be passed through and under what circumstances. The specific behavior of the passthrough feature is controlled by the `PassThrough` IMDB Cache general connection attribute.

Figure 8–3 Passthrough feature



Replicating cache groups

You can use an active standby pair to replicate AWT cache groups and read-only cache groups.

You can recover from a complete failure of a site by creating a special disaster recovery read-only subscriber as part of the active standby pair configuration. This special subscriber, located at a remote disaster recovery site, can propagate updates to a second Oracle database, also located at the disaster recovery site.

For more information

For more information about IMDB Cache, see *Oracle In-Memory Database Cache User's Guide*.

For more information about aging in cache groups, see "Implementing aging in a cache group" in *Oracle In-Memory Database Cache User's Guide*. For information about aging in tables that are not in cache groups, see "Implementing aging in your tables" in *Oracle TimesTen In-Memory Database Operations Guide*.

For more information about the passthrough feature, see "Setting a passthrough level" in *Oracle In-Memory Database Cache User's Guide*.

For more information about replicating cache groups, see "Cache groups and replication" in *Oracle TimesTen In-Memory Database Replication Guide*.

9

TimesTen and IMDB Cache Administration

This chapter includes the following topics:

Installing TimesTen and IMDB Cache

TimesTen and IMDB Cache software is easy to install. On UNIX systems, TimesTen is installed by a simple set-up script. On Windows, TimesTen is installed by running InstallShield®.

Access Control

TimesTen and IMDB Cache are installed with *Access Control* to allow only users with specific privileges to access particular TimesTen features.

TimesTen Access Control uses standard SQL statements to establish TimesTen user accounts with specific privilege levels. TimesTen offers object-level access control as well as database-level access control.

Command line administration

Most TimesTen and IMDB Cache administration tasks are performed with command line utilities. The following table summarizes common utilities:

Name	Description
`ttAdmin`	A general utility for managing TimesTen databases and IMDB Caches. Used to specify policies for automatically or manually loading and unloading databases from RAM, as well as to starting and stopping TimesTen cache agents and replication agents.
`ttBackup` and `ttRestore`	Used to create a backup copy of a database and restore it at a later time.

Name	Description
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 databases.
ttMigrate	Used to save tables and cache group definitions to a binary data file. Also used to restore tables and cache group definitions from the binary file.
ttRepAdmin	Used to monitor replication status.
ttSize	Used to estimate the amount of space to allocate for a table in the database.
ttStatus	Used to display information that describes the current state of TimesTen or IMDB Cache.
ttTraceMon	Used to enable and disable the TimesTen and IMDB Cache internal tracing facilities.
ttXactAdmin	Used to list ownership, status, log and lock information for each outstanding transaction. The ttXactAdmin utility also allows users to commit, abort or forget an XA transaction branch.

SQL administration

TimesTen provides SQL statements for administrative activities such as creating and managing tables, replication schemes, cache groups, materialized views, and indexes.

The metadata for each TimesTen database is stored in a group of system tables. Applications can use SQL SELECT queries on these tables to monitor the current state of a database.

Administrators can use the ttIsql utility for SQL interaction with a database. For example, there are several built-in ttIsql commands that display information on database structures.

SQL Developer

Oracle SQL Developer is a graphical tool for database development tasks. Use SQL Developer to:

- Browse, create, and edit database objects and PL/SQL programs
- Automate cache group operations
- Manipulate and export data
- Execute SQL and PL/SQL statements and scripts
- View and create reports

SQL Developer is a Java application that supports direct-linked and client/server connections to the TimesTen databases. Support for connecting to multiple databases enables SQL Developer users to work with data in the TimesTen and the Oracle databases concurrently.

ODBC Administrator

The ODBC Administrator is a utility program used on Windows to create, configure and delete data source definitions. You can use it to define a data source and set connection attributes.

Upgrading TimesTen and the IMDB Cache

TimesTen and the IMDB Cache provide the facilities to perform three types of upgrades:

- In-place upgrades
- Offline upgrades
- Online upgrades

In-place upgrades

In-place upgrades are typically used to move to a new patch release of TimesTen or IMDB Cache.

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

Offline upgrades

Offline upgrades are performed by using the `ttMigrate` utility to export the database into an external file and to restore the database with the desired changes.

Use offline upgrades to perform the following tasks:

- Move to a new major TimesTen or IMDB Cache release
- Move to a different directory or machine
- Reduce database size

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

Online upgrades

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

Use online upgrades to perform the following tasks:

- Move to a new major release of TimesTen or IMDB Cache and retain continuous availability to the database
- Increase or reduce the database size
- Move the database to a new location or machine

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

For more information

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

For more information about Access Control, see *Oracle TimesTen In-Memory Database Operations Guide*.

For more information about SQL Developer, see *Oracle SQL Developer TimesTen In-Memory Database Support User's Guide*.

For more information about general administration of TimesTen, see "Managing TimesTen Databases" and "Working with Data in a TimesTen Database" in *Oracle TimesTen In-Memory Database Operations Guide*. These chapters include the use of the ODBC Administrator.

For more information about administering TimesTen replication, see *Oracle TimesTen In-Memory Database Replication Guide*.

For a complete list of SQL statements, see *Oracle TimesTen In-Memory Database SQL Reference*.

For a complete list of TimesTen command-line utilities, see *Oracle TimesTen In-Memory Database Reference*.

Index

G

H

I

J

L

M

O

P

Q

R

S

T

U

X