Oracle TimesTen In-Memory Database 7.0 Good Practices Guide

An Oracle White Paper
July 2007

Oracle TimesTen In-Memory Database 7.0
Good Practices Guide

Oracle TimesTen In-Memory Database 7.0
Good Practices Guide

1 Introduction

This document describes how to develop an application with Oracle TimesTen In-Memory Database for maximum performance and robustness. This document assumes the reader is somewhat familiar with the basic functionality of the Oracle TimesTen In-Memory Database and is in the process of planning or developing an application on TimesTen. This document is a supplement to the Oracle TimesTen product documentation set.

The following sections describe a few programming concepts and terminology for a TimesTen application.

1.1 TimesTen direct-linked application

The term "direct-linked application" denotes an application that is connecting to the TimesTen database using direct-linked connection mode (i.e. the TimesTen database is embedded within the application).

The term "direct-linked application" denotes an application that is connecting to the TimesTen database using "direct-linked connection mode" (i.e. the TimesTen database is embedded within the application).

With direct-linked mode, C and C++ applications explicitly link in one of the following shared libraries:

- `libtten.so` (Solaris, Linux)
- `libtten.sl` (HP-UX)
- `tten70.lib` (Windows)

For Java applications using the direct-linked mode, the TimesTen data source connection string is of the form "jdbc:TimesTen:direct:…"

1.2 TimesTen client/server application

This term denotes an application that uses the TimesTen client/server interface to connect to the TimesTen database.

With client/server mode, C and C++ application explicitly link in one of the following shared libraries:

- `libttclient.so` (Solaris, Linux)
- `libttclient.sl` (HP-UX)
- `ttcl70.lib` (Windows)

For Java applications using the client/server connection mode, the TimesTen data source connection string is of the form `"jdbc:TimesTen:client:…"`

Client/server applications can reside on the same machine as the TimesTen database or reside on a different machine. Since IPC and network overhead are involved when using the client/server connection mode, applications experience longer response times and lower throughput compared to direct-linked applications.

1.3 Installation directory install_dir

This is the directory where you installed TimesTen. Refer to the *Oracle TimesTen In-Memory Database Installation Guide* for more information.

By default, this directory is:

- `/opt/TimesTen/`*InstanceName* (UNIX; instance installed by root)
- `$HOME/TimesTen/`*InstanceName* (UNIX; instance installed by non-root user)
- `C:\TimesTen\tt70` (Windows)

TimesTen instances can be installed by root or non-root users depending on your operation environment.

1.4 TTClasses library for C++ users

The TimesTen C++ interface classes, known as TTClasses, were written to provide an easy-to-use, high performance interface to the TimesTen database.

The TimesTen C++ interface classes, known as TTClasses, provide an easy-to-use, high performance interface to the TimesTen database. The C++ class library provides "wrappers" around the most common ODBC functionality including:

- SQL query execution
- Event notification (XLA)
- System catalog information.

In addition, the TTClasses design incorporates many of the recommended practices covered in this document. TTClasses is shipped with the Oracle TimesTen In-Memory Database. For a description of the TTClasses interface, see the *TimesTen In-Memory Database TTClasses Guide*.

The TimesTen product installation includes many useful sample programs that demonstrate how to use TTClasses. Among the example programs are programs that demonstrate:

- How to monitor and administer a TimesTen data store
- How to implement an event notification program using TimesTen's XLA
- How to estimate the data store's size
- How to administer XLA bookmarks

2 Maximizing performance

This section recommends coding practices for a TimesTen application to achieve optimal performance. The information in this section should be used in conjunction with the chapter “Data Store Performance Tuning” in the *Oracle TimesTen In-Memory Database Operations Guide*.

2.1 Use a direct-linked connection for response-time critical application

Direct-linked applications can achieve significantly better performance than client/server applications because database operations are executed directly from the application process' address space without incurring inter-process communication and network roundtrip overhead.

If you have control on how your application accesses the TimesTen database and if you can run your application on the same machine as the TimesTen database, we recommend that you use the direct-linked connection mode. Direct-linked applications can achieve significantly better performance than client/server applications because database operations are executed directly from the application process' address space without incurring inter-process communication and network roundtrip overhead.

TimesTen supports multi-process and multi-threaded applications, a TimesTen database can be shared across processes that are using direct-linked and client/server connection modes.

For Java applications that utilize JDBC to access the TimesTen database, it is simple to use direct-linked connection mode. Just change your JDBC connection string to the form of "jdbc:TimesTen:direct:…"

For applications accessing data cached from the Oracle Database, the use of a direct-linked connection improves your application's performance the same way it does for TimesTen data that is not cached from the Oracle Database.

For applications accessing data that are cached from the Oracle Database, the use of a direct-linked connection improves the application performance the same way it does to data that's not cached from the Oracle Database.

If your application cannot use a direct-linked connection, the use of a client/server connection still provides significantly higher performance than accessing a conventional disk-based relational database. This is primarily due to the TimesTen product being designed and optimized for in-memory data structure and layout.

As a general guideline, applications should consider connecting to a TimesTen data store in client/server mode in the following situations:

- If the application must run on a different host from the host where the TimesTen database resides.
- If a 32-bit client application needs to connect to a 64-bit TimesTen database and the 32-bit client application cannot be recompiled in 64-bit. If the application is or can be relocated to the same host as the TimesTen data store, but cannot be recompiled in 64-bit mode, we recommend using the TimesTen shared memory inter-process communications (SHMIPC). SHMIPC provides significant performance improvements over TCP/IP for client/server database connectivity.

If you design a system that requires a large number of TimesTen client/server connections, you should be aware that each data store connection consumes operating system resources on the server host. This factor should be included in the sizing and operating system tuning of the server host.

2.2 Prepare all SQL statements in advance

For best performance, you should prepare all SQL statements that are executed more than once in advance.

For best performance, you should prepare all SQL statements that are executed more than once in advance. This is true for all relational databases, but for TimesTen and its extremely fast transaction rates, the time spent to compile a statement can actually take several times longer than it takes to execute it.

In addition to preparing statements in advance, input parameters and output columns for those statements should be bound in advance:

- ODBC users use the `SQLPrepare` function to prepare statements in advance.
- TTClasses users use the `TTCmd::Prepare` method to prepare statements in advance.

- JDBC users use the `PreparedStatement` class to prepare statements in advance.

Another best practice is to prepare your queries after you have updated statistics on your tables.

2.3 Update statistics on all tables

The TimesTen query optimizer generally chooses good query plans. However, the optimizer needs additional information about the tables involved in complex queries in order to choose good plans.

Table statistics are an essential aid to the optimizer in choosing a good plan. By knowing the number of rows and data distribution of column values for a table, the optimizer has a much better chance of choosing an efficient query plan to access that table.

It is generally a good idea to update statistics on all tables in your database before preparing queries that access those tables.

It is generally a good idea to update statistics on all tables in your database before preparing queries that access those tables.

If you update statistics on your tables before they are populated, then the queries are optimized with the assumption that the tables contain no rows (or very few rows). If you later populate your tables with millions of rows and then execute the queries, the plans that worked well for tables containing few rows may now be very slow as the optimization was not done for the current set of data in the table.

For optimal performance, you should update statistics on your tables to reflect the current population in the tables, in particular, after you have added or deleted a significant number of rows in the tables. You should prepare your queries after you have updated the table statistics.

For more information about updating statistics, consult the descriptions of the built-in procedures **ttOptUpdateStats** and **ttOptEstimateStats** in the "Built-In Procedures" chapter of the *Oracle TimesTen In-Memory Database API Reference Guide* (tt_ref.pdf).

2.4 Tune your connections and use connection pooling

If your application is multithreaded and opens multiple connections to the same TimesTen database, you should pay attention to and manage your connections carefully. As an in-memory database, TimesTen uses available processor resources efficiently and is, in general, CPU-bound (as supposed to disk-bound for many applications running on disk-based RDBMS). In general, it is difficult to achieve best performance if there are more concurrent connections active simultaneously than there are CPUs on your machine to handle the operations, and concurrent connections may result in lower application throughput.

If your application requires a lot of connections, you should ensure that connections are held efficiently for the transactions. One approach for avoiding this problem is to use connection pooling. A good example of connection pooling is the `TTConnectionPool` class in TTClasses. Also, commercial J2EE application servers typically use connection pooling by default.

2.5 Close cursors promptly

You do not need to commit TimesTen read-only transactions. However, it is important to close read-only cursors promptly in order to release all resources held by a read-only SQL query.

Similar to the Oracle Database, you do not need to commit TimesTen read-only transactions. However, it is important to close read-only cursors promptly in order to release all resources (such as temporary space used for a sort operation) held by a read-only SQL query.

The following methods close a SQL cursor:

- For ODBD, use SQLFreeStmt(SQL_CLOSE)
- For TTClasses, use **TTCmd::Close**
- For JDBC, use **PreparedStatement.close**

2.6 Control the frequency of disk writes

Most applications require some level of data persistence and transaction durability for recoverability and system restart purposes. For example, if the system crashes because of hardware or software failure, your applications may require that recent updates to the database be recoverable. There are several ways to configure the TimesTen database using the **DurableCommit** attribute:

DurableCommit = 0 provides the best transaction response time since control is returned to the application as soon as the transaction is committed to the in-memory log buffer.

- The first option is to allow the TimesTen transaction log manager to flush the in-memory transaction log buffer to disk asynchronously from the application's transaction commit calls. The transaction log manager has a background log flusher whose job is to persist the transaction log data to disk in an efficient manner. This deferred durability option is achieved by setting the connection attribute **DurableCommit = 0**. This commit setting provides the best transaction response time since control is returned to the application as soon as the transaction is committed to the in-memory log buffer. However, applications that take advantage of this feature do incur some level of vulnerability in the event of a system failure. The vulnerability is no worse than the in-memory transaction log buffer size worth of transactions.

DurableCommit = 1 causes every transaction commit in that connection to flush the transaction log to disk; as a result, your application has complete durability.

- The second option is to connect to the TimesTen database with the connection attribute setting **DurableCommit = 1**. This causes every transaction commit in that connection to flush the transaction log to disk. As a result, your application has complete durability. However, this setting can cause the application's write throughput to be significantly reduced because the application is blocked for each transaction commit waiting for the transaction data to be written to disk before control is returned to the application. Higher throughput can be achieved if the application can be configured to have lots of connections to achieve concurrent writes to the database.

TimesTen offers a flexible approach toward durability control. An application may control the durability setting at the connection level and/or at the transaction level. The application can call the **ttDurableCommit** built-in procedure at any time to "flush" the transaction data to disk. For some applications it makes sense to call **ttDurableCommit** at specific time intervals to guarantee the data vulnerability to a fixed amount of time.

If your application is required to guarantee no data loss in the event of a single point of failure, you may consider an alternative solution to setting **DurableCommit** to 1. See discussion in the next section.

TimesTen Replication and Cache Connect to Oracle product options can be used to provide additional transaction durability and availability, regardless of the **DurableCommit** setting chosen.

2.7 Alternative solution for DurableCommit = 1

Some applications are required to guarantee no data loss in the event of a single point of failure. However, if your application durably commits every transaction, you may not achieve your throughput and response time goals since you are at the mercy of how fast your disk can finish the I/O for every transaction. When your transaction path involves disk I/O, the application will likely suffer inconsistent response times during peak transaction load. If your business requires that you meet some service-level agreement (SLA) with a predefined application response time, you may need an alternative solution to setting **DurableCommit = 1**.

As an alternative to using durable commits, an application can use 2-SAFE Replication between two TimesTen nodes to ensure no data loss.

As an alternative to using durable commits, you can use 2-SAFE Replication between two TimesTen nodes. With 2-SAFE, your transactions are guaranteed to be committed in-memory on both the primary and the standby nodes before control is returned to the application, thus ensuring no data loss in the event of a single point of failure. In addition to the data integrity guarantee, your application also benefits from the consistent response time that cannot be achieved if there is disk I/O in the transaction path.

Refer to the *TimesTen to TimesTen Replication Guide* for detailed information on how to configure and use 2-SAFE Replication.

2.8 Create the right Indexes for your tables

Knowing how many indexes to create for good database performance is a bit tricky. If you create too few indexes, some of your frequent data operations perform more slowly than usual. If you create too many indexes, the INSERT/UPDATE/DELETE operations may take longer because of the extra time needed to update the indexes.

There are a few design decisions to consider when designing your TimesTen table and index schema.

TimesTen supports two types of indexes: hash indexes and T-tree indexes.

- TimesTen supports two types of indexes: hash index and T-tree index. A well-tuned hash index is faster than the corresponding T-tree index for exact match lookups, but hash indexes cannot be used for range queries (T-tree indexes can be used for both exact match and range lookups, and for sorts, such as for SQL queries involving ORDER BY, GROUP BY or DISTINCT).
- If you don't specify a unique hash index on your primary key, a T-tree index is automatically created for you (this is true for TimesTen 7.0 or later releases).
- If you specify a unique hash index, be sure to set the index with appropriate size. Use the "PAGES=" option of the CREATE TABLE statement to specify the expected size of the table. Divide the number of rows expected in the table by 256 for the number of pages to specify. Example:

```
CREATE TABLE EMP (
       ID     NUMBER NOT NULL PRIMARY KEY,
       NAME   VARCHAR2(100)
) UNIQUE HASH ON (ID) PAGES=500;
```

Hash indexes provide better performance over T-tree indexes for equality predicates in cases where either one can be used.

- Hash indexes provide better performance over T-tree indexes for equality predicates in cases where either index can be used. However, hash indexes require more space than T-tree indexes. Specifying too many pages for the index wastes space. Specifying too few pages hurts the performance because of hash buckets overflow. Hash indexes that are in-appropriately sized can be much slower than a tree index.
- If your table content increased significantly, you should resize your hash index; this is done with the `ALTER TABLE` statement.

- Full table scan performance over a table with zero T-tree indexes can be improved by including a T-tree index (any T-tree index), even if the table scan does not reference columns in that index. This is not intuitive but is easily demonstrated. Thus, you should consider building at least one T-tree index for every table referenced by your application's full table scans.

Let's go over a few different indexing scenarios using the simple query example:

```
        SELECT ... FROM T1 WHERE COL1 = ? AND COL2 = ?
```

Scenario 1:

There are two indexes created on T1:

- Hash index: (COL1, COL2)
- T-tree index: (COL1, COL2, COL3)

In this case, both indexes can be used to answer this query.

The hash index can be used because the columns in the WHERE clause exactly match the columns in the hash index. The T-tree index can also be used to answer the query because the columns in the WHERE clause are the leading prefix (the first two) of the columns in the index.

The TimesTen optimizer chooses the hash index because it is faster.

Scenario 2:

There are two indexes created on T1:

- Hash index: (COL1, COL2, COL3)
- T-tree index: (COL1, COL2)

In this case, only the T-tree index can be used to answer the query.

The hash index cannot be used because one of the columns in the hash index (COL3) is not specified in the WHERE clause of the query.

The T-tree indexed columns exactly match the columns in the WHERE clause of the query.

The TimesTen optimizer chooses the T-tree index.

Scenario 3:

There are two indexes on T1:

- Hash index: (COL1)
- T-tree index: (COL3, COL1, COL2)

In this case, only the hash index can be used to answer the query.

The hash index can be used because all of its columns are contained in the WHERE clause of the query. There are additional columns in the WHERE clause of the query, so every row read from the hash index will have the "COL2 = ?" predicate applied before the query result is returned.

The T-tree index cannot be used because the T-tree is first sorted by COL3, and the query does not refer to COL3.

The TimesTen optimizer chooses the hash index.

Scenario 4:

There are two indexes on T1:

Hash index: (COL1, COL2, COL3)

T-tree index: (COL3, COL1, COL2)

In this case, neither index can be efficiently used to answer the query, although the T-tree index can be used for a full table scan.

The hash index cannot be used for the same reason as in Scenario 2. The T-tree index cannot be used because the columns in the query (COL1, COL2) do not constitute a leading prefix of the index.

Since neither index can be used, the SQL engine must perform a table scan to answer the query, and as a result the query's performance is poor. A temporary index may be created if necessary, but performance-wise, it is still poor.

You should choose carefully which indexes should be created based on the frequency of the queries that you perform.

As you can see from these examples, you should choose carefully which indexes should be created based on the frequency of the queries that you perform.

For more information about creating the correct indexes, see the sections on “Tuning statements and Use Indexes” and “Select hash or T-tree indexes appropriately” in the *Oracle TimesTen In-Memory Database Operations Guide.*

2.9 Review the query plan

If you find that a specific query runs more slowly than expected, it is possible that the TimesTen optimizer is not choosing the optimal query plan to answer that query. You should generate the query plan and review it.

If you find that a specific query runs more slowly than expected, it is possible that the TimesTen optimizer is not choosing the optimal query plan to answer that query. You should generate the query plan and review it. The *Oracle TimesTen In-Memory Database Operations Guide* has detailed documentation on how to generate a query plan and how to view the plan.

To display the optimizer plan from the **ttIsql** command line utility, use the commands:

```
autocommit 0;
showplan 1;
```

When reviewing the query plan, pay attention to the predicates that are participating in the query evaluation but are not indexed. If you can create an index for the non-indexed predicates, it helps your query performance. Let’s review an example on how a simple change can provide a huge benefit to your query performance.

In the example below, we extracted one of the steps in a test query where the main table (`SHIPMENT`) is relatively big. It has over 10 million rows. When we first ran the query, it took about 800 seconds to complete, which was too slow and not what was expected. We then generated the query plan and saw that one of the steps had the following plan::

```
STEP:     10
LEVEL:     8
OPERATION:  TblLkTtreeScan
TBLNAME:   SHIPMENT
IXNAME:   SHIPMENT_IDX1
INDEXED:   <NULL>
NOT INDEXED:  TBL1.OB_FLG <> 'Y'
             AND TBL1.SHIPMENT_ID = CLIENT.SHIPMENT_ID
             AND TBL1.SHIPMENT_QUAL =
                                 CLIENT.SHIPMENT_QUAL
             AND TBL1.CARRIER = CLIENT.CARRIER
             AND TBL1.ROLE = 'Y'
```

Several predicates were listed in the “`NOT INDEXED`” line, and in fact, there were no predicates listed in the “`INDEXED`” line. We did not have an index for the `SHIPMENT_ID, SHIPMENT_QUAL` and the `CARRIER` columns and the optimizer had to scan the entire table to evaluate each of the predicates. We fixed the issue by creating an index for those 3 columns, and we got a much improved plan for step 10 as shown below:

```
 STEP:    10
 LEVEL:    8
 OPERATION:  RowLkTtreeScan
 TBLNAME:   SHIPMENT
 IXNAME:   SHIPMENT_IDX0
 INDEXED:   TBL1.SHIPMENT_ID = CLIENT.SHIPMENT_ID
             AND TBL1.SHIPMENT_QUAL =
CLIENT.SHIPMENT_QUAL
             AND TBL1.CARRIER = CLIENT.CARRIER
 NOT-INDEXED:  TBL1.OB_FLG <> 'Y' AND TBL1.ROLE = 'Y'
```

This is a much better plan. The optimizer chooses the row lock with the appropriate index to evaluate the predicates, resulting in a 400 times improvement in the query time. The modified query ran for 2 seconds (instead of 800 seconds prior to the change).

As with all performance tuning, your mileage varies. The important point here is that you should take the time to read the query plan and make necessary changes for better query performance.

2.10 Turn AUTOCOMMIT OFF and commit regularly

By turning `autocommit` off and explicitly committing multiple SQL statements in a single transaction an application's performance can be greatly improved.

By default in ODBC and JDBC, all connections to the database have *`autocommit`* turned on. This means that each individual SQL statement is committed in its own transaction. By turning *`autocommit`* off and explicitly committing multiple SQL statements in a single transaction, an application's performance can be greatly improved. This makes particular sense for large operations, such as bulk inserts or bulk deletes. (TTClasses turns off autocommit as by default.)

However, it is also possible to group too many operations into a single transaction where locks are held for a much longer time for each transaction, resulting in much less concurrency in the system. In general, bulk insert, update and delete operations tend to work most effectively when you commit every few thousand rows.

2.11 Use XLA acknowledgements efficiently

The XLA acknowledgement interface is designed to ensure that an application not only receives a message but successfully processes it. Acknowledging an update resets the application's XLA bookmark to the last record that was read. This prevents previously returned records from being read again, ensuring that an application does not receive previously acknowledged records if the bookmark is reused when an application reconnects to XLA.

JMS/XLA can automatically acknowledge XLA update messages, or applications can choose to acknowledge messages explicitly. You specify how updates should be acknowledged when you create the *`Session`*. JMS/XLA supports three acknowledgement modes:

- `AUTO_ACKNOWLEDGE` – Updates are automatically acknowledged when they are received. Each message is delivered only once. In `AUTO_ACKNOWLEDGE` mode, JMS/XLA does not prefetch multiple records. (The *`xlaprefetch`* attribute in the topic is ignored.)
- `DUPS_OK_ACKNOWLEDGE` – Updates are automatically acknowledged, but duplicate messages may be delivered if the application fails. JMS/XLA prefetches records according to the *`xlaprefetch`* attribute specified for the topic and sends an acknowledgement when the last record in a prefetched block is read. If the application fails before reading all of the prefetched records, then all of the records in the block are presented to the application when it restarts.

`CLIENT_ACKNOWLEDGE`--Applications are responsible for acknowledging receipt of update messages by calling `acknowledge` on the `MapMessage`. JMS/XLA prefetches records according to the `xlapbrefetch` attribute specified for the topic.

2.11.1 Prefetch updates

Prefetching multiple update records simultaneously is more efficient than obtaining each update record from XLA individually.

Prefetching multiple update records simultaneously is more efficient than obtaining each update record from XLA individually. If possible, you should design your application to tolerate duplicate updates so you can use DUPS_OK_ACKNOWLEDGE or explicitly acknowledge updates.

2.11.2 Acknowledge updates

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

2.12 Considerations for Java applications

TimesTen's native API is ODBC. C/C++ applications that use the TimesTen ODBC driver experience somewhat better performance compared with Java applications that use the TimesTen JDBC driver. That said, an application using the TimesTen direct-linked JDBC driver incurs no overhead for inter-process communication and network roundtrips, and is therefore significantly faster than most other JDBC drivers used in client/server connection mode.

The JDBC performance is influenced by the types of queries executed (SELECT vs. UPDATE/INSERT/DELETE) and the transaction mix (reads vs. writes).

JDBC performance is influenced by the types of queries executed (SELECT vs. UPDATE/INSERT/DELETE) and the transaction mix (reads vs. writes). The factors affecting performance include:

- The number of bound parameters for prepared statements: Increasing the number of bound parameters tends to decrease Java application performance.
- The number of columns returned by a SELECT statement: Increasing the number of select columns tends to decrease Java application performance.
- The types of parameters and columns: Data types consisting of a larger number of bytes are typically slower in JDBC.

Fundamentally, a Java application has poorer performance when the amount of data that is transferred between the application and the database increases.

There are other factors inherit in Java that could influence the performance of a Java application:

- JVM garbage collection: Java application can experience response time spikes during JVM garbage collection. This garbage collection can make it difficult to guarantee predictable response times for high performance "real-time" applications. Careful tuning of JVM garbage collection is essential to minimize potential throughput dips and response time spikes.
- Java's runtime memory management: The design of the Java runtime memory management subsystem can often result in hidden data copying. Additionally, you should ensure that there is sufficient Java heap space for the application.
- Depending on your application requirements for throughput and scalability, you may need to experiment with performance tuning of a heavily multi-threaded Java application versus using multiple JVMs.

Certain JVMs are better tuned for specific platforms. Evaluate your options by running different JVMs on your platform of choice.

Use the MAXROWS option for SELECT statements when returning large result sets.

2.13 Create indexes after loading large tables

You can minimize the time it takes to load data by creating T-tree indexes after loading the data.

If your application design allows it, you can minimize the time it takes to load data by creating T-tree indexes after loading the data. In this situation, the sequence of operations should be:

1. Load data into tables. (If needed, you may disable logging and use database-level locking to achieve better performance during the table load operation.)
2. Create T-tree indexes.
3. Update statistics.
4. Prepare queries.

This is relevant only for large data load operations (millions of rows).

3 Concurrency and scalability tuning

This section provides tuning tips for applications running on SMP machines to achieve optimal application response time and throughput using Oracle TimesTen.

3.1 Configure memory log buffer size correctly

The TimesTen transaction log buffer size has a default size of 64MB. This can be tuned to suit your application needs. If your application is write intensive, creating a bigger memory log buffer reduces log contention and avoids log buffer waits. Monitor the `LOG_BUFFER_WAIT` value in the system `MONITOR` table and increase the log buffer size if needed. Refer to the "System and Data Store Tuning" section in the *Oracle TimesTen In-Memory Database Operations Guide* for more details.

3.2 Separate transaction logs from checkpoint files

For write intensive applications, put your transaction logs and checkpoint files on separate disks to reduce I/O contention.

For write intensive applications, put your transaction logs and checkpoint files on separate disks to reduce I/O contention. This is particularly important if you need to have consistent response times and throughput for your application requests. If your checkpoint files are on the same disk spindle, the checkpoint operation competes with the transaction logging activities going to the same disk.

3.2.1 Disk speed matters

Faster disks play a role in your application update throughput. Keep in mind that log records are generated as a result of your application executing INSERT, UPDATE and DELETE operations. Log records are also generated if you are caching read-only data from the Oracle Database with AutoRefresh. When the Cache Connect agent refreshes the TimesTen database with updates from the Oracle Database, the refresh operations generate log records as the updates are applied to the TimesTen cache tables. Faster disks allow the TimesTen log manager to complete the write to disk faster, thus reducing the possibility that the application is waiting for the log buffer to be available. You can monitor `LOG_BUFFER_WAIT` in the system `MONITOR` table for I/O contention.

3.3 Use the appropriate RAM policy

By default, the TimesTen database is unloaded from memory when the last connection to the database disconnects. Unloading and reloading the database frequently can increase connection overhead unnecessarily. To ensure that the database does not get unloaded and reloaded frequently, you can use the **ttAdmin** utility to set the RAM policy to keep the database loaded even after the last connection exits. Refer to the "Avoid Connection Overhead" section in the *Oracle TimesTen In-Memory Database Operations Guide*.

Deleting a very large number of records in a single transaction (or single SQL statement) can have negative impact to the overall system performance.

3.4 Avoid large delete statements

Deleting a very large number of records in a single transaction (or single SQL statement) can have a negative impact to overall system performance:

- Large delete operations are I/O intensive because they generate lots of log records. Every deleted record needs to be logged in case you need to roll back the transaction.
- Large delete transactions slow down other concurrent operations in the system
- If the target table is also a table being replicated, the transaction log data needs to be transmitted to the subscriber nodes because TimesTen Replication uses physical replication by transmitting committed transactions to the subscriber nodes.

Instead of using DELETE FROM, consider using TRUNCATE TABLE or DELETE FIRST.

Instead of using `DELETE FROM`, consider the following alternatives to reduce resource consumption:

- Use `TRUNCATE TABLE` if you need to delete all records in a table
- Use `DELETE FIRST` to break down the number of records in a transaction to allow more concurrency in the system.

3.5 Shorten long transactions

Long running transactions can cause other operations to fail with lock timeout errors because resources acquired for the long transaction remain locked for a long period of time. Long transactions, in general, reduce overall system concurrency and application performance.

4 Maximizing stability and high availability

Operational and system planning is important for all database management systems, including TimesTen. This section describes several important operations you should follow to manage your TimesTen database better.

4.1 Perform regular checkpoints

If checkpoint operations are not performed regularly, database recovery can take longer.

A checkpoint file contains a snapshot of the TimesTen in-memory database image on disk. Checkpoint files are used to reload an existing TimesTen database into memory. If checkpoint operations are not performed regularly, database recovery can take longer because the transactions that did not make it to the checkpoint files need to be applied from the transaction log files. The more transactions that need to be applied from the logs, the longer it takes to reload the database into memory. Under normal system shutdown or database unload operations, TimesTen performs a "final" checkpoint operation to capture the current database image to disk. Restarting the database from a "final" checkpoint file is the fastest way to reload an existing TimesTen database.

The default setting for checkpoint frequency is every 10 minutes (600 seconds). You can alter the checkpoint frequency using the **ttCkptConfig** built-in procedure or the **CkptFrequency** connection attribute. In addition to the time frequency, you can also configure checkpoints using the log volume. Using log volume for checkpoint frequency is ideal if your application has frequent bursts of activities. If you have intimate knowledge of your application workload and transaction rate, you may tune the checkpoint frequency to suit your needs. Otherwise, start with the default setting for checkpoint frequency.

4.2 Avoid transaction log accumulation

You should monitor your transaction logs to make sure that log files are being purged regularly to avoid accumulation of a large number of log files. Transaction log files can accumulate if any of the following situations are true:

- No checkpoint operations have occurred for a long time. Transaction logs cannot be purged until they have been captured in both of the checkpoint files.
- Transactions are held by Replication. If Replication is behind in replicating the transactions to its subscriber nodes, the transaction logs cannot be purged until the transactions have been replicated to the remote node(s)
- Transactions are held by XLA applications. If XLA bookmarks are not moved by the application, the transaction logs cannot be purged.

If after regular checkpoint operations, the log files are not purged, you can find out who is holding the logs by running the **ttLogHolds** built-in procedure. Refer to the *Oracle TimesTen In-Memory Database API Reference Guide* (tt_ref.pdf), for more details.

4.3 Backup your database

Just like managing any other database, doing regular backups of your TimesTen database is a good practice for handling database recovery due to unexpected failures or human errors. The TimesTen **ttBackup** utility provides the ability to backup your TimesTen database. The **ttRestore** utility does the restoration of the TimesTen database from the backup file.

4.4 Plan for high availability

If data availability is one of the business requirements for your application, you should consider using TimesTen Replication.

If data availability is one of the business requirements for your application, you should consider using the TimesTen Replication product option to provide you the ability to handle single point of failures with a standby replica of your production database. Refer to the *TimesTen to TimesTen Replication Guide* (replication.pdf).

Oracle TimesTen 7.0 Good Practices Guide
July 2007
Author: TimesTen Development Team

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
oracle.com