Oracle® TimesTen In-Memory Database

Cache Guide

Release 26.1
F54449-01
March 2026

Oracle TimesTen In-Memory Database Cache Guide, Release 26.1

F54449-01

Contents

About This Content

This document covers TimesTen support for cache operations.

Audience

This guide is for anyone developing or supporting applications to cache data from an Oracle database in a TimesTen database. Cache operations enable the caching of subsets of an Oracle database into cache tables within a TimesTen database for improved response time in the application tier. Cache tables can be read-only or updatable. Applications read and update the cache tables using standard Structured Query Language (SQL) while data synchronization between the TimesTen database and the Oracle database is performed automatically.

Conventions

The following text conventions are used in this document.

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.

1

Paths to Explore Cache in TimesTen

TimesTen cache is a robust feature with many concepts and options for caching data between a TimesTen database and an Oracle database. When you are starting to learn about caching within TimesTen, there are three learning paths:

- Accelerate your Applications - Achieve Blazing Fast SQL With an Oracle TimesTen Cache. LiveLab: This LiveLab will help you to become familiar with setting up and using a TimesTen cache. The lab focuses briefly on concepts and heavily on the tasks for how to quickly configure, create, and use a TimesTen cache.
- *Oracle TimesTen In-Memory Database Getting Started with Cache Guide*: After going through the LiveLab, use this getting started guide for an overview of the basics of cache provided by TimesTen. This guide covers the most popular options and uses the default options when showing you how to create the three most popular cache group types. Start with Overview of Cache.
- *Oracle TimesTen In-Memory Database Cache Guide*: After you are familiar with the basics of caching, you can see a full explanation of concepts and details of the TimesTen cache feature. This is the advanced guide for cache. Start with Overview of Cache Groups.

2

Cache Concepts

Use caching to improve the performance for your applications access to data. You can cache Oracle Database data and reduce the workload on the Oracle database.

The TimesTen mechanism that enables read and write caching for Oracle database tables is called a cache group. A cache group can represent one or more related tables on an Oracle database. Each Oracle table is represented in the cache group with a cache table. You can read from or write to the cache tables. TimesTen connects to the backend Oracle database to load or update data as appropriate.

TimesTen supports all cache group types. See Cache Group Types.

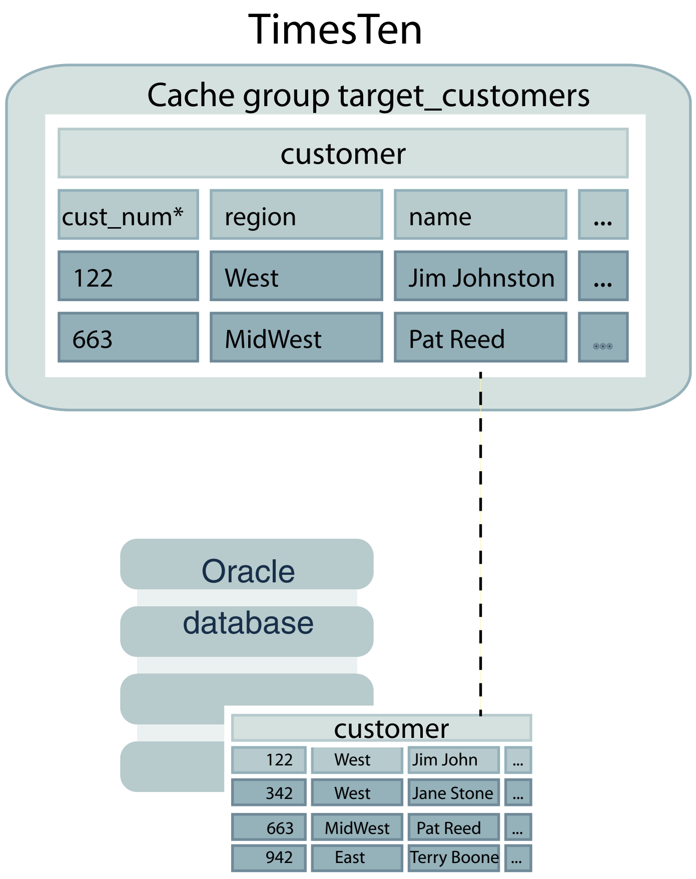
This chapter includes the following topics:

Overview of Cache Groups

Cache groups define the Oracle database data to be cached in a TimesTen database. A cache group can be defined to cache all or part of a single Oracle database table or a set of related Oracle database tables.

Figure 2-1 shows the `target_customers` cache group that caches a subset of a single Oracle Database table `customer`.

Figure 2-1 Single Table Cache Group



You can cache multiple Oracle database tables in the same cache group by defining a root table and one or more child tables. A cache group can contain only one root table.

The root table does not reference any table with a foreign key constraint. In a cache group with multiple tables, each child table must reference the root table or another child table in the same cache group using a foreign key constraint. Cache tables in a multiple-table cache group must be related to each other in the TimesTen database through foreign key constraints.

See Multiple-Table Cache Group.

While you may have multiple TimesTen databases that synchronize with the same Oracle database, they each operate independently. Thus, any data cached in separate TimesTen databases each synchronize with the Oracle database independently.

An Oracle database table cannot be cached in separate cache groups within the same TimesTen database. However, the table can be cached in separate cache groups within different TimesTen databases. If the table is cached in separate cache groups and the same cache instance is changed simultaneously on multiple TimesTen databases, there is no guarantee as to the order in which the changes are propagated to the cached Oracle database table. The contents of such cache groups in different TimesTen databases may not be consistent at a given point in time.

Cache Instance

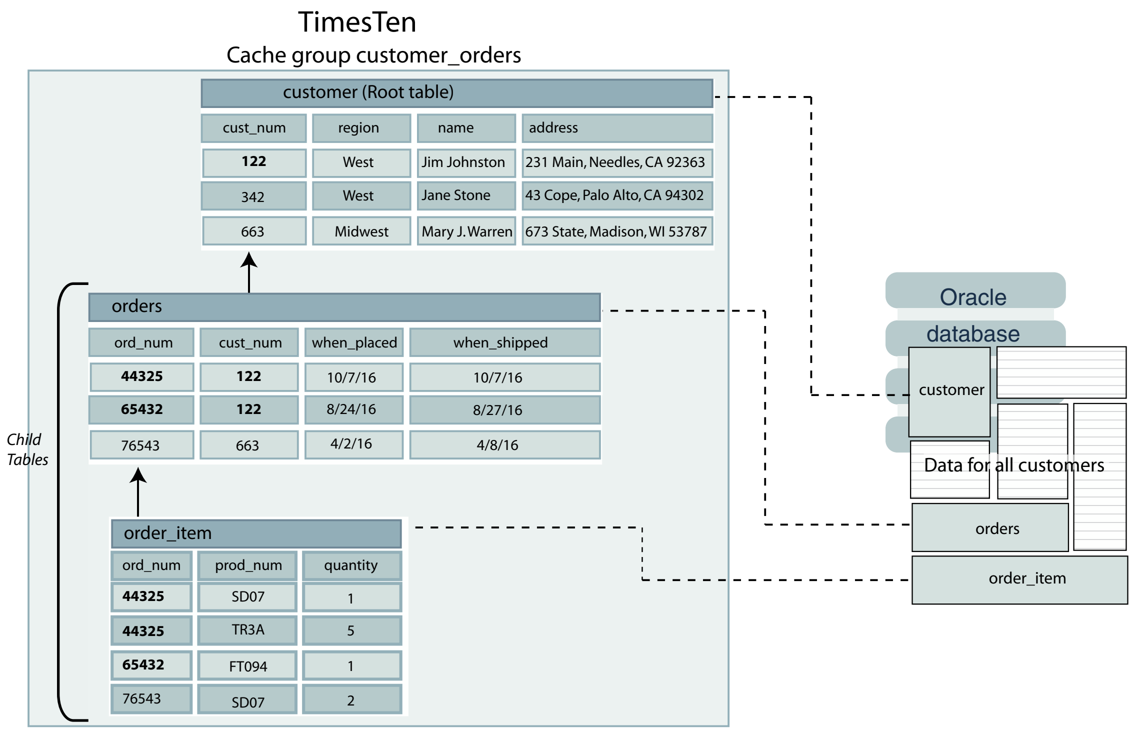
Data is loaded from an Oracle database into a cache group within a TimesTen database in units called cache instances.

A cache instance is defined as a single row in the cache group's root table together with the set of related rows in the child tables.

Figure 2-2 shows three tables in the `customer_orders` cache group. The root table is `customer`. `orders` and `order_item` are child tables. The cache instance identified by the row with the value `122` in the `cust_num` primary key column of the customer table includes:

- The two rows with the value `122` in the `cust_num` column of the orders table (whose value in the `ord_num` primary key column is `44325` or `65432`), and
- The three rows with the value `44325` or `65432` in the `ord_num` column of the `order_item` table

Figure 2-2 Multiple-Table Cache Group



customer (Root table)

cust_num	region	name	address
122	West	Jim Johnston	231 Main, Needles, CA 92363
342	West	Jane Stone	43 Cope, Palo Alto, CA 94302
663	Midwest	Mary J. Warren	673 State, Madison, WI 53787

orders

ord_num	cust_num	when_placed	when_shipped
44325	**122**	10/7/16	10/7/16
65432	**122**	8/24/16	8/27/16
76543	663	4/2/16	4/8/16

order_item

ord_num	prod_num	quantity
44325	SD07	1
44325	TR3A	5
65432	FT094	1
76543	SD07	2

Cache Group Types

There are several cache group types from which you can choose depending on the application needs.

TimesTen supports the following types of cache groups:

- Read-only cache group

A read-only cache group enforces a caching behavior in which committed changes on cached tables in the Oracle database are automatically refreshed to the cache tables in the TimesTen database. Using a read-only cache group is suitable for reference data that is heavily accessed by applications.

The two types of read-only cache groups are:

- Static read-only cache group: With a static read-only cache group, you use manual load requests to load data. You can use manual refresh requests. However, most read-only cache groups use autorefresh operations to refresh modified data at specified time intervals. When using autorefresh, the manual refresh requests are not necessary.
- Dynamic read-only cache group: With a dynamic read-only cache group, the application relies on data dynamically loading when data is requested with a qualified `SELECT... WHERE` SQL statement. Most read-only cache groups use autorefresh operations to refresh modified data at specified time intervals.

Log-based cache groups offer an alternative to traditional read-only caching by using redo logs to synchronize data between Oracle and TimesTen. This approach is well-suited for both static and dynamic cache groups, where the data is regularly refreshed and kept in sync with Oracle. See Log-Based Read-Only Cache Groups with TimesTen Replication.

See Read-Only Cache Group.

- Asynchronous WriteThrough (AWT) cache group

  An AWT cache group enforces a caching behavior in which committed changes on cache tables in the TimesTen database are automatically propagated to the cached tables in the Oracle database in asynchronous fashion. Using an AWT cache group is suitable for high speed data capture and online transaction processing.

  See Asynchronous WriteThrough (AWT) Cache Group.

Other types of cache groups include:

- Synchronous writethrough (SWT) cache group

  An SWT cache group enforces a caching behavior in which committed changes on cache tables in the TimesTen database are automatically propagated to the cached tables in the Oracle database in synchronous fashion.

  See Synchronous WriteThrough (SWT) Cache Group.

- User managed cache group

  A user managed cache group defines customized caching behavior.

  For example, you can define a cache group that does not use automatic refresh or automatic propagation where committed changes on the cache tables are manually propagated or flushed to the cached Oracle Database tables.

  You can also define a cache group that uses both automatic propagation in synchronous fashion on every table and automatic refresh.

  TimesTen supports user managed cache groups.

  See User Managed Cache Group.

- Hybrid cache group

  All other cache groups require multiple table cache groups to have strict parent-child relationships for all tables on a TimesTen database as well as the Oracle database. With hybrid cache groups, the cache tables on a Oracle database must be related, but the root (parent) table must only exist on the TimesTen database. That is, you can dynamically load from cache tables that do not have a root table on the Oracle database. A hybrid cache group is a dynamic read-only cache group where the root table is created in the TimesTen database and does not exist in the Oracle database.

See Hybrid Cache Group.

Transmitting Changes Between the TimesTen and Oracle Databases

Transmitting committed changes between the TimesTen cache tables and the cached Oracle Database tables keeps these tables in the two databases synchronized.

You can transmit changes between TimesTen and Oracle databases manually or automatically.

- Manually load cache groups: You can manually load cache instances that are not in the TimesTen cache tables from the Oracle database tables using `LOAD CACHE GROUP` statement. This statement only loads committed inserts on the cached Oracle database tables into the TimesTen cache tables. New cache instances are loaded into the cache tables, but cache instances that already exist in the cache tables are not updated or deleted even if the corresponding rows in the cached Oracle database tables have been updated or deleted. A load operation is primarily used to initially populate a cache group.
- Manually refresh cache groups: You can manually refresh cache instances into the TimesTen cache tables from the Oracle database tables using the `REFRESH CACHE GROUP` statement. This statement replaces cache instances in the TimesTen cache tables with the most current data from the cached Oracle database tables including cache instances that are already exist in the cache tables. A refresh operation is primarily used to update the contents of a cache group with committed changes on the cached Oracle database tables after the cache group has been initially populated.
- Manually propagate committed changes: Use a `FLUSH CACHE GROUP` statement to manually propagate committed changes on the TimesTen cache tables to the cached Oracle database tables.
- Dynamically load cache groups: A dynamic cache group is one that is created with the `DYNAMIC` keyword. Data is dynamically loaded on demand into the TimesTen cache tables from the cached Oracle database tables for dynamic cache groups when a qualifying `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statement is issued on one of the cache tables. A cache instance is automatically loaded from the cached Oracle database tables when a qualified statement does not find the data in the cache table, but the data exists in the cached Oracle database table. Typically, data automatically ages out from dynamically loaded cache tables when it is no longer being used. This action is similar to a `LOAD CACHE GROUP` statement, but dynamically issued. Dynamic cache groups are only supported in TimesTen.

  Note

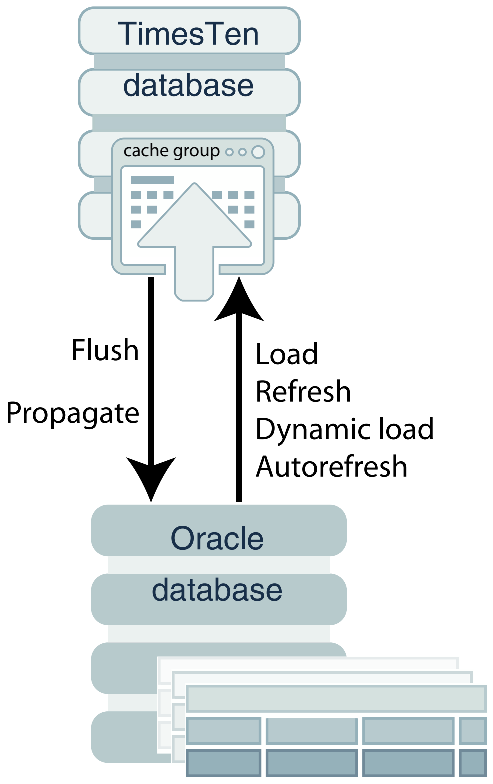
  A static cache group is one that is created without the `DYNAMIC` keyword.

- Automatically refresh cache groups: Autorefresh operations automatically replace cache instances in the TimesTen cache tables with the most current data from the cached Oracle database tables including cache instances that already exist in the cache tables. Autorefresh operations update the contents of a cache group with committed changes on the cached Oracle database tables after the cache group has been initially populated. This action is similar to a `REFRESH CACHE GROUP` statement, but automatically performed. Cache instances are automatically refreshed when the cache group is created with the `AUTOREFRESH` cache table attribute. The `AUTOREFRESH` cache group attribute can be used in a read-only or a user managed cache group to automatically refresh committed changes on cached Oracle Database tables into the TimesTen cache tables. The `AUTOREFRESH` cache group attribute can be defined on static or dynamic cache groups.

- Automatic propagation of changes to the Oracle database: When you specify the `PROPAGATE` cache table attribute when creating AWT, SWT, or user managed cache groups, then committed changes on cache tables in the TimesTen database are automatically propagated to the cached Oracle Database tables. This action is similar to a `FLUSH CACHE GROUP` statement, but automatically performed.

Load, refresh, dynamic load and autorefresh are operations that transmit committed changes on cached tables in the Oracle database to the cache tables in the TimesTen database. Load and refresh are manual operations; dynamic load and autorefresh are automatic operations. Propagate and flush are operations that transmit committed changes on cache tables in the TimesTen database to the cached tables in the Oracle database. Flush is a manual operation and propagate is an automatic operation.

Figure 2-3 Transmitting Committed Changes Between the TimesTen and Oracle Databases



The `DYNAMIC` keyword designates whether the cache group is a static or dynamic cache group:

- Static cache group: Defined when the `DYNAMIC` keyword is not supplied when creating the cache group. In a static cache group, cache instances are loaded manually into the TimesTen cache tables from an Oracle database. Within a static cache group, data is initially loaded into the cache tables from the cached Oracle database tables using a `LOAD CACHE GROUP` statement. After which, you can refresh the data with a `REFRESH CACHE GROUP` statement or automatically refresh the data if defined to use autorefresh. Once the cache tables are loaded, the user can run queries. A static cache group is appropriate when the set of data to cache is static and can be predetermined before applications begin performing operations on the cache tables. By default, cache groups are static.

- Dynamic cache group: Defined when the cache group is created with the `DYNAMIC` keyword. Within a dynamic cache group, data can be loaded into the cache group from an Oracle database either dynamically on demand or manually with `LOAD CACHE GROUP` or `REFRESH CACHE GROUP` statements. A manual refresh or an autorefresh operation on a dynamic cache group can result in existing cache instances being updated or deleted, but committed changes on Oracle database data that are not being cached do not result in new cache instances being loaded into its cache tables. A dynamic cache group is appropriate when the set of data you need to cache is small compared to the full size of the data that exists in the tables in the Oracle database.

  The data should be preloaded from the Oracle database before applications perform operations on the cache tables.

Choose static or dynamic load when deciding how much data you want to cache. Ideally, a manual load is faster. Use dynamic load to automate loading new data or to specify how much data to load into memory.

Any cache group type (read-only, AWT, SWT, user managed) can be defined as a static cache group. All cache group types except a user managed cache group that uses both the `AUTOREFRESH` cache group attribute and the `PROPAGATE` cache table attribute can be defined as a dynamic cache group.

See Methods for Transmitting Changes Between TimesTen and Oracle Databases.

See Asynchronous WriteThrough (AWT) Cache Group and Synchronous WriteThrough (SWT) Cache Group.

Using Oracle GoldenGate as an Alternative Cache Refresh Mechanism

You may prefer to use Oracle GoldenGate to refresh data from the backend Oracle database to TimesTen instead of using the built-in native cache refresh mechanism of TimesTen.

You can use GoldenGate instead of the native cache refresh mechanism of TimesTen to provide the equivalent of static read-only cache groups. All other types of cache functionality must use the TimesTen native cache mechanism.

The following are the advantages when using GoldenGate as your cache refresh mechanism:

- GoldenGate provides a lighter weight change data capture pipeline on the Oracle database, especially if you have multiple TimesTen databases caching data from the same Oracle database.
- The triggers and tracking tables that are required by TimesTen for cache operations in an Oracle Database are not required.
- You can cache data from multiple Oracle databases into a single TimesTen database.
- You can cache data from a backend database that is not an Oracle database if the database supported by GoldenGate.

See Using GoldenGate as an Alternative to Native Read-Only Cache Groups.

High Availability Caching Solution

You can configure cache to achieve high availability of cache tables, and facilitate failover and recovery while maintaining connectivity to the Oracle database.

A TimesTen database that is a participant in an active standby pair replication scheme can provide high availability for cache tables in a read-only or an AWT cache group.

An active standby pair provides for fault tolerance of a TimesTen database. Oracle Real Application Clusters (Oracle RAC) and Data Guard provides for high availability of an Oracle database.

See Replicating Cache Tables in TimesTen, Using Cache in an Oracle RAC Environment, and Using Cache with Data Guard.

3
Setting Up a Caching Infrastructure

Before you can start caching Oracle database data in a TimesTen database, you must first install TimesTen.

Follow the instructions provided in Overview of the Installation Process in TimesTen Classic in the *Oracle TimesTen In-Memory Database Installation, Migration, and Upgrade Guide*. After which, perform these tasks for setting up the TimesTen and Oracle database systems:

- Configuring the Oracle Database to Cache Data
- Configuring a TimesTen Database to Cache Oracle Database Data
- Testing the Connectivity Between the TimesTen and Oracle Databases
- Managing the Cache Agent

Configuring the Oracle Database to Cache Data

The following sections describe the tasks that must be performed on the Oracle database by the `sys` user:

- Create the Oracle Database Users and Default Tablespace
- Grant Privileges to the Oracle Cache Administration User
- Create Oracle Database Objects Used to Manage Data Caching

Create the Oracle Database Users and Default Tablespace

Create a default tablespace to store meta information about cache operations. Create a cache administration user that creates, owns, and maintains Oracle database objects that store information used to manage the cache environment for a TimesTen database and enforce predefined behaviors of particular cache group types.

Perform the following on the Oracle database:

> **Note**
>
> If you are using Oracle Autonomous Database Serverless or Oracle Autonomous Database on Dedicated Exadata Infrastructure for the Transaction Processing workload type, use the preconfigured databases services LOW or TP. In addition, if you are using a multitenant container database (CDB) or pluggable database (PDB), note the specific instructions below on how to create the cache administration user and grant this user privileges in a CDB or PDB.

1. Create a default tablespace that stores information about cache operations.

   This tablespace is used for storing cache management objects that should not be shared with other applications. While you may also store Oracle database tables that are cached in a TimesTen database, we strongly recommend that this tablespace be used solely by the TimesTen database for cache management.

See Managing a Cache Environment with Oracle Database Objects for a list of Oracle database tables used by the cache administration user.

In the following SQL*Plus example, the default tablespace is `cachetblsp` and defines a 5 GB data file named `tt_cache.f`. Choose a size that is appropriate for your particular needs. Provide the `SEGMENT SPACE MANAGEMENT AUTO` clause so that the Oracle database automatically manages the free space of all segments in the tablespace (useful for monitoring autorefresh).

Note

The use of the `sys@tnsservicename as sysdba` user in this example is applicable only for a test environment.

```
% cd timesten_home/install/oraclescripts
% sqlplus sys@tnsservicename as sysdba
Enter password: password
```

This example uses the `sys@tnsservicename as sysdba` user since the `sys@tnsservicename` user is able to grant the required privileges. For the Transaction Processing workload type, use the `admin` user instead. You can use any Oracle database user that has the appropriate privileges. See Required Privileges for Cache Administration User for Cache Operations.

For the non-autonomous Oracle Database, use SQL*Plus to create a default tablespace. In the following example, the name of the default tablespace is `cachetblsp`:

```
SQL> CREATE TABLESPACE cachetblsp DATAFILE 'tt_cache.f' SIZE 5G
 SEGMENT SPACE MANAGEMENT AUTO;

Tablespace created.
```

Skip this step for the AutonomousTransaction Processing. Autonomous Transaction Processing automatically configures default data and temporary tablespaces for the database. Adding, removing, or modifying tablespaces is not allowed. Autonomous Transaction Processing creates one or multiple tablespaces automatically depending on the storage size.

2. Create an Oracle cache administration user that creates, owns, and maintains Oracle database objects that store information used to manage the cache environment for a TimesTen database and enforce predefined behaviors of particular cache group types.

   If you are using a multitenant container database (CDB) or pluggable database (PDB), the Oracle cache administrator user can be one of the following:

   - Local user: A local user is a database user that can operate only within a single PDB. You must assign cache privileges only within the PDB in which this user exists.
   - Common user: A common user is a database user known in every container and has the same identity in the CBD root and in every existing and future PDB in the CDB. You must assign cache privileges within each PDB in the CDB in which you want to use cache.

Note

Each TimesTen database can be managed by only a single cache administration user on the Oracle database. However, a single cache administration user can manage multiple TimesTen databases. You can specify one or more cache administration users where each manages one or more TimesTen databases.

See Caching the Same Oracle Table on Two or More TimesTen Databases.

Designate the tablespace as the default tablespace for the Oracle cache administration user. This user creates tables in this tablespace that are used to store information about the cache environment and its cache groups. Other Oracle database objects (such change log tables, replication metadata tables, and triggers) are used to enforce the predefined behaviors of cache groups with autorefresh and AWT cache groups are created in the same tablespace. To create and manage these objects, the Oracle cache administration user must have a high level of privileges. A cache group with autorefresh refers to a read-only cache group or a user managed cache group that uses the `AUTOREFRESH MODE INCREMENTAL` cache group attribute.

See Managing a Cache Environment with Oracle Database Objects for a list of Oracle Database tables and triggers owned by the cache administration user.

Note

If you create multiple cache administration users, each may use the same or different tablespace as their default tablespace.

As the `sys` user, use SQL*Plus to create the Oracle database cache administration user `cacheadmin`. In the example below, the default tablespace for the `cacheadmin` user is `cachetblsp`.

For the non-autonomous Oracle Database, the following SQL*Plus example creates the cache administration user:

```
SQL> CREATE USER cacheadmin IDENTIFIED BY orapwd
    DEFAULT TABLESPACE cachetblsp QUOTA UNLIMITED ON cachetblsp;
```

For Autonomous Transaction Processing, the following SQL*Plus example creates the cache administration user:

```
SQL> CREATE USER cacheadmin IDENTIFYED BY orapwd
    QUOTA UNLIMITED ON DATA;
```

3. Identify one or more existing schemas (or create a new schema) with schema owners that own Oracle database tables that are to be cached in a TimesTen database. The tables to be cached may or may not already exist.

Grant Privileges to the Oracle Cache Administration User

The cache administration user must be granted a high level of privileges depending on the cache group types created and the operations performed on these cache groups.

The main privileges required for the Oracle cache administration user can be granted in bulk by running the SQL*Plus script *`timesten_home`*`/install/oraclescripts/`

`grantCacheAdminPrivileges.sql` as the `sys` user. This script grants the cache administration user the minimum set of privileges required to perform cache operations.

If you are using a multitenant container database (CDB) or pluggable database (PDB), run the SQL*Plus script *`timesten_home`*`/install/oraclescripts/grantCacheAdminPrivileges.sql` to assign cache privileges as follows:

- If the cache administrator user is a local user: You must assign cache privileges only within the PDB in which this user exists. This is the preferred method.
- If the cache administrator user is a common user: You must assign cache privileges within each PDB in the CDB in which you want to use cache. Do not run the SQL*Plus script to grant privileges to the common user in the CBD root.

See Create Oracle Database Objects Used to Manage Data Caching.

You also need to grant the Oracle cache administration user privileges based on the type of cache operation. The entire list of privileges required for this user for each cache operation are listed in Required Privileges for Cache Administration User for Cache Operations.

Create Oracle Database Objects Used to Manage Data Caching

You request TimesTen to create Oracle database objects owned by the cache administration user, such as cache and replication metadata tables, change log tables, and triggers when particular cache environment and cache group operations are performed.

Some of these objects are used to enforce the predefined behaviors of cache groups with autorefresh and AWT cache groups.

These Oracle database objects are automatically created if the cache administration user has been granted the required privileges with one of the following SQL*Plus scripts:

- The grantCacheAdminPrivileges.sql Script: Run this script to grant all required privileges to the cache administration user that are required to create Oracle database objects used to manage the caching of Oracle database data when particular cache group operations are performed. The cache administration user then automatically creates Oracle database objects used to manage caching Oracle database data in a TimesTen database.
- The initCacheAdminSchema.sql Script: Run this script to grant all required privileges except for the `CREATE CLUSTER`, `CREATE INDEXTYPE`, `CREATE OPERATOR`, `CREATE PROCEDURE`, `CREATE SEQUENCE`, `CREATE TABLE`, and `EXECUTE ON SYS.DBMS_LOB` package privileges. For security reasons, you may not want to grant these privileges. The cache administration user then automatically creates all Oracle database objects used to manage caching Oracle database data in a TimesTen database, except for cache groups that use autorefresh.

If you want to check if the Oracle cache administration user has all of the necessary privileges that are required for cache operations, you can run the The checkAdminPrivileges.sql Script.

The grantCacheAdminPrivileges.sql Script

The `grantCacheAdminPrivileges.sql` script grants privileges to the cache administration user that are required to automatically create Oracle Database objects used to manage the caching of Oracle Database data when particular cache group operations are performed.

See Required Privileges for Cache Administration User for Cache Operations for a complete list of privileges that need to be granted to the cache administration user in order to perform particular cache group and cache table operations.

Run the `timesten_home/install/oraclescripts/grantCacheAdminPrivileges.sql` as the `sys` user. The cache administration user name is passed as an argument to the `grantCacheAdminPrivileges.sql` script.

Note

Alternatively, you can create these objects as described in The initCacheAdminSchema.sql Script before performing any cache group operations if, for security purposes, you do not want to grant certain privileges to the cache administration user required to automatically create objects necessary for managing autorefresh.

In addition to the privileges granted to the cache administration user by running the `grantCacheAdminPrivileges.sql` script, this user may also need to be granted privileges such as `SELECT` or `INSERT` on the cached Oracle Database tables depending on the types of cache groups you create, and the operations that you perform on the cache groups and their cache tables.

As the `sys` user, use SQL*Plus to run the `grantCacheAdminPrivileges.sql` script to grant privileges to the cache administration user. The cache administration user then automatically creates Oracle Database objects used to manage caching Oracle Database data in a TimesTen database.

The `grantCacheAdminPrivileges.sql` script requires the Oracle database cache administration user name as input, which is `cacheadmin` in this example.

```
SQL> @grantCacheAdminPrivileges cacheadmin
SQL> exit
```

For example, with cache groups with autorefresh, the Oracle database objects used to enforce the predefined behaviors of these cache group types are automatically created if the objects do not already exist and one of the following occurs:

- The cache group is created with its autorefresh state set to `PAUSED` or `ON`.
- The cache group is created with its autorefresh state set to `OFF` and then altered to either `PAUSED` or `ON`.

The initCacheAdminSchema.sql Script

The Oracle database cache administration user requires certain privileges to automatically create the Oracle database objects.

The cache administration user requires privileges used to:

- Store information about TimesTen databases that are associated with a particular cache environment.
- Enforce the predefined behaviors of cache groups with autorefresh. In this case, the cache administration user requires certain privileges to automatically create these Oracle database objects.
- Enforce the predefined behavior for AWT cache groups.

For security purposes, if you do not want to grant the `CREATE CLUSTER`, `CREATE INDEXTYPE`, `CREATE OPERATOR`, `CREATE PROCEDURE`, `CREATE SEQUENCE`, `CREATE TABLE`, and `EXECUTE ON SYS.DBMS_LOB` package privileges to the cache administration user required to automatically create the Oracle Database objects, you can use the `initCacheAdminSchema.sql` script. See Required Privileges for Cache Administration User for Cache Operations for a full list of privileges granted by this script.

To create the Oracle Database tables and triggers used to enforce the predefined behaviors of particular cache group types, run the SQL*Plus script *`timesten_home`*`/install/oraclescripts/initCacheAdminSchema.sql` as the `sys` user. These objects must be created before you can create cache groups with autorefresh and AWT cache groups. The `initCacheAdminSchema.sql` script requires the cache administration user name as input.

In addition to the privileges granted to the cache administration user by running the `initCacheAdminSchema.sql` script, you may need to grant the user privileges such as `SELECT` or `INSERT` on the cached Oracle Database tables depending on the types of cache groups you create and the operations that you perform on the cache groups and their cache tables.

As the `sys` user, use SQL*Plus to run the `initCacheAdminSchema.sql` script to create Oracle Database objects, which are used to manage caching data. These Oracle Database objects enforce the predefined behaviors of a cache group with autorefresh and AWT cache groups, and grant a limited set of privileges to the cache administration user. In the following example, the Oracle database cache administration user name is `cacheadmin`.

```
SQL> @initCacheAdminSchema "cacheadmin"
SQL> exit
```

Other Oracle database objects associated with Oracle database tables that are cached in a cache group with autorefresh are needed to enforce the predefined behaviors of these cache group types. See Manually Creating Oracle Database Objects for Cache Groups with Autorefresh for details about how to create these additional objects as part of the steps for creating a cache group with autorefresh.

To view a list of the Oracle database objects created and used by TimesTen to manage the caching of Oracle database data, run the following query in SQL*Plus as the `sys` user:

```
SQL> SELECT owner, object_name, object_type FROM all_objects WHERE object_name
  2  LIKE 'TT\___%' ESCAPE '\';
```

The query returns a list of tables, indexes, and triggers owned by the cache administration user.

The checkAdminPrivileges.sql Script

The `checkAdminPrivileges.sql` script checks that the cache administration user has been granted the required privileges to automatically create Oracle Database objects used to manage the caching of Oracle Database data when particular cache group operations are performed. This script checks that the user running the script has all of the privileges granted in the `grantCacheAdminPrivileges.sql` script.

See Required Privileges for Cache Administration User for Cache Operations for a complete list of privileges that need to be granted to the cache administration user in order to perform particular cache group and cache table operations.

Run the *`timesten_home`*`/install/oraclescripts/checkAdminPrivileges.sql` as the cache administration user.

Use SQL*Plus on the Oracle Database system from an operating system shell or command prompt, and connect to the Oracle database instance as the cache administration user that you want checked for privileges. The following example shows that the user has all of the required privileges.

```
SQL> @checkAdminPrivileges.sql
**** Checking privileges for cache administrator user ****
**** User has all privileges for a cache administrator user ****
```

The following example shows the output if you have missing privileges needed as a cache administration user on an Oracle database:

```
SQL> @checkAdminPrivileges.sql
**** Checking privileges for cache administrator user ****
Missing CREATE OPERATOR
Missing CREATE INDEXTYPE
Missing CREATE CLUSTER
Missing EXECUTE ON SYS.DBMS_LOCK
Missing EXECUTE ON SYS.DBMS_DDL
Missing EXECUTE ON SYS.DBMS_FLASHBACK
Missing EXECUTE ON SYS.DBMS_LOB
Missing SELECT on SYS.GV$LOCK
Missing SELECT on SYS.GV$SESSION
Missing SELECT on SYS.DBA_DATA_FILES
Missing SELECT on SYS.V$DATABASE
Missing SELECT on GV$PROCESS
Missing UNLIMITED TABLESPACE
Missing SELECT ANY TRANSACTION
Missing table ARDL_CG_COUNTER
**** User missing privileges. Missing privilege count: 15 ****
```

Configuring a TimesTen Database to Cache Oracle Database Data

Certain operations must be performed on the TimesTen database by the instance administrator or the TimesTen cache administration user.

- Specify Database Connection Definition for Cache
- Create the TimesTen Users
- Grant Privileges to the TimesTen Users
- Providing Cache Administration User Credentials
- Registering the Cache Administration User Name and Password
- Cache Group Requirements for Credentials

Specify Database Connection Definition for Cache

You can modify certain cache-related connection attributes to define connection attributes.

- Set the Net Service Name for the Oracle Database in the tnsnames.ora File
- Define a DSN for the TimesTen Database

Set the Net Service Name for the Oracle Database in the tnsnames.ora File

For cache in TimesTen, set the `TNS_ADMIN` environment variable to indicate the full path to the directory where the `tnsnames.ora` file is located. This is for access to Oracle Database data.

For Autonomous Transaction Processing, use the preconfigured databases services LOW or TP:

- *`databasename`*`_tp`
- *`databasename`*`_low`

1. Ensure that the main daemon is stopped before you modify the `tnsnames.ora` file.

   ```
   ttDaemonAdmin -stop
   ```

2. Set the `TNS_ADMIN` location for the cache agent with the `ttInstanceModify -tnsadmin` option to set the path to the `tnsnames.ora` file. Specify the full path to the directory where the file is located.

   ```
   ttInstanceModify -tnsadmin /TimesTen/conf
   ```

3. For cache in TimesTen, set the `TNS_ADMIN` environment variable to indicate the full path to the directory where the `tnsnames.ora` file is located. Set this variable in the user's profile script so that it will persist.

   ```
   export TNS_ADMIN=/TimesTen/conf
   ```

4. Restart the main daemon to capture this setting.

   ```
   ttDaemonAdmin -start
   ```

Add the net service name for the non-autonomous Oracle Database into the `tnsnames.ora` file. The following is an example of defining `orcl` in a `tnsnames.ora` file:

```
orcl =
 (DESCRIPTION =
   (ADDRESS = (PROTOCOL = TCP)(HOST = myhost)
     (PORT = 1521))
   (CONNECT_DATA =
     (SERVICE_NAME = myhost.example.com)))
```

For Autonomous Transaction Processing, the following is an example of defining the `orcl_low` in a `tnsnames.ora` file:

```
orcl_low =
 (DESCRIPTION =
   (ADDRESS = (PROTOCOL = TCP)(HOST = adb.us-phoenix-1.oraclecloud.com)
     (PORT = 1521))
   (CONNECT_DATA =
     (SERVICE_NAME = orcl_low.adb.oraclecloud.com)))
```

Note

TimesTen supports both TCP and mTLS-based connections for Oracle Autonomous Database on Dedicated Exadata Infrastructure and only mTLS-based connections for Oracle Autonomous Database Serverless.

Define a DSN for the TimesTen Database

A TimesTen database that caches data from an Oracle database can be referenced by either a system DSN or a user DSN. A TimesTen database is implicitly created the first time the instance administrator user connects to it using a DSN. When creating a DSN for a TimesTen database that caches data from an Oracle database, pay special attention to the settings of the connection attributes.

See Managing TimesTen Databases in *Oracle TimesTen In-Memory Database Operations Guide*.

On UNIX or Linux, the system DSN is located in the *timesten_home*/conf/sys.odbc.ini file. As described in Connecting to a TimesTen Database, the sys.odbc.ini file contains the DSN definitions.

This example defines cache1 and cache1cs ODBC Data Source Names (DSNs).

Note

ODBC is TimesTen's native API, though TimesTen also provides, or supports, many other commonly used database APIs such as JDBC, Oracle Call Interface, ODP.NET, cx_Oracle (for Python) and node-oracledb (for Node.js).

- Direct connection: The cache1 DSN is a direct mode, or server DSN. It uses the TimesTen 22.1 Driver. It defines the parameters and connectivity for a database hosted by this TimesTen instance. Tools, utilities, and applications running on this host (myhost) can connect through this DSN using TimesTen's low latency 'direct mode' connectivity mechanism.
- Client-server connection: This database is also accessible remotely using TimesTen's client-server connectivity. The cache1cs DSN is a client DSN and uses the TimesTen 22.1 Client Driver. It defines connectivity parameters for a server DSN that tools, utilities, and applications can connect to using TimesTen's client-server connectivity mechanism. In this example, the DSN defines client-server access for the local cache1 server DSN.

All of these connection attributes can be set in a direct DSN or a connection string, unless otherwise stated.

- DataStore specifies the fully qualified directory path name of the database and the file name prefix. This name is not a file name. In this example, DataStore is set to /disk1/databases/database1.
- PermSize specifies the allocated size of the database's permanent region in MB. The PermSize value must be smaller than the physical RAM on the machine. Set this to a value that enables you to store all of your data. The PermSize value could be from a few GB to several TB. This example sets the permanent region to 1024 MB.
- TempSize indicates the total amount of memory in MB allocated to the temporary region for the database. This example sets the temporary region to 256 MB.
- LogBufMB specifies the size of the internal transaction log buffer for the database. This example sets the transaction log buffer to 256 MB.
- LogFileSize specifies the maximum size of transaction log files in megabytes. This example sets the maximum size of transaction log files to 256 MB.
- DatabaseCharacterSet must match the Oracle database character set. In this example, the database character set is AL32UTF8.

Note

You can determine the Oracle database character set by running the following query in SQL*Plus as any user:

```
SQL> SELECT value FROM nls_database_parameters
       WHERE parameter='NLS_CHARACTERSET';
```

- `ConnectionCharacterSet` specifies the character encoding for the connection. Generally, you should choose a connection character set that matches your terminal settings or data source. In this example, the connection character set is `AL32UTF8`.
- `OracleNetServiceName` must be set to the net service name of the Oracle database instance. This example sets this to `orcl`. This is the same name that was set in the `tnsnames.ora` file.

  For Microsoft Windows systems, the net service name of the Oracle database instance must be specified in the **Oracle Net Service Name** field of the TimesTen Cache tab within the TimesTen ODBC Setup dialog box.
- `CacheAdminWallet=1` specifies that credentials for the Oracle cache administration user that are registered with the `ttCacheUidPwdSet` built-in procedure are stored in an Oracle Wallet, rather than in memory.
- `UID` specifies the name of the TimesTen cache administration user. The `UID` connection attribute can be specified in a direct DSN, a client DSN, or a connection string.
- `PwdWallet` specifies the wallet in which credentials are stored for users. You can provide the TimesTen user name and password within a wallet. You can also provide the cache administrator users and respective passwords in a wallet. The cache administration user credentials are necessary when performing cache operations and connecting to the Oracle database.
- If you are not using `PwdWallet` to specify a wallet, then use `PWD` to specify the password of the TimesTen cache administration user specified in the `UID` connection attribute. The `PWD` connection attribute can be specified in a Direct DSN, a Client DSN, or a connection string.
- If you are not using `PwdWallet` to specify a wallet, then use `OraclePWD` to specify the password of the Oracle Database cache administration user that has the same name as the TimesTen cache administration user specified in the `UID` connection attribute.

  Note

  See Create the TimesTen Users.
- `PassThrough` can be set to control whether statements are to be run in the TimesTen database or passed through to be processed in the Oracle database. See Setting a Passthrough Level.
- `LockLevel` must be set to its default of 0 (row-level locking) because cache does not support database-level locking.
- `ReplicationApplyOrdering` and `CacheAWTParallelism` control parallel propagation of changes to TimesTen cache tables in an AWT cache group to the corresponding Oracle Database tables. See Improving AWT Throughput with Parallel Propagation to the Oracle Database.

Then, there is an entry for the client DSN. The client DSN specifies the location of the TimesTen database with the following attributes:

- The `TTC_Server_DSN` attribute specifies the server DSN of the intended database.
- The `TTC_Server` attribute specifies the server (and the port number if you do not want to use the default port number) for the database.

In the `sys.odbc.ini` file, create a TimesTen DSN `cache1` and set the following connection attributes. The `cache1` DSN specifies a TimesTen database that caches data from an Oracle database.

```
[ODBC Data Sources]
cache1=TimesTen 22.1 Driver
cache1cs=TimesTen 22.1 Client Driver

[cache1]
DataStore=/disk1/databases/database1
PermSize=1024
TempSize=256
LogBufMB=256
LogFileSize=256
DatabaseCharacterSet=AL32UTF8
ConnectionCharacterSet=AL32UTF8
OracleNetServiceName=orcl
CacheAdminWallet=1

[cache1cs]
TTC_SERVER_DSN=CACHE1
TTC_SERVER=myhost/6625
ConnectionCharacterSet=AL32UTF8
```

Create the TimesTen Users

First, you must create a user who performs cache group operations on the TimesTen database. We refer to this user as the TimesTen database *cache administration user*.

The TimesTen cache administration user must have the same name as the Oracle cache administration user that accesses the cached Oracle Database tables. The password of the TimesTen database cache administration user can be different than the password of the Oracle cache administration user.

Note

You can create multiple cache administration users on a TimesTen database, such as one for each TimesTen DBA. However, you can only define a single cache administration user on the Oracle database for this particular TimesTen database. (You can use the same Oracle cache administration user for all TimesTen databases that connect to the Oracle database or define a separate cache administration user for each TimesTen database.) If you create multiple TimesTen cache administration users, one or more of these users can use the same Oracle cache administration user.

The TimesTen cache administration user must be assigned privileges to perform cache operations. The TimesTen cache administration user creates the cache groups. It may perform operations such as loading or refreshing a cache group (although these operations can be performed by any TimesTen user that has sufficient privileges). The TimesTen cache administration user can also monitor various aspects of the caching environment, such as asynchronous operations that are performed on cache groups such as autorefresh.

The second user that you must create is a cache table user that owns the cache tables on TimesTen and has the same name as the Oracle Database schema owner who owns Oracle Database tables to be cached in the TimesTen database. We refer to these users as *cache table users*, because the TimesTen cache tables are to be owned by these users. Therefore, the owner and name of a TimesTen cache table is the same as the schema owner and name of the corresponding cached Oracle Database table. The password of a cache table user can be different than the password of the Oracle Database schema owner with the same name.

The following example creates the TimesTen users. It uses the `ttIsql` utility to connect to the `cache1` DSN as the instance administrator. One of the most frequently used TimesTen utilities is the `ttIsql` utility. This is an interactive SQL utility that serves the same purpose for TimesTen as SQL*Plus does for Oracle Database.

- Creates the TimesTen database cache administration user `cacheadmin` whose name (in this example) is the same as the Oracle cache administration user.
- Creates a cache table user `sales` whose name is the same as the Oracle Database schema owner of the Oracle Database tables to be cached in the TimesTen database.

```
% ttIsql cache1
Command> CREATE USER cacheadmin IDENTIFIED BY ttpwd;
Command> CREATE USER sales IDENTIFIED BY ttpwd;
```

Grant Privileges to the TimesTen Users

The privileges that the TimesTen users require depend on the types of cache groups you create and the operations that you perform on the cache groups.

All of the privileges required for the TimesTen cache administration user for each cache operation are listed in Required Privileges for Cache Administration User for Cache Operations.

You must grant required privileges to the cache administration user. This example grants the TimesTen cache administration user `cacheadmin` the following required privileges to perform the noted operations:

- Set the cache administration user and password (`CACHE_MANAGER`).
- Start or stop the cache agent and replication agent processes on the TimesTen database (`CACHE_MANAGER`).
- Set a cache agent start policy (`CACHE_MANAGER`).
- Set a replication agent start policy (`ADMIN`)
- Create cache groups to be owned by the TimesTen cache administration user (`CREATE [ANY] CACHE GROUP`, inherited by the `CACHE_MANAGER` privilege; `CREATE [ANY] TABLE` to create the underlying cache tables which are to be owned by the cache table user).
- Alter, load, refresh, flush, unload or drop a cache group requires the appropriate privilege:
  – `ALTER ANY CACHE GROUP`
  – `LOAD {ANY CACHE GROUP | ON` *`cache_group_name`*
  – `REFRESH {ANY CACHE GROUP | ON` *`cache_group_name`*
  – `FLUSH {ANY CACHE GROUP | ON` *`cache_group_name`*
  – `UNLOAD {ANY CACHE GROUP | ON` *`cache_group_name`*
  – `DROP ANY CACHE GROUP` and `DROP ANY TABLE`
- Required privileges for other cache operations, such as for read-only cache groups, dynamic load operations, incremental autorefresh, full autorefresh and asynchronous writethrough, are listed in Required Privileges for Cache Administration User for Cache Operations.

As the instance administrator, use the `ttIsql` utility to grant the `cacheadmin` cache administration user the required privileges:

```
Command> GRANT CREATE SESSION, CACHE_MANAGER, CREATE ANY TABLE TO cacheadmin;
Command> exit
```

Providing Cache Administration User Credentials

If you are running a request that does not require access to the Oracle database, you can proceed without needing to provide credentials for the Oracle database. That is, you can connect with only the user name and password for connecting to the TimesTen database. However, when you want to perform cache operations that require connecting to the Oracle database, then you must provide the appropriate credentials to be able to connect to both the TimesTen and Oracle databases.
Once the cache administration users are created with their respective passwords, these credentials need to be provided in two places for cache operations to proceed.

- Provided on the connection string: When you are connecting to the TimesTen database and are planning on performing cache operations that require TimesTen to connect to the Oracle database, the cache administration users and respective passwords are required. You can provide these either with the cache administration user and passwords saved in an Oracle Wallet (preferred) pointed to by the `PwdWallet` connection attribute or specified distinctly within the `UID`, `PWD`, `PWDCrypt`, and `OraclePWD` connection attributes. Providing credentials in a wallet is more secure than supplying a password on the connect string with the `PWD` or `PWDCrypt` connection attributes.

  See Providing Cache Administration User Credentials When Connecting.

- Registered internally within TimesTen: There are cache operations that TimesTen performs for you. In order for TimesTen to connect to the Oracle database successfully to perform these cache operations, TimesTen needs to have the Oracle cache administration user and password credentials registered internally. In TimesTen Classic, the internal registration is accomplished when you run the `ttCacheUidPwdSet` built-in procedure. You can specify that the results of the `ttCacheUidPwdSet` built in procedure are saved in an Oracle Wallet (preferred) or in memory. The credentials are saved within an Oracle Wallet when you set the `CacheAdminWallet=1` in the DSN as a first connection attribute (which is best set in the DSN).

  See Registering the Cache Administration User Name and Password.

Providing Cache Administration User Credentials When Connecting

When you are connecting to the TimesTen database with the intent on performing cache operations that require TimesTen to connect to the Oracle database, then the cache administration users and their respective passwords are required.

Supply the cache administration user credentials in the connection string either by:

- You can provide the Oracle and TimesTen cache administration users credentials within an Oracle Wallet. This method requires you to first save the cache administration user credentials in an identifiable Oracle Wallet with the `ttUser` utility. After creating the wallet, the particular wallet is identified by `UID` and `PwdWallet` connection attributes on the connection string. This is the preferred method as it is more secure. See Connect Using an Oracle Wallet with Credentials.

  > **Note**
  >
  > Most sections in this book provide security credentials for both cache administration users with an Oracle Wallet.

- You can provide the cache administration user name and passwords on the connection string. Specify the cache administration user name in the `UID` connection attribute. Specify the TimesTen cache administration user password in the `PWD` or `PWDCrypt` connection attribute and the Oracle cache administration user password in the `OraclePWD` connection attribute. See Connect Using Connection Attributes for Credentials.

Connect Using an Oracle Wallet with Credentials

You can provide credentials for cache administration users by saving them in an Oracle Wallet, which then can be used for connecting to both the TimesTen and Oracle databases.

Use the `ttUser -setPwd` command to store the TimesTen cache administration user and password in a wallet. Use the `ttUser -setOraclePwd` command to store the Oracle cache administration user and password in a wallet.

This section describes the process to add cache administration user passwords to an Oracle Wallet.

The following example shows how to use the `ttUser` utility to add both cache administration users to an Oracle Wallet in the `/wallets/cacheadminwallet` directory.

1. If it does not already exist, make a directory for your wallet. This example users `/wallets` as the directory for the wallet.

   ```
   % mkdir /wallets
   ```

2. Run the `ttUser -setPwd` command to store the TimesTen cache administration user credentials. The `ttUser` utility requires that you provide a subdirectory name that identifies the wallet (since you cannot change the name of an Oracle Wallet). This example provides `cacheadminwallet` as the subdirectory name for the wallet. If `cacheadminwallet` directory does not exist, then the `ttUser` utility creates the `cacheadminwallet` subdirectory and then creates the Oracle Wallet in the `/wallets/cacheadminwallet` directory. The `ttUser` utility prompts for the password for the TimesTen cache administration user `cacheadmin`, which is added to the wallet.

   ```
   % ttUser -setPwd -wallet /wallets/cacheadminwallet -uid cacheadmin
   Enter password:
   ```

3. Run the `ttUser -setOraclePwd` command to store the Oracle cache administration user credentials. The `ttUser` utility prompts for the password for the Oracle cache administration user `cacheadmin`, which is added to the wallet in `/wallets/cacheadminwallet`.

   ```
   % ttUser -setOraclePwd -wallet /wallets/cacheadminwallet -uid cacheadmin
   Enter password:
   ```

After the credentials are stored within an Oracle Wallet, provide the user name and location of the wallet on the connection string. The `PwdWallet` connection attribute identifies the wallet. The `UID` connection attribute identifies which credentials to locate within the wallet.

```
connect “dsn=cache1;uid=cacheadmin;PwdWallet=/wallets/cacheadminwallet”;
```

For client/server connections, the wallet must exist on the client.

See Providing the Cache Administration User Names and Passwords in an Oracle Wallet in the *Oracle TimesTen In-Memory Database Security Guide* for full details on how to store credentials in an Oracle Wallet. See PwdWallet and ttUser in the *Oracle TimesTen In-Memory Database Reference*.

Connect Using Connection Attributes for Credentials

You can provide credentials for cache administration users using connection attributes when connecting to the Oracle database.

In the connection string, specify the cache administration user name in the `UID` connection attribute. Specify the cache administration user credentials that are saved in the Oracle Wallet identified in the `PwdWallet` connection attribute.

```
% ttIsql "DSN=cache1;UID=cacheadmin;PwdWallet=/wallets/cacheadminwallet"
```

See Providing Both Cache Administration Users and Passwords in the *Oracle TimesTen In-Memory Database Security Guide* for more information on providing cache credentials.

Registering the Cache Administration User Name and Password

You must register the Oracle database cache administration user name and password internally in the TimesTen database before any cache group operation can be issued.

The TimesTen database and some TimesTen utilities and built-in procedures perform cache operations on your behalf. In order to connect to the Oracle database, TimesTen must have the credentials of the Oracle cache administration user and password registered internally.

The cache agent connects to the Oracle database as the Oracle cache administration user to create and maintain Oracle Database objects that store information used to enforce predefined behaviors of particular cache group types. In addition, both the cache and replication agents connect to the Oracle database with the internally registered Oracle cache administration user credentials to manage Oracle database operations.

The Oracle database cache administration user name and password need to be registered only once in each TimesTen database that caches Oracle Database data unless the cache administration user name or its password is changed. For example, if you modify the password of the cache administration user, if the TimesTen database is destroyed and re-created, or if the Oracle cache administration user name is dropped and re-created in the Oracle database, the Oracle cache administration user name and password must be registered again.

The Oracle cache administration user name cannot be changed if there are cache groups in the database. The cache groups must be dropped before you can drop and recreate the cache administration user. See Changing Cache User Names and Passwords.

Registering the Cache Administration User Name and Password in TimesTen

In TimesTen, you can register the Oracle cache administration user name and password by calling the `ttCacheUidPwdSet` built-in procedure after connecting as the Timesten cache administration user.

Before you register the Oracle cache administration user and password internally within the TimesTen database, you must decide if you want to save these credentials in an Oracle Wallet (recommended) or within memory (the default). To save the Oracle cache administration user credentials within an Oracle Wallet, ensure that the `CacheAdminWallet` connection attribute is set to 1 (best set in your DSN).

This example connects as the `cacheadmin` cache administration user providing credentials in a wallet. After connection, the example calls `ttCacheUidPwdSet` providing the Oracle cache administration user name and password, which registers the Oracle cache administration user name and password within TimesTen.

```
% ttIsql "DSN=cache1;UID=cacheadmin;PwdWallet=/wallets/cacheadminwallet"
Command> call ttCacheUidPwdSet('cacheadmin','orapwd');
```

The credentials can also be registered from a command line by running a `ttAdmin -cacheUidPwdSet` utility command as a TimesTen external user with the `CACHE_MANAGER` privilege. The `ttAdmin` utility prompts for the password.

```
% ttAdmin -cacheUidPwdSet -cacheUid cacheadmin orapwd cache1
Enter password:
```

See ttCacheUidPwdSet and ttAdmin in *Oracle TimesTen In-Memory Database Reference*.

Cache Group Requirements for Credentials

Because of the synchronous or asynchronous nature of some cache groups, TimesTen uses credentials set as follows:

- When you connect to the TimesTen database to work with AWT or read-only cache groups, TimesTen Classic uses the credentials that are registered with the `ttCacheUidPwdSet` built-in procedure when connecting to the Oracle database on behalf of these cache groups.
- When you connect to the TimesTen database to work with SWT or user managed cache groups or passthrough operations, TimesTen Classic connects to the Oracle database using the current user's credentials provided in the connection string. This can be either the credentials stored in a wallet designated by the `UID` and `PwdWallet` connection attributes or the `UID`, `PWD`, and `OraclePwd` connection attributes.
- When you are using dynamic load, the credentials used depend on if you are using connection pooling or not.
  - When `UseCacheConnPool=` 0 (the default), connection pooling is disabled. In this case, TimesTen Classic connects to the Oracle database using the current user's credentials provided in the connection string when performing a dynamic load request.
  - When `UseCacheConnPool=` 2, connection pooling is enabled. In this case, TimesTen Classic connects to the Oracle database using the credentials that have been registered with the `ttCacheUidPwdSet` built-in procedure when performing a dynamic load request.

Testing the Connectivity Between the TimesTen and Oracle Databases

If connectivity has been successfully established, the query returns the version of the Oracle database.

To test the connectivity between the TimesTen and Oracle databases, set the passthrough level to 3 and run the following query, to be processed on the Oracle database, as the TimesTen cache administration user:

```
Command> passthrough 3;
Command> SELECT * FROM V$VERSION;
Command> passthrough 0;
```

If it does not, check the following for correctness:

- The Oracle net service name set in the `OracleNetServiceName` connection attribute and the state of the Oracle database server
- The settings of the shared library search path environment variable such as `LD_LIBRARY_PATH` or `SHLIB_PATH`
- The setting of the cache administration user name in the TimesTen database

You can retrieve the Oracle cache administration user name setting by calling the `ttCacheUidGet` built-in procedure as the TimesTen cache administration user:

```
Command> call ttCacheUidGet;
```

In TimesTen, the Oracle cache administration user name can also be returned from a command line by running a `ttAdmin -cacheUidGet` utility command as a TimesTen external user with the `CACHE_MANAGER` privilege:

```
% ttAdmin -cacheUidGet cache1
```

Managing the Cache Agent

The cache agent process performs cache operations (such as autorefresh and loading a cache group), as well as manages Oracle Database objects used to enforce the predefined behaviors of particular cache group types.

You can check the status of the cache agent. See #unique_62.

- Starting the Cache Agent
- Stopping the Cache Agent
- Set a Cache Agent Start Policy in TimesTen

Starting the Cache Agent

The cache agent is a TimesTen daemon process that manages many of the cache-related functions for a TimesTen database. You can manually start the cache agent.

To start the cache agent, in TimesTen, call the `ttCacheStart` built-in procedure as the TimesTen cache administration user:

```
Command> call ttCacheStart;
```

You can also start the cache agent from a command line by running a `ttAdmin -cacheStart` utility command as a TimesTen external user with the `CACHE_MANAGER` privilege:

```
% ttAdmin -cacheStart cache1
```

Stopping the Cache Agent

You can manually stop the cache agent.

To stop the cache agent, in TimesTen, call the `ttCacheStop` built-in procedure as the TimesTen cache administration user:

```
Command> call ttCacheStop;
```

You can also stop the cache agent from a command line by running a `ttAdmin -cacheStop` utility command as a TimesTen external user with the `CACHE_MANAGER` privilege:

```
% ttAdmin -cacheStop cache1
```

Do not stop the cache agent immediately after you have dropped or altered a cache group with autorefresh. Instead, wait for at least two minutes to allow the cache agent to clean up Oracle Database objects such as change log tables and triggers that were created and used to manage the cache group.

The `ttCacheStop` built-in procedure has an optional parameter and the `ttAdmin -cacheStop` utility command has an option `-stopTimeout` that specifies how long the TimesTen main daemon process waits for the cache agent to stop. If the cache agent does not stop within the specified timeout period, the TimesTen daemon stops the cache agent. The default cache agent stop timeout is 100 seconds. A value of 0 specifies to wait indefinitely.

Note

The TimesTen X/Open XA and Java Transaction API (JTA) implementations do not work with cache. The start of any XA or JTA transaction fails if the cache agent is running.

Set a Cache Agent Start Policy in TimesTen

A cache agent start policy determines how and when the cache agent process starts on a TimesTen database.

The cache agent start policy can be set to:

- `manual`
- `always`
- `norestart`

The default start policy is `manual`, which means the cache agent must be started manually by calling the `ttCacheStart` built-in procedure or running a `ttAdmin -cacheStart` utility command. To manually stop a running cache agent process, call the `ttCacheStop` built-in procedure or run a `ttAdmin -cacheStop` utility command.

When the start policy is set to `always`, the cache agent starts automatically when the TimesTen main daemon process starts. With the `always` start policy, the cache agent cannot be stopped when the main daemon is running unless the start policy is first changed to either `manual` or `norestart`. Then issue a manual stop by calling the `ttCacheStop` built-in procedure or running a `ttAdmin -cacheStop` utility command.

With the `manual` and `always` start policies, the cache agent automatically restarts when the database recovers after a failure such as a database invalidation.

Setting the cache agent start policy to `norestart` means the cache agent must be started manually by calling the `ttCacheStart` built-in procedure or running a `ttAdmin -cacheStart` utility command, and stopped manually by calling the `ttCacheStop` built-in procedure or running a `ttAdmin -cacheStop` utility command.

With the `norestart` start policy, the cache agent does not automatically restart when the database recovers after a failure such as a database invalidation. You must restart the cache agent manually by calling the `ttCacheStart` built-in procedure or running a `ttAdmin -cacheStart` utility command.

Note

See ttAdmin, ttCachePolicySet, ttCacheStart and ttCacheStop in the *Oracle TimesTen In-Memory Database Reference*.

You can set the cache agent start policy in TimesTen by calling the `ttCachePolicySet` built-in procedure as the TimesTen cache administration user:

```
Command> call ttCachePolicySet('always');
```

It can also be set from a command line by running a `ttAdmin -cachePolicy` utility command as a TimesTen external user with the `CACHE_MANAGER` privilege:

```
% ttAdmin -cachePolicy norestart cache1
```

4

Defining Cache Groups

There are several different types of cache groups. There are reasons for when to use each type of cache group for different purposes, performance and availability needs. In addition, there are different features that you can add to provide certain functionality to a specific cache group type.

Cache Groups and Cache Tables

A cache group defines the Oracle Database data to cache in the TimesTen database. When you create a cache group, cache tables are created in the TimesTen database that correspond to the Oracle Database tables being cached.

A separate table definition must be specified in the cache group definition for each Oracle Database table that is being cached. The owner, table name, and cached column names of a TimesTen cache table must match the schema owner, table name, and column names of the corresponding cached Oracle Database table. The cache table can contain all or a subset of the columns and rows of the cached Oracle Database table. Each TimesTen cache table must have a primary key.

An Oracle Database table cannot be cached in more than one cache group within the same TimesTen database. However, the table can be cached in separate cache groups in different TimesTen databases.

If a table is cached in separate AWT cache groups and the same cache instance is updated simultaneously on multiple TimesTen databases, there is no guarantee as to the order in which the updates are propagated to the cached Oracle Database table. In this case, the contents of the updated cache table may be inconsistent between the TimesTen databases.

Before you define the cache group table, create the Oracle Database tables that are to be cached. Each table should be either:

- An Oracle Database table with a primary key on non-nullable columns. The TimesTen cache table primary key must be defined on the full Oracle Database table primary key. For example, if the cached Oracle Database table has a composite primary key on columns c1, c2 and c3, the TimesTen cache table must also have a composite primary key on columns c1, c2 and c3.

  The following example shows how to create a cache group from an Oracle Database table with a composite primary key. The following job_history table was created with a composite key on the Oracle database:

  ```
  CREATE TABLE job_history
      (employee_id NUMBER(6) NOT NULL,
      start_date DATE NOT NULL,
      end_date DATE NOT NULL,
      job_id VARCHAR2(10) NOT NULL,
      department_id NUMBER(4),
      PRIMARY KEY(employee_id, start_date));
  Table created.
  ```

  Create the cache group on the TimesTen database with all columns of the composite primary key:

  ```
  CREATE WRITETHROUGH CACHE GROUP job_hist_cg
          FROM sales.job_history
          (employee_id NUMBER(6) NOT NULL,
          start_date DATE NOT NULL,
          end_date DATE NOT NULL,
          job_id VARCHAR2(10) NOT NULL,
          department_id NUMBER(4),
          PRIMARY KEY(employee_id, start_date));
  ```

- An Oracle Database table with non-nullable columns upon which a unique index is defined on one or more of the non-nullable columns in the table. The TimesTen cache table primary key must be defined on all of the columns in the unique index. For example, if the unique index for the Oracle Database table is made up of multiple columns c1, c2, and c3, the TimesTen cache table must have a composite primary key on columns c1, c2, and c3.

  The following examples show how Oracle Database unique indexes were defined on tables with non-nullable columns.

  ```
  SQL> CREATE TABLE regions(
        region_id NUMBER NOT NULL,
        region_name VARCHAR2(25));
  Table created.
  SQL> CREATE UNIQUE INDEX region_idx
        ON regions(region_id);
  Index created.

  SQL> CREATE TABLE products(
        prod_id INT NOT NULL,
        cust_id INT NOT NULL,
        quantity_sold INT NOT NULL,
        time_id DATE NOT NULL);
  Table created.
  SQL> CREATE UNIQUE INDEX products_index ON products(prod_id, cust_id);
  Index created.
  ```

Based on these Oracle Database tables and unique indexes, you can create cache groups on a TimesTen database for these tables using the unique index columns as the primary key definition as shown below:

```
Command> CREATE WRITETHROUGH CACHE GROUP region_cg
  FROM sales.regions
  (region_id NUMBER NOT NULL PRIMARY KEY,
   region_name VARCHAR2(25));

Command> CREATE WRITETHROUGH CACHE GROUP products_cg
  FROM sales.products
  (prod_id INT NOT NULL, cust_id INT NOT NULL,
   quantity_sold INT NOT NULL, time_id DATE NOT NULL,
   PRIMARY KEY(prod_id, cust_id));
```

A TimesTen database can contain multiple cache groups. A cache group can contain one or more cache tables.

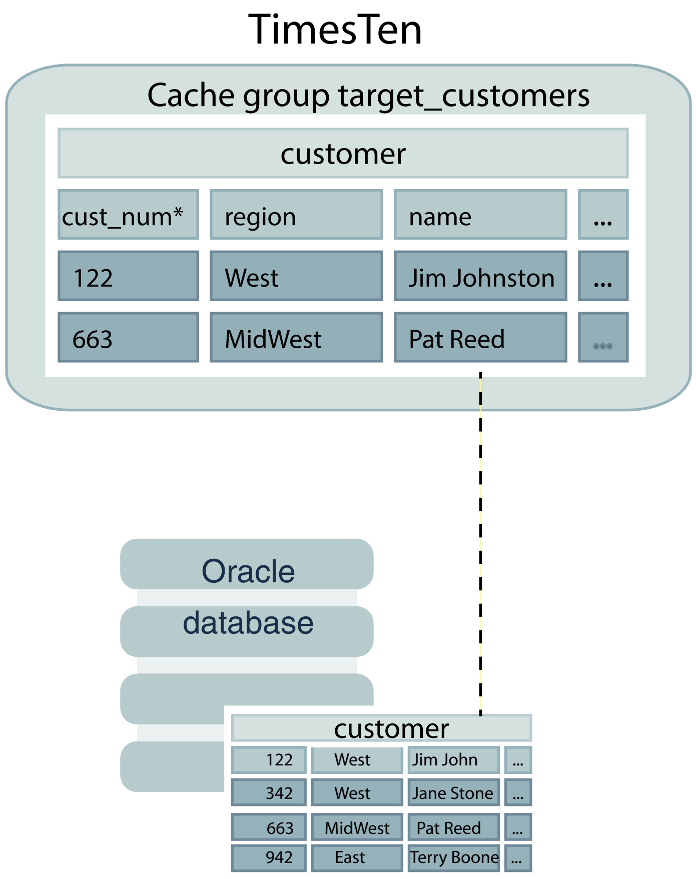
Creating indexes on a cache table in TimesTen can help speed up particular queries issued on the table in the same fashion as on a TimesTen regular table. You can create non-unique indexes on a TimesTen cache table. Do not create unique indexes on a cache table that do not match any unique index on the cached Oracle Database table. Otherwise, it can cause unique constraint failures in the cache table that do not occur in the cached Oracle Database table, and result in these tables in the two databases being no longer synchronized with each other when autorefresh operations are performed.

Single-Table Cache Group

The simplest cache group is one that caches a single Oracle Database table. In a single-table cache group, there is a root table but no child tables.

Figure 4-1 shows a single-table cache group `target_customers` that caches the `customer` table.

Figure 4-1   Cache Group with a Single Table



Multiple-Table Cache Group

A multiple-table cache group is one that defines a root table and one or more child tables.

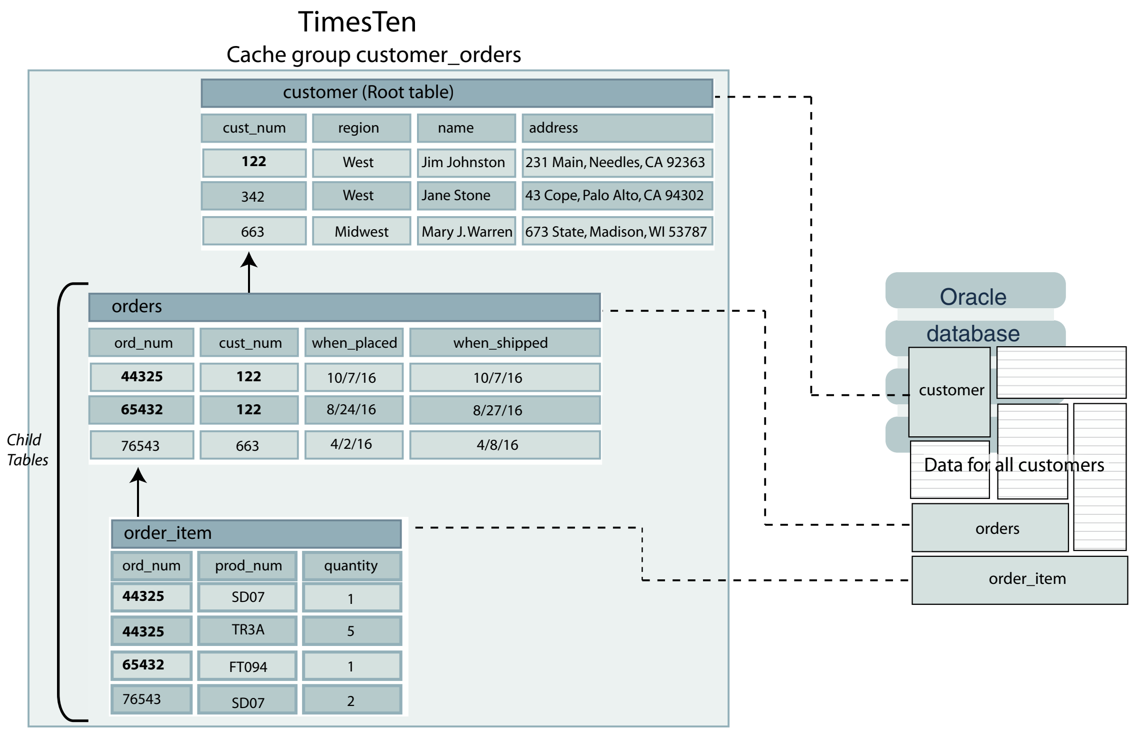
A cache group can only contain one root table. The root table does not reference any table with a foreign key constraint.

In a cache group with multiple cache tables on TimesTen, each child table must reference the primary key or a unique index of the root table or of another child table in the same cache group using a foreign key constraint. Cache tables defined in a multiple-table cache group must be related to each other in TimesTen through foreign key constraints. However, the corresponding tables in the Oracle database do not necessarily need to be related to each other. The tables on the Oracle database can be related:

- Related through a foreign key constraint.
- Related without a foreign key constraint. You may have tables on the Oracle database that are not related through a foreign key constraint. However, you want to cache the data within these separate tables on TimesTen. The user application could maintain a relationship between tables that is not enforced by foreign key constraints on the Oracle database.

Figure 4-2 shows a multiple-table cache group `customer_orders` that caches the `customer`, `orders` and `order_item` tables. Each parent table in the `customer_orders` cache group has a primary key that is referenced by a child table through a foreign key constraint. The `customer` table is the root table of the cache group because it does not reference any table in the cache group with a foreign key constraint. The primary key of the root table is considered the primary key of the cache group. The `orders` table is a child table of the customer root table. The `order_item` table is a child table of the `orders` child table.

Figure 4-2 Cache Group with Multiple Tables



cust_num	region	name	address
122	West	Jim Johnston	231 Main, Needles, CA 92363
342	West	Jane Stone	43 Cope, Palo Alto, CA 94302
663	Midwest	Mary J. Warren	673 State, Madison, WI 53787

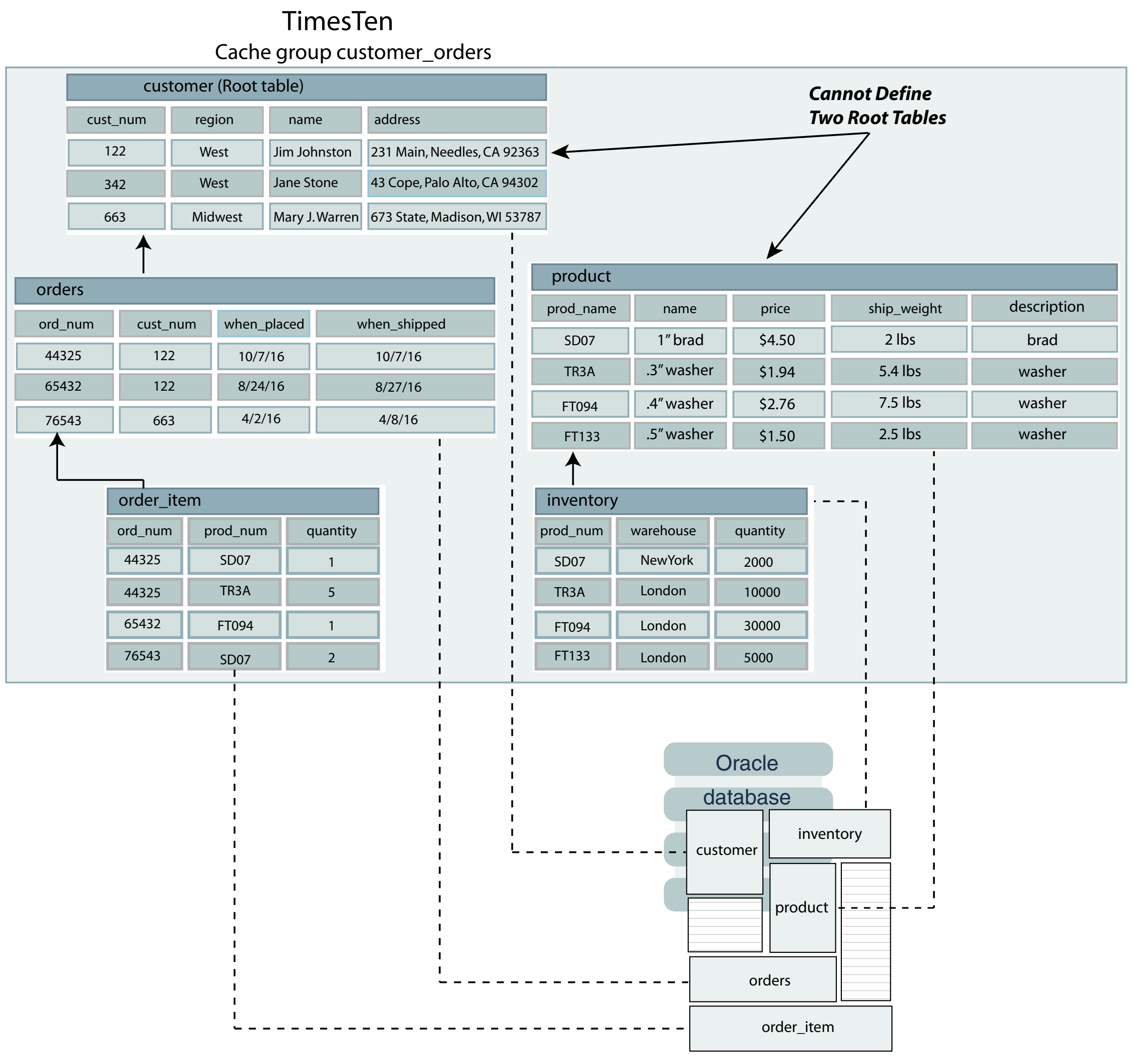
ord_num	cust_num	when_placed	when_shipped
44325	**122**	10/7/16	10/7/16
65432	**122**	8/24/16	8/27/16
76543	663	4/2/16	4/8/16

ord_num	prod_num	quantity
44325	SD07	1
44325	TR3A	5
65432	FT094	1
76543	SD07	2

The table hierarchy in a multiple-table cache group can designate child tables to be parents of other child tables. A child table cannot reference more than one parent table. However, a parent table can be referenced by more than one child table.

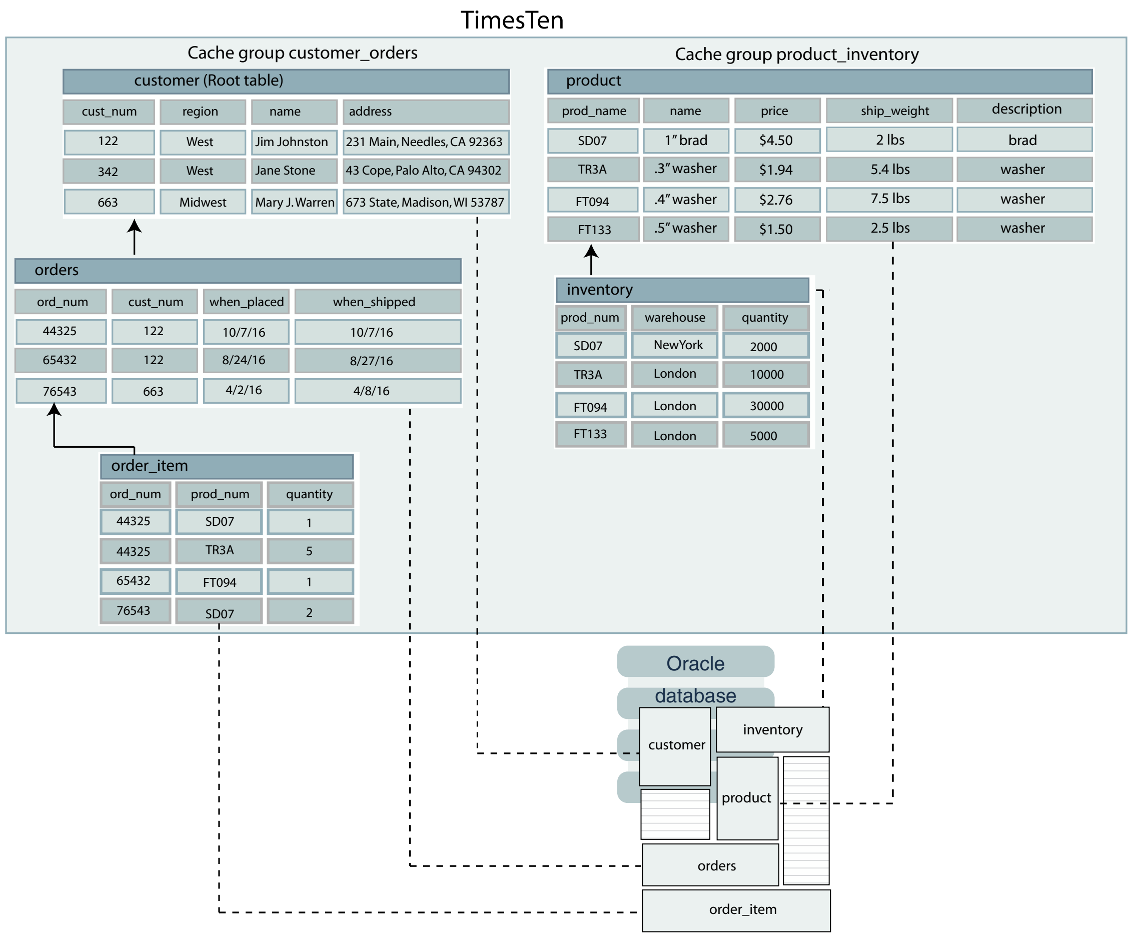
Figure 4-3 shows an improper cache table hierarchy. Neither the customer nor the product table references a table in the cache group with a foreign key constraint. This results in the cache group having two root tables which is invalid.

Figure 4-3 Problem: Cache Group Contains Two Root Tables



To resolve this problem and cache all the tables, create a cache group which contains the `customer`, `orders`, and `order_item` tables, and a second cache group which contains the `product` and the `inventory` tables as shown in Figure 4-4.

Figure 4-4 Solution: Create Two Cache Groups


TimesTen
Cache group customer_orders
customer (Root table)
cust_num | region | name | address
122 | West | Jim Johnston | 231 Main, Needles, CA 92363
342 | West | Jane Stone | 43 Cope, Palo Alto, CA 94302
663 | Midwest | Mary J. Warren | 673 State, Madison, WI 53787
orders
ord_num | cust_num | when_placed | when_shipped
44325 | 122 | 10/7/16 | 10/7/16
65432 | 122 | 8/24/16 | 8/27/16
76543 | 663 | 4/2/16 | 4/8/16
order_item
ord_num | prod_num | quantity
44325 | SD07 | 1
44325 | TR3A | 5
65432 | FT094 | 1
76543 | SD07 | 2
Cache group product_inventory
product
prod_name | name | price | ship_weight | description
SD07 | 1" brad | $4.50 | 2 lbs | brad
TR3A | .3" washer | $1.94 | 5.4 lbs | washer
FT094 | .4" washer | $2.76 | 7.5 lbs | washer
FT133 | .5" washer | $1.50 | 2.5 lbs | washer
inventory
prod_num | warehouse | quantity
SD07 | NewYork | 2000
TR3A | London | 10000
FT094 | London | 30000
FT133 | London | 5000
Oracle database
customer
inventory
product
orders
order_item

Creating a Cache Group

You create cache groups by using a `CREATE CACHE GROUP` SQL statement or by using Oracle SQL Developer, a graphical tool.

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

Cache groups must be created by and are owned by the TimesTen cache administration user.

You cannot cache Oracle Database data in a temporary database.

Cache groups are identified as either system managed or user managed. System managed cache groups enforce specific behaviors, while the behavior of a user managed cache group can be customized.

System managed cache groups include:

- Read-Only Cache Group: Committed updates on the cached Oracle Database tables are automatically refreshed to the cache tables on TimesTen. The TimesTen cache tables cannot be updated directly.

- Asynchronous WriteThrough (AWT) Cache Group: Committed updates on the TimesTen cache tables are automatically and asynchronously propagated to the cached Oracle Database tables.
- Synchronous WriteThrough (SWT) Cache Group: Committed updates on the TimesTen cache tables are automatically and synchronously propagated to the cached Oracle Database tables.
- Hybrid Cache Group: Dynamically load committed updates from cache tables that do not have a root table on the Oracle database.

User Managed Cache Group: Customize caching behavior. If the system managed cache groups do not satisfy your application's requirements, you can create a user-managed cache group that defines customized caching behavior with cache table attributes.

You can define how data is loaded:

- Static cache group: Cache instances are loaded manually into the TimesTen cache tables.
- Dynamic cache group: Cache instances are loaded into the TimesTen cache tables on demand from an Oracle database using a dynamic load operation or manually using a load operation.

See Transmitting Changes Between the TimesTen and Oracle Databases.

The following topics also apply to creating a cache group:

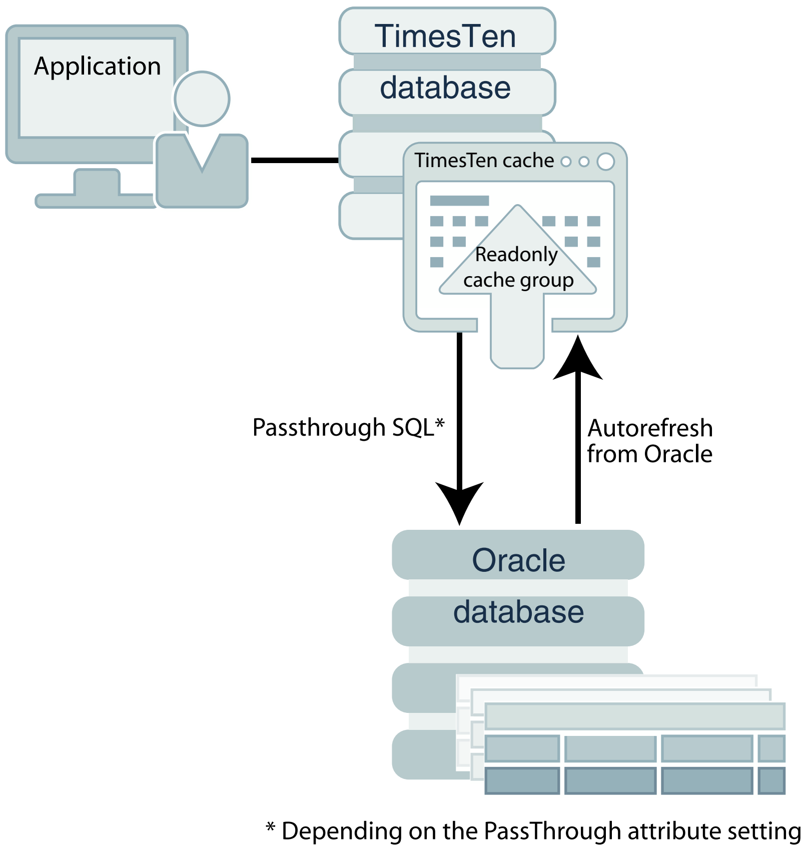
- Creating a Dynamic Cache Group with the DYNAMIC Keyword that enables dynamic load of new cache instances updated on cached Oracle database tables into TimesTen cache groups.
- Automatically Refreshing a Cache Group: The `AUTOREFRESH` cache table attribute specifies that committed changes on cached Oracle Database tables are automatically refreshed to read-only TimesTen cache tables.
- Using a WHERE Clause: You can restrict the rows to cache in the TimesTen database for particular cache group types.
- ON DELETE CASCADE Cache Table Attribute: Specifies that when rows containing referenced key values are deleted from a parent table, rows in child tables with dependent foreign keys are also deleted.
- Creating a Hash Index on the Primary Key Columns of the Cache Table: Specifies that a hash index rather than a range index is created on the primary key columns of the cache table.

Read-Only Cache Group

A read-only cache group enforces a caching behavior where the TimesTen cache tables cannot be updated directly, and committed changes on the cached Oracle Database tables are automatically refreshed to the cache tables.

See Figure 4-5.

Figure 4-5   Read-Only Cache Group



If the TimesTen database is unavailable for whatever reason, you can still update the Oracle Database tables that are cached in a read-only cache group. When the TimesTen database returns to operation, updates that were committed on the cached Oracle Database tables while the TimesTen database was unavailable are automatically refreshed to the TimesTen cache tables.

TimesTen supports both static and dynamic read-only cache groups.

> **Note**
>
> When TimesTen manages operations for read only cache groups, it connects to the Oracle database using the Oracle cache administration user name and password. For more details, see Registering the Cache Administration User Name and Password.

On the Oracle Database:

The following is an example of a definition of the Oracle Database tables that are to be cached in read-only cache groups. The Oracle Database tables are owned by the schema user `sales`.

```
CREATE TABLE customer
(cust_num NUMBER(6) NOT NULL PRIMARY KEY,
 region   VARCHAR2(10),
 name     VARCHAR2(50),
 address  VARCHAR2(100));
```

```
CREATE TABLE orders
(ord_num      NUMBER(10) NOT NULL PRIMARY KEY,
 cust_num     NUMBER(6) NOT NULL,
 order_det    JSON,
 when_placed  DATE NOT NULL,
 when_shipped DATE NOT NULL);
```

For cached tables that are going to be included in read-only cache groups, the Oracle cache administration user must be granted the SELECT privilege on these cached tables. In this example, these tables are sales.customer and sales.orders tables.

See Required Privileges for Cache Administration User for Cache Operations for all required privileges for different activities.

On the Oracle database, connect as an administrator and grant the following privileges:

```
SQL> GRANT SELECT ON sales.customer TO cacheadmin;
SQL> GRANT SELECT ON sales.orders TO cacheadmin;
```

On the TimesTen database:

Connect as the TimesTen cache administration user. Use the CREATE READONLY CACHE GROUP statement to create a read-only cache group.

The following statement creates a read-only cache group customer_orders that caches the tables sales.customer (root table) and sales.orders (child table):

```
CREATE READONLY CACHE GROUP customer_orders
FROM sales.customer
 (cust_num NUMBER(6) NOT NULL,
  region   VARCHAR2(10),
  name     VARCHAR2(50),
  address  VARCHAR2(100),
  PRIMARY KEY(cust_num)),
sales.orders
 (ord_num      NUMBER(10) NOT NULL,
  cust_num     NUMBER(6) NOT NULL,
  order_det    JSON,
  when_placed  DATE NOT NULL,
  when_shipped DATE NOT NULL,
  PRIMARY KEY(ord_num),
  FOREIGN KEY(cust_num) REFERENCES sales.customer(cust_num));
```

By default, all read-only cache groups are defined with incremental autorefresh paused with the default interval value. Perform a LOAD CACHE GROUP statement for the first load of the read-only cache group since the cache tables are empty. The autorefresh state changes from PAUSED to ON after the LOAD CACHE GROUP statement completes.

The cache tables in a read-only cache group cannot be updated directly. However, you can set the passthrough level to 2 to allow committed update operations issued on a TimesTen cache table to be passed through and processed on the cached Oracle Database table, and then have the updates be automatically refreshed into the cache table. See Setting a Passthrough Level.

When you're working with trigger-based cache groups in TimesTen, it's important to understand that the effects of a passed through statement (such as an update, insert, or delete) on the cache tables do not occur within the same transaction in which the update operation was issued on the backend database. Instead, they are seen after the passed through update operation has been committed on the Oracle database and the next automatic refresh of the cache group has occurred. The Oracle cache administration user must be granted the INSERT, UPDATE and DELETE privileges on the Oracle database tables that are cached in the read-only cache group in order for the passed through update operations to be processed on the cached Oracle database tables.

Log-based cache groups in Oracle TimesTen, are primarily designed for both static and dynamic cache groups, where data is continuously refreshed and synchronized from Oracle using redo logs. It enables automatic refresh and asynchronous data propagation from Oracle to TimesTen by reading changes from the Oracle logs. See Configuring GoldenGate for Log-Based Cache Autorefresh on TimesTen.

If you manually created the Oracle database objects used to enforce the predefined behaviors of a cache group with autorefresh as described in The initCacheAdminSchema.sql Script, you need to set the autorefresh state to `OFF` when creating the cache group.

Then you need to run the `ttIsql` utility's `cachesqlget` command to generate a SQL*Plus script used to create a log table and a trigger in the Oracle database for each Oracle Database table that is cached in the read-only cache group. See Manually Creating Oracle Database Objects for Cache Groups with Autorefresh for how to create these objects.

Restrictions with Read-Only Cache Groups

Certain restrictions apply to read-only cache groups.

The following restrictions apply when using a read-only cache group:

- The cache tables on TimesTen cannot be updated directly.
- Only the `ON DELETE CASCADE` and `UNIQUE HASH ON` cache table attributes can be used in the cache table definitions.

  See ON DELETE CASCADE Cache Table Attribute.

  See Creating a Hash Index on the Primary Key Columns of the Cache Table.
- A `FLUSH CACHE GROUP` statement cannot be issued on the cache group.

  See Flushing a User Managed Cache Group.
- A `TRUNCATE TABLE` statement issued on a cached Oracle Database table is not automatically refreshed to the TimesTen cache table.
- A `LOAD CACHE GROUP` statement can only be issued on the cache group if the cache tables are empty, unless the cache group is dynamic.

  See Manually Loading and Refreshing a Cache Group.

  See Creating a Dynamic Cache Group with the DYNAMIC Keyword.
- The autorefresh state must be `PAUSED` before you can issue a `LOAD CACHE GROUP` statement on the cache group, unless the cache group is dynamic, in which case the autorefresh state must be `PAUSED` or `ON`. The `LOAD CACHE GROUP` statement cannot contain a `WHERE` clause, unless the cache group is dynamic, in which case the `WHERE` clause must be followed by a `COMMIT EVERY` *n* `ROWS` clause.

  See Automatically Refreshing a Cache Group.

  See Using a WHERE Clause.
- The autorefresh state must be `PAUSED` before you can issue a `REFRESH CACHE GROUP` statement on the cache group. The `REFRESH CACHE GROUP` statement cannot contain a `WHERE` clause.

  See Manually Loading and Refreshing a Cache Group.

- All tables and columns referenced in `WHERE` clauses when creating, loading or unloading the cache group must be fully qualified. For example:

  `owner``.`*`table_name`* and *`owner`*`.`*`table_name`*`.`*`column_name`*

- Least recently used (LRU) aging cannot be specified on the cache group, unless the cache group is dynamic where LRU aging is defined by default.

  See LRU Aging in TimesTen.

- Read-only cache groups cannot cache Oracle Database views or materialized views.

Asynchronous WriteThrough (AWT) Cache Group

An Asynchronous WriteThrough (AWT) cache group enforces a caching behavior where committed changes on the TimesTen cache tables are automatically and asynchronously propagated to the cached Oracle Database tables.

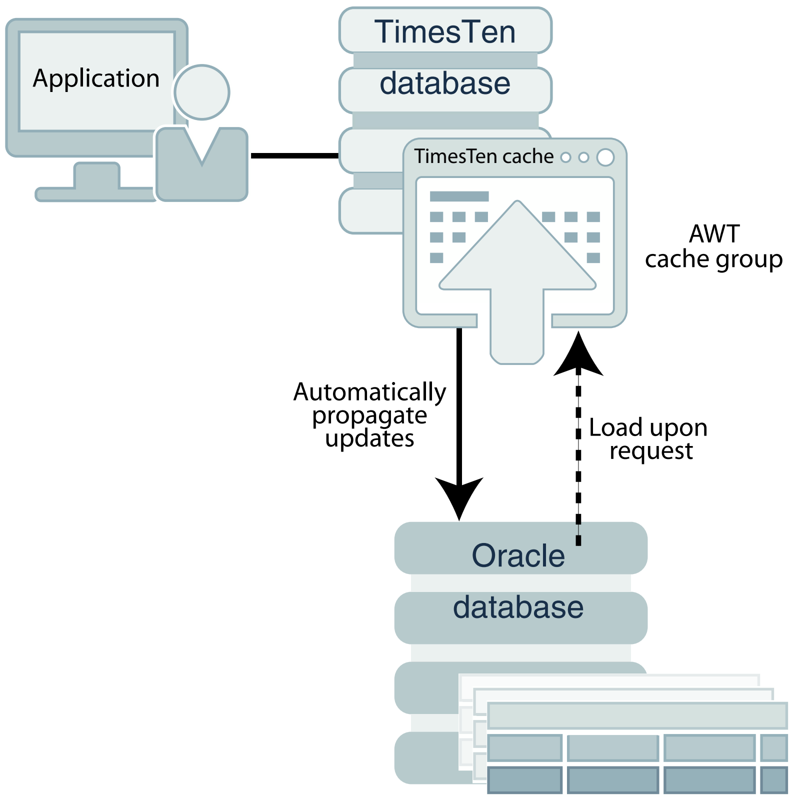
See Figure 4-6.

Only TimesTen supports AWT cache groups.

Note

You should avoid running DML statements on Oracle Database tables cached in an AWT cache group. This can result in an error condition. See Restrictions with AWT Cache Groups.

Figure 4-6 AWT Cache Group



The transaction commit on a TimesTen database occurs asynchronously from the commit on an Oracle database. This enables an application to continue issuing transactions on a TimesTen database without waiting for the Oracle Database transaction to complete. However, your application cannot ensure when the transactions are completed on an Oracle database.

You can update cache tables in an AWT cache group even if the Oracle database is unavailable. When the Oracle database returns to operation, updates that were committed on the cache tables while the Oracle database was unavailable are automatically propagated to the cached Oracle Database tables.

Note

When TimesTen manages operations for AWT cache groups, it connects to the Oracle database using the Oracle cache administration user name and password set with the `ttCacheUidPwdSet` built-in procedure. For more details on `ttCacheUidPwdSet`, see Registering the Cache Administration User Name and Password.

Since an AWT cache group propagates data from the TimesTen database to the Oracle database, any data modified by the user in the cached tables on the Oracle database is not automatically uploaded from the Oracle database to the TimesTen database. In this case, you must manually run a `REFRESH CACHE GROUP` SQL statement to have any changes done to the Oracle database transmitted to the TimesTen database.

Processing of any `REFRESH CACHE GROUP` or `UNLOAD CACHE GROUP` statement for an AWT cache group waits until updates on any of the rows modified on the TimesTen database have been propagated to the Oracle database.

On the Oracle Database:

The following is an example of a definition of the Oracle database table that is to be cached in an AWT cache group. The Oracle database table is owned by the schema user `sales`.

```
CREATE TABLE customer
(cust_num NUMBER(6) NOT NULL PRIMARY KEY,
 region   VARCHAR2(10),
 name     VARCHAR2(50),
 address  VARCHAR2(100));
```

When the cached tables are a part of an AWT cache group, then the Oracle cache administration user must be granted the `SELECT`, `INSERT`, `UPDATE`, and `DELETE` privileges on any cached tables. In this example, the table is the `sales.customer` table.

See Required Privileges for Cache Administration User for Cache Operations for all required privileges for different activities.

On the Oracle database as an administrator, grant the following privileges:

```
SQL> GRANT SELECT, INSERT, UPDATE, DELETE ON sales.customer TO cacheadmin;
```

On the TimesTen database:

Connect as the TimesTen cache administraiton user. Use the `CREATE ASYNCHRONOUS WRITETHROUGH CACHE GROUP` statement to create an AWT cache group.

The following statement creates an AWT cache group `new_customers` that caches the `sales.customer` table:

```
CREATE ASYNCHRONOUS WRITETHROUGH CACHE GROUP new_customers
FROM sales.customer
 (cust_num NUMBER(6) NOT NULL,
  region   VARCHAR2(10),
  name     VARCHAR2(50),
  address  VARCHAR2(100),
  PRIMARY KEY(cust_num));
```

The following sections describe configuration, behavior, and management for AWT cache groups:

- Starting and Stopping the Replication Agent
- Setting a Replication Agent Start Policy
- Monitoring Propagation of Transactions to the Oracle Database
- Disabling Propagation of Committed Changes
- Configuring Parallel Propagation to the Oracle Database
- What an AWT Cache Group Does and Does Not Guarantee
- Restrictions with AWT Cache Groups
- Reporting Oracle Database Permanent Errors for AWT Cache Groups

Starting and Stopping the Replication Agent

Performing asynchronous writethrough operations requires that the replication agent be running on the TimesTen database that contains AWT cache groups.

Running a `CREATE ASYNCHRONOUS WRITETHROUGH CACHE GROUP` statement creates a replication scheme that enables committed changes on the TimesTen cache tables to be asynchronously propagated to the cached Oracle Database tables.

After you have created AWT cache groups, start the replication agent on the TimesTen database by calling the `ttRepStart` built-in procedure as the cache administration user. This connects using an Oracle Wallet that contains the passwords for both cache administration users.

Connect as the TimesTen cache administration user and provide credentials in the Oracle Wallet.

```
% ttIsql "DSN=cache1;UID=cacheadmin;PwdWallet=/wallets/cacheadminwallet"
Command> call ttRepStart;
```

It can also be started from a command line by running a `ttAdmin -repStart` utility command as a TimesTen external user with the `CACHE_MANAGER` privilege:

```
% ttAdmin -repStart cache1
```

The replication agent does not start unless there is at least one AWT cache group or replication scheme in the TimesTen database.

If the replication agent is running, it must be stopped before you can issue another `CREATE ASYNCHRONOUS WRITETHROUGH CACHE GROUP` statement or a `DROP CACHE GROUP` statement on an AWT cache group.

You can stop the replication agent by calling the `ttRepStop` built-in procedure as the cache administration user:

```
Command> call ttRepStop;
```

You can also stop the replication agent from a command line with the `ttAdmin -repStop` utility command as a TimesTen external user with the `CACHE_MANAGER` privilege:

```
% ttAdmin -repStop cache1
```

Setting a Replication Agent Start Policy

Performing asynchronous writethrough operations requires that the replication agent be running on the TimesTen database that contains AWT cache groups. You can set a replication agent start policy to determine how and when the replication agent process starts on a TimesTen database.

The default start policy is `manual` which means the replication agent must be started manually by calling the `ttRepStart` built-in procedure or running a `ttAdmin -repStart` utility command. To manually stop a running replication agent process, call the `ttRepStop` built-in procedure or run a `ttAdmin -repStop` utility command.

The start policy can be set to `always` so that the replication agent starts automatically when the TimesTen main daemon process starts. With the `always` start policy, the replication agent cannot be stopped when the main daemon is running unless the start policy is changed to either `manual` or `norestart` and then a manual stop is issued by calling the `ttRepStop` built-in procedure or running a `ttAdmin -repStop` utility command.

With the `manual` and `always` start policies, the replication agent automatically restarts after a failure such as a database invalidation.

The start policy can be set to `norestart` which means the replication agent must be started manually by calling the `ttRepStart` built-in procedure or running a `ttAdmin -repStart` utility command, and stopped manually by calling the `ttRepStop` built-in procedure or running a `ttAdmin -repStop` utility command.

With the `norestart` start policy, the replication agent does not automatically restart after a failure such as a database invalidation. You must restart the replication agent manually by calling the `ttRepStart` built-in procedure or running a `ttAdmin -repStart` utility command.

Perform the following to set the replication agent start policy:

1. Before you set a replication agent start policy, grant the `ADMIN` privilege to the TimesTen cache administration user as the instance administrator.

   ```
   % ttIsql cache1
   Command> GRANT ADMIN TO cacheadmin;
   Command> exit
   ```

2. Set the replication agent start policy by calling the `ttRepPolicySet` built-in procedure as the TimesTen cache administration user:

   ```
   % ttIsql "DSN=cache1;UID=cacheadmin;PwdWallet=/wallets/cacheadminwallet"
   Command> call ttRepPolicySet('manual');
   Command> exit
   ```

   Alternately, set the replication agent start policy from a command line by running a `ttAdmin -repPolicy` utility command as a TimesTen external user with the `ADMIN` privilege:

   ```
   % ttAdmin -repPolicy always cache1
   ```

Monitoring Propagation of Transactions to the Oracle Database

Since the AWT cache group uses the replication agent to asynchronously propagate transactions to the Oracle database, these transactions remain in the transaction log buffer and transaction log files until the replication agent confirms they have been fully processed by the Oracle database.

You can monitor the propagation for these transactions with the `ttLogholds` built-in procedure.

When you call the `ttLogHolds` built-in procedure, the description field contains `_ORACLE` to identify the transaction log hold for the AWT cache group propagation.

```
Command> call ttLogHolds();
< 0, 18958336, Checkpoint                    , cache1.ds0 >
< 0, 19048448, Checkpoint                    , cache1.ds1 >
< 0, 19050904, Replication                   , ADC6160529:_ORACLE >
3 rows found.
```

See the Show Replicated Log Records section in the *Oracle TimesTen In-Memory Database Replication Guide*.

You can also improve performance by configuring parallel propagation to the Oracle Database. See Improving AWT Throughput with Parallel Propagation to the Oracle Database.

Disabling Propagation of Committed Changes

If there are updates from DML statements that you do not want propagated to the Oracle database, then you can disable propagation of committed changes (as a result of running DML statements) within the current transaction to the Oracle database by setting the flag in the `ttCachePropagateFlagSet` built-in procedure to zero.

After the flag is set to zero, the effects of running any DML statements are never propagated to the back-end Oracle database. Thus, these updates exist only on the TimesTen database. You can then re-enable propagation by resetting the flag to one with the `ttCachePropagateFlagSet` built-in procedure. After the flag is set back to one, propagation of all committed changes to the Oracle database resumes. The propagation flag automatically resets to one after the transaction is committed or rolled back. See ttCachePropagateFlagSet in the *Oracle TimesTen In-Memory Database Reference*.

Configuring Parallel Propagation to the Oracle Database

To improve throughput for an AWT cache group, you can configure multiple threads that act in parallel to propagate and apply transactional changes to the Oracle database.

Parallel propagation enforces transactional dependencies and applies changes in AWT cache tables to Oracle Database tables in commit order. See Improving AWT Throughput with Parallel Propagation to the Oracle Database.

What an AWT Cache Group Does and Does Not Guarantee

An AWT cache group comes with some guarantees.

An AWT cache group *can* guarantee that:

- No transactions are lost because of communication failures between the TimesTen and Oracle databases.
- If the replication agent is not running or loses its connection to the Oracle database, automatic propagation of committed changes on the TimesTen cache tables to the cached Oracle Database tables resumes after the agent restarts or reconnects to the Oracle database.

- Transactions are committed in the Oracle database in the same order they were committed in the TimesTen database.

An AWT cache group *cannot* guarantee that:

- All transactions committed successfully in the TimesTen database are successfully propagated to and committed in the Oracle database. Execution errors on the Oracle database cause the transaction in the Oracle database to be rolled back. For example, an update on the Oracle database may fail because of a unique constraint violation. Transactions that contain execution errors are not retried.

  Execution errors are considered permanent errors and are reported to the `*TimesTenDatabaseFileName*.awterrs` file that resides in the same directory as the TimesTen database's checkpoint files. See Reporting Oracle Database Permanent Errors for AWT Cache Groups.
- The absolute order of Oracle Database updates is preserved because TimesTen does not resolve update conflicts. The following are some examples:
  - In two separate TimesTen databases (`DB1` and `DB2`), different AWT cache groups cache the same Oracle Database table. An update is committed on the cache table in `DB1`. An update is then committed on the cache table in `DB2`. The two cache tables reside in different TimesTen databases and cache the same Oracle Database table. Because the writethrough operations are asynchronous, the update from `DB2` may get propagated to the Oracle database before the update from `DB1`, resulting in the update from `DB1` overwriting the update from `DB2`.
  - An update is committed on a cache table in an AWT cache group. The same update is committed on the cached Oracle Database table using a passthrough operation. The cache table update, which is automatically and asynchronously propagated to the Oracle database, may overwrite the passed through update that was processed directly on the cached Oracle Database table depending on when the propagated update and the passed through update is processed on the Oracle database. For this and other potential error conditions, TimesTen recommends that you do not run DML statements directly against Oracle Database tables cached in an AWT cache group. For more information, see Restrictions with AWT Cache Groups.

Restrictions with AWT Cache Groups

Certain restrictions apply when using an AWT cache group.

The following restrictions apply when using an AWT cache group:

- Only the `ON DELETE CASCADE` and `UNIQUE HASH ON` cache table attributes can be used in the cache table definitions.

  See ON DELETE CASCADE Cache Table Attribute.

  See Creating a Hash Index on the Primary Key Columns of the Cache Table.
- A `FLUSH CACHE GROUP` statement cannot be issued on the cache group.

  See Flushing a User Managed Cache Group.
- The cache table definitions cannot contain a `WHERE` clause.

  See Using a WHERE Clause.
- A `TRUNCATE TABLE` statement cannot be issued on the cache tables.
- AWT cache groups cannot cache Oracle Database views or materialized views.
- The replication agent must be stopped before creating or dropping an AWT cache group.

See Starting and Stopping the Replication Agent.

- Committed changes on the TimesTen cache tables are not propagated to the cached Oracle Database tables unless the replication agent is running.
- To create an AWT cache group, the length of the absolute path name of the TimesTen database cannot exceed 248 characters.
- You should avoid running DML statements on Oracle Database tables cached in an AWT cache group. This could result in an error condition. Any insert, update, or delete operation on the cached Oracle Database table can negatively affect the operations performed on TimesTen for the affected rows. TimesTen does not detect or resolve update conflicts that occur on the Oracle database. Committed changes made directly on a cached Oracle Database table may be overwritten by a committed update made on the TimesTen cache table when the cache table update is propagated to the Oracle database. In addition, deleting rows on the cached Oracle Database table could cause an empty update if TimesTen tries to update a row that no longer exists.

  To ensure that not all data is restricted from DML statements on Oracle Database, you can partition the data on Oracle Database to separate the data that is to be included in the AWT cache group from the data to be excluded from the AWT cache group.
- TimesTen performs deferred checking when determining whether a single SQL statement causes a constraint violation with a unique index.

  For example, suppose there is a unique index on a cached Oracle Database table's `NUMBER` column, and a unique index on the same `NUMBER` column on the TimesTen cache table. There are five rows in the cached Oracle Database table and the same five rows in the cache table. The values in the `NUMBER` column range from 1 to 5.

  An `UPDATE` statement is issued on the cache table to increment the value in the `NUMBER` column by 1 for all rows. The operation succeeds on the cache table but fails when it is propagated to the cached Oracle Database table.

  This occurs because TimesTen performs the unique index constraint check at the end of the statement's processing after all the rows have been updated. The Oracle database, however, performs the constraint check each time after a row has been updated.

  Therefore, when the row in the cache table with value 1 in the `NUMBER` column is changed to 2 and the update is propagated to the Oracle database, it causes a unique constraint violation with the row that has the value 2 in the `NUMBER` column of the cached Oracle Database table.

Reporting Oracle Database Permanent Errors for AWT Cache Groups

If transactions are not successfully propagated to and committed in the Oracle database, then the permanent errors cause the transaction in the Oracle database to be rolled back.

For example, an update on the Oracle database may fail because of a unique constraint violation. Transactions that contain permanent errors are not retried.

Permanent errors are always reported to the *`TimesTenDatabaseFileName`*`.awterrs` text file that resides in the same directory as the TimesTen database checkpoint files. See Oracle Database Errors Reported by TimesTen for AWT in the *Oracle TimesTen In-Memory Database Monitoring and Troubleshooting Guide* for information about the contents of this file.

You can configure TimesTen to report these errors in both ASCII and XML formats with the `ttCacheConfig` built-in procedure.

Note

Do not pass in any values to the *`tblOwner`* and *`tblName`* parameters for `ttCacheConfig` as they are not applicable to setting the format for the errors file.

- To configure TimesTen to report permanent errors to only the *`TimesTenDatabaseFileName`*`.awterrs` text file, call the `ttCacheConfig` built-in procedure with the `ASCII` parameter. This is the default.

  ```
  Command> call ttCacheConfig('AwtErrorXmlOutput',,,'ASCII');
  ```

- To configure TimesTen to report permanent errors to both the *`TimesTenDatabaseFileName`*`.awterrs` text file as well as to an XML file named *`TimesTenDatabaseFileName`*`.awterrs.xml`, call the `ttCacheConfig` built-in procedure with the `XML` parameter.

  ```
  Command> call ttCacheConfig('AwtErrorXmlOutput',,,'XML');
  ```

Note

Before calling `ttCacheConfig` to direct permanent errors to the XML file, you must first stop the replication agent. Then, restart the replication agent after the built-in procedure completes.

See ttCacheConfig in the *Oracle TimesTen In-Memory Database Reference*.

When you configure error reporting to be reported in XML format, the following two files are generated when Oracle Database permanent errors occur:

- *`TimesTenDatabaseFileName`*`.awterrs.xml` contains the Oracle Database permanent error messages in XML format.
- *`TimesTenDatabaseFileName`*`.awterrs.dtd` is the file that contains the XML Document Type Definition (DTD), which is used when parsing the *`TimesTenDatabaseFileName`*`.awterrs.xml` file.

  The XML DTD, which is based on the XML 1.0 specification, is a set of markup declarations that describes the elements and structure of a valid XML file containing a log of errors. The XML file is encoded using UTF-8. The following are the elements for the XML format.

  Note

  For more information on reading and understanding XML Document Type Definitions, see `http://www.w3.org/TR/REC-xml/`.

  ```
  <!ELEMENT ttawterrorreport (awterrentry*) >
  <!ELEMENT awterrentry(header, (failedop)?, failedtxn) >
  <!ELEMENT header (time, datastore, oracleid, transmittingagent, errorstr,
   (ctn)?, (batchid)?, (depbatchid)?) >
  <!ELEMENT failedop (sql) >
  <!ELEMENT failedtxn ((sql)+) >
  <!ELEMENT time (hour, min, sec, year, month, day) >
  ```

```
<!ELEMENT hour (#PCDATA) >
<!ELEMENT min (#PCDATA) >
<!ELEMENT sec (#PCDATA) >
<!ELEMENT year (#PCDATA) >
<!ELEMENT month (#PCDATA) >
<!ELEMENT day (#PCDATA) >
<!ELEMENT datastore (#PCDATA) >
<!ELEMENT oracleid (#PCDATA) >
<!ELEMENT transmittingagent (transmitingname, pid, threadid) >
<!ELEMENT pid (#PCDATA) >
<!ELEMENT threadid (#PCDATA) >
<!ELEMENT transmittingname (#PCDATA) >
<!ELEMENT errorstr (#PCDATA) >
<!ELEMENT ctn (timestamp, seqnum) >
<!ELEMENT timestamp(#PCDATA) >
<!ELEMENT seqnum(#PCDATA) >
<!ELEMENT batchid(#PCDATA) >
<!ELEMENT depbatchid(#PCDATA) >
<!ELEMENT sql(#PCDATA) >
```

Synchronous WriteThrough (SWT) Cache Group

A synchronous writethrough (SWT) cache group enforces a caching behavior where committed changes on the TimesTen cache tables are automatically and synchronously propagated to the cached Oracle Database tables.

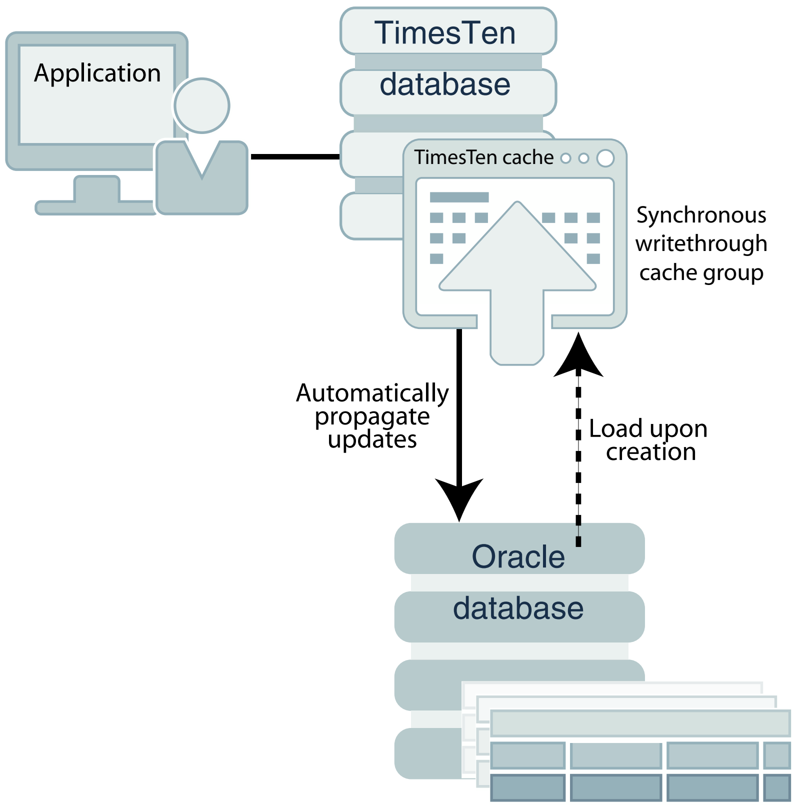
See Figure 4-7.

Only TimesTen supports SWT cache groups.

Note

You should avoid running DML statements on Oracle Database tables cached in an SWT cache group. This can result in an error condition. See Restrictions with SWT Cache Groups.

Figure 4-7 Synchronous WriteThrough Cache Group



The transaction commit on the TimesTen database occurs synchronously with the commit on the Oracle database. When an application commits a transaction in the TimesTen database, the transaction is processed in the Oracle database before it is processed in TimesTen. The application is blocked until the transaction has completed in both the Oracle and TimesTen databases.

If the transaction fails to commit in the Oracle database, the application must roll back the transaction in TimesTen. If the Oracle Database transaction commits successfully but the TimesTen transaction fails to commit, the cache tables in the SWT cache group are no longer synchronized with the cached Oracle Database tables.

Note

The behavior and error conditions for how commit occurs on both the TimesTen and Oracle databases when committing propagated updates is the same commit process on a user managed cache group with the `PROPAGATE` cache attribute that is described in PROPAGATE Cache Table Attribute.

To manually resynchronize the cache tables with the cached Oracle Database tables, call the `ttCachePropagateFlagSet` built-in procedure to disable update propagation, and then reissue the transaction in the TimesTen database after correcting the problem that caused the transaction commit to fail in TimesTen. Then, call the `ttCachePropagateFlagSet` built-in procedure to re-enable update propagation. You can also resynchronize the cache tables with the cached Oracle Database tables by reloading the accompanying cache groups.

The following is an example definition of the Oracle Database table that is to be cached in an example SWT cache group. The Oracle Database table is owned by the schema user `sales`.

```
CREATE TABLE product
(prod_num    VARCHAR2(6) NOT NULL PRIMARY KEY,
 name        VARCHAR2(30),
 price       NUMBER(8,2),
 ship_weight NUMBER(4,1));
```

The Oracle cache administration user, `cacheadmin`, must be granted certain privileges when creating a cache group. For SWT cache groups, the required privileges are `SELECT`, `INSERT`, `UPDATE`, and `DELETE` privileges on any cached tables. In this example, the table is the `sales.product` table.

See Required Privileges for Cache Administration User for Cache Operations for all required privileges for different activities.

On the Oracle database as an administrator, grant the following privileges:

```
SQL> GRANT SELECT, INSERT, UPDATE, DELETE ON sales.product TO cacheadmin;
```

On the TimesTen database:

Connect as the TimesTen cache administration user. Use the `CREATE SYNCHRONOUS WRITETHROUGH CACHE GROUP` statement to create an SWT cache group.

The following statement creates a synchronous writethrough cache group `top_products` that caches the `sales.product` table:

```
CREATE SYNCHRONOUS WRITETHROUGH CACHE GROUP top_products
FROM sales.product
 (prod_num    VARCHAR2(6) NOT NULL,
  name        VARCHAR2(30),
  price       NUMBER(8,2),
  ship_weight NUMBER(4,1),
  PRIMARY KEY(prod_num));
```

When TimesTen manages operations for SWT cache groups, it connects to the Oracle database using the current user's credentials provided on the connection string. The current user's credentials can be provided with an Oracle Wallet pointed to by the `PwdWallet` connection attribute or with the `UID`, `PWD`, and `OraclePwd` connection attributes. TimesTen does not connect to the Oracle database with the Oracle cache administration user name and password registered with the `ttCacheUidPwdSet` built-in procedure when managing SWT cache group operations. See Providing Cache Administration User Credentials When Connecting and Registering the Cache Administration User Name and Password.

Restrictions with SWT Cache Groups

There are certain restrictions when using an SWT cache group.

The following restrictions apply when using an SWT cache group:

- Only the `ON DELETE CASCADE` and `UNIQUE HASH ON` cache table attributes can be used in the cache table definitions.

  See ON DELETE CASCADE Cache Table Attribute for more information about the `ON DELETE CASCADE` cache table attribute.

  See Creating a Hash Index on the Primary Key Columns of the Cache Table for more information about the `UNIQUE HASH ON` cache table attribute.

- A `FLUSH CACHE GROUP` statement cannot be issued on the cache group.

  See Flushing a User Managed Cache Group for more information about the `FLUSH CACHE GROUP` statement
- The cache table definitions cannot contain a `WHERE` clause.

  See Using a WHERE Clause for more information about `WHERE` clauses in cache group definitions and operations.
- A `TRUNCATE TABLE` statement cannot be issued on the cache tables.
- SWT cache groups cannot cache Oracle Database views or materialized views.
- You should avoid running DML statements directly on Oracle Database tables cached in an SWT cache group. This could result in an error condition. Any insert, update, or delete operation on the cached Oracle Database table can negatively affect the operations performed on TimesTen for the affected rows. TimesTen does not detect or resolve update conflicts that occur on the Oracle database. Committed changes made directly on a cached Oracle Database table may be overwritten by a committed update made on the TimesTen cache table when the cache table update is propagated to the Oracle database. In addition, deleting rows on the cached Oracle Database table could cause an empty update if TimesTen tries to update a row that no longer exists.

  To ensure that not all data is restricted from DML statements on Oracle Database, you can partition the data on Oracle Database to separate the data that is to be included in the SWT cache group from the data to be excluded from the SWT cache group.

Hybrid Cache Group

A hybrid cache group is a dynamic read-only cache group where the root table is created in the TimesTen database and does not exist in the Oracle database.

A cache group is a set of tables related through foreign keys that cache data from tables in an Oracle database. Each cache group includes one root table that does not reference any of the other tables. Foreign keys on all other cache tables in the cache group reference exactly one other table in the cache group. In other words, the foreign key relationships form a tree. For multiple table cache groups, you determined the relationship between the tables by defining which table is the root table, which tables are direct child tables of the root table, and which tables are the child tables of other child tables. Historically, all tables within the cache group exist in the Oracle database.

With a hybrid cache group, you can dynamically load from cache tables that do not have a root table on the Oracle database. A hybrid cache group is a dynamic read-only cache group where the root table is created in the TimesTen database and does not exist in the Oracle database.

- TimesTen creates the root table on the TimesTen database from the definition of the hybrid cache group. Note that you should not create this table on the Oracle database.
- The only columns allowed in the root table definition are the columns defining the primary key.
- All other cache tables must exist in the Oracle database.
- The root table must be referenced by at least one child table through a foreign key relationship.

The following sections describe how to use a hybrid cache group:

- Creating a Hybrid Cache Group
- Specifying the Dynamic Load for a Hybrid Cache Group

- Automatic Passthrough for Hybrid Cache Groups
- Restrictions for a Dynamic Hybrid Read-Only Cache Group

Creating a Hybrid Cache Group

You can use the `CREATE DYNAMIC HYBRID READONLY CACHE GROUP` statement to create a dynamic hybrid read-only cache group where the root table exists only on TimesTen.

The following are the definitions of the tables that are to be cached in the `customer_orders` dynamic hybrid read-only cache group.

1. The customer root table exists only on the TimesTen database and contains only a primary key. You do not create the root table in the Oracle database as it is created by TimesTen when you specify the root table in the `CREATE DYNAMIC HYBRID READONLY CACHE GROUP` statement.
2. Customers can have more than one order and each order can go to a different location. To track the order status for each customer location, the locations and orders tables are created on the Oracle database and are children of the customer table.

   With the `customer_id` as part of the composite key for both the `locations` and `orders` tables, you can print out the status of all orders for each customer location. In addition, the `invoices` table (as a child of the orders table) can be queried to determine if the order has been paid.

   ```
   CREATE TABLE locations
     (customer_id NUMBER(6),
      location_id NUMBER(6),
      name VARCHAR2(255) NOT NULL,
      street CHAR(30) NOT NULL,
      city CHAR(20) NOT NULL,
      state CHAR(2) NOT NULL,
      zipcode CHAR(10) NOT NULL,
    PRIMARY KEY (customer_id, location_id));

   CREATE TABLE orders
     (order_id NUMBER,
      location_id NUMBER(6),
      customer_id NUMBER(6),
      when_placed  DATE NOT NULL,
      status NUMBER(2) NOT NULL,
   PRIMARY KEY (order_id, location_id, customer_id));

   CREATE TABLE invoices
     (invoice_id NUMBER PRIMARY KEY,
      order_id NUMBER,
      total   NUMBER,
      paid    NUMBER);
   ```
3. The Oracle cache administration user must be granted the `SELECT` privilege on the cached tables. In this example, these tables are `sales.locations`, `sales.orders` and `sales.invoices` tables.

   See Required Privileges for Cache Administration User for Cache Operations for all required privileges for different activities.

   On the Oracle database as an administrator, grant the following privileges:

   ```
   SQL> GRANT SELECT ON sales.locations TO cacheadmin;
   SQL> GRANT SELECT ON sales.orders TO cacheadmin;
   SQL> GRANT SELECT ON sales.invoices TO cacheadmin;
   ```

4. On the TimesTen database, connect as the TimesTen cache administration user to create the cache group. Use the `CREATE DYNAMIC HYBRID READONLY CACHE GROUP` statement to create the customer root table on TimesTen and a dynamic hybrid read-only cache group called `customer_orders`, which caches the Oracle database tables: `locations`, `orders`, and `invoices` (child tables). Note that the `locations` and `orders` cache tables reference the primary key of the `customer` root table that exists on the TimesTen database.

   Note

   See CREATE CACHE GROUP in the Oracle TimesTen In-Memory Database SQL Reference.

```
CREATE DYNAMIC HYBRID READONLY CACHE GROUP customer_orders
FROM customer
 (customer_id NUMBER(6) NOT NULL,
  PRIMARY KEY(customer_id)),

 locations
  (customer_id NUMBER(6),
   location_id NUMBER(6),
   name VARCHAR2(255) NOT NULL,
   street CHAR(30) NOT NULL,
   city CHAR(20) NOT NULL,
   state CHAR(2) NOT NULL,
   zipcode CHAR(10) NOT NULL,
 PRIMARY KEY (customer_id, location_id),
 FOREIGN KEY (customer_id) REFERENCES customer(customer_id)),

 orders
  (order_id NUMBER,
   location_id NUMBER(6),
   customer_id NUMBER(6),
   when_placed  DATE NOT NULL,
   status NUMBER(2) NOT NULL,
 PRIMARY KEY (order_id, location_id, customer_id),
 FOREIGN KEY (customer_id) REFERENCES customer(customer_id)),

 invoices
  (invoice_id NUMBER,
   order_id NUMBER,
   total    NUMBER,
   paid     NUMBER,
 PRIMARY KEY (invoice_id),
 FOREIGN KEY (order_id) REFERENCES order(order_id));
```

Specifying the Dynamic Load for a Hybrid Cache Group

For hybrid cache groups, you can specify a derived table within the `FROM` clause of the `SELECT` statement or include more than one table of the same hybrid cache group in the same query.

Dynamic load occurs after evaluating the rules specified in Guidelines for Dynamic Load.

Using a Derived Table

For hybrid cache groups, you can specify a derived table within the `FROM` clause of the `SELECT` statement. If the query specifies multiple tables including the derived table, then the materialized result of the derived table with the dynamic load condition is treated as a parent table (but only if the derived table specifies a single first child table of the hybrid cache group).

See DerivedTable in the Oracle TimesTen In-Memory Database SQL Reference.

Example 4-1 Using a Derived Table

The following query uses a derived table within the `FROM` clause of the `SELECT` statement. The materialized result of the derived table is treated as the parent table orders when determining if the query qualifies for a dynamic load. The following query uses a derived table within the `FROM` clause of the `SELECT` statement. The materialized result of the derived table is treated as the parent table orders when determining if the query qualifies for a dynamic load.

```
SELECT * FROM
 (SELECT customer_id FROM orders WHERE customer_id=? AND ROWNUM <= 5);
```

Including Multiple Tables

More than one table of the same hybrid cache group can be included in the same query.

- One or more first level child tables of the same hybrid cache group can be included in a query (including the option of a derived table that includes a first level child table):
  - Specifies the same foreign key as the other first child tables or derived table.
  - Includes a join condition that equates its foreign key with the foreign key of other first child tables or derived table.
- Any included grandchild table of the same hybrid cache group must:
  - Include a foreign key join condition with either the derived table or a first level child table of the same hybrid cache group.
  - Not be included in an outer table join with its parent table.

The following examples demonstrate the conditions that do and do not trigger a dynamic load for a hybrid cache group. All of these examples are based on the `customer_orders` hybrid cache group example defined in Creating a Hybrid Cache Group.

Example 4-2 Dynamic Load Condition Using Multiple First Level Child Tables

The following query triggers a dynamic load since two first level child tables (`orders` and `locations`) specify the same dynamic load condition.

```
SELECT * FROM orders, locations
 WHERE orders.customer_id=:id and locations.customer_id=:id;
```

And the following query triggers a dynamic load since the locations table equates its foreign key with the `orders` table foreign key.

```
SELECT * FROM orders, locations
 WHERE orders.customer_id=:id and locations.customer_id=:id;
```

Example 4-3 Dynamic Load Condition Using a First Level Child Table and a Derived Table

The following query triggers a dynamic load since two first level child tables (`orders` and `locations`) specify the same dynamic load condition. The `locations` table equates its foreign key with the dynamically loaded foreign key from the `orders` table.

The derived table is temporarily named `cust` as that name is provided directly after the derived table specification.

```
SELECT * FROM
  (SELECT customer_id,order_id FROM orders
          WHERE customer_id=:id and ROWNUM <= 5) cust, invoices, locations
```

```
 WHERE
    invoices.order_id = cust.order_id and locations.customer_id=cust.customer_id;
```

Example 4-4   Dynamic Load Condition Using a First Level Child Table and Grandchild Table

The following query example triggers a dynamic load since the dynamic load condition is on a derived table that includes the `orders` table (a first level child table of the `customer_orders` hybrid cache group). It also includes the grandchild table `invoices` that is included in a foreign key join condition with the derived table `cust`.

Temporarily, the derived table name is `orders` and is treated as a parent table.

```
SELECT * FROM
  (SELECT customer_id,order_id FROM orders
          WHERE customer_id=? and ROWNUM <= 5) cust, invoices
 WHERE invoices.order_id = cust.order_id;
```

Example 4-5   Dynamic Load Using Grandchild Table Joined with Derived Table

The following query triggers a dynamic load because the `invoices` table (as a grandchild table) is joined with the derived table `cust` through a foreign key join:

```
SELECT * FROM
  (SELECT customer_id,order_id FROM orders
          WHERE customer_id=? and ROWNUM <= 5) cust, invoices
 WHERE invoices.order_id = cust.order_id;
```

Example 4-6   No Dynamic Load Example with First Level Child Table

The following query does not trigger a dynamic load because the first level child `locations` table specifies a different dynamic load condition than the derived table (`cust`) load condition:

```
SELECT * FROM
  (SELECT customer_id,order_id FROM orders
          WHERE customer_id=:id and ROWNUM <= 5) cust, invoices, locations
 WHERE invoices.order_id = cust.order_id and locations.customer_id=:id2;
```

Example 4-7   No Dynamic Load Example Using Grandchild Table

The following query does not trigger a dynamic load because the `invoices` grandchild table is not joined through a foreign key join with its parent, the `orders` table.

```
SELECT * FROM
  (SELECT customer_id,order_id FROM orders
          WHERE customer_id=? and ROWNUM <= 5) cust, invoices
 WHERE invoices.invoice_id=?;
```

Example 4-8   No Dynamic Load Second Example Using Grandchild Table

The following query does not trigger a dynamic load because the `invoices` grandchild table is included in an outer table of a join with its parent, the `orders` table.

```
SELECT * FROM invoices LEFT JOIN
  (SELECT customer_id,order_id FROM orders
          WHERE customer_id=? and ROWNUM <= 5) cust
 ON invoices.order_id = cust.order_id;
```

Automatic Passthrough for Hybrid Cache Groups

Set the `TT_DynamicPassthrough` optimizer hint to notify TimesTen to pass through qualified `SELECT` statements to the Oracle database for cache groups created without a WHERE clause.

For cache groups without a `WHERE` clause, you can set the `TT_DynamicPassthrough(`*N*`)` optimizer hint that notifies TimesTen to pass through any `SELECT` statement to the Oracle database if it results in a dynamic load of a cache instance with >= *N* number of rows. See Automatic Passthrough of Dynamic Load to the Oracle Database.

Restrictions for a Dynamic Hybrid Read-Only Cache Group

Restrictions for using a dynamic hybrid read-only cache group.

The following are restrictions for a dynamic hybrid read-only cache group:

- You can execute a `SELECT` statement on the root table, as this may help in diagnosing problems. However, a dynamic load is not triggered if you execute a `SELECT` on the root table in TimesTen.
- Hybrid cache groups do not support manually loading the cache group with the `LOAD CACHE GROUP` statement.
- LRU aging is on by default for dynamic cache groups, including hybrid cache groups. Currently, time-based aging is not supported for hybrid cache groups.
- Currently, the `WHERE` clause is not supported in `CREATE CACHE GROUP` for hybrid cache groups.
- Currently, the `WITH ID` clause is not supported in `UNLOAD CACHE GROUP` for hybrid cache groups.

User Managed Cache Group

If the system managed cache groups (read-only, AWT, SWT) do not satisfy your application's requirements, you can create a user managed cache group that defines customized caching behavior.

Create a user managed cache group with customized caching behavior with one or more of the following cache table attributes:

Only TimesTen supports user-managed cache groups.

Note

When TimesTen manages operations for user managed cache groups, it connects to the Oracle database using the current user's credentials provided on the connection string. The current user's credentials can be provided with an Oracle Wallet pointed to by the `PwdWallet` connection attribute or with the `UID`, `PWD`, and `OraclePwd` connection attributes. TimesTen does not connect to the Oracle database with the Oracle cache administration user name and password registered with the `ttCacheUidPwdSet` built-in procedure when managing SWT cache group operations. See Providing Cache Administration User Credentials When Connecting and Registering the Cache Administration User Name and Password..

- You can specify the READONLY Cache Table Attribute on individual cache tables in a user managed cache group to define read-only behavior where the data is refreshed on TimesTen from the Oracle database at the table level.
- You can specify the `PROPAGATE` cache table attribute on individual cache tables in a user managed cache group to define synchronous writethrough behavior at the table level. The

PROPAGATE Cache Table Attribute specifies that committed changes on the cache table are automatically and synchronously propagated to the cached Oracle Database table.

- You can define a user managed cache group to automatically refresh and propagate committed changes between the Oracle and TimesTen databases by using the `AUTOREFRESH` cache group attribute and the `PROPAGATE` cache table attribute. Using both attributes enables bidirectional transmit, so that committed changes on the TimesTen cache tables or the cached Oracle Database tables are propagated or refreshed to each other.

  See Automatically Refreshing a Cache Group for more information about defining an autorefresh mode, interval, and state.
- You can use the `LOAD CACHE GROUP`, `REFRESH CACHE GROUP`, and `FLUSH CACHE GROUP` statements to manually control the transmit of committed changes between the Oracle and TimesTen databases.

  See Manually Loading and Refreshing a Cache Group for more information about the `LOAD CACHE GROUP` and `REFRESH CACHE GROUP` statements. See Flushing a User Managed Cache Group for more information about the `FLUSH CACHE GROUP` statement.
- You can cache Oracle Database materialized views in a user managed cache group that does not use either the `PROPAGATE` or `AUTOREFRESH` cache group attributes. The cache group must be manually loaded and flushed. You cannot cache Oracle Database views.

The following sections provide more information about user managed cache groups:

- READONLY Cache Table Attribute
- PROPAGATE Cache Table Attribute
- Examples of User Managed Cache Groups

READONLY Cache Table Attribute

The `READONLY` cache table attribute can be specified only for cache tables in a user managed cache group.

`READONLY` specifies that the cache table cannot be updated directly. By default, a cache table in a user managed cache group is updatable.

Unlike a read-only cache group where all of its cache tables are read-only, in a user managed cache group individual cache tables can be specified as read-only using the `READONLY` cache table attribute.

The following restrictions apply when using the `READONLY` cache table attribute:

- If the cache group uses the `AUTOREFRESH` cache group attribute, the `READONLY` cache table attribute must be specified on all or none of its cache tables.

  See Automatically Refreshing a Cache Group for more information about using the `AUTOREFRESH` cache group attribute.
- You cannot use both the `READONLY` and `PROPAGATE` cache table attributes on the same cache table.

  See PROPAGATE Cache Table Attribute for more information about using the `PROPAGATE` cache table attribute.
- A `FLUSH CACHE GROUP` statement cannot be issued on the cache group unless one or more of its cache tables use neither the `READONLY` nor the `PROPAGATE` cache table attribute.

See Flushing a User Managed Cache Group for more information about the `FLUSH CACHE GROUP` statement.

- After the `READONLY` cache table attribute has been specified on a cache table, you cannot change this attribute unless you drop the cache group and re-create it.

PROPAGATE Cache Table Attribute

The `PROPAGATE` cache table attribute can be specified only for cache tables in a user managed cache group.

`PROPAGATE` specifies that committed changes on the TimesTen cache table as part of a TimesTen transaction are automatically and synchronously propagated to the cached Oracle Database table. If the `PROPAGATE` cache table attribute is not specified, then the default setting for a cache table in a user managed cache group is the `NOT PROPAGATE` cache table attribute (which does not propagate committed changes on the cache table to the cached Oracle table).

All SQL statements run by an application on cached tables are applied to the cached tables immediately. All of these operations are buffered until the transaction commits or reaches a memory upper limit. At this time, all operations are propagated to the tables in the Oracle database.

Note

If the TimesTen database or its daemon fails unexpectedly, the results of the transaction on either the TimesTen or Oracle databases are not guaranteed.

Since the operations in the transaction are applied to tables in both the TimesTen and Oracle databases, the process for committing is as follows:

1. After the operations are propagated to the Oracle database, the commit is first attempted in the Oracle database.
   - If an error occurs when applying the operations on the tables in the Oracle database, then all operations are rolled back on the tables on the Oracle database. If the commit fails in the Oracle database, the commit is not attempted in the TimesTen database and the application must roll back the TimesTen transaction. If the user tries to run another statement, an error displays informing them of the need for a roll back. As a result, the Oracle database never misses updates committed in TimesTen.
2. If the commit succeeds in the Oracle database, the commit is attempted in the TimesTen database.
   - If the transaction successfully commits on the Oracle database, the user's transaction is committed on TimesTen (indicated by the commit log record in the transaction log) and notifies the application. If the application ends abruptly before TimesTen informs it of the success of the local commit, TimesTen is still able to finalize the transaction commit on TimesTen based on what is saved in the transaction log.
   - If the transaction successfully commits on the Oracle database and a failure occurs before returning the status of the commit on TimesTen, then no record of the successful commit is written into the transaction log and the transaction is rolled back.
   - If the commit fails in TimesTen, an error message is returned from TimesTen indicating the cause of the failure. You then need to manually resynchronize the cache tables with the Oracle Database tables.

Note

See Synchronous WriteThrough (SWT) Cache Group for information on how to re-synchronize the cache tables with the Oracle Database tables.

You can disable propagation of committed changes on the TimesTen cached tables to the Oracle database with the `ttCachePropagateFlagSet` built-in procedure. This built-in procedure can enable or disable automatic propagation so that committed changes on a cache table on TimesTen for the current transaction are never propagated to the cached Oracle Database table. You can then re-enable propagation for DML statements by resetting the flag to one with the `ttCachePropagateFlagSet` built-in procedure. After the flag is set back to one, propagation of committed changes to the Oracle database resumes. The propagation flag automatically resets to one after the transaction is committed or rolled back. See ttCachePropagateFlagSet in the *Oracle TimesTen In-Memory Database Reference*.

The following restrictions apply when using the `PROPAGATE` cache table attribute:

- If the cache group uses the `AUTOREFRESH` cache group attribute, the `PROPAGATE` cache table attribute must be specified on all or none of its cache tables.

  See Automatically Refreshing a Cache Group for more information about using the `AUTOREFRESH` cache group attribute.
- If the cache group uses the `AUTOREFRESH` cache group attribute, the `NOT PROPAGATE` cache table attribute cannot be explicitly specified on any of its cache tables.
- You cannot use both the `PROPAGATE` and `READONLY` cache table attributes on the same cache table.

  See READONLY Cache Table Attribute for more information about using the `READONLY` cache table attribute.
- A `FLUSH CACHE GROUP` statement cannot be issued on the cache group unless one or more of its cache tables use neither the `PROPAGATE` nor the `READONLY` cache table attribute.

  See Flushing a User Managed Cache Group for more information about the `FLUSH CACHE GROUP` statement.
- After the `PROPAGATE` cache table attribute has been specified on a cache table, you cannot change this attribute unless you drop the cache group and re-create it.
- The `PROPAGATE` cache table attribute cannot be used when caching Oracle Database materialized views.
- TimesTen does not perform a conflict check to prevent a propagate operation from overwriting data that was updated directly on a cached Oracle Database table. Therefore, updates should only be performed directly on the TimesTen cache tables or the cached Oracle Database tables, but not both.

Examples of User Managed Cache Groups

Examples are provided for the definition of the Oracle Database tables that are to be cached in the user managed cache groups.

On the Oracle Database:

These Oracle database tables are owned by the schema user `sales`.

```
CREATE TABLE active_customer
 (custid NUMBER(6) NOT NULL PRIMARY KEY,
  name   VARCHAR2(50),
  addr   VARCHAR2(100),
  zip    VARCHAR2(12),
  region VARCHAR2(12) DEFAULT 'Unknown');

CREATE TABLE ordertab
 (orderid NUMBER(10) NOT NULL PRIMARY KEY,
  custid  NUMBER(6) NOT NULL);

CREATE TABLE cust_interests
 (custid   NUMBER(6) NOT NULL,
  interest VARCHAR2(10) NOT NULL,
  PRIMARY KEY (custid, interest));

CREATE TABLE orderdetails
 (orderid  NUMBER(10) NOT NULL,
  itemid   NUMBER(8) NOT NULL,
  quantity NUMBER(4) NOT NULL,
  PRIMARY KEY (orderid, itemid));
```

The Oracle cache administration user must be granted the `SELECT` privilege on any cached tables. In this example, the table is the `sales.active_customer` table.

On the Oracle database as an administrator, grant the following privileges:

```
SQL> GRANT SELECT ON sales.active_customer TO cacheadmin;
```

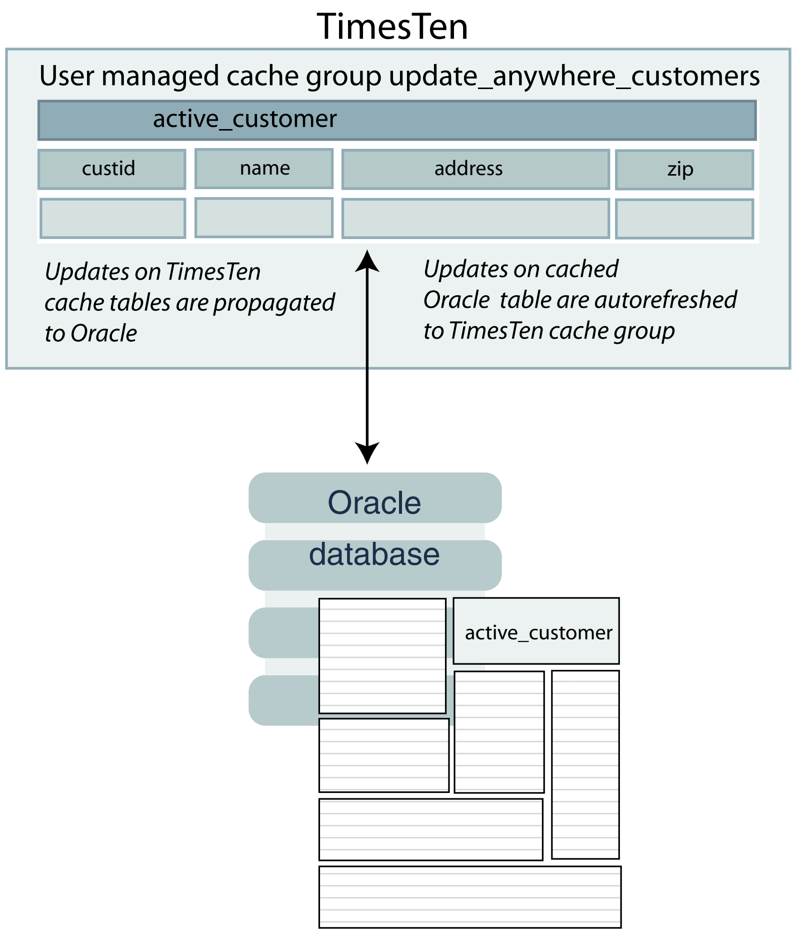
On the TimesTen database:

Connect as the TimesTen cache administration user. Use the `CREATE USERMANAGED CACHE GROUP` statement to create a user managed cache group.

The following statement creates a user managed cache group `update_anywhere_customers` that caches the `sales.active_customer` table as shown in Figure 4-8:

```
CREATE USERMANAGED CACHE GROUP update_anywhere_customers
AUTOREFRESH MODE INCREMENTAL INTERVAL 30 SECONDS
FROM sales.active_customer
 (custid NUMBER(6) NOT NULL,
  name   VARCHAR2(50),
  addr   VARCHAR2(100),
  zip    VARCHAR2(12),
  PRIMARY KEY(custid),
  PROPAGATE);
```

Figure 4-8 Single-Table User Managed Cache Group


TimesTen
User managed cache group update_anywhere_customers
active_customer
custid
name
address
zip
Updates on TimesTen cache tables are propagated to Oracle
Updates on cached Oracle table are autorefreshed to TimesTen cache group
Oracle database
active_customer

In this example, all columns except `region` from the `sales.active_customer` table are cached in TimesTen. Since this is defined with the `PROPAGATE` cache table attribute, updates committed on the `sales.active_customer` cache table on TimesTen are transmitted to the `sales.active_customer` cached Oracle Database table. Since the user managed cache table is also defined with the `AUTOREFRESH` cache attribute, any committed changes on the `sales.active_customer` Oracle Database table are transmitted to the `update_anywhere_customers` cached table.

In this example, the `AUTOREFRESH` cache group attribute specifies that committed changes on the `sales.active_customer` cached Oracle Database table are automatically refreshed to the TimesTen `sales.active_customer cache` table every 30 seconds.

If you manually created the Oracle Database objects used to enforce the predefined behaviors of a user managed cache group that uses the `AUTOREFRESH MODE INCREMENTAL` cache group attribute as described in The initCacheAdminSchema.sql Script, you need to set the autorefresh state to `OFF` when creating the cache group.

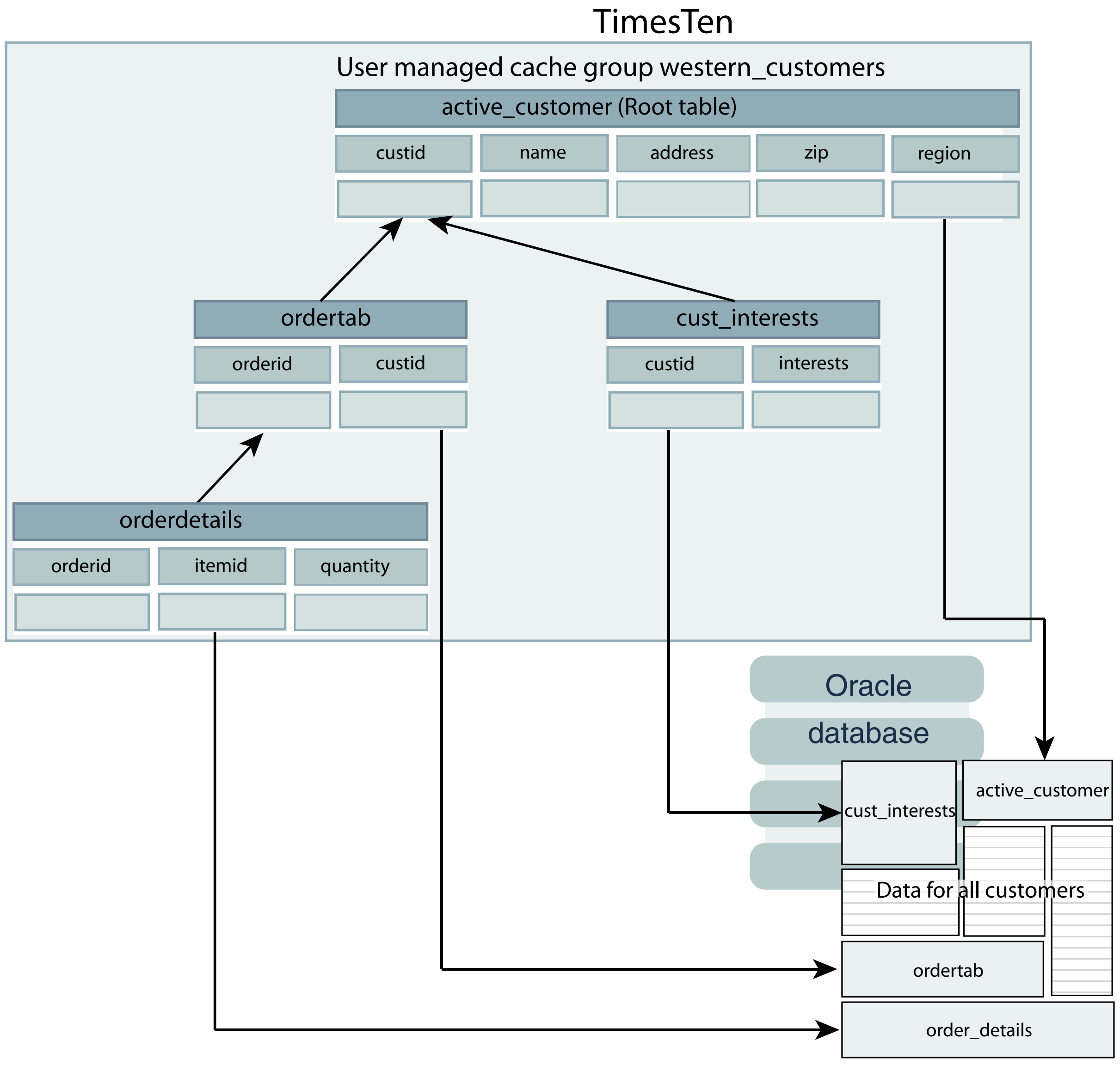
Then you need to run the `ttIsql` utility's `cachesqlget` command to generate a SQL*Plus script used to create a log table and a trigger in the Oracle database for each Oracle Database table that is cached in the user managed cache group.

See Manually Creating Oracle Database Objects for Cache Groups with Autorefresh.

The following statement creates a multiple-table user managed cache group `western_customers` that caches the `sales.active_customer`, `sales.ordertab`, `sales.cust_interests`, and `sales.orderdetails` tables as shown in Figure 4-9:

```
CREATE USERMANAGED CACHE GROUP western_customers
FROM sales.active_customer
 (custid NUMBER(6) NOT NULL,
  name   VARCHAR2(50),
  addr   VARCHAR2(100),
  zip    VARCHAR2(12),
  region VARCHAR2(12),
  PRIMARY KEY(custid),
  PROPAGATE)
  WHERE (sales.active_customer.region = 'West'),
sales.ordertab
 (orderid NUMBER(10) NOT NULL,
  custid  NUMBER(6) NOT NULL,
  PRIMARY KEY(orderid),
  FOREIGN KEY(custid) REFERENCES sales.active_customer(custid),
  PROPAGATE),
sales.cust_interests
 (custid   NUMBER(6) NOT NULL,
  interest VARCHAR2(10) NOT NULL,
  PRIMARY KEY(custid, interest),
  FOREIGN KEY(custid) REFERENCES sales.active_customer(custid),
  READONLY),
sales.orderdetails
 (orderid  NUMBER(10) NOT NULL,
  itemid   NUMBER(8) NOT NULL,
  quantity NUMBER(4) NOT NULL,
  PRIMARY KEY(orderid, itemid),
  FOREIGN KEY(orderid) REFERENCES sales.ordertab(orderid))
  WHERE (sales.orderdetails.quantity >= 5);
```

Figure 4-9 Multiple-Table User Managed Cache Group



Only customers in the West region who ordered at least 5 of the same item are cached.

Each cache table in the `western_customers` cache group contains a primary key. Each child table references a parent table with a foreign key constraint. The `sales.active_customer` root table and the `sales.orderdetails` child table each contain a `WHERE` clause to restrict the rows to be cached. The `sales.active_customer` root table and the `sales.ordertab` child table both use the PROPAGATE Cache Table Attribute so that committed changes on these cache tables are automatically propagated to the cached Oracle Database tables. The `sales.cust_interests` child table uses the READONLY Cache Table Attribute so that it cannot be updated directly.

Using a WHERE Clause

A cache table definition in a `CREATE CACHE GROUP` statement can contain a `WHERE` clause to restrict the rows to cache in the TimesTen database for particular cache group types.

You can also specify a `WHERE` clause in a `LOAD CACHE GROUP`, `UNLOAD CACHE GROUP`, `REFRESH CACHE GROUP` or `FLUSH CACHE GROUP` statement for particular cache group types. Some statements, such as `LOAD CACHE GROUP` and `REFRESH CACHE GROUP`, may result in concatenated `WHERE` clauses in which the `WHERE` clause for the cache table definition is evaluated before the `WHERE` clause in the `LOAD CACHE GROUP` or `REFRESH CACHE GROUP` statement.

The following restrictions apply to `WHERE` clauses used in cache table definitions and cache group operations:

- `WHERE` clauses can only be specified in the cache table definitions of a `CREATE CACHE GROUP` statement for read-only and user managed cache groups.
- A `WHERE` clause can be specified in a `LOAD CACHE GROUP` statement except on a static cache group with autorefresh.

  See Manually Loading and Refreshing a Cache Group for more information about the `LOAD CACHE GROUP`.
- A `WHERE` clause can be specified in a `REFRESH CACHE GROUP` statement except on a cache group with autorefresh.

  See Manually Loading and Refreshing a Cache Group for more information about the `REFRESH CACHE GROUP` statement.
- A `WHERE` clause can be specified in a `FLUSH CACHE GROUP` statement on a user managed cache group that allows committed changes on the TimesTen cache tables to be flushed to the cached Oracle Database tables.

  See Flushing a User Managed Cache Group for more information about the `FLUSH CACHE GROUP` statement.
- `WHERE` clauses in a `CREATE CACHE GROUP` statement cannot contain a subquery. Therefore, each `WHERE` clause cannot reference any table other than the one in its cache table definition. However, a `WHERE` clause in a `LOAD CACHE GROUP`, `UNLOAD CACHE GROUP`, `REFRESH CACHE GROUP` or `FLUSH CACHE GROUP` statement may contain a subquery.
- A `WHERE` clause in a `LOAD CACHE GROUP`, `REFRESH CACHE GROUP` or `FLUSH CACHE GROUP` statement can reference only the root table of the cache group, unless the `WHERE` clause contains a subquery.
- All tables and columns referenced in `WHERE` clauses when creating, loading, refreshing, unloading or flushing the cache group must be fully qualified. For example:

  `owner.table_name` and *`owner.table_name.column_name`*

Proper Placement of WHERE Clause in a CREATE CACHE GROUP Statement

In a multiple-table cache group, a `WHERE` clause in a particular table definition should not reference any table in the cache group other than the table itself. For example, the following `CREATE CACHE GROUP` statements are valid:

```
CREATE READONLY CACHE GROUP customer_orders
FROM sales.customer
 (cust_num NUMBER(6) NOT NULL,
  region   VARCHAR2(10),
  name     VARCHAR2(50),
  address  VARCHAR2(100),
  PRIMARY KEY(cust_num))
  WHERE (sales.customer.cust_num < 100),
sales.orders
```

```
 (ord_num      NUMBER(10) NOT NULL,
  cust_num     NUMBER(6) NOT NULL,
  when_placed  DATE NOT NULL,
  when_shipped DATE NOT NULL,
  PRIMARY KEY(ord_num),
  FOREIGN KEY(cust_num) REFERENCES sales.customer(cust_num));

CREATE READONLY CACHE GROUP customer_orders
FROM sales.customer
 (cust_num NUMBER(6) NOT NULL,
  region   VARCHAR2(10),
  name     VARCHAR2(50),
  address  VARCHAR2(100),
  PRIMARY KEY(cust_num)),
sales.orders
 (ord_num      NUMBER(10) NOT NULL,
  cust_num     NUMBER(6) NOT NULL,
  when_placed  DATE NOT NULL,
  when_shipped DATE NOT NULL,
  PRIMARY KEY(ord_num),
  FOREIGN KEY(cust_num) REFERENCES sales.customer(cust_num))
  WHERE (sales.orders.cust_num < 100);
```

The following statement is not valid because the `WHERE` clause in the child table's definition references its parent table:

```
CREATE READONLY CACHE GROUP customer_orders
FROM sales.customer
 (cust_num NUMBER(6) NOT NULL,
  region   VARCHAR2(10),
  name     VARCHAR2(50),
  address  VARCHAR2(100),
  PRIMARY KEY(cust_num)),
sales.orders
 (ord_num      NUMBER(10) NOT NULL,
  cust_num     NUMBER(6) NOT NULL,
  when_placed  DATE NOT NULL,
  when_shipped DATE NOT NULL,
  PRIMARY KEY(ord_num),
  FOREIGN KEY(cust_num) REFERENCES sales.customer(cust_num))
  WHERE (sales.customer.cust_num < 100);
```

Similarly, the following statement is not valid because the `WHERE` clause in the parent table's definition references its child table:

```
CREATE READONLY CACHE GROUP customer_orders
FROM sales.customer
 (cust_num NUMBER(6) NOT NULL,
  region   VARCHAR2(10),
  name     VARCHAR2(50),
  address  VARCHAR2(100),
  PRIMARY KEY(cust_num))
  WHERE (sales.orders.cust_num < 100),
sales.orders
 (ord_num      NUMBER(10) NOT NULL,
  cust_num     NUMBER(6) NOT NULL,
  when_placed  DATE NOT NULL,
  when_shipped DATE NOT NULL,
  PRIMARY KEY(ord_num),
  FOREIGN KEY(cust_num) REFERENCES sales.customer(cust_num));
```

Referencing Oracle Database PL/SQL Functions in a WHERE Clause

A user-defined PL/SQL function in the Oracle database can be invoked indirectly in a WHERE clause within a CREATE CACHE GROUP, LOAD CACHE GROUP, or REFRESH CACHE GROUP (for dynamic cache groups only) statement.

After creating the function, create a public synonym for the function. Then grant the EXECUTE privilege on the function to PUBLIC.

For example, in the Oracle database:

```
CREATE OR REPLACE FUNCTION get_customer_name
(c_num sales.customer.cust_num%TYPE) RETURN VARCHAR2 IS
c_name sales.customer.name%TYPE;
BEGIN
  SELECT name INTO c_name FROM sales.customer WHERE cust_num = c_num;
  RETURN c_name;
END get_customer_name;

CREATE PUBLIC SYNONYM retname FOR get_customer_name;
GRANT EXECUTE ON get_customer_name TO PUBLIC;
```

Then in the TimesTen database, for example, you can create a cache group with a WHERE clause that references the Oracle Database public synonym that was created for the function:

```
CREATE READONLY CACHE GROUP top_customer
FROM sales.customer
 (cust_num NUMBER(6) NOT NULL,
  region   VARCHAR2(10),
  name     VARCHAR2(50),
  address  VARCHAR2(100),
  PRIMARY KEY(cust_num))
WHERE sales.customer.name = retname(100);
```

For cache group types that allow a WHERE clause on a LOAD CACHE GROUP or REFRESH CACHE GROUP statement, you can invoke the function indirectly by referencing the public synonym that was created for the function. For example, you can use the following LOAD CACHE GROUP statement to load the AWT cache group new_customers:

```
LOAD CACHE GROUP new_customers WHERE name = retname(101) COMMIT EVERY 0 ROWS;
```

Specifying Automatic Refresh with the AUTOREFRESH Cache Group Attribute

The AUTOREFRESH cache group attribute can be specified when creating a read-only cache group or a user managed cache group using a CREATE CACHE GROUP statement.

AUTOREFRESH specifies that committed changes on cached Oracle Database tables are automatically refreshed to the cache tables on TimesTen. Autorefresh is defined by default on read-only cache groups. See Automatically Refreshing a Cache Group.

Creating a Dynamic Cache Group with the DYNAMIC Keyword

You define whether your cache group is dynamically loaded during cache group definition with the DYNAMIC keyword.

Note

Only TimesTen supports the `DYNAMIC` keyword in its cache groups.

See Dynamic Cache Groups.

Creating a Hash Index on the Primary Key Columns of the Cache Table

The `UNIQUE HASH ON` cache table attribute can be specified for cache tables in any cache group type. Hash indexes give faster full key equality lookups but can be used for inequality (<, <-, >, >=) or range lookups.

`UNIQUE HASH ON` specifies that a hash index rather than a range index is created on the primary key columns of the cache table. The columns specified in the hash index must be identical to the columns in the primary key. The `UNIQUE HASH ON` cache table attribute is also used to specify the size of the hash index.

The following example demonstrates how to use the `UNIQUE HASH ON` cache table attribute on the cache table's definition.

```
CREATE ASYNCHRONOUS WRITETHROUGH CACHE GROUP new_customers
FROM sales.customer
 (cust_num NUMBER(6) NOT NULL,
  region   VARCHAR2(10),
  name     VARCHAR2(50),
  address  VARCHAR2(100),
  PRIMARY KEY(cust_num))
  UNIQUE HASH ON (cust_num) PAGES = 100;
```

See CREATE CACHE GROUP in the *Oracle TimesTen In-Memory Database SQL Reference*.

ON DELETE CASCADE Cache Table Attribute

The `ON DELETE CASCADE` cache table attribute can be specified for cache tables in any cache group type.

`ON DELETE CASCADE` specifies that when rows containing referenced key values are deleted from a parent table, rows in child tables with dependent foreign keys are also deleted.

The following example demonstrates how to use the `ON DELETE CASCADE` cache table attribute on the child table's foreign key definition:

```
CREATE READONLY CACHE GROUP customer_orders
FROM sales.customer
 (cust_num NUMBER(6) NOT NULL,
  region   VARCHAR2(10),
  name     VARCHAR2(50),
  address  VARCHAR2(100),
  PRIMARY KEY(cust_num)),
sales.orders
 (ord_num      NUMBER(10) NOT NULL,
  cust_num     NUMBER(6) NOT NULL,
  when_placed  DATE NOT NULL,
  when_shipped DATE NOT NULL,
  PRIMARY KEY(ord_num),
  FOREIGN KEY(cust_num) REFERENCES sales.customer(cust_num) ON DELETE CASCADE);
```

All paths from a parent table to a child table must be either "delete" paths or "do not delete" paths. There cannot be some "delete" paths and some "do not delete" paths from a parent table to a child table. Specify the `ON DELETE CASCADE` cache table attribute for child tables on a "delete" path.

The following restrictions apply when using the `ON DELETE CASCADE` cache table attribute:

- For AWT and SWT cache groups, and for TimesTen cache tables in user managed cache groups that use the `PROPAGATE` cache table attribute, foreign keys in cache tables that use the `ON DELETE CASCADE` cache table attribute must be a proper subset of the foreign keys in the cached Oracle Database tables that use the `ON DELETE CASCADE` attribute. `ON DELETE CASCADE` actions on the cached Oracle Database tables are applied to the cache tables on TimesTen as individual deletes. `ON DELETE CASCADE` actions on the cache tables are applied to the cached Oracle Database tables as a cascaded operation.
- Matching of foreign keys between the cache tables on TimesTen and the cached Oracle Database tables is enforced only when the cache group is being created. A cascade delete operation may not work if the foreign keys on the cached Oracle Database tables are altered after the cache group is created.

See CREATE CACHE GROUP in the *Oracle TimesTen In-Memory Database SQL Reference*.

Caching Oracle Database Synonyms

You can cache a private synonym in an AWT, SWT or user managed cache group that does not use the `AUTOREFRESH` cache group attribute.

The private synonym can reference a public or private synonym, but it must eventually reference a table because it is the table that is actually being cached.

The table that is directly or indirectly referenced by the cached synonym can be owned by a user other than the Oracle Database user with the same name as the owner of the cache group that caches the synonym. The table must reside in the same Oracle database as the synonym. The cached synonym itself must be owned by the Oracle Database user with the same name as the owner of the cache group that caches the synonym.

Caching Oracle Database LOB Data

You can cache Oracle Database large object (LOB) data in cache groups in TimesTen.

TimesTen caches the data as follows:

- Oracle Database `CLOB` data is cached as TimesTen `VARCHAR2` data.
- Oracle Database `BLOB` data is cached as TimesTen `VARBINARY` data.
- Oracle Database `NCLOB` data is cached as TimesTen `NVARCHAR2` data.

The following example shows how to cache Oracle Database LOB data

1. Create a table in the Oracle database that has LOB fields.

   ```
   CREATE TABLE t (
     i INT NOT NULL PRIMARY KEY
     , c CLOB
     , b BLOB
     , nc NCLOB);
   ```

2. Insert values into the Oracle Database table. The values are implicitly converted to TimesTen VARCHAR2, VARBINARY, OR NVARCHAR2 data types.

   ```
   INSERT INTO t VALUES (1
     , RPAD('abcdefg8', 2048, 'abcdefg8')
     , HEXTORAW(RPAD('123456789ABCDEF8', 4000, '123456789ABCDEF8'))
     , RPAD('abcdefg8', 2048, 'abcdefg8')
   );

   1 row inserted.
   ```

3. Create a dynamic AWT cache group and start the replication agent.

   ```
   CREATE DYNAMIC ASYNCHRONOUS WRITETHROUGH CACHE GROUP cg1
     FROM t
    (i INT NOT NULL PRIMARY KEY
     , c VARCHAR2(4194303)
     , b VARBINARY(4194303)
     , nc NVARCHAR2(2097152));

   CALL ttrepstart;
   ```

4. Load the data dynamically into the TimesTen cache group.

   ```
   SELECT * FROM t WHERE i = 1;

   I:    1
   C:    abcdefg8abcdefg8abcdefg8...
   B:    123456789ABCDEF8123456789...
   NC:   abcdefg8abcdefg8abcdefg8...

   1 row found.
   ```

Restrictions on Caching Oracle Database LOB Data

There are restrictions when caching Oracle Database LOB data into TimesTen.

These restrictions apply to caching Oracle Database LOB data in TimesTen cache groups:

- Column size is enforced when a cache group is created. VARBINARY, VARCHAR2 and NVARCHAR2 data types have a size limit of 4 megabytes. Values that exceed the user-defined column size are truncated at run time without notification.
- Empty values in fields with CLOB and BLOB data types are initialized but not populated with data. Empty CLOB and BLOB fields are treated as follows:
  – Empty LOB fields in the Oracle database are returned as NULL values.
  – Empty VARCHAR2 and VARBINARY fields in TimesTen are propagated as NULL values.

In addition, cache groups that are configured for autorefresh operations have these restrictions on caching LOB data:

- When LOB data is updated in the Oracle database by OCI functions or the DBMS_LOB PL/SQL package, the data is not automatically refreshed in the cache group in TimesTen. This occurs because TimesTen caching operations depend on Oracle Database triggers, and Oracle Database triggers are not processed when these types of updates occur. TimesTen does not notify the user that updates have occurred without being refreshed in TimesTen. When the LOB data is updated in the Oracle database through a SQL statement, a trigger is fired and autorefresh brings in the change.
- Since autorefresh operations always refresh entire rows, LOB data in the cache is updated when any other column in the same row is updated.

Implementing Aging in a Cache Group for TimesTen

You can define an aging policy for a cache group in TimesTen that specifies the aging type, the aging attributes, and the aging state. TimesTen supports two aging types, least recently used (LRU) aging and time-based aging.

LRU aging deletes the least recently used or referenced data based on a specified database usage range. Time-based aging deletes data based on a specified data lifetime and frequency of the aging process. You can use both LRU and time-based aging in the same TimesTen database, but you can define only one aging policy for a particular cache group.

An aging policy is specified in the cache table definition of the root table in a `CREATE CACHE GROUP` statement and applies to all cache tables in the cache group because aging is performed at the cache instance level. When rows are deleted from the cache tables by aging out, the rows in the cached Oracle Database table are not deleted.

You can add an aging policy to a cache group by using an `ALTER TABLE` statement on the root table. You can change the aging policy of a cache group by using `ALTER TABLE` statements on the root table to drop the existing aging policy and then add a new aging policy.

This section describes cache group definitions that contain an aging policy.

- LRU Aging in TimesTen
- Time-Based Aging in TimesTen
- Manually Scheduling an Aging Process in TimesTen

LRU Aging in TimesTen

LRU aging enables you to maintain the amount of memory used in a TimesTen database within a specified threshold by deleting the least recently used data. LRU aging can be defined for all cache group types except static cache groups with autorefresh enabled. LRU aging is defined by default on dynamic cache groups.

Define an LRU aging policy for a cache group by using the `AGING LRU` clause in the cache table definition of the `CREATE CACHE GROUP` statement. Aging occurs automatically if the aging state is set to its default of `ON`.

The following example defines an LRU aging policy on the AWT cache group `new_customers`:

```
CREATE ASYNCHRONOUS WRITETHROUGH CACHE GROUP new_customers
FROM sales.customer
 (cust_num NUMBER(6) NOT NULL,
  region   VARCHAR2(10),
  name     VARCHAR2(50),
  address  VARCHAR2(100),
  PRIMARY KEY(cust_num))
AGING LRU ON;
```

There are two LRU aging policies:

- LRU aging based on set thresholds for the amount of permanent memory in use. This is the default. Once you create (or alter) a table to use LRU aging, the LRU aging policy defaults to using the default thresholds for permanent memory in use. See Defining LRU Aging Based on Thresholds for Permanent Memory in Use in the *Oracle TimesTen In-Memory Database Operations Guide*.

- LRU aging based on row thresholds for a specified root tables of your cache groups. See Defining LRU Aging Based on Row Thresholds for Tables in the *Oracle TimesTen In-Memory Database Operations Guide*.

Both types of LRU aging can co-exist. Row threshold based aging takes precedence over permanent memory in use based aging.

If a row has been accessed or referenced since the last aging cycle, it is not eligible for LRU aging in the current aging cycle. A row is considered to be accessed or referenced if at least one of the following is true:

- The row is used to build the result set of a `SELECT` or an `INSERT ... SELECT` statement.
- The row has been marked to be updated or deleted in a pending transaction.

In a multiple-table cache group, if a row in a child table has been accessed or referenced since the last aging cycle, then neither the related row in the parent table nor the row in the child table is eligible for LRU aging in the current aging cycle.

The `ALTER TABLE` statement can be used to perform the following tasks associated with changing or defining an LRU aging policy on a cache group:

- Change the aging state of a cache group by specifying the root table and using the `SET AGING` clause.
- Add an LRU aging policy to a cache group that has no aging policy defined by specifying the root table and using the `ADD AGING LRU` clause.
- Drop the LRU aging policy on a cache group by specifying the root table and using the `DROP AGING` clause.

To change the aging policy of a cache group from LRU to time-based, use an `ALTER TABLE` statement on the root table with the `DROP AGING` clause to drop the LRU aging policy. Then use an `ALTER TABLE` statement on the root table with the `ADD AGING USE` clause to add a time-based aging policy.

You must stop the cache agent before you add, alter or drop an aging policy on a cache group with autorefresh.

Time-Based Aging in TimesTen

Time-based aging deletes data from a cache group based on the aging policy's specified data lifetime and frequency. Time-based aging can be defined for all cache group types in TimesTen.

The data lifetime defines the minimum age of data within the table. The comparison of the time is based on the timestamp, so data may not become a candidate for aging until longer than the specified lifetime (but never less that the specified lifetime).

Define a time-based aging policy for a cache group by using the `AGING USE` clause in the cache table definition of the `CREATE CACHE GROUP` statement. Aging occurs automatically if the aging state is set to its default of `ON`.

On the Oracle Database:

The following example are the definitions of the Oracle Database tables that are to be cached in the AWT cache group. The Oracle Database tables are owned by the schema user `sales`.

```
CREATE TABLE orders
(ord_num      NUMBER(10) NOT NULL PRIMARY KEY,
 cust_num     NUMBER(6) NOT NULL,
 when_placed  DATE NOT NULL,
 when_shipped DATE NOT NULL);

CREATE TABLE order_item
(orditem_id NUMBER(12) NOT NULL PRIMARY KEY,
 ord_num    NUMBER(10),
 prod_num   VARCHAR2(6),
 quantity   NUMBER(3));
```

The Oracle cache administration user must be granted the `SELECT`, `INSERT`, `UPDATE`, and `DELETE` privileges on any cached tables. In this example, the table is the `sales.orders` and `sales.order_item` tables.

See Required Privileges for Cache Administration User for Cache Operations for all required privileges for different activities.

On the Oracle database as an administrator, grant the following privileges:

```
SQL> GRANT SELECT, INSERT, UPDATE, DELETE ON sales.orders TO cacheadmin;
SQL> GRANT SELECT, INSERT, UPDATE, DELETE ON sales.order_item TO cacheadmin;
```

On the TimesTen database:

The following example defines a time-based aging policy on the AWT cache group `ordered_items`:

```
CREATE ASYNCHRONOUS WRITETHROUGH CACHE GROUP ordered_items
FROM sales.orders
 (ord_num      NUMBER(10) NOT NULL,
  cust_num     NUMBER(6) NOT NULL,
  when_placed  DATE NOT NULL,
  when_shipped DATE NOT NULL,
  PRIMARY KEY(ord_num))
AGING USE when_placed LIFETIME 45 DAYS CYCLE 60 MINUTES ON,
sales.order_item
 (orditem_id NUMBER(12) NOT NULL,
  ord_num    NUMBER(10),
  prod_num   VARCHAR2(6),
  quantity   NUMBER(3),
  PRIMARY KEY(orditem_id),
  FOREIGN KEY(ord_num) REFERENCES sales.orders(ord_num));
```

Cache instances that are greater than 45 days old based on the difference between the current system timestamp and the timestamp in the `when_placed` column of the `sales.orders` table are candidates for aging. The aging process checks every 60 minutes to see if there are cache instances that can be automatically aged out or deleted from the cache tables.

The `AGING USE` clause requires the name of a non-nullable `TIMESTAMP` or `DATE` column used for time-based aging. We refer to this column as the timestamp column.

For each row, the value in the timestamp column stores the date and time when the row was most recently inserted or updated. The values in the timestamp column is maintained by your application. If the value of this column is unknown for particular rows and you do not want those rows to be aged out of the table, define the timestamp column with a large default value.

You can create an index on the timestamp column to optimize performance of the aging process.

You cannot add a column to an existing table and then use that column as the timestamp column because added columns cannot be defined as non-nullable. You cannot drop the timestamp column from a table that has a time-based aging policy defined.

Specify the lifetime in days, hours, minutes or seconds after the `LIFETIME` keyword in the `AGING USE` clause.

The value in the timestamp column is subtracted from the current system timestamp. The result is then truncated to the specified lifetime unit (day, hour, minute, second) and compared with the specified lifetime value. If the result is greater than the lifetime value, the row is a candidate for aging.

After the `CYCLE` keyword, specify the frequency in which aging occurs in days, hours, minutes or seconds. The default aging cycle is 5 minutes. If you specify an aging cycle of 0, aging is continuous.

The `ALTER TABLE` statement can be used to perform the following tasks associated with changing or defining a time-based aging policy on a cache group:

- Change the aging state of a cache group by specifying the root table and using the `SET AGING` clause.
- Change the lifetime by specifying the root table and using the `SET AGING LIFETIME` clause.
- Change the aging cycle by specifying the root table and using the `SET AGING CYCLE` clause.
- Add a time-based aging policy to a cache group that has no aging policy defined by specifying the root table and using the `ADD AGING USE` clause.
- Drop the time-based aging policy on a cache group by specifying the root table and using the `DROP AGING` clause.

To change the aging policy of a cache group from time-based to LRU, use an `ALTER TABLE` statement on the root table with the `DROP AGING` clause to drop the time-based aging policy. Then use an `ALTER TABLE` statement on the root table with the `ADD AGING LRU` clause to add an LRU aging policy.

You must stop the cache agent before you add, alter or drop an aging policy on a cache group with autorefresh.

Manually Scheduling an Aging Process in TimesTen

Use the `ttAgingScheduleNow` built-in procedure to manually start a one-time aging process on a specified table or on all tables that have an aging policy defined.

The aging process starts as soon as you call the built-in procedure unless there is already an aging process in progress. Otherwise the manually started aging process begins when the aging process that is in progress has completed. After the manually started aging process has completed, the start of the table's next aging cycle is set to the time when `ttAgingScheduleNow` was called if the table's aging state is `ON`.

The following example shows how the `ttAgingScheduleNow` built-in procedure starts a one-time aging process on the `sales.orders` table based on the time `ttAgingScheduleNow` is called:

```
Command> CALL ttAgingScheduleNow('sales.orders');
```

Rows in the `sales.orders` root table that are candidates for aging are deleted as well as related rows in the `sales.order_item` child table.

When you call the `ttAgingScheduleNow` built-in procedure, the aging process starts regardless of whether the table's aging state is `ON` or `OFF`. If you want to start an aging process on a particular cache group, specify the name of the cache group's root table when you call the built-in procedure. If the `ttAgingScheduleNow` built-in procedure is called with no parameters, it starts an aging process and then resets the start of the next aging cycle on all tables in the TimesTen database that have an aging policy defined.

Calling the `ttAgingScheduleNow` built-in procedure does not change the aging state of any table. If a table's aging state is `OFF` when you call the built-in procedure, the aging process starts, but it is not scheduled to run again after the process has completed. To continue aging a table whose aging state is `OFF`, you must call `ttAgingScheduleNow` again or change the table's aging state to `ON`.

To manually control aging on a cache group, disable aging on the root table by using an `ALTER TABLE` statement with the `SET AGING OFF` clause. Then call `ttAgingScheduleNow` to start an aging process on the cache group.

Configuring a Sliding Window in TimesTen

You can use time-based aging to implement a sliding window for a cache group.

In a sliding window configuration, new rows are inserted into and old rows are deleted from the cache tables on a regular schedule so that the tables contain only the data that satisfies a specific time interval.

You can configure a sliding window for a cache group by using incremental autorefresh mode and defining a time-based aging policy. The autorefresh operation checks the timestamp of the rows in the cached Oracle Database tables to determine whether new data should be refreshed into the TimesTen cache tables. The system time and the time zone must be identical on the Oracle Database and TimesTen systems.

If the cache group does not use incremental autorefresh mode, you can configure a sliding window by using a `LOAD CACHE GROUP`, `REFRESH CACHE GROUP`, or `INSERT` statement, or a dynamic load operation to bring new data into the cache tables.

The following example configures a sliding window on the read-only cache group `recent_shipped_orders`:

```
CREATE READONLY CACHE GROUP recent_shipped_orders
AUTOREFRESH MODE INCREMENTAL INTERVAL 1440 MINUTES STATE ON
FROM sales.orders
 (ord_num      NUMBER(10) NOT NULL,
  cust_num     NUMBER(6) NOT NULL,
  when_placed  DATE NOT NULL,
  when_shipped DATE NOT NULL,
  PRIMARY KEY(ord_num))
AGING USE when_shipped LIFETIME 30 DAYS CYCLE 24 HOURS ON;
```

New data in the `sales.orders` cached Oracle Database table are automatically refreshed into the `sales.orders` TimesTen cache table every 1440 minutes. Cache instances that are greater than 30 days old based on the difference between the current system timestamp and the timestamp in the `when_shipped` column are candidates for aging. The aging process checks every 24 hours to see if there are cache instances that can be aged out of the cache tables. Therefore, this cache group stores orders that have been shipped within the last 30 days.

The autorefresh interval and the lifetime used for aging determine the duration that particular rows remain in the cache tables. It is possible for data to be aged out of the cache tables before it has been in the cache tables for its lifetime. For example, for a read-only cache group if the autorefresh interval is 3 days and the lifetime is 30 days, data that is already 3 days old when it is refreshed into the cache tables is deleted after 27 days because aging is based on the timestamp stored in the rows of the cached Oracle Database tables that gets loaded into the TimesTen cache tables, not when the data is refreshed into the cache tables.

Replicating Cache Tables in TimesTen

To achieve high availability in TimesTen, configure an active standby pair replication scheme for cache tables in a read-only cache group or an AWT cache group.

An active standby pair that replicates cache tables from one of these cache group types can automatically change the role of a TimesTen database as part of failover and recovery with minimal chance of data loss. Cache groups themselves provide resilience from Oracle database outages, further strengthening system availability. An active standby pair replication scheme provides for high availability of a TimesTen database.

Note

This section describes one scenario in including cache groups within an active standby pair replication scheme. See Administering an Active Standby Pair with Cache Groups in *Oracle TimesTen In-Memory Database Replication Guide* for more scenarios for including AWT and read-only cache groups in an active standby pair replication scheme.

Oracle Real Application Clusters (Oracle RAC) provides for high availability of an Oracle database. See Using Cache in an Oracle RAC Environment.

Perform the following tasks to configure an active standby pair for TimesTen databases that cache Oracle Database tables:

- Create and Configure the Active Database
- Create and Configure the Standby Database
- Create and Configure the Read-Only Subscriber Database

Create and Configure the Active Database

This example shows how to create and configure the active database in an active standby pair replication scheme.

The following is the definition of the `cacheactive` DSN for the active database of the active standby pair:

```
[cacheactive]
DataStore=/users/OracleCache/cacheact
PermSize=64
OracleNetServiceName=orcl
DatabaseCharacterSet=WE8ISO8859P1
CacheAdminWallet=1
```

Note

If you set the `CacheAdminWallet` as a first connection attribute (normally set in the DSN), then when you register the cache administration user credentials with the `ttCacheUidPwdSet` built-in procedure, they are stored in an Oracle Wallet rather than in memory.

Start the ttIsql utility and connect to the cacheactive DSN as the instance administrator to create the database. Then create the TimesTen cache administration user cacheadmin whose name is the same as the Oracle cache administration user.

Then, create a cache table user sales whose name is the same as the Oracle Database schema user who owns the Oracle Database tables to be cached in the TimesTen database.

```
% ttIsql cacheactive
Command> CREATE USER cacheadmin IDENTIFIED BY timesten;
Command> CREATE USER sales IDENTIFIED BY timesten;
```

As the instance administrator, use the ttIsql utility to grant the TimesTen cache administration user cacheadmin the privileges required as well as create an active standby pair replication scheme which requires the ADMIN privilege:

```
Command> GRANT CREATE SESSION, CACHE_MANAGER,
        CREATE ANY TABLE, ADMIN TO cacheadmin;
Command> exit
```

Start the ttIsql utility and connect to the cacheactive DSN as the TimesTen cache administration user. Set the Oracle cache administration user name and password by calling the ttCacheUidPwdSet built-in procedure.

```
% ttIsql "DSN=cacheactive;UID=cacheadmin;PwdWallet=/wallets/cacheadminwallet"
Command> CALL ttCacheUidPwdSet('cacheadmin','orapwd');
```

If desired, you can test the connectivity between the active database and the Oracle database using the instructions stated in Testing the Connectivity Between the TimesTen and Oracle Databases.

Start the cache agent on the active database by calling the ttCacheStart built-in procedure as the TimesTen cache administration user:

```
Command> CALL ttCacheStart;
```

The following statement is the definition of the Oracle Database table that is to be cached in a dynamic AWT cache group. The Oracle Database table is owned by the schema user sales.

```
CREATE TABLE subscriber
(subscriberid       NUMBER(10) NOT NULL PRIMARY KEY,
 name               VARCHAR2(100) NOT NULL,
 minutes_balance    NUMBER(5) NOT NULL,
 last_call_duration NUMBER(4) NOT NULL);
```

The Oracle cache administration user must be granted the SELECT privilege on the sales.subscriber table so that the TimesTen cache administration user can create an AWT cache group that caches this table. The Oracle cache administration user must be granted the INSERT, UPDATE and DELETE Oracle Database privileges for the sales.subscriber table for asynchronous writethrough operations to be applied to the Oracle Database.

Then, create cache groups in the TimesTen database with the CREATE DYNAMIC ASYNCHRONOUS WRITETHROUGH CACHE GROUP statement as the TimesTen cache administration user. For example, the following statement creates a dynamic AWT cache group subscriber_accounts that caches the sales.subscriber table:

```
CREATE DYNAMIC ASYNCHRONOUS WRITETHROUGH CACHE GROUP subscriber_accounts
FROM sales.subscriber
 (subscriberid       NUMBER(10) NOT NULL PRIMARY KEY,
  name               VARCHAR2(100) NOT NULL,
  minutes_balance    NUMBER(5) NOT NULL,
  last_call_duration NUMBER(4) NOT NULL);
```

As the TimesTen cache administration user, create an active standby pair replication scheme in the active database using a `CREATE ACTIVE STANDBY PAIR` statement.

In the following example, `cacheact`, `cachestand` and `subscr` are the file name prefixes of the checkpoint and transaction log files of the active database, standby database and read-only subscriber database. `sys3`, `sys4` and `sys5` are the host names of the TimesTen systems where the active database, standby database and read-only subscriber database reside, respectively.

```
Command> CREATE ACTIVE STANDBY PAIR cacheact ON "sys3", cachestand ON "sys4"
        SUBSCRIBER subscr ON "sys5";
```

As the TimesTen cache administration user, start the replication agent on the active database by calling the `ttRepStart` built-in procedure. Then declare the database as the active by calling the `ttRepStateSet` built-in procedure.

```
Command> CALL ttRepStart;
Command> CALL ttRepStateSet('active');
```

Create and Configure the Standby Database

This example shows how to create and configure a standby database in an active standby pair replication scheme.

The following is the definition of the `cachestandby` DSN for the standby database of the active standby pair:

```
[cachestandby]
DataStore=/users/OracleCache/cachestand
PermSize=64
OracleNetServiceName=orcl
DatabaseCharacterSet=WE8ISO8859P1
CacheAdminWallet=1
```

Note

If you set the `CacheAdminWallet` as a first connection attribute (normally set in the DSN), then when you register the cache administration user credentials with the `ttCacheUidPwdSet` built-in procedure, they are stored in an Oracle Wallet rather than in memory.

As the instance administrator, create the standby database as a duplicate of the active database by running a `ttRepAdmin -duplicate` utility command from the standby database system. The instance administrator user name of the active database's and standby database's instances must be identical.

Use the `-keepCG` option so that cache tables in the active database are duplicated as cache tables in the standby database, because the standby database is connected with the Oracle database.

In the following example:

- The `-from` option specifies the file name prefix of the active database's checkpoint and transaction log files.
- The `-host` option specifies the host name of the TimesTen system where the active database resides.

- The `-uid` and `-pwd` options specify a user name and password of a TimesTen internal user defined in the active database that has been granted the `ADMIN` privilege.
- The `-cacheuid` and `-cachepwd` options specify the Oracle cache administration user name and password.
- `cachestandby` is the DSN of the standby database.
- The `-keepCG` option specifies that the standby database keeps the cache groups defined on the active database.

```
% ttRepAdmin -duplicate -from cacheact -host "sys3" -uid cacheadmin -pwd timesten
    -cacheuid cacheadmin -cachepwd orapwd -keepCG cachestandby
```

Start the `ttIsql` utility and connect to the `cachestandby` DSN as the cache administration user. Set the Oracle cache administration user name and password by calling the `ttCacheUidPwdSet` built-in procedure.

```
% ttIsql "DSN=cachestandby;UID=cacheadmin;PwdWallet=/wallets/cacheadminwallet"
Command> CALL ttCacheUidPwdSet('cacheadmin','orapwd');
```

If desired, you can test the connectivity between the standby database and the Oracle database using the instructions stated in Testing the Connectivity Between the TimesTen and Oracle Databases.

Start the cache agent on the standby database by calling the `ttCacheStart` built-in procedure as the TimesTen cache administration user:

```
Command> CALL ttCacheStart;
```

As the TimesTen cache administration user, start the replication agent on the standby database by calling the `ttRepStart` built-in procedure.

```
Command> CALL ttRepStart;
```

Create and Configure the Read-Only Subscriber Database

This example demonstrates how to create and configure a read-only subscriber within an active standby pair replication scheme.

The following is the definition of the `rosubscriber` DSN for the read-only subscriber database of the active standby pair:

```
[rosubscriber]
DataStore=/users/OracleCache/subscr
PermSize=64
DatabaseCharacterSet=WE8ISO8859P1
```

As the instance administrator, create the read-only subscriber database as a duplicate of the standby database by running a `ttRepAdmin -duplicate` utility command from the read-only subscriber database system. The instance administrator user name of the standby database and read-only subscriber database must be identical.

Use the `-noKeepCG` option so that cache tables in the standby database are duplicated as regular tables in the read-only subscriber database because the read-only subscriber database is not connected with the Oracle database.

In the following example:

- The `-from` option specifies the file name prefix of the standby database's checkpoint and transaction log files.

- The `-host` option specifies the host name of the TimesTen system where the standby database resides.
- The `-uid` and `-pwd` options specify a user name and password of a TimesTen internal user defined in the standby database that has been granted the `ADMIN` privilege.
- `rosubscriber` is the DSN of the read-only subscriber database.

```
% ttRepAdmin -duplicate -from cachestand -host "sys4" -uid cacheadmin -pwd timesten
    -noKeepCG rosubscriber
```

As the TimesTen cache administration user, start the replication agent on the read-only subscriber database by calling the `ttRepStart` built-in procedure.

```
% ttIsql "DSN=rosubscriber;UID=cacheadmin;PwdWallet=/wallets/cacheadminwallet"
Command> CALL ttRepStart;
Command> exit
```

Online Defragmentation of a TimesTen Database

Online defragmentation is the process to defragment the database to make memory available for the existing tables. When the online defragmentation feature is enabled, it defragments not only the regular database tables but also cache tables, including those within autorefresh cache groups.

Defragmentation of cache tables in autorefresh cache groups happens after each autorefresh cycle. All standby cache tables are defragged as if they are regular user tables. Therefore, the defragmentation of these tables is governed by the defrag settings (`DefragThreshold`, `DefragSpeed`, and `DefragCycle`). If the standby database is promoted to the active database, its cache tables are defragmented after each autorefresh cycle. Defragmentation of Asynchronous WriteThrough (AWT), Synchronous writethrough (SWT), and User managed cache tables occurs when the background defragmentation process determines it is necessary. For details on how to configure the online defragmentation connection attributes, see Online Defragmentation of TimesTen Databases in the *Oracle TimesTen In-Memory Database Operations Guide*.

For user managed cache groups, manual refresh operation can run into lock conflicts if the `LockWait=0`. To prevent lock conflicts between the manual refresh of user managed cache groups and defragmentation process, set the application's `LockWait` value greater than `0`. For details on how to configure `LockWait` attribute, see LockWait in the *Oracle TimesTen In-Memory Database Reference*.

If you set `DefragSpeed` to a non-zero value, it enables the defragmentation for the cache group's tables.

If you set `DefragSpeed=0`, it disables the defragmentation of the database, including the cache tables.

The `DefragCycle` setting does not affect when defragmentation is performed on read-only autorefresh cache groups. `DefragCycle` only affects tables that are not read-only cache tables or autorefresh enabled cache tables. For read-only dynamic cache groups, autorefresh is enabled, which causes the cache agent to perform defragmentation after each autorefresh cycle. See DefragThreshold, DefragSpeed, and DefragCycle in the *Oracle TimesTen In-Memory Database Reference*.

5

Methods for Transmitting Changes Between TimesTen and Oracle Databases

You can transmit changes between TimesTen and Oracle databases manually or automatically.

- Manually load cache groups: You can manually load or refresh cache instances into the TimesTen cache tables from the Oracle database tables using `LOAD CACHE GROUP` or `REFRESH CACHE GROUP` statements.
- Manually propagate committed changes: Use a `FLUSH CACHE GROUP` statement to propagate committed changes on the TimesTen cache tables to the cached Oracle Database tables.
- Automatically refresh cache groups: You can cause the cache instances to be automatically refreshed with the `AUTOREFRESH` cache table attribute. Automatic refresh can be defined on cache groups that are either explicitly or dynamically loaded.
- Dynamically load data on demand: When you define a cache group with the `DYNAMIC` keyword, then the data in a cache group is dynamically loaded on demand.
- Automatic propagation of changes to the Oracle database: When you configure the `PROPAGATE` cache table attribute on the TimesTen cache tables, then committed changes are automatically propagated to the cached Oracle Database tables.

See Transmitting Changes Between the TimesTen and Oracle Databases for an overview of each of these methods.

Note

You can use SQL statements or SQL Developer to perform most of the operations in this chapter. For more information about SQL Developer, see *Oracle TimesTen In-Memory Database SQL Developer Support User's Guide*.

The following sections describe these operations:

- Manually Loading and Refreshing a Cache Group
- Flushing a User Managed Cache Group
- Unloading a Cache Group
- Automatically Refreshing a Cache Group
- Manually or Dynamically Loading Cache Groups
- Dynamic Cache Groups
- Determining the Number of Cache Instances Affected by an Operation
- Setting a Passthrough Level

Manually Loading and Refreshing a Cache Group

You can manually insert or update cache instances into the TimesTen cache tables from the cached Oracle Database tables using either a `LOAD CACHE GROUP` or `REFRESH CACHE GROUP` statement.

The differences between loading and refreshing a cache group are:

- The `LOAD CACHE GROUP` statement only loads committed inserts on the cached Oracle Database tables into the TimesTen cache tables. New cache instances are loaded into the cache tables, but cache instances that already exist in the cache tables are not updated or deleted even if the corresponding rows in the cached Oracle Database tables have been updated or deleted. A load operation is primarily used to initially populate a cache group.
- The `REFRESH CACHE GROUP` statement replaces cache instances in the TimesTen cache tables with the most current data from the cached Oracle Database tables including cache instances that are already exist in the cache tables. A refresh operation is primarily used to update the contents of a cache group with committed changes on the cached Oracle Database tables after the cache group has been initially populated.

  For a static cache group, a refresh operation is equivalent to issuing an `UNLOAD CACHE GROUP` statement followed by a `LOAD CACHE GROUP` statement on the cache group. In effect, all committed inserts, updates and deletes on the cached Oracle Database tables are refreshed into the cache tables. New cache instances may be loaded into the cache tables. Cache instances that already exist in the cache tables are updated or deleted if the corresponding rows in the cached Oracle Database tables have been updated or deleted. See Unloading a Cache Group for more information about the `UNLOAD CACHE GROUP` statement.

  For a dynamic cache group, a refresh operation only refreshes committed updates and deletes on the cached Oracle Database tables into the cache tables because only existing cache instances in the cache tables are refreshed. New cache instances are not loaded into the cache tables so after the refresh operation completes, the cache tables contain either the same or fewer number of cache instances. To load new cache instances into the cache tables of a dynamic cache group, use a `LOAD CACHE GROUP` statement or perform a dynamic load operation. See Dynamic Cache Groups for more information about a dynamic load operation.

For most cache group types, you can use a `WHERE` clause in a `LOAD CACHE GROUP` or `REFRESH CACHE GROUP` statement to restrict the rows to be loaded or refreshed into the cache tables.

If the cache table definitions use a `WHERE` clause, only rows that satisfy the `WHERE` clause are loaded or refreshed into the cache tables even if the `LOAD CACHE GROUP` or `REFRESH CACHE GROUP` statement does not use a `WHERE` clause.

If the cache group has a time-based aging policy defined, only cache instances where the timestamp in the root table's row is within the aging policy's lifetime are loaded or refreshed into the cache tables.

To prevent a load or refresh operation from processing a large number of cache instances within a single transaction, which can greatly reduce concurrency and throughput, use the `COMMIT EVERY` *n* `ROWS` clause to specify a commit frequency unless you are using the `WITH ID` clause. If you specify `COMMIT EVERY 0 ROWS`, the load or refresh operation is processed in a single transaction.

In addition, if the load operation is for a large amount of data, use parallelism to increase throughput by specifying the number of threads with the `PARALLEL` clause.

A `LOAD CACHE GROUP` or `REFRESH CACHE GROUP` statement that uses the `COMMIT EVERY` *n* `ROWS` clause must be performed in its own transaction without any other operations within the same transaction.

The following example loads new cache instances into the TimesTen cache tables in the `customer_orders` cache group from the cached Oracle Database tables:

```
LOAD CACHE GROUP customer_orders COMMIT EVERY 256 ROWS PARALLEL 2;
```

The following example loads into the TimesTen cache tables using a `WHERE` clause in the `new_customers` cache group from the cached Oracle Database tables. The `WHERE` clause specifies new cache instances for customers whose customer number is greater than or equal to 5000:

```
LOAD CACHE GROUP new_customers WHERE (sales.customer.cust_num >= 5000)
  COMMIT EVERY 256 ROWS;
```

The following example refreshes cache instances in the TimesTen cache tables within the `top_products` cache group from the cached Oracle Database tables:

```
REFRESH CACHE GROUP top_products COMMIT EVERY 256 ROWS;
```

The following example refreshes in the TimesTen cache tables within the `update_anywhere_customers` cache group from the cached Oracle Database tables. The `WHERE` clause specifies cache instances of customers located in zip code 60610:

```
REFRESH CACHE GROUP update_anywhere_customers
  WHERE (sales.customer.zip = '60610') COMMIT EVERY 256 ROWS;
```

See LOAD CACHE GROUP and REFRESH CACHE GROUP in *Oracle TimesTen In-Memory Database SQL Reference*.

The rest of this section includes these topics:

- Loading and Refreshing a Cache Group Using a WITH ID Clause
- Loading and Refreshing a Multiple-Table Cache Group
- Improving the Performance of Loading or Refreshing a Large Number of Cache Instances
- Example of Manually Loading and Refreshing a Static Cache Group
- Example of Manually Loading and Refreshing a Dynamic Cache Group

Loading and Refreshing a Cache Group Using a WITH ID Clause

The `WITH ID` clause of the `LOAD CACHE GROUP` or `REFRESH CACHE GROUP` statement enables you to load or refresh a cache group based on values of the primary key columns without having to use a `WHERE` clause.

The `WITH ID` clause is more convenient than the equivalent `WHERE` clause if the primary key contains more than one column. Using the `WITH ID` clause allows you to load one cache instance at a time. It also enables you to roll back the transaction containing the load or refresh operation, if necessary, unlike the equivalent statement that uses a `WHERE` clause because using a `WHERE` clause also requires specifying a `COMMIT EVERY` *n* `ROWS` clause.

The following example loads a cache group using a `WITH ID` clause. A cache group `recent_orders` contains a single cache table `sales.orderdetails` with a primary key of `(orderid, itemid)`. If a customer calls about an item within a particular order, the information can be obtained by loading the cache instance for the specified order number and item number.

Load the `sales.orderdetails` cache table in the `recent_orders` cache group with the row whose value in the `orderid` column of the `sales.orderdetails` cached Oracle Database table is 1756 and its value in the `itemid` column is 573:

```
LOAD CACHE GROUP recent_orders WITH ID (1756,573);
```

The following is an equivalent `LOAD CACHE GROUP` statement that uses a `WHERE` clause:

```
LOAD CACHE GROUP recent_orders WHERE orderid = 1756 and itemid = 573
  COMMIT EVERY 256 ROWS;
```

A `LOAD CACHE GROUP` or `REFRESH CACHE GROUP` statement issued on a cache group with autorefresh cannot contain a `WITH ID` clause unless the cache group is dynamic.

You cannot use the `COMMIT EVERY` *n* `ROWS` clause with the `WITH ID` clause.

Loading and Refreshing a Multiple-Table Cache Group

If you are loading or refreshing a multiple-table cache group while the cached Oracle Database tables are concurrently being updated, set the isolation level in the TimesTen database to serializable before issuing the `LOAD CACHE GROUP` or `REFRESH CACHE GROUP` statement.

This causes TimesTen to query the cached Oracle Database tables in a serializable fashion during the load or refresh operation so that the loaded or refreshed cache instances in the cache tables are guaranteed to be transactionally consistent with the corresponding rows in the cached Oracle Database tables. After you have loaded or refreshed the cache group, set the isolation level back to read committed for better concurrency when accessing elements in the TimesTen database.

Improving the Performance of Loading or Refreshing a Large Number of Cache Instances

You can improve the performance of loading or refreshing a large number of cache instances into a cache group by specifying the operation to be multithreaded with the `PARALLEL` clause of the `LOAD CACHE GROUP` or `REFRESH CACHE GROUP` statement.

If you do not specify the `PARALLEL` clause, the load or refresh operation will be single-threaded. Specifying the `PARALLEL` clause to create multithreaded processes for a load or refresh provides a performance benefit if you have large data sets to be loaded or if the round trip time to Oracle is large. For example, if the data in the Oracle database is large, then an initial full load of the cache group can prove to be time consuming. Specifying multiple threads for the initial load can improve performance for that operation. However, note that multithreaded processes require more time to initiate than a single-threaded process and multithreaded processes use more system resources.

Specify the number of threads to use when processing the load or refresh operation. You can specify 2 to 10 threads. Do not specify more threads than the number of CPUs available on your system or you may encounter decreased performance than if you had not used the `PARALLEL` clause.

Note

There is no default for the `PARALLEL` clause.

You cannot use the `WITH ID` clause with the `PARALLEL` clause. You can use the `COMMIT EVERY n ROWS` clause with the `PARALLEL` clause as long as `n` is greater than 0. In addition, you cannot use the `PARALLEL` clause for dynamic read-only cache groups or when database level locking is enabled. See REFRESH CACHE GROUP in the *Oracle TimesTen In-Memory Database SQL Reference*.

The following example refreshes cache instances in the TimesTen cache tables using a `PARALLEL` clause. This example refreshes the `western_customers` cache group from the cached Oracle Database tables using one thread to fetch rows from the cached Oracle Database tables and one thread to insert the rows into the cache tables:

```
LOAD CACHE GROUP western_customers COMMIT EVERY 256 ROWS PARALLEL 2;
```

The number of threads that you specify with the `PARALLEL` clause are assigned to readers and inserters. TimesTen recommends to have the number of readers >= number of inserters, because readers are slower than inserters. By default, only one thread is assigned to a reader to fetch rows from the cached Oracle Database tables. Since there is only one reader by default, then TimesTen assigns only one of the other threads as an inserter to insert the rows into the TimesTen cache tables. Using more inserters for a single reader offers no benefit.

If you want to specify more than 2 threads, use both the `PARALLEL` clause with the `READERS` clause to specify the number of readers to assign. The number of inserters assigned is the number of parallel threads minus the number of readers.

```
LOAD CACHE GROUP western_customers COMMIT EVERY 256 ROWS
                          PARALLEL 6 READERS 4;
```

This specifies 6 threads with 4 of the threads assigned to readers and 2 threads assigned to inserters.

Example of Manually Loading and Refreshing a Static Cache Group

This example shows the definition of an Oracle Database table that is to be cached in a static AWT cache group.

On the Oracle database:

The Oracle Database table is owned by the schema user `sales`.

```
CREATE TABLE customer
(cust_num NUMBER(6) NOT NULL PRIMARY KEY,
 region   VARCHAR2(10),
 name     VARCHAR2(50),
 address  VARCHAR2(100));
```

The following is the data in the `sales.customer` cached Oracle Database table.

```
CUST_NUM   REGION    NAME              ADDRESS
--------   -------   ---------------   ---------------------------
       1   West      Frank Edwards     100 Pine St. Portland OR
       2   East      Angela Wilkins    356 Olive St. Boston MA
       3   Midwest   Stephen Johnson   7638 Walker Dr. Chicago IL
```

On the TimesTen database, connect as the TimesTen cache administration user. The following statement creates a static AWT cache group `new_customers` that caches the `sales.customer` table:

```
CREATE ASYNCHRONOUS WRITETHROUGH CACHE GROUP new_customers
FROM sales.customer
 (cust_num NUMBER(6) NOT NULL,
  region   VARCHAR2(10),
  name     VARCHAR2(50),
  address  VARCHAR2(100),
  PRIMARY KEY(cust_num));
```

The `sales.customer` TimesTen cache table is initially empty.

```
Command> SELECT * FROM sales.customer;
0 rows found.
```

The following `LOAD CACHE GROUP` statement loads the three cache instances from the cached Oracle Database table into the TimesTen cache table:

```
Command> LOAD CACHE GROUP new_customers COMMIT EVERY 256 ROWS;
3 cache instances affected.
Command> SELECT * FROM sales.customer;
< 1, West, Frank Edwards, 100 Pine St. Portland OR >
< 2, East, Angela Wilkins, 356 Olive St. Boston MA >
< 3, Midwest, Stephen Johnson, 7638 Walker Dr. Chicago IL >
```

On the Oracle database, modify the cached Oracle Database table by inserting a new row, updating an existing row, and deleting an existing row:

```
SQL> INSERT INTO customer
  2  VALUES (4, 'East', 'Roberta Simon', '3667 Park Ave. New York NY');
SQL> UPDATE customer SET name = 'Angela Peterson' WHERE cust_num = 2;
SQL> DELETE FROM customer WHERE cust_num = 3;
SQL> COMMIT;
SQL> SELECT * FROM customer;
CUST_NUM   REGION    NAME              ADDRESS
--------   -------   ---------------   ---------------------------
       1   West      Frank Edwards     100 Pine St. Portland OR
       2   East      Angela Peterson   356 Olive St. Boston MA
       4   East      Roberta Simon     3667 Park Ave. New York NY
```

Back on the TimesTen database as the TimesTen cache administration, run a `REFRESH CACHE GROUP` statement on a static cache group, which is processed by unloading and then reloading the cache group. As a result, the cache instances in the cache table matches the rows in the cached Oracle Database table.

```
Command> REFRESH CACHE GROUP new_customers COMMIT EVERY 256 ROWS;
3 cache instance affected.
Command> SELECT * FROM sales.customer;
< 1, West, Frank Edwards, 100 Pine St. Portland OR >
< 2, East, Angela Peterson, 356 Olive St. Boston MA >
< 4, East, Roberta Simon, 3667 Park Ave. New York NY >
```

Example of Manually Loading and Refreshing a Dynamic Cache Group

This example shows the definition of an Oracle Database table that is to be cached in a dynamic AWT cache group.

On the Oracle database, connect as the schema owner, `sales`.

The Oracle Database table is owned by the schema user `sales`.

```
CREATE TABLE customer
(cust_num NUMBER(6) NOT NULL PRIMARY KEY,
 region   VARCHAR2(10),
 name     VARCHAR2(50),
 address  VARCHAR2(100));
```

The following is the data in the `sales.customer` cached Oracle Database table.

```
CUST_NUM   REGION    NAME              ADDRESS
--------   -------   ---------------   ---------------------------
       1   West      Frank Edwards     100 Pine St. Portland OR
       2   East      Angela Wilkins    356 Olive St. Boston MA
       3   Midwest   Stephen Johnson   7638 Walker Dr. Chicago IL
```

On the TimesTen database, connect as the TimesTen cache administration user. The following statement creates a dynamic AWT cache group `new_customers` that caches the `sales.customer` table:

```
CREATE DYNAMIC ASYNCHRONOUS WRITETHROUGH CACHE GROUP new_customers
FROM sales.customer
 (cust_num NUMBER(6) NOT NULL,
  region   VARCHAR2(10),
  name     VARCHAR2(50),
  address  VARCHAR2(100),
  PRIMARY KEY(cust_num));
```

The `sales.customer` TimesTen cache table is initially empty:

```
Command> SELECT * FROM sales.customer;
0 rows found.
```

The following `LOAD CACHE GROUP` statement loads the three cache instances from the cached Oracle Database table into the TimesTen cache table:

```
Command> LOAD CACHE GROUP new_customers COMMIT EVERY 256 ROWS;
3 cache instances affected.
Command> SELECT * FROM sales.customer;
< 1, West, Frank Edwards, 100 Pine St. Portland OR >
< 2, East, Angela Wilkins, 356 Olive St. Boston MA >
< 3, Midwest, Stephen Johnson, 7638 Walker Dr. Chicago IL >
```

Back on the Oracle database, connect as the `sales` schema user and modify the cached Oracle Database table by inserting a new row, updating an existing row, and deleting an existing row:

```
SQL> INSERT INTO customer
  2  VALUES (4, 'East', 'Roberta Simon', '3667 Park Ave. New York NY');
SQL> UPDATE customer SET name = 'Angela Peterson' WHERE cust_num = 2;
SQL> DELETE FROM customer WHERE cust_num = 3;
SQL> COMMIT;
SQL> SELECT * FROM customer;
CUST_NUM   REGION    NAME              ADDRESS
--------   -------   ---------------   ---------------------------
       1   West      Frank Edwards     100 Pine St. Portland OR
       2   East      Angela Peterson   356 Olive St. Boston MA
       4   East      Roberta Simon     3667 Park Ave. New York NY
```

On the TimesTen database, a `REFRESH CACHE GROUP` statement issued on a dynamic cache group only refreshes committed updates and deletes on the cached Oracle Database tables into the cache tables. New cache instances are not loaded into the cache tables. Therefore, only existing cache instances are refreshed. As a result, the number of cache instances in the cache tables are either fewer than or the same as the number of rows in the cached Oracle Database tables.

```
Command> REFRESH CACHE GROUP new_customers COMMIT EVERY 256 ROWS;
2 cache instances affected.
Command> SELECT * FROM sales.customer;
< 1, West, Frank Edwards, 100 Pine St. Portland OR >
< 2, East, Angela Peterson, 356 Olive St. Boston MA >
```

A subsequent `LOAD CACHE GROUP` statement loads one cache instance from the cached Oracle Database table into the TimesTen cache table because only committed inserts are loaded into the cache table. Therefore, only new cache instances are loaded. Cache instances that already exist in the cache tables are not changed because of a `LOAD CACHE GROUP` statement, even if the corresponding rows in the cached Oracle Database tables were updated or deleted.

```
Command> LOAD CACHE GROUP new_customers COMMIT EVERY 256 ROWS;
1 cache instance affected.
Command> SELECT * FROM sales.customer;
< 1, West, Frank Edwards, 100 Pine St. Portland OR >
< 2, East, Angela Peterson, 356 Olive St. Boston MA >
< 4, East, Roberta Simon, 3667 Park Ave. New York NY >
```

Flushing a User Managed Cache Group

The `FLUSH CACHE GROUP` statement manually propagates committed inserts and updates on TimesTen cache tables in a user managed cache group to the cached Oracle Database tables.

A flush operation can manually propagate multiple committed transactions on cache tables to the cached Oracle Database tables. This statement is available only for user managed cache groups. Delete operations are not flushed or manually propagated.

Automatic propagation is initiated when you use the `PROPAGATE` cache table attribute when defining your cache group. TimesTen then automatically propagates committed inserts, updates and deletes at commit time to the Oracle database in the order that they are committed in TimesTen.

You cannot flush a user managed cache group with the `FLUSH CACHE GROUP` statement that uses the `AUTOREFRESH` cache group attribute.

You can flush a user managed cache group with the `FLUSH CACHE GROUP` statement if at least one of its cache tables uses neither the `PROPAGATE` nor the `READONLY` cache table attribute.

You can use a `WHERE` clause or `WITH ID` clause in a `FLUSH CACHE GROUP` statement to restrict the rows to be flushed to the cached Oracle Database tables.

The following example flushes a cache group with the `FLUSH CACHE GROUP` statement. It manually propagates committed insert and update operations on the TimesTen cache tables in the `western_customers` cache group to the cached Oracle Database tables:

```
FLUSH CACHE GROUP western_customers;
```

See FLUSH CACHE GROUP in *Oracle TimesTen In-Memory Database SQL Reference*.

The following example creates a user managed cache group with the `PROPAGATE` cache table attribute:

```
CREATE USERMANAGED CACHE GROUP updateanywherecustomers
AUTOREFRESH
       MODE INCREMENTAL
       INTERVAL 30 SECONDS
       STATE ON
FROM
customer (custid INT NOT NULL,
       name CHAR(100) NOT NULL,
       addr CHAR(100),
       zip INT,
       PRIMARY KEY(custid),
       PROPAGATE);
```

See PROPAGATE Cache Table Attribute in this book and CREATE CACHE GROUP in the *Oracle TimesTen In-Memory Database SQL Reference*.

Unloading a Cache Group

You can delete some or all cache instances from the cache tables in a cache group with the `UNLOAD CACHE GROUP` statement.

Unlike the `DROP CACHE GROUP` statement, the cache tables themselves are not dropped when a cache group is unloaded.

To prevent an unload operation from processing a large number of cache instances within a single transaction, which could reduce concurrency and throughput, use the `COMMIT EVERY` *n* `ROWS` clause to specify a commit frequency.

Use caution when using the `UNLOAD CACHE GROUP` statement with cache groups with autorefresh. An unloaded row can reappear in the cache table as the result of an autorefresh operation if the row, or its related parent or child rows, are updated in the cached Oracle Database table.

Processing of the `UNLOAD CACHE GROUP` statement for an AWT cache group waits until updates on the rows have been propagated to the Oracle database.

The following example unloads all cache instances from all cache tables in the `customer_orders` cache group. A commit frequency is specified, so the operations is performed over several transactions by committing every 256 rows:

```
UNLOAD CACHE GROUP customer_orders COMMIT EVERY 256 ROWS;
```

The following statement unloads all cache instances from all cache tables in the `customer_orders` cache group in a single transaction. A single transaction should only be used if the data within `customer_orders` is small:

```
UNLOAD CACHE GROUP customer_orders;
```

The following equivalent statements delete the cache instance for customer number 227 from the cache tables in the `new_customers` cache group:

```
UNLOAD CACHE GROUP new_customers WITH ID (227);
UNLOAD CACHE GROUP new_customers WHERE (sales.customer.cust_num = 227);
```

See UNLOAD CACHE GROUP in the *Oracle TimesTen In-Memory Database SQL Reference*.

Automatically Refreshing a Cache Group

You can configure automatic refresh with the `AUTOREFRESH` cache group attribute.

- AUTOREFRESH Cache Group Attribute Overview

- Altering a Cache Group to Change the AUTOREFRESH Mode, Interval or State
- Manually Creating Oracle Database Objects for Cache Groups with Autorefresh
- Initiating an Immediate Autorefresh in TimesTen
- Disabling Full Autorefresh for Cache Groups
- Loading and Refreshing a Static Cache Group with Autorefresh
- Loading and Refreshing a Dynamic Cache Group with Autorefresh

AUTOREFRESH Cache Group Attribute Overview

The `AUTOREFRESH` cache group attribute can be specified when creating a read-only cache group or a user managed cache group using a `CREATE CACHE GROUP` statement.

`AUTOREFRESH` specifies that committed changes on cached Oracle Database tables are automatically refreshed to the TimesTen cache tables. Autorefresh is defined by default on read-only cache groups.

The `AUTOREFRESH` method determines how cache consistency is maintained either through trigger-based capture or GoldenGate's log-based capture. It allows TimesTen to automatically refresh the cache tables by tracking changes on the Oracle Database. This method can be used to manage cache group refreshes either through log-based capture or through triggers, based on how the cache is configured.

The following are the default settings of the autorefresh attributes:

- The autorefresh mode is incremental.
- The autorefresh interval is 5 minutes.
- The autorefresh state is `PAUSED`.
- The autorefresh method is trigger.

The following sections describe each of the autorefresh attributes:

- Autorefresh Mode Attribute Settings
- Autorefresh Interval and State Settings
- Restrictions for Autorefresh

If you create a unique index on a cache group with the `AUTOREFRESH` cache group attribute, the index is changed to a non-unique index to avoid a constraint violation. A constraint violation could occur with a unique index because conflicting updates could occur in the same statement processing on the Oracle Database table, while each row update is processed separately in TimesTen. If the unique index exists on the Oracle Database table that is being cached, then uniqueness is enforced on the Oracle Database table and does not need to be verified again in TimesTen.

Autorefresh Mode Attribute Settings

You can set the autorefresh mode to designate how the automatic refresh is to perform.

TimesTen supports two autorefresh mode settings:

- `FULL`: All cache tables are automatically refreshed, based on the cache group's autorefresh interval, by unloading all their rows and then reloading from the cached Oracle Database tables.

There is no overhead when using full autorefresh mode, but there may be performance implications.

- `INCREMENTAL`: Committed changes on cached Oracle Database tables are automatically refreshed to the TimesTen cache tables based on the cache group's autorefresh interval.

  There is overhead when using incremental autorefresh mode, but the performance is better than when using full autorefresh.

Some applications choose incremental autorefresh instead of full autorefresh mode for performance reasons. A full autorefresh can affect performance because:

- More rows are refreshed with a full autorefresh.
- A full autorefresh runs within a single transaction with no parallelism.

Even if you use incremental autorefresh on your cache group, the first load is a full refresh. In addition, TimesTen may perform a full autorefresh for recovery for certain error scenarios.

Note

You can disallow full autorefresh with the `DisableFullAutorefresh` cache configuration parameter. See Disabling Full Autorefresh for Cache Groups.

When using incremental autorefresh mode, committed changes on cached Oracle Database tables are tracked in change log tables in the Oracle database. Because incremental autorefresh tracks committed changes on the Oracle database, incremental autorefresh mode incurs some overhead to refresh the cache group for each committed update on the cached Oracle Database tables. Under certain circumstances, it is possible for some change log records to be deleted (truncated) from the change log table before they are automatically refreshed to the TimesTen cache tables. If this occurs, TimesTen initiates a full automatic refresh on the cache group.

See Managing the Cache Administration User's Tablespace for information on how to configure an action to take when the tablespace that the change log tables reside in becomes full.

See Managing a Cache Environment with Oracle Database Objects for information on the change log tables in the Oracle Database.

The change log table on the Oracle database does not have column-level resolution because of performance reasons. Thus, the autorefresh operation updates all of the columns in a row. XLA reports that all of the columns in the row have changed even if the data did not actually change in each column. See XLA and TimesTen Event Management in the *Oracle TimesTen In-Memory Database C Developer's Guide* or Using JMS/XLA for Event Management in the *Oracle TimesTen In-Memory Database Java Developer's Guide*.

If you have a dynamic read-only cache group with autorefresh, you can reduce contention and improve performance. See Reducing Contention for Dynamic Read-Only Cache Groups with Incremental Autorefresh and Reducing Lock Contention for Read-Only Cache Groups with Autorefresh and Dynamic Load and Options for Reducing Contention Between Autorefresh and Dynamic Load Operations.

Autorefresh Interval and State Settings

The autorefresh interval determines how often autorefresh operations occur in minutes, seconds or milliseconds.

Cache groups with the same autorefresh interval are refreshed within the same transaction. You can specify continuous autorefresh with an autorefresh interval of 0 milliseconds. With continuous autorefresh, the next autorefresh cycle is scheduled as soon as possible after the last autorefresh cycle has ended.

In TimesTen, you can manually initiate an immediate autorefresh operation with the `ttCacheAutorefresh` built-in procedure. See ttCacheAutorefresh in *Oracle TimesTen In-Memory Database Reference*.

The autorefresh state can be set to `ON`, `OFF`, or `PAUSED`.

- `ON`: Autorefresh operations are scheduled by TimesTen when the cache group's autorefresh state is `ON`.
- `OFF`: When the cache group's autorefresh state is `OFF`, committed changes on the cached Oracle Database tables are not tracked. When you change the state from `OFF` to `ON`, a full autorefresh is performed.
- `PAUSED`: When the cache group's autorefresh state is `PAUSED`, committed changes on the cached Oracle Database tables are tracked in the Oracle database, but are not automatically refreshed to the TimesTen cache tables until the state is changed to `ON`.

By default, a cache group is created with autorefresh state set to `PAUSED`. This provides you a choice of how and when the initial full load is performed.

- If the data in the Oracle database is large, then an initial full load of the cache group can prove to be time consuming. The recommended option is to run a manual load with parallelism with the `LOAD CACHE GROUP... PARALLEL` statement. The autorefresh state automatically changes to `ON` after the initial load completes.
- If the data on the Oracle database is small, change the state to `ON` with an `ALTER CACHE GROUP`. Changing the state to `ON` when an initial load has not yet been performed causes the initial load to be performed and autorefresh operations to start.

After the initial load is completed, you can change the state to `PAUSED` at any time. When you change the state to `ON`, then incremental autorefresh resumes for static cache groups that were created with incremental autorefresh.

If the data on the Oracle database is too large to perform an initial full load, you can disable all full load operations. See Disabling Full Autorefresh for Cache Groups.

Restrictions for Autorefresh

There are restrictions when using the `AUTOREFRESH` cache group attribute.

- A `FLUSH CACHE GROUP` statement cannot be issued on the cache group.

  See Flushing a User Managed Cache Group.
- A `TRUNCATE TABLE` statement issued on a cached Oracle Database table is not automatically refreshed to the TimesTen cache table. Before issuing a `TRUNCATE TABLE` statement on a cached Oracle Database table, use an `ALTER CACHE GROUP` statement to change the autorefresh state of the cache group that contains the cache table to `PAUSED`.

  See Altering a Cache Group to Change the AUTOREFRESH Mode, Interval or State.

  After issuing the `TRUNCATE TABLE` statement on the cached Oracle Database table, use a `REFRESH CACHE GROUP` statement to manually refresh the cache group.
- A `LOAD CACHE GROUP` statement can only be issued if the cache tables are empty, unless the cache group is dynamic.

See Manually Loading and Refreshing a Cache Group and Creating a Dynamic Cache Group with the DYNAMIC Keyword.

- The autorefresh state must be `PAUSED` before you can issue a `LOAD CACHE GROUP` statement on the cache group, unless the cache group is dynamic. If the cache group is dynamic, the autorefresh state must be `PAUSED` or `ON`.
- The `LOAD CACHE GROUP` statement cannot contain a `WHERE` clause, unless the cache group is dynamic. If the cache group is dynamic, the `WHERE` clause must be followed by a `COMMIT EVERY` *n* `ROWS` clause.

  See Using a WHERE Clause.
- The autorefresh state must be `PAUSED` before you can issue a `REFRESH CACHE GROUP` statement on the cache group. The `REFRESH CACHE GROUP` statement cannot contain a `WHERE` clause.
- All tables and columns referenced in `WHERE` clauses when creating, loading or unloading the cache group must be fully qualified. For example:

  owner.*table_name* and *owner*.*table_name*.*column_name*
- To use the `AUTOREFRESH` cache group attribute in a user managed cache group, all of the cache tables must be specified with the `PROPAGATE` cache table attribute or all of the cache tables must be specified the `READONLY` cache table attribute.
- You cannot specify the `AUTOREFRESH` cache group attribute in a user managed cache group that contains cache tables that explicitly use the `NOT PROPAGATE` cache table attribute.
- The `AUTOREFRESH` cache table attribute cannot be used when caching Oracle Database materialized views in a user managed cache group.
- LRU aging cannot be specified on the cache group, unless the cache group is dynamic where LRU aging is defined by default.

  See LRU Aging in TimesTen.
- If you want to use replication with a static cache group with autorefresh on TimesTen Classic, you can only use an active standby pair replication scheme. Any other type of replication scheme is not allowed with a static cache group with autorefresh on TimesTen Classic.

Altering a Cache Group to Change the AUTOREFRESH Mode, Interval or State

After creating a cache group with autorefresh, you can use `ALTER CACHE GROUP` statement to change autorefresh mode, interval or state.

In TimesTen, you can change the cache group's autorefresh mode, interval or state.

You cannot use `ALTER CACHE GROUP` to instantiate automatic refresh for a cache group that was originally created without autorefresh defined.

If you change a cache group's autorefresh state to `OFF` or drop a cache group that has an autorefresh operation in progress:

- The autorefresh operation stops if the setting of the `LockWait` connection attribute is greater than 0. The `ALTER CACHE GROUP` or `DROP CACHE GROUP` statement preempts the autorefresh operation.

- The autorefresh operation continues if the `LockWait` connection attribute is set to 0. The `ALTER CACHE GROUP` or `DROP CACHE GROUP` statement is blocked until the autorefresh operation completes or the statement fails with a lock timeout error.

The following example alters the autorefresh attributes of a cache group in TimesTen. These statements change the autorefresh mode, interval and state of the `customer_orders` cache group:

```
ALTER CACHE GROUP customer_orders SET AUTOREFRESH MODE FULL;
ALTER CACHE GROUP customer_orders SET AUTOREFRESH INTERVAL 30 SECONDS;
ALTER CACHE GROUP customer_orders SET AUTOREFRESH STATE ON;
```

TimesTen returns asynchronously after executing the `ALTER CACHE GROUP` statement. However, there may be a delay for the cache agent to implement the change for the new state, mode or interval.

Manually Creating Oracle Database Objects for Cache Groups with Autorefresh

There are certain procedures you need to do if you created the Oracle Database objects used to enforce the predefined behaviors of a cache group with autorefresh with the `initCacheAdminSchema.sql` script.

See The initCacheAdminSchema.sql Script.

1. Set the autorefresh state to `OFF` when creating the cache group.
2. Run the `ttIsql` utility's `cachesqlget` command with the `INCREMENTAL_AUTOREFRESH` option and the `INSTALL` flag as the TimesTen cache administration user. This command generates a SQL*Plus script used to create a cache log table and a trigger in the Oracle database for each Oracle Database table that is cached in the autorefresh cache group. These Oracle Database objects track updates on the cached Oracle Database tables so that the updates can be automatically refreshed to the cache tables.

   Note

   The `ttCacheSQLGet` built-in procedure provides the same functionality as the `ttIsql cachesqlget` command.

3. Use SQL*Plus to run the script generated by the `ttIsql` utility's `cachesqlget` command as the `sys` user.
4. Run an `ALTER CACHE GROUP` statement to change the autorefresh state of the cache group to `PAUSED`.

The following examples shows how to create a read-only cache group when Oracle Database objects are created with the `initCacheAdminSchema.sql` script.

The first statement creates a read-only cache group `customer_orders` with the autorefresh state set to `OFF`. The SQL*Plus script generated by the `ttIsql` utility's `cachesqlget` command is saved to the `/tmp/obj.sql` file. The last statement changes the autorefresh state of the cache group to `PAUSED`.

```
CREATE READONLY CACHE GROUP customer_orders
AUTOREFRESH STATE OFF
FROM sales.customer
 (cust_num NUMBER(6) NOT NULL,
  region   VARCHAR2(10),
  name     VARCHAR2(50),
  address  VARCHAR2(100),
  PRIMARY KEY(cust_num)),
sales.orders
 (ord_num      NUMBER(10) NOT NULL,
  cust_num     NUMBER(6) NOT NULL,
  when_placed  DATE NOT NULL,
  when_shipped DATE NOT NULL,
  PRIMARY KEY(ord_num),
  FOREIGN KEY(cust_num) REFERENCES sales.customer(cust_num));

% ttIsql "DSN=cache1;UID=cacheadmin;PwdWallet=/wallets/cacheadminwallet"
Command> cachesqlget INCREMENTAL_AUTOREFRESH customer_orders INSTALL /tmp/obj.sql;
Command> exit

% sqlplus sys as sysdba
Enter password: password
SQL> @/tmp/obj
SQL> exit

ALTER CACHE GROUP customer_orders SET AUTOREFRESH STATE PAUSED;
```

See ttIsql and ttCacheSqlGet in *Oracle TimesTen In-Memory Database Reference*.

Initiating an Immediate Autorefresh in TimesTen

In TimesTen, if the Oracle Database tables have been updated with data that needs to be applied to cache tables without waiting for the next autorefresh operation, you can call the `ttCacheAutorefresh` built-in procedure.

The `ttCacheAutorefresh` built-in procedure initiates an immediate refresh operation and resets the autorefresh cycle to start at the moment you invoke `ttCacheAutorefresh`.

The refresh operation is full or incremental depending on how the cache group is configured. The autorefresh state must be `ON` when `ttCacheAutorefresh` is called.

The autorefresh operation typically refreshes all cache groups sharing the same refresh interval in one transaction in order to preserve transactional consistency across these cache groups. Therefore, although you specify a specific cache group when you call `ttCacheAutorefresh`, the autorefresh operation occurs in one transaction for all cache groups that share the autorefresh interval with the specified cache group. If there is an existing transaction with table locks on objects that belong to the affected cache groups, `ttCacheAutofresh` returns an error without taking any action.

You can choose to run `ttCacheAutorefresh` asynchronously (the default) or synchronously. In synchronous mode, `ttCacheAutorefresh` returns an error if the refresh operation fails.

After calling `ttCacheAutorefresh`, you must commit or roll back the transaction before subsequent work can be performed.

This example calls `ttCacheAutorefresh` for the `ttuser.western_customers` cache group, using asynchronous mode.

```
Command> call ttCacheAutorefresh('ttuser', 'western_customers');
```

Disabling Full Autorefresh for Cache Groups

If performance is a concern, you can set the `DisableFullAutorefresh` cache configuration parameter to 1 to disallow full autorefresh requests for all cache groups defined with incremental autorefresh.

If you do disallow full autorefresh, then the initial load for each cache group requires a manual load since the initial load requires a full refresh.

You can disable full autorefresh using the `DisableFullAutorefresh` cache configuration parameter in TimesTen.

Note

The default value is `0` for the `DisableFullAutorefresh` cache configuration parameter, which specifies full autorefresh behavior. Full autorefresh is only supported on TimesTen.

```
call ttCacheConfig('DisableFullAutorefresh',,,'1');
```

You can query the current value of the `DisableFullAutorefresh` parameter.

```
call ttCacheConfig('DisableFullAutorefresh');
```

If a full autorefresh is triggered for a cache group, TimesTen changes the cache group status to `disabled`. After which, all autorefresh operations cease on the cache group. You are notified of this action with a message logged in both the daemon and user log files. See Error, Warning, and Informational Messages in the *Oracle TimesTen In-Memory Database Operations Guide*.

The TimesTen database status is set to `recovering` when at least one of its cache groups have an autorefresh status of `disabled` or `recovering`. You can check the state of a database and cache group with the `ttCacheDbCgStatus` built-in procedure. The following example shows that:

- `Recovering`: Some or all the cache groups with the `AUTOREFRESH` attribute in the database are being resynchronized with the Oracle database server. The status of at least one cache group is `recovering`.
- `Disabled`: The `cg1` cache group is `disabled`.

```
Command> call ttCacheDbCgStatus('ttuser','cg1');
< recovering, disabled >
1 row found.
```

When you set the `DisableFullAutorefresh` cache configuration parameter to 1, then the `DeadDbRecovery` cache configuration parameter automatically changes to `Manual`. TimesTen restores the original setting for the `DeadDbRecovery` cache configuration parameter if you change the `DisableFullAutorefresh` cache configuration parameter to 0.

If the autorefresh status of a cache group is either disabled or dead, its cache tables are no longer being automatically refreshed when updates are committed on the cached Oracle Database tables. The cache group must be recovered in order to resynchronize the cache tables with the cached Oracle Database tables.

- For each cache group whose autorefresh status is disabled, a `REFRESH CACHE GROUP` statement must be issued in order to resume autorefresh operations for these cache groups.
- For each dynamic cache group whose autorefresh status is disabled, an `UNLOAD CACHE GROUP` statement must be issued in order to resume autorefresh operations for these cache groups.
- See Impact of Failed Autorefresh Operations on TimesTen Databases for details on how to specify recovery when the autorefresh status of a cache group is `dead`.

The following example shows the steps to manually refresh a disabled cache group.

1. Pause autorefresh for the cache group and return the cache group status to `OK` with the `ALTER CACHE GROUP SET AUTOREFRESH STATE PAUSED` statement.
2. Manually request a full refresh with the `REFRESH CACHE GROUP` statement (optionally, with parallelism).

```
ALTER CACHE GROUP cg_static SET AUTOREFRESH STATE PAUSED;
REFRESH CACHE GROUP cg_static COMMIT EVERY 500 ROWS PARALLEL 2;
```

Perform the following to reload a dynamic cache group:

1. To return the cache group status to `OK`, pause autorefresh for the cache group with the `ALTER CACHE GROUP SET AUTOREFRESH STATE PAUSED` statement.
2. Unload the disabled dynamic cache group with the `UNLOAD CACHE GROUP` statement.
3. Optionally, you can load the cache group with the `LOAD CACHE GROUP` statement (optionally, with parallelism) or initiate a dynamic load. See Dynamic Cache Groups.

The following example reloads the `cg` dynamic cache group:

```
ALTER CACHE GROUP cg_dynamic SET AUTOREFRESH STATE PAUSED;
UNLOAD CACHE GROUP cg_dynamic COMMIT EVERY 500 ROWS;
LOAD CACHE GROUP cg_dynamic COMMIT EVERY 500 ROWS PARALLEL 2;
```

Loading and Refreshing a Static Cache Group with Autorefresh

If the autorefresh state of a static cache group is `PAUSED`, the autorefresh state is changed to `ON` after a `LOAD CACHE GROUP` or `REFRESH CACHE GROUP` statement issued on the cache group completes.

The following restrictions apply when manually loading or refreshing a static cache group with autorefresh:

- A `LOAD CACHE GROUP` statement can only be issued if the cache tables are empty.
- The autorefresh state must be `PAUSED` before you can issue a `LOAD CACHE GROUP` statement.
- The autorefresh state must be `PAUSED` before you can issue a `REFRESH CACHE GROUP` statement.
- A `LOAD CACHE GROUP` statement cannot contain a `WHERE` clause.
- A `LOAD CACHE GROUP` or `REFRESH CACHE GROUP` statement cannot contain a `WITH ID` clause.
- A `REFRESH CACHE GROUP` statement cannot contain a `WHERE` clause.
- All tables and columns referenced in a `WHERE` clause when loading the cache group must be fully qualified. For example:

  `owner``.`*`table_name`* and *`owner`*`.`*`table_name`*`.`*`column_name`*

When an autorefresh operation occurs on a static cache group, all committed inserts, updates and deletes on the cached Oracle Database tables since the last autorefresh cycle are refreshed into the cache tables. New cache instances may be loaded into the cache tables. Cache instances that already exist in the cache tables are updated or deleted if the corresponding rows in the cached Oracle Database tables have been updated or deleted.

Loading and Refreshing a Dynamic Cache Group with Autorefresh

If the autorefresh state of a dynamic cache group is `PAUSED`, the autorefresh state is changed to `ON` automatically after specific events occur.

- Its cache tables are initially empty, and then a dynamic load, a `LOAD CACHE GROUP` or an unconditional `REFRESH CACHE GROUP` statement issued on the cache group completes.
- Its cache tables are not empty, and then an unconditional `REFRESH CACHE GROUP` statement issued on the cache group completes.

If the autorefresh state of a dynamic cache group is `PAUSED`, the autorefresh state remains at `PAUSED` after any of the following events occur:

- Its cache tables are initially empty, and then a `REFRESH CACHE GROUP ... WITH ID` statement issued on the cache group completes.
- Its cache tables are not empty, and then a dynamic load, a `REFRESH CACHE GROUP ... WITH ID`, or a `LOAD CACHE GROUP` statement issued on the cache group completes.

For a dynamic cache group, an autorefresh operation is similar to a `REFRESH CACHE GROUP` statement that only refreshes committed updates and deletes on the cached Oracle Database tables since the last autorefresh cycle into the cache tables because only existing cache instances in the cache tables are refreshed. New cache instances are not loaded into the cache tables. To load new cache instances into the cache tables of a dynamic cache group, use a `LOAD CACHE GROUP` statement or perform a dynamic load operation. See Dynamic Cache Groups.

The following restrictions apply when manually loading or refreshing a dynamic cache group with automatic refresh:

- The autorefresh state must be `PAUSED` or `ON` before you can issue a `LOAD CACHE GROUP` statement.
- The autorefresh state must be `PAUSED` before you can issue a `REFRESH CACHE GROUP` statement.
- A `LOAD CACHE GROUP` statement that contains a `WHERE` clause must include a `COMMIT EVERY` *`n`* `ROWS` clause after the `WHERE` clause.
- A `REFRESH CACHE GROUP` statement cannot contain a `WHERE` clause.
- All tables and columns referenced in a `WHERE` clause when loading the cache group must be fully qualified. For example:

  `owner``.`*`table_name`* and *`owner`*`.`*`table_name`*`.`*`column_name`*

Manually or Dynamically Loading Cache Groups

You define whether your cache group is manually or dynamically loaded during cache group definition.

> **Note**
>
> A static cache group is one that is created without the `DYNAMIC` keyword. Static cache groups are supported in TimesTen.
> A dynamic cache group is created with the `DYNAMIC` keyword. Dynamic cache groups are only supported in TimesTen.

- Manually loaded: You can manually load data into either a static cache group or a dynamic cache group. You will always manually load data into a static cache group. Perform the initial load of data into either static or dynamic cache groups from cached Oracle Database tables using a `LOAD CACHE GROUP` statement.
- Dynamically loaded on demand: For dynamic cache groups only, TimesTen can dynamically load data on demand. Data is automatically loaded into the TimesTen cache tables from the cached Oracle Database tables when a qualifying `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statement is issued on one of the cache tables and the data does not exist in the cache table but does exist in the cached Oracle Database table.

With both static and dynamic cache groups, a `LOAD CACHE GROUP` statement manually loads into the designated cache tables qualified data that exists in the cached Oracle Database tables but not in the cache tables in TimesTen. However, if a row exists in a cache table but a newer version exists in the cached Oracle Database table, a `LOAD CACHE GROUP` statement does not load that row into the cache table even if it satisfies the predicate of the statement.

By contrast, a `REFRESH CACHE GROUP` statement manually reloads qualifying rows that exists in the cache tables, effectively refreshing the content of the cache. For a static cache group, the rows that are refreshed are all the rows that satisfy the predicate of the `REFRESH CACHE GROUP` statement. However, for a dynamic cache group, the rows that are refreshed are the ones that satisfy the predicate and already exist in the cache tables. In other words, rows that end up being refreshed are the ones that have been updated or deleted in the cached Oracle Database table, but not the ones that have been inserted. Therefore, a refresh operation processes only the rows that are already in the cache tables. No new rows are loaded into the cache tables of a dynamic cache group as a result of a refresh.

The data in the cache instance of a dynamic read-only cache group is consistent with the data in the corresponding rows of the Oracle Database tables. At any instant in time, the data in a cache instance of a static cache group is consistent with the data in the corresponding rows of the Oracle Database tables, taking into consideration the state and the interval settings for autorefresh.

Dynamic Cache Groups

You define whether your cache group is dynamically loaded by specifying the `DYNAMIC` keyword during cache group definition.

When a qualifying SQL statement queries rows that do not exist in the TimesTen database, then TimesTen automatically loads the relevant cache instances from the Oracle database tables into dynamic cache groups. A dynamic load of a cache instance is similar to a `LOAD CACHE GROUP` statement in that it retrieves and automatically loads a qualified cache instance on demand from the Oracle database to the TimesTen database. A cache instance consists of row from the root table of any cache group (that is uniquely identified by either a primary key or a unique index on the root table) and all related rows in the child tables associated by foreign key relationships. Dynamic load operations do not update or delete cache instances that already exist in the cache tables even if the corresponding rows in the cached Oracle Database tables have been updated or deleted. Dynamic load operations are used to dynamically provide data for the application. Often, dynamic load operations are combined with aging, so that data can be aged out when not needed and dynamically loaded when needed.

> **Note**
>
> The `REFRESH CACHE GROUP` statement and autorefresh are used to update or delete cache instances that already exist in the TimesTen database. You can use autorefresh to automatically populate changes made to cache instances in the Oracle Database.

For example, a call center application may not want to preload all of its customers' information into TimesTen as it may be very large. Instead, you can define the cache group with the `DYNAMIC` keyword. After which, the cache group can use dynamic load on demand so that a specific customer's information is loaded only when needed such as when the customer calls or logs onto the system.

This following example creates a dynamic read-only cache group `online_customers` that caches the `sales.customer` table:

```
CREATE DYNAMIC READONLY CACHE GROUP online_customers
FROM sales.customer
 (cust_num NUMBER(6) NOT NULL,
  region   VARCHAR2(10),
  name     VARCHAR2(50),
  address  VARCHAR2(100),
  PRIMARY KEY(cust_num));
```

Any system managed cache group type (read-only, AWT, SWT or hybrid) can be defined with the `DYNAMIC` keyword. A user managed cache group can be defined with the `DYNAMIC` keyword unless it uses both the `AUTOREFRESH` and the `PROPAGATE` cache table attributes.

> **Note**
>
> If you have a dynamic read-only cache group with incremental autorefresh, you can reduce contention and improve performance with either of the methods described in Options for Reducing Contention Between Autorefresh and Dynamic Load Operations.

When a cache group is enabled for dynamic load, a cache instance is uniquely identified either by a primary key, a unique index on any table, or a foreign key of a child table. If a row in the cached Oracle Database table satisfies the `WHERE` clause and the row is not in the TimesTen database, then the entire associated cache instance is loaded in order to maintain the defined relationships between primary keys and foreign keys of the parent and child tables. When a cache group is enabled for dynamic load, the dynamic load operation typically loads only one cache instance into the root table of any cache group, unless you specifically request to load multiple cache instances (as described in Dynamically Loading Multiple Cache Instances).

The `WHERE` clause must specify one of the following for a dynamic load to occur:

- An equality condition with constants and/or parameters on all columns of a primary key or a foreign key of any table of the cache group. If more than one table of a cache group is referenced, each must be connected by an equality condition on the primary or foreign key relationship.
- A mixture of equality or `IS NULL` conditions on all columns of a unique index, provided that you use at least one equality condition. That is, you can perform a dynamic load where

some columns of the unique index are `NULL`. The unique index must be created on the root table of the cache group.

> **Note**
>
> Dynamic loading based on a primary key search of the root table performs faster than primary key searches on a child table or foreign key searches on a child table.

The dynamic load runs in a different transaction than the user transaction that triggers the dynamic load. The dynamic load transaction is committed before the SQL statement that triggers the dynamic load has finished processing. Thus, if the user transaction is rolled back, the dynamically loaded data remains in the cache group.

> **Note**
>
> If the Oracle database is down, the following error is returned:
>
> ```
> 5219: Temporary Oracle connection failure error in OCISessionBegin():
> ORA-01034: ORACLE not available
> ```

The following sections describes dynamic load for cache groups:

- Enabling or Disabling Dynamic Load
- Guidelines for Dynamic Load
- Examples of Dynamic Load of a Single Cache Instance
- Dynamically Loading Multiple Cache Instances
- Returning Errors for Dynamic Load

Enabling or Disabling Dynamic Load

You can enable or disable dynamic load with the `DynamicLoadEnable` connection attribute.

- 0 - Disables dynamic load of Oracle Database data to a single dynamic cache group for the current connection.
- 1 (default) - Enables dynamic load of Oracle Database data to a single dynamic cache group per statement for the current connection.

You can set the `DynamicLoadEnable` optimizer hint to temporarily enable or disable dynamic loading of a single cache instance for a particular transaction. You can set the `DynamicLoadEnable` optimizer hint with one of the following methods:

- Use the `ttIsql` utility `set dynamicloadenable` command.
- Call the `ttOptSetFlag` built-in procedure with the `DynamicLoadEnable` flag set to the desired value. The following example sets dynamic loading to 1.

  ```
  call ttOptSetFlag('DynamicLoadEnable', 1)
  ```

Note

See DynamicLoadEnable, ttIsql or ttOptSetFlag in the *Oracle TimesTen In-Memory Database Reference*.

You can also set connection attributes with the `SQLSetConnectOption` ODBC function (ODBC 2.5) or the `SQLSetConnectAttr` function (ODBC 3.5). See the Option Support for ODBC 2.5 SQLSetConnectOption and SQLGetConnectOption and Attribute Support for ODBC 3.5 SQLSetConnectAttr and SQLGetConnectAttr sections in the *Oracle TimesTen In-Memory Database C Developer's Guide*.

Guidelines for Dynamic Load

This section details the guidelines for a dynamic load to occur of cache instances for each cache group referenced in the main query.

Note

Examples for these guidelines are provided in Examples of Dynamic Load of a Single Cache Instance.

Dynamic load of a cache instance is available only for the following types of statements issued on a cache table in a dynamic cache group:

- When an `INSERT` statement inserts values into any of the child tables of a cache instance that does not currently exist in the TimesTen tables, the cache instance to which the new row belongs dynamically loads. The insert operation for the new child row is propagated to the cached Oracle Database table.
- `SELECT`, `UPDATE`, or `DELETE` statements require that the `WHERE` clause have the conditions as stated in Dynamic Cache Groups.

The `SELECT`, `UPDATE`, or `DELETE` statements for which dynamic load is available must satisfy the following conditions:

- If the statement contains a subquery, only the cache group with tables referenced in the main query are considered for a dynamic load.
- If the statement references multiple tables of the cache group, the statement must include an equality join condition between the primary keys and foreign keys for all parent and child relationships.
- The statement cannot contain the `UNION`, `INTERSECT`, or `MINUS` set operators.
- The statement can reference non-cache tables.
- The statement can reference cache tables from only one dynamic cache group.

Dynamic load of a cache instance occurs when you set `DynamicLoadEnable=1` and the request passes the following rules:

- Dynamic load of a cache instance does not occur for a cache group if any table of the cache group is specified more than once in any `FROM` clause.
- Only the conditions specified in the query are considered for dynamic load, which excludes any derived conditions.

- If any cache group is referenced only in a subquery, it is not considered for a dynamic load.
- When using an active standby pair replication scheme, dynamic load cannot occur in any subscriber.

The following considerations can affect dynamic load:

- If tables within multiple cache groups or non-cache group tables are specified in the main query, the join order influences if the cache instance is loaded. If during the processing of the query, a dynamic load is possible and necessary to produce the query results, the dynamic load occurs. However, if no rows are returned, then some or all of the cache instances are not dynamically loaded.
- If a statement specifies more than the dynamic load condition on tables of a cache group, the cache instance may be dynamically loaded even though the additional conditions are not qualified for the statement.

You can use aging with a dynamic cache group. TimesTen supports two aging types, least recently used (LRU) aging and time-based aging. By default, the data in a dynamic cache group is subject to LRU aging. Time-based aging on a dynamic cache group overrides LRU aging. If the cache group has a time-based aging policy defined, the timestamp in the root table's row must be within the aging policy's lifetime in order for the cache instance to be loaded.

Rows in a dynamic AWT cache group must be propagated to the Oracle database before they become candidates for aging.

You can use the `ttAgingLRUConfig` built-in procedure to override the default or current LRU aging attribute settings for the aging cycle and TimesTen database space usage thresholds. See Implementing Aging in a Cache Group for TimesTen.

Examples of Dynamic Load of a Single Cache Instance

Provides an example that defines Oracle database tables, which are then cached into a dynamic AWT cache group.

The following is the definition of the Oracle Database tables that are to be cached in a dynamic AWT cache group. The Oracle Database table is owned by the schema user `sales`.

```
CREATE TABLE customer
(cust_num NUMBER(6) NOT NULL PRIMARY KEY,
 region   VARCHAR2(10),
 name     VARCHAR2(50),
 address  VARCHAR2(100));

CREATE TABLE orders
(ord_num      NUMBER(10) NOT NULL PRIMARY KEY,
 cust_num     NUMBER(6) NOT NULL,
 when_placed  DATE NOT NULL,
 when_shipped DATE NOT NULL);

CREATE TABLE orderdetails
 (orderid  NUMBER(10) NOT NULL,
  itemid   NUMBER(8) NOT NULL,
  quantity NUMBER(4) NOT NULL,
  PRIMARY KEY (orderid, itemid));
```

For example, the following data is in the `sales.customer` cached Oracle Database table.

```
CUST_NUM   REGION    NAME              ADDRESS
--------   -------   ---------------   ---------------------------
       1   West      Frank Edwards     100 Pine St., Portland OR
```

```
        2   East      Angela Wilkins    356 Olive St., Boston MA
        3   Midwest   Stephen Johnson   7638 Walker Dr., Chicago IL
```

On the TimesTen database, connect as the TimesTen cache administration user. Then, run the following statement to create a dynamic AWT cache group `new_customers` that caches the `sales.customer`, `sales.orders`, and `sales.orderdetails` tables:

```
CREATE DYNAMIC ASYNCHRONOUS WRITETHROUGH CACHE GROUP new_customers
FROM sales.customer
 (cust_num NUMBER(6) NOT NULL,
  region   VARCHAR2(10),
  name     VARCHAR2(50),
  address  VARCHAR2(100),
  PRIMARY KEY(cust_num)),
sales.orders
 (ord_num      NUMBER(10) NOT NULL,
  cust_num     NUMBER(6) NOT NULL,
  when_placed  DATE NOT NULL,
  when_shipped DATE NOT NULL,
  PRIMARY KEY(ord_num),
  FOREIGN KEY(cust_num) REFERENCES sales.customer(cust_num)),
sales.orderdetails
 (orderid  NUMBER(10) NOT NULL,
  itemid   NUMBER(8) NOT NULL,
  quantity NUMBER(4) NOT NULL,
  PRIMARY KEY(orderid, itemid),
  FOREIGN KEY(orderid) REFERENCES sales.orders(order_num));
```

The following examples show the default behavior as `DynamicLoadEnable` defaults to 1:

The `sales.customer` TimesTen cache table is initially empty:

```
Command> SELECT * FROM sales.customer;
0 rows found.
```

The following `SELECT` statement with an equality condition on the primary key for the `sales.customer` table results in a dynamic load of a single cache instance:

```
Command> SELECT * FROM sales.customer WHERE cust_num = 1;
< 1, West, Frank Edwards, 100 Pine St., Portland OR >
```

If you do not use an equality condition on the primary key and you do not configure for dynamic load of multiple cache instances, then no dynamic load occurs for this example, since it would result in multiple cache instances. See Dynamically Loading Multiple Cache Instances for details on how to configure for this scenario.

```
Command> SELECT * FROM sales.customer WHERE cust_num IN (1,2);
```

The following example contains equality expressions on all of the primary key columns for a primary key composite. The `orderdetails` table has a composite primary key of `orderid` and `itemid`.

```
UPDATE sales.orderdetails SET quantity = 5 WHERE orderid=2280 AND itemid=663;
```

The following example shows an `INSERT` into the `orders` child table, which initiates a dynamic load. However, if you tried to insert into the `customer` table, which is the parent, no dynamic load occurs.

```
INSERT INTO orders VALUES(1,1, DATE '2012-01-25', DATE '2012-01-30');
```

The following UPDATE statement dynamically loads one cache instance from the cached Oracle Database table into the TimesTen cache table, updates the instance in the cache table, and then automatically propagates the update to the cached Oracle Database table:

```
Command> UPDATE sales.customer SET name = 'Angela Peterson' WHERE cust_num = 2;
Command> SELECT * FROM sales.customer;
< 1, West, Frank Edwards, 100 Pine St., Portland OR >
< 2, East, Angela Peterson, 356 Olive St., Boston MA >
```

The following is the updated data in the sales.customer cached Oracle Database table:

```
CUST_NUM   REGION    NAME              ADDRESS
--------   -------   ---------------   --------------------------
       1   West      Frank Edwards     100 Pine St., Portland OR
       2   East      Angela Peterson   356 Olive St., Boston MA
       3   Midwest   Stephen Johnson   7638 Walker Dr., Chicago IL
```

The following DELETE statement dynamically loads one cache instance from the cached Oracle Database table into the TimesTen cache table, deletes the instance from the cache table, and then automatically propagates the delete to the cached Oracle Database table:

```
Command> DELETE FROM sales.customer WHERE cust_num = 3;
Command> SELECT * FROM sales.customer;
< 1, West, Frank Edwards, 100 Pine St., Portland OR >
< 2, East, Angela Peterson, 356 Olive St., Boston MA >
```

The following is the updated data in the sales.customer cached Oracle Database table.

```
CUST_NUM   REGION    NAME              ADDRESS
--------   -------   ---------------   --------------------------
       1   West      Frank Edwards     100 Pine St., Portland OR
       2   East      Angela Peterson   356 Olive St., Boston MA
```

The following is an example of a dynamic load performed using all columns of a unique index on the root table. The departments table is defined in a dynamic AWT cache group. A unique index is created on this cache group consisting of the manager_id and location_id.

The following creates the departments table on the Oracle database.

```
Command> CREATE TABLE departments(
        department_id INT NOT NULL PRIMARY KEY,
        department_name VARCHAR(10) NOT NULL,
        technical_lead INT NOT NULL,
        manager_id INT,
        location_id INT NOT NULL);
```

The following creates the dynamic AWT cache group and a unique index on the dept_cg root table:

```
Command> CREATE DYNAMIC ASYNCHRONOUS WRITETHROUGH CACHE GROUP dept_cg
        FROM departments
        (department_id INT NOT NULL PRIMARY KEY,
         department_name VARCHAR(10) NOT NULL,
         technical_lead INT NOT NULL,
         manager_id INT, location_id INT NOT NULL);

Command> CREATE UNIQUE INDEX dept_idx
         ON departments
         (manager_id,
          location_id);
```

The following inserts three records into the departments table on the Oracle database:

```
Command> INSERT INTO departments
              VALUES (1, 'acct', 1, 1, 100);
1 row inserted.
Command> INSERT INTO departments
          VALUES (2, 'hr', 2, 2, 200);
1 row inserted.
Command> INSERT INTO departments
          VALUES (3, 'owner', 3, NULL, 300);
1 row inserted.
Command> commit;
```

On TimesTen, dynamically load a cache instance based on the unique index:

```
Command> SELECT * FROM departments;
0 rows found.
Command> SELECT * FROM departments
           WHERE manager_id IS NULL AND location_id=300;
< 3, owner, 3, <NULL>, 300 >
1 row found.
Command> SELECT * FROM departments;
< 3, owner, 3, <NULL>, 300 >
1 row found.
Command> SELECT * FROM departments
          WHERE manager_id=2 AND location_id=200;
< 2, legal, 2, 2, 200 >
1 row found.
Command> SELECT * FROM departments;
< 2, legal, 2, 2, 200 >
< 3, owner, 3, <NULL>, 300 >
2 rows found.
```

Dynamically Loading Multiple Cache Instances

If configured, TimesTen can dynamically load multiple cache instances for dynamic cache groups that contain only a single table.

TimesTen Classic dynamically loads cache instances associated with a primary key that do not already exist in the cache group. Any cache instances associated with a primary key that already exist in the cache group are not reloaded. As a result, your query may return partial results. If a cache instance already exists on TimesTen, these cache instances can only be updated with either an autorefresh operation or a `REFRESH CACHE GROUP` statement.

The following sections describe methods for dynamically loading multiple cache instances:

- Dynamically Loading Multiple Cache Instances with Multiple Primary Keys
- Dynamically Loading Multiple Cache Instances Without Multiple Primary Keys

Dynamically Loading Multiple Cache Instances with Multiple Primary Keys

TimesTen Classic can dynamically load multiple cache instances for a `SELECT` statement that includes more than one primary key referenced in the `WHERE` clause on a single table cache group.

You can dynamically load multiple cache instances by specifying multiple primary key values in the `WHERE` clause.

- Only supported with `SELECT` statements.
- Only supported on a single table cache group.

- For a multiple column primary key, all columns of the primary key must be specified in the WHERE clause.
- Each primary key in the WHERE clause must use conditions with either an IN operator and/or a single value from an equality condition.

By default, the DynamicLoadMultiplePKs or TT_DynamicLoadMultiplePKs statement, transaction or connection level hint is set to 1. This must be enabled for dynamic load for multiple cache instances using more than one primary key.

- Statement level hint:

  ```
  /*+TT_DynamicLoadMultiplePKs(1)*/
  ```

- Transaction level hint:

  ```
  Call ttOptSetFlag(DynamicLoadMultiplePKs, 1)
  ```

- Connection level hint:

  ```
  OptimizerHint = TT_DynamicLoadMultiplePKs(1)
  ```

Note

See Use Optimizer Hints to Modify the Execution Plan in the *Oracle TimesTen In-Memory Database Operations Guide*, ttOptSetFlag in the *Oracle TimesTen In-Memory Database Reference* and Optimizer Hints in the *Oracle TimesTen In-Memory Database SQL Reference*.

The following examples use a cache group of products_cg that caches the Oracle database products table.

```
CREATE DYNAMIC READONLY CACHE GROUP products_cg FROM
  products(prod_type INT NOT NULL, prod_id BIGINT NOT NULL, prod_name VARCHAR2(100),
                 prod_weight NUMBER, PRIMARY KEY(prod_type, prod_id));
```

The following examples demonstrate SELECT statements with a WHERE clause with multiple primary keys that use conditions with either an IN operator and/or a single value from an equality condition to return the name and weight of multiple products.

If the primary key of a root table is composed of two columns, x and y, the following SELECT queries do result in a dynamic load:

- Both columns of the primary key use a condition with an IN operator.

  (x,y) IN ((1,2),(3,4))

  ```
  SELECT p.prod_name, p.prod_weight
   FROM products p
    WHERE (((prod_type, p.prod_id) IN ((1,2), (10,20) , (100, 200))));
  ```

- Both columns of the primary key use conditions with an IN operator.

  (x IN (1,3)) AND (y IN (2,4))

  ```
  SELECT p.prod_name, p.prod_weight
   FROM products p WHERE ((p.prod_type IN (1, 10, 100)) AND (p.prod_id IN (2, 20,
  200)));
  ```

- Both columns of the primary key use conditions with an equality condition resulting in a single value.

  (x=1 OR x=3) AND (y=2 OR y=4)

```
SELECT p.prod_name, p.prod_weight
 FROM products p WHERE ((p.prod_type = 10 OR p.prod_type=100) AND
    (p.prod_id = 20 OR p.prod_id = 200));
```

Dynamically Loading Multiple Cache Instances Without Multiple Primary Keys

TimesTen Classic can dynamically load multiple cache instances without using multiple primary keys referenced in a `WHERE` clause on a single table cache group.

If the query tries to load cache instances that both exist and do not exist in the database, then the entire dynamic load operation does not execute. The dynamic load only executes if none of the cache instances requested already exist in the TimesTen database.

By default, TimesTen Classic does not dynamically load multiple cache instances for a single table cache group when the `SELECT` statement has an arbitrary `WHERE` clause, unless you set one of the following statement, transaction or connection level hints to 1.

- Statement level hint:

  ```
  /*+TT_DynamicLoadRootTbl (1)*/
  ```

- Transaction level hint:

  ```
  Call ttOptSetFlag(DynamicLoadRootTbl , 1)
  ```

- Connection level hint:

  ```
  OptimizerHint = TT_DynamicLoadRootTbl (1)
  ```

Note

See Use Optimizer Hints to Modify the Execution Plan in the *Oracle TimesTen In-Memory Database Operations Guide*, ttOptSetFlag in the *Oracle TimesTen In-Memory Database Reference* and Optimizer Hints in the *Oracle TimesTen In-Memory Database SQL Reference*.

Restrictions for dynamic load of multiple cache instances with arbitrary `WHERE` clause

In order for a dynamic load of multiple cache instances for a single table cache group, the `SELECT` statement query must comply with the following:

- The results of the `WHERE` clause do not include any cache instances that currently exist in the TimesTen database.
- The `WHERE` clause must be supported by the Oracle Database SQL syntax.
- Does not qualify for any other dynamic load condition.
- Does not use aggregation.
- No other table is referenced within the query. That is, the `SELECT` statement does not specify any `JOIN` clauses or any subqueries embedded within the `WHERE` clause.
- Does not use the `SELECT`...`FOR UPDATE` clause or the `INSERT` … `FOR SELECT` clause.

Examples

These examples use the following cache group definition on the TimesTen database:

```
CREATE DYNAMIC READONLY CACHE GROUP cust_orders FROM
    customers(cust_id BIGINT NOT NULL PRIMARY KEY, cust_name VARCHAR2(100),
```

```
    cust_street VARCHAR2(200), cust_state VARCHAR2(2), cust_zip VARCHAR2(10))
    WHERE (customers.cust_state = 'CA');
```

Data is inserted into the Oracle database.

```
INSERT INTO customers(cust_id, cust_name, cust_street, cust_state, cust_zip)
  VALUES (100, 'Tom Hanks', '100 Rodeo Dr', 'CA', '90210');
INSERT INTO customers(cust_id, cust_name, cust_street, cust_state, cust_zip)
  VALUES (200, 'Fred Rogers', '1 Make-Believe Ave', 'CA', '90210');
```

None of the requested customers are in the cache group on the TimesTen database; thus, all of the requested customers (and their orders) are dynamically loaded and their names are returned by the query.

```
SELECT c.cust_name
 FROM customers c
  WHERE (c.cust_zip like '90210%');
<'Tom Hanks'>
<'Fred Rogers'>
```

Another customer and full data is inserted into the Oracle database:

```
INSERT INTO customers(cust_id, cust_name, cust_street, cust_state, cust_zip)
  VALUES (300, 'Matthew Rhys', '2 Moscow Cir', 'CA', '90210');
```

On the TimesTen database, the following query is executed. Since the cache group already has at least 1 row that satisfies the query, the dynamic load is not triggered. Thus, only data that currently exists in the cache group are returned for the query.

```
SELECT c.cust_name
 FROM customers c
  WHERE (c.cust_zip like '90210%');
<'Tom Hanks'>
<'Fred Rogers'>
```

Returning Errors for Dynamic Load

You can configure TimesTen to return an error if a `SELECT`, `UPDATE` or `DELETE` statement does not meet the requirements.

See Guidelines for Dynamic Load for requirements of a dynamic load.

The `DynamicLoadErrorMode` connection attribute controls what happens when an application runs a SQL operation against a dynamic cache group and the SQL operation cannot use dynamic load in a particular connection.

- When `DynamicLoadErrorMode` is set to a value of 0, dynamic load happens to any cache group referenced in the query that is qualified for dynamic load. Cache groups that do not qualify are not dynamically loaded and no errors are returned. When `DynamicLoadEnable=1`, no dynamic load occurs if the query references more than one cache group.
- When `DynamicLoadErrorMode` is set to a value of 1, a query fails with an error if any dynamic cache group referenced in the query is not qualified for dynamic load. The error indicates the reason why the dynamic load cannot occur.

To set the connection attribute solely for a particular transaction, use one of the following:

- Use the `ttIsql` utility `set dynamicloaderrormode 1` command.
- Call the `ttOptSetFlag` built-in procedure with the `DynamicLoadErrorMode` flag and the optimizer value set to 1.

```
call ttOptSetFlag('DynamicLoadErrorMode', 1)
```

Call the `ttOptSetFlag` built-in procedure with the `DynamicLoadErrorMode` flag and the optimizer value set to 0 to suppress error reporting when a statement does not comply with dynamic load requirements.

Determining the Number of Cache Instances Affected by an Operation

You can use mechanisms to determine how many cache instances were loaded by a `LOAD CACHE GROUP` statement, refreshed by a `REFRESH CACHE GROUP` statement, flushed by a `FLUSH CACHE GROUP` statement, or unloaded by an `UNLOAD CACHE GROUP` statement.

- Call the `SQLRowCount()` ODBC function.
- Invoke the `Statement.getUpdateCount()` JDBC method.
- Call the `OCIAttrGet()` OCI function with the `OCI_ATTR_ROW_COUNT` option.

Setting a Passthrough Level

When an application issues SQL statements on a TimesTen connection, the SQL statement can be performed in the TimesTen database or passed through to the Oracle database to be performed. Whether the SQL statement is performed in the TimesTen or Oracle database depends on the composition of the statement and the setting of the `PassThrough` connection attribute.

You can set the `PassThrough` connection attribute to define which statements are to be performed locally in TimesTen and which are to be redirected to the Oracle database for processing.

The passthrough level can be set at any time and takes effect immediately. The value can be set to 0 through 3. When appropriate within passthrough levels 1 through 3, TimesTen connects to the Oracle database using the current user's credentials. You can use either an Oracle Wallet set up with the cache administration user credentials pointed to by the `PWDWallet` connection attribute or provide the cache administration user name in the `UID` connection attribute and the `OraclePwd` connection attribute as the Oracle password. See Providing Both Cache Administration Users and Passwords in the *Oracle TimesTen In-Memory Database Security Guide*.

Passing through update operations to the Oracle database for processing is not recommended when issued on cache tables in an AWT or SWT cache group. See Considerations for Using Passthrough.

Note

A transaction that contains operations that are replicated with `RETURN TWOSAFE` cannot have a `PassThrough` setting greater than 0. If `PassThrough` is greater than 0, an error is returned and the transaction must be rolled back.

When `PassThrough` is set to 0, 1, or 2, the following behavior occurs when a dynamic load condition exists:

- A dynamic load can occur for a `SELECT` operation on cache tables in any dynamic cache group type.
- A dynamic load for an `INSERT`, `UPDATE`, or `DELETE` operation can only occur on cached tables with dynamic AWT or SWT cache groups.

See Dynamic Cache Groups.

The following sections describe the different passthrough options:

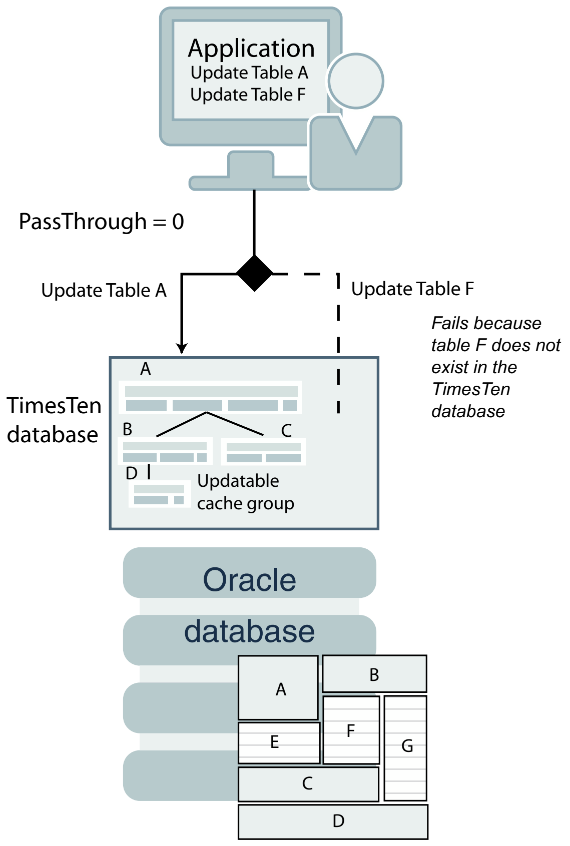
- PassThrough=0
- PassThrough=1
- PassThrough=2
- PassThrough=3
- Considerations for Using Passthrough
- Changing the Passthrough Level for a Connection or Transaction
- Automatic Passthrough of Dynamic Load to the Oracle Database

PassThrough=0

`PassThrough=0` is the default setting and specifies that all SQL statements are to be performed in the TimesTen database.

Figure 5-1 shows that Table A is updated on the TimesTen database. Table F cannot be updated because it does not exist in TimesTen.

Figure 5-1 PassThrough=0



PassThrough=1

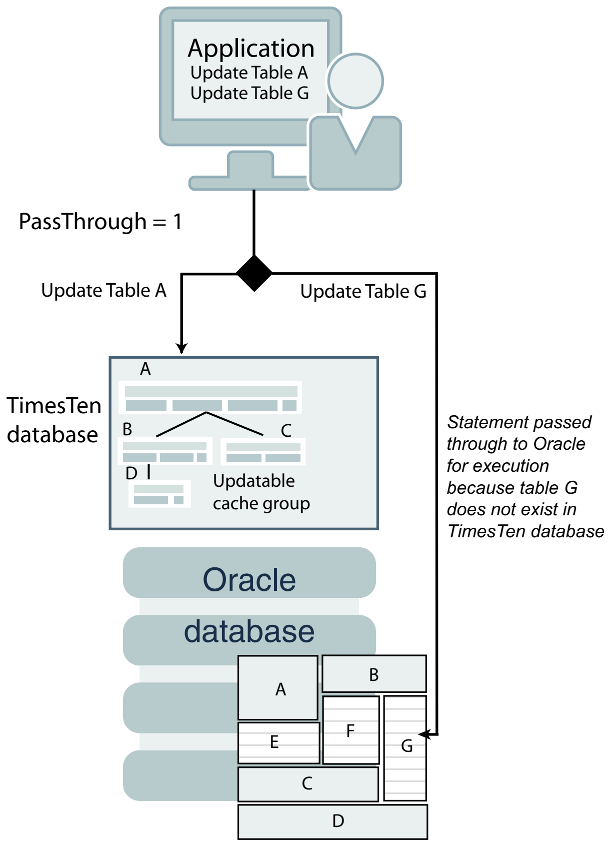
`PassThrough=1` specifies that all DDL are run on TimesTen and most SQL statements are run on TimesTen unless the tables referenced only exist on the Oracle database or the SQL statement can only be parsed or understood on the Oracle database.

Set `PassThrough=1` to specify that:

- DDL statements are always executed on TimesTen.
- `INSERT`, `UPDATE` and `DELETE` statements are run on TimesTen unless they reference one or more tables that do not exist in TimesTen. If they reference one or more tables that do not exist in TimesTen, then these statements are passed through to run on the Oracle database.
- If SQL statements generate a syntax error in TimesTen, include keywords that do not exist in TimesTen SQL, or if one or more tables referenced within the statement do not exist in TimesTen, then these statements are passed through to run on the Oracle database.
- If TimesTen cannot parse `INSERT`, `UPDATE` or `DELETE` statements, TimesTen returns an error and the statement is not passed through to the Oracle database.

Figure 5-2 shows that Table A is updated in the TimesTen database, while Table G is updated in the Oracle database because Table G does not exist in the TimesTen database.

Figure 5-2 PassThrough=1



PassThrough=2

`PassThrough=2` specifies that `INSERT`, `UPDATE` and `DELETE` statements performed on tables in read-only cache groups or user managed cache groups with the `READONLY` cache table attribute are passed through to the Oracle database.

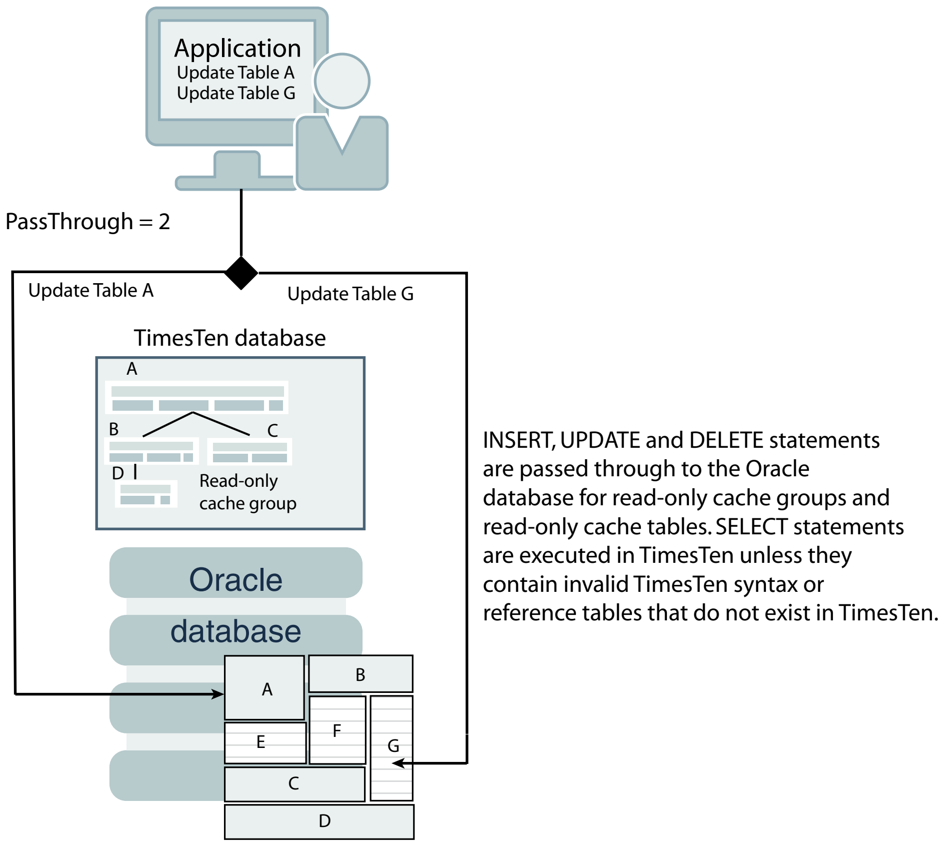
`Passthrough=1` behavior applies for all other operations and cache group types.

Note

You are responsible in preventing conflicts that may occur if you update the same row in a cache table in TimesTen as another user updates the cached Oracle Database table concurrently.

Figure 5-3 shows that updates to Table A and Table G in a read-only cache group are passed through to the Oracle database.

Figure 5-3 PassThrough=2

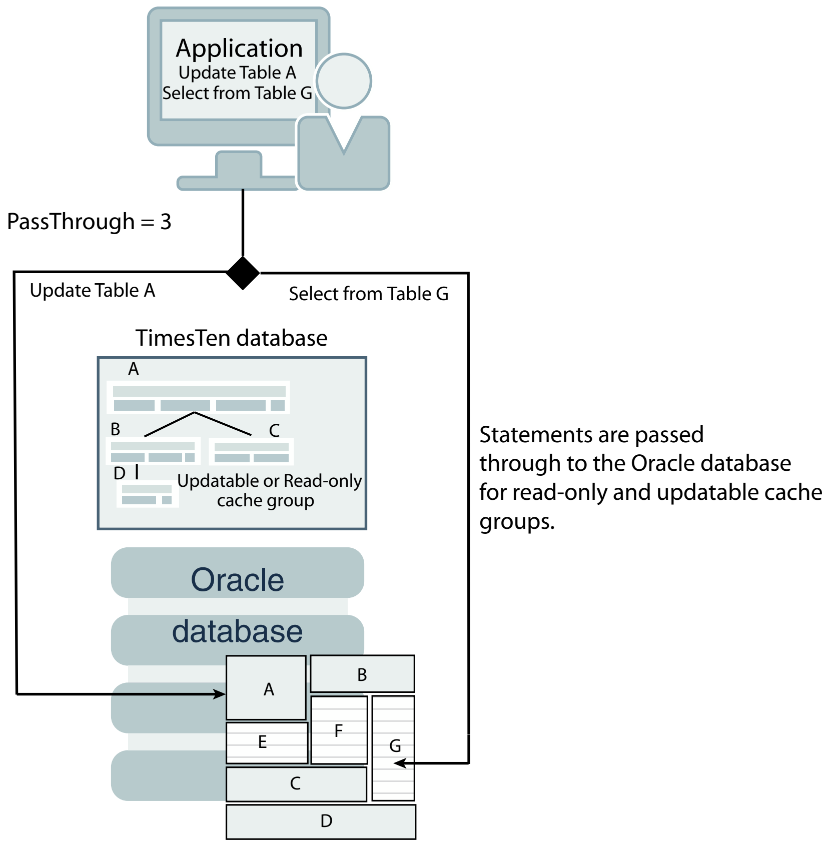

Application
Update Table A
Update Table G
PassThrough = 2
Update Table A
Update Table G
TimesTen database
A
B
C
D
Read-only cache group
Oracle database
A
B
E
F
G
C
D
INSERT, UPDATE and DELETE statements are passed through to the Oracle database for read-only cache groups and read-only cache tables. SELECT statements are executed in TimesTen unless they contain invalid TimesTen syntax or reference tables that do not exist in TimesTen.

PassThrough=3

`PassThrough=3` specifies that all statements are passed through to the Oracle database for processing.

Figure 5-4 shows that Table A is updated on the Oracle database for a read-only or updatable cache group. A `SELECT` statement that references Table G is also passed through to the Oracle database.

Figure 5-4 PassThrough=3



Considerations for Using Passthrough

Passing through update operations to the Oracle database for processing is not recommended when issued on cache tables in an AWT or SWT cache group.

- Committed changes on cache tables in an AWT cache group are automatically propagated to the cached Oracle Database tables in asynchronous fashion. However, passing through an update operation to the Oracle database for processing within the same transaction as the update on the cache table in the AWT cache group renders the propagate of the cache table update synchronous, which may have undesired results.
- Committed changes on cache tables in an SWT cache group can result in self-deadlocks if, within the same transaction, updates on the same tables are passed through to the Oracle database for processing.

A PL/SQL block cannot be passed through to the Oracle database for processing. Also, you cannot pass through to Oracle Database for processing a reference to a stored procedure or function that is defined in the Oracle database but not in the TimesTen database.

For more information about how the `PassThrough` connection attribute setting determines which statements are performed in the TimesTen database and which are passed through to the Oracle database for processing and under what circumstances, see PassThrough in *Oracle TimesTen In-Memory Database Reference*.

Note

The passthrough feature uses OCI to communicate with the Oracle database. The OCI diagnostic framework installs signal handlers that may impact signal handling that you use in your application. You can disable OCI signal handling by setting `DIAG_SIGHANDLER_ENABLED=FALSE` in the `sqlnet.ora` file. Refer to Fault Diagnosability in OCI in *Oracle Call Interface Programmer's Guide*.

Changing the Passthrough Level for a Connection or Transaction

You can override the current passthrough level using the `ttIsql` utility's `set passthrough` command which applies to the current transaction.

You can also override the setting for a specific transaction by calling the `ttOptSetFlag` built-in procedure with the `PassThrough` flag. The following procedure call sets the passthrough level to 3:

```
CALL ttOptSetFlag('PassThrough', 3);
```

The `PassThrough` flag setting takes effect when a statement is prepared and it is the setting that is used when the statement is performed even if the setting has changed from the time the statement was prepared to when the statement is performed. After the transaction has been committed or rolled back, the original connection setting takes effect for all subsequently prepared statements.

Automatic Passthrough of Dynamic Load to the Oracle Database

Set the `TT_DynamicPassthrough` optimizer hint to notify TimesTen Classic to pass through qualified `SELECT` statements to the Oracle database for cache groups created without a `WHERE` clause.

When an application issues statements on a TimesTen connection, the statement can be executed in the TimesTen database or passed through to the Oracle database for resolution. If passed through to the Oracle database, the results are returned but the cache instance is not loaded. Whether the statement is executed on the TimesTen or Oracle databases depends on the composition of the statement and the setting of the `PassThrough` connection attribute.

In TimesTen Classic, for cache groups that are created without a `WHERE` clause, you can limit the number of rows that are dynamically loaded from the Oracle database into the cache instance. You can set the `TT_DynamicPassthrough(`*N*`)` optimizer hint, where *N* is the limit to the number of rows allowed to load into the cache instance. If any `SELECT` statement to the Oracle database would return a result with > *N* number of rows, then the statement is passed through to the Oracle database and the results are not loaded into the cache instance.

By default, the `SELECT` statement for a dynamic load of a cache group that qualifies for dynamic load is executed on the TimesTen Classic database and all rows of the cache instances are loaded. In addition, if you provide the optimizer hint and set *N*=0, then all rows are loaded into the cache instance on the TimesTen Classic database.

This optimizer hint is supported as connection and statement level hints.

Statement level hint:

```
/*+TT_DynamicPassThrough (1)*/
```

Connection level hint:

```
OptimizerHint = TT_DynamicPassThrough (1)
```

The following example is a statement level optimizer hint requesting a dynamic passthrough of a `SELECT` statement to the Oracle database if a dynamic load returns 1000 rows or more for the `SELECT` statement.

```
SELECT /*+ TT_DynamicPassThrough(1000)*/ ...
```

See Setting a Passthrough Level.

See Optimizer Hints in the Oracle TimesTen In-Memory Database SQL Reference and Use Optimizer Hints to Modify the Execution Plan in the Oracle TimesTen In-Memory Database Operations Guide.

6

Managing a Caching Environment

You can manage and monitor various aspects of a caching system such as cache groups and the cache agent process.

Checking the Status of the Cache and Replication Agents in TimesTen

In TimesTen, you can use either the `ttAdmin` or `ttStatus` utility to check whether the cache agent and replication agent processes are running as well as determine each agent's start policy.

You can use a `ttAdmin -query` command to determine the status of the cache and replication agents, as well as the cache and replication agent start policies for a TimesTen database:

```
% ttAdmin -query cache1
RAM Residence Policy            : inUse
Replication Agent Policy        : manual
Replication Manually Started    : True
Cache Agent Policy              : always
Cache Agent Manually Started    : True
```

See ttAdmin in *Oracle TimesTen In-Memory Database Reference*.

Using the `ttStatus` utility without any commands shows all status information for cache and replication for all TimesTen instances:

```
% ttStatus
TimesTen status report as of Thu May  7 13:42:01 2009

Daemon pid 9818 port 4173 instance myinst
TimesTen server pid 9826 started on port 4175
------------------------------------------------------------------------
Data store /disk1/databases/database1
There are 38 connections to the data store
```

```
Shared Memory KEY 0x02011c82 ID 895844354
PL/SQL Memory KEY 0x03011c82 ID 895877123 Address 0x10000000
Type            PID     Context     Connection Name              ConnID
Cache Agent     1019    0x0828f840  Handler                           2
Cache Agent     1019    0x083a3d40  Timer                             3
Cache Agent     1019    0x0842d820  Aging                             4
Cache Agent     1019    0x08664fd8  Garbage Collector(-1580741728)    5
Cache Agent     1019    0x084d6ef8  Marker(-1580213344)               6
Cache Agent     1019    0xa5bb8058  DeadDsMonitor(-1579684960)        7
Replication     18051   0x08c3d900  RECEIVER                          8
Replication     18051   0x08b53298  REPHOLD                           9
Replication     18051   0x08af8138  REPLISTENER                      10
Replication     18051   0x08a82f20  LOGFORCE                         11
Replication     18051   0x08bce660  TRANSMITTER                      12
Subdaemon       9822    0x080a2180  Manager                        2032
Subdaemon       9822    0x080ff260  Rollback                       2033
Subdaemon       9822    0x08548c38  Flusher                        2034
Subdaemon       9822    0x085e3b00  Monitor                        2035
Subdaemon       9822    0x0828fc10  Deadlock Detector              2036
Subdaemon       9822    0x082ead70  Checkpoint                     2037
Subdaemon       9822    0x08345ed0  Aging                          2038
Subdaemon       9822    0x083a1030  Log Marker                     2039
Subdaemon       9822    0x083fc190  AsyncMV                        2040
Subdaemon       9822    0x084572f0  HistGC                         2041
Replication policy  : Manual
Replication agent is running.
Cache Agent policy  : Always
TimesTen's Cache agent is running for this data store
PL/SQL enabled.
------------------------------------------------------------------------
```

The information displayed by the `ttStatus` utility include the following that pertains to TimesTen for each TimesTen instance:

- The names of the cache agent process threads that are connected to the TimesTen database
- The names of the replication agent process threads that are connected to the TimesTen database
- Status on whether the cache agent is running
- Status on whether the replication agent is running
- The cache agent start policy
- The replication agent start policy

See ttStatus in *Oracle TimesTen In-Memory Database Reference*.

Cache Agent and Replication Connection Recovery

When a connection from the cache agent to the Oracle database fails, the cache agent attempts to connect every 10 seconds. If the cache agent cannot connect to the Oracle database, the cache agent restarts after 10 minutes. This behavior repeats forever.

When a connection from the replication agent to the Oracle database fails, the replication agent attempts to reconnect to the Oracle database after 120 seconds. If it cannot reconnect after 120 seconds, the replication agent stops and does not restart.

If Fast Application Notification (FAN) is enabled on the Oracle database, the cache agent and the replication agent receive immediate notification of connection failures. If FAN is not enabled, the agents may wait until a TCP timeout occurs before becoming aware that the connection has failed.

If the Oracle Real Application Clusters (Oracle RAC) is enable on the Oracle database, along with FAN and Transparent Application Failover (TAF), then TAF manages the connection to a new Oracle Database instance. See Using Cache in an Oracle RAC Environment.

Managing a Cache Environment with Oracle Database Objects

For a cache group with trigger-based autorefresh, TimesTen creates a change log table and two triggers in the Oracle database for each cache table in the cache group. One trigger is fired for each `INSERT` statement and another trigger is fired for each `UPDATE` or `DELETE` statement on the cached Oracle Database table.

These triggers record the primary key of the changed rows in the change log table.

The cache agent periodically scans the change log table for modified keys and then joins this table with the cached Oracle Database table to get a snapshot of the latest changes.

Note

If you cache the same Oracle database table in a cache group on two different TimesTen databases, we recommend that you use the same cache administration user name on both TimesTen databases as the owner of the cache table on each TimesTen database. See Caching the Same Oracle Table on Two or More TimesTen Databases.

For each cache administration user, TimesTen creates the following Oracle Database tables, where *`version`* is an internal TimesTen version number and *`object-ID`* is the ID of the cached Oracle Database table:

Table Name	Description
`TT_`*`version`*`_AGENT_STATUS`	Created when the first cache group is created. Stores information about each Oracle Database table cached in a cache group with autorefresh.
`TT_`*`version`*`_AR_PARAMS`	Created when the cache administration user name and password is set. Stores the action to take when the cache administration user's tablespace is full.
`TT_`*`version`*`_ARDL_CG_COUNTER`	Created when you execute either the `grantCacheAdminPrivileges.sql` or the `initCacheAdminSchema.sql` scripts or when the cache administration user and password are set. Contains information used for reducing contention for dynamic read-only cache groups with incremental autorefresh. See Reducing Contention for Dynamic Read-Only Cache Groups with Incremental Autorefresh.
`TT_`*`version`*`_CACHE_STATS`	Created when the cache administration user name and password is set.
`TT_`*`version`*`_CACHED_COLUMNS`	Stores list of columns that are cached. Created when you run the `initCacheAdminSchema.sql` script or when you set the cache administration user and password.

Table Name	Description
TT_*version*_DATABASES	Created when the cache administration user name and password is set. Stores the autorefresh status for all TimesTen databases that cache data from the Oracle database.
TT_*version*_DB_PARAMS	Created when the cache administration user name and password is set. Stores the cache agent timeout, recovery method for dead cache groups, and the cache administration user's tablespace usage threshold.
TT_version_DBSPECIFIC_PARAMS	Internal use.
TT_*version*_DDL_L	Created when the cache administration user name and password is set. Tracks DDL statements issued on cached Oracle Database tables.
TT_*version*_DDL_TRACKING	Created when the cache administration user name and password is set. Stores a flag indicating whether tracking of DDL statements on cached Oracle Database tables is enabled or disabled.
TT_*version*_LOG_SPACE_STATS	Created when the cache administration user and password are set. Contains statistics used to monitor the cache administration user table space. See Managing the Cache Administration User's Tablespace.
TT_*version*_REPACTIVESTANDBY	Created when the first AWT cache group is created. Tracks the state and roles of TimesTen databases containing cache tables in an AWT cache group that are replicated in an active standby pair replication scheme.
TT_*version*_REPPEERS	Created when the first AWT cache group is created. Tracks the time and commit sequence number of the last update on the cache tables that was asynchronously propagated to the cached Oracle Database tables.
TT_*version*_SYNC_OBJS	Created when the first cache group is created.
TT_*version*_USER_COUNT	Created when the first cache group is created. Stores information about each cached Oracle Database table.
TT_*version*_*object-ID*_L	One change log table is created per Oracle Database table cached in a cache group with autorefresh when the cache group is created. Tracks updates on the cached Oracle Database table.

For each cache administration user, TimesTen creates the following Oracle Database triggers, where *version* is an internal TimesTen version number, *object-ID* is the ID of the cached Oracle Database table, and *schema-ID* is the ID of user who owns the cached Oracle Database table:

Trigger Name	Description
TT_*version*_REPACTIVESTANDBY_T	Created when the first AWT cache group is created. When fired, inserts rows into the TT_*version*_REPACTIVESTANDBY table.

Trigger Name	Description
TT_*version*_*object-ID*_T	This trigger is created for each Oracle Database table cached in a cache group with autorefresh when the cache group is created. Fires for each update or delete operation issued on the cached Oracle Database table to track operations in the TT_*version*_*object-ID*_L change log table. The use of triggers for the autorefresh mechanism applies only to trigger-based cache groups and not to log-based cache groups.
TT_*version*_*object-ID*_TI	This trigger is created for all autorefreshed cached tables, except for autorefreshed cached tables that are exclusively cached as root tables in dynamic read-only cache groups. Fires for each insert operation issued on the cached Oracle Database table to track operations in the TT_*version*_*object-ID*_L change log table. The use of triggers for the autorefresh mechanism applies only to trigger-based cache groups and not to log-based cache groups.
TT_*version*_*schema-ID*_DDL_T	One trigger for each user who owns cached Oracle Database tables. Created when a cache group is created after tracking of DDL statements has been enabled. Fired for each DDL statement issued on a cached Oracle Database table to track operations in the TT_*version*_DDL_L table.

Monitoring Cache Groups

You can obtain information on cache groups and monitor the status of cache group operations.

- Using the ttIsql Utility cachegroups Command
- Monitoring Autorefresh Operations on Cache Groups
- Monitoring AWT Cache Groups
- Configuring a Transaction Log File Threshold for AWT Cache Groups
- Tracking DDL Statements Issued on Cached Oracle Database Tables

Using the ttIsql Utility cachegroups Command

You can obtain information about cache groups in a TimesTen database using the `ttIsql` utility `cachegroups` command.

```
% ttIsql "DSN=cache1;UID=cacheadmin;PwdWallet=/wallets/cacheadminwallet"
Command> cachegroups;

Cache Group CACHEADMIN.RECENT_SHIPPED_ORDERS:

  Cache Group Type: Read Only
  Autorefresh: Yes
  Autorefresh Mode: Incremental
  Autorefresh State: On
  Autorefresh Interval: 1440 Minutes
  Autorefresh Status: ok
  Aging: Timestamp based uses column WHEN_SHIPPED lifetime 30 days cycle 24 hours on

  Root Table: SALES.ORDERS
  Table Type: Read Only


Cache Group CACHEADMIN.SUBSCRIBER_ACCOUNTS:

  Cache Group Type: Asynchronous Writethrough (Dynamic)
  Autorefresh: No
  Aging: LRU on

  Root Table: SALES.SUBSCRIBER
  Table Type: Propagate

Cache Group CACHEADMIN.WESTERN_CUSTOMERS:

  Cache Group Type: User Managed
  Autorefresh: No
  Aging: No aging defined

  Root Table: SALES.ACTIVE_CUSTOMER
  Where Clause: (sales.active_customer.region = 'West')
  Table Type: Propagate

  Child Table: SALES.ORDERTAB
  Table Type: Propagate

  Child Table: SALES.ORDERDETAILS
  Where Clause: (sales.orderdetails.quantity >= 5)
  Table Type: Not Propagate

  Child Table: SALES.CUST_INTERESTS
  Table Type: Read Only

3 cache groups found.
```

The information displayed by the `ttIsql` utility's `cachegroups` command include:

- Cache group type, including whether the cache group is dynamic
- Autorefresh attributes (mode, state, interval) and status, if applicable
- Aging policy, if applicable
- Name of root table and, if applicable, name of child tables
- Cache table `WHERE` clause, if applicable
- Cache table attributes (read-only, propagate, not propagate)

See ttIsql in *Oracle TimesTen In-Memory Database Reference*.

Monitoring Autorefresh Operations on Cache Groups

TimesTen offers several mechanisms to obtain information and statistics about autorefresh operations on cache groups.

See Monitoring Cache Groups with Autorefresh in *Oracle TimesTen In-Memory Database Monitoring and Troubleshooting Guide*.

This example explains how to check and interpret the statistics of log-based AUTOREFRESH cache groups in Oracle TimesTen by using the `SYS.V$CACHE_GROUP_LOG_STATS` view and the `ttCacheLogStats()` procedure.

```
Command> describe sys.v$cache_group_log_stats;
> View SYS.V$CACHE_GROUP_LOG_STATS:
> Columns:
> CACHE_GROUP_OWNER VARCHAR2 (30) INLINE
> CACHE_GROUP_NAME VARCHAR2 (30) INLINE
> NUMINSERTS TT_BIGINT
> NUMUPDATES TT_BIGINT
> NUMDELETES TT_BIGINT
> NUMNOOPS TT_BIGINT
> NUMLOADWAITS TT_BIGINT
> ELEMENTID TT_INTEGER NOT NULL
```

The same output is:

```
Command> call ttCacheLogStats();
< CACHEUSER, CG0, 1, 0, 0, 0, 0 >
< CACHEUSER, CG1, 2, 1, 2, 0, 0 >
2 rows found.
```

This output means:

For cache group `CG0` owned by `CACHEUSER`:

- 1 insert operation was processed.
- 0 updates, deletes, no-ops, or load waits occurred.

For cache group `CG1` owned by `CACHEUSER`:

- 2 inserts, 1 update, and 2 deletes were processed.
- No no-op events or load waits occurred.

Query output is:

```
select * from sys.v$cache_group_log_stats order by cache_group_name;
< CACHEUSER, CG0, 1, 0, 0, 0, 0, 1 >
< CACHEUSER, CG1, 2, 1, 2, 0, 0, 1 >
2 rows found.
```

This output shows the same information but includes the `ELEMENTID` (value 1 here for both groups), which uniquely identifies the element internally.

Monitoring AWT Cache Groups

TimesTen offers several mechanisms to obtain information and statistics about operations in AWT cache groups.

See AWT Performance Monitoring in *Oracle TimesTen In-Memory Database Monitoring and Troubleshooting Guide*.

Configuring a Transaction Log File Threshold for AWT Cache Groups

In TimesTen, the replication agent uses the transaction log to determine which updates on cache tables in AWT cache groups have been propagated to the cached Oracle Database tables and which updates have not. If updates are not being automatically propagated to the Oracle database because of a failure, transaction log files accumulate on the file system. Examples of a failure that prevents propagation are that the replication agent is not running or the Oracle database server is unavailable. See Monitoring Accumulation of Transaction Log Files in *Oracle TimesTen In-Memory Database Operations Guide*.

You can call the `ttCacheAWTThresholdSet` built-in procedure as the TimesTen cache administration user to set a threshold for the number of transaction log files that can accumulate before TimesTen stops tracking updates on cache tables in AWT cache groups. The default threshold is 0. This built-in procedure can only be called if the TimesTen database contains AWT cache groups.

After the threshold has been exceeded, you need to manually synchronize the cache tables with the cached Oracle Database tables using an `UNLOAD CACHE GROUP` statement followed by a `LOAD CACHE GROUP` statement. TimesTen may purge transaction log files even if they contain updates that have not been propagated to the cached Oracle Database tables.

The following example sets a transaction log file threshold for AWT cache groups. In this example, if the number of transaction log files that contain updates on cache tables in AWT cache groups exceeds 5, TimesTen stops tracking updates and can then purge transaction log files that may contain unpropagated updates:

```
% ttIsql "DSN=cache1;UID=cacheadmin;PwdWallet=/wallets/cacheadminwallet"
Command> CALL ttCacheAWTThresholdSet(5);
```

You can call the `ttCacheAWTThresholdGet` built-in procedure to determine the current transaction log file threshold setting:

```
Command> CALL ttCacheAWTThresholdGet;
< 5 >
Command> exit
```

Tracking DDL Statements Issued on Cached Oracle Database Tables

When a DDL statement is issued on a cached Oracle Database table, this statement can be tracked in the Oracle Database `TT_`*`version`*`_DDL_L` table when the Oracle Database `TT_`*`version_schema-ID`*`_DDL_T` trigger is fired to insert a row into the table. The *`version`* is an internal TimesTen version number and *`schema-ID`* is the ID of user that owns the cached Oracle Database table.
A trigger is created for each Oracle Database user that owns cached Oracle Database tables. One DDL tracking table is created to store DDL statements issued on any cached Oracle Database table. The Oracle cache administration user owns the `TT_`*`version`*`_DDL_L` table and the `TT_`*`version_schema-ID`*`_DDL_T` trigger.

By default, DDL statements are not tracked. On TimesTen, you can enable tracking of DDL statements issued on cached Oracle Database tables, call the `ttCacheDDLTrackingConfig` built-in procedure as the TimesTen cache administration user. The following example enables tracking of DDL statements issued on cached Oracle Database tables:

```
% ttIsql "DSN=cache1;UID=cacheadmin;PwdWallet=/wallets/cacheadminwallet"
Command> CALL ttCacheDDLTrackingConfig('enable');
```

The `TT_`*`version`*`_DDL_L` table and `TT_`*`version_schema-ID`*`_DDL_T` trigger are automatically created if the Oracle cache administration user has been granted the set of required privileges including `CREATE TRIGGER`, `CREATE SEQUENCE`, `CREATE TYPE`, `CREATE PROCEDURE`, `CREATE TABLE` and `CREATE ANY TRIGGER`. These Oracle Database objects are created when you create a cache group after tracking of DDL statements has been enabled.

On TimesTen, if you manually created the Oracle Database objects used to manage the caching of Oracle Database data, you need to run the `ttIsql` utility `cachesqlget` command with the `ORACLE_DDL_TRACKING` option and the `INSTALL` flag as the TimesTen cache administration user. This command should be run for each Oracle Database user that owns cached Oracle Database tables that you want to track DDL statements on. Running this command generates a SQL*Plus script used to create the `TT_`*`version`*`_DDL_L` table and `TT_`*`version_schema-ID`*`_DDL_T` trigger in the Oracle database.

After generating the script, use SQL*Plus to run the script as the `sys` user.

The following example creates DDL tracking table and trigger when Oracle Database objects are manually created. In this example, the SQL*Plus script generated by the `ttIsql` utility `cachesqlget` command is saved to the `/tmp/trackddl.sql` file. The owner of the cached Oracle Database table `sales` is passed as an argument to the command.

```
% ttIsql "DSN=cache1;UID=cacheadmin;PwdWallet=/wallets/cacheadminwallet"
Command> cachesqlget ORACLE_DDL_TRACKING sales INSTALL /tmp/trackddl.sql;
Command> exit

% sqlplus sys as sysdba
Enter password: password
SQL> @/tmp/trackddl
SQL> exit
```

You can run the `ttCacheInfo` utility or the *`timesten_home`*`/install/oraclescripts/cacheInfo.sql` SQL*Plus script as the Oracle cache administration user to display information about the Oracle Database objects used to track DDL statements issued on cached Oracle Database tables. The following example runs the `cacheInfo.sql` SQL*Plus script.

```
% cd timesten_home/install/oraclescripts
% sqlplus cacheadmin/orapwd
SQL> @cacheInfo.sql
***************** Database Information    *********************
Database name: DATABASE1
Unique database name: database1
Primary database name:
Database Role: PRIMARY
Database Open Mode: READ WRITE
Database Protection Mode: MAXIMUM PERFORMANCE
Database Protection Level: UNPROTECTED
Database Flashback On: NO
Database Current SCN: 21512609
*************************************************************
*************Autorefresh Objects Information  ***************
Grid name: grid1 (7D03C680-BD93-4233-A4CF-B0EDB0064F3F)
Timesten database name: database1
Cache table name: SALES.CUSTOMERS
Change log table name: tt_07_96977_L
Number of rows in change log table: 4
Maximum logseq on the change log table: 1
Timesten has autorefreshed updates upto logseq: 1
Number of updates waiting to be autorefreshed: 0
Number of updates that has not been marked with a valid logseq: 0
*************DDL Tracking Object Information  ***************
Common DDL Log Table Name: TT_07_DDL_L
```

```
DDL Trigger Name: TT_07_315_DDL_T
Schema for which DDL Trigger is tracking: SALES
Number of cache groups using the DDL Trigger: 10
****************************

PL/SQL procedure successfully completed.
```

The information returned for each Oracle Database user that owns cached Oracle Database tables includes the name of the DDL tracking table, the name of its corresponding DDL trigger, the name of the user that the DDL trigger is associated with, and the number of cache groups that cache a table owned by the user associated with the DDL trigger.

If a cache group contains more than one cache table, each cache table owned by the user associated with the DDL trigger contributes to the cache group count.

See SQL*Plus Scripts for Cache in this book and ttCacheDDLTrackingConfig and ttCacheInfo in *Oracle TimesTen In-Memory Database Reference*.

Changing Cache User Names and Passwords

You can change any of the user names or passwords for the TimesTen cache administration user or its companion Oracle cache administration user.

1. If you want to modify the TimesTen cache administration user or password, perform the following:

   Note

   Passwords for both the TimesTen cache administration user and its companion Oracle cache administration user can be changed at any time.

   The name for the TimesTen cache administration user must be the same as its companion Oracle cache administration user; however, the passwords may be different. See Create the TimesTen Users.

   a. On the TimesTen database, if you want to modify the password of the TimesTen cache administration user, then use the `ALTER USER` statement on the active master.

   ```
   Command> ALTER USER cacheadmin IDENTIFIED BY newpwd;
   ```

   b. On the back-end Oracle database, you can modify the companion Oracle cache administration user password with the `ALTER USER` statement. If you are working on TimesTen, you can use `Passthrough 3` to run this directly on the Oracle database.

   ```
   Command> passthrough 3;
   Command> ALTER USER cacheadmin IDENTIFIED BY newpwd;
   ```

   Note

   If you have modified the password for the companion Oracle cache administration user, reconnect to the TimesTen database as the TimesTen cache administration user providing passwords for the TimesTen cache administration user and its companion Oracle cache administration user.

c. If you want to change the TimesTen cache administration user, you must first drop all cache groups that the TimesTen cache administration user owns before dropping the existing user and creating a new user. The Oracle cache administration user name can only be changed when there are no cache groups on the TimesTen database.

> **Note**
>
> Alternatively, if you want to use a different user as the TimesTen cache administration user, ensure that it has the correct privileges and a companion Oracle cache administration user with the correct privileges.

In addition, since the TimesTen cache administration user must have a companion Oracle cache administration user with the same name, you must either:

- Drop all tables owned by the current companion Oracle cache administration user, drop the user, and then re-create it with the same name as the new TimesTen cache administration user.
- Choose another Oracle user that has the same name as the TimesTen cache administration user and provides the same functionality.

See Create the TimesTen Users.

d. On TimesTen, if the TimesTen cache administration user name or password are defined in the `sys.odbc.ini` (or `odbc.ini`) file, update the new TimesTen cache administration user name or password in the `sys.odbc.ini` (or `odbc.ini`) file on both the active and standby masters.

2. If you want to modify the Oracle cache administration user or its password, perform the following:

   a. On the back-end Oracle database, you can modify the Oracle cache administration password with the `ALTER USER` statement. The password of the Oracle cache administration user can be changed at any time.

   If you are working on TimesTen, you can use `Passthrough 3` to run this directly on the Oracle database.

```
Command> passthrough 3;
Command> ALTER USER cacheadmin IDENTIFIED BY newpwd;
```

   b. If you want to change the Oracle cache administration user, you must first drop all cache groups on the TimesTen database that the Oracle cache administration user manages before you can drop the Oracle cache administration user on the Oracle database and create a new user. Dropping the cache groups on TimesTen removes all metadata associated with those cache groups.

   When you create a new Oracle cache administration user on the Oracle database, you must follow the same instructions for creating a Oracle cache administration user that are provided in the Create the Oracle Database Users and Default Tablespace.

   c. Set the new user name or password for the Oracle cache administration user.

   On TimesTen, run the `ttCacheUidPwdSet` built-in procedure on the active master database.

Note

See Registering the Cache Administration User Name and Password.

```
Command> call ttCacheUidPwdSet('cacheadmin','newpwd');
```

Dropping Oracle Database Objects Used by Cache Groups with Autorefresh

A TimesTen database is unavailable, for example, when the TimesTen system is taken offline or the database has been destroyed without dropping its cache groups with autorefresh.

Oracle database objects used to implement autorefresh operations also continue to exist in the Oracle database when a TimesTen database is no longer being used but still contains cache groups with autorefresh. Rows continue to accumulate in the change log tables. This impacts autorefresh performance on other TimesTen databases. Therefore, it is desirable to clean up these Oracle database objects associated with the unavailable or abandoned TimesTen database.

When using TimesTen, run the `*timesten_home*/install/oraclescripts/cacheCleanUp.sql` SQL*Plus script as the Oracle cache administration user to clean up the Oracle database objects used to implement autorefresh operations. The host name of the TimesTen system and the TimesTen database (including its path) are passed as arguments to the `cacheCleanUp.sql` script.

You can run the `ttCacheInfo` utility or the `cacheInfo.sql` script as the Oracle cache administration user to determine the host and database names.

The `cacheInfo.sql` script or the `ttCacheInfo` utility can be used to determine whether any objects used to implement autorefresh operations exist in the Oracle database.

The following example demonstrates how to drop Oracle database objects for cache groups with autorefresh. This example uses the `cacheCleanUp.sql` script for a TimesTen system. It drops the change log tables and triggers associated with the `customers` and `orders` cache tables.

```
% cd timesten_home/install/oraclescripts
% sqlplus cacheadmin/orapwd
SQL> @cacheCleanUp "sys1" "/disk1/databases/database1"

*****************************OUTPUT**************************************
Performing cleanup for object_id: 69959 which belongs to table : CUSTOMERS
Executing: delete from tt_07_agent_status where host = sys1 and datastore =
/disk1/databases/database1 and object_id = 69959
Executing: drop table tt_07_69959_L
Executing: drop trigger tt_07_69959_T
Executing: delete from tt_07_user_count where object_id = object_id1
Performing cleanup for object_id: 69966 which belongs to table : ORDERS
Executing: delete from tt_07_agent_status where host = sys1 and datastore =
/disk1/databases/database1 and object_id = 69966
Executing: drop table tt_07_69966_L
Executing: drop trigger tt_07_69966_T
Executing: delete from tt_07_user_count where object_id = object_id1
**************************************************************************
```

See SQL*Plus Scripts for Cache in this guide and ttCacheInfo in *Oracle TimesTen In-Memory Database Reference*.

Impact on Cache Groups When Modifying the Oracle Database Schema

When you need to issue DDL statements such as `CREATE`, `DROP` or `ALTER` on cached Oracle Database tables in order to make changes to the Oracle Database schema, drop the affected cache groups before you modify the Oracle Database schema. Otherwise operations such as autorefresh may fail.

You do *not* need to drop cache groups if you are altering the Oracle Database table to add a column.

To issue other DDL statements for Oracle Database tables, first perform the following tasks:

1. Use `DROP CACHE GROUP` statements to drop all cache groups that cache the affected Oracle Database tables. If you are dropping an AWT cache group, use the `ttRepSubscriberWait` built-in procedure to make sure that all committed changes on the cache tables have been propagated to the cached Oracle Database tables before the cache group is dropped.

```
% ttIsql "DSN=cache1;UID=cacheadmin;PwdWallet=/wallets/cacheadminwallet"
Command> CALL ttRepSubscriberWait('_AWTREPSCHEME','TTREP','_ORACLE','sys1',-1);
```

2. Stop the cache agent.
3. Make the desired changes to the Oracle Database schema.
4. Use `CREATE CACHE GROUP` statements to re-create the cache groups, if feasible.

If you want to truncate an Oracle Database table that is cached in a cache group with autorefresh, perform the following tasks:

1. Use an `ALTER CACHE GROUP` statement to set the cache group's autorefresh state to `PAUSED`.
2. Truncate the Oracle Database table.
3. Manually refresh the cache group using a `REFRESH CACHE GROUP` statement without a `WHERE` or `WITH ID` clause.

Autorefresh operations resume after you refresh the cache group.

Impact of Failed Autorefresh Operations on TimesTen Databases

TimesTen does not delete rows in the change log tables when the cache agent is not running on a TimesTen database. In this case, you can set a cache agent timeout to prevent rows from accumulating in the change log tables.

A change log table is created in the Oracle cache administration user's tablespace for each Oracle Database table that is cached in a cache group with autorefresh. For each update operation issued on these cached Oracle Database tables, a row is inserted into their change log table to keep track of updates that need to be applied to the TimesTen cache tables upon the next incremental autorefresh cycle. TimesTen periodically deletes rows in the change log tables that have been applied to the cache tables.

An Oracle Database table cannot be cached in more than one cache group within a TimesTen database. However, an Oracle Database table can be cached in more than one TimesTen database. This results in an Oracle Database table corresponding to multiple TimesTen cache tables. If updates on cached Oracle Database tables are not being automatically refreshed into all of their corresponding cache tables because the cache agent is not running on one or more of the TimesTen databases that the Oracle Database tables are cached in, rows in their change log tables are not deleted by default. The cache agent may not be running on a particular TimesTen database because the agent was either stopped or never started, the database was destroyed, or the TimesTen instance is down. As a result, rows accumulate in the change log tables and degrade the performance of autorefresh operations on cache tables in TimesTen databases where the cache agent is running. This can also cause the Oracle cache administration user's tablespace to fill up.

For example, if a single Oracle Database table is cached by two or more TimesTen databases where one of the TimesTen databases is unable to connect to the Oracle database, then autorefresh for the disconnected TimesTen database is not performed. Instead, the records in the change log table accumulate (so that the disconnected TimesTen database can catch up once a connection to the Oracle database is established). If the `AgentTimeout` parameter is set to 0 (the default), then all change log records are kept indefinitely until they have been applied to all its cache tables. The change log records of the other TimesTen databases are not purged even though the transaction logs are already applied to the local TimesTen database. Alternatively, you can set the `AgentTimeout` parameter to define a specific timeout to wait before purging the saved change log records and stop the accumulation of these change log records.

The following criteria must be met in order for TimesTen to delete rows in the change log tables when the cache agent is not running on a TimesTen database and a cache agent timeout is set:

- Oracle Database tables are cached in cache groups with autorefresh enabled within more than one TimesTen database.
- The cache agent is running on at least one of the TimesTen databases but is not running on at least another database.
- Rows in the change log tables have been applied to the cache tables on all TimesTen databases where the cache agent is running.
- For those databases where the cache agent is not running, the agent process has been down for a period of time that exceeds the cache agent timeout.

To set the cache agent timeout and prevent rows from accumulating in the change log tables, set the `AgentTimeout` parameter with the `ttCacheConfig` built-in procedure as the TimesTen cache administration user from any of the TimesTen databases that cache data from the Oracle database. Pass the `AgentTimeout` string to the *`Param`* parameter and the timeout setting as a numeric string to the *`Value`* parameter. Do not pass in any values to the *`tblOwner`* and *`tblName`* parameters as they are not applicable to setting a cache agent timeout.

In the following example, the cache agent timeout is set to 900 seconds (15 minutes):

```
% ttIsql "DSN=cache1;UID=cacheadmin;PwdWallet=/wallets/cacheadminwallet"
Command> CALL ttCacheConfig('AgentTimeout',,,'900');
```

To determine the current cache agent timeout setting, call `ttCacheConfig` passing only the `AgentTimeout` string to the *`Param`* parameter:

```
Command> CALL ttCacheConfig('AgentTimeout');
< AgentTimeout, <NULL>, <NULL>, 900 >
```

The default cache agent timeout setting is 0, which means that all change log records are kept indefinitely until they have been applied to all its cache tables. If you set the cache agent timeout to a value between 1 and 600 seconds, the timeout is set to 600 seconds. The cache agent timeout applies to all TimesTen databases that cache data from the same Oracle database and have the same Oracle cache administration user name setting.

When determining a proper cache agent timeout setting, consider the time it takes to load the TimesTen database into memory, the time to start the cache agent process, potential duration of network outages, and anticipated duration of planned maintenance activities.

Each TimesTen database, and all of its cache groups have an autorefresh status to determine whether any deleted rows from the change log tables were not applied to the cache tables in the cache groups. If rows were deleted from the change log tables and not applied to some cache tables because the cache agent on the database was down for a period of time that exceeded the cache agent timeout, those cache tables are no longer synchronized with the cached Oracle Database tables. Subsequent updates on the cached Oracle Database tables are not automatically refreshed into the cache tables until the accompanying cache group is recovered.

The following are the possible statuses for a cache group with autorefresh:

- `ok`: All of the deleted rows from the change log tables were applied to its cache tables. Incremental autorefresh operations continue to occur on the cache group.
- `disabled` or `dead`: Some of the deleted rows from the change log tables were not applied to its cache tables so the cache tables are not synchronized with the cached Oracle Database tables. Autorefresh operations have ceased on the cache group and do not resume until the cache group has been recovered.
- `recovering`: The cache group is being recovered. Once recovery completes, the cache tables are synchronized with the cached Oracle Database tables, the cache group's autorefresh status is set to `ok`, and incremental autorefresh operations resume on the cache group.

The following are the possible autorefresh statuses for a TimesTen database:

- `alive`: All of its cache groups with autorefresh have an autorefresh status of OK.
- `dead`: All of its cache groups with autorefresh have an autorefresh status of dead.
- `recovering`: At least one of its cache groups with autorefresh have an autorefresh status of recovering.

If the cache agent on a TimesTen database is down for a period of time that exceeds the cache agent timeout, the autorefresh status of the database is set to `dead`. Also, the autorefresh status of all cache groups with autorefresh within that database are set to `dead`.

Call the `ttCacheDbCgStatus` built-in procedure as the TimesTen cache administration user to determine the autorefresh status of a cache group and its accompanying TimesTen database. Pass the owner of the cache group to the *`cgOwner`* parameter and the name of the cache group to the *`cgName`* parameter.

In the following example, the autorefresh status of the database is `alive` and the autorefresh status of the `cacheadmin.customer_orders` read-only cache group is `ok`:

```
% ttIsql "DSN=cache1;UID=cacheadmin;PwdWallet=/wallets/cacheadminwallet"
Command> CALL ttCacheDbCgStatus('cacheadmin','customer_orders');
< alive, ok >
```

To view only the autorefresh status of the database and not of a particular cache group, call `ttCacheDbCgStatus` without any parameters:

```
Command> CALL ttCacheDbCgStatus;
< dead, <NULL> >
```

If the autorefresh status of a cache group is `ok`, its cache tables are being automatically refreshed based on its autorefresh interval. If the autorefresh status of a database is `alive`, the autorefresh status of all its cache groups with autorefresh are `ok`.

If the autorefresh status of a cache group is `disabled` or `dead`, its cache tables are no longer being automatically refreshed when updates are committed on the cached Oracle Database tables. The cache group must be recovered in order to resynchronize the cache tables with the cached Oracle Database tables. See Disabling Full Autorefresh for Cache Groups.

You can configure a recovery method for cache groups whose autorefresh status is `dead`.

Call the `ttCacheConfig` built-in procedure as the TimesTen cache administration user from any of the TimesTen databases that cache data from the Oracle database. Pass the `DeadDbRecovery` string to the *`Param`* parameter and the recovery method as a string to the *`Value`* parameter. Do not pass in any values to the *`tblOwner`* and *`tblName`* parameters as they are not applicable to setting a recovery method for dead cache groups.

The following are the valid recovery methods:

- `Normal`: When the cache agent starts, a full autorefresh operation is performed on cache groups whose autorefresh status is `dead` in order to recover those cache groups. This is the default recovery method. However, if you set the `DisableFullAutorefresh` cache configuration parameter to 1, then the `DeadDbRecovery` cache configuration parameter automatically changes to `Manual`.
- `Manual`: For each static cache group whose autorefresh status is `dead`, a `REFRESH CACHE GROUP` statement must be issued in order to recover these cache groups after the cache agent starts.

  For each dynamic cache group whose autorefresh status is `dead`, a `REFRESH CACHE GROUP` or `UNLOAD CACHE GROUP` statement must be issued in order to recover these cache groups after the cache agent starts.
- `None`: Cache groups whose autorefresh status is `dead` must be dropped and then re-created after the cache agent starts in order to recover them.

In the following example, the recovery method is set to `Manual` for cache groups whose autorefresh status is `dead`:

```
% ttIsql "DSN=cache1;UID=cacheadmin;PwdWallet=/wallets/cacheadminwallet"
Command> CALL ttCacheConfig('DeadDbRecovery',,,'Manual');
```

To determine the current recovery method for dead cache groups, call `ttCacheConfig` passing only the `DeadDbRecovery` string to the *`Param`* parameter:

```
Command> CALL ttCacheConfig('DeadDbRecovery');
< DeadDbRecovery, <NULL>, <NULL>, manual >
```

The recovery method applies to all cache groups with autorefresh in all TimesTen databases that cache data from the same Oracle database and have the same Oracle cache administration user name setting.

When a cache group begins the recovery process, its autorefresh status is changed from `dead` to `recovering`, and the status of the accompanying TimesTen database is changed to `recovering`, if it is currently `dead`.

After the cache group has been recovered, its autorefresh status is changed from `recovering` to `ok`. Once all cache groups have been recovered and their autorefresh statuses are `ok`, the status of the accompanying TimesTen database is changed from `recovering` to `alive`.

A full autorefresh operation requires more system resources to process than an incremental autorefresh operation when there is a small volume of updates to refresh and a large number of rows in the cache tables. If you need to bring a TimesTen database down for maintenance activities and the volume of updates anticipated during the downtime on the Oracle Database tables that are cached in cache groups with autorefresh is small, you can consider temporarily setting the cache agent timeout to 0. When the database is brought back up and the cache agent restarted, incremental autorefresh operations resumes on cache tables in cache groups with autorefresh. Full autorefresh operations are avoided because the autorefresh status on the accompanying cache groups were not changed from `ok` to `dead` so those cache groups do not need to go through the recovery process. Make sure to set the cache agent timeout back to its original value once the database is back up and the cache agent has been started.

See ttCacheConfig in the *Oracle TimesTen In-Memory Database Reference*.

Managing the Cache Administration User's Tablespace

You can manage the cache administration user's tablespace.

- Defragmenting Change Log Tables in the Tablespace
- Receiving Notification on Tablespace Usage
- Recovering from a Full Tablespace

Defragmenting Change Log Tables in the Tablespace

Prolonged use or a heavy workload of the change log tables for cache groups with autorefresh can result in fragmentation of the tablespace. This issue applies only to trigger-based cache groups, as they rely on change log tables for tracking changes.

In order to prevent degradation of the tablespace from fragmentation of the change log tables, TimesTen calculates the percentage of fragmentation for the change log tables as a ratio of used space to the total size of the space. If this ratio falls below a defined threshold, TimesTen alerts you of the necessity for defragmentation of the change log tables by logging a message. By default, this threshold is set to 40%.

Note

Messages are logged to the user and support error logs. For details, see Error, Warning, and Informational Messages in the *Oracle TimesTen In-Memory Database Operations Guide*.

To set the fragmentation threshold, call the `ttCacheConfig` built-in procedure as the TimesTen cache administration user from any of the TimesTen databases that cache data from the Oracle database. Pass the `AutoRefreshLogFragmentationWarningPCT` string to the *`Param`* parameter and the threshold setting as a numeric string to the *`Value`* parameter.

To set the time interval for how often to calculate the fragmentation percentage, call the `ttCacheConfig` built-in procedure as the TimesTen cache administration user from any of the TimesTen databases that cache data from the Oracle database. Pass the `AutorefreshLogMonitorInterval` string to the *`Param`* parameter and the time interval (in seconds) as a numeric string to the *`Value`* parameter.

> **Note**
>
> Do not pass in any values to the *`tblOwner`* and *`tblName`* parameters as they are not applicable to setting the fragmentation threshold or the time interval for the threshold calculation.

In the following example, the fragmentation threshold is set to 50% and the time interval for calculating the fragmentation threshold is set to 3600 seconds:

```
% ttIsql "DSN=cache1;UID=cacheadmin;PwdWallet=/wallets/cacheadminwallet"
Command> CALL ttCacheConfig('AutoRefreshLogFragmentationWarningPCT',,,'50');
< AutoRefreshLogFragmentationWarningPCT, <NULL>, <NULL>, 50 >
1 row found.
Command> CALL ttCacheConfig('AutorefreshLogMonitorInterval',,,'3600');
< AutorefreshLogMonitorInterval, <NULL>, <NULL>, 3600 >
1 row found.
```

To determine the current fragmentation threshold setting, call `ttCacheConfig` passing the `AutoRefreshLogFragmentationWarningPCT` string to the *`Param`* parameter:

```
Command> CALL ttCacheConfig('AutoRefreshLogFragmentationWarningPCT');
< AutoRefreshLogFragmentationWarningPCT, <NULL>, <NULL>, 50 >
```

You can either manually initiate defragmentation or configure TimesTen to automatically defragment. To configure what action is taken when the ratio falls below the fragmentation threshold, call the `ttCacheConfig` built-in procedure with the `AutoRefreshLogDeFragmentAction` string to the *`Param`* parameter and the desired action as the *`Value`* parameter as follows:

> **Note**
>
> Do not pass in any values to the *`tblOwner`* and *`tblName`* parameters as they are not applicable to setting the defragmentation action.

- `Manual`. This is the default. No action is taken to defragment the change log tables. Any defragmentation must be performed manually by running the `ttCacheAutoRefreshLogDeFrag` built-in procedure. See Manually Defragmenting the Change Log Tables for Cache Groups with Autorefresh.
- `Compact`: TimesTen defragments the change log tables.
- `CompactAndReclaim`: TimesTen defragments the change log tables and reclaims the space.

> **Note**
>
> When reclaiming space, the change log table is briefly locked, which temporarily suspends writing into the base table.

In the following example, the action is set to `CompactAndReclaim` so that when the fragmentation ratio falls below the threshold, TimesTen defragments the change log tables and reclaims the space:

```
% ttIsql "DSN=cache1;UID=cacheadmin;PwdWallet=/wallets/cacheadminwallet"
Command> CALL ttCacheConfig('AutoRefreshLogDeFragmentAction',,,'CompactAndReclaim');
< AutoRefreshLogDeFragmentAction, <NULL>, <NULL>, compactandreclaim >
1 row found.
```

To determine the current fragmentation threshold setting, call `ttCacheConfig` passing the `AutoRefreshLogDeFragmentAction` string to the *`Param`* parameter:

```
Command> CALL ttCacheConfig('AutoRefreshLogDeFragmentAction');
< AutoRefreshLogDeFragmentAction , <NULL>, <NULL>, compactandreclaim >
```

You can discover the fragmentation percentage of the tablespace and when the last defragmentation operation was performed with the following returned columns from the `ttCacheAutorefreshStatsGet` built-in procedure:

- `AutoRefreshLogFragmentationPCT`: The current fragmentation percentage for the tablespace.
- `AutoRefreshLogFragmentationTS`: The timestamp of when the last fragmentation percentage was calculated.
- `autorefLogDeFragCnt`: The count for how many times the tables in this particular cache group have been defragmented.

See ttCacheConfig and ttCacheAutorefreshStatsGet in the *Oracle TimesTen In-Memory Database Reference*.

Manually Defragmenting the Change Log Tables for Cache Groups with Autorefresh

To manually initiate a defragmentation of the change log tables, call the `ttCacheAutorefreshLogDeFrag` built-in procedure as the TimesTen cache administration user from any of the TimesTen databases that cache data from the Oracle database.

Pass in one of the following strings as the parameter:

- `Compact`: Defragment the change log tables.
- `CompactAndReclaim`: Defragment the change log tables and reclaim the space.

Note

When reclaiming space, the change log table is briefly locked, which temporarily suspends writing into the base table.

The following example manually defragments the change log tables with the `ttCacheAutoRefreshLogDeFrag` built-in procedure providing the `CompactAndReclaim` option:

```
% ttIsql "DSN=cache1;UID=cacheadmin;PwdWallet=/wallets/cacheadminwallet"
Command> CALL ttCacheAutoRefreshLogDeFrag('CompactAndReclaim');
```

See ttCacheAutorefreshLogDefrag in the *Oracle TimesTen In-Memory Database Reference*.

Receiving Notification on Tablespace Usage

In order to avoid the tablespace becoming full, you can configure TimesTen to return a warning to the application when an update operation (such as an `UPDATE`, `INSERT` or `DELETE` statement) is issued on trigger-based cached Oracle database tables and causes the usage of the Oracle cache administration user's tablespace to exceed a specified threshold.

Call the `ttCacheConfig` built-in procedure as the TimesTen cache administration user from any of the TimesTen databases that cache tables from the Oracle database. Pass the `AutoRefreshLogTblSpaceUsagePCT` string to the *`Param`* parameter and the threshold as a numeric string to the *`Value`* parameter. The threshold value represents the percentage of space used in the Oracle cache administration user's tablespace upon which a warning is returned to the application when an update operation is issued on a cached Oracle Database table. Do not pass in any values to the *`tblOwner`* and *`tblName`* parameters as they are not applicable to setting a warning threshold for the usage of the Oracle cache administration user's tablespace.

The Oracle cache administration user must be granted the `SELECT` privilege on the Oracle Database `SYS.DBA_DATA_FILES` table in order for the TimesTen cache administration user to set a warning threshold on the Oracle cache administration user's tablespace usage, and for the Oracle cache administration user to monitor its tablespace to determine if the configured threshold has been exceeded.

The following example configures a warning to be returned to the application that issues an update operation on a cached Oracle Database table if it results in the usage of the Oracle cache administration user's tablespace to exceed 80 percent:

```
% ttIsql "DSN=cache1;UID=cacheadmin;PwdWallet=/wallets/cacheadminwallet"
Command> CALL ttCacheConfig('AutoRefreshLogTblSpaceUsagePCT',,,'80');
```

To determine the current Oracle cache administration user's tablespace usage warning threshold, call `ttCacheConfig` passing only the `AutoRefreshLogTblSpaceUsagePCT` string to the *`Param`* parameter:

```
Command> CALL ttCacheConfig('AutoRefreshLogTblSpaceUsagePCT');
< AutoRefreshLogTblSpaceUsagePCT, <NULL>, <NULL>, 80 >
```

The default Oracle cache administration user's tablespace usage warning threshold is 0 percent which means that no warning is returned to the application regardless of the tablespace usage. The Oracle cache administration user's tablespace usage warning threshold applies to all TimesTen databases that cache tables from the same Oracle database and have the same Oracle cache administration user name setting.

See ttCacheConfig in the *Oracle TimesTen In-Memory Database Reference*.

Recovering from a Full Tablespace

By default, when the Oracle cache administration user's tablespace is full, an error is returned to the application when it attempts a DML operation, such as an `UPDATE`, `INSERT` or `DELETE` statement, on a particular cached Oracle Database table.

Rather than TimesTen returning an error to the Oracle Database application when the Oracle cache administration user's tablespace is full, you can configure TimesTen to delete existing rows from the change log tables to make space for new rows when an update operation is issued on a particular cached Oracle Database table. If some of the deleted change log table rows have not been applied to the cache tables, a full autorefresh operation is performed on those cache tables in each TimesTen database that contains the tables upon the next autorefresh cycle.

Call the `ttCacheConfig` built-in procedure as the TimesTen cache administration user from any of the TimesTen databases that cache tables from the Oracle database. Pass the `TblSpaceFullRecovery` string to the *`Param`* parameter, the owner and name of the cached Oracle Database table to the *`tblOwner`* and *`tblName`* parameters, respectively, on which you want to configure an action to take if the Oracle cache administration user's tablespace becomes full, and the action itself as a string to the *`Value`* parameter.

The following are the valid actions:

- `None`: Return an Oracle Database error to the application when an update operation is issued on the cached Oracle Database table. This is the default action.
- `Reload`: Delete rows from the change log table and perform a full autorefresh operation on the cache table upon the next autorefresh cycle when an update operation is issued on the cached Oracle Database table.

The following example configures an action when the Oracle cache administration user's tablespace becomes full. In this example, rows are deleted from the change log table and a full autorefresh operation is performed on the cache table upon the next autorefresh cycle when an update operation is issued on the `sales.customer` cached Oracle Database table while the Oracle cache administration user's tablespace is full:

```
% ttIsql "DSN=cache1;UID=cacheadmin;PwdWallet=/wallets/cacheadminwallet"
Command> CALL ttCacheConfig('TblSpaceFullRecovery','sales','customer','Reload');
```

To determine the current action to take when an update operation is issued on a particular cached Oracle Database table if the Oracle cache administration user's tablespace is full, call `ttCacheConfig` passing only the `TblSpaceFullRecovery` string to the *`Param`* parameter, and the owner and name of the cached Oracle Database table to the *`tblOwner`* and *`tblName`* parameters, respectively:

```
Command> CALL ttCacheConfig('TblSpaceFullRecovery','sales','customer');
< TblSpaceFullRecovery, SALES, CUSTOMER, reload >
```

The action to take when update operations are issued on a cached Oracle Database table while the Oracle cache administration user's tablespace is full applies to all TimesTen databases that cache tables from the same Oracle database and have the same Oracle cache administration user name setting.

See ttCacheConfig in the *Oracle TimesTen In-Memory Database Reference*.

Backing Up and Restoring a TimesTen Database with Cache Groups

TimesTen databases containing cache groups can be backed up and restored with either the `ttBackup` or `ttMigrate` utilities.

- If the restored database connects to the same backend Oracle database, then use the `ttBackup` and `ttRestore` utilities, then drop and recreate all cache groups in the restored TimesTen database. If they are static cache groups, you may be required to reload them. For dynamic cache groups, the reload is optional as data is pulled in from the Oracle database as it is referenced.

Note

If another TimesTen database is used to connect to the original backend Oracle database (and now no longer connects) and if all cache groups in the TimesTen database were not cleanly dropped, then run the `cacheCleanUp.sql` SQL*Plus script against the original Oracle database to remove all leftover objects. Specify the host and path for the original TimesTen database.

See SQL*Plus Scripts for Cache.

- If the restored database connects to a different backend Oracle database than what it had originally connected with, then perform one of the following:
  - Backing Up and Restoring Using the ttBackup and ttRestore Utilities
  - Backing Up and Restoring TimesTen Database with the ttMigrate Utility

Backing Up and Restoring Using the ttBackup and ttRestore Utilities

When you use the `ttBackup` utility, it backs up the TimesTen database with all of its data at a particular time.

Thus, if you want to use these cache groups again, restoring this backup requires additional action as the restored data within the cache groups are out of date and out of sync with the data in the backend Oracle database. See Back Up, Restore, and Migrate Data in TimesTen Classic in the *Oracle TimesTen In-Memory Database Installation, Migration, and Upgrade Guide*.

Note

See ttBackup and ttRestore in the *Oracle TimesTen In-Memory Database Reference*.

If the restored database connects to a different backend Oracle database than what it had originally connected with and you want to use the `ttBackup` and `ttRestore` utilities to backup and restore your database, then perform the following:

1. Run the `ttBackup` utility command to backup the database and its objects into a binary file. For example, to backup the `cache1` database using the `/tmp/dump` directory for temporary storage:

   ```
   % ttBackup -dir /tmp/dump -connstr "DSN=cache1"
   ```

2. Stop the cache agent.

   ```
   % ttIsql -connstr "DSN=cache1;UID=cacheadmin;PwdWallet=/wallets/cacheadminwallet"
   Command> call ttCacheStop;
   ```

3. (Optional) Drop all cache groups from the TimesTen database. Since the database still exists with its cache groups, TimesTen recommends that you drop the cache groups.

   ```
   Command> DROP CACHE GROUP readcache;
   Command> exit;
   Disconnecting...
   Done.
   ```

4. Destroy the database before restoring in the same or another location.

   ```
   % ttDestroy cache1
   ```

5. Clean up objects on the Oracle database. Run the *timesten_home*`/install/oraclescripts/cacheCleanUp.sql` SQL*Plus script from the current database install as the Oracle cache administration user to drop the Oracle Database objects used to implement autorefresh operations. The host name of the TimesTen system and the TimesTen database (including its path) are passed as arguments to the `cacheCleanUp.sql` script.

   You can run the `ttCacheInfo` utility or the `cacheInfo.sql` script as the Oracle cache administration user to determine the host and database names.

```
% cd timesten_home/install/oraclescripts
% sqlplus cacheadmin/orapwd
SQL> @cacheCleanUp "sys1" "/users/OracleCache/cache1"

*****************************OUTPUT**************************************
Performing cleanup for object_id: 69959 which belongs to table : CUSTOMER
Executing: delete from tt_07_agent_status where host = sys1 and datastore =
/users/OracleCache/cache1 and object_id = 69959
Executing: drop table tt_07_69959_L
Executing: drop trigger tt_07_69959_T
Executing: delete from tt_07_user_count where object_id = object_id1
Performing cleanup for object_id: 69966 which belongs to table : ORDERS
Executing: delete from tt_07_agent_status where host = sys1 and datastore =
/users/OracleCache/cache1 and object_id = 69966
Executing: drop table tt_07_69966_L
Executing: drop trigger tt_07_69966_T
Executing: delete from tt_07_user_count where object_id = object_id1
**************************************************************************
```

6. Restore the database with the `ttRestore` utility and then delete the temporary directory.

```
% ttRestore -dir /tmp/dump -connstr "DSN=cache1"
Restore started ...
Restore complete

% rm -r /tmp/dump
```

7. In order to re-synchronize the data within the cache groups, you must drop and recreate the cache groups:

   a. Connect to the TimesTen database providing the cache administration user credentials.

   b. Drop the cache groups that were restored with the `ttRestore` utility. Because the data is out of sync, you may see errors.

   c. Register the Oracle cache administration user name and password with the `ttCacheUidPwdSet` built-in procedure.

   d. Start the cache agent.

   e. Recreate and, if required, reload the cache groups.

```
% ttIsql -connstr "DSN=cache1;UID=cacheadmin;PwdWallet=/wallets/cacheadminwallet"

Command> DROP CACHE GROUP readcache;
Command> call ttCacheUidPwdSet('cacheadmin','orapwd');
Command> call ttCacheStart;
Command> CREATE READONLY CACHE GROUP readcache
        AUTOREFRESH INTERVAL 5 SECONDS
        FROM sales.readtab
        (keyval NUMBER NOT NULL PRIMARY KEY, str VARCHAR2(32));
Command> LOAD CACHE GROUP readcache COMMIT EVERY 256 ROWS;
2 cache instances affected.
```

Note

If the restored TimesTen database is not able to connect to any backend Oracle database, then TimesTen cannot autorefresh the data for the read-only cache groups.

Backing Up and Restoring TimesTen Database with the ttMigrate Utility

The `ttMigrate` utility saves tables and indexes from a TimesTen database into a binary file.

When a cache group is migrated and included in the binary file, it includes the cache group definition and schema; however, the data of the cache group is not migrated.

> **Note**
>
> See Back Up, Restore, and Migrate Data in TimesTen Classic in the *Oracle TimesTen In-Memory Database Installation, Migration, and Upgrade Guide* and ttMigrate in the *Oracle TimesTen In-Memory Database Reference*.

If the restored database connects to a different backend Oracle database than what it had originally connected with and you want to use the `ttMigrate` utility for backing up and restoring the database, then perform the following:

1. Stop the cache agent.

   ```
   % ttIsql -connstr "DSN=cache1;UID=cacheadmin;PwdWallet=/wallets/cacheadminwallet"
   Command> call ttCacheStop;
   Command> exit
   Disconnecting...
   Done.
   ```

2. Run the `ttMigrate -c` utility command to save the database and its objects into a binary file.

   ```
   % ttMigrate -c "DSN=cache1" cache1.ttm
   ...
   Saving profile DEFAULT
   Profile successfully saved.

   Saving profile SYSTEM
   Profile successfully saved.

   Saving user PUBLIC
   User successfully saved.

   Saving user CACHEADMIN
   User successfully saved.

   Saving user sales
   User successfully saved.

   Saving cache group CACHEADMIN.READCACHE
     Saving cached table SALES.READTAB
   Cache group successfully saved.
   ```

3. (Optional) Drop all cache groups from the TimesTen database. Since the database still exists with its cache groups, TimesTen recommends that you drop the cache groups. When you drop all cache groups before destroying the TimesTen database, all metadata on the Oracle Database for these cache groups is deleted. However, if you use the `cacheCleanup.sql` script in a future step, this script deletes the metadata on the Oracle Database.

   You may see errors reported, which can be ignored.

```
Command> DROP CACHE GROUP readcache;
Command> exit
Disconnecting...
Done.
```

4. Destroy the TimesTen database.

   ```
   % ttDestroy cache1
   ```

5. Clean up objects on the Oracle database: If you did not drop that cache groups in an earlier step, you can run the *timesten_home*`/install/oraclescripts/cacheCleanUp.sql` SQL*Plus script as the Oracle cache administration user to drop the Oracle Database objects used to implement autorefresh operations. The host name of the TimesTen system and the TimesTen database (including its path) are passed as arguments to the `cacheCleanUp.sql` script.

   You can run the `ttCacheInfo` utility or the `cacheInfo.sql` script as the Oracle cache administration user to determine the host and database names.

   ```
   % cd timesten_home/install/oraclescripts
   % sqlplus cacheadmin/orapwd
   SQL> @cacheCleanUp "sys1" "/users/OracleCache/cache1"

   *****************************OUTPUT**************************************
   Performing cleanup for object_id: 69959 which belongs to table : CUSTOMER
   Executing: delete from tt_05_agent_status where host = sys1 and datastore =
   /users/OracleCache/cache1 and object_id = 69959
   Executing: drop table tt_05_69959_L
   Executing: drop trigger tt_05_69959_T
   Executing: delete from tt_05_user_count where object_id = object_id1
   Performing cleanup for object_id: 69966 which belongs to table : ORDERS
   Executing: delete from tt_05_agent_status where host = sys1 and datastore =
   /users/OracleCache/cache1 and object_id = 69966
   Executing: drop table tt_05_69966_L
   Executing: drop trigger tt_05_69966_T
   Executing: delete from tt_05_user_count where object_id = object_id1
   **************************************************************************
   ```

6. Create and restore the database:

   a. Create the TimesTen database with a first connection request.

   b. Create the cache table user and the TimesTen cache administration user. Grant appropriate privileges to these users.

      Note

      Depending on which TimesTen release you are migrating from, the users and privileges may or may not be migrated. See ttMigrate in the *Oracle TimesTen In-Memory Database Reference*.

      ```
      % ttIsql cache1
      Command> CREATE USER cacheadmin IDENTIFIED BY timesten;
       User created.

      Command> GRANT CREATE SESSION, CACHE_MANAGER, CREATE ANY TABLE TO cacheadmin;
      Command> CREATE USER sales IDENTIFIED BY timesten;
      User created.

      Command> exit
      ```

```
Disconnecting...
Done.
```

c. Register the Oracle cache administrator user name and password with the `ttCacheUidPwdSet` built-in procedure.

```
% ttIsql -connstr "DSN=cache1;UID=cacheadmin;PwdWallet=/wallets/cacheadminwallet"
Command> call ttCacheUidPwdSet('cacheadmin','orapwd');
Command> exit
Disconnecting...
Done.
```

7. Restore the database from the saved binary file with the `ttMigrate -r` utility command.

```
% ttMigrate -r -relaxedUpgrade -cacheuid cacheadmin -cachepwd orapwd
 -connstr "DSN=cache1;UID=cacheadmin;PwdWallet=/wallets/cacheadminwallet"
 cache1.ttm
...
Restoring profile DEFAULT
Profile successfully restored.

Restoring profile SYSTEM
Profile successfully restored.

Restoring user CACHEADMIN
  Restoring privileges...
  Privileges restored.
User successfully restored.

Restoring user sales
  Restoring privileges...
  Privileges restored.
User successfully restored.

Restoring cache group CACHEADMIN.READCACHE
  Restoring cached table SALES.READTAB
  1/1 cached table restored.
Cache group successfully restored.
```

8. Connect to the restored database and reset the cache autorefresh state:

   a. Connect to the TimesTen database with ttIsql.

   b. Start the cache agent.

   c. Alter the cache groups to set autorefresh state to `ON`.

```
% ttIsql -connstr "DSN=cache1;UID=cacheadmin;PwdWallet=/wallets/cacheadminwallet"
Command> call ttCacheStart;
Command> ALTER CACHE GROUP readcache SET AUTOREFRESH STATE ON;
```

Note

If the restored TimesTen database is not able to connect to any backend Oracle database, then TimesTen cannot autorefresh the data for the read-only cache groups.

Migrating the Oracle Database Requires Cleaning Up Cache Objects

When you set up cache, cache objects and metadata are installed on both the TimesTen and Oracle databases. When you migrate the Oracle database, the cache metadata on the back-end Oarcle database is no longer correct. Thus, before you migrate your Oracle database, you must clean up cache objects and metadata from both the TimesTen and Oracle databases.

1. Drop all cache groups with the `DROP CACHE GROUP` statement.

   The following example connects as the TimesTen cache administration user to the `cache1` database and drops the `customer_orders` cache group.

   ```
   % ttIsql "DSN=cache1;UID=cacheadmin;PwdWallet=/wallets/cacheadminwallet"
   Command> DROP CACHE GROUP customer_orders;
   ```

   See Dropping a Cache Group.

2. Clean up cache on both TimesTen and Oracle databases. If you do not clean up cache on both TimesTen and Oracle databases, you will encounter cache errors.

   - When using TimesTen, run the *`timesten_home`*`/install/oraclescripts/cacheCleanUp.sql` SQL*Plus script as the Oracle cache administration user to clean up the Oracle database cache objects and metadata used for cache operations. The host name of the TimesTen system and the TimesTen database (including its path) are passed as arguments to the `cacheCleanUp.sql` script.
   - The following example shows the TimesTen database contains one read-only cache group `customer_orders` with cache tables `sales.customers` and `sales.orders`. This example uses the `cacheCleanUp.sql` script for a TimesTen system. It drops the change log tables and triggers associated with the two cache tables.

     ```
     % cd timesten_home/install/oraclescripts
     % sqlplus cacheadmin/orapwd
     SQL> @cacheCleanUp "sys1" "/disk1/databases/database1"

     *****************************OUTPUT**************************************
     Performing cleanup for object_id: 69959 which belongs to table : CUSTOMERS
     Executing: delete from tt_07_agent_status where host = sys1 and datastore =
     /disk1/databases/database1 and object_id = 69959
     Executing: drop table tt_07_69959_L
     Executing: drop trigger tt_07_69959_T
     Executing: delete from tt_07_user_count where object_id = object_id1
     Performing cleanup for object_id: 69966 which belongs to table : ORDERS
     Executing: delete from tt_07_agent_status where host = sys1 and datastore =
     /disk1/databases/database1 and object_id = 69966
     Executing: drop table tt_07_69966_L
     Executing: drop trigger tt_07_69966_T
     Executing: delete from tt_07_user_count where object_id = object_id1
     **************************************************************************
     ```

3. Perform the Oracle database migration.
4. If the Oracle migration eliminated the Oracle cache administration user and its tablespace, then set up cache again on the Oracle database. Check the TimesTen database to ensure that the TimesTen cache administration user and the schema user still exists. See Setting Up a Caching Infrastructure.
5. Recreate the cache groups. See Defining Cache Groups.

7

Cache Performance

The following sections contain information about cache performance.

Note

See Monitoring Cache Groups with Autorefresh and Poor Autorefresh Performance in the *Oracle TimesTen In-Memory Database Monitoring and Troubleshooting Guide* for extensive information about monitoring autorefresh operations and improving autorefresh performance.

See AWT Performance Monitoring and Possible Causes of Poor AWT Performance in the *Oracle TimesTen In-Memory Database Monitoring and Troubleshooting Guide*.

Dynamic Load Performance

Dynamic loading of a single cache instance based on a primary key search of the root table has faster performance than primary key searches on a child table or foreign key searches on a child table.

See Dynamic Cache Groups.

Dynamic loading of multiple cache instances may have faster performance than loading single cache instances. See Dynamically Loading Multiple Cache Instances.

If you combine dynamic load operations with autorefresh operations, you may experience some contention. See Improving Performance for Autorefresh Operations for details on how to improve your performance in this situation.

There can be a performance cost when opening a new connection for a dynamic load operation. You can reduce the cost of opening new connections by creating a cache connection pool. You may want to use a cache connection pool if your application requires frequent dynamic load operations that would create too many open connections to the Oracle database. See Managing a Cache Connection Pool to the Oracle Database for Dynamic Load Requests.

Managing a Cache Connection Pool to the Oracle Database for Dynamic Load Requests

When a qualifying `SELECT` statement is issued on any dynamic read-only cache table and the data does not exist in the cache table (but does exist in the base Oracle database table), this results in a cache miss. After which, Timesten performs a dynamic load to retrieve the data from the Oracle database (either over an existing or a new connection to the Oracle database) and inserts the rows into the cache group.

There can be a performance cost when opening a new connection for the dynamic load. You can reduce the cost of opening new client connections by creating a cache connection pool.

By default, a client connection to the Oracle database remains open until the application's connection to TimesTen is closed. When the application initiates a dynamic load, each client connection is associated with a connection to the Oracle database (when using cache). If you use several client connections, TimesTen's requests for new client connections to the Oracle database could exceed the maximum number of client connections allowed to the Oracle database.

Applications can have multiple dynamic load requests spread across multiple client connections to the Oracle database, which could result in too many open client connections to the back-end Oracle database. Alternately, there could be applications across multiple TimesTen databases performing dynamic loads against the same Oracle database. For client/server applications with multiple client connections per server, you can configure TimesTen to use the cache connection pool for all client connections that are used for dynamic load operations from the Oracle database. The cache connection pool can only be utilized by an application using a client connection as the pooled connections are shared across all client connections.

Dynamic load requests will use an existing client connection to the Oracle database from the cache connection pool (rather than creating a new one) to reduce the total number of open client connections. Once the dynamic load request completes, the connection is returned to the cache connection pool.

Using an existing connection from the cache connection pool optimizes your application performance by:

- Reducing the cost of starting a dedicated Oracle server process (or thread) for each newly requested connection.
- Reducing the total number of Oracle server processes (threads) by sharing them amongst client connections rather than having each process (thread) dedicated to a single connection. However, if there are no available client connections in the cache connection pool, the dynamic load operation waits until a connection is added to the pool.
- Enabling the sharing of session level server resources, such as memory, between client connections.

Once the connection is returned to the cache connection pool, the application logically sees the client connection as disconnected. Thus, if an application contains passthrough statements (DDL or DML statements performed in the Oracle database), any passthrough statement must be committed or rolled back before the dynamic load is requested or an error is thrown. You can set autocommit to `ON` or run the commit or rollback within the transaction before the dynamic load.

Note

If an application runs a higher than expected number of dynamic load requests and performance is critical, then you might consider either:

- Removing or minimizing passthrough statements with DDL or DML statements (which can slow down performance) from any application using the cache connection pool.
- Maintaining a completely separate client connection directly to the Oracle Database to run its SQL directly against the Oracle database, rather than using passthrough statements to run SQL indirectly through TimesTen.

To decide whether to use the cache connection pool, evaluate if any applications request a high number of dynamic load operations from the Oracle database (resulting in too many open client connections to the Oracle database).

The following sections describe how to use the cache connection pool for your dynamic read-only cache groups:

- Enable the Cache Connection Pool
- Size the Cache Connection Pool
- Use the ChildServer Connection Attribute to Identify a Child Server Process
- Dynamically Applying Cache Connection Pool Sizing Modifications
- Example Demonstrating Management of the Cache Connection Pool
- Limiting the Number of Connections to the Oracle Database
- Restrictions for the Cache Connection Pool

Enable the Cache Connection Pool

You can specify that TimesTen creates a cache connection pool on the TimesTen server when it starts up.

If a cache connection pool is created, then a dynamic load request from a client/server connection acquires a connection from the cache connection pool, performs the dynamic load, and returns the connection to the cache connection pool after the dynamic load request completes. The cache connection pool is destroyed when the TimesTen server shuts down.

Note

The cache connection pool can only be initiated from client-server applications (using multithreaded mode) and is used only for dynamic loads initiated for dynamic read-only cache groups.

To enable client/server connection requests to use the cache connection pool, an application must specify the following connection attributes when connecting.

- `MaxConnsPerServer` connection attribute: This connection attribute sets the maximum number of client/server connections that can be created for each child server process. When the value is set to > 1, each TimesTen child server can handle multiple client connections where each client/server connection is multithreaded. You can only use the cache connection pool with a multithreaded client/server connection.

When `MaxConnsPerServer` connection attribute is set to 1, TimesTen creates one single-threaded client/server connection for each child server process.

- `ServersPerDSN` connection attribute: Value designates the number of child server processes to spawn for the TimesTen server. Default is 1.

  Each new incoming connection spawns a new child server process up to the value specified by the `ServersPerDSN` connection attribute. When the maximum number of child server processes is reached, the existing child server processes handle multiple connections (up to the number specified in `MaxConnsPerServer`) in a round-robin method. That is, if you specify `ServersPerDSN` = 2 and `MaxConnsPerServer` = 3, then the first two connections would spawn two child server processes. The third through the sixth connections would be handled by these child server processes, where each child server process would service every other connection.

  Once all of the child server processes have the maximum allowed number of connections, the next incoming connection starts a new set of child server processes.

  The `ServersPerDSN` and `MaxConnsPerServer` connection attributes are used to designate how to distribute connections across multiple child server processes.

- `UseCacheConnPool` connection attribute: Must be enabled (set to 2) to use the cache connection pool. When the `UseCacheConnPool` connection attribute is enabled, the cache connection pool is created and used for dynamic load operations initiated by multithreaded client/server connections. If the `UseCacheConnPool` connection attribute is disabled (set to 0), then the cache connection pool is not created and the dynamic load operations perform using the existing behavior. See UseCacheConnPool in the *Oracle TimesTen In-Memory Database Reference*.

Note

You may also want to limit the number of connections to the Oracle database. See Limiting the Number of Connections to the Oracle Database.

The following example specifies connection attributes for the cache connection pool in the DSN definition:

The `cache1` DSN definition in the `sys.odbc.ini` file specifies `UseCacheConnPool=2`, `ServersPerDSN=2` and `MaxConnsPerServer=3`.

```
[cache1]
DataStore=/users/OracleCache/database1
PermSize=64
OracleNetServiceName=oracledb
DatabaseCharacterSet=AL32UTF8
UseCacheConnPool=2
ServersPerDSN=2
MaxConnsPerServer=3
```

Alternatively, you can specify both of the connection attributes on the command line when connecting from the application.

```
ttIsql
"DSN=cache1;OracleNetServiceName=oracledb;UseCacheConnPool=2;ServersPerDSN=2;MaxConnsPerS
erver=3"
```

Note

See the MaxConnsPerServer, ServersPerDSN, and UseCacheConnPool sections in the *Oracle TimesTen In-Memory Database Reference*.

Size the Cache Connection Pool

You can appropriately size the cache connection pool to avoid contention for connections with the `ttCacheConnPoolSet` built-in procedure.

The `ttCacheConnPoolSet` built-in procedure saves the values of these parameters in the Oracle database, which are then used as the default values when restarting the TimesTen server. Once applied to each TimesTen server, the values specified are used for the cache connection pool across all client/server applications for a TimesTen database.

If you want to modify these values after the TimesTen server starts, you can change the cache connection pool sizing parameters on the Oracle database using the `ttCacheConnPoolSet` built-in procedure. After which, you can re-initialize the TimesTen server by either:

- Restarting the TimesTen server to re-initialize the server (and all child server processes) with the new sizing parameters.
- Dynamically re-initializing each TimesTen server with the cache connection pool parameters saved on the Oracle database with the `ttCacheConnPoolApply` built-in procedure. See Dynamically Applying Cache Connection Pool Sizing Modifications.

You can run the `ttCacheConnPoolSet` built-in procedure from a direct connection, a single-threaded client/server connection or a multithreaded client/server connection.

Note

See the ttCacheConnPoolSet in the *Oracle TimesTen In-Memory Database Reference*.

For example, the following initiates the minimum and maximum number of pooled connections to be between 10 and 32 connections and the increment is 1. The maximum idle time by the client is set to 10 seconds. And all dynamic load operations will wait for an available connection from the cache connection pool.

```
Command> call ttCacheConnPoolSet(10, 32, 1, 10, 0);
```

Set the minimum and maximum size of the cache connection pool to levels where connections are available when needed. If no connections are available in the pool, dynamic load operations stall until a connection from the pool is available (unless you set `ConnNoWait=1`). If a connection to the Oracle database times out, you receive an error denoting a loss of the connection, sometimes requiring a rollback on TimesTen.

You can query what the cache connection pool parameters are with the `ttCacheConnPoolGet` built-in procedure.

See Example Demonstrating Management of the Cache Connection Pool.

Use the ChildServer Connection Attribute to Identify a Child Server Process

In a client/server environment, TimesTen can create multiple TimesTen child server processes to handle incoming requests from clients. You can use the `ChildServer` connection attribute to identify a specific child server process when performing certain cache connection pool administrative functions, such as the `ttCacheConnPoolGet('current')` or `ttCacheConnPoolApply` built-in procedures.

The target child server process is identified by a value specified using the `ChildServer=`*n* connection attribute, where *n* is a number ranging from 1 to the number of running child server processes. When you specify the `ChildServer` connection attribute, then the client process connects using the identified child server process. If the attribute is not specified, then the client process connects using a randomly selected child server process.

See ttCacheConnPoolApply and ttCacheConnPoolGet in the *Oracle TimesTen In-Memory Database Reference*. See Example Demonstrating Management of the Cache Connection Pool.

Dynamically Applying Cache Connection Pool Sizing Modifications

The cache connection pool parameters are saved in the Oracle database, which are used to initialize the cache connection pool for the TimesTen database every time that the TimesTen server restarts. The sizing is set on the Oracle database with the `ttCacheConnPoolSet` built-in procedure. This sizing applies to each TimesTen server and child server processes when started.

However, you can dynamically resize the cache connection pool parameters for each child server process (while the database is running) with the `ttCacheConnPoolApply` built-in procedure.

- Execute the `ttCacheConnPoolSet` built-in procedure to set a new set of parameters that are stored on the Oracle database.
- Connect to the child server process.
- Dynamically associate the new set of cache connection pool parameters for this particular child server process with the `ttCacheConnPoolApply` built-in procedure.

For example, the following connects to the child server process identified as 1 and applies the new cache connection pool configuration to this child server process. It does the same process for child server process 2 (given that `ServersPerDSN`=2).

```
Command> connect "DSN=cache1;ChildServer=1;";
Command> call ttCacheConnPoolApply;
Command> disconnect;

Command> connect "DSN=cache1;ChildServer=2;";
Command> call ttCacheConnPoolApply;
Command> disconnect;
```

You can run the `ttCacheConnPoolApply` built-in procedure only from a multithreaded client/server connection.

If the cache connection pool fails, you can recreate the pool by running the `ttCacheConnPoolApply` built-in procedure from any child server process.

See Example Demonstrating Management of the Cache Connection Pool.

Example Demonstrating Management of the Cache Connection Pool

This example shows how to set new values for the cache connection pool and apply them to two separate child server processes.

This example uses the cache1 DSN as shown in Enable the Cache Connection Pool that enables the cache connection pool. It also assumes that you have set the cache administrator and password as described in Registering the Cache Administration User Name and Password.

```
/* Since ServerPerDSN is set to two and MaxConnsPerServer is set to 3, the first
 and second connections spawn off both child server processes. And then you can
 create four more connections to reach the MaxConnsPerServer maximum, which are
 routed by the TimesTen server to the appropriate child server process (using a
 round robin method).*/
Command> connect "DSN=cache1;" as conn1;
Command> connect "DSN=cache1;" as conn2;
Command> connect "DSN=cache1;" as conn3;
Command> connect "DSN=cache1;" as conn4;
Command> connect "DSN=cache1;" as conn5;
Command> connect "DSN=cache1;" as conn6;

Command> use conn1;

/* Query the values for the cache connection pool that are saved on the Oracle database*/
Command> call ttCacheConnPoolGet('saved');
< 1, 10, 1, 10, 0, -1, -1, -1>

/* Change the configuration of the cache connection pool */
Command> call ttCacheConnPoolSet(1, 20, 1, 10, 0);

/* Query existing values for cache connection pool saved on the Oracle data base.
 Since these are the saved values, this returns -1 for OpenCount, BusyCount
 and LastOraErr. */
Command> call ttCacheConnPoolGet('saved');
< 1, 20, 1, 10, 0, -1, -1, -1 >

/* Query existing values for the current cache connection pool on this TimesTen database
*/
Command> call ttCacheConnPoolGet('current');
< 1, 10, 1, 10, 0, 1, 0, 0 >

/* Connect to the child server process 1 using the ChildServer=1 connection
 attribute. Apply the saved values as the current values to the cache connection
 pool for child server process identified as ChildServer 1. */
Command> connect "DSN=cache1;ChildServer=1;";
Command> call ttCacheConnPoolApply;
Command> disconnect;

/* Connect to the child server process 1 using the ChildServer=1 connection
 attribute. Apply the saved values as the current values to the cache connection
 pool for child server process identified as ChildServer 2. */
Command> connect "DSN=cache1;ChildServer=2;";
Command> call ttCacheConnPoolApply;
Command> disconnect;

/* Query values for the cache connection pool in ChildServer 1 */
Command> use conn1;
Command> call ttCacheConnPoolGet('current');
< 1, 20, 1, 10, 0, 1, 0, 0 >
```

```
/* Query values for the cache connection pool in ChildServer 2 */
Command> use conn2;
Command> call ttCacheConnPoolGet('current');
< 1, 20, 1, 10, 0, 1, 0, 0 >
```

Limiting the Number of Connections to the Oracle Database

You can optimize performance while ensuring a limit to the number of connections to the Oracle database.

Tuning the total number of connections depends on the following:

> **Note**
>
> These calculations assume that all connections to the Oracle database are client/server connections using a multithreaded server. The connections referred to in the rest of this section are only those used for dynamic load operations. There can be other connections from TimesTen to the Oracle database that are not accounted for in these calculations.

- **N**: The number of connections to the Oracle database.
- **P**: The limit on the number of connections for each cache connection pool, where each TimesTen child server process has a cache connection pool. You can set this with the `MaxSize` cache connection pool parameter using the `ttCacheConnPoolSet` built-in procedure.
- **S**: The maximum number of child server processes that can be spawned for new connections. Currently, there is no direct way to limit the number of child server processes. Indirectly, you can influence the number of child server processes by setting the `MaxConnsPerServer` and `Connections` connection attributes. You should measure **S** on your system when your system is in a steady state that represents the typical operating conditions.
- **M**: The maximum number of connections for each child server process, which you can set with the `MaxConnsPerServer` connection attribute.
- **D**: The maximum number of connections to a DSN, which is set with the `Connections` connection attribute.

The number of connections (**N**) to the Oracle database is equal to the maximum number of TimesTen child server processes (**S**) times the maximum number of connections for each cache connection pool (**P**).

```
N=S*P
```

The maximum number of connections (**D**) to the DSN is equal to the maximum number of connections for each child server process (**M**) times the maximum number of TimesTen child server processes (**S**).

```
D=M*S
```

With the above calculation, you can also state:

```
S=D/M
```

Since there is no hard limit that we can configure for the number of TimesTen child server processes, we substitute for **S** to get the following equation:

```
N=(D*P)/M
```

Assuming that all connections to the Oracle database are client/server connections, then the maximum number of connections to the Oracle database arising from cache connection pools is equal to the maximum number of connections to the DSN (set by the `Connections` connection attribute) times the number of connections for each cache connection pool (set by the `MaxSize` cache connection pool parameter), which is then divided by the maximum number of connections for each child server process (set by the `MaxConnsPerServer` connection attribute).

Restrictions for the Cache Connection Pool

There are restrictions when using the cache connection pool.

- You cannot use the cache connection pool in conjunction with the Oracle Database Resident Connection Pooling feature.
- The cache connection pool is only supported for multithreaded client/server connections, where the `MaxConnsPerServer` connection attribute must be greater than 1.
- The cache connection pool is only used for dynamic load operations for dynamic read-only cache groups.

Improving AWT Throughput

There are best practice methods to improve throughput for AWT cache groups.

- Improving AWT Throughput with Parallel Propagation to the Oracle Database
- Improving AWT Throughput with SQL Array Processing

Improving AWT Throughput with Parallel Propagation to the Oracle Database

To improve throughput for an AWT cache group, you can configure multiple threads that act in parallel to propagate and apply transactional changes to the Oracle database. Parallel propagation enforces transactional dependencies and applies changes in AWT cache tables to Oracle Database tables in commit order.

Parallel propagation is supported for AWT cache groups with the following configurations:

- AWT cache groups involved in an active standby pair replication scheme
- AWT cache groups in a single TimesTen database (without a replication scheme configuration)
- AWT cache groups configured with any aging policy

The following data store attributes enable parallel propagation and control the number of threads that operate in parallel to propagate changes from AWT cache tables to the corresponding Oracle Database tables:

- `ReplicationApplyOrdering` enables parallel propagation by default.
- `ReplicationParallelism` defines the number of transmitter threads on the source database and the number of receiver threads on the target database for parallel replication in a replication scheme. This value can be between 2 and 32 when used solely for parallel replication. The default is 1. In addition, the value of `ReplicationParellelism` cannot exceed half the value of `LogBufParallelism`.

- `CacheAWTParallelism`, when set, determines the number of threads used in parallel propagation of changes from AWT cache tables to the Oracle Database tables. Set this attribute to a number from 2 to 31. The default is 1.

Parallel propagation for an AWT cache group is configured with one of the following scenarios:

- `ReplicationApplyOrdering` is set to 0 and `ReplicationParallelism` is greater than 1.

  If you do not set `CacheAWTParallelism`, the number of threads that apply changes to Oracle Database is 2 times the setting for `ReplicationParallelism`. For example, if `ReplicationParallelism=3`, the number of threads that apply changes to Oracle Database tables is 6. In this case, `ReplicationParallelism` can only be set from 2 to 16; otherwise, twice the value would exceed the maximum number of 31 threads for parallel propagation. If the value is set to 16, the maximum number of threads defaults to 31.

- `ReplicationApplyOrdering` is set to 0, `ReplicationParallelism` is equal to or greater than 1, and `CacheAWTParallelism` is greater than 1. The value for `CacheAWTParallelism` must be greater than or equal to the value set for `ReplicationParallelism` and less than or equal to 31.

  If `CacheAWTParallelism` is not specified, then `ReplicationParallelism` is used to determine the number of threads that are used for parallel propagation to Oracle Database. However, since this value is doubled for parallel propagation threads, you can only set `ReplicationParallelism` to a number from 2 to 16. If the value is set to 16, the maximum number of threads defaults to 31.

  If both `ReplicationParallelism` and `CacheAWTParallelism` attributes are set, the value set in `CacheAWTParallelism` configures the number of threads used for parallel propagation. The setting for `CacheAWTParallelism` determines the number of apply threads for parallel propagation and the setting for `ReplicationParallelism` determines the number of threads for parallel replication. Thus, if `ReplicationParallelism` is set to 4 and `CacheAWTParallelism` is set to 6, then the number of threads that apply changes to Oracle Database tables is 6. This enables the number of threads used to be different for parallel replication and parallel propagation to Oracle Database tables.

Note

See Configuring Parallel Replication in the *Oracle TimesTen In-Memory Database Replication Guide*. See ReplicationApplyOrdering, ReplicationParallelism, and CacheAWTParallelism in the *Oracle TimesTen In-Memory Database Reference*.

These data store attributes are interrelated. Table 7-1 shows the result with the combination of the various possible attribute values.

Table 7-1 Results of Parallel Propagation Data Store Attribute Relationships

ReplicationApply Ordering	ReplicationParallelism	CacheAWTParallelism	Number of Parallel Propagation Threads
Set to 0, which enables parallel propagation	Set to > 1 for multiple tracks and <= 16.	Not specified.	Set to twice the value of `ReplicationParallelism`.

Table 7-1 (Cont.) Results of Parallel Propagation Data Store Attribute Relationships

ReplicationApply Ordering	ReplicationParallelism	CacheAWTParallelism	Number of Parallel Propagation Threads
Set to 0, which enables parallel propagation	Set to > 16 and <= 32 for multiple tracks.	Not specified.	Error is thrown. If `CacheAWTParallelism` is not set, then 2 times the value set in `ReplicationParallelism` specifies the number of threads. Thus, in this case, `ReplicationParallelism` cannot be greater than 16.
Set to 0, which enables parallel propagation	Set to > 1 and <= 32 for multiple tracks.	Set to >= to `ReplicationParallelism`.	Set to number specified by `CacheAWTParallelism`.
Set to 0, which enables parallel propagation	Set to > 1 and <= 32 for multiple tracks.	Set to < `ReplicationParallelism`.	Error is thrown at database creation. The `CacheAWTParallelism` must be set to a value greater than or equal to `ReplicationParallelism`.
Set to 0, which enables parallel propagation	Set to 1 or not specified. Single track.	Set to > 1	Set to number specified by `CacheAWTParallelism`.
Set to 1, which disables parallel propagation.	N/A	Set to > 1	Error is thrown at database creation, since parallelism is turned off, but `CacheAWTParallelism` is set to a value, expecting parallel propagation to be enabled.

Foreign keys in Oracle Database tables that are to be cached must have indexes created on the foreign keys. Consider these Oracle Database tables:

```
CREATE TABLE parent (c1 NUMBER PRIMARY KEY NOT NULL);
CREATE TABLE child (c1 NUMBER PRIMARY KEY NOT NULL,
                    c2 NUMBER REFERENCES parent(c1));
CREATE TABLE grchild (c1 NUMBER PRIMARY KEY NOT NULL,
                      c2 NUMBER REFERENCES parent(c1),
                      c3 NUMBER REFERENCES parent(c1));
```

These indexes must be created:

```
CREATE INDEX idx_1 ON child(c2);
CREATE INDEX idx_2 ON grchild(c2);
CREATE INDEX idx_3 ON grchild(c3);
```

The following sections describe restrictions, configuration and checks for parallel propagation:

- Table Constraint Restrictions When Using Parallel Propagation for AWT Cache Groups
- Manually Initiate Check for Missing Constraints for an AWT Cache Group
- Configuring Batch Size for Parallel Propagation for AWT Cache Groups

Table Constraint Restrictions When Using Parallel Propagation for AWT Cache Groups

When you use parallel propagation for AWT cache groups, you must manually enforce data consistency.

Any unique index, unique constraint, or foreign key constraint that exists on columns in the Oracle Database tables that are to be cached should also be created on the AWT cache tables within TimesTen. If you cannot create these constraints on the AWT cache tables and you have configured for parallel propagation, then TimesTen serializes any transactions with DML operations to any table with missing constraints. For example, if a unique index created on a table in the Oracle database cannot be created on the corresponding cached table in TimesTen, all transactions for this table are serialized.

TimesTen automatically checks for missing constraints on the Oracle database that are not cached on TimesTen when you issue any of the following SQL statements:

- When you create an AWT cache group with the `CREATE ASYNCHRONOUS CACHE GROUP` statement
- When you create a unique index on an AWT cache table with the `CREATE UNIQUE INDEX` statement
- When you drop a unique index on an AWT cache table with the `DROP INDEX` statement

Note

You can manually initiate a check for missing constraints with the `ttCacheCheck` built-in procedure. For example, TimesTen does not automatically check for missing constraints after a schema change on cached Oracle Database tables. After any schema change on the Oracle database, you should perform an manual check for missing constraints by running `ttCacheCheck` on the TimesTen database.

See Manually Initiate Check for Missing Constraints for an AWT Cache Group for other conditions where you should manually check for missing constraints.

If the check notes missing constraints on the cached tables, TimesTen issues warnings about each missing constraint.

For the following scenarios, the cached table is marked so that transactions that include DML operations are serialized when propagated to the Oracle database.

- Transactions that apply DML operations to AWT cache tables that are missing unique indexes or unique constraints.
- Missing foreign key constraints for tables within a single AWT cache group.
  - If both the referencing table and the referenced table for the foreign key relationship are in the same AWT cache group and the foreign key relationship is not defined, both tables are marked for transaction serialization.
  - If the referencing table is in an AWT cache group and the referenced table is not in an AWT cache group, the table inside the cache group is not marked for transaction serialization. Only a warning is issued to notify the user of the missing constraint.
  - If the referenced table is in an AWT cache group and the referencing table is not in an AWT cache group, the table inside the cache group is not marked for transaction serialization. Only a warning is issued to notify the user of the missing constraint.
- Missing foreign key constraints between cache groups. When you have tables defined in separate AWT cache groups that are missing a foreign key constraint, both tables are marked for serialized transactions.

- If a missing foreign key constraint causes a chain of foreign key constraints to be broken between two AWT cache groups, transactions for all tables within both AWT cache groups are serialized.

> **Note**
>
> An Oracle Database trigger may introduce an operational dependency of which TimesTen may not be aware. In this case, you should either disable parallel propagation for the AWT cache group or do not cache the table in an AWT cache group on which the trigger is created.

The following is an example of missing constraints when creating an AWT cache group. This example creates two tables in the `sales` schema in the Oracle database. There is a foreign key relationship between `active_customer` and the `ordertab` tables. Because the examples use these tables for parallel propagation, an index is created on the foreign key in the `ordertab` table.

```
SQL> CREATE TABLE active_customer
       (custid NUMBER(6) NOT NULL PRIMARY KEY,
        name VARCHAR2(50),
        addr VARCHAR2(100),
        zip VARCHAR2(12),
        region VARCHAR2(12) DEFAULT 'Unknown');
Table created.

SQL> CREATE TABLE ordertab
       (orderid NUMBER(10) NOT NULL PRIMARY KEY,
        custid NUMBER(6) NOT NULL);
Table created.

SQL> ALTER TABLE ordertab
      ADD CONSTRAINT cust_fk
       FOREIGN KEY (custid) REFERENCES active_customer(custid);
Table altered.

SQL> CREATE INDEX order_idx on ordertab (custid);
```

TimesTen automatically checks for missing constraints when each `CREATE CACHE GROUP` is issued. In the following example, a single cache group is created that includes the `active_customer` table. Only a warning is issued since the `active_customer` is the referenced table and the referencing table, `ordertab`, is not in any AWT cache group. The `active_customer` table is not marked for serialized transactions.

```
CREATE WRITETHROUGH CACHE GROUP update_cust
 FROM sales.active_customer
 (custid NUMBER(6) NOT NULL PRIMARY KEY,
 name VARCHAR2(50),
 addr VARCHAR2(100),
 zip VARCHAR2(12));
Warning  5297: The following Oracle foreign key constraints on AWT cache table
SALES.ACTIVE_CUSTOMER contain cached columns that do not have corresponding
foreign key constraints on TimesTen: SALES.CUST_FK [Outside of CG].
```

The following example creates two AWT cache groups on TimesTen, one that includes the `active_customer` table and the other includes the `ordertab` table. There is a missing foreign key constraint between the cache groups. Thus, a warning is issued for both tables, but only the ordertab table is marked for serial transactions since it is the referencing table that should contain the foreign key.

```
CREATE WRITETHROUGH CACHE GROUP update_cust
 FROM sales.active_customer
 (custid NUMBER(6) NOT NULL PRIMARY KEY,
 name VARCHAR2(50),
 addr VARCHAR2(100),
 zip VARCHAR2(12);
Warning  5297: The following Oracle foreign key constraints on AWT cache table
sales.update_customer contain cached columns that do not have corresponding
foreign key constraints on TimesTen: ordertab.cust_fk [Outside of CG].

CREATE WRITETHROUGH CACHE GROUP update_orders
 FROM sales.ordertab
 (orderid NUMBER(10) NOT NULL PRIMARY KEY,
  custid NUMBER(6) NOT NULL);
Warning  5295: Propagation will be serialized on AWT cache table
SALES.ORDERTAB because the following Oracle foreign key constraints on this
table contain cached columns that do not have corresponding foreign key
constraints on TimesTen: ORDERTAB.CUST_FK [Across AWT cache groups].
```

Manually Initiate Check for Missing Constraints for an AWT Cache Group

The ttCacheCheck built-in procedure performs the same check for missing constraints for cached tables on the Oracle database as performed automatically by TimesTen.

The ttCacheCheck provides appropriate messages about missing constraints and the tables marked for serialized propagation. With the ttCacheCheck built-in procedure, you can check for missing constraints for a given cache group or for all cache groups in TimesTen to ensure that all cache groups are not missing constraints.

Note

Since ttCacheCheck updates system tables to indicate if DML performed against a table should or should not be serialized, you must commit or roll back after the ttCacheCheck built-in completes.

See ttCacheCheck in the *Oracle TimesTen In-Memory Database Reference*.

You may need to manually call the ttCacheCheck built-in procedure to update the known dependencies after any of the following scenarios:

- After dropping a series of AWT cache groups on TimesTen with the DROP CACHE GROUP statement.
- After adding or dropping a unique index, unique constraint, or foreign key on an Oracle Database table that is cached in an AWT cache group. If you do not call the ttCacheCheck built-in procedure after adding a constraint, you may receive a run time error on the AWT cache group. After dropping a constraint, TimesTen may serialize transactions even if it is not necessary. Calling the ttCacheCheck built-in procedure verifies whether serialization is necessary.
- You can use this built-in procedure to determine why some transactions are being serialized.

Note

The `ttCacheCheck` built-in procedure cannot be called while the replication agent is running.

If a DDL statement is being performed on an AWT cache group when `ttCacheCheck` is called, then `ttCacheCheck` waits for the statement to complete or until the timeout period is reached.

If you have not defined the `CacheAwtParallelism` data store attribute to greater than one or the specified cache group is not an AWT cache group, then the `ttCacheCheck` built-in procedure returns an empty result set.

The following example shows the user manually running the `ttCacheCheck` built-in procedure to determine if there are any missing constraints for an AWT cache group `update_orders` that is owned by `cacheadmin`. A result set is returned that includes the error message. The `ordertab` table in the `update_orders` cache group is marked for serially propagated transactions.

```
Command> call ttCacheCheck(NULL, 'cacheadmin', 'update_orders');

< CACHEADMIN, UPDATE_ORDERS, CACHEADMIN, ORDERTAB, Foreign Key, CACHEADMIN,
CUST_FK, 1, Transactions updating this table will be serialized to Oracle
because: The missing foreign key connects two AWT cache groups.,
table CACHEADMIN.ORDERTAB constraint CACHEADMIN.CUST_FK foreign key(CUSTID)
references CACHEADMIN.ACTIVE_CUSTOMER(CUSTID) >
1 row found.
```

Whenever the cache group schema changes in either the TimesTen or Oracle databases, you can call `ttCacheCheck` against all AWT cache groups to verify all constraints. The following example shows the user manually running the `ttCacheCheck` built-in procedure to determine if there are any missing constraints for any AWT cache group in the entire TimesTen database by providing a `NULL` value for all input parameters. A result set is returned that includes any error messages.

```
Command> call ttCacheCheck(NULL, NULL, NULL);

< CACHEADMIN, UPDATE_ORDERS, CACHEADMIN, ORDERTAB, Foreign Key, CACHEADMIN,
CUST_FK, 1, Transactions updating this table will be serialized to Oracle
because: The missing foreign key connects two AWT cache groups.,
table CACHEADMIN.ORDERTAB constraint CACHEADMIN.CUST_FK foreign key(CUSTID)
references CACHEADMIN.ACTIVE_CUSTOMER(CUSTID) >
1 row found.
```

Configuring Batch Size for Parallel Propagation for AWT Cache Groups

When using AWT cache groups, TimesTen batches together one or more transactions that are to be applied in parallel to the back-end Oracle database. The `CacheParAwtBatchSize` parameter configures a threshold value for the number of rows included in a single batch. Once the maximum number of rows is reached, TimesTen includes the rest of the rows in the transaction (TimesTen does not break up any transactions), but does not add any more transactions to the batch.
For example, a user sets the `CacheParAwtBatchSize` to 200. For the next AWT propagation, there are three transactions, each with 120 rows, that need to be propagated and applied to the Oracle database. TimesTen includes the first two transactions in the first batch for a total of 240 rows. The third transaction is included in a second batch.

The default value for the `CacheParAwtBatchSize` parameter is 125 rows. The minimum value is 1. See ttDBConfig in the *Oracle TimesTen In-Memory Database Reference*.

You can retrieve the current value of `CacheParAwtBatchSize` as follows:

```
call ttDBConfig('CacheParAwtBatchSize');
< CACHEPARAWTBATCHSIZE, 125 >
1 row found.
```

You can set the `CacheParAwtBatchSize` parameter to 200 as follows:

```
call ttDBConfig('CacheParAwtBatchSize','200');
< CACHEPARAWTBATCHSIZE, 200 >
1 row found
```

Set the `CacheParAwtBatchSize` parameter only when advised by Oracle Support, who analyzes the workload and any dependencies in the workload to determine if a different value for `CacheParAwtBatchSize` could improve performance. Dependencies exist when transactions concurrently change the same data. Oracle Support may advise you to reduce this value if there are too many dependencies in the workload.

Improving AWT Throughput with SQL Array Processing

The `CacheAWTMethod` connection attribute setting determines whether to use the PL/SQL processing method or SQL array processing method for asynchronous writethrough propagation when applying changes to the Oracle database.

- PL/SQL processing method: AWT bundles all pending operations into a single PL/SQL collection that is sent to the Oracle database server to be performed. This processing method is appropriate when there are mixed transactions and network latency between TimesTen and the Oracle database server. It is efficient for most use cases when the workload consists of mixed `INSERT`, `UPDATE`, and `DELETE` statements to the same or different tables. By default, TimesTen uses the PL/SQL processing method (`CacheAWTMethod=1`).
- SQL array processing method: Consider changing `CacheAWTMethod` to 0 when the changes consist of mostly repeated sequences of the same operation (`INSERT`, `UPDATE`, or `DELETE`) against the same table. For example, SQL array processing is very efficient when a user does an update that affects several rows of a table. Updates are grouped together and sent to the Oracle database in a single batch.

The PL/SQL processing method transparently falls back to SQL array processing mode temporarily when it encounters one of the following:

- A statement that is over 32761 bytes in length.
- A statement that references a column of type `BINARY FLOAT`, `BINARY DOUBLE` and `VARCHAR/VARBINARY` of length greater than 4000 bytes.

Note

You can also set this value with the `ttDBConfig` built-in procedure with the `CacheAwtMethod` parameter. See ttDBConfig in the *Oracle TimesTen In-Memory Database Reference*.

See CacheAWTMethod in *Oracle TimesTen In-Memory Database Reference*.

Improving Performance for Autorefresh Operations

These best practice recommendations are intended to improve performance for autorefresh operations specifically in trigger-based cache groups.

- Minimizing Delay for Cached Data with Continuous Autorefresh
- Reducing Contention for Dynamic Read-Only Cache Groups with Incremental Autorefresh
- Reducing Lock Contention for Read-Only Cache Groups with Autorefresh and Dynamic Load
- Options for Reducing Contention Between Autorefresh and Dynamic Load Operations
- Improving Performance When Reclaiming Memory During Autorefresh Operations
- Running Large Transactions with Incremental Autorefresh Read-Only Cache Groups
- Configuring a Select Limit for Incremental Autorefresh for Read-Only Cache Groups

Minimizing Delay for Cached Data with Continuous Autorefresh

You can specify continuous autorefresh with an autorefresh interval of 0 milliseconds. With continuous autorefresh, the next autorefresh cycle is scheduled as soon as possible after the last autorefresh cycle has ended.

Continuous autorefresh could result in a higher resource usage when there is a low workload rate on the Oracle database, since the cache agent could be performing unnecessary round-trips to the Oracle database.

See CREATE CACHE GROUP and ALTER CACHE GROUP in the *Oracle TimesTen In-Memory Database SQL Reference*.

Reducing Contention for Dynamic Read-Only Cache Groups with Incremental Autorefresh

Most autorefresh and dynamic load operations coordinate their access to the Oracle database for correctness. For trigger-based cache groups only, the default TimesTen coordination behavior may, in some cases, lead to contention between autorefresh and dynamic load operations

If you have dynamic read-only cache groups with incremental autorefresh, then:

- Multiple dynamic load operations could be blocked by autorefresh operations.
- Autorefresh operations are frequently delayed while waiting for dynamic load operations to complete.

Enabling the `DynamicLoadReduceContention` database system parameter is useful for dynamic cache groups by changing the way that autorefresh and dynamic load operations coordinate, which results in reduced contention between autorefresh and dynamic load operations.

- Dynamic load operations are never blocked by autorefresh operations (due to additional synchronization).
- Autorefresh operations are not completely delayed by dynamic load operations. Instead, autorefresh operations will wait a short while for concurrently executing dynamic load operations to be notified that a new autorefresh operation is starting. This enables dynamic

load operations to synchronize in tandem with concurrently executing autorefresh operations.

> **Note**
>
> You cannot change the value of the `DynamicLoadReduceContention` database system parameter if there are any dynamic read-only cache groups or if the cache or replication agents are running. In order to change the value of this parameter, you must unload and drop (and later recreate) any existing dynamic read only cache groups, then stop the cache and replication agents.

The following example sets `DynamicLoadReduceContention=1`:

```
call ttDbConfig('DynamicLoadReduceContention','1');
```

You can query the current value of the `DynamicLoadReduceContention` parameter.

```
call ttDbConfig('DynamicLoadReduceContention');
```

> **Note**
>
> See ttDBConfig in the *Oracle TimesTen In-Memory Database Reference*.

Requirements for Setting DynamicLoadReduceContention

There are requirements when using the `DynamicLoadReduceContention` database system parameter.

The `DynamicLoadReduceContention` database system parameter requires the following to be enabled:

- Required Oracle Database privileges: You must grant two additional Oracle Database privileges to the cache administration user:
  - `EXECUTE ON SYS.DBMS_FLASHBACK`
  - `SELECT ANY TRANSACTION`

  These are granted to the cache administration user when you execute the `grantCacheAdminPrivileges.sql` and `initCacheAdminSchema.sql` scripts.
- Support for Oracle Database: This feature requires the use of the Oracle Database Flashback Transaction Queries.With Oracle Database 12.2.0.1 with Multitenant option, Flashback Transaction Queries only supports Local Undo. You cannot use this feature with Oracle Database 12.2.0.1 Multitenant option with Shared Undo.
- Required settings for active standby pair replication scheme:
  - Both active and standby masters must be installed. If you are replicating between active and standby masters where each is installed with different TimesTen versions, then this parameter cannot be enabled if one of the TimesTen versions does not support this feature.
  - The `DynamicLoadReduceContention` database system parameter must be set to the same value on both the active and standby masters.

Otherwise, an error is written to the daemon log. Replication will not progress until the settings and TimesTen versions conform on both the active and standby masters.

Reducing Lock Contention for Read-Only Cache Groups with Autorefresh and Dynamic Load

Your application can time out because of a lock contention between autorefresh and dynamic load requests.

An autorefresh operation automatically loads committed changes on cached Oracle Database tables into the cache tables in TimesTen. A dynamic load operation requests data from the Oracle database (originating from a `SELECT` statement) and inserts the rows into the cache group. Both the autorefresh and dynamic load operations require access to the cache metadata, which could cause a lock contention.

At the end of an autorefresh operation, TimesTen updates the metadata to track the autorefresh progress. If you have requested guaranteed durability by setting the `DurableCommits` connection attribute to 1, then the autorefresh updates to the metadata are always durably committed. If you have requested delayed durability by setting the `DurableCommits` connection attribute to 0 (the default), then TimesTen must ensure that the autorefresh updates to the metadata are durably committed before the garbage collector can clean up the autorefresh tracking tables stored in the Oracle database.

When a durable commit is initiated for the metadata, any previous non-durable committed transactions in the transaction log buffer that have not been flushed to the file system are also a part of the durable commit. On hosts with busy or slow file systems, the durable commit could be slow enough to lock out dynamic load requests for an undesirable amount of time.

If you notice that your application is timing out because of a lock contention between autorefresh and dynamic load requests, you can set the `CacheCommitDurable` cache configuration parameter to 0 with the `ttCacheConfig` built-in procedure. This reduces the occurrence of lock contention between autorefresh and dynamic load requests in the same application by:

- Running a non-durable commit of the autorefresh changes made to the metadata.
- Using a separate thread in the cache agent to durably commit the autorefresh changes before the garbage collector cleans up the autorefresh tracking tables stored in the Oracle database. This results in a slight performance cost as garbage collection is delayed until after the durable commit completes.

The lock is removed after the non-durable commit of the autorefresh changes to the metadata. After which, there is no longer a lock held on the metadata and any dynamic load requests for the recently refreshed tables can continue processing without waiting. However, if there is an error and database recovery starts, autorefresh may need to reapply any committed transactions that did not flush to disk before a failure.

The following example sets `CacheCommitDurable=0`:

```
call ttCacheConfig('CacheCommitDurable',,,'0');
```

You can query the current value of the `CacheCommitDurable` parameter.

```
call ttCacheConfig('CacheCommitDurable');
```

See ttCacheConfig in the *Oracle TimesTen In-Memory Database Reference*.

Options for Reducing Contention Between Autorefresh and Dynamic Load Operations

There are two methods to reduce contention between autorefresh and dynamic load operations.

You can enable each or both if:

- If you see error messages indicating lock contention between autorefresh and dynamic load operations, then enable the `DynamicLoadReduceContention` database system parameter by setting the value to 1 with the `ttDbConfig` built-in procedure. See Reducing Contention for Dynamic Read-Only Cache Groups with Incremental Autorefresh.
- If you notice that commit operations for autorefresh are taking an unusually long time, then look for a `TT47087` informational message in the support log. Locate the `tt1stXactCommitTime` and `tt2ndXactCommitTime` entries within this message. If the time indicated for either of both of these entries unusually high or is a major portion of the time indicated in the Duration entry, this may indicate that the durable commit of transaction logs is slow. In this case, you have the option to set the `CacheCommitDurable` cache configuration parameter to 0 with the `ttCacheConfig` built-in procedure. For more details on the `CacheCommitDurable` cache configuration parameter, see Reducing Lock Contention for Read-Only Cache Groups with Autorefresh and Dynamic Load.

Enable both options if there is a small autorefresh interval in conjunction with a high number of dynamic load requests.

Improving Performance When Reclaiming Memory During Autorefresh Operations

As described Transaction Reclaim Operations in the *Oracle TimesTen In-Memory Database Operations Guide*, TimesTen resource cleanup occurs during the reclaim phase of a transaction commit.

To improve performance, a number of transaction log records are cached in memory to reduce the need to access the transaction log file in the commit buffer. However, TimesTen must access the transaction log if the transaction is larger than the reclaim buffer.

When you are using autorefresh for your cache groups, the cache agent has its own reclaim buffer to manage the transactions that are committed within autorefresh operations. If the cache agent reclaim buffer is too small, the commit operations during autorefresh can take longer than expected as it must access the transaction log file. To avoid any performance issues, you can configure a larger reclaim buffer for the cache agent so that the cache agent can handle larger transactions in memory at reclaim time.

When using an active standby pair replication scheme to replicate autorefresh operations, the replication agent applies the same autorefresh operations as part of the replication. Thus, the replication agents on both the active and standby nodes have their own reclaim buffers that should be configured to be the same size or greater than the cache agent reclaim buffer.

The `ttDbConfig` built-in procedure provides the following parameters for setting the maximum size for the reclaim buffers for both the cache agent and the replication agent. (The memory for the reclaim buffers are allocated out of temporary memory.)

- `CacheAgentCommitBufSize` sets the maximum size for the reclaim buffer for the cache agent.

- `RepAgentCommitBufSize` sets the maximum size for the reclaim buffer for the replication agent. You should configure the maximum size for the reclaim buffer on both the active and standby nodes. It is recommended that you set the size for the reclaim buffers to the same value on both nodes, but not required.

> **Note**
>
> For more details, see ttDBConfig in the *Oracle TimesTen In-Memory Database Reference*.

To determine if you should increment the size for the cache agent reclaim buffer, evaluate the `CommitBufMaxReached` and `CommitBufNumOverflows` statistics provided by the `ttCacheAutorefIntervalStatsGet` built-in procedure. See Retrieving Statistics on Autorefresh Transactions.

Running Large Transactions with Incremental Autorefresh Read-Only Cache Groups

At certain times, you may run large transactions such as for the end of the month, the end of a quarter, or the end of the year transactions. You may also have situations where you modify or add a large amount of data in the Oracle database over a short period of time.

For trigger-based read-only cache groups with incremental autorefresh, TimesTen could run out of permanent space when an autorefresh operation applies either of these cases. Therefore, for these situations, you can configure an autorefresh transaction limit, where the large amount of data is broken up, applied, and committed over several smaller transactions.

> **Note**
>
> The autorefresh transaction limit can only be set for static read-only cache groups.

The `ttCacheAutorefreshXactLimit` built-in procedure enables you to direct autorefresh to commit after running a specific number of operations. This option applies to all incremental autorefresh read-only cache groups that are configured with the same autorefresh interval.

Since the single transaction is broken up into several smaller transactions, transactional consistency cannot be maintained while autorefresh is in progress. Once the autorefresh cycle completes, the data is transactionally consistent. To protect instance consistency, we recommend that you set the autorefresh transaction limit only on cache groups with only a single table, since instance consistency between the parent and child tables is not guaranteed. When the autorefresh transaction limit is turned on, TimesTen does not enforce the foreign key relationship that protects instance consistency. Once you turn off the autorefresh transaction limit for incremental autorefresh read-only cache groups, both instance and transactional consistency are maintained again.

Note

If you are using an active standby pair, you must call the `ttCacheAutorefreshXactLimit` built-in procedure for the same values on both the active and standby masters.

The following sections describe how to configure an autorefresh transaction limit.

- Using ttCacheAutorefreshXactLimit
- Example of Potential Transactional Inconsistency
- Retrieving Statistics to Evaluate Performance When a Transaction Limit is Set

Using ttCacheAutorefreshXactLimit

Note

See ttCacheAutorefreshXactLimit in the *Oracle TimesTen In-Memory Database Reference*.

It is only for the trigger-based cache groups. For the month end processing, there can be a large number updates in a single transaction for the Oracle tables that are cached in cache groups with autorefresh. In order to ensure that the large transaction does not fill up permanent memory, you can enable autorefresh to commit after every 256 (or any other user specified number) operations with the `ttCacheAutorefreshXactLimit` built-in procedure.

Turn on an autorefresh transaction limit for incremental autorefresh read-only cache groups before a large transaction with the `ttCacheAutorefreshXactLimit` built-in procedure where the *`value`* is set to `ON` or to a specific number of operations. Then, when autorefresh finishes updating the cached tables in TimesTen, turn off the autorefresh transaction limit for incremental autorefresh read-only cache groups with the `ttCacheAutorefreshXactLimit` built-in procedure.

The following example sets up the transaction limit to commit after every 256 operations for all incremental autorefresh read-only cache groups that are defined with an interval value of 10 seconds.

```
call ttCacheAutorefreshXactLimit('10000', 'ON');
```

After the month end process has completed and the incremental autorefresh read-only cache groups are refreshed, disable the transaction limit for incremental autorefresh read-only cache groups that are defined with the interval value of 10 seconds.

```
call ttCacheAutorefreshXactLimit('10000', 'OFF');
```

To enable the transaction limit for incremental autorefresh read-only cache groups to commit after every 1024 operations, provide 1024 as the value as follows:

```
call ttCacheAutorefreshXactLimit('10000', '1024');
```

Example of Potential Transactional Inconsistency

This example shows how to create two incremental autorefresh read-only cache groups.

The following example uses the employee and departments table, where the department id of the department table is a foreign key that points to the department id of the employee table.

The following example creates two incremental autorefresh read-only cache groups, where each is in its own cache group. The autorefresh transaction limit is enabled with `ttCacheAutorefreshXactLimit` before a large transaction and is disabled after it completes.

1. Before you initiate the large transaction, invoke `ttCacheAutorefreshXactLimit` to set the interval value and the number of operations after which to automatically commit. The following sets the number of operations to three (which is intentionally low to show a brief example) for all incremental autorefresh read-only cache groups with a two second interval.

   ```
   CALL ttCacheAutorefreshXactLimit('2000', '3');
   < 2000, 3 >
   1 row found.
   ```

2. Create the incremental autorefresh read-only cache groups with interval of two seconds. This example creates two static (non-dynamic) read-only cache groups, where each contains a single table.

   ```
   CREATE READONLY CACHE GROUP cgDepts AUTOREFRESH MODE INCREMENTAL
    INTERVAL 2 SECONDS
   FROM departments
       ( department_id    NUMBER(4) PRIMARY KEY
       , department_name  VARCHAR2(30) NOT NULL
       , manager_id       NUMBER(6)
       , location_id      NUMBER(4)
       );

   CREATE READONLY CACHE GROUP cgEmpls AUTOREFRESH MODE INCREMENTAL
    INTERVAL 2 SECONDS
   FROM employees
       ( employee_id    NUMBER(6) PRIMARY KEY
       , first_name     VARCHAR2(20)
       , last_name      VARCHAR2(25) NOT NULL
       , email          VARCHAR2(25) NOT NULL UNIQUE
       , phone_number   VARCHAR2(20)
       , hire_date      DATE NOT NULL
       , job_id         VARCHAR2(10) NOT NULL
       , salary         NUMBER(8,2)
       , commission_pct NUMBER(2,2)
       , manager_id     NUMBER(6)
       , department_id  NUMBER(4)
       );
   ```

3. Run a `LOAD CACHE GROUP` statement for both cache groups with autorefresh.

   ```
   LOAD CACHE GROUP cgDepts COMMIT EVERY 256 ROWS;
   27 cache instances affected.

   LOAD CACHE GROUP cgEmpls COMMIT EVERY 256 ROWS;
   107 cache instances affected.
   ```

You can have inconsistency within the table during an autorefresh as shown with the employees table.

1. On TimesTen, select the minimum and maximum salary of all employees.

```
SELECT MIN(salary), MAX(salary) FROM employees;
< 2100, 24000 >
1 row found.
```

2. On the Oracle database, add 100,000 to everyone's salary.

   ```
   UPDATE employees SET salary = salary + 100000;
   107 rows updated.
   ```

3. On TimesTen, when you run the `SELECT` again (while the autorefresh transactions are commmitted after every 3 records), it shows that while the maximum salary has updated, the minimum salary is still the old value.

   ```
   SELECT MIN(salary), MAX(salary) FROM employees;
   < 2100, 124000 >
   1 row found.
   ```

4. However, once the autorefresh completes, transactional consistency is maintained. For this example, once the autorefresh process completes, all salaries have increased by 100,000.

   ```
   SELECT MIN(salary), MAX(salary) FROM employees;
   < 102100, 124000 >
   1 row found.
   ```

5. The large transaction is complete, so disable the transaction limit for cache groups with a 2 second interval autorefresh.

   ```
   call ttCacheAutorefreshXactLimit('2000', 'OFF');
   ```

You can have transactional inconsistency between cache groups if you run a SQL statement while the autorefresh process is progressing. The following `SELECT` statement example runs against the employees and department table in the `cgDepts` autorefresh cache group. With this example, since the foreign key is not enforced on TimesTen and the autorefresh process applies several transactions, the employee table updates may be inserted before the department updates.

In addition, all of the updates for both tables in the cache group are not applied until the autorefresh cycle has completed. In the following example, the `SELECT` statement is performed before the autorefresh process is complete. Thus, the results do not show all of the expected data, such as the department name and several employees (some of the lawyers in the legal department 1000) are missing.

```
SELECT e.department_id, d.DEPARTMENT_NAME, e.FIRST_NAME, e.LAST_NAME
       FROM employees e, departments d
       WHERE e.DEPARTMENT_ID  = d.DEPARTMENT_ID (+)
       AND e.department_id >= 1000 ORDER BY 1,2,3,4;
< 1000, Legal, Alec, Dunkle >
< 1000, Legal, Barry, Strong >
< 1000, Legal, Leigh, Harrison >
3 rows found.
```

However, after the autorefresh process completes, transactional consistency is maintained. The following shows the same `SELECT` statement performed after the autorefresh is complete. All expected data, the department information and all of the new lawyers, are updated.

```
SELECT e.department_id, d.DEPARTMENT_NAME, e.FIRST_NAME, e.LAST_NAME
       FROM employees e, departments d
       WHERE e.DEPARTMENT_ID  = d.DEPARTMENT_ID (+)
       AND e.department_id >= 1000 ORDER BY 1,2,3,4;
< 1000, Legal, Alec, Dunkle >
< 1000, Legal, Barry, Strong >
< 1000, Legal, Leigh, Harrison >
< 1000, Legal, John, Crust >
< 1000, Legal, Robert, Wright >
```

```
< 1000, Legal, Robert, Smith >
6 rows found.
```

For cache groups with autorefresh that have more than one table, you can also experience transactional inconsistency if you run SQL statements while the autorefresh process is in progress.

1. Initiate the transaction limit for incremental cache groups with autorefresh of 2 seconds with the `ttCacheAutorefreshXactLimit` built-in procedure and create a single autorefresh cache group with two tables: the employees and departments tables.

   ```
   CALL ttCacheAutorefreshXactLimit('2000', '3');
   < 2000, 3 >
   1 row found.

   CREATE READONLY CACHE GROUP cgDeptEmpls AUTOREFRESH MODE INCREMENTAL
    INTERVAL 2 SECONDS
   FROM departments
        ( department_id    NUMBER(4) PRIMARY KEY
        , department_name  VARCHAR2(30) NOT NULL
        , manager_id       NUMBER(6)
        , location_id      NUMBER(4)
        )
      , employees
        ( employee_id    NUMBER(6) PRIMARY KEY
        , first_name     VARCHAR2(20)
        , last_name      VARCHAR2(25) NOT NULL
        , email          VARCHAR2(25) NOT NULL UNIQUE
        , phone_number   VARCHAR2(20)
        , hire_date      DATE NOT NULL
        , job_id         VARCHAR2(10) NOT NULL
        , salary         NUMBER(8,2)
        , commission_pct NUMBER(2,2)
        , manager_id     NUMBER(6)
        , department_id  NUMBER(4)
        , foreign key(department_id) references departments(department_id)
        );
   ```

2. Manually load the cache group.

   ```
   LOAD CACHE GROUP cgDeptEmpls COMMIT EVERY 256 ROWS;
   27 cache instances affected.
   ```

3. Run a `SELECT` statement on TimesTen that uploads all of the legal department data.

   ```
   SELECT e.department_id, d.department_name, count(*)
          FROM employees e, departments d
          WHERE e.department_id  = d.department_id (+)
          GROUP BY e.department_id, d.department_name
          ORDER BY 1 desc;
   < 110, Accounting, 2 >
   < 100, Finance, 6 >
   < 90, Executive, 3 >
   < 80, Sales, 34 >
   < 70, Public Relations, 1 >
   < 60, IT, 5 >
   < 50, Shipping, 45 >
   < 40, Human Resources, 1 >
   < 30, Purchasing, 6 >
   < 20, Marketing, 2 >
   < 10, Administration, 1 >
   11 rows found.
   ```

4. On Oracle, insert a new legal department, numbered 1000, with 6 new lawyers in both the employee and department tables.
5. When performing a SELECT statement on TimesTen during the autorefresh process, only data on two of the lawyers in department 1000 have been uploaded into TimesTen.

```
SELECT e.department_id, d.department_name, count(*)
       FROM employees e, departments d
       WHERE e.department_id  = d.department_id (+)
       GROUP BY e.department_id, d.department_name
       ORDER BY 1 desc;
< 1000, Legal, 2 >
< 110, Accounting, 2 >
< 100, Finance, 6 >
< 90, Executive, 3 >
< 80, Sales, 34 >
< 70, Public Relations, 1 >
< 60, IT, 5 >
< 50, Shipping, 45 >
< 40, Human Resources, 1 >
< 30, Purchasing, 6 >
< 20, Marketing, 2 >
< 10, Administration, 1 >
12 rows found.
```

6. However, after the autorefresh process completes, all 6 employees (lawyers) in the legal department have been uploaded to TimesTen. Now, it is transactionally consistent.

```
SELECT e.department_id, d.department_name, COUNT(*)
       FROM employees e, departments d
       WHERE e.department_id  = d.department_id (+)
       GROUP BY e.department_id, d.department_name
       ORDER BY 1 desc;
< 1000, Legal, 6 >
< 110, Accounting, 2 >
< 100, Finance, 6 >
< 90, Executive, 3 >
< 80, Sales, 34 >
< 70, Public Relations, 1 >
< 60, IT, 5 >
< 50, Shipping, 45 >
< 40, Human Resources, 1 >
< 30, Purchasing, 6 >
< 20, Marketing, 2 >
< 10, Administration, 1 >
12 rows found.
```

7. The large transaction is complete, so disable the transaction limit for cache groups with a 2 second autorefresh interval.

```
call ttCacheAutorefreshXactLimit('2000', 'OFF');
```

Retrieving Statistics to Evaluate Performance When a Transaction Limit is Set

To see how a autorefresh transaction limit for a particular autorefresh interval is performing, you can retrieve statistics for the last 10 incremental autorefresh transactions for this autorefresh interval with the ttCacheAutorefIntervalStatsGet built-in procedure.

See Retrieving Statistics on Autorefresh Transactions.

Configuring a Select Limit for Incremental Autorefresh for Read-Only Cache Groups

To facilitate incremental autorefresh for trigger-based read-only cache groups, TimesTen runs a table join query on both the Oracle database base table and its corresponding change log table to retrieve the incremental changes. However, if both tables are very large, the join query can be slow. In addition, if the Oracle database base table is continuously updated while the join-query is processing, you may receive the `ORA-01555` “Snapshot too old" error from a long-running autorefresh query.

To avoid this situation, you can configure incremental autorefresh with a select limit for static read-only cache groups, which joins the Oracle database base table with a limited number of rows from the autorefresh change log table. You can configure a select limit with the `ttCacheAutorefreshSelectLimit` built-in procedure.

Note

The select limit can only be set for static read-only cache groups. To protect instance consistency, we recommend that you set the select limit only on cache groups with only a single table.

Autorefresh continues to apply changes to the cached table incrementally until all the rows in the autorefresh change log table have been applied. When there are no rows left to apply, the autorefresh thread sleeps for the rest of the interval period.

Note

See ttCacheAutorefreshSelectLimit in the *Oracle TimesTen In-Memory Database Reference*.

For example, before a large transaction, you can call the `ttCacheAutorefreshSelectLimit` built-in procedure to set a select limit to 1000 rows for cache groups with incremental autorefresh where the interval value is 10 seconds. The following example sets the *`value`* to `ON`.

```
Command> call ttCacheAutorefreshSelectLimit('10000', 'ON');
< 10000, ON >
1 row found.
```

The following example set a select limit to 2000 rows for cache groups with incremental autorefresh where the interval value is 7 seconds.

```
Command> call ttCacheAutorefreshSelectLimit('7000', '2000');
< 7000, 2000 >
1 row found.
```

You can disable any select limit for cache groups with incremental autorefresh where the interval value is 10 seconds by setting the *`value`* to `OFF`.

```
Command> call ttCacheAutorefreshSelectLimit('10000', 'OFF');
< 10000, OFF >
1 row found.
```

The following sections describe details when configuring a select limit for static read-only cache groups with incremental autorefresh.

- See How to Determine Which Intervals Have a Particular Select Limit to determine which intervals have a select limit.
- See Retrieving Statistics on Autorefresh Transactions to retrieve statistics for incremental autorefresh transactions for this autorefresh interval. This determines how a select limit for a particular autorefresh interval is performing.

How to Determine Which Intervals Have a Particular Select Limit

To determine the interval for a cache group, use `ttIsql` and run the `cachegroups` command.

```
> cachegroups cgowner.cgname;
```

This returns all attributes for the *`cgowner.cgname`* cache group including the interval.

To determine which intervals have a select limit, you can run the following query on the Oracle database where <*`cacheAdminUser`*> is the cache administrator, <*`hostName`*> is the host name of the machine where the TimesTen database is located, <*`databaseFileName`*> is the database path taken from the `DataStore` attribute, and substitute the version number (such as 07) for the *`xx`*.

```
SELECT * FROM <cacheAdminUser>.tt_XX_arinterval_params
 WHERE param='AutorefreshSelectEveryN'
   AND host='<hostName>'
   AND database like '%<databaseFileName>%'
 ORDER BY arinterval;
```

For example, if the cache administrator user name is `pat`, the host name is `myhost`, the database file name is `myTtDb`, and 07 is substituted for *`xx`* that is the TimesTen minor release number then:

```
SELECT * FROM pat.tt_07_arinterval_params
 WHERE param='AutorefreshSelectEveryN'
   AND host='myhost'
   AND database like '%myTtDb%'
 ORDER BY arinterval;
```

The interval is stored in milliseconds.

Retrieving Statistics to Evaluate Performance When Using a Select Limit

To see how a select limit for a particular autorefresh interval is performing, you can retrieve statistics for incremental autorefresh transactions for this autorefresh interval with the `ttCacheAutorefIntervalStatsGet` built-in procedure.

See Retrieving Statistics on Autorefresh Transactions.

Retrieving Statistics on Autorefresh Transactions

Call the `ttCacheAutorefIntervalStatsGet` built-in procedure for statistical information about the last 10 autorefresh cycles for a particular autorefresh interval defined for an incremental autorefresh read-only cache group.

Note

See ttCacheAutorefIntervalStatsGet in the *Oracle TimesTen In-Memory Database Reference*.

This built-in procedure is useful if you have set an transaction limit or a select limit for incremental, autorefresh read-only cache groups. See Running Large Transactions with Incremental Autorefresh Read-Only Cache Groups and Configuring a Select Limit for Incremental Autorefresh for Read-Only Cache Groups.

The following example shows how to call the `ttCacheAutorefIntervalStatsGet` built-in procedure to retrieve statistics for incremental autorefresh read-only cache groups that have been defined as static and have the interval of 2 seconds:

```
Command> call ttCacheAutorefIntervalStatsGet(2000, 1);

< 2000, 1, 21, 2013-04-30 06:05:38.000000, 100, 3761, 3761, 822, 1048576,
1280, 0, 58825, 63825, 13590, 0, 0, 0, 0, 0 >
< 2000, 1, 20, 2013-04-30 06:05:37.000000, 100, 85, 85, 18, 1048576, 1280,
0, 55064, 60064, 12768, 0, 0, 0, 0, 0 >
< 2000, 1, 19, 2013-04-30 06:05:32.000000, 100, 3043, 3043, 666, 1048576,
1280, 0, 54979, 59979, 12750, 0, 0, 0, 0, 0 >
< 2000, 1, 18, 2013-04-30 06:05:30.000000, 100, 344, 344, 74, 1048576,
1280, 0, 51936, 56936, 12084, 0, 0, 0, 0, 0 >
< 2000, 1, 17, 2013-04-30 06:05:28.000000, 100, 1826, 1826, 382, 1048576,
1280, 0, 51592, 56592, 12010, 0, 0, 0, 0, 0 >
< 2000, 1, 16, 2013-04-30 06:05:26.000000, 100, 55, 55, 12, 1048576,
1280, 0, 49766, 54766, 11628, 0, 0, 0, 0, 0 >
< 2000, 1, 15, 2013-04-30 06:05:22.000000, 100, 2901, 2901, 634, 1048576,
1280, 0, 49711, 54711, 11616, 0, 0, 0, 0, 0 >
< 2000, 1, 14, 2013-04-30 06:05:21.000000, 100, 55, 55, 12, 1048576,
1280, 0, 46810, 51810, 10982, 0, 0, 0, 0, 0 >
< 2000, 1, 13, 2013-04-30 06:05:10.000000, 100, 5844, 5844, 1263, 1048576,
1280, 0, 46755, 51755, 10970, 0, 0, 0, 0, 0 >
< 2000, 1, 12, 2013-04-30 06:05:08.000000, 100, 607, 607, 132, 1048576,
1280, 0, 40911, 45911, 9707, 0, 0, 0, 0, 0 >

10 rows found.
```

Caching the Same Oracle Table on Two or More TimesTen Databases

For each cache administration user, TimesTen creates a change log table and trigger (as part of what is created to manage caching) in the Oracle database for each cache table in the cache group. A trigger is fired for each committed insert, update, or delete operation on the cached Oracle Database table; the action is logged in the change log table.

If you cache the same Oracle database table in a cache group on two different TimesTen databases, we recommend that you use the same cache administration user name on both TimesTen databases as the owner of the cache table on each TimesTen database.

When you use the same cache administration user, only one trigger and change log table are created to manage the changes to the base table. Thus, it is efficient and does not slow down the application.

If you create separate cache administration users on each TimesTen database to own the cache group that caches the same Oracle table, then separate triggers and change log tables exist on the Oracle database for the same table: one for each cache administration user. For example, if you have two separate TimesTen databases, each with their own cache administration user, two triggers fire for each DML operation on the base table, each of which are stored in a separate change log table. Firing two triggers and managing the separate change log tables can slow down the application.

The only reason to create separate cache administration users is if one of the TimesTen databases that caches the same table has a slow autorefresh rate or a slow connection to the Oracle database. In this case, having a single cache administration user on both TimesTen databases slows down the application on the faster connection, as it waits for the updates to be propagated to the slower database.

8

Cleaning Up the Caching Environment

There are specific tasks that need to be performed in the TimesTen and Oracle databases to drop cache groups. You should shut down all components when using AWT cache groups.

Stopping the Replication Agent

If you are using AWT cache groups that use an active standby pair replication scheme, call the `ttRepStop` built-in procedure to stop the replication agent.

This must be done on each TimesTen database of the active standby pair including any read-only subscriber databases, and any standalone TimesTen databases that contain AWT cache groups.

From the `cache1`, `cache2`, `cacheactive`, `cachestandby` and `rosubscriber` databases, call the `ttRepStop` built-in procedure as the TimesTen cache administration user to stop the replication agent on the database:

```
Command> CALL ttRepStop;
```

Dropping a Cache Group

Use the `DROP CACHE GROUP` statement to drop a cache group and its cache tables.

Oracle Database objects used to manage the caching of Oracle Database data are automatically dropped when you use the `DROP CACHE GROUP` statement to drop a cache group.

If you issue a `DROP CACHE GROUP` statement on a cache group that has an autorefresh operation in progress:

- The autorefresh operation stops if the `LockWait` connection attribute setting is greater than 0. The `DROP CACHE GROUP` statement preempts the autorefresh operation.
- The autorefresh operation continues if the `LockWait` connection attribute setting is 0. The `DROP CACHE GROUP` statement is blocked until the autorefresh operation completes or the statement fails with a lock timeout error.

If you have created an AWT cache group, a replication scheme is created to enable committed changes on its cache tables to be asynchronously propagated to the cached Oracle tables. This replication scheme is automatically dropped when you drop the AWT cache group. Thus, perform the following before dropping an AWT cache group:

1. Use the `ttRepSubscriberWait` built-in procedure to make sure that all committed changes on its cache tables have been propagated to the cached Oracle Database tables before dropping the AWT cache group.

   ```
   % ttIsql "DSN=cache1;UID=cacheadmin;PwdWallet=/wallets/cacheadminwallet"
   Command> CALL ttRepSubscriberWait('_AWTREPSCHEME','TTREP','_ORACLE','sys1',-1);
   ```

2. The cache tables in an AWT cache group are replicated in an active standby pair. If the cache tables are the only tables that are being replicated, drop the active standby pair using a `DROP ACTIVE STANDBY PAIR` statement before dropping the AWT cache groups.

   Run the following statement as the TimesTen cache administration user on the `cacheactive`, `cachestandby` and `rosubscriber` databases to drop the active standby pair replication scheme:

   ```
   Command> DROP ACTIVE STANDBY PAIR;
   Command> exit
   ```

Perform the following when dropping a cache group:

1. Run an `ALTER CACHE GROUP` statement to set the autorefresh state to `OFF` for cache groups with autorefresh.

2. Before you can drop a cache group, you must grant the `DROP ANY TABLE` privilege to the TimesTen cache administration user. Run the following statement as the instance administrator on the `cache1`, `cache2`, `cacheactive` and `cachestandby` databases to grant the `DROP ANY TABLE` privilege to the TimesTen cache administration user. The following example shows the SQL statement issued from the `cache1` database:

   ```
   % ttIsql cache1
   Command> GRANT DROP ANY TABLE TO cacheadmin;
   Command> exit
   ```

3. Use a `DROP CACHE GROUP` statement to drop the cache groups from the standalone TimesTen databases and, if using an AWT cache group, the active and standby databases.

   Run the following statement as the TimesTen cache administration user on the `cache1`, `cache2`, `cacheactive` and `cachestandby` databases to drop the `subscriber_accounts` cache group. The following example shows the SQL statement issued from the `cache1` database:

   ```
   % ttIsql "DSN=cache1;UID=cacheadmin;PwdWallet=/wallets/cacheadminwallet"
   Command> DROP CACHE GROUP subscriber_accounts;
   ```

   The `DROP CACHE GROUP` statement updates the metadata on the Oracle database. The objects are dropped if no other TimseTen databases are caching the same tables.

Stopping the Cache Agent

TimesTen provides commands to stop a cache agent.

In TimesTen, call the `ttCacheStop` built-in procedure to stop the cache agent. This must be done on all standalone TimesTen databases and, if used, the active and standby databases of the active standby pair.

From the `cache1`, `cache2`, `cacheactive` and `cachestandby` databases, issue the following built-in procedure call to stop the cache agent on the database:

```
Command> CALL ttCacheStop;
Command> exit
```

Destroying the TimesTen Databases

TimesTen provides commands to destroy a TimesTen database.

1. Ensure you backup all your data, since it will be discarded in the destruction process.
2. Make sure that you drop all cache groups before you attempt to destroy a database. If you cannot drop the cache groups, then use the `-force` option on the destroy operation in the next step. See Dropping a Cache Group.
3. Perform the destroy operation:

   In TimesTen, if the TimesTen databases are no longer needed, you can use the `ttDestroy` utility to destroy the databases.

   Note

   In TimesTen, if the RAM policy designates that the database stays in memory, then this may prevent you from destroying the database. For example, if the RAM policy is set to `always`, then you must change the RAM policy to `manual` and run the `ttAdmin -ramunload` command to unload the database before destroying the database. See Specifying a RAM Policy section in the *Oracle TimesTen In-Memory Database Operations Guide*.

   The following example shows the `ttDestroy` utility connecting to and then destroying the `cache1` database:

   ```
   % ttDestroy cache1
   ```
4. If you used the `-force` option on the destroy operation, run the `cacheCleanup.sql` script to cleanup the metadata and Oracle database objects.

See Installed SQL*Plus Scripts.

Dropping Oracle Database Users and Objects

Use SQL*Plus as the `sys` user to drop the Oracle cache administration user `cacheadmin` and all objects such as tables and triggers owned by the cache administration user.

Then drop the `TT_CACHE_ADMIN_ROLE` role, and the default tablespace `cachetblsp` used by the Oracle cache administration user including the contents of the tablespace and its data file.

```
% sqlplus sys as sysdba
Enter password: password
SQL> DROP USER cacheadmin CASCADE;
SQL> DROP ROLE TT_CACHE_ADMIN_ROLE;
SQL> DROP TABLESPACE cachetblsp INCLUDING CONTENTS AND DATAFILES;
SQL> exit
```

Also, you can run TimesTen SQL*Plus scripts to drop the Oracle Database objects used to implement autorefresh operations. See Managing a Cache Environment with Oracle Database Objects.

Scheduling a Shutdown of Active Standby Pair with AWT Cache Groups

When you are using active standby pairs with AWT cache groups, the environment includes both an active and a standby master, potentially one or more subscribers, and at least one Oracle Database.

The following is the recommended method when you initiate a scheduled shutdown of outstanding transactions in this environment. This order of events provides the time needed to finish applying outstanding transactions before shut down and minimizes the time needed to restart all components.

1. Shut down all applications.
2. Ensure that all transactions have propagated to the Oracle database.
3. Shut down TimesTen.
4. Shut down the Oracle Database.

Then, when you are ready to restart all components:

1. Restart the Oracle Database.
2. Restart TimesTen.
3. Restart any applications.

You can shut down all of these products in any order without error. The order matters only to maximize performance and reduce the need for preserving unapplied transactions. For example, when you are using AWT cache groups within the active standby pair and if you shut down the Oracle database before TimesTen, then all unapplied transactions accumulate in the TimesTen transaction logs. Thus, when you restart TimesTen and Oracle, you could potentially have a lower throughput while pending transactions are applied to the Oracle database. Thus, shutting down TimesTen before the Oracle database provides the most efficient method for your scheduled shutdown and startup. In addition, shutting down the applications before TimesTen stops any additional requests from being sent to an unavailable TimesTen database.

9

Using Cache in an Oracle RAC Environment

The following sections describe how to use cache in an Oracle Real Application Clusters (Oracle RAC) environment:

How Cache Works in an Oracle RAC Environment

Oracle RAC enables multiple Oracle Database instances to access one Oracle database with shared resources, including all data files, control files, PFILEs and redo log files that reside on cluster-aware shared file systems. Oracle RAC handles read/write consistency and load balancing while providing high availability.

Fast Application Notification (FAN) is an Oracle RAC feature that is integrated with Oracle Call Interface (OCI) in Oracle Database. FAN publishes information about changes in the cluster to applications that subscribe to FAN events. FAN prevents unnecessary operations such as the following:

- Attempts to connect when services are down
- Attempts to finish processing a transaction when the server is down
- Waiting for TCP/IP timeouts

See Oracle Real Application Clusters Real Application Clusters Administration and Deployment Guide for more information about Oracle RAC and FAN.

To facilitate cache operations, TimesTen uses OCI integrated with FAN to receive notification of Oracle Database events. With FAN, TimesTen detects connection failures within a minute. Without FAN, it can take several minutes for TimesTen to receive notification of an Oracle Database failure. Without FAN, TimesTen detects a connection failure the next time the connection is used or when a TCP/IP timeout occurs. TimesTen can recover quickly from Oracle Database failures without user intervention.

TimesTen also uses Transparent Application Failover (TAF), which is a feature of Oracle Net Services that enables you to specify how you want applications to reconnect after a failure.

See Oracle AI Database Database Net Services Administrator's Guide for more information about TAF. TAF attempts to reconnect to the Oracle database for four minutes. If this is not successful, the cache agent restarts and attempts to reconnect with the Oracle database every minute.

Note

You can configure how long TAF retries when establishing a connection with the `AgentFailoverTimeout` parameter. For details, see Setting Up Cache in an Oracle RAC Environment.

OCI applications can use one of the following types of Oracle Net failover functionality:

- `None`: No failover functionality is used. This can also be specified to prevent failover from happening. This is the default failover functionality.
- `Session`: If an application's connection is lost, a new connection is automatically created for the application. This type of failover does not attempt to recover selects.
- `Select`: This type of failover enables applications that began fetching rows from a cursor before failover to continue fetching rows after failover.

The behavior of cache operations depend on the actions of TAF and how TAF is configured. By default, TAF and FAN callbacks are installed if you are using cache in an Oracle RAC environment. If you do not want TAF and FAN capabilities, set the `RACCallback` connection attribute to 0.

Table 9-1 shows the behaviors of cache operations in an Oracle RAC environment with different TAF failover types.

Table 9-1 Behavior of Cache Operations in an Oracle RAC Environment

Operation	TAF Failover Type	Behavior After a Failed Connection on the Oracle Database
Autorefresh	None	The cache agent automatically stops, restarts and waits until a connection can be established on the Oracle database. This behavior is the same as in a non-Oracle RAC environment. No user intervention is needed.
Autorefresh	`Session`	One of the following occurs: • All failed connections are recovered. Autorefresh operations that were in progress are rolled back and retried. • If TAF times out or cannot recover the connection, the cache agent automatically stops, restarts and waits until a connection can be established on the Oracle database. • In all cases, no user intervention is needed.
Autorefresh	`Select`	One of the following occurs: • Autorefresh operations resume from the point of connection failure. • Autorefresh operations that were in progress are rolled back and retried. • If TAF times out or cannot recover the connection, the cache agent automatically stops, restarts and waits until a connection can be established on the Oracle database. • In all cases, no user intervention is needed.
AWT	None	The receiver thread of the replication agent for the AWT cache group exits. A new thread is spawned and tries to connect to the Oracle database. No user intervention is needed.

Table 9-1 (Cont.) Behavior of Cache Operations in an Oracle RAC Environment

Operation	TAF Failover Type	Behavior After a Failed Connection on the Oracle Database
AWT	`Session`, `Select`	One of the following occurs: • If the connection is recovered and there are uncommitted DML operations in the transaction, the transaction is rolled back and then reissued. • If the connection is recovered and there are no uncommitted DML operations, new operations can be issued without rolling back. In all cases, no user intervention is needed.
SWT, propagate, flush, and passthrough	None	The application is notified of the connection loss. The cache agent disconnects from the Oracle database and the current transaction is rolled back. All modified session attributes are lost. During the next passthrough operation, the cache agent tries to reconnect to the Oracle database. This behavior is the same as in a non-Oracle RAC environment. No user intervention is needed.
SWT, propagate, flush and passthrough SWT, propagate and flush	`Session` `Select`	One of the following occurs: • The connection to the Oracle database is recovered. If there were open cursors, DML or lock operations on the lost connection, an error is returned and the user must roll back the transaction before continuing. Otherwise, the user can continue without rolling back. • If TAF times out or cannot recover the connection, the application is notified of the connection loss. The cache agent disconnects from the Oracle database and the current transaction is rolled back. All modified session attributes are lost. During the next passthrough operation, the cache agent tries to reconnect to the Oracle database. In this case, no user intervention is needed.
Passthrough	`Select`	The connection to the Oracle database is recovered. If there were DML or lock operations on the lost connection, an error is returned and the user must roll back the transaction before continuing. Otherwise, the user can continue without rolling back.
Load and refresh	None	The application receives a loss of connection error.
Load and refresh	`Session`	One of the following occurs: • The load or refresh operation succeeds. • An error is returned stating that a fetch operation on Oracle Database cannot be processed.

Table 9-1 (Cont.) Behavior of Cache Operations in an Oracle RAC Environment

Operation	TAF Failover Type	Behavior After a Failed Connection on the Oracle Database
Load and refresh	`Select`	One of the following occurs: • If the Oracle Database cursor is open and the cursor is recovered, or if the Oracle Database cursor is not open, then the load or refresh operation succeeds. • An error is returned if TAF was unable to recover either the session or open Oracle Database cursors. **Note**: An error is less likely to be returned than if the TAF failover type is Session.

Restrictions on Using Cache in an Oracle RAC Environment

There are some restrictions for cache support of Oracle RAC.

The restrictions for cache support of Oracle RAC are:

- Cache operations are limited to Oracle RAC, FAN and TAF capabilities. For example, if all nodes for a service fail, the service is not restarted. TimesTen waits for the user to restart the service.
- TAF does not recover `ALTER SESSION` operations. The user is responsible for restoring changed session attributes after a failover.
- For cache operations, TimesTen uses OCI integrated with FAN. This interface automatically spawns a thread to wait for an Oracle Database event. This is the only TimesTen feature that spawns a thread in a TimesTen application with the direct driver. Adapt your application to account for this thread creation. If you do not want the extra thread, set the `RACCallback` connection attribute to 0 so that TAF and FAN are not used.

Setting Up Cache in an Oracle RAC Environment

You can set up a cache in a TimesTen database to cache data within an Oracle RAC environment.

After you install Oracle RAC and cache, perform the following to set up a cache for an Oracle RAC environment:

1. On TimesTen, set the TAF timeout, in minutes, with the `ttCacheConfig` `AgentFailoverTimeout` parameter. The `AgentFailoverTimeout` parameter configures how long TAF retries when establishing a connection. TAF attempts to reconnect to the Oracle database for the duration of this timeout. The default is four minutes. If this is not successful, the cache agent restarts and attempts to reconnect with the Oracle database every minute; the replication agent restarts any threads that cannot connect to the Oracle database.
2. Make sure that the TimesTen daemon, the cache agent, and the following Oracle Database components are started:
   - Oracle Database instances
   - Oracle Database listeners

- Oracle Database service that is used for cache operations

3. Verify that the TimesTen `RACCallback` connection attribute is set to 1 (default). See RACCallback in the *Oracle TimesTen In-Memory Database Reference*.
4. Use the `DBMS_SERVICE.MODIFY_SERVICE` function or Oracle Enterprise Manager to enable publishing of FAN events. This changes the value in the `AQ_HA_NOTIFICATIONS` column of the Oracle Database `ALL_SERVICES` view to `YES`.

   See *Oracle Database PL/SQL Packages and Types Reference* for more information about the `DBMS_SERVICE` Oracle Database PL/SQL package.
5. Enable TAF on the Oracle Database service used for cache operations on TimesTen with *one* of the following methods:
   - Create a service for TimesTen in the Oracle Database `tnsnames.ora` file with the following settings:
     – `LOAD_BALANCE=ON` (optional)
     – `FAILOVER_MODE=(TYPE=SELECT)` *or* `FAILOVER_MODE=(TYPE=SESSION)`
   - Use the `DBMS_SERVICE.MODIFY_SERVICE` function to set the TAF failover type.

     See Oracle AI Database Database Net Services Administrator's Guide for more information about enabling TAF.
6. If you have a TimesTen application that uses the direct driver, link it with a thread library so that it receives FAN notifications. FAN spawns a thread to monitor for failures.

10

Using Cache with Data Guard

You need to configure cache when you want cache to work with either synchronous or asynchronous Data Guard. Data Guard support is only included within TimesTen.

- Components of MAA for Cache
- Cache in TimesTen Works with Asynchronous Active Data Guard
- Cache in TimesTen Works with Synchronous Data Guard

Components of MAA for Cache

Oracle Maximum Availability Architecture (MAA) is Oracle Database's best practices blueprint based on proven Oracle Database high availability (HA) technologies and recommendations. The goal of MAA is to achieve the optimal high availability architecture at the lowest cost and complexity.

To be compliant with MAA, cache must support Oracle Real Application Clusters (Oracle RAC) and Oracle Data Guard, as well as have its own HA capability.

Cache provides its own HA capability through active standby pair replication of cache tables in read-only and AWT cache groups. See Using Cache in an Oracle RAC Environment.

Oracle Data Guard provides the management, monitoring, and automation software infrastructure to create and maintain one or more synchronized standby Oracle databases to protect data from failures, disasters, errors, and corruptions. If the primary Oracle database becomes unavailable because of a planned or an unplanned outage, Data Guard can switch any standby Oracle database to the primary role, thus minimizing downtime and preventing any data loss. See *Oracle Data Guard Concepts and Administration*.

The MAA framework supports cache tables in static read-only and AWT cache groups. For cache tables in dynamic cache groups of any cache group type, SWT cache groups, and user managed cache groups that use the `AUTOREFRESH` cache group attribute, TimesTen cannot access the Oracle database during a failover and switchover because cache applications wait until the failover and switchover completes.

In general, however, all cache groups types are supported with synchronous Data Guard or Data Guard during planned maintenance.

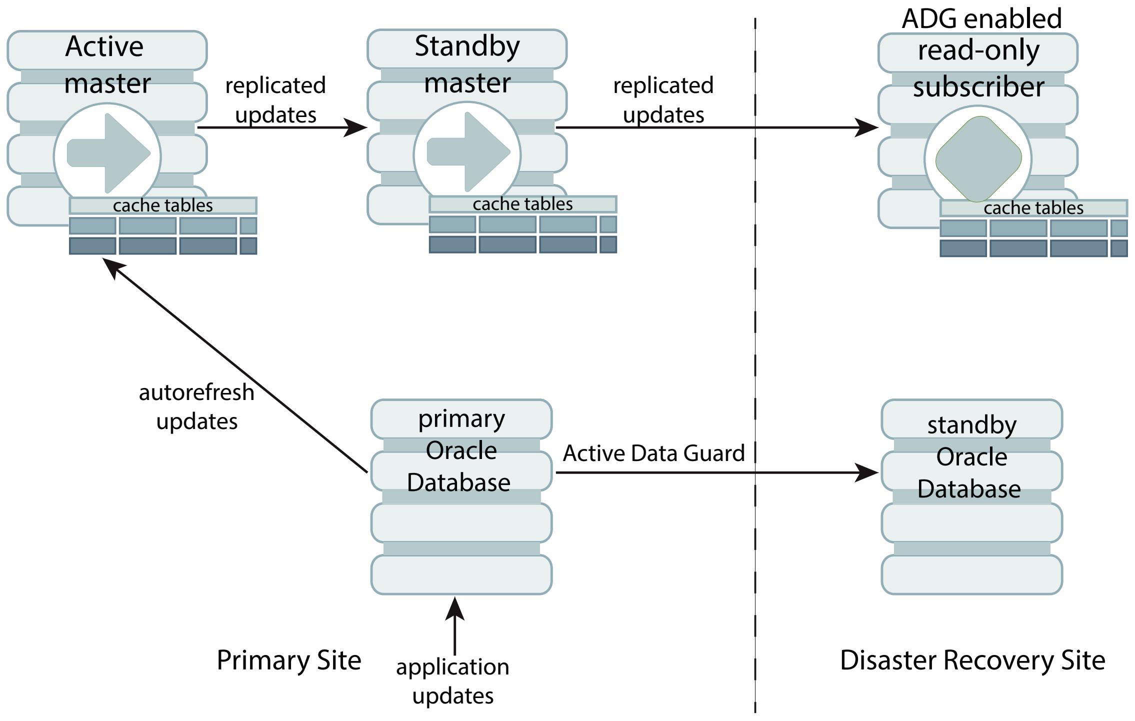
Cache in TimesTen Works with Asynchronous Active Data Guard

You can cache tables from an Oracle Active Data Guard with the asynchronous redo transport mode into read-only cache groups.

When using cache with Active Data Guard, you can only use read-only cache groups that are replicated within an active standby pair replication scheme.

The Active Data Guard configuration includes a primary Oracle database that communicates over an asynchronous transport to a single physical standby Oracle database. As shown in Figure 10-1, the primary Oracle database is located on the primary site, while the standby Oracle database is located on a disaster recovery site.

Figure 10-1 Recommended Configuration for Asynchronous Active Data Guard



On TimesTen, the read-only cache groups on the primary site are autorefreshed from the primary Oracle database; however, the only transactions that are autorefreshed are those whose changes have been successfully replicated to the standby Oracle database. Once refreshed to the active master, all changes are then propagated to the TimesTen standby master and a read-only subscriber using standard TimesTen replication processes.

For the best failover and recovery action, you should locate the read-only subscriber on the same disaster recovery site as the standby Oracle database. Create this read-only subscriber with the `ttRepAdmin -duplicate -activeDataGuard` utility option, which replicates the read-only cache groups directly to the subscriber as it would to a standby master database. That is, instead of the cache groups being converted to tables when replicated to a subscriber, the cache groups themselves are replicated to the read-only subscriber. This is to provide a recovery and failover option if the primary site fails. See Recovery After Failure When Using Asynchronous Active Data Guard.

The following sections provide more details on the environment for asynchronous Active Data Guard when using replicated read-only cache groups:

- Configuring the Primary and Standby Oracle Databases
- Configuring the Active Standby Pair with Read-Only Cache Groups
- Recovery After Failure When Using Asynchronous Active Data Guard

Configuring the Primary and Standby Oracle Databases

When you create and configure Active Data Guard with primary and standby Oracle databases, ensure that the configuration includes specific configuration that supports the TimesTen cache environment.

1. Configure both the primary and standby Oracle databases to use Flashback queries. See Configuring Recovery Settings in the *Oracle Database 2 Day DBA guide*.
2. The Data Guard configuration must be managed by the Data Guard Broker so that the TimesTen daemon processes and application clients respond faster to failover and switchover events. See the *Data Guard Broker* guide.
3. Create two supporting database services on both the primary and standby Oracle databases in the Oracle Cluster. One database service points to the primary Oracle Database and the other points to the physical standby Oracle Database. You can create these either through role based services or through system triggers.

   See the following sections for details.

   - Configuring Oracle Database Services Through Role Based Services
   - Configuring Oracle Database Services Through System Triggers

Configuring Oracle Database Services Through Role Based Services

You can automatically control the startup of Oracle database services on both the primary and standby Oracle databases by assigning a database role to each service.

An Oracle database service automatically starts when the Oracle database starts if the Oracle database policy is set to `AUTOMATIC` and if the service role matches the current role of the database. In this case, the role for the Oracle database is either in the primary or standby role as part of the Active Data Guard configuration.

Configure services with the `srvctl` utility identically on all Oracle databases in the Data Guard configuration. The following example shows two services created identically on both the primary and the standby Oracle databases. See srvctl add service in the *Oracle Database Administrator's Guide*.

The following steps add the `primaryrole` and `standbyrole` database services to both the primary and standby Oracle databases when the primary Oracle database is located in Austin and the standby Oracle database is located in Houston.

1. On the primary Oracle database, add the `primaryrole` database service. While this Oracle database acts as the primary, this service is started.

   ```
   srvctl add service -d Austin -s primaryrole -r ssa1,ssa2,ssa3,
    ssa4 -l PRIMARY -q TRUE -e SESSION -m BASIC -w 10 -z 150
   ```

2. On the primary Oracle database, add the `standbyrole` database service. This service starts only if this Oracle database switches to the standby role and then provides real-time reporting on the standby Oracle database.

   ```
   srvctl add service -d Austin -s standbyrole -r ssa1,ssa2,ssa3,
    ssa4 -l PHYSICAL_STANDBY -q TRUE -e SESSION -m BASIC -w 10 -z 150
   ```

3. On the standby Oracle database, add the `primaryrole` database service. This service starts only if this Oracle database switches to the primary role.

   ```
   srvctl add service -d Houston -s primaryrole -r ssb1,ssb2,ssb3,
    ssb4 -l PRIMARY -q TRUE -e SESSION -m BASIC -w 10 -z 150
   ```

4. On the standby Oracle database, add the `standbyrole` database service. While this Oracle database acts as the standby, this service is started and then provides real-time reporting on the standby Oracle database.

   ```
   srvctl add service -d Houston -s standbyrole -r ssb1,ssb2,ssb3,
    ssb4 -l PHYSICAL_STANDBY -q TRUE -e SESSION -m BASIC -w 10 -z 150
   ```

5. Run the following SQL statement on the primary Oracle database so that the service definitions are transmitted and applied to the physical standby Oracle database.

   ```
   EXECUTE DBMS_SERVICE.CREATE_SERVICE('standbyrole', 'standbyrole', NULL,
    NULL, TRUE, 'BASIC', 'SESSION', 150, 10, NULL);
   ```

6. Add connection aliases in the appropriate `tnsnames.ora` files to identify the primary and standby Oracle databases and specify the database service names for each.

   ```
   primaryinstance=
     (DESCRIPTION_LIST=
       (LOAD_BALANCE=off)
       (FAILOVER=on)
       (DESCRIPTION=(ADDRESS_LIST=(LOAD_BALANCE=on)
              (ADDRESS=(PROTOCOL=TCP)(HOST=myhost1)(PORT=1521)))
                  (CONNECT_DATA=(SERVER=DEDICATED)(SERVICE_NAME=primaryrole)))

       (DESCRIPTION=(ADDRESS_LIST=(LOAD_BALANCE=on)
              (ADDRESS=(PROTOCOL=TCP)(HOST=myhost2)(PORT=1521)))
                  (CONNECT_DATA=(SERVER=DEDICATED)(SERVICE_NAME=primaryrole))))

   standbyinstance=
      (DESCRIPTION_LIST=
        (LOAD_BALANCE=off)
        (FAILOVER=on)
        (DESCRIPTION=(ADDRESS_LIST=(LOAD_BALANCE=on)
               (ADDRESS=(PROTOCOL=TCP)(HOST=myhost1)(PORT=1521)))
                   (CONNECT_DATA=(SERVER=DEDICATED)(SERVICE_NAME=standbyrole)))

        (DESCRIPTION=(ADDRESS_LIST=(LOAD_BALANCE=on)
               (ADDRESS=(PROTOCOL=TCP)(HOST=myhost2)(PORT=1521)))
                  (CONNECT_DATA=(SERVER=DEDICATED)(SERVICE_NAME=standbyrole))))
   ```

7. On the primary Oracle database, start the `primaryrole` database service.

   ```
   srvctl start service -d Austin -s primaryrole
   ```

8. On the standby Oracle database, start the `standbyrole` database service.

   ```
   srvctl start service -d Houston -s standbyrole
   ```

Configuring Oracle Database Services Through System Triggers

You can perform certain steps to create the `primaryrole` and `standbyrole` database services on the primary Oracle database using triggers. After creation, these are replicated to the standby Oracle database.

1. Create the `primaryrole` and `standbyrole` database services in the primary Oracle database.

   ```
   exec DBMS_SERVICE.CREATE_SERVICE(
    service_name => 'primaryrole',
    network_name => 'primaryrole',
    aq_ha_notifications => true, failover_method => 'BASIC',
    failover_type => 'SELECT', failover_retries => 180, failover_delay => 1 );

   exec DBMS_SERVICE.CREATE_SERVICE(
    service_name => 'standbyrole',
    network_name => 'standbyrole',
    aq_ha_notifications => true, failover_method => 'BASIC',
    failover_type => 'SELECT', failover_retries => 180, failover_delay => 1 );
   ```

2. Create the `primaryrole` and `standbyrole` triggers in the primary Oracle database for when the database starts.

```
CREATE OR REPLACE TRIGGER manage_OCIService
after startup on database
DECLARE
  role VARCHAR(30);
BEGIN
  SELECT DATABASE_ROLE INTO role FROM V$DATABASE;
  IF role = 'PRIMARY' THEN
    BEGIN
      DBMS_SERVICE.START_SERVICE('primaryrole');
    EXCEPTION
      WHEN OTHERS THEN
        NULL;
    END;
    BEGIN
      DBMS_SERVICE.STOP_SERVICE('standbyrole');
    EXCEPTION
      WHEN OTHERS THEN
        NULL;
    END;
  ELSE
    BEGIN
      DBMS_SERVICE.STOP_SERVICE('primaryrole');
    EXCEPTION
      WHEN OTHERS THEN
        NULL;
    END;
    BEGIN
      DBMS_SERVICE.START_SERVICE('standbyrole');
    EXCEPTION
      WHEN OTHERS THEN
        NULL;
    END;
  END IF;
END;
```

3. Create the following trigger on the primary Oracle database to run when the database changes roles:

```
CREATE OR REPLACE TRIGGER manage_OCIService2
AFTER DB_ROLE_CHANGE ON DATABASE
DECLARE
  role VARCHAR(30);
BEGIN
  SELECT DATABASE_ROLE INTO role FROM V$DATABASE;
  IF role = 'PRIMARY' THEN
    BEGIN
      DBMS_SERVICE.START_SERVICE('primaryrole');
    EXCEPTION
      WHEN OTHERS THEN
        NULL;
    END;
    BEGIN
      DBMS_SERVICE.STOP_SERVICE('standbyrole');
    EXCEPTION
      WHEN OTHERS THEN
        NULL;
    END;
  ELSE
    BEGIN
      DBMS_SERVICE.STOP_SERVICE('primaryrole');
    EXCEPTION
      WHEN OTHERS THEN
        NULL;
    END;
    BEGIN
      DBMS_SERVICE.START_SERVICE('standbyrole');
    EXCEPTION
      WHEN OTHERS THEN
        NULL;
    END;
  END IF;
END;
```

4. Add connection aliases in the appropriate `tnsnames.ora` files to identify the primary and standby Oracle databases and specify the database service names for each.

```
primaryinstance=
  (DESCRIPTION_LIST=
    (LOAD_BALANCE=off)
    (FAILOVER=on)
    (DESCRIPTION=(ADDRESS_LIST=(LOAD_BALANCE=on)
           (ADDRESS=(PROTOCOL=TCP)(HOST=myhost1)(PORT=1521)))
               (CONNECT_DATA=(SERVER=DEDICATED)(SERVICE_NAME=primaryrole)))

    (DESCRIPTION=(ADDRESS_LIST=(LOAD_BALANCE=on)
           (ADDRESS=(PROTOCOL=TCP)(HOST=myhost2)(PORT=1521)))
               (CONNECT_DATA=(SERVER=DEDICATED)(SERVICE_NAME=primaryrole))))

standbyinstance=
   (DESCRIPTION_LIST=
     (LOAD_BALANCE=off)
     (FAILOVER=on)
     (DESCRIPTION=(ADDRESS_LIST=(LOAD_BALANCE=on)
            (ADDRESS=(PROTOCOL=TCP)(HOST=myhost1)(PORT=1521)))
                (CONNECT_DATA=(SERVER=DEDICATED)(SERVICE_NAME=standbyrole)))

     (DESCRIPTION=(ADDRESS_LIST=(LOAD_BALANCE=on)
            (ADDRESS=(PROTOCOL=TCP)(HOST=myhost2)(PORT=1521)))
               (CONNECT_DATA=(SERVER=DEDICATED)(SERVICE_NAME=standbyrole))))
```

5. Restart both of the Oracle databases to enable the trigger to start and stop the correct database services. Alternatively, if you do not want to restart both Oracle databases, you can start and stop the appropriate database services on each Oracle database as follows:

   On the primary Oracle database:

```
exec DBMS_SERVICE.START_SERVICE('primaryrole');
exec DBMS_SERVICE.STOP_SERVICE('standbyrole');
```

   On the standby Oracle database:

```
exec DBMS_SERVICE.STOP_SERVICE('primaryrole');
exec DBMS_SERVICE.START_SERVICE('standbyrole');
```

Configuring the Active Standby Pair with Read-Only Cache Groups

The Active Data Guard with asynchronous redo transport mode supports an active standby pair replication scheme that only contains replicated read-only cache groups.

All replicated read-only cache groups must be created before you create the active standby pair. You cannot exclude a replicated read-only cache group when you are creating the active standby pair and you cannot add another replicated read-only cache group to the active standby pair after creation.

When you create and configure an active standby pair to support replicated read-only cache groups, perform the following to support asynchronous Active Data Guard:

1. When you create the active standby pair, we recommend that you keep both the active and standby masters within the same physical site. They can be on different hosts within the same site.
2. If you want a read-only subscriber for disaster recovery, you can add a read-only subscriber on the same disaster recovery site as the standby Oracle database and enable the subscriber for cache groups. The subscriber that you should create when using Active Data Guard is created with a duplicate operation with the `ttRepAdmin -duplicate -activeDataGuard` options.

   The `-activeDataGuard` option, which is solely for the Active Data Guard environment, enables the subscriber to keep replicated read-only cache groups intact as it would for a standby master. Since the subscriber retains these cache groups, you must provide the Oracle cache administration user name and password on the `ttRepAdmin` utility command line.

   > **Note**
   >
   > Alternatively, you can use the `ttRepDuplicateEx` C function setting the `TT_REPDUP_ADG` flag in `ttRepDuplicateExArg.flags`.

   The following example creates a read-only subscriber on the disaster recovery site duplicating from the standby master providing the `-activeDataGuard` option, the cache administration user name and passwords.

   ```
   ttRepAdmin -duplicate -from master2 -host node1
    -uid cacheadmin -pwd timesten -cacheuid cacheadmin -cachepwd orapwd
    -activeDataGuard adgsubscriber
   ```
3. Create the cache environment on the primary Oracle database. You do not need to perform any of these steps on the standby Oracle database.
4. On the primary Oracle database, grant the Oracle cache administration user the `EXECUTE` privilege for the `SYS.DBMS_FLASHBACK` package. This privilege is granted as part of the `initCacheAdminSchema.sql` and `grantCacheAdminPrivileges.sql` scripts.
5. Configure the same connection attributes that you would for a TimesTen database that caches data from an Oracle database. In addition, since we are also monitoring transactions from the standby Oracle database, configure the `StandbyNetServiceName` connection attribute with the net service name of the standby Oracle database instance.

   On Microsoft Windows systems, the net service name of the Oracle database instance is specified in the **Oracle Net Service Name** field of the TimesTen Cache tab within the TimesTen ODBC Setup dialog box. The standby Oracle database instance is specified in the **Standby Oracle Net Service Name** field on the same page.

   Configure the `StandbyNetServiceName` `ODBC.INI` attribute on the active master to configure the net service name of the physical standby Oracle database:

   ```
   [cachedb]
   DataStore=/myDb/cachedb
   PermSize=256
   TempSize=256
   DatabaseCharacterSet=WE8DEC
   OracleNetServiceName=primaryinstance
   StandbyNetServiceName=standbyinstance
   ```

Recovery After Failure When Using Asynchronous Active Data Guard

There are recommended recovery procedures if the primary Oracle database fails, the standby Oracle database fails, or the entire primary site fails taking down the primary Oracle database as well as the active and standby masters.

- Failure of the Standby Oracle Database
- Failure of the Primary Oracle Database
- Failure of the Primary Site

Failure of the Standby Oracle Database

When the standby Oracle database in an Active Data Guard configuration fails, the cache agent retries the connection to the standby Oracle database.

Note

You can notify the cache agent of whether the standby Oracle database is active or has failed by calling the `ttCacheADGStandbyStateSet` built-in procedure with either the `ON` or the `FAILED` arguments.

- If a timeout is set, then the cache agent waits for the amount of time specified with the `ttCacheADGStandbyTimeoutSet` built-in procedure. If the standby Oracle database has not recovered after this period, then the cache agent sets the state of the standby Oracle database by calling the `ttCacheADGStandbyStateSet` built-in procedure with the `FAILED` argument and then facilitates autorefresh using only the primary Oracle database.
- If no timeout has been set with the `ttCacheADGStandbyTimeoutSet` built-in procedure (default value is 0), then the cache agent continues to wait on the standby Oracle database, unless you inform the cache agent that the standby Oracle database is not recovering by calling the `ttCacheADGStandbyStateSet` built-in procedure with the `FAILED` argument.

Once the state of the standby Oracle database is set to `FAILED`, the cache agent resumes autorefresh with only the primary Oracle database until you reset the state of the standby Oracle database by calling the `ttCacheADGStandbyStateSet` built-in procedure with the `ON` argument. Even if the standby Oracle database eventually does recover, the cache agent does not recognize that the standby Oracle database is active until you reset its state to `ON`.

Once the state of the standby Oracle database is set to `ON`, the cache agent pauses to wait for the standby Oracle database to catch up to the primary Oracle database. After which, the cache agent resumes autorefresh from the primary Oracle database for those transactions that have successfully replicated to the standby Oracle database.

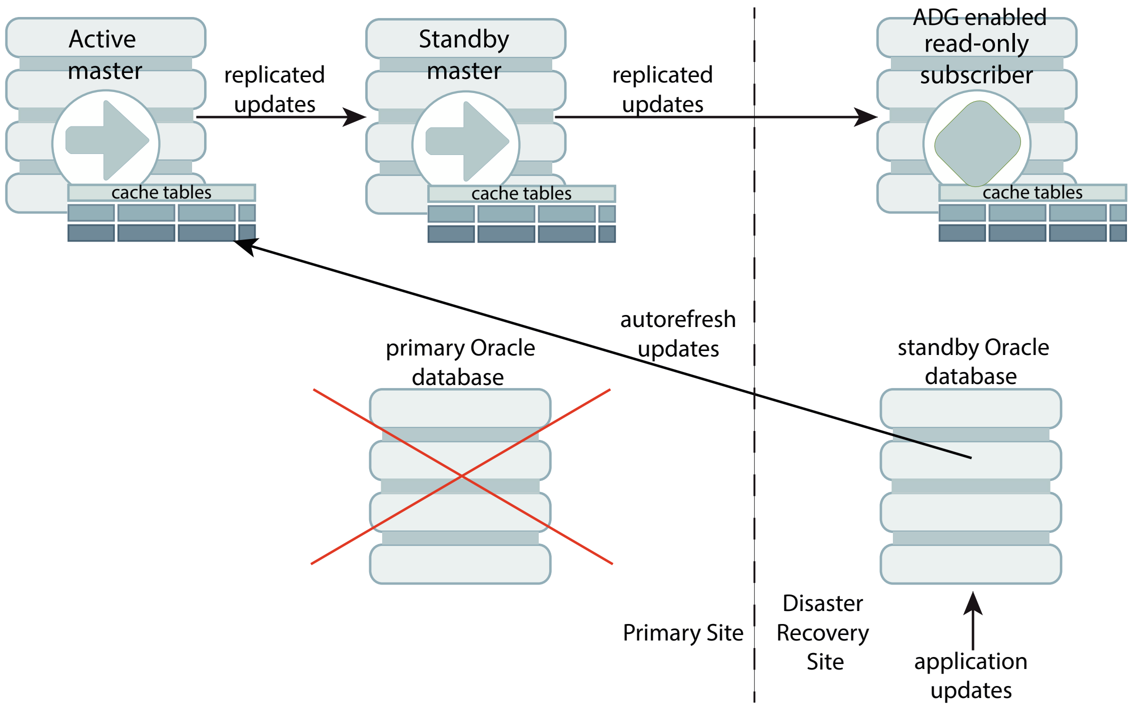
You can restore the original Active Data Guard configuration by dropping the active standby pair and then loading the cache groups.

See ttCacheADGStandbyTimeoutSet and ttCacheADGStandbyStateSet in the *Oracle TimesTen In-Memory Database Reference*.

Failure of the Primary Oracle Database

If the primary Oracle database fails, then Data Guard switches over to the standby Oracle database and the TimesTen cache agent switches autorefresh over to the new primary Oracle database.

Figure 10-2   Failure of the Primary Oracle Database



Failure of the Primary Site

If the entire site where the primary Oracle database as well as the active and standby master databases are located fails, then the standby Oracle database becomes the primary Oracle database.

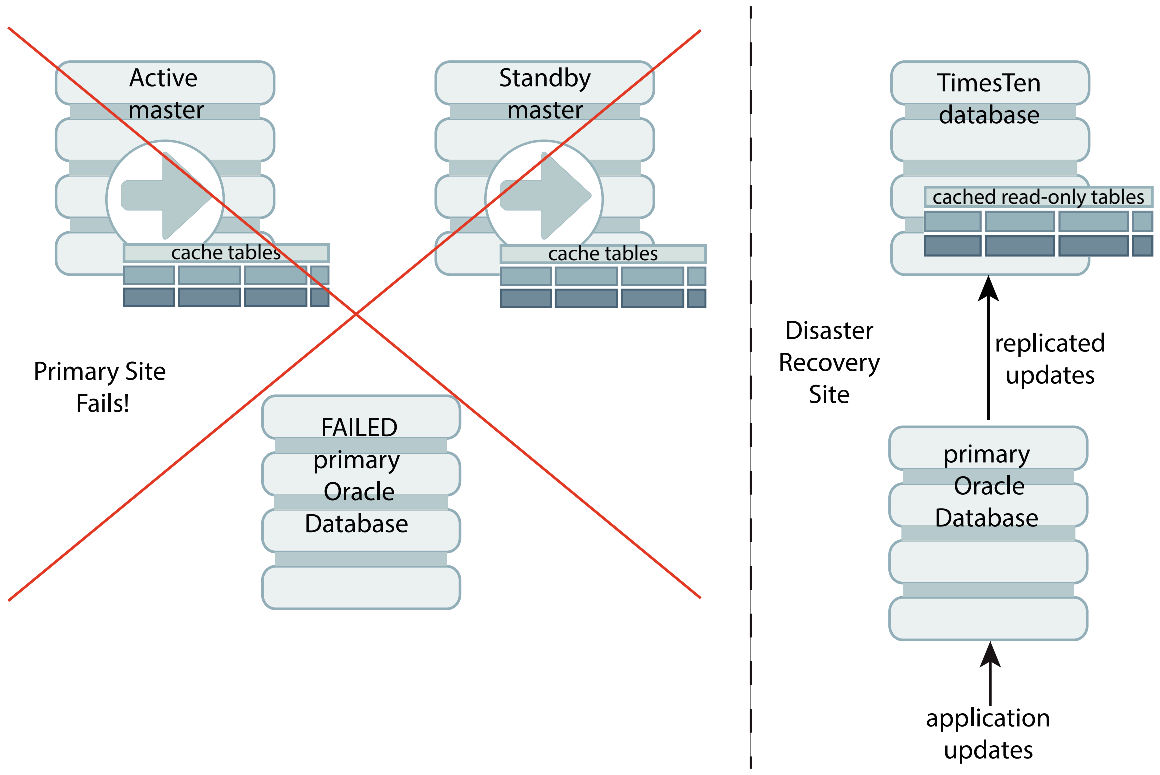
After which, you may want the disaster recovery site to become the primary TimesTen database. Thus, on the disaster recovery site, the standby Oracle database is now a sole Oracle database and the read-only subscriber becomes a single TimesTen database that caches data in the Oracle database.

Transform the subscriber into a single TimesTen database with cached tables by:

1. Drop the active standby pair on the TimesTen database on the disaster recovery site.
2. Alter the existing read-only cache groups on the disaster recovery site to set the autorefresh state to on.

After which, the cache tables on the TimesTen database in the disaster recovery site receive updates from the new primary Oracle database.

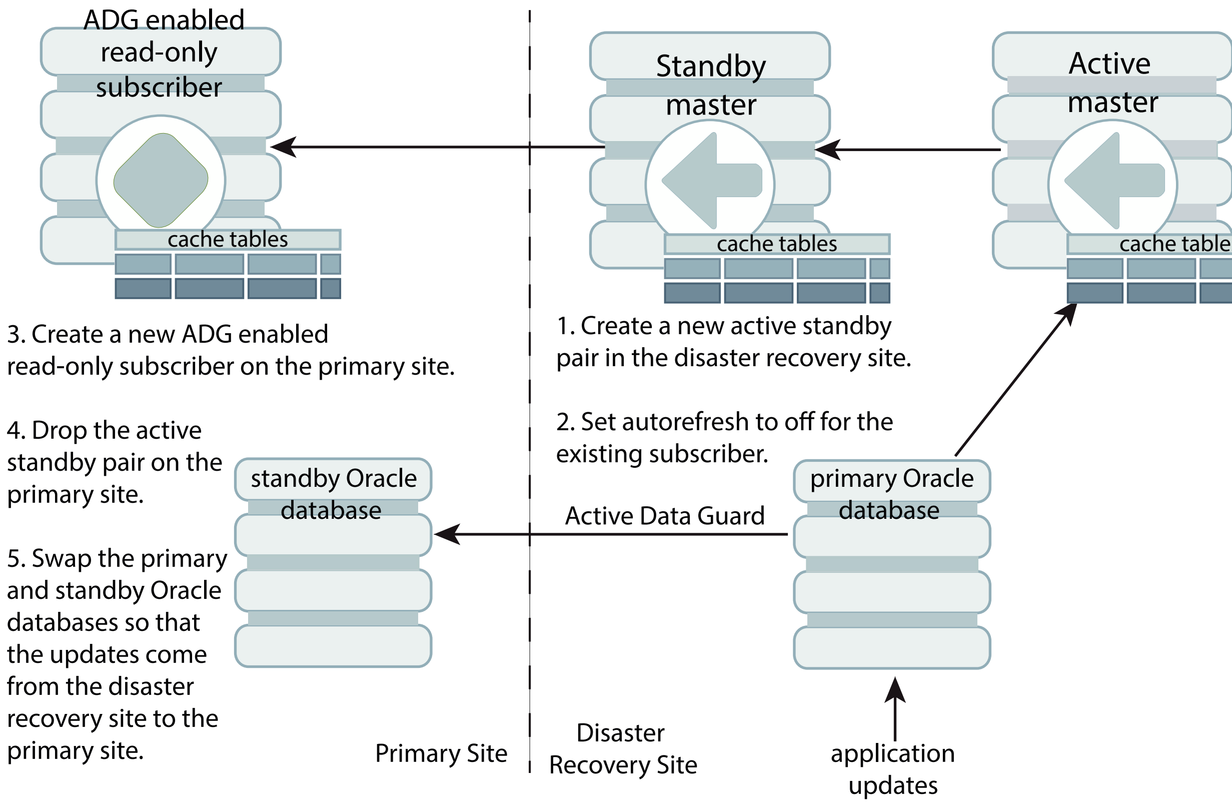
Figure 10-3 Recovery After Failure of Primary Site



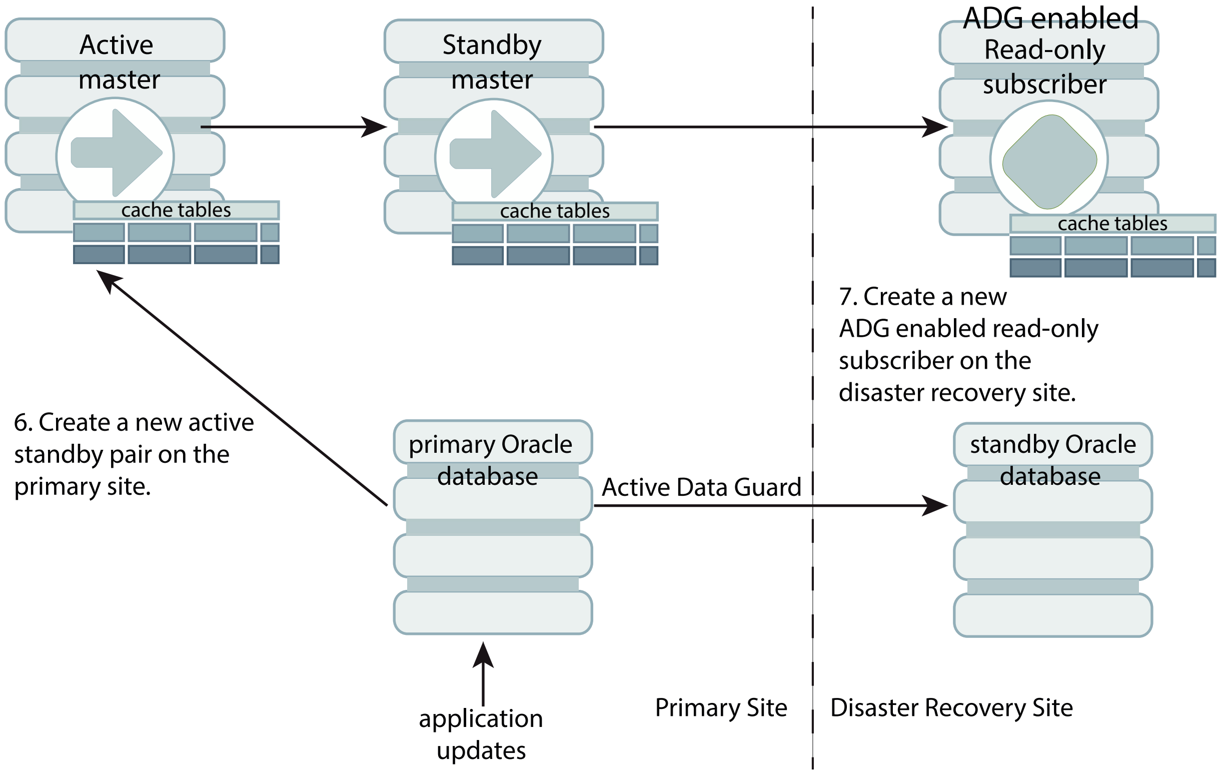
The following is the process to recover a failed primary site and rebuild your environment to the original state:

1. Create a new active standby pair on the disaster recovery site.
2. Alter the existing read-only cache groups on the disaster recovery site to set the autorefresh state to off to stop any future updates from the primary Oracle database.
3. Create the ADG enabled read-only subscriber on the recovered primary site.
4. Drop the active standby pair on the ADG enabled read-only subscriber on the primary site, if it still exists after recovering the primary site.
5. Switch over the Oracle databases in the Active Data Guard. Currently, the applications are updating the primary Oracle database on the disaster recovery site. However, once you recover the Oracle database on the primary site, we want it to take over again as the primary and to make the Oracle database on the disaster recovery site as the secondary.

   The TimesTen database starts to receive updates from the Oracle database on the primary site.


ADG enabled read-only subscriber
cache tables
Standby master
cache tables
Active master
cache table
3. Create a new ADG enabled read-only subscriber on the primary site.
1. Create a new active standby pair in the disaster recovery site.
4. Drop the active standby pair on the primary site.
2. Set autorefresh to off for the existing subscriber.
standby Oracle database
primary Oracle database
Active Data Guard
5. Swap the primary and standby Oracle databases so that the updates come from the disaster recovery site to the primary site.
Primary Site
Disaster Recovery Site
application updates

6. Create a new active standby pair on the primary site.
7. Create a new ADG enabled read-only subscriber on the disaster recovery site.



Cache in TimesTen Works with Synchronous Data Guard

Cache in TimesTen works with synchronous physical standby failover and switchover and logical standby switchover as long as the object IDs for cached Oracle Database tables remain the same on the primary and standby Oracle databases.

Object IDs can change if the table is dropped and re-created, altered, or a truncated flashback operation or online segment shrink is performed.

During a transient upgrade, a physical standby Oracle database is transformed into a logical standby Oracle database. For the time that the standby Oracle database is logical, the user must ensure that the object IDs of the cached Oracle Database tables do not change. Specifically, tables that are cached should not be dropped and re-created, truncated, altered, flashed back or have an online segment shrunk.

The following sections describe how to configure the Oracle and TimesTen databases.

- Configuring the Oracle Databases for TimesTen and Synchronous Data Guard
- Configuring the TimesTen Database to Work with Synchronous Data Guard

Configuring the Oracle Databases for TimesTen and Synchronous Data Guard

You can configure TimesTen to fail over and switch over when using synchronous Data Guard.

In order for TimesTen to fail over and switch over properly, configure the primary and standby Oracle databases using the following steps:

1. The Data Guard configuration must be managed by the Data Guard Broker so that the TimesTen daemon processes and application clients respond faster to failover and switchover events.
2. If you are configuring an Oracle RAC database, use the Oracle Enterprise Manager Cluster Managed Database Services Page to create Oracle database services that TimesTen and its client applications use to connect to the Oracle primary database. See Workload Management with Dynamic Database Services in *Oracle Real Application Clusters Administration and Deployment Guide*.
3. If you created the Oracle database service in step 2, use the `MODIFY_SERVICE` function of the `DBMS_SERVICE` PL/SQL package to modify the service to enable high availability notification to be sent through Advanced Queuing (AQ) by setting the `aq_ha_notifications` attribute to `TRUE`. To configure server side TAF settings, set the failover attributes, as shown in the following example:

```
BEGIN
DBMS_SERVICE.MODIFY_SERVICE
(service_name => 'DBSERV',
 goal => DBMS_SERVICE.GOAL_NONE,
 dtp => false,
 aq_ha_notifications => true,
 failover_method => 'BASIC',
 failover_type => 'SELECT',
 failover_retries => 180,
 failover_delay => 1);
END;
```

4. If you did not create the database service in step 2, use the `CREATE_SERVICE` function of the `DBMS_SERVICE` PL/SQL package to create the database service, enable high availability notification, and configure server side TAF settings:

```
BEGIN
DBMS_SERVICE.CREATE_SERVICE
(service_name => 'DBSERV',
 network_name => 'DBSERV',
 goal => DBMS_SERVICE.GOAL_NONE,
 dtp => false,
 aq_ha_notifications => true,
 failover_method => 'BASIC',
 failover_type => 'SELECT',
 failover_retries => 180,
 failover_delay => 1);
END;
```

5. Create two triggers to relocate the database service to a Data Guard standby database (Oracle RAC or non-Oracle RAC) after it has switched to the primary role. The first trigger fires on the system start event and starts up the `DBSERV` service:

```
CREATE OR REPLACE TRIGGER manage_service
AFTER STARTUP ON DATABASE
DECLARE
  role VARCHAR(30);
BEGIN
  SELECT database_role INTO role FROM v$database;
  IF role = 'PRIMARY' THEN
    dbms_service.start_service('DBSERV');
  END IF;
END;
```

   The second trigger fires when the standby database remains open during a failover and switchover upon a database role change. It relocates the `DBSERV` service from the old primary to the new primary database and disconnects any connections to that service on the old primary database so that TimesTen and its client applications can reconnect to the new primary database:

```
CREATE OR REPLACE TRIGGER relocate_service
AFTER DB_ROLE_CHANGE ON DATABASE
DECLARE
  role VARCHAR(30);
BEGIN
  SELECT database_role INTO role FROM v$database;
  IF role = 'PRIMARY' THEN
    dbms_service.start_service('DBSERV');
  ELSE
    dbms_service.stop_service('DBSERV');
  dbms_lock.sleep(2);
  FOR x IN (SELECT s.sid, s.serial#
            FROM v$session s, v$process p
            WHERE s.service_name='DBSERV' AND s.paddr=p.addr)
    LOOP
      BEGIN
        EXECUTE IMMEDIATE
           'ALTER SYSTEM DISCONNECT SESSION
           ''' || x.sid || ','|| x.serial# || ''' IMMEDIATE';
        EXCEPTION WHEN OTHERS THEN
        BEGIN
          DBMS_OUTPUT.PUT_LINE(DBMS_UTILITY.FORMAT_ERROR_STACK);
        END;
      END;
    END LOOP;
  END IF;
END;
```

6. As an option, to reduce the performance impact to TimesTen applications and minimize the downtime during a physical or logical standby database switchover, run the following procedure right before initiating the Data Guard switchover to a physical or logical standby database:

   ```
   DECLARE
     role varchar(30);
   BEGIN
     SELECT database_role INTO role FROM v$database;
     IF role = 'PRIMARY' THEN
       dbms_service.stop_service('DBSERV');
     dbms_lock.sleep(2);
     FOR x IN (SELECT s.sid, s.serial#
               FROM v$session s, v$process p
               WHERE s.service_name='DBSERV' AND s.paddr=p.addr)
       LOOP
         BEGIN
           EXECUTE IMMEDIATE
               'ALTER SYSTEM DISCONNECT SESSION
               ''' || x.sid || ',' || x.serial# || ''' IMMEDIATE';
           EXCEPTION WHEN OTHERS THEN
           BEGIN
             DBMS_OUTPUT.PUT_LINE(DBMS_UTILITY.FORMAT_ERROR_STACK);
           END;
         END;
       END LOOP;
     ELSE
       dbms_service.start_service('DBSERV');
     END IF;
   END;
   ```

   This procedure should be performed first on the physical or logical standby database, and then on the primary database, right before the switchover process. Before running the procedure for a physical standby database switchover, Active Data Guard must be enabled on the physical standby database.

Before performing a switchover to a logical standby database, stop the Oracle Database service for TimesTen on the primary database and disconnect all sessions connected to that service. Then start the service on the standby database.

At this point, cache applications try to reconnect to the standby database. If a switchover occurs, there is no wait required to migrate the connections from the primary database to the standby database. This eliminates the performance impact on TimesTen and its applications.

Configuring the TimesTen Database to Work with Synchronous Data Guard

Configure TimesTen to receive notification of FAN HA events and to avoid reconnecting to a failed Oracle Database instance. Use the Oracle client shipped with TimesTen.

1. Create an Oracle Net service name that includes all primary and standby hosts in `ADDRESS_LIST`. For example:

   ```
   DBSERV =
   (DESCRIPTION =
     (ADDRESS_LIST =
     (ADDRESS = (PROTOCOL = TCP)(HOST = PRIMARYDB)(PORT = 1521))
     (ADDRESS = (PROTOCOL = TCP)(HOST = STANDBYDB)(PORT = 1521))
   ```

```
  (LOAD_BALANCE = yes)
  )
  (CONNECT_DATA= (SERVICE_NAME=DBSERV))
)
```

2. In the client's `sqlnet.ora` file, set the `SQLNET.OUTBOUND_CONNECT_TIMEOUT` parameter to enable clients to quickly traverse an address list in the event of a failure. For example, if a client attempts to connect to a host that is unavailable, the connection attempt is bounded to the time specified by the `SQLNET.OUTBOUND_CONNECT_TIMEOUT` parameter, after which the client attempts to connect to the next host in the address list. Connection attempts continue for each host in the address list until a connection is made.

   Setting the `SQLNET.OUTBOUND_CONNECT_TIMEOUT` parameter to a value of 3 seconds suffices in most environments. For example, add the following entry to the `sqlnet.ora` file:

   ```
   SQLNET.OUTBOUND_CONNECT_TIMEOUT=3
   ```

11 Using GoldenGate as an Alternative to Native Read-Only Cache Groups

Oracle GoldenGate is Oracle's primary data replication and data exchange technology. GoldenGate supports a multitude of databases as sources for data capture as well as targets for data delivery.

TimesTen can be deployed in several ways, including as an in-memory cache for data that resides in an Oracle database. TimesTen provides functionality to enable it to act as a native cache for Oracle database. This technology supports both read-only and read-write caching.

If your caching use case provides read-only caching, you may prefer to use GoldenGate to refresh data from the backend database to the TimesTen cache, instead of using the TimesTen native cache functionality.

The following sections describe when and how to use GoldenGate as the cache refresh mechanism for TimesTen:

- Considering Factors When Using GoldenGate as the Cache Refresh Mechanism
- Configuring GoldenGate to Provide Cache Refresh Functionality for TimesTen Workflow

Considering Factors When Using GoldenGate as the Cache Refresh Mechanism

There are several reasons to use GoldenGate as the cache refresh mechanism for read-only cache groups.

If you are planning to use GoldenGate as the cache refresh mechanism, consider that:

- GoldenGate cache refresh supports functionality similar to TimesTen static and dynamic read-only cache groups. All other types of cache groups (Asynchronous WriteThrough, Synchronous WriteThrough, User Managed, and so on) are not currently supported to use GoldenGate as the cache refresh mechanism. These types of cache groups must use the TimesTen native caching mechanisms.
- GoldenGate for Oracle TimesTen supports delivery of data to user tables, instead of cache groups. Since GoldenGate uses regular tables instead of cache groups, create your cache tables in TimesTen as regular tables (using the `CREATE TABLE` statement) and not as cache groups (using the `CREATE READONLY CACHE GROUP` statement).
- When using GoldenGate as the cache refresh mechanism, any read-only cached tables in TimesTen are not truly read-only. Applications are not automatically prevented from modifying data in the tables; however, any modifications can be overwritten if GoldenGate refreshes newly modified data into the table from the back-end database. You can mitigate this by having the tables owned by a dedicated user separate from the application users and assigning database privileges to ensure that application users only have read access to the cache tables.
- A best practice is to use a dedicated TimesTen database user for the GoldenGate apply process. Cached tables should be owned by this user and application users should be granted only read (`SELECT`) privileges on the cached tables.

- GoldenGate only refreshes cache tables in TimesTen with modified data. You must perform an initial load of data from the source database into TimesTen. The initial table data load is used to establish data synchronization when instantiating GoldenGate replication.
- You must set the `DatabaseCharacterSet` TimesTen database parameter to the same value as the Oracle Database database character set.
- All GoldenGate connections to the TimesTen database must use a connection that explicitly sets `ConnectionCharacterSet` to the same value as `DatabaseCharacterSet`.

Configuring GoldenGate to Provide Cache Refresh Functionality for TimesTen Workflow

You can set up a caching environment between GoldenGate and TimesTen.

The following versions are used in this procedure and the examples:

- The source database is an Oracle database running a recent release (18c or later).
- The target database is TimesTen release 26.1.1.1.0 or later.
- The GoldenGate release is 23.1 or later. The GoldenGate parallel Replicat process, which can improve replication throughput in some use cases. You can install GoldenGate on the same machine as the source database.

The tasks for setting up a GoldenGate caching environment with TimesTen are as follows:

Task	Description	More Information
Install Oracle database	Install, configure, and prepare the source database. In most scenarios, the source database already exists and contains the tables that you desire to cache.	Installing Oracle AI Database
Install and configure target TimesTen database	In general, deploy the cache on a different host from the source database, which is the host where the application processes run.	Overview of the Installation Process in TimesTen Classic in the *Oracle TimesTen In-Memory Database Installation, Migration, and Upgrade Guide*
Choose on-box or off-box deployment	Decide if you will run the GoldenGate apply processes on the same host as the target TimesTen database (an on-box deployment) or on a different host to the target TimesTen database (an off-box deployment).	Choosing On-Box or Off-Box for Deployment of a GoldenGate Replicat Process
Install and configure TimesTen client instance	If you have chosen an off-box deployment, install a TimesTen client instance on the GoldenGate apply host and configure it to connect to the TimesTen database.	Installing and Configuring a TimesTen Client Instance (for Off-Box Deployments Only)
Install GoldenGate at the source and target databases	Install and prepare the source and target databases for use with GoldenGate.	Installing Oracle GoldenGate in the *Microservices Architecture Documentation*.

Tasks for Setting up a GoldenGate Caching Environment

Task	Description	More Information
Create Oracle database users and tables	Create Oracle users and tables and grant the necessary privileges.	Task 1: Create Oracle Users and Tables
Configure GoldenGate for data extraction	Configure GoldenGate data capture for the source database tables that you wish to cache in TimesTen.	Task 2: Prepare Oracle Database for GoldenGate Extraction
Create TimesTen database users and tables	Create the TimesTen users and tables that you wish to cache from the source database. Grant the necessary privileges.	Task 3: Create TimesTen Users and Tables
Configure GoldenGate for data replication	Configure the GoldenGate apply mechanism (Replicat process) for the TimesTen database tables that correspond to the source database tables	Task 4: Prepare TimesTen Database for GoldenGate Replication
Perform an initial data load	Perform an initial data load to populate the TimesTen cache tables from the corresponding source database tables. This process usually involves some GoldenGate specific actions as well as the actual data loading.	Task 5: Perform the Initial Data Load
Verify that replication is working	Activate GoldenGate continuous real-time replication to provide ongoing data change synchronization from the source database to TimesTen.	Task 6: Verify GoldenGate Replication

Choosing On-Box or Off-Box for Deployment of a GoldenGate Replicat Process

When you deploy GoldenGate for TimesTen, you ultimately instantiate a set of processes that are responsible for receiving all replicated data from the GoldenGate source, storing it in a (local) trail file, reading the replicated data from the trail file and applying it to the target TimesTen database.

On-Box	Off-Box
If you deploy GoldenGate for TimesTen in the same host, VM, container, or pod as the target TimesTen database, then you can use either direct mode or client-server connectivity. This is known as an *on-box* deployment in GoldenGate terms. Generally, direct mode connectivity is preferred and recommended for this scenario	If you deploy GoldenGate for TimesTen in a different host, VM, container, or pod to the target TimesTen database, then you have to use client-server connectivity. In GoldenGate terms this is an *off-box* deployment.
TimesTen direct mode is a local only connectivity method that enables applications to interact with a local (same host) TimesTen database.	TimesTen client-server mode provides regular client-server connectivity through TCP/IP connections.

On-Box	Off-Box
Direct mode connections use a highly efficient mechanism that eliminates inter-process communication, context switches and other overheads. Direct mode delivers the lowest possible data access latency together with high throughput. Use of direct mode is limited to application processes that are executing in one of the following environments: • In the same bare metal host as the TimesTen database. • In the same virtual machine as the TimesTen database. • In the same container as the TimesTen database or, for Kubernetes environments, in a container in the same pod as the TimesTen database container.	In client-server mode, the applications can run anywhere that has suitable network connectivity to the host where TimesTen is running.
Direct mode connections offer better performance with less overhead. Using direct mode connections will significantly increase the complexity if you want high availability when using a combination of TimesTen and GoldenGate configurations. For example, when you combine GoldenGate with either a TimesTen active-standby pair, automated failover and recovery for GoldenGate is significantly more complex compared to an off-box configuration using client-server connections. Host resources (CPU, memory, storage) must be sufficient to accommodate the TimesTen database instance, the TimesTen database, all GoldenGate processes, all associated processing plus any other local processing (such as applications).	Client-server mode potentially offers more flexibility than direct mode, but this flexibility comes at the cost of increased overhead and lower performance due to network latency, additional processing, and so on.

Installing and Configuring a TimesTen Client Instance (for Off-Box Deployments Only)

When using an off-box deployment, you need to prepare the host where GoldenGate for TimesTen will be installed.

1. Install TimesTen and then create a client instance. See Installation of TimesTen Classic on Linux or UNIX in the *Oracle TimesTen In-Memory Database Installation, Migration, and Upgrade Guide*.
2. Add a suitable client DSN to the client instance `sys.odbc.ini` file to enable connections to the target TimesTen database. See Define a DSN for the TimesTen Database.
   In this example, the client DSN is named `cache1` and the host name where the TimesTen database is running is `myttserver.example.com`. The TimesTen server is listening on port 6625 (the default). This hostname must be resolvable on the client system through DNS or `/etc/hosts` and regular TCP connectivity must be functional between the client and server systems. Port 6625 on the server must not be blocked by a firewall. Note the setting for `ConnectionCharacterSet`.

```
[ODBC Data Sources]
cache1=TimesTen 26.1 Client Driver

[cache1]
TTC_SERVER=myttserver.example.com/6625
```

```
TTC_SERVER_DSN=cache1
ConnectionCharacterSet=AL32UTF8
```

Setting up a GoldenGate Caching Environment

This section covers the set of tasks that you need to do to understand the process of setting up a GoldenGate caching environment.

In this example, Oracle GoldenGate for TimesTen is deployed in on-box mode, using direct mode connectivity. For details, see Choosing On-Box or Off-Box for Deployment of a GoldenGate Replicat Process.

Perform the following tasks:

- Task 1: Create Oracle Users and Tables
- Task 2: Prepare Oracle Database for GoldenGate Extraction
- Task 3: Create TimesTen Users and Tables
- Task 4: Prepare TimesTen Database for GoldenGate Replication
- Task 5: Perform the Initial Data Load
- Task 6: Verify GoldenGate Replication

Task 1: Create Oracle Users and Tables

It is important to run procedures to create users and target tables on the Oracle database to use GoldenGate for caching.

1. Before you create users and tables, you need to configure the Oracle database for cache operations. You have to connect to the Oracle database as a database administrator. For details, see Configuring the Oracle Database to Cache Data.

2. Create the Oracle cache administration user, `cacheadmin` (who also owns the cache group), with the password `ttpwd`. This user also serves as a GoldenGate admin user. Create the Oracle application schema user `oratt` with the same password `ttpwd`. The following example creates the Oracle users using the SQL*Plus utility:

```
SQL> create user cacheadmin identified by ttpwd;
User created.
SQL> create user oratt identified by ttpwd;
User created.
```

3. To grant privileges to the Oracle users:

```
SQL> GRANT CREATE SESSION, RESOURCE, ALTER SYSTEM to oratt;
Grant succeeded.
SQL> GRANT SELECT ANY TRANSACTION TO oratt;
Grant succeeded.
SQL> GRANT SELECT ANY DICTIONARY TO oratt;
Grant succeeded.
SQL> GRANT FLASHBACK ANY TABLE TO oratt;
Grant succeeded.
SQL> GRANT UNLIMITED TABLESPACE TO oratt;
Grant succeeded.
SQL> GRANT SELECT ANY TABLE TO cacheadmin;
Grant succeeded.
SQL> GRANT INSERT ANY TABLE TO cacheadmin;
Grant succeeded.
SQL> GRANT UPDATE ANY TABLE TO cacheadmin;
Grant succeeded.
```

```
SQL> GRANT DELETE ANY TABLE TO cacheadmin;
Grant succeeded.
```

Now, run the `grantCacheAdminPrivileges.sql` script to grant privileges to the cache administration user `cacheadmin` that are required to create Oracle database objects. See The grantCacheAdminPrivileges.sql Script.

4. Create tables and insert rows into them. The `oratt` user owns three tables. The two tables `customer` and `orders` are cached in TimesTen.

```
SQL> CREATE TABLE customer
(
    custid     VARCHAR2(10) NOT NULL,
    firstname  VARCHAR2(20) NOT NULL,
    lastname   VARCHAR2(20) NOT NULL,
    address    VARCHAR2(128) NOT NULL,
    phone      VARCHAR2(16) NOT NULL,
    PRIMARY KEY (custid)
);
Table created.

SQL> CREATE TABLE orders
(
    orderid    NUMBER(10,0) NOT NULL,
    custid     VARCHAR2(10) NOT NULL,
    orderdate  DATE NOT NULL,
    priority   CHAR(1),
    amount     NUMBER(12,2) NOT NULL,
    PRIMARY KEY (orderid),
    FOREIGN KEY (custid) REFERENCES customer(custid)
);
Table created.

SQL> CREATE TABLE item
(
    itemno     NUMBER(4,0) NOT NULL,
    orderid    NUMBER(10,0) NOT NULL,
    itemcode   VARCHAR2(10) NOT NULL,
    quantity   NUMBER(4,0) NOT NULL,
    price      NUMBER(6,2) NOT NULL,
    totalvalue NUMBER(10,2) NOT NULL,
    PRIMARY KEY (orderid,itemno),
    FOREIGN KEY (orderid) REFERENCES order(orderid)
);
Table created.

SQL> INSERT INTO customer VALUES('C000000001', 'Fred', 'Bloggs', 'Nice Villas,
  Pleasant Town', '+16072321234');
1 row inserted.
SQL> INSERT INTO orders VALUES(123456, 'C000000001', '21/10/2021', 'N', 430.46);
1 row inserted.
SQL> INSERT INTO item VALUES(1, 123456, 'I000001725', 2, 15.25, 30.50);
1 row inserted.
SQL> INSERT INTO item VALUES(2, 123456, 'I000207351', 4, 99.99, 399.96);
1 row inserted.
COMMIT;
```

Task 2: Prepare Oracle Database for GoldenGate Extraction

You can follow several steps to prepare the Oracle database for GoldenGate extraction.

Note

After installing GoldenGate, assign the necessary user privileges that are required for connecting to Oracle database from Oracle GoldenGate. For details, see Oracle in the *Microservices Architecture Documentation*.

To prepare your database for GoldenGate extraction:

1. Use the Oracle GoldenGate Configuration Assistant (OGGCA) utility to create deployments and the Service Manager process on a host machine. Add a deployment using the OGGCA wizard and then create a Service Manager. For details, see Add a Deployment in the *Microservices Architecture Documentation*.
2. Start the Service Manager. See Start and Stop the Service Manager in the *Microservices Architecture Documentation*.
3. Start the Admin Client and then connect to your deployment, see About Admin Client in the *Microservices Architecture Documentation*.
4. The Extract process is the extraction or the data capture mechanism of Oracle GoldenGate. Before you add an Extract, ensure that the following settings are configured:
   - Add database connections
   - Add SCHEMATRANDATA
   - Register an Extract

   For details, see Add an Online Extract in the *Microservices Architecture Documentation*.
5. Create a parameter file for an Extract and edit it (see Create a Parameter File for Extract in the *Microservices Architecture Documentation*). In this example, the FQDN (Fully Qualified Domain Name) of the system hosting the TimesTen database is `tthost1.example.com` (see Set the Net Service Name for the Oracle Database in the tnsnames.ora File). Edit the parameter file with the following content for the GoldenGate Extract process:

   ```
   EXTRACT tt
   USERID cacheadmin, PASSWORD ttpwd
   RMTHOST tthost1.example.com, MGRPORT 7809
   RMTTRAIL dirdat/tr
   TABLE oratt.customer
   TABLE oratt.orders
   TABLE oratt.item
   ```
6. Before you start an Extract process, determine the current SCN value (see Task 5: Perform the Initial Data Load). Now, start the Extract process. For details, see Start or Stop Extract in the *Microservices Architecture Documentation*.

Task 3: Create TimesTen Users and Tables

Perform procedures to create users and the target tables on the TimesTen that support GoldenGate replication.

1. Before you create users and tables, you need to configure the TimesTen database for cache operations. For details, see Configuring a TimesTen Database to Cache Oracle Database Data.
2. Create the TimesTen database cache administration user `cacheadmin` whose name will be same as the Oracle database cache administration user and it will also be a GoldenGate user.

To create these users in your TimesTen database, connect, using the `ttIsql` utility, to the TimesTen database as the instance administrator user and execute:

```
Command>CREATE USER cacheadmin IDENTIFIED BY ttpwd;
User created.
```

3. To create a cache table user `oratt` whose name is same as the Oracle database schema user of the tables to be cached in the TimesTen database:

   ```
   Command>CREATE USER oratt IDENTIFIED BY ttpwd;
   User created.
   ```

4. Grant privileges to the TimesTen users:

   ```
   Command>GRANT CREATE SESSION, CACHE_MANAGER, CREATE ANY TABLE TO
   cacheadmin;
   Command>GRANT CREATE SESSION, CREATE ANY TABLE TO oratt;
   ```

5. Create the target tables in the TimesTen database. Make the tables owned by the user `cacheadmin`. Connect to the database, using ttIsql, as the user `cacheadmin`:

   ```
   $ ttIsql -connStr "DSN=cache1;UID=cacheadmin;PWD=ttpwd"
   ```

6. Run the following SQL statements to create the tables on TimesTen database:

   ```
   Command>CREATE TABLE customer
   (
       custid     VARCHAR2(10) NOT NULL,
       firstname  VARCHAR2(20) NOT NULL,
       lastname   VARCHAR2(20) NOT NULL,
       address    VARCHAR2(128) NOT NULL,
       phone      VARCHAR2(16) NOT NULL,
       PRIMARY KEY (custid)
   );
   Table created.
   Command>CREATE TABLE orders
   (
       orderid    NUMBER(10,0) NOT NULL,
       custid     VARCHAR2(10) NOT NULL,
       orderdate  DATE NOT NULL,
       priority   CHAR(1),
       amount     NUMBER(12,2) NOT NULL,
       PRIMARY KEY (orderid),
       FOREIGN KEY (custid) REFERENCES customer(custid)
   );
   Table created.
   Command>CREATE TABLE item
   (
       itemno     NUMBER(4,0) NOT NULL,
       orderid    NUMBER(10,0) NOT NULL,
       itemcode   VARCHAR2(10) NOT NULL,
       quantity   NUMBER(4,0) NOT NULL,
       price      NUMBER(6,2) NOT NULL,
       totalvalue NUMBER(10,2) NOT NULL,
       PRIMARY KEY (orderid,itemno),
       FOREIGN KEY (orderid) REFERENCES order(orderid)
   );
   Table created.
   quit;
   ```

Task 4: Prepare TimesTen Database for GoldenGate Replication

To prepare the TimesTen database for GoldenGate replication, few steps are required to ensure smooth and efficient data transfer between the source and target systems.

> **Note**
>
> After installing GoldenGate, assign the necessary user privileges that are required for connecting to TimesTen database from Oracle GoldenGate to prepare the database for replication. For details, see TimesTen in the *Microservices Architecture Documentation*.

To prepare your database for GoldenGate replication:

1. Use the Oracle GoldenGate Configuration Assistant (OGGCA) utility to create deployments and the Service Manager process on your host machine. Add a deployment using the OGGCA wizard and then create a Service Manager. For details, see Add a Deployment in the *Microservices Architecture Documentation*.
2. Start the Service Manager. See Start and Stop the Service Manager in the *Microservices Architecture Documentation*.
3. Start the Admin Client and then connect to your deployment, see About Admin Client in the *Microservices Architecture Documentation*.
4. The Replicat is a process that delivers data to a target system. Before you begin adding a Replicat, make sure that the following settings are configured:
   - Add database connections
   - Add SCHEMATRANDATA

   For details, see Add a Replicat in the *Microservices Architecture Documentation*
5. Create a parameter file for a Replicat and edit it (see Configure a Replicat Parameter File in the *Microservices Architecture Documentation*). In this example, edit the parameter file with the following content for setting up a Replicat:

   ```
   REPLICAT rep
   TARGETDB cache1, USERID cacheadmin, PASSWORD ttpwd
   MAP oratt.*, TARGET oratt.*
   ```

   Here `oratt.*` is the table owner and name on the source database, and `oratt.*` is the table owner and name in the TimesTen database. The table names should be same in both the source and the target databases. You can specify multiple MAP directives or use wildcards for multiple tables.
6. Before you start a Replicat process, determine the current SCN value (see Task 5: Perform the Initial Data Load). Now, start a Replicat process and specify the SCN, see Stop, Start a Replicat in the *Microservices Architecture Documentation*.

Task 5: Perform the Initial Data Load

Before starting the cache operations is to perform an initial data load of what is currently in the tables that are to be cached.

1. On the host with the Oracle database, determine the current SCN value (using SQL*Plus):

```
SQL> SELECT CURRENT_SCN FROM V$DATABASE;

CURRENT_SCN
-----------
    2791297
```

2. On the host with the TimesTen database, connect to the TimesTen database as the `cacheadmin` user specifying both the TimesTen and Oracle database passwords for this user:

```
$ ttIsql -connStr "DSN=cache1;UID=cacheadmin;PWD=ttpwd;OraclePWD=ttpwd"
```

3. Load the data for each of the tables based on the Oracle database SCN value determined above:

```
Command>call ttLoadFromOracle('oratt', 'customer',
      'SELECT * FROM oratt.customer AS OF scn 2791297');
Command>call ttLoadFromOracle('oratt', 'orders', 'SELECT * FROM oratt.orders
      AS OF SCN 2791297');
Command>call ttLoadFromOracle('oratt', 'item', 'SELECT * FROM oratt.item
      AS OF SCN 2791297');
```

4. Update the optimizer statistics for the tables that you just loaded to ensure optimal query plans in TimesTen:

```
Command> statsupdate customer;
Command> statsupdate orders;
Command> statsupdate item;
```

Task 6: Verify GoldenGate Replication

Once you have replication set up, verify that replication is working.

Update data in the replicated tables by inserting, updating, and/or deleting rows in the Oracle database. For example, you add a new row to the `customer` table in the Oracle database:

```
SQL> INSERT INTO customer VALUES('C000000002', 'John', 'Beavans', 'Pretty Villas Down
Town', '+16072321256');
```

On the TimesTen database, select from the replicated tables and verify that the changes are being propagated from the Oracle database. In this example, it shows the same row is propagated to the `customer` table in the TimesTen database.

```
Command> select * from customer;
<C000000001, Fred, Bloggs, Nice Villas Pleasant Town, +16072321234>
…..
….
…..
<C000000002, John, Beavans, Pretty Villas Down Town, +16072321256>
```

12

Configuring GoldenGate for Log-Based Cache Autorefresh on TimesTen

Oracle GoldenGate allows you to replicate Oracle data into TimesTen static and dynamic log-based read-only cache groups.

The following sections explain how to create a log-based read-only cache group using GoldenGate and TimesTen replication:

- About Log-Based and Trigger-Based Read-Only Cache Groups
- Before You Begin
- Creating a Log-Based Read-Only Cache with GoldenGate
- Managing Timeout Issues in GoldenGate
- Scenarios for Log-Based Read-Only Cache Groups

About Log-Based and Trigger-Based Read-Only Cache Groups

Log-based read-only cache groups provide an alternative approach to trigger-based read-only cache groups by leveraging redo logs to synchronize data between Oracle and TimesTen. This method is ideal for both static and dynamic cache groups, where the data is continuously refreshed and synchronized from Oracle. It enables automatic refresh and asynchronous data propagation from Oracle to TimesTen by reading changes from the Oracle logs. Since it uses redo logs, it reduces the load on the Oracle database. It is ideal for scenarios when there is a high volume of data and the data is used frequently. This cache group is perfect for high-performance read applications that need real-time access to Oracle data with minimal delay and overhead.

A trigger-based read-only cache group synchronizes data changes between the Oracle database and the TimesTen cache using database triggers instead of redo logs. Oracle triggers capture changes such as inserts, updates, and deletes on the source tables, which are then propagated to TimesTen. This lowers DML latency in propagating changes to the TimesTen cache. These triggers activate immediately when data changes occur, enabling caching to detect and respond. This method adds overhead to transactions, making it harder to maintain.

Before You Begin

Install the following software:

1. Install and configure the Oracle database 18c or later. See Installing Oracle AI Database in the *Database Installation Guide for Linux*.
2. Install and configure a TimesTen instance release 26.1.1.1.0 or later. See Overview of the Installation Process in TimesTen Classic in the *Oracle TimesTen In-Memory Database Installation, Migration, and Upgrade Guide*.
3. Install GoldenGate with Replicat release 23.10.3.25.11 or later. See Installing Oracle GoldenGate in the *Microservices Architecture Documentation*.

Creating a Log-Based Read-Only Cache with GoldenGate

This section details the tasks to set up a log-based static and dynamic read-only cache in TimesTen with GoldenGate.

In this example, Oracle GoldenGate for TimesTen is deployed in on-box mode, using direct mode connectivity. For details, see Choosing On-Box or Off-Box for Deployment of a GoldenGate Replicat Process.

Perform the following tasks:

- Task 1: Create Oracle Users and Tables
- Task 2: Prepare Oracle Database for GoldenGate Extraction
- Task 3: Create TimesTen Users and Tables
- Task 4: Prepare TimesTen Database for GoldenGate Replication
- Task 5: Verify GoldenGate Replication

Task 1: Create Oracle Users and Tables

Follow these steps to create users and target tables on the Oracle database to support GoldenGate replication.

1. Configure the Oracle database for cache operations. You have to connect to the Oracle database as a database administrator. For details, see Configuring the Oracle Database to Cache Data.
2. Create the Oracle cache administration user, `cacheadmin` (who also owns the cache group), with the password `ttpwd`. This user also serves as a GoldenGate admin user. Create the Oracle application schema user `oratt` with the same password `ttpwd`. The following example creates the Oracle users using the SQL*Plus utility:

   ```
   SQL> create user cacheadmin identified by ttpwd;
   User created.
   SQL> create user oratt identified by ttpwd;
   User created.
   ```

3. To grant privileges to the Oracle users:

   ```
   SQL> GRANT CREATE SESSION, RESOURCE, ALTER SYSTEM to oratt;
   Grant succeeded.
   SQL> GRANT SELECT ANY TRANSACTION TO oratt;
   Grant succeeded.
   SQL> GRANT SELECT ANY DICTIONARY TO oratt;
   Grant succeeded.
   SQL> GRANT FLASHBACK ANY TABLE TO oratt;
   Grant succeeded.
   SQL> GRANT UNLIMITED TABLESPACE TO oratt;
   Grant succeeded.
   SQL> GRANT SELECT ANY TABLE TO cacheadmin;
   Grant succeeded.
   SQL> GRANT INSERT ANY TABLE TO cacheadmin;
   Grant succeeded.
   SQL> GRANT UPDATE ANY TABLE TO cacheadmin;
   Grant succeeded.
   SQL> GRANT DELETE ANY TABLE TO cacheadmin;
   Grant succeeded.
   ```

Now, run the `grantCacheAdminPrivileges.sql` script to grant privileges to the cache administration user `cacheadmin` that are required to create Oracle database objects. See The grantCacheAdminPrivileges.sql Script.

4. Create tables and insert rows into them. The `oratt` user owns two tables. This example shows two tables `customer` and `orders` that are cached in TimesTen.

```
SQL> CREATE TABLE customer
(
    custid     VARCHAR2(10) NOT NULL,
    firstname  VARCHAR2(20) NOT NULL,
    lastname   VARCHAR2(20) NOT NULL,
    address    VARCHAR2(128) NOT NULL,
    phone      VARCHAR2(16) NOT NULL,
    PRIMARY KEY (custid)
);
Table created.

SQL> CREATE TABLE orders
(
    orderid    NUMBER(10,0) NOT NULL,
    custid     VARCHAR2(10) NOT NULL,
    orderdate  DATE NOT NULL,
    priority   CHAR(1),
    amount     NUMBER(12,2) NOT NULL,
    PRIMARY KEY (orderid),
    FOREIGN KEY (custid) REFERENCES customer(custid)
);
Table created.

SQL> INSERT INTO customer VALUES('C000000001', 'Fred', 'Bloggs', 'Nice Villas,
  Pleasant Town', '+16072321234');
1 row inserted.
SQL> INSERT INTO customer VALUES('C000000002', 'Johnny', 'Lever', 'Sam Villas,
  Green Town', '+16072324445');
1 row inserted.
SQL> INSERT INTO orders VALUES(123456, 'C000000001', '21/10/2024', 'N', 430.46);
1 row inserted.
SQL> INSERT INTO orders VALUES(789123, 'C000000002', '21/10/2025', 'N', 156.25);
1 row inserted.
COMMIT;
```

Task 2: Prepare Oracle Database for GoldenGate Extraction

Prepare the Oracle database for GoldenGate extraction.

Note

After installing GoldenGate, assign the necessary user privileges that are required for connecting to Oracle database from Oracle GoldenGate. For details, see Oracle in the *Microservices Architecture Documentation*.

To prepare your database for GoldenGate extraction:

1. Use the Oracle GoldenGate Configuration Assistant (OGGCA) utility to create deployments and the Service Manager process on a host machine. Add a deployment using the OGGCA wizard and then create a Service Manager. For details, see Add a Deployment in the *Microservices Architecture Documentation*.

2. Start the Service Manager. See Start and Stop the Service Manager in the *Microservices Architecture Documentation*.
3. Start the Admin Client and then connect to your deployment, see About Admin Client in the *Microservices Architecture Documentation*.
4. The Extract process is the extraction or the data capture mechanism of Oracle GoldenGate. Before you add an Extract, ensure that the following settings are configured:
   - Add database connections
   - Add SCHEMATRANDATA
   - Register an Extract

   For details, see Add an Online Extract in the *Microservices Architecture Documentation*.
5. While adding an Extract, edit a parameter file (see Create a Parameter File for Extract in the *Microservices Architecture Documentation*). In this example, the FQDN (Fully Qualified Domain Name) of the system hosting the TimesTen database is `tthost1.example.com` (see Set the Net Service Name for the Oracle Database in the tnsnames.ora File). This example shows how to edit the parameter file with the following content for the GoldenGate Extract process:

   ```
   EXTRACT tt
   USERID cacheadmin, PASSWORD ttpwd
   RMTHOST tthost1.example.com, MGRPORT 7809
   RMTTRAIL dirdat/tr
   TABLE oratt.customer
   TABLE oratt.orders
   ```
6. Now, start the Extract process. See Start or Stop Extract in the *Microservices Architecture Documentation*.

Task 3: Create TimesTen Users and Tables

You can carry out procedures to create users and target tables in TimesTen to enable support for GoldenGate replication. Performing these procedures prepares TimesTen to securely and efficiently handle data replicated by GoldenGate, ensuring smooth and reliable synchronization between systems.

1. Configure TimesTen as a cache for an Oracle Database. You must define the database connection settings. For details, see Specify Database Connection Definition for Cache.
2. Create the TimesTen database cache administration user `cacheadmin`, whose name in this example matches the Oracle cache administration user. This user also serves as a GoldenGate administration user.
   Create a cache table user `oratt`, whose name matches the Oracle Database schema owner of the tables to be cached in the TimesTen database.

   The following example creates the TimesTen users. It uses the `ttIsql` utility to connect to the `cache1` DSN as the instance administrator:

   ```
   Command> CREATE USER cacheadmin IDENTIFIED BY ttpwd;
   User created.
   Command> CREATE USER oratt IDENTIFIED BY ttpwd;
   User created.
   ```
3. Assign privileges to the TimesTen users (see Grant Privileges to the TimesTen Users):

   ```
   Command> GRANT CREATE SESSION, CACHE_MANAGER, CREATE ANY TABLE TO cacheadmin;
   Command> GRANT CREATE SESSION, CREATE ANY TABLE TO oratt;
   ```

4. Two tables `customer` and `orders` are created in the Oracle database. In TimesTen, create a log-based, read-only cache group for the `customer` and `orders` tables.

   a. To create a log-based cache group for the `customer` and `orders` tables, connect as `cacheadmin` user:

      ```
      $ ttIsql -connStr "DSN=cache1;UID=cacheadmin;PWD=ttpwd;OraclePwd=ttpwd”
      Command> call ttcacheuidpwdset('cacheadmin','ttpwd');
      Command> call ttcachestart();
      Command> CREATE DYNAMIC READONLY CACHE GROUP customer_orders
      FROM oratt.customer
       (cust_num NUMBER(6) NOT NULL,
        region   VARCHAR2(10),
        name     VARCHAR2(50),
        address  VARCHAR2(100),
        PRIMARY KEY(cust_num)),
      oratt.orders
       (ord_num      NUMBER(10) NOT NULL,
        cust_num     NUMBER(6) NOT NULL,
        when_placed  DATE NOT NULL,
        when_shipped DATE NOT NULL,
        PRIMARY KEY(ord_num),
        FOREIGN KEY(cust_num) REFERENCES oratt.customer(cust_num))method log;
      1 cachegroup created.
      ```

      Cache groups can only be created with autorefresh state paused. In this state, all operations from GoldenGate wait until the autorefresh state is set to on and ensure that GoldenGate replicat can begin at the lowest load SCN for all of your cache groups, provided that you have the same initial data on both sides (Oracle and TimesTen). Next, you configure GoldenGate to capture data from Oracle database and apply it to TimesTen. See Task 4: Prepare TimesTen Database for GoldenGate Replication.

   b. GoldenGate refreshes cache tables in TimesTen with modified data. Thus, before starting a GoldenGate Replicat process for continuous replication, you need to perform an initial load of data to populate the cached tables in the TimesTen database with the rows from the source database tables. The cache group initial load is performed.

      ```
      Command> LOAD CACHE GROUP customer_orders COMMIT EVERY 1000 ROWS;
      200000 cache instances affected.
      ```

      TimesTen uses the current SCN from Oracle database when performing a cache group load. This SCN is important because it tells you exactly when the data was loaded from the Oracle database. Once the load completes, the autorefresh state will be set to on, allowing GoldenGate operations on the cache group's tables to proceed provided their SCN is greater than the SCN used for the initial load. This behavior applies to both static and dynamic read-only cache groups.

   c. Before starting the Replicat process, get the initial load SCN:

      ```
      Command> cachegroups;
        Cache Group cacheadmin.customer_orders:
        Cache Group Type: Log-Based Read
      Only
        Autorefresh:
      Yes
        Autorefresh State:
      On
        Load SCN:
      1827747
        Aging: No aging defined
        Root Table: oratt.customer
      ```

```
  Child Table: oratt.orders
  Table Type: Read Only
```

An alternative method to retrieve the SCN is by using the built-in procedure `ttCacheLoadSCNGet` or a system view (see SYS.V$CACHE_GROUP_LOAD_STATUS):

```
Command> CALL ttCacheLoadSCNGet('cacheadmin', 'customer_orders');
Load_SCN
-----------
1827747
```

d. Before you initiate Replicat at the specified SCN, you need to prepare TimesTen database for the GoldenGate replication. For details, see Task 4: Prepare TimesTen Database for GoldenGate Replication.

Task 4: Prepare TimesTen Database for GoldenGate Replication

To prepare the TimesTen database for GoldenGate replication, few steps are required to ensure smooth and efficient data transfer between the source and target systems.

Note

After installing GoldenGate, assign the necessary user privileges that are required for connecting to TimesTen database from Oracle GoldenGate to prepare the database for replication. For details, see TimesTen in the *Microservices Architecture Documentation*.

To prepare your database for GoldenGate replication:

1. Use the Oracle GoldenGate Configuration Assistant (OGGCA) utility to create deployments and the Service Manager process on your host machine. Add a deployment using the OGGCA wizard and then create a Service Manager. For details, see Add a Deployment in the *Microservices Architecture Documentation*.
2. Start the Service Manager. See Start and Stop the Service Manager in the *Microservices Architecture Documentation*.
3. Start the Admin Client and then connect to your deployment, see About Admin Client in the *Microservices Architecture Documentation*.
4. The Replicat is a process that delivers data to a target system. Before you begin adding a Replicat, make sure that the following settings are configured:
   - Add database connections
   - Add SCHEMATRANDATA

   For details, see Add a Replicat in the *Microservices Architecture Documentation*
5. Create a parameter file for a Replicat and edit it (see Configure a Replicat Parameter File in the *Microservices Architecture Documentation*). This example shows how to edit the parameter file with the following content for setting up a Replicat:

   ```
   REPLICAT rep
   TARGETDB cache1, USERID cacheadmin, PASSWORD ttpwd
   MAP oratt.*, TARGET oratt.*
   ```

   Here `oratt.*` is the table owner and name on the source database, and `oratt.*` is the table owner and name in the TimesTen database. The table names should be same in both the source and the target databases to avoid errors. You can specify multiple MAP directives or use wildcards for multiple tables.

6. Start the Replicat process from the specified SCN.

> **Note**
>
> When multiple log-based read-only cache groups exist, you must use the lowest SCN across all of them.

For example, to get the initial load SCN, you can either run the `cachegroups` command (see Step3c to Step3e in) or the system view command on your TimesTen that shows the current status of cache group loading operations:

```
Command> describe sys.v$cache_group_load_status;
View SYS.V$CACHE_GROUP_LOAD_STATUS:
Columns:
CACHE_GROUP_OWNER TT_CHAR (31) NOT NULL
CACHE_GROUP_NAME TT_CHAR (31) NOT NULL
MINIMUMLOADSCN CHAR (20)

Command> select * from sys.v$cache_group_load_status order by customer_orders;
< cacheadmin , customer_orders , 1827747 >
1 row found.
```

To start a Replicat process starting from the SCN, see Stop, Start a Replicat in the *Microservices Architecture Documentation*.

Task 5: Verify GoldenGate Replication

Once you have set up replication, verify that it is working.

Update data in the replicated tables by inserting, updating, and/or deleting rows in the Oracle database. For example, you add a new row to the `customer` table in the Oracle database:

```
SQL> INSERT INTO customer VALUES('C000000003', 'Sara', 'Beavans', 'Pretty Villas Down
Town', '+16072329988');
```

On the TimesTen database, select from the replicated tables and verify that the changes are being propagated from the Oracle database. In this example, it shows the same row is propagated to the `customer` table in the TimesTen database.

```
Command> select * from customer;
<C000000001, Fred, Bloggs, Nice Villas Pleasant Town, +16072321234>
…..
…..
<C000000003, Sara, Beavans, Pretty Villas Down Town, +16072329988>
```

Managing Timeout Issues in GoldenGate

GoldenGate operations may experience a load wait situation when propagating changes into a log-based cache group, which could lead to a timeout.

The timeout can happen under the following conditions:

- The cache group is in a paused autorefresh state and changes are being propagated into it.

  - Log-based cache groups must be loaded before they can be propagated into. These waits and log messages are designed to notify users of the issue.
  - The daemon log shows the following message:
    Autorefresh was not able to acquire lock on one of the cache groups.
- The operation being propagated by GoldenGate is more recent than the ongoing dynamic loads.
  - This wait ensures cache instance consistency and is necessary for correctness.
  - The daemon log displays the following message with the details of the affected table and SCNs:
    GoldenGate connection waiting for pending dynamic loads.

In both cases, the daemon log periodically issues warnings about long waits. Additionally, the `loadWait` statistic increments each time there is a wait when propagating a change to a parent table row.

Similar to other long-running operations, these can eventually time out. If the timeout errors are not managed by the Replicat, it may cause the Replicat to terminate and result in error TT5988, "*GoldenGate autorefresh operation has exceeded the timeout_desc timeout [timeout_secs secs].info*" .

The timeout will trigger based on the lowest defined connection-level timeout between LockWait (lock timeout) and SQLQueryTimeout. For details, see LockWait and SQLQueryTimeout in the *Oracle TimesTen In-Memory Database Reference*.

Scenarios for Log-Based Read-Only Cache Groups

This section explains various scenarios for log-based read-only cache groups.

The scenarios are:

- Log-Based Read-Only Cache Groups with TimesTen Replication
- Log-Based Read-Only Cache Groups with Active Data Guard

Log-Based Read-Only Cache Groups with TimesTen Replication

When log-based read-only cache groups are replicated by an active standby pair, the cache group's System Change Number (SCN) must also be replicated between the active and standby databases.

TimesTen active standby pair replication is configured to provide high availability, data recovery, and scalability in this architecture of read-only cache groups. This replication is similar to the trigger-based read-only cache groups. Failover scenarios differ based on the GoldenGate replicat’s connection to TimesTen.

When you decide to use replication, it's important to carefully consider whether to select direct or client/server mode for the GoldenGate replicat connection to TimesTen. A direct connection provides better performance but necessitates manual intervention during replication failover. On the other hand, a client/server connection is slower than direct but includes automatic failover capabilities in case of failure.

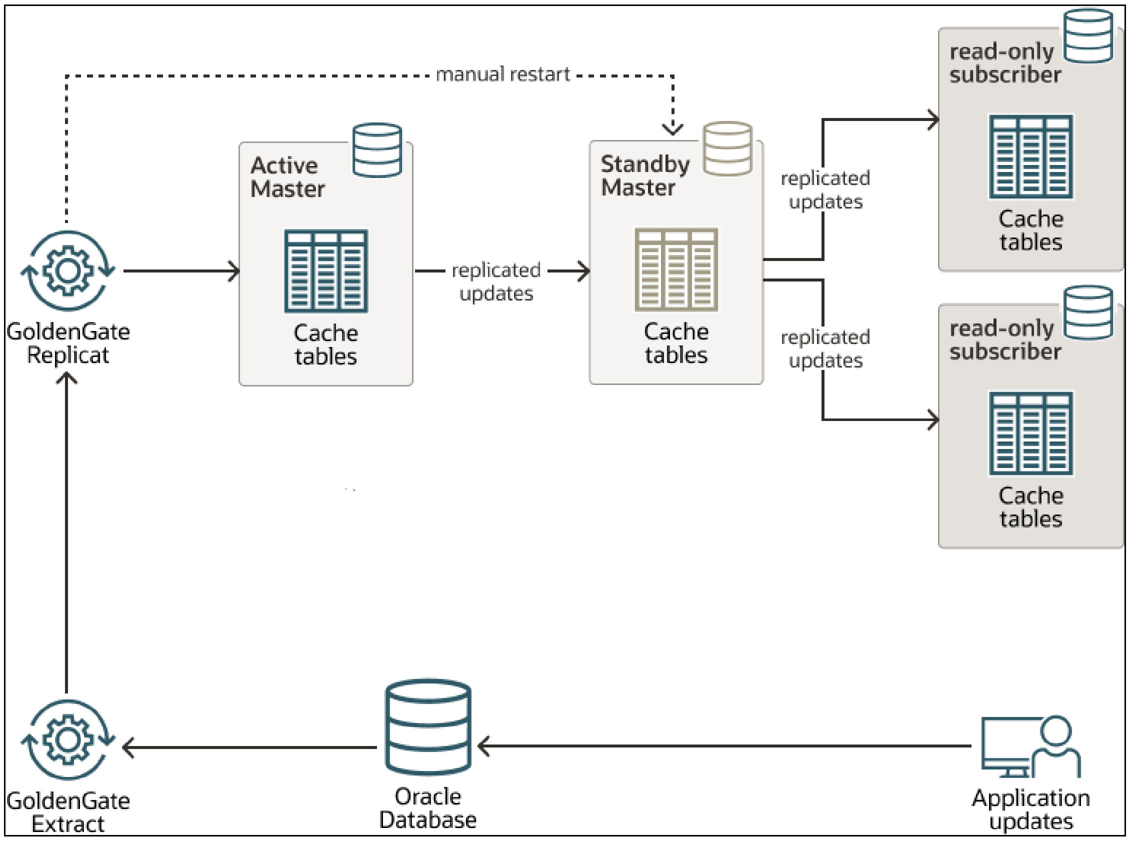
There are two types of connection modes:

- Direct
- Client/Server

For details, see Choosing On-Box or Off-Box for Deployment of a GoldenGate Replicat Process.

TimesTen with GoldenGate in Direct Mode

In this configuration, the GoldenGate Replicat process has a direct connection with TimesTen. In direct mode, if the Active Master fails, GoldenGate Replicat does not automatically fail over to the Standby Master.



The image depicts a data replication architecture utilizing Oracle GoldenGate components, namely GoldenGate Extract and GoldenGate Replicat. TimesTen replication ensures high availability, data recovery, and scalability in this architecture.

At the core of the system is the Oracle database, where end users or applications perform regular application updates (such as insert, update, and delete operations). To ensure these changes are propagated to other systems for consistency and redundancy, the GoldenGate Extract component continuously captures the changes from the Oracle database. This captured data is then passed on to the GoldenGate Replicat process.

GoldenGate Replicat applies these extracted changes to the Active Master, which holds the primary replicated version of the data in cache tables for fast access. This setup does not support automatic failover. When the Active Master goes down, the direct connection from GoldenGate Replicat does not automatically failover to the Standby Master. The GoldenGate Replicat abends in case of failure, such as when the host running the TimesTen database is powered off.

The GoldenGate connection must be then switched manually to the Standby Master, after which the Replicat process needs to be restarted from the SCN where it previously abended.

See Switching from Nonintegrated Replicat to Parallel Nonintegrated Replicat in the *Microservices Architecture Documentation*.

Note

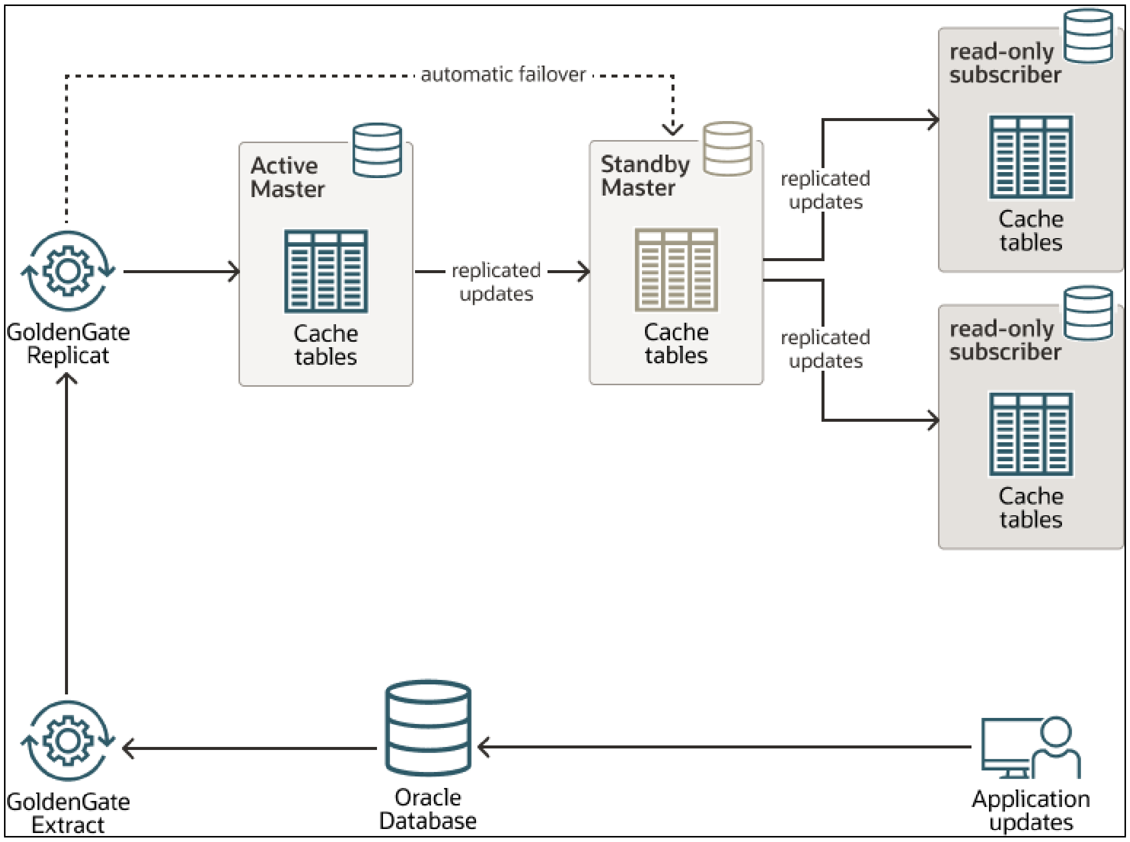
It is recommended to store the trail files generated by the GoldenGate capture process on a shared disk, enabling the Replicat process to access them and resume replication from the SCN where it previously abended.

This architecture provides a robust and scalable solution for data replication and high availability, ensuring that business applications can continue to operate without interruption even in the event of failures.

TimesTen with GoldenGate in Client/Server Mode

In this configuration, the GoldenGate Replicat process has client/server connection with TimesTen.

In client/server mode, TimesTen automatically fails over GoldenGate Replicat connections to the standby master if the active master fails.



> **Note**
>
> It is recommended to store the trail files generated by the GoldenGate capture process on a shared disk, enabling the Replicat process to access them and resume replication from the SCN where it previously abended.

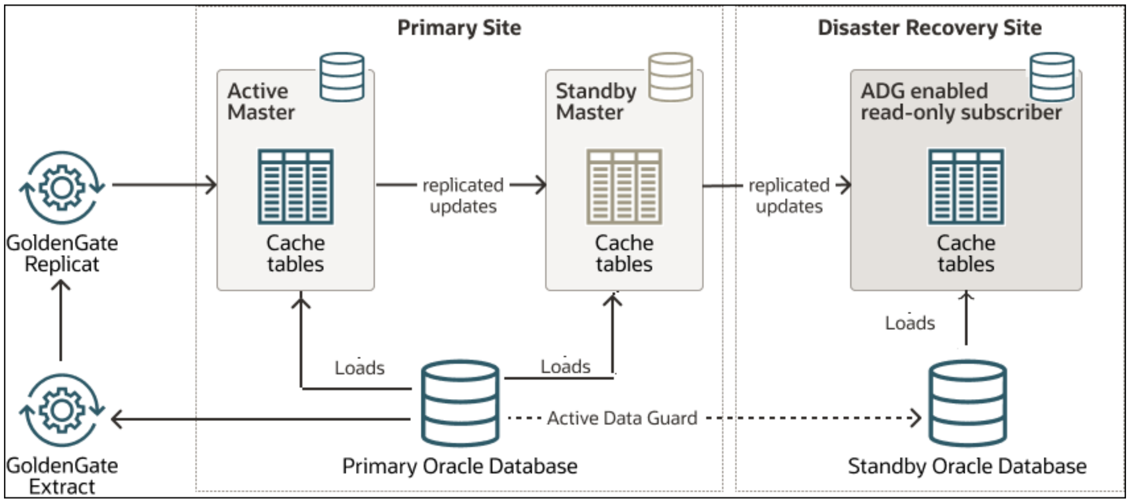
Log-Based Read-Only Cache Groups with Active Data Guard

When using Active Data Guard (ADG) with the Oracle database, specify the `StandbyNetServiceName`.

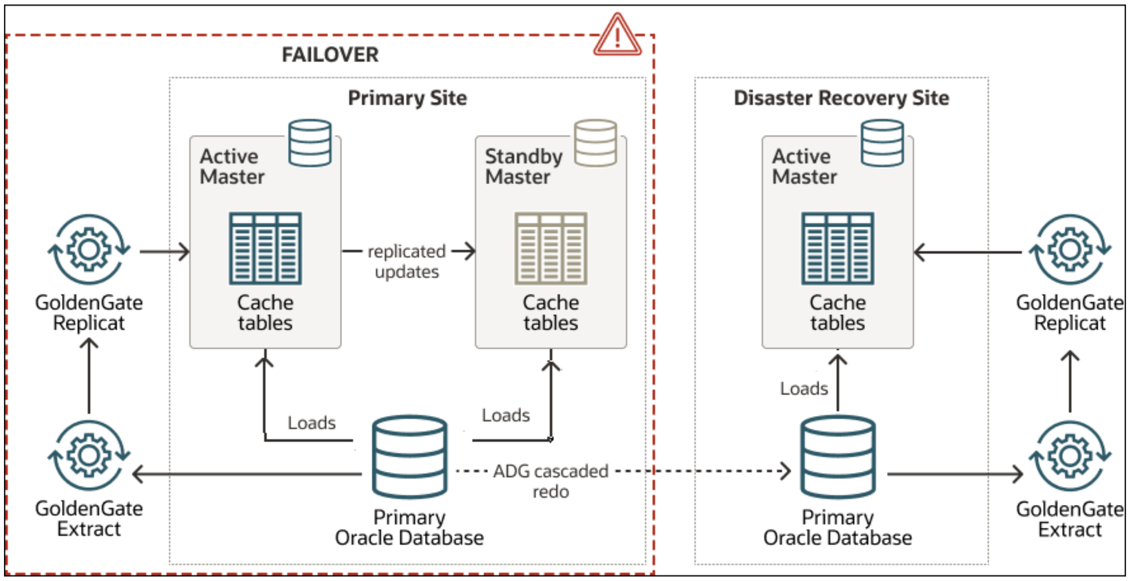
The current SCN is taken from the `StandbyNetServiceName`. Manual and dynamic loads run on the primary database i.e Oracle database which is specified in the `OracleNetServiceName`. See Using Cache with Data Guard.

Physical Standbys with GoldenGate Extract on the Primary Database

In this configuration, the GoldenGate Extract process reads directly from the primary database, resulting in an impact similar to that of a non-ADG setup. However, its effect on the primary database's workload performance remains lower than that of the default trigger-based cache mechanism.



With the recommended configuration, if the primary site goes down, the primary Oracle database fails over to the standby Oracle database. To recover the read-only subscriber cache database in this disaster recovery scenario, you have to drop the active standby pair configuration from the read-only subscriber database. During failover, the cache functions as it does in a non-ADG configuration, meaning the standby Oracle database is promoted to the new primary database.

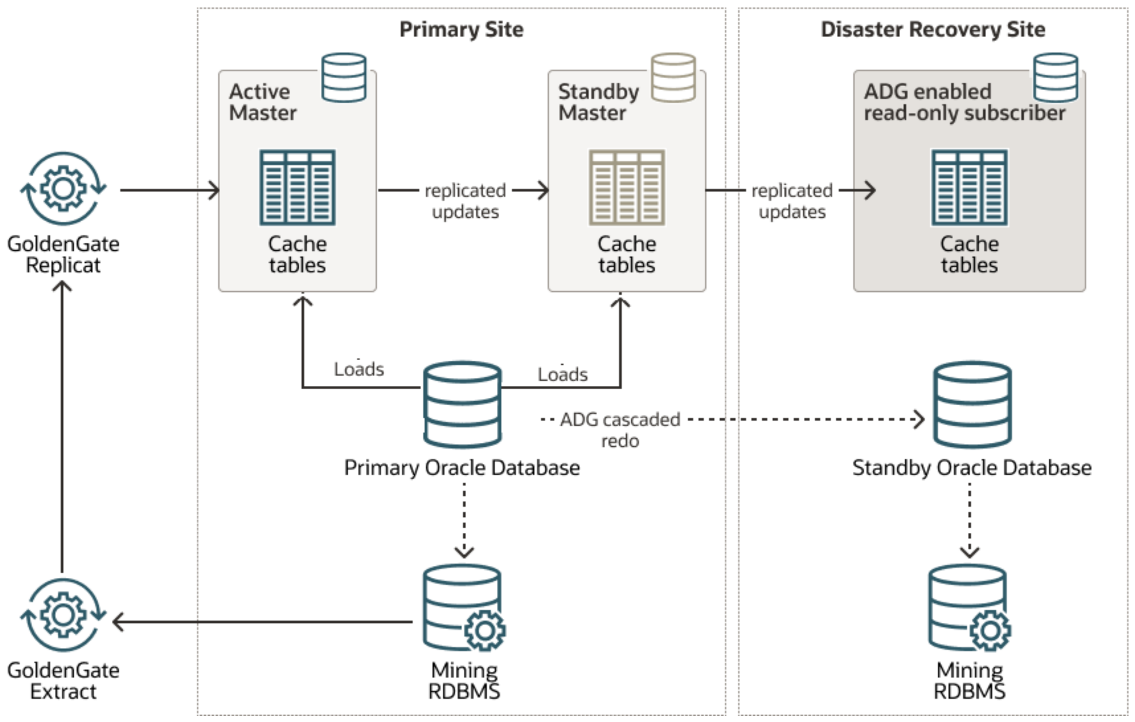


In this scenario, you must start the GoldenGate Extract process on the standby Oracle database using the failover SCN. Similarly, the Replicat process to the TimesTen subscriber which is now functioning as the active master database must also be started from the failover SCN.

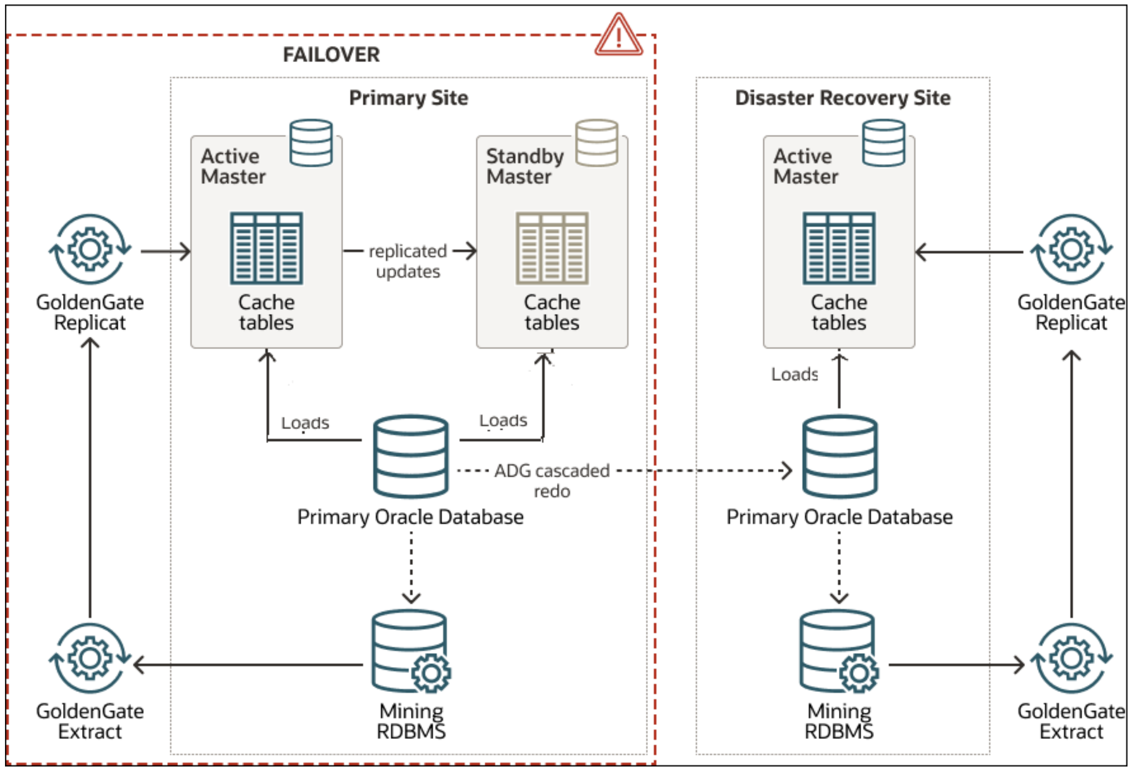
Cascaded Standbys with GoldenGate Extract on a Downstream Mining RDBMS

GoldenGate recommends an alternative configuration for scenarios where running an Extract on the primary database causes excessive impact. In this setup, Active Data Guard enables cascaded redo, allowing the primary database's redo to be distributed to multiple downstream mining databases, including standby databases. See Downstream Extract for Oracle GoldenGate Deployment in the *Oracle GoldenGate Microservices Architecture Solutions*.

GoldenGate capture can then operate from the mining databases, with each site hosting its own mining database. In remote sites, each mining database receives forwarded redo logs either directly from the primary database or from a physical standby database.



During failover, if the primary site goes down, the standby database is promoted to the new primary database, and the ADG cascaded redo is configured to switchover automatically to the new primary database. To recover the read-only subscriber cache database in this disaster recovery scenario, you have to drop the active standby pair configuration from the read-only subscriber database.



You must start the GoldenGate Extract process from the mining databases using the failover SCN. Similarly, the Replicat process to the TimesTen subscriber which is now functioning as the active master database must also be started from the failover SCN. For details, see Recovering from a Failure of the Active Database in the *Oracle TimesTen In-Memory Replication Guide*.

A

Required Privileges for Cache Administration User for Cache Operations

The privileges that the cache administration users require depends on the types of cache groups you create and the operations that you perform on the cache groups.

The privileges required for the Oracle cache administration user are listed in the first column and the privileges required for the TimesTen cache administration user for each cache operation are listed in the second column in Table A-1.

Note that the `CACHE_MANAGER` privilege confers these privileges:

- `CREATE ANY CACHE GROUP`
- `ALTER ANY CACHE GROUP`
- `DROP ANY CACHE GROUP`
- `FLUSH ANY CACHE GROUP`
- `LOAD ANY CACHE GROUP`
- `UNLOAD ANY CACHE GROUP`
- `REFRESH ANY CACHE GROUP`
- `FLUSH` (object)
- `LOAD` (object)
- `UNLOAD` (object)
- `REFRESH` (object)

The `CACHE_MANAGER` privilege also includes the ability to start and stop the cache agent and the replication agent.

See Privilege Hierarchy in the *Oracle TimesTen In-Memory Database SQL Reference*.

Table A-1 Oracle Database and TimesTen User Privileges Required for Cache Operations

Cache Operation	Privileges Required for Oracle Database Cache Administration User[1]	Privileges Required for TimesTen Cache Administration User[2]
Minimum privileges required	At minimum, the Oracle cache administration user must have the `CREATE TYPE` privilege	At minimum, the TimesTen cache administration user must have the `CREATE SESSION` privilege.

Table A-1 (Cont.) Oracle Database and TimesTen User Privileges Required for Cache Operations

Cache Operation	Privileges Required for Oracle Database Cache Administration User[1]	Privileges Required for TimesTen Cache Administration User[2]

Initialize the Oracle cache administration user with the `grantCacheAdminPrivileges.sql` script, which grants these privileges.	`CREATE ANY TRIGGER`[3,4]
	`CREATE PROCEDURE`[4]
	`CREATE SEQUENCE`
	`CREATE SESSION`
	`CREATE TABLE`
	`CREATE TYPE`
	`EXECUTE ON SYS.DBMS_DDL` package
	`EXECUTE ON SYS.DBMS_FLASHBACK` package
	`EXECUTE ON SYS.DBMS_LOB` package
	`EXECUTE ON SYS.DBMS_LOCK` package
	`SELECT ANY TRANSACTION`
	`SELECT ON SYS.ALL_OBJECTS`
	`SELECT ON SYS.ALL_SYNONYMS`
	`SELECT ON SYS.DBA_DATA_FILES`
	`SELECT ON SYS.GV_$LOCK`
	`SELECT ON SYS.GV_$SESSION`
	`SELECT ON SYS.USER_FREE_SPACE`
	`SELECT ON SYS.USER_SYS_PRIVS`
	`SELECT ON SYS.USER_TS_QUOTAS`
	`SELECT ON SYS.USER_USERS`
	`SELECT ON SYS.V_$DATABASE`
	`SELECT ON SYS.V_$PROCESS`
	`SELECT ON SYS.V_$SESSION`
	`TT_CACHE_ADMIN_ROLE`
	`UNLIMITED TABLESPACE`
		None

Table A-1 (Cont.) Oracle Database and TimesTen User Privileges Required for Cache Operations

Cache Operation	Privileges Required for Oracle Database Cache Administration User[1]	Privileges Required for TimesTen Cache Administration User[2]

Initialize the Oracle cache administration user with the `initCacheAdminSchema.sql` script, which grants these privileges.	`CREATE ANY TRIGGER`
	`CREATE SESSION`
	`CREATE TYPE`
	`EXECUTE ON SYS.DBMS_DDL` package
	`EXECUTE ON SYS.DBMS_FLASHBACK` package
	`EXECUTE ON SYS.DBMS_LOCK` package
	`SELECT ANY TRANSACTION`
	`SELECT ON SYS.ALL_OBJECTS`
	`SELECT ON SYS.ALL_SYNONYMS`
	`SELECT ON SYS.DBA_DATA_FILES`
	`SELECT ON SYS.GV_$LOCK`
	`SELECT ON SYS.GV_$SESSION`
	`SELECT ON SYS.USER_FREE_SPACE`
	`SELECT ON SYS.USER_SYS_PRIVS`
	`SELECT ON SYS.USER_TS_QUOTAS`
	`SELECT ON SYS.USER_USERS`
	`SELECT ON SYS.V_$DATABASE`
	`SELECT ON SYS.V_$PROCESS`
	`SELECT ON SYS.V_$SESSION`
	`TT_CACHE_ADMIN_ROLE`
	`UNLIMITED TABLESPACE`
		None

Set the Oracle cache administration user or TimesTen cache administration user name and password:
• In TimesTen, you can call the `ttCacheUidPwdSet` built-in procedure.
• In TimesTen, you can run the `ttAdmin -cacheUidPwdSet` utility command.
	`CREATE PROCEDURE`[4]
	`CREATE SEQUENCE`
	`CREATE SESSION`
	`CREATE TABLE`
	`CREATE TRIGGER`
	`CREATE TYPE`
	Requires access to the default tablespace on the Oracle database. See Create the Oracle Database Users and Default Tablespace.
		`CACHE_MANAGER`
		Requires access to the default tablespace on the Oracle database. See Create the Oracle Database Users and Default Tablespace.

Get the Oracle cache administration user or TimesTen cache administration user name with either: • Call the `ttCacheUidGet` built-in procedure. • In TimesTen, you can run the `ttAdmin -cacheUidGet` utility command.	None	`CACHE_MANAGER`

Table A-1 (Cont.) Oracle Database and TimesTen User Privileges Required for Cache Operations

Cache Operation	Privileges Required for Oracle Database Cache Administration User[1]	Privileges Required for TimesTen Cache Administration User[2]
Start the cache agent with either: • In TimesTen, you can call the `ttCacheStart` built-in procedure. • In TimesTen, you can run the `ttAdmin -cacheStart` utility command.	`CREATE SESSION`	`CACHE_MANAGER`
Stop the cache agent • In TimesTen, you can call the `ttCacheStop` built-in procedure • In TimesTen, you can run the `ttAdmin -cacheStop` utility command •	None	`CACHE_MANAGER`
In TimesTen, set a cache agent start policy with either: • Call the `ttCachePolicySet` built-in procedure. • Run the `ttAdmin -cachePolicy` utility command.	`CREATE SESSION`[4]	`CACHE_MANAGER`
In TimesTen, return the cache agent start policy setting: • Call the `ttCachePolicyGet` built-in procedure.	`CREATE SESSION`	None
In TimesTen, start the replication agent with either: • Call the `ttRepStart` built-in procedure. • Run the `ttAdmin -repStart` utility command.	None	`CACHE_MANAGER`
In TimesTen, stop the replication agent with either: • Call the `ttRepStop` built-in procedure. • Run the `ttAdmin -repStop` utility command.	None	`CACHE_MANAGER`
In TimesTen, set a replication agent start policy • Call the `ttRepPolicySet` built-in procedure • Run the `ttAdmin -repPolicy` utility command	None	`ADMIN`
In TimesTen, `CREATE ACTIVE STANDBY PAIR` with `INCLUDE CACHE GROUP` when the cache group created is an AWT cache group	`CREATE TRIGGER` Creating a cache group requires access to the default tablespace on the Oracle database. See Create the Oracle Database Users and Default Tablespace.	Creating a cache group requires access to the default tablespace on the Oracle database. See Create the Oracle Database Users and Default Tablespace.

Table A-1 (Cont.) Oracle Database and TimesTen User Privileges Required for Cache Operations

Cache Operation	Privileges Required for Oracle Database Cache Administration User[1]	Privileges Required for TimesTen Cache Administration User[2]
In TimesTen, duplicate the database with `ttRepAdmin -duplicate` when using an AWT cache group within an active standby pair replication scheme	`CREATE TRIGGER`	None

`CREATE [DYNAMIC] READONLY CACHE GROUP` with `AUTOREFRESH MODE INCREMENTAL`	`CREATE PROCEDURE`[4]
	`CREATE SEQUENCE`
	`CREATE SESSION`
	`CREATE TABLE`
	`CREATE TYPE`
	`SELECT ON` *`table_name`*[5]
	`CREATE ANY TRIGGER`[4]
	Creating a cache group requires access to the default tablespace on the Oracle database. See Create the Oracle Database Users and Default Tablespace.
		`CREATE [ANY] CACHE GROUP`[6]
		`CREATE [ANY] TABLE`[7]
		Creating a cache group requires access to the default tablespace on the Oracle database. See Create the Oracle Database Users and Default Tablespace.

In TimesTen, `CREATE [DYNAMIC] READONLY CACHE GROUP` with `AUTOREFRESH MODE FULL`	`CREATE SESSION` `SELECT ON` *`table_name`*[5] Creating a cache group requires access to the default tablespace on the Oracle database. See Create the Oracle Database Users and Default Tablespace.	`CREATE [ANY] CACHE GROUP`[6] `CREATE [ANY] TABLE`[7] Creating a cache group requires access to the default tablespace on the Oracle database. See Create the Oracle Database Users and Default Tablespace.

In TimesTen, `CREATE [DYNAMIC] ASYNCHRONOUS WRITETHROUGH CACHE GROUP`	`CREATE PROCEDURE`[4]
	`CREATE SEQUENCE`
	`CREATE SESSION`
	`CREATE TABLE`
	`CREATE TRIGGER`
	`CREATE TYPE`
	`SELECT ON` *`table_name`*[5]
	Creating a cache group requires access to the default tablespace on the Oracle database. See Create the Oracle Database Users and Default Tablespace.
		`CREATE [ANY] CACHE GROUP`[6]
		`CREATE [ANY] TABLE`[7]
		Creating a cache group requires access to the default tablespace on the Oracle database. See Create the Oracle Database Users and Default Tablespace.

In TimesTen, `CREATE [DYNAMIC] SYNCHRONOUS WRITETHROUGH CACHE GROUP`	`CREATE SESSION` `SELECT ON` *`table_name`*[5] Creating a cache group requires access to the default tablespace on the Oracle database. See Create the Oracle Database Users and Default Tablespace.	`CREATE [ANY] CACHE GROUP`[6] `CREATE [ANY] TABLE`[7] Creating a cache group requires access to the default tablespace on the Oracle database. See Create the Oracle Database Users and Default Tablespace.

Table A-1 (Cont.) Oracle Database and TimesTen User Privileges Required for Cache Operations

Cache Operation	Privileges Required for Oracle Database Cache Administration User[1]	Privileges Required for TimesTen Cache Administration User[2]
In TimesTen, `CREATE [DYNAMIC] USERMANAGED CACHE GROUP` (see variants in following rows)	`CREATE SESSION` `SELECT ON` *`table_name`*[5] Creating a cache group requires access to the default tablespace on the Oracle database. See Create the Oracle Database Users and Default Tablespace.	`CREATE [ANY] CACHE GROUP`[6] `CREATE [ANY] TABLE`[7] Creating a cache group requires access to the default tablespace on the Oracle database. See Create the Oracle Database Users and Default Tablespace.

In TimesTen, `CREATE [DYNAMIC] USERMANAGED CACHE GROUP` with `AUTOREFRESH MODE INCREMENTAL`	`CREATE PROCEDURE`[4]
	`CREATE SEQUENCE`
	`CREATE SESSION`
	`CREATE TABLE`
	`CREATE TYPE`
	`SELECT ON` *`table_name`*[5]
	`CREATE ANY TRIGGER`[4]
	Creating a cache group requires access to the default tablespace on the Oracle database. See Create the Oracle Database Users and Default Tablespace.
		`CREATE [ANY] CACHE GROUP`[6]
		`CREATE [ANY] TABLE`[7]
		Creating a cache group requires access to the default tablespace on the Oracle database. See Create the Oracle Database Users and Default Tablespace.

In TimesTen, `CREATE [DYNAMIC] USERMANAGED CACHE GROUP` with `AUTOREFRESH MODE FULL`	`CREATE SESSION` `SELECT ON` *`table_name`*[5] Creating a cache group requires access to the default tablespace on the Oracle database. See Create the Oracle Database Users and Default Tablespace.	`CREATE [ANY] CACHE GROUP`[6] `CREATE [ANY] TABLE`[7] Creating a cache group requires access to the default tablespace on the Oracle database. See Create the Oracle Database Users and Default Tablespace.
In TimesTen, `CREATE [DYNAMIC] USERMANAGED CACHE GROUP` with `READONLY`	`CREATE SESSION` `SELECT ON` *`table_name`*[5] Creating a cache group requires access to the default tablespace on the Oracle database. See Create the Oracle Database Users and Default Tablespace.	`CREATE [ANY] CACHE GROUP`[6] `CREATE [ANY] TABLE`[7] Creating a cache group requires access to the default tablespace on the Oracle database. See Create the Oracle Database Users and Default Tablespace.
In TimesTen, `CREATE [DYNAMIC] USERMANAGED CACHE GROUP` with `PROPAGATE`	`CREATE SESSION` `SELECT ON` *`table_name`*[5] Creating a cache group requires access to the default tablespace on the Oracle database. See Create the Oracle Database Users and Default Tablespace.	`CREATE [ANY] CACHE GROUP`[6] `CREATE [ANY] TABLE`[7] Creating a cache group requires access to the default tablespace on the Oracle database. See Create the Oracle Database Users and Default Tablespace.

Table A-1 (Cont.) Oracle Database and TimesTen User Privileges Required for Cache Operations

Cache Operation	Privileges Required for Oracle Database Cache Administration User[1]	Privileges Required for TimesTen Cache Administration User[2]

`ALTER CACHE GROUP SET AUTOREFRESH STATE PAUSED`	`CREATE PROCEDURE`[4]
	`CREATE SEQUENCE`
	`CREATE SESSION`
	`CREATE TABLE`
	`CREATE TRIGGER`
	`CREATE TYPE`
	`SELECT ON` *`table_name`*[5,8]
	`CREATE ANY TRIGGER`[4,8]
		`ALTER ANY CACHE GROUP`[9]

`ALTER CACHE GROUP SET AUTOREFRESH STATE ON`	`CREATE PROCEDURE`[4]
	`CREATE SEQUENCE`
	`CREATE SESSION`
	`CREATE TABLE`
	`CREATE TYPE`
	`SELECT ON` *`table_name`*[5,8]
	`CREATE ANY TRIGGER`[4,8]
		`ALTER ANY CACHE GROUP`[9]

`ALTER CACHE GROUP SET AUTOREFRESH STATE OFF`	`CREATE SESSION`	`ALTER ANY CACHE GROUP`[9]
In TimesTen, `ALTER CACHE GROUP SET AUTOREFRESH MODE FULL`	`CREATE SESSION`	`ALTER ANY CACHE GROUP`[9]

In TimesTen, `ALTER CACHE GROUP SET AUTOREFRESH MODE INCREMENTAL`	`CREATE PROCEDURE`[4]
	`CREATE SEQUENCE`
	`CREATE SESSION`
	`CREATE TABLE`
	`CREATE TYPE`
	`SELECT ON` *`table_name`*[5]
	`CREATE ANY TRIGGER`[4]
		`ALTER ANY CACHE GROUP`[9]

`ALTER CACHE GROUP SET AUTOREFRESH INTERVAL`	`CREATE SESSION` `SELECT ON` *`table_name`*[5,10]	`ALTER ANY CACHE GROUP`[9]
`LOAD CACHE GROUP`	`CREATE SESSION` `SELECT ON` *`table_name`*[5]	`LOAD {ANY CACHE GROUP | ON` *`cache_group_name`*`}`[9]
`REFRESH CACHE GROUP`	`CREATE SESSION` `SELECT ON` *`table_name`*[5]	`REFRESH {ANY CACHE GROUP | ON` *`cache_group_name`*`}`[9] `EXECUTE ON SYS.DBMS_FLASHBACK` package on the Oracle Database
`FLUSH CACHE GROUP`	`SELECT ON` *`table_name`*[5] `CREATE SESSION` `UPDATE ON` *`table_name`*[5] `INSERT ON` *`table_name`*[5]	`SELECT ON` *`table_name`*[5] `FLUSH {ANY CACHE GROUP | ON` *`cache_group_name`*`}`[9]
`UNLOAD CACHE GROUP`	None	`UNLOAD {ANY CACHE GROUP | ON` *`cache_group_name`*`}`[9]

Table A-1 (Cont.) Oracle Database and TimesTen User Privileges Required for Cache Operations

Cache Operation	Privileges Required for Oracle Database Cache Administration User[1]	Privileges Required for TimesTen Cache Administration User[2]
`DROP CACHE GROUP`	`CREATE SESSION`	`DROP ANY CACHE GROUP`[9] `DROP ANY TABLE`[11]
In TimesTen, synchronous writethrough or propagate	`CREATE SESSION` `INSERT ON` *`table_name`*[5, 12] `UPDATE ON` *`table_name`*[5, 12] `DELETE ON` *`table_name`*[5, 12]	`INSERT ON` *`table_name`*[13] `UPDATE ON` *`table_name`*[13] `DELETE ON` *`table_name`*[13]
In TimesTen, asynchronous writethrough	`CREATE SESSION` `INSERT ON` *`table_name`*[5] `UPDATE ON` *`table_name`*[5] `DELETE ON` *`table_name`*[5]	`INSERT ON` *`table_name`*[13] `UPDATE ON` *`table_name`*[13] `DELETE ON` *`table_name`*[13]
In TimesTen, asynchronous writethrough when the `CacheAWTMethod` connection attribute is set to 1	`CREATE PROCEDURE` **Note:** This privilege is an addition to the privileges needed for any asynchronous writethrough cache group.	None
In TimesTen, asynchronous writethrough cache for Oracle Database `CLOB`, `BLOB` and `NCLOB` fields when the `CacheAWTMethod` connection attribute is set to 1	`EXECUTE` privilege on the Oracle Database `DBMS_LOB` PL/SQL package **Note:** This privilege is an addition to the privileges needed for any asynchronous writethrough cache group.	None
Incremental autorefresh	`SELECT ON` *`table_name`*[5]	None
Full autorefresh	`SELECT ON` *`table_name`*[5]	None
In TimesTen, dynamic load	`CREATE SESSION` `SELECT ON` *`table_name`*[5]	`SELECT ON` *`table_name`*[13] `UPDATE ON` *`table_name`*[13] `DELETE ON` *`table_name`*[13] `INSERT ON` *`table_name`*[13]
In TimesTen, aging	None	`DELETE {ANY TABLE | ON` *`table_name`*`}`[13]
In TimesTen, set the LRU aging attributes • Call the `ttAgingLRUConfig` built-in procedure • Call the `ttAgingTableLRUConfig` built-in procedure	None	`ADMIN`
Generate Oracle Database SQL statements to manually install or uninstall Oracle Database objects • Run the `ttIsql` utility's `cachesqlget` command • Call the `ttCacheSQLGet` built-in procedure	`CREATE SESSION`	`CACHE_MANAGER`

Table A-1 (Cont.) Oracle Database and TimesTen User Privileges Required for Cache Operations

Cache Operation	Privileges Required for Oracle Database Cache Administration User[1]	Privileges Required for TimesTen Cache Administration User[2]
In TimesTen, disable or enable propagation of committed cache table updates to the Oracle database • Call the `ttCachePropagateFlagSet` built-in procedure	None	`CACHE_MANAGER`
Configure cache agent timeout and recovery method for cache groups with autorefresh • Call the `ttCacheConfig` built-in procedure	`CREATE SESSION`	`CACHE_MANAGER`
In TimesTen, set the AWT transaction log file threshold • Call the `ttCacheAWTThresholdSet` built-in procedure	None	`CACHE_MANAGER`
In TimesTen, enable or disable monitoring of AWT cache groups • Call the `ttCacheAWTMonitorConfig` built-in procedure	None	`CACHE_MANAGER`
Enable or disable tracking of DDL statements issued on cached Oracle Database tables • Call the `ttCacheDDLTrackingConfig` built-in procedure	`CREATE SESSION`	`CACHE_MANAGER`

1 At minimum, the Oracle cache administration user must have the `CREATE TYPE` privilege.

2 At minimum, the TimesTen cache administration user must have the `CREATE SESSION` privilege.

3 If the Oracle cache administration user will not create cache groups with autorefresh, then you can grant the `CREATE TRIGGER` privilege instead of the `CREATE ANY TRIGGER` privilege.

4 Required if the cache agent start policy is being set to `always` or `norestart`.

5 Required on all Oracle Database tables cached in the TimesTen cache group except for tables owned by the Oracle cache administration user.

6 The `CACHE_MANAGER` privilege includes the `CREATE [ANY] CACHE GROUP` privilege. `ANY` is required if the TimesTen cache administration user creates cache groups owned by a user other than itself.

7 `ANY` is required if any of the cache tables are owned by a user other than the TimesTen cache administration user.

8 Required if the cache group's autorefresh mode is incremental and initial autorefresh state is `OFF`, and the Oracle Database objects used to manage the caching of Oracle Database data are automatically created.

9 Required if the TimesTen user accessing the cache group does not own the cache group.

10 Required if the cache group's autorefresh mode is incremental.

11 Required if the TimesTen user accessing the cache group does not own all its cache tables.

12 The privilege must be granted to the Oracle Database user with the same name as the TimesTen cache administration user if the Oracle Database user is not the Oracle cache administration user.

13 Required if the TimesTen user accessing the cache table does not own the table.

B

SQL*Plus Scripts for Cache

TimesTen is installed with SQL*Plus scripts that are used to perform various cache configuration, administrative and monitoring tasks, and provide links to more information including examples.

All scripts are installed in the *timesten_home*/install/oraclescripts directory.

Installed SQL*Plus Scripts

There are SQL*Plus scripts that are installed with TimesTen.

- cacheCleanUp.sql: This script drops Oracle Database objects such as change log tables and triggers used to implement autorefresh operations for TimesTen. This script is used when a TimesTen database containing cache groups with autorefresh is unavailable because the TimesTen system is offline, or the database was destroyed without dropping its cache groups with autorefresh. Run this script as the cache administration user. Provide the host name of the TimesTen system and the TimesTen database (including its path) as arguments. See Dropping Oracle Database Objects Used by Cache Groups with Autorefresh.

  This example uses the cacheCleanUp.sql script for a TimesTen system.

```
% cd timesten_home/install/oraclescripts
% sqlplus cacheadmin/orapwd
SQL> @cacheCleanUp "sys1" "/disk1/databases/database1"

*****************************OUTPUT**************************************
Performing cleanup for object_id: 69959 which belongs to table : CUSTOMER
Executing: delete from tt_07_agent_status where host = sys1 and datastore =
/disk1/databases/database1 and object_id = 69959
Executing: drop table tt_07_69959_L
Executing: drop trigger tt_07_69959_T
Executing: delete from tt_07_user_count where object_id = object_id1
Performing cleanup for object_id: 69966 which belongs to table : ORDERS
Executing: delete from tt_07_agent_status where host = sys1 and datastore =
/disk1/databases/database1 and object_id = 69966
Executing: drop table tt_07_69966_L
Executing: drop trigger tt_07_69966_T
Executing: delete from tt_07_user_count where object_id = object_id1
**************************************************************************
```

- cacheInfo.sql: This script returns change log table information for all Oracle Database tables cached in a cache group with autorefresh, and information about Oracle Database objects used to track DDL statements issued on cached Oracle Database tables. This script is used to monitor autorefresh operations on cache groups and DDL statements issued on cached Oracle Database tables. Run this script as the cache administration user. You can alternatively use the ttCacheInfo utility.

  The following example runs the cacheInfo.sql SQL*Plus script.

```
% cd timesten_home/install/oraclescripts
% sqlplus cacheadmin/orapwd
SQL> @cacheInfo.sql
***************** Database Information    *********************
```

```
Database name: DATABASE1
Unique database name: database1
Primary database name:
Database Role: PRIMARY
Database Open Mode: READ WRITE
Database Protection Mode: MAXIMUM PERFORMANCE
Database Protection Level: UNPROTECTED
Database Flashback On: NO
Database Current SCN: 21512609
*************************************************************
*************Autorefresh Objects Information  ***************
Grid name: grid1 (7D03C680-BD93-4233-A4CF-B0EDB0064F3F)
Timesten database name: database1
Cache table name: SALES.CUSTOMERS
Change log table name: tt_07_96977_L
Number of rows in change log table: 4
Maximum logseq on the change log table: 1
Timesten has autorefreshed updates upto logseq: 1
Number of updates waiting to be autorefreshed: 0
Number of updates that has not been marked with a valid logseq: 0
*************DDL Tracking Object Information  ***************
Common DDL Log Table Name: TT_07_DDL_L
DDL Trigger Name: TT_07_315_DDL_T
Schema for which DDL Trigger is tracking: SALES
Number of cache groups using the DDL Trigger: 10
****************************

PL/SQL procedure successfully completed.
```

See Monitoring Autorefresh Operations on Cache Groups and Tracking DDL Statements Issued on Cached Oracle Database Tables in this guide and ttCacheInfo in *Oracle TimesTen In-Memory Database Reference*.

- `grantCacheAdminPrivileges.sql`: This script grants privileges to the cache administration user that are required to automatically create Oracle Database objects used to manage the caching of Oracle Database data when particular cache group operations are performed. This includes the `TT_CACHE_ADMIN_ROLE` role that defines privileges on Oracle Database tables. Run this script as the `sys` user. See Create Oracle Database Objects Used to Manage Data Caching.

  The following example for a non-autonomous Oracle Database grants the required SQL privileges to the `cacheadmin` user for cache operations in the Oracle database:

```
@grantCacheAdminPrivileges.sql cacheadmin


Please enter the administrator user id
The value chosen for administrator user id is cacheadmin

***************** Creation of TT_CACHE_ADMIN_ROLE starts ******************
0. Creating TT_CACHE_ADMIN_ROLE role
** Creation of TT_CACHE_ADMIN_ROLE done successfully **
***************** Initialization for cache admin begins ******************
0. Granting the CREATE SESSION privilege to CACHEADMIN
1. Granting the TT_CACHE_ADMIN_ROLE to CACHEADMIN
2. Granting the DBMS_LOCK package privilege to CACHEADMIN
3. Granting the DBMS_DDL package privilege to CACHEADMIN
4. Granting the DBMS_FLASHBACK package privilege to CACHEADMIN
5. Granting the CREATE SEQUENCE privilege to CACHEADMIN
6. Granting the CREATE CLUSTER privilege to CACHEADMIN
```

```
7. Granting the CREATE OPERATOR privilege to CACHEADMIN
8. Granting the CREATE INDEXTYPE privilege to CACHEADMIN
9. Granting the CREATE TABLE privilege to CACHEADMIN
10. Granting the CREATE PROCEDURE  privilege to CACHEADMIN
11. Granting the CREATE ANY TRIGGER  privilege to CACHEADMIN
12. Granting the GRANT UNLIMITED TABLESPACE privilege to CACHEADMIN
13. Granting the DBMS_LOB package privilege to CACHEADMIN
14. Granting the SELECT on SYS.ALL_OBJECTS privilege to CACHEADMIN
15. Granting the SELECT on SYS.ALL_SYNONYMS privilege to CACHEADMIN
16. Checking if the cache administrator user has permissions on the default
tablespace
     Permission exists
18. Granting the CREATE TYPE privilege to CACHEADMIN
19. Granting the SELECT on SYS.GV$LOCK privilege to CACHEADMIN
20. Granting the SELECT on SYS.GV$SESSION privilege  to CACHEADMIN
21. Granting the SELECT on SYS.DBA_DATA_FILES privilege  to CACHEADMIN
22. Granting the SELECT on SYS.USER_USERS privilege  to CACHEADMIN
23. Granting the SELECT on SYS.USER_FREE_SPACE privilege  to CACHEADMIN
24. Granting the SELECT on SYS.USER_TS_QUOTAS privilege  to CACHEADMIN
25. Granting the SELECT on SYS.USER_SYS_PRIVS privilege  to CACHEADMIN
26. Granting the SELECT on SYS.V$DATABASE privilege  to CACHEADMIN
(optional)
27. Granting the SELECT on SYS.GV$PROCESS privilege  to CACHEADMIN
(optional)
28. Granting the SELECT ANY TRANSACTION privilege to CACHEADMIN
29. Creating the TTCACHEADM.TT_07_ARDL_CG_COUNTER table
30. Granting SELECT privilege on TTCACHEADM.TT_07_ARDL_CG_COUNTER table to
PUBLIC
********* Initialization for cache admin user done successfully *********
```

For Autonomous Transaction Processing, Step 16 output is as follows:

```
16. Checking if the cache administrator user has permissions on the default
tablespace
```

```
No existing permission.
```

Autonomous Transaction Processing automatically configures tablespaces. Therefore, this permission is not necessary.

- `checkAdminPrivileges.sql`: This script checks that the cache administration user has all of the necessary privileges (those that are provided when you run the `grantCacheAdminPrivileges.sql` script) that are required for cache operations. Run this script as the user that you want checked. If privileges are missing, you can either have the `sys` user grant the missing privileges or run the `grantCacheAdminPrivileges.sql` script for this user. See The checkAdminPrivileges.sql Script.

  Use SQL*Plus on the Oracle Database system from an operating system shell or command prompt, and connect to the Oracle database instance as the user (in most cases, the cache administration user) that you want checked for privileges. The following example shows that the user has all of the required privileges.

  ```
  SQL> @checkAdminPrivileges.sql
  **** Checking privileges for cache administrator user ****
  **** User has all privileges for a cache administrator user ****
  ```

  The following example shows the output if you have missing privileges needed on an Oracle database:

```
SQL> @checkAdminPrivileges.sql
**** Checking privileges for cache administrator user ****
Missing CREATE OPERATOR
Missing CREATE INDEXTYPE
Missing CREATE CLUSTER
Missing EXECUTE ON SYS.DBMS_LOCK
Missing EXECUTE ON SYS.DBMS_DDL
Missing EXECUTE ON SYS.DBMS_FLASHBACK
Missing EXECUTE ON SYS.DBMS_LOB
Missing SELECT on SYS.GV$LOCK
Missing SELECT on SYS.GV$SESSION
Missing SELECT on SYS.DBA_DATA_FILES
Missing SELECT on SYS.V$DATABASE
Missing SELECT on GV$PROCESS
Missing UNLIMITED TABLESPACE
Missing SELECT ANY TRANSACTION
Missing table ARDL_CG_COUNTER
**** User missing privileges. Missing privilege count: 15 ****
```

- `initCacheAdminSchema.sql`: This script grants a minimal set of privileges to the cache administration user and manually creates Oracle Database objects used to manage the caching of Oracle Database data. This includes the `TT_CACHE_ADMIN_ROLE` role that defines privileges on Oracle Database tables. Run this script as the `sys` user. See The initCacheAdminSchema.sql Script.

  In the following example, the Oracle database cache administration user name is `cacheadmin`.

  ```
  @initCacheAdminSchema cacheadmin
  ```

C

Compatibility Between TimesTen and Oracle Databases

The following sections list compatibility issues between TimesTen and Oracle Databases. The list is not complete, but it indicates areas that require special attention.

Summary of Compatibility Issues

There are a few compatibility issues between the TimesTen and Oracle databases.

Consider the following differences between TimesTen and Oracle databases:

- TimesTen and Oracle database metadata are stored differently. See API Compatibility.
- TimesTen and Oracle databases have different transaction isolation models. See Transaction Semantics.
- TimesTen and Oracle databases have different connection and statement properties. For example, TimesTen does not support catalog names, scrollable cursors or updateable cursors.
- Sequences are not cached and synchronized between the TimesTen database and the corresponding Oracle database. See SQL Expressions.
- Side effects of Oracle Database triggers and stored procedures are not reflected in the TimesTen database until after an automatic or manual refresh operation.

Transaction Semantics

TimesTen and Oracle Database transaction semantics differ in a few ways.

- Oracle Database serializable transactions can fail at commit time because the transaction cannot be serialized. TimesTen uses locking to enforce serializability.
- Oracle Database can provide both statement-level and transaction-level consistency by using a multi-version consistency model. TimesTen does not provide statement-level consistency. TimesTen provides transaction-level consistency by using serializable isolation.
- Oracle Database users can lock tables manually through SQL. This locking feature is not supported in TimesTen.
- Oracle Database supports savepoints while TimesTen does not.

- In Oracle Database, a transaction can be set to be read-only or read/write. This is not supported in TimesTen.

See Transaction Management in *Oracle TimesTen In-Memory Database Operations Guide*.

API Compatibility

There are methods from the JDBC and ODBC APIs that have a compatibility issue with cache.

The following sections list methods from the JDBC and ODBC APIs that have a compatibility issue with cache.

- JDBC API Compatibility
- ODBC API Compatibility

JDBC API Compatibility

There are compatibility issues that apply to the JDBC API.

Compatibility issues that apply to JDBC include the following:

- JDBC database metadata functions return TimesTen metadata. If you want Oracle metadata, connect to the Oracle Database directly.
- The set/get connection and statement attributes are performed on TimesTen.
- All Oracle `java.sql.ResultSet` metadata (length, type, label) is returned in TimesTen data type lengths. The column labels that are returned are TimesTen column labels.
- Oracle extensions (`oracle.sql` and `oracle.jdbc` packages) are not supported.
- Java stored procedures are not supported in TimesTen.

java.sql.Connection

The following `Connection` methods have no compatibility issues:

```
close()
commit()
createStatement()
prepareCall()
prepareStatement()
rollback()
setAutoCommit()
```

The following methods are run locally in TimesTen:

```
getCatalog()
getMetaData
get/setTransactionIsolation()
isReadOnly()
isClosed()
nativeSQL()
setCatalog()
setReadOnly()
```

Note

See Transaction Semantics for restrictions for the `get/setTransactionIsolation()` methods.

The `isClosed()` method returns only the TimesTen connection status.

java.sql.Statement

The following `Statement` methods have no compatibility issues:

```
addBatch()
clearBatch()
close()
execute()
executeBatch()
executeQuery()
executeUpdate()
getResultSet()
getUpdateCount()
getWarnings()
```

The following methods run locally in TimesTen:

```
cancel()
get/setMaxFieldSize()
get/setMaxRows()
get/setQueryTimeout()
getMoreResults()
setEscapeProcessing()
setCursorName()
```

java.sql.ResultSet

The following `ResultSet` methods have no compatibility issues:

```
close()
findColumn(int) and findColumn(string)
getXXX(number) and getXXX(name)
getXXXStream(int) and getXXXStream(string)
getMetaData()
```

java.sql.PreparedStatement

The following `PreparedStatement` methods have no compatibility issues:

```
addBatch()
close()
execute()
executeUpdate()
executeQuery()
getResultSet()
getUpdateCount()
setXXX()
setXXXStream()
```

The following methods run locally in TimesTen:

```
cancel()
get/setMaxFieldSize()
get/setMaxRows()
get/setQueryTimeout()
getMoreResults()
setEscapeProccessing()
setCursorName()
```

java.sql.CallableStatement

The same restrictions as shown for the `java.sql.Statement` and `java.sql.PreparedStatement` interfaces apply to `CallableStatement`.

- In a `WRITETHROUGH` cache group, if `PassThrough=1`, indirect DML operations that are hidden in stored procedures or induced by triggers may be passed through without being detected by Cache Connect to Oracle.
- Stored procedures that update, insert, or delete from `READONLY` cache group tables will be autorefreshed within another transaction in an asynchronous fashion. Thus, the changes do not appear within the same transaction that the stored procedure was processed within and there may be some time lapse before the changes are autorefreshed into the cache table.

java.sql.ResultSetMetaData

The following `ResultSetMetaData` methods have no compatibility issues:

```
getColumnCount()
getColumnType()
getColumnLabel()
getColumnName()
getTableName()
isNullable()
```

The following methods run locally in TimesTen:

```
getSchemaName()
getCatalogName()
getColumnDisplaySize()
getColumnType()
getColumnTypeName()
getPrecision()
getScale()
isAutoIncrement()
isCaseSensitive()
isCurrency()
isDefinitelyWritable()
isReadOnly()
isSearchable()
isSigned()
isWritable()
```

Stream Support

There are compatibility issues related to streams.

The compatibility issues related to streams are:

- The JDBC driver fully fetches the data into an in-memory buffer during a call to the `executeQuery()` or `next()` methods. The `getXXXStream()` entry points return a stream that reads data from this buffer.
- Oracle supports up to 2 GB of long or long raw data. When cached, TimesTen converts `LONG` data into `VARCHAR2` data. TimesTen converts `LONG RAW` data into `VARBINARY` data. Both `VARCHAR2` and `VARBINARY` data types can store up to a maximum 4,194,304 (2^{22}) bytes).
- Oracle always streams `LONG`/`LONG RAW` data even if the application does not call `getXXXStream()`.
- TimesTen does not support the `mark()`, `markSupported()`, and `reset()` methods.

ODBC API Compatibility

Cache in TimesTen is compatible with a subset of ODBC functions.

Table C-1 describes the compatibility of ODBC functions.

Table C-1 ODBC Function Compatibility With Cache in TimesTen

Function Name	Compatibility
`SQLBindParameter`	Default TimesTen behavior matches Oracle Database behavior. See Parameter Binding and Statement Execution in *Oracle TimesTen In-Memory Database C Developer's Guide*.
`SQLBrowseConnect`, `SQLColumnPrivileges`, `SQLExtendedFetch`, `SQLMoreResults`, `SQLSetPos`, `SQLSetScrollOptions`, `SQLTablePrivileges`	Not supported.
`SQLCancel`	There are some restrictions. In particular, `SQLCancel` cannot cancel TimesTen administrative operations. See the `SQLCancel` entry in ODBC 2.5 Function Support in the *Oracle TimesTen In-Memory Database C Developer's Guide*.
`SQLGetCursorName`	There are some restrictions. See the `SQLGetCursorName` entry in ODBC 2.5 Function Support in the *Oracle TimesTen In-Memory Database C Developer's Guide*.

SQL Compatibility

This section compares TimesTen's SQL implementation with Oracle Database SQL.

The purpose is to provide users with a list of Oracle Database SQL features not supported in TimesTen or supported with different semantics.

- Schema Objects
- Non-Schema Objects
- Differences Between Oracle Database and TimesTen Tables
- Data Type Support
- SQL Operators
- SELECT Statements
- SQL Subqueries

- SQL Functions
- SQL Expressions
- INSERT/DELETE/UPDATE/MERGE Statements
- TimesTen-Only SQL and Built-In Procedures
- PL/SQL Constructs

Schema Objects

TimesTen does not recognize some of the schema objects that are supported in Oracle Database.

TimesTen returns a syntax error when a statement manipulates or uses these objects. TimesTen passes the statement to Oracle Database. The unsupported objects are:

Clusters
Objects created by the `CREATE DATABASE` statement
Objects created by the `CREATE JAVA` statement
Database links
Database triggers
Dimensions
Extended features
External procedure libraries
Index-organized tables
Mining models
Partitions
Object tables, types and views
Operators

TimesTen supports views and materialized views, but it cannot cache an Oracle Database view. TimesTen can cache an Oracle Database materialized view in a user-managed cache group without the `AUTOREFRESH` cache group attribute and `PROPAGATE` cache table attribute. The cache group must be manually loaded and flushed.

Caching Oracle Database Partitioned Tables

TimesTen can cache Oracle Database partitioned tables at the table level, but individual partitions cannot be cached.

The following describes how operations on partitioned tables affect cache groups:

- DDL operations on a table that has partitions do not affect the cache group unless there is data loss. For example, if a partition with data is truncated, an `AUTOREFRESH` operation does not delete the data from the corresponding cached table.
- `WHERE` clauses in any cache group operations cannot reference individual partitions or sub-partitions. Any attempt to define a single partition of a table returns an error.

Non-Schema Objects

TimesTen does not recognize some of the schema objects that are supported in Oracle Database.

TimesTen returns a syntax error when a statement manipulates or uses these objects. TimesTen passes the statement to Oracle Database. The unsupported objects are:

Contexts
Directories
Editions
Restore points
Roles
Rollback segments
Tablespaces

Differences Between Oracle Database and TimesTen Tables

TimesTen supports a subset of the Oracle Database features.

The Oracle Database table features that TimesTen does not support are:

- `ON DELETE SET NULL`
- Check constraints
- Foreign keys that reference the table on which they are defined

Data Type Support

Certain Oracle Database data types are not supported by TimesTen.

`TIMESTAMP WITH TIME ZONE`
`TIMESTAMP WITH LOCAL TIME ZONE`
`INTERVAL YEAR TO MONTH`
`INTERVAL DAY TO SECOND`
`UROWID`
`BFILE`
Oracle Database-supplied types
User-defined types

The following TimesTen data types are not supported by Oracle Database:

`TT_CHAR`
`TT_VARCHAR`
`TT_NCHAR`
`TT_NVARCHAR`
`TT_BINARY`
`TT_VARBINARY`
`TINYINT` and `TT_TINYINT`
`TT_SMALLINT`
`TT_INTEGER`
`TT_BIGINT`
`TT_DECIMAL`

TT_DATE
TIME and TT_TIME
TT_TIMESTAMP

Note

TimesTen NCHAR and NVARCHAR2 data types are encoded as UTF-16. Oracle Database NCHAR and NVARCHAR2 data types are encoded as either UTF-16 or UTF-8.

To cache an Oracle Database NCHAR or NVARCHAR2 column, the Oracle Database NLS_NCHAR_CHARACTERSET encoding must be AL16UTF16, not AL32UTF8.

SQL Operators

TimesTen supports a subset of operators and predicates that are supported by the Oracle Database:

unary -
+, -, *, /
=, <, >, <=, >=, <>, !=
||
IS NULL, IS NOT NULL
LIKE (Oracle Database LIKE operator ignores trailing spaces, but TimesTen does not)
BETWEEN
IN
NOT IN (list)
AND
OR
+ (outer join)
ANY, SOME
ALL (list)
EXISTS
UNION
MINUS
INTERSECT

To run a bitwise AND operation of two bit vector expressions, TimesTen uses the ampersand character (&) between the expressions while Oracle Database uses the BITAND function with the expressions as arguments.

SELECT Statements

TimesTen supports a subset of clauses of a SELECT statement that are supported by the Oracle Database:

- FOR UPDATE
- ORDER BY, including NULLS FIRST and NULLS LAST
- GROUP BY, including ROLLUP, GROUPING_SETS and grouping expression lists
- Table alias

- Column alias
- Subquery factoring clause with constructor

Oracle Database supports flashback queries, which are queries against a database that is in some previous state (for example, a query on a table as of yesterday). TimesTen does not support flashback queries.

TimesTen does not support the `CONNECT BY` clause.

SQL Subqueries

TimesTen supports a subset of subqueries that are supported by the Oracle Database.

`IN` (subquery)
`>,<,= ANY` (subquery)
`>,=,< SOME` (subquery)
`EXISTS` (subquery)
`>,=,<` (scalar subquery)
Subqueries in `WHERE` clause of `DELETE/UPDATE`
Subqueries in `FROM` clause
Subquery factoring clause (`WITH` constructor)

Note

A nonverifiable scalar subquery is a scalar subquery whose 'single-row-result-set' property cannot be determined until runtime. TimesTen allows at most one nonverifiable scalar subquery in the entire query and the subquery cannot be specified in an `OR` expression.

SQL Functions

TimesTen supports a subset of functions that are supported by the Oracle Database.

```
ABS
ADD_MONTHS
ASCIISTR
AVG
CAST
CEIL
COALESCE
CONCAT
COUNT
CHR
DECODE
DENSE_RANK
EMPTY_BLOB
EMPTY_CLOB
EXTRACT
FIRST_VALUE
FLOOR
```

GREATEST
GROUP_ID
GROUPING
GROUPING_ID
INSTR
LAST_VALUE
LEAST
LENGTH
LOWER
LPAD
LTRIM
MAX
MIN
MOD
MONTHS_BETWEEN
NCHR
NLS_CHARSET
NLS_CHARSET_NAME
NLSSORT
NULLIF
NUMTOYMINTERVAL
NUMTODSINTERVAL
NVL
POWER
RANK
REPLACE
ROUND
ROW_NUMBER
RPAD
RTRIM
SIGN
SQRT
SUBSTR
SUM
SYS_CONTEXT
SYSDATE
TO_BLOB
TO_CLOB
TO_CHAR
TO_DATE
TO_LOB
TO_NCLOB
TO_NUMBER
TRIM
TRUNC
UID
UNISTR
UPPER

`USER`

These TimesTen functions are not supported by Oracle Database:

```
CURRENT_USER
GETDATE
ORA_SYSDATE
SESSION_USER
SYSTEM_USER
TIMESTAMPADD
TIMESTAMPDIFF
TT_HASH
TT_SYSDATE
```

TimesTen and the Oracle Database interpret the literal `N'\UNNNN'` differently. In TimesTen, `N'\u`*`nnnn`*`'` (where *nnnn* is a number) is interpreted as the national character set character with the code *nnnn*. In the Oracle Database, `N'\u`*`nnnn`*`'` is interpreted as 6 literal characters. The `\u` is not treated as an escape. This difference causes unexpected behavior. For example, loading a cache group with a `WHERE` clause that contains a literal can fail. This can also affects dynamic loading. Applications should use the `UNISTR` SQL function instead of literals.

SQL Expressions

TimesTen supports a subset of expressions that are supported by the Oracle Database.

Column Reference
Sequence
`NULL`
`()`
Binding parameters
`CASE` expression
`ROWID` pseudocolumn
`ROWNUM` pseudocolumn

TimesTen and Oracle Database treat literals differently. See the description of *`HexadecimalLiteral`* in Constants in *Oracle TimesTen In-Memory Database SQL Reference*.

INSERT/DELETE/UPDATE/MERGE Statements

TimesTen supports certain DML statements that are also supported by the Oracle Database.

- `INSERT INTO ... VALUES`
- `INSERT INTO ... SELECT`
- `UPDATE WHERE` expression (expression may contain a subquery)
- `DELETE WHERE` expression (expression may contain a subquery)

TimesTen does not support updating of primary key values except when the new value is the same as the old value.

TimesTen-Only SQL and Built-In Procedures

There are TimesTen SQL statements and functions and built-in procedures that are not supported by the Oracle Database.

With `PassThrough`=3, these statements are passed to Oracle Database for processing and an error is generated.

- All TimesTen cache group DDL and DML statements, including `CREATE CACHE GROUP`, `DROP CACHE GROUP`, `ALTER CACHE GROUP`, `LOAD CACHE GROUP`, `UNLOAD CACHE GROUP`, `REFRESH CACHE GROUP` and `FLUSH CACHE GROUP`.
- All TimesTen replication management DDL statements, including `CREATE REPLICATION`, `DROP REPLICATION`, `ALTER REPLICATION`, `CREATE ACTIVE STANDBY PAIR`, `ALTER ACTIVE STANDBY PAIR` and `DROP ACTIVE STANDBY PAIR`.
- `FIRST` *n* clause.
- `ROWS` *m* `TO` *n* clause.
- All TimesTen built-in procedures. See Built-In Procedures in *Oracle TimesTen In-Memory Database Reference*.
- TimesTen specific syntax for character and unicode strings are not always converted to the Oracle Database syntax when using `PassThrough`=3.

  Note

  For more details on TimesTen support for unicode strings, see *Character and Unicode Strings* in the *Oracle TimesTen In-Memory Database Reference*.

  – Supplying `\046` converts to the `&` symbol on TimesTen, but is not converted to this symbol when passed through to an Oracle database. The `\`*xyz* notation is not supported by the Oracle database. To send a character through to an Oracle database, pass it as an argument within the `CHR()` function with the decimal value of the character.

  – TimesTen enables depicting a unicode value (a four-digit hexadecimal number) within a character string with the `\u`*xyzw* syntax (for `NCHAR` and `NVARCHAR2` only) where you substitute the unicode value for *xyzw*, as in`\ufe4a`.

    The `\u`*xyzw* notation is not supported by the Oracle database. Thus, any unicode strings in `NCHAR` or `NVARCHAR2` columns passed through to an Oracle database must be passed as an argument within the `UNISTR()` function without the `u` character.

    The following example inserts the unicode values `'0063'` and `'0064'`, which are the `a` and `b` characters respectively. Since we are using `PassThrough`=3, this statement is performed on the Oracle database; thus, we do not provide the `u` character as we would if this was performed on TimesTen.

    ```
    Command> INSERT INTO my_tab VALUES (UNISTR(n'\0063\0064'));
    1 row inserted.
    ```

PL/SQL Constructs

TimesTen supports a subset of stored procedure constructs, functions, data types, packages and package bodies that are supported by Oracle Database.

See Overview of PL/SQL Features in the *Oracle TimesTen In-Memory Database PL/SQL Developer's Guide*.

Mappings Between Oracle Database and TimesTen Data Types

When you choose data types for columns in the TimesTen cache tables, consider the data types of the columns in the Oracle Database tables and choose an equivalent or compatible data type for the columns in the cache tables.

Note

TimeTen cache, including passthrough, does not support the Oracle Database `ROWID` data type. However, you can cast a `ROWID` data type to a `CHAR(18)` when provided on the `SELECT` list in a SQL query.

The following example demonstrates the error that is returned when you do not cast the `ROWID` data type. Then, the example shows the correct casting of a `ROWID` data type to `CHAR(18)`:

```
Command> SET PASSTHROUGH 3;
Passthrough command has set autocommit off.
Command> SELECT ROWID FROM dual;
 5115: Unsupported type mapping for column ROWID
The command failed.
Command> SELECT CAST (ROWID AS CHAR(18)) FROM DUAL;
< AAAAB0AABAAAAEoAAA >
1 row found.
```

Primary and foreign key columns are distinguished from non-key columns. The data type mappings allowed for key columns in a cache table are shown in Table C-2.

Table C-2 Data Type Mappings Allowed for Key Columns

Oracle Database Data Type	TimesTen Data Type
`NUMBER(`*p*,*s*`)`	`NUMBER(`*p*`,` *s*`)` **Note:** `DECIMAL(`*p*`,` *s*`)` or `NUMERIC(`*p*`,` *s*`)` can also be used. They are aliases for `NUMBER(`*p*`,` *s*`)`.

`NUMBER(`*p*`,0)`
`INTEGER`
	`TT_TINYINT`
	`TT_SMALLINT`
	`TT_INTEGER`
	`TT_BIGINT`
	`NUMBER(`*p*`,0)`

`NUMBER`	`TT_TINYINT`
	`TT_SMALLINT`
	`TT_INTEGER`
	`TT_BIGINT`
	`NUMBER`

`CHAR(`*n*`)`	`CHAR(`*n*`)`
`VARCHAR2(`*n*`)`	`VARCHAR2(`*n*`)`

Table C-2 (Cont.) Data Type Mappings Allowed for Key Columns

Oracle Database Data Type	TimesTen Data Type
RAW(*n*)	VARBINARY(*n*)
DATE	DATE
TIMESTAMP(*n*)	TIMESTAMP(*n*)
NCHAR(*n*)	NCHAR(*n*)
NVARCHAR2(*n*)	NVARCHAR2(*n*)

Table C-3 shows the data type mappings allowed for non-key columns in a cache table.

Table C-3 Data Type Mappings Allowed for Non-Key Columns

Oracle Database Data Type	TimesTen Data Type
NUMBER(*p*,*s*)	NUMBER(*p*,*s*) REAL FLOAT BINARY_FLOAT DOUBLE BINARY_DOUBLE
NUMBER(*p*,0) INTEGER	TT_TINYINT TT_SMALLINT TT_INTEGER TT_BIGINT NUMBER(*p*,0) FLOAT BINARY_FLOAT DOUBLE BINARY_DOUBLE
NUMBER	TT_TINYINT TT_SMALLINT TT_INTEGER TT_BIGINT NUMBER REAL FLOAT BINARY_FLOAT DOUBLE BINARY_DOUBLE
CHAR(*n*)	CHAR(*n*)
VARCHAR2(*n*)	VARCHAR2(*n*)
RAW(*n*)	VARBINARY(*n*)

Table C-3 (Cont.) Data Type Mappings Allowed for Non-Key Columns

Oracle Database Data Type	TimesTen Data Type
`LONG`	`VARCHAR2(`*n*`)` Where *n* can be any valid value within the range defined for the `VARCHAR2` data type.
`LONG RAW`	`VARBINARY(`*n*`)` Where *n* can be any valid value within the range defined for the `VARBINARY` data type.

`DATE`	`DATE`
	`TIMESTAMP(0)`

`TIMESTAMP(`*n*`)`	`TIMESTAMP(`*n*`)`

`FLOAT(`*n*`)`
Note: Includes `DOUBLE` and `FLOAT`, which are equivalent to `FLOAT(126)`. Also includes `REAL`, which is equivalent to `FLOAT(63)`.
	`FLOAT(`*n*`)`
	`BINARY_DOUBLE`
	Note: `FLOAT(126)` can be declared as `DOUBLE`. `FLOAT(63)` can be declared as `REAL`.

`BINARY_FLOAT`	`BINARY_FLOAT`
`BINARY_DOUBLE`	`BINARY_DOUBLE`
`NCHAR(`*n*`)`	`NCHAR(`*n*`)`
`NVARCHAR2(`*n*`)`	`NVARCHAR2(`*n*`)`
`CLOB`	`VARCHAR2(`*n*`)` Where 1 <= *n* <= 4 MB.
`BLOB`	`VARBINARY(`*n*`)` Where 1 <= *n* <= 4 MB.
`NCLOB`	`NVARCHAR2(`*n*`)` Where 1 <= *n* <= 2 MB.

`JSON`	`JSON`
	`VARCHAR2(`*n*`)`
	Where 1 <= *n* <= 4 MB.