

**Oracle® TimesTen In-Memory Database**

PL/SQL Developer's Guide

Release 18.1

**E61201-06**

July 2020

Copyright © 1996, 2020, Oracle and/or its affiliates.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs (including any operating system, integrated software, any programs embedded, installed or activated on delivered hardware, and modifications of such programs) and Oracle computer documentation or other Oracle data delivered to or accessed by U.S. Government end users are "commercial computer software" or "commercial computer software documentation" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, reproduction, duplication, release, display, disclosure, modification, preparation of derivative works, and/or adaptation of i) Oracle programs (including any operating system, integrated software, any programs embedded, installed or activated on delivered hardware, and modifications of such programs), ii) Oracle computer documentation and/or iii) other Oracle data, is subject to the rights and limitations specified in the license contained in the applicable contract. The terms governing the U.S. Government's use of Oracle cloud services are defined by the applicable contract for such services. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Inside are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Epyc, and the AMD logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

---

---

# Contents

<b>Preface</b> .....	vii
Audience .....	viii
Related documents .....	viii
Conventions .....	viii
Documentation Accessibility .....	ix
<b>What's New</b> .....	xi
New features in Release 18.1.1.1.0 .....	xi
<b>1 Introduction to PL/SQL in TimesTen</b>	
<b>Features of PL/SQL in TimesTen</b> .....	1-1
<b>TimesTen PL/SQL components and operations</b> .....	1-2
Application interaction with TimesTen and PL/SQL .....	1-2
PL/SQL in TimesTen versus PL/SQL in Oracle Database .....	1-3
SQL statements in PL/SQL blocks .....	1-3
Execution of PL/SQL from SQL .....	1-4
<b>Audiences for this document</b> .....	1-4
Developers experienced with Oracle Database and Oracle Database PL/SQL .....	1-5
Developers experienced with TimesTen .....	1-5
<b>TimesTen Quick Start and sample applications</b> .....	1-5
<b>2 Programming Features in PL/SQL in TimesTen</b>	
<b>PL/SQL blocks</b> .....	2-1
<b>PL/SQL variables and constants</b> .....	2-2
<b>SQL function calls from PL/SQL</b> .....	2-5
<b>PL/SQL control structures</b> .....	2-5
Conditional control .....	2-6
Iterative control .....	2-6
CONTINUE statement .....	2-7
<b>PL/SQL procedures and functions</b> .....	2-8
Creating procedures and functions .....	2-8
Executing procedures and functions .....	2-9
Using synonyms for procedures and functions .....	2-11
<b>PL/SQL packages</b> .....	2-12
Package concepts .....	2-13

Creating and using packages .....	2-13
Using synonyms for packages.....	2-16
<b>How to pass data between an application and PL/SQL.....</b>	<b>2-16</b>
Using bind variables from an application .....	2-17
IN, OUT, and IN OUT parameter modes .....	2-18
<b>Use of SQL in PL/SQL programs.....</b>	<b>2-18</b>
Static SQL in PL/SQL for queries and DML statements.....	2-19
Dynamic SQL in PL/SQL (EXECUTE IMMEDIATE statement) .....	2-20
FORALL and BULK COLLECT operations.....	2-22
RETURNING INTO clause .....	2-23
TimesTen PL/SQL with TimesTen Cache.....	2-23
Large objects (LOBs) .....	2-25
About LOBs.....	2-25
LOB locators.....	2-25
Temporary LOBs.....	2-26
Differences between TimesTen LOBs and Oracle Database LOBs .....	2-26
Using LOBs .....	2-27
PL/SQL package support for LOBs .....	2-28
Passthrough LOBs.....	2-28
<b>Use of cursors in PL/SQL programs.....</b>	<b>2-28</b>
<b>Wrapping PL/SQL source code.....</b>	<b>2-29</b>
<b>Differences in TimesTen: transaction behavior.....</b>	<b>2-31</b>

### 3 Data Types in PL/SQL in TimesTen

<b>Understanding the data type environments .....</b>	<b>3-1</b>
<b>Understanding and using PL/SQL data types .....</b>	<b>3-2</b>
PL/SQL data type categories .....	3-2
Predefined PL/SQL scalar data types.....	3-2
PLS_INTEGER and BINARY_INTEGER data types .....	3-3
SIMPLE_INTEGER data type.....	3-4
ROWID data type.....	3-4
LOB data types .....	3-4
PL/SQL composite data types .....	3-4
Using collections in PL/SQL.....	3-5
Using records in PL/SQL .....	3-5
Using associative arrays from applications .....	3-6
PL/SQL REF CURSORS .....	3-7
<b>Data type conversion .....</b>	<b>3-9</b>
Conversion between PL/SQL data types .....	3-9
Conversion between application data types and PL/SQL or SQL data types.....	3-9
<b>Differences in TimesTen: data type considerations.....</b>	<b>3-10</b>
Conversion between PL/SQL and TimesTen SQL data types .....	3-11
Date and timestamp formats: NLS_DATE_FORMAT and NLS_TIMESTAMP_FORMAT .....	3-12
Non-supported data types.....	3-13

### 4 Errors and Exception Handling

<b>Understanding exceptions .....</b>	<b>4-1</b>
---------------------------------------	------------

About exceptions.....	4-1
Exception types.....	4-2
<b>Trapping exceptions</b> .....	4-2
Trapping predefined TimesTen errors.....	4-3
Trapping user-defined exceptions.....	4-4
Using the RAISE statement .....	4-4
Using the RAISE_APPLICATION_ERROR procedure.....	4-5
<b>Retrying after transient errors (PL/SQL)</b> .....	4-6
<b>Showing errors in ttSql</b> .....	4-6
<b>Differences in TimesTen: exception handling and error behavior</b> .....	4-7
TimesTen PL/SQL transaction and rollback behavior for unhandled exceptions.....	4-8
TimesTen error messages and SQL codes .....	4-9
Warnings not visible in PL/SQL .....	4-9
Unsupported predefined errors.....	4-10
Possibility of runtime errors after clean compile (use of Oracle Database SQL parser).....	4-10
Use of TimesTen expressions at runtime.....	4-10

## 5 Examples Using TimesTen SQL in PL/SQL

<b>Examples using the SELECT...INTO statement in PL/SQL</b> .....	5-1
Using SELECT... INTO to return sum of salaries .....	5-1
Using SELECT...INTO to query another user's table.....	5-2
<b>Example using the INSERT statement</b> .....	5-2
<b>Examples using input and output parameters and bind variables</b> .....	5-3
Using IN and OUT parameters .....	5-3
Using IN OUT parameters.....	5-4
Using associative arrays.....	5-5
<b>Examples using cursors</b> .....	5-6
Fetching values.....	5-6
Using the %ROWCOUNT and %NOTFOUND attributes.....	5-8
Using cursor FOR loops .....	5-8
<b>Examples using FORALL and BULK COLLECT</b> .....	5-9
Using FORALL with SQL%BULK_ROWCOUNT .....	5-9
Using BULK COLLECT INTO with queries .....	5-10
Using BULK COLLECT INTO with cursors .....	5-11
Using SAVE EXCEPTIONS with BULK COLLECT.....	5-12
<b>Examples using EXECUTE IMMEDIATE</b> .....	5-13
Using EXECUTE IMMEDIATE to create a table .....	5-14
Using EXECUTE IMMEDIATE with a single row query.....	5-14
Using EXECUTE IMMEDIATE to alter a connection attribute.....	5-15
Using EXECUTE IMMEDIATE to call a TimesTen built-in procedure.....	5-16
Using EXECUTE IMMEDIATE with TimesTen-specific syntax .....	5-16
<b>Examples using RETURNING INTO</b> .....	5-16
Using the RETURNING INTO clause with a record .....	5-17
Using BULK COLLECT INTO with the RETURNING INTO clause .....	5-17
<b>Example querying a system view</b> .....	5-18

## 6 PL/SQL Environment

PL/SQL connection attributes .....	6-1
PL/SQL database configuration parameters .....	6-7
The ttSrcScan utility .....	6-7

## 7 TimesTen Supplied PL/SQL Packages

DBMS_LOB .....	7-2
DBMS_LOCK .....	7-4
DBMS_OUTPUT .....	7-5
DBMS_PREPROCESSOR .....	7-6
DBMS_RANDOM .....	7-7
DBMS_SQL .....	7-8
DBMS_UTILITY .....	7-10
TT_DB_VERSION .....	7-12
TT_STATS .....	7-13
UTL_FILE .....	7-14
UTL_IDENT .....	7-16
UTL_RAW .....	7-17
UTL_RECOMP .....	7-19

## 8 TimesTen PL/SQL Support: Reference Summary

### Index

---

---

# Preface

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

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

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

This document covers TimesTen support for PL/SQL.

The following topics are discussed in the preface:

- [Audience](#)
- [Related documents](#)
- [Conventions](#)
- [Documentation Accessibility](#)

## Audience

This document is intended for anyone developing or supporting applications that use PL/SQL with TimesTen. Although it provides some overview, you should be familiar with PL/SQL or have access to more detailed documentation. This manual emphasizes TimesTen-specific functionality.

You should also be familiar with TimesTen, SQL (Structured Query Language), and database operations.

You would typically use PL/SQL through some programming interface such as those mentioned above, so should also consult the appropriate TimesTen developer documentation.

Also see "[Audiences for this document](#)" on page 1-4, which goes into more detail.

## Related documents

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

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

In particular, these Oracle Database documents may be of interest:

- *Oracle Database PL/SQL Language Reference*
- *Oracle Database PL/SQL Packages and Types Reference*
- *Oracle Database SQL Language Reference*
- *Oracle Database Reference*

In addition, numerous third-party documents are available that describe PL/SQL in detail.

## Conventions

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

---

---

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

---

---

This document uses the following text conventions:

Convention	Meaning
<i>italic</i>	Italic type indicates terms defined in text, book titles, or emphasis.
monospace	Monospace type indicates commands, URLs, procedure and function names, package names, attribute names, directory names, file names, text that appears on the screen, or text that you enter.

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

TimesTen documentation uses the following variables for path, file, and user names.

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

---



---

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

---



---

## Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at

<http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc>.

### Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit

<http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info> or visit

<http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs> if you are hearing impaired.



---

---

## What's New

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

### New features in Release 18.1.1.1.0

- There are new error codes for manual retry after transient errors. TimesTen automatically resolves most transient errors (which is particularly important for TimesTen Scaleout), but if your application detects certain error codes, it is suggested to retry the current transaction or most recent API call, as applicable. Refer to "[Retrying after transient errors \(PL/SQL\)](#)" on page 4-6 for details.



---

---

# Introduction to PL/SQL in TimesTen

TimesTen supports PL/SQL (Procedural Language Extension to SQL), a programming language that enables you to integrate procedural constructs with SQL in your database. TimesTen Release 18.1 implements the PL/SQL language from Oracle Database release 12.1.0.2. As such, most PL/SQL features present in that release of Oracle Database are also present in TimesTen, operating in essentially the same way. (Refer to [Chapter 8, "TimesTen PL/SQL Support: Reference Summary"](#) for differences.)

This chapter provides a brief introduction to TimesTen PL/SQL, covering the following topics:

- [Features of PL/SQL in TimesTen](#)
- [TimesTen PL/SQL components and operations](#)
- [Audiences for this document](#)
- [TimesTen Quick Start and sample applications](#)

## Features of PL/SQL in TimesTen

PL/SQL support in TimesTen enables you to do the following:

- Take full advantage of the PL/SQL programming language.
- Execute PL/SQL from your client applications that use these APIs:
  - ODBC
  - JDBC
  - Oracle Call Interface (OCI)
  - Oracle Pro\*C/C++
  - Oracle Data Provider for .NET (ODP.NET)
  - TTClasses (TimesTen C++ library)
- Execute TimesTen SQL from PL/SQL.
- Create, alter, or drop standalone procedures, functions, packages and package bodies.
- Use PL/SQL packages to extend your database functionality and to provide PL/SQL access to SQL features.
- Handle exceptions and errors in your PL/SQL applications.
- Set connection attributes in your database to customize your PL/SQL environment.

- Alter session parameters so you can manage your PL/SQL environment.
- Display PL/SQL metadata in your database by using PL/SQL system views.

---

**Note:** For information about PL/SQL security, refer to "Privileges for PL/SQL statements and operations" in *Oracle TimesTen In-Memory Database Security Guide*.

---

## TimesTen PL/SQL components and operations

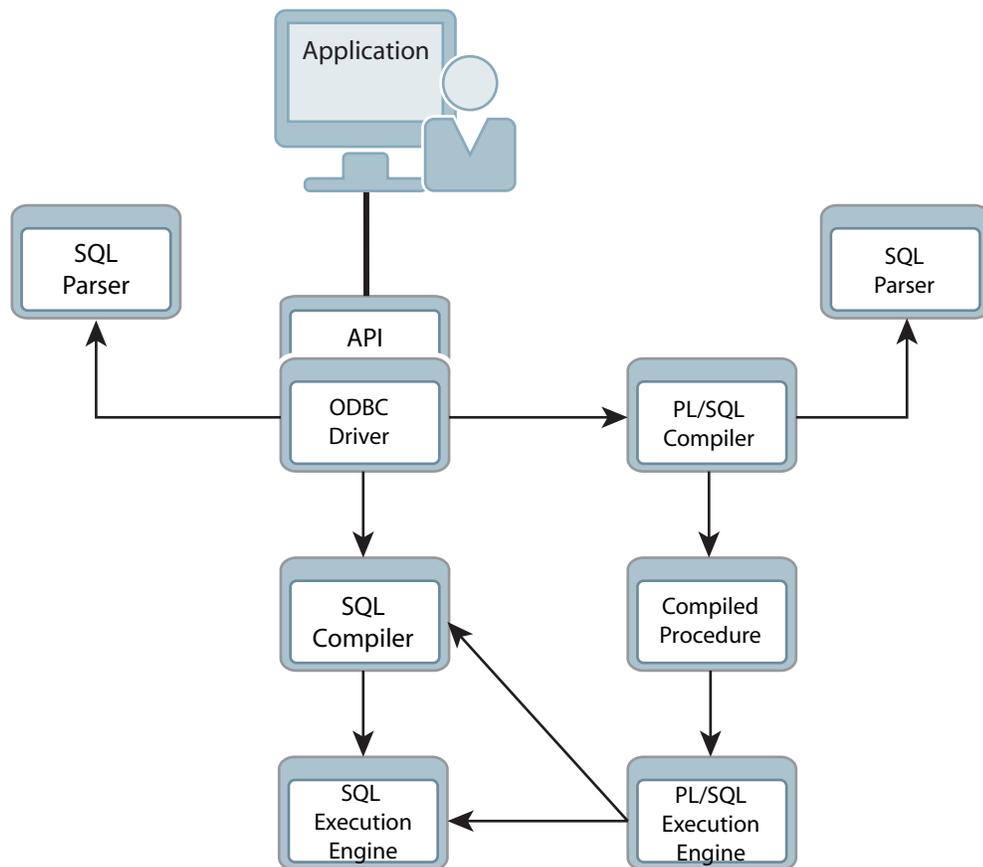
This section provides an overview of PL/SQL operations in TimesTen, including discussion of how an application interacts with PL/SQL and how PL/SQL components interact with other components of TimesTen. The following topics are covered:

- [Application interaction with TimesTen and PL/SQL](#)
- [PL/SQL in TimesTen versus PL/SQL in Oracle Database](#)

### Application interaction with TimesTen and PL/SQL

Figure 1–1 shows the PL/SQL components and their interactions with each other and with other TimesTen components during PL/SQL operations.

**Figure 1–1** *TimesTen PL/SQL components*



An application uses the API of its choice—ODBC, JDBC, OCI, Pro\*C, ODP.NET, or TTCclasses—to send requests to the database. ODBC is the TimesTen native API, so each of the other APIs ultimately calls the ODBC layer.

The ODBC driver calls the TimesTen SQL parser to examine each incoming request and determine whether it is SQL or PL/SQL. The request is then passed to the appropriate subsystem within TimesTen. PL/SQL source and SQL statements are compiled, optimized and executed by the PL/SQL subsystem and SQL subsystem, respectively.

The PL/SQL compiler is responsible for generating executable code from PL/SQL source, while the SQL compiler does the same for SQL statements. Each compiler generates intermediate code that can then be executed by the appropriate PL/SQL or SQL execution engine. This executable code, along with metadata about the PL/SQL blocks, is then stored in tables in the database.

When PL/SQL blocks are executed, the PL/SQL execution engine is invoked. As PL/SQL blocks in turn invoke SQL, the PL/SQL execution engine calls the TimesTen SQL compiler and the TimesTen SQL execution engine to handle SQL execution.

---



---

**Note:** The introduction of PL/SQL into TimesTen has little impact on applications that do not use it. If applications execute SQL directly, then requests are passed from the TimesTen ODBC driver to the TimesTen SQL compiler and execution engine in the same way as in previous releases.

---



---

## PL/SQL in TimesTen versus PL/SQL in Oracle Database

PL/SQL processing in TimesTen is largely identical to the processing in Oracle Database. The PL/SQL compiler and execution engine that are included with TimesTen originated in Oracle Database, and the relationship between PL/SQL components and the SQL compiler and execution engine is comparable. The tables used to store PL/SQL units are the same in TimesTen and Oracle Database, as are the views that are available to query information about stored PL/SQL units.

Beyond these basic similarities, however, are some potentially significant differences. These are detailed in the following subsections:

- [SQL statements in PL/SQL blocks](#)
- [Execution of PL/SQL from SQL](#)

### SQL statements in PL/SQL blocks

In TimesTen, as in Oracle Database, PL/SQL blocks may include SQL statements. Consider the anonymous block in the following example:

```
Command> create table tab2 (x number, last_name VARCHAR2 (25) INLINE NOT NULL);
Command> declare
    x number;
begin
    select salary into x from employees where last_name = 'Whalen';
    insert into tab2 values(x, 'Whalen');
end;
/
```

PL/SQL procedure successfully completed.

The PL/SQL compiler in TimesTen calls a copy of the Oracle Database SQL parser to analyze and validate the syntax of such SQL statements. This Oracle Database parser is included in TimesTen for this purpose. As part of this processing, PL/SQL may rewrite parts of the SQL statements (for example, by removing `INTO` clauses or replacing PL/SQL variables with binds). This processing is identical in TimesTen and in Oracle Database. The rewritten SQL statements are then included in the executable code for the PL/SQL block. When the PL/SQL block is executed, these SQL statements are compiled and executed by the TimesTen SQL subsystem.

In Oracle Database, the same SQL parser is used by the PL/SQL compiler and the SQL compiler. In TimesTen, however, different SQL parsers are used. TimesTen PL/SQL uses the Oracle Database SQL parser, while TimesTen SQL uses the native TimesTen SQL parser. This difference is typically, but not always, transparent to the end user. In particular, be aware of the following:

- SQL statements in TimesTen PL/SQL programs must obey Oracle Database SQL syntax. While TimesTen SQL is generally a subset of Oracle Database SQL, there are some expressions that are permissible in TimesTen SQL but not in Oracle Database SQL. Such TimesTen-specific SQL operations cannot be used within PL/SQL *except* by using dynamic SQL through `EXECUTE IMMEDIATE` statements or the `DBMS_SQL` package. See "[Dynamic SQL in PL/SQL \(EXECUTE IMMEDIATE statement\)](#)" on page 2-20.
- SQL statements that would be permissible in Oracle Database are accepted by the PL/SQL compiler as valid even if they cannot be executed by TimesTen. If SQL features are used that TimesTen does not support, compilation of a PL/SQL block may be successful, but a runtime error would occur when the PL/SQL block is executed.

### Execution of PL/SQL from SQL

In Oracle Database, PL/SQL blocks can invoke SQL statements, and SQL statements can in turn invoke PL/SQL functions. For example, a stored procedure can invoke an `UPDATE` statement that employs a user-written PL/SQL function in its `WHERE` clause.

In TimesTen, a SQL statement cannot invoke a PL/SQL function.

In addition, TimesTen does not support triggers. (See "XLA and TimesTen Event Management" in *Oracle TimesTen In-Memory Database C Developer's Guide* for information about XLA, a high-performance, asynchronous TimesTen alternative to triggers.)

## Audiences for this document

There are two primary developer audiences for this document:

- Developers experienced with Oracle Database and Oracle Database PL/SQL who want to learn how to use PL/SQL in TimesTen: These readers want to learn the differences between PL/SQL in Oracle Database and PL/SQL in TimesTen.
- Developers experienced with TimesTen who are not familiar with PL/SQL: These readers need general information about PL/SQL.

The following subsections note areas of particular interest in this document for each audience.

## Developers experienced with Oracle Database and Oracle Database PL/SQL

Developers experienced with Oracle Database PL/SQL can bypass much of this document, which covers many general concepts of PL/SQL. Likely areas of interest, particularly differences in PL/SQL functionality between Oracle Database and TimesTen, include the following. Note that TimesTen-specific considerations are discussed at the end of [Chapter 2](#), [Chapter 3](#), and [Chapter 4](#) and throughout [Chapter 8](#).

- ["Executing procedures and functions"](#) on page 2-9: This includes a comparison between how you can execute them in TimesTen and in Oracle Database.
- ["Differences in TimesTen: transaction behavior"](#) on page 2-31: This discusses cursor behavior when a transaction ends in TimesTen.
- ["Differences in TimesTen: data type considerations"](#) on page 3-10: This includes TimesTen-specific conversions, and types that TimesTen does not support.
- ["Differences in TimesTen: exception handling and error behavior"](#) on page 4-7: This describes differences in error support, handling, and reporting.
- [Chapter 6, "PL/SQL Environment"](#): This includes discussion of TimesTen connection attributes.
- [Chapter 7, "TimesTen Supplied PL/SQL Packages"](#): This documents the subset of Oracle Database PL/SQL packages that TimesTen supports.
- [Chapter 8, "TimesTen PL/SQL Support: Reference Summary"](#): This reference chapter provides a detailed treatment of differences between TimesTen PL/SQL and Oracle Database PL/SQL.

## Developers experienced with TimesTen

Most of this document is targeted for readers without prior PL/SQL experience, especially prior TimesTen users who are not familiar with PL/SQL, and nearly the entire document should be useful. In particular, [Chapter 2, "Programming Features in PL/SQL in TimesTen,"](#) will help these readers get started and [Chapter 5, "Examples Using TimesTen SQL in PL/SQL,"](#) includes some additional examples.

[Chapter 8, "TimesTen PL/SQL Support: Reference Summary,"](#) lists differences between TimesTen PL/SQL and Oracle Database PL/SQL and may be of less interest.

## TimesTen Quick Start and sample applications

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

After you have configured your environment, you can confirm that everything is set up correctly by compiling and running the sample applications. For TimesTen Classic, applications are located under the Quick Start `sample_code` directory. For instructions on compiling and running them, see the instructions in the subdirectories. For TimesTen Scaleout, clone the `oracle-timesten-examples` GitHub repository and follow the instructions in the README files.

For TimesTen Classic, the following are included:

- Schema and setup: The `build_sampledb` script (`.sh` on Linux or UNIX or `.bat` on Windows) creates a sample database and schema. Run this script before using the sample applications.

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

---

---

## Programming Features in PL/SQL in TimesTen

One of the advantages of PL/SQL in TimesTen is the ability to integrate PL/SQL procedural constructs with the flexible and powerful TimesTen SQL language.

This chapter surveys the main PL/SQL programming features described in "Overview of PL/SQL" in *Oracle Database PL/SQL Language Reference*. Working from simple examples, you will learn how to use PL/SQL in TimesTen. Unless otherwise noted, the examples have the same results in TimesTen as in Oracle Database.

See the end of the chapter for TimesTen-specific considerations. See "[TimesTen PL/SQL components and operations](#)" on page 1-2 for an overview of how applications interact with TimesTen in general and PL/SQL in particular.

The following are the main topics of this chapter:

- [PL/SQL blocks](#)
- [PL/SQL variables and constants](#)
- [SQL function calls from PL/SQL](#)
- [PL/SQL control structures](#)
- [PL/SQL procedures and functions](#)
- [PL/SQL packages](#)
- [How to pass data between an application and PL/SQL](#)
- [Use of SQL in PL/SQL programs](#)
- [Use of cursors in PL/SQL programs](#)
- [Wrapping PL/SQL source code](#)
- [Differences in TimesTen: transaction behavior](#)

---

---

**Note:** Except where stated otherwise, the examples in this guide use the TimesTen `ttIsql` utility. In order to display output in the examples, the setting `SET SERVEROUTPUT ON` is used. For more information on `ttIsql`, see "`ttIsql`" in *Oracle TimesTen In-Memory Database Reference*.

---

---

### PL/SQL blocks

The basic unit of a PL/SQL source program is the *block*, or *anonymous block*, which groups related declarations and statements. TimesTen supports PL/SQL blocks.

A PL/SQL block is defined by the keywords DECLARE, BEGIN, EXCEPTION, and END.

[Example 2-1](#) shows the basic structure of a PL/SQL block.

---

---

**Note:** If you use TimesTen Application-Tier Database Cache (TimesTen Cache): A PL/SQL block cannot be passed through to Oracle Database. (Also see "[TimesTen PL/SQL with TimesTen Cache](#)" on page 2-23.)

---

---

**Example 2-1 PL/SQL block structure**

```
DECLARE    -- (optional)
           -- Variables, cursors, user-defined exceptions
BEGIN      -- (mandatory)
           -- PL/SQL statements
EXCEPTION  -- (optional)
           -- Actions to perform when errors occur
END        -- (mandatory)
```

You can define either anonymous or named blocks in your PL/SQL programs. This example creates an anonymous block that queries the `employees` table and returns the data in a PL/SQL variable:

```
Command> SET SERVEROUTPUT ON;
Command> DECLARE
           v_fname VARCHAR2 (20);
BEGIN
           SELECT first_name
              INTO v_fname
             FROM employees
            WHERE employee_id = 100;
           DBMS_OUTPUT.PUT_LINE (v_fname);
END;
/
```

Steven

PL/SQL procedure successfully completed.

## PL/SQL variables and constants

You can define variables and constants in PL/SQL and then use them in procedural statements and in SQL anywhere an expression can be used.

For example:

```
Command> DECLARE
           v_hiredate DATE;
           v_deptno  NUMBER (2) NOT NULL := 10;
           v_location VARCHAR2 (13) := 'San Francisco';
           c_comm    CONSTANT NUMBER := 1400;
```

You can use the `%TYPE` attribute to declare a variable according to either a TimesTen column definition or another declared variable. For example, use `%TYPE` to create variables `emp_lname` and `min_balance`:

```
Command> DECLARE
           emp_lname employees.last_name%TYPE;
           balance  NUMBER (7,2);
           min_balance balance%TYPE:= 1000;
BEGIN
```

```

        SELECT last_name INTO emp_lname FROM employees WHERE employee_id = 100;
        DBMS_OUTPUT.PUT_LINE (emp_lname);
        DBMS_OUTPUT.PUT_LINE (min_balance);
    END;
    /
King
1000

```

PL/SQL procedure successfully completed.

You can assign a value to a variable in the following ways.

- With the assignment operator (:=), as shown in [Example 2-2](#)
- By selecting or fetching values into it, as shown in [Example 2-3](#)
- By passing the variable as an OUT or IN OUT parameter to a subprogram (procedure or function) and then assigning the value inside the subprogram, as shown in [Example 2-4](#)

---

**Note:** The DBMS\_OUTPUT package used in these examples is supplied with TimesTen. For information on this and other supplied packages, refer to [Chapter 7, "TimesTen Supplied PL/SQL Packages"](#).

---

#### **Example 2-2 Assigning values to variables with the assignment operator**

```

Command> DECLARE -- Assign values in the declarative section
    wages NUMBER;
    hours_worked NUMBER := 40;
    hourly_salary NUMBER := 22.50;
    bonus NUMBER := 150;
    country VARCHAR2(128);
    counter NUMBER := 0;
    done BOOLEAN;
    valid_id BOOLEAN;
    emp_rec1 employees%ROWTYPE;
    emp_rec2 employees%ROWTYPE;
    TYPE commissions IS TABLE OF NUMBER INDEX BY PLS_INTEGER;
    comm_tab commissions;
BEGIN -- Assign values in the executable section
    wages := (hours_worked * hourly_salary) + bonus;
    country := 'France';
    country := UPPER('Canada');
    done := (counter < 100);
    valid_id := TRUE;
    emp_rec1.first_name := 'Amy';
    emp_rec1.last_name := 'Feiner';
    emp_rec1 := emp_rec2;
    comm_tab(5) := 20000 * 0.15;
END;
/

```

PL/SQL procedure successfully completed.

---

---

**Note:** This example uses records, which are composite data structures that have fields with different data types. You can use the %ROWTYPE attribute, as shown, to declare a record that represents a row in a table or a row from a query result set. Records are further discussed under "[PL/SQL composite data types](#)" on page 3-4.

---

---

**Example 2-3 Using SELECT INTO to assign values to variables**

Select 10% of an employee's salary into the bonus variable:

```
Command> DECLARE
          bonus NUMBER(8,2);
          emp_id NUMBER(6) := 100;
        BEGIN
          SELECT salary * 0.10 INTO bonus FROM employees
             WHERE employee_id = emp_id;
          DBMS_OUTPUT.PUT_LINE (bonus);
        END;
        /
2400
```

PL/SQL procedure successfully completed.

**Example 2-4 Assigning values to variables as parameters of a subprogram**

Declare the variable new\_sal and then pass the variable as a parameter (sal) to procedure adjust\_salary. Procedure adjust\_salary computes the average salary for employees with job\_id='ST\_CLERK' and then updates sal. After the procedure is executed, the value of the variable is displayed to verify that the variable was correctly updated.

```
Command> DECLARE
          new_sal NUMBER(8,2);
          emp_id NUMBER(6) := 126;
        PROCEDURE adjust_salary (emp_id NUMBER, sal IN OUT NUMBER) IS
          emp_job VARCHAR2(10);
          avg_sal NUMBER(8,2);
        BEGIN
          SELECT job_id INTO emp_job FROM employees
             WHERE employee_id = emp_id;
          SELECT AVG(salary) INTO avg_sal FROM employees
             WHERE job_id = emp_job;
          DBMS_OUTPUT.PUT_LINE ('The average salary for ' || emp_job
             || ' employees: ' || TO_CHAR(avg_sal));
          sal := (sal + avg_sal)/2;
          DBMS_OUTPUT.PUT_LINE ('New salary is ' || sal);
        END;
        BEGIN
          SELECT AVG(salary) INTO new_sal FROM employees;
          DBMS_OUTPUT.PUT_LINE ('The average salary for all employees: '
             || TO_CHAR(new_sal));
          adjust_salary(emp_id, new_sal);
          DBMS_OUTPUT.PUT_LINE ('Salary should be same as new salary ' ||
             new_sal);
        END;
        /
The average salary for all employees: 6461.68
The average salary for ST_CLERK employees: 2785
New salary is 4623.34
```

Salary should be same as new salary 4623.34

PL/SQL procedure successfully completed.

---



---

**Note:** This example illustrates the ability to nest PL/SQL blocks within blocks. The outer anonymous block contains an enclosed procedure. This PROCEDURE statement is distinct from the CREATE PROCEDURE statement documented in "PL/SQL procedures and functions" on page 2-8, which creates a subprogram that remains stored in the user's schema.

---



---

## SQL function calls from PL/SQL

Most SQL functions are supported for calls directly from PL/SQL. In the first example that follows, the function RTRIM is used as a PL/SQL function in a PL/SQL assignment statement. In the second example, it is used as a SQL function in a static SQL statement.

### Example 2-5 Using the RTRIM function from PL/SQL

Use the TimesTen PL/SQL RTRIM built-in function to remove the right-most "x" and "y" characters from the string. Note that RTRIM is used in a PL/SQL assignment statement.

```
Command> DECLARE p_var VARCHAR2(30);
          BEGIN
              p_var := RTRIM ('RTRIM Examplexxxxxyxy', 'xy');
              DBMS_OUTPUT.PUT_LINE (p_var);
          END;
          /
RTRIM Example
```

PL/SQL procedure successfully completed.

### Example 2-6 Using the RTRIM function from SQL

Use the TimesTen SQL function RTRIM to remove the right-most "x" and "y" characters from the string. Note that RTRIM is used in a static SQL statement.

```
Command> DECLARE tt_var VARCHAR2 (30);
          BEGIN
              SELECT RTRIM ('RTRIM Examplexxxxxyxy', 'xy')
                 INTO tt_var FROM DUAL;
              DBMS_OUTPUT.PUT_LINE (tt_var);
          END;
          /
RTRIM Example
```

PL/SQL procedure successfully completed.

You can refer to information about SQL functions in TimesTen under "Expressions" in *Oracle TimesTen In-Memory Database SQL Reference*. See "SQL Functions in PL/SQL Expressions" in *Oracle Database PL/SQL Language Reference* for information about support for SQL functions in PL/SQL.

## PL/SQL control structures

Control structures are among the PL/SQL extensions to SQL. TimesTen supports the same control structures as Oracle Database.

The following control structures are discussed in this section.

- [Conditional control](#)
- [Iterative control](#)
- [CONTINUE statement](#)

## Conditional control

The IF-THEN-ELSE and CASE constructs are examples of conditional control. In [Example 2-7](#), the IF-THEN-ELSE construct is used to determine the salary raise of an employee based on the current salary. The CASE construct is also used to choose the course of action to take based on the job\_id of the employee.

### **Example 2-7 Using the IF-THEN-ELSE and CASE constructs**

```
Command> DECLARE
        jobid employees.job_id%TYPE;
        empid employees.employee_id%TYPE := 115;
        sal employees.salary%TYPE;
        sal_raise NUMBER(3,2);
BEGIN
    SELECT job_id, salary INTO jobid, sal from employees
        WHERE employee_id = empid;
    CASE
    WHEN jobid = 'PU_CLERK' THEN
        IF sal < 3000 THEN sal_raise := .12;
        ELSE sal_raise := .09;
        END IF;
    WHEN jobid = 'SH_CLERK' THEN
        IF sal < 4000 THEN sal_raise := .11;
        ELSE sal_raise := .08;
        END IF;
    WHEN jobid = 'ST_CLERK' THEN
        IF sal < 3500 THEN sal_raise := .10;
        ELSE sal_raise := .07;
        END IF;
    ELSE
        BEGIN
            DBMS_OUTPUT.PUT_LINE('No raise for this job: ' || jobid);
        END;
    END CASE;
    DBMS_OUTPUT.PUT_LINE ('Original salary ' || sal);
    -- Update
    UPDATE employees SET salary = salary + salary * sal_raise
    WHERE employee_id = empid;
END;
/
Original salary 3100
```

PL/SQL procedure successfully completed.

## Iterative control

An iterative control construct executes a sequence of statements repeatedly, as long as a specified condition is true. Loop constructs are used to perform iterative operations.

There are three loop types:

- Basic loop

- FOR loop
- WHILE loop

The basic loop performs repetitive actions without overall conditions. The FOR loop performs iterative actions based on a count. The WHILE loops perform iterative actions based on a condition.

### **Example 2–8 Using a WHILE loop**

```
Command> CREATE TABLE temp (tempid NUMBER(6),
    tempsal NUMBER(8,2),
    tempname VARCHAR2(25));
Command> DECLARE
    sal employees.salary%TYPE := 0;
    mgr_id employees.manager_id%TYPE;
    lname employees.last_name%TYPE;
    starting_empid employees.employee_id%TYPE := 120;
BEGIN
    SELECT manager_id INTO mgr_id
    FROM employees
    WHERE employee_id = starting_empid;
    WHILE sal <= 15000 LOOP -- loop until sal > 15000
        SELECT salary, manager_id, last_name INTO sal, mgr_id, lname
        FROM employees WHERE employee_id = mgr_id;
    END LOOP;
    INSERT INTO temp VALUES (NULL, sal, lname); -- insert NULL for tempid
    COMMIT;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        INSERT INTO temp VALUES (NULL, NULL, 'Not found'); -- insert NULLs
        COMMIT;
END;
/
```

PL/SQL procedure successfully completed.

```
Command> SELECT * FROM temp;
< <NULL>, 24000, King >
1 row found.
```

## **CONTINUE statement**

The CONTINUE statement enables you to transfer control within a loop back to a new iteration.

### **Example 2–9 Using the CONTINUE statement**

In this example, the first `v_total` assignment is executed for each of the 10 iterations of the loop. The second `v_total` assignment is executed for the first five iterations of the loop. The CONTINUE statement transfers control within a loop back to a new iteration, so for the last five iterations of the loop, the second `v_total` assignment is not executed. The end `v_total` value is 70.

```
Command> DECLARE
    v_total SIMPLE_INTEGER := 0;
BEGIN
    FOR i IN 1..10 LOOP
        v_total := v_total + i;
        DBMS_OUTPUT.PUT_LINE ('Total is : ' || v_total);
        CONTINUE WHEN i > 5;
    END LOOP;
END;
```

```
        v_total := v_total + i;
        DBMS_OUTPUT.PUT_LINE ('Out of loop Total is: ' || v_total);
    END LOOP;
END;
/
Total is : 1
Out of loop Total is: 2
Total is : 4
Out of loop Total is: 6
Total is : 9
Out of loop Total is: 12
Total is : 16
Out of loop Total is: 20
Total is : 25
Out of loop Total is: 30
Total is : 36
Total is : 43
Total is : 51
Total is : 60
Total is : 70
```

PL/SQL procedure successfully completed.

## PL/SQL procedures and functions

Procedures and functions are PL/SQL blocks that have been defined with a specified name. This section covers the following:

- [Creating procedures and functions](#)
- [Executing procedures and functions](#)
- [Using synonyms for procedures and functions](#)

---

---

**Important:** TimesTen Scaleout does not support user-defined PL/SQL procedures and functions.

---

---

## Creating procedures and functions

In TimesTen Classic, you can create standalone subprograms (stored procedures or functions) at the database level with the `CREATE PROCEDURE` or `CREATE FUNCTION` statement.

Optionally use `CREATE OR REPLACE PROCEDURE` or `CREATE OR REPLACE FUNCTION` if you want the subprogram to be replaced if it already exists.

Use `ALTER PROCEDURE` or `ALTER FUNCTION` to explicitly compile a procedure or function or modify the compilation options. (To recompile a procedure or function that is part of a package, recompile the package using the `ALTER PACKAGE` statement.)

In TimesTen, syntax for `CREATE PROCEDURE` and `CREATE FUNCTION` is a subset of what is supported in Oracle Database. For information on these statements and the `ALTER PROCEDURE` and `ALTER FUNCTION` statements in TimesTen, see "SQL Statements" in *Oracle TimesTen In-Memory Database SQL Reference*.

**Notes:**

- If you use replication: PL/SQL DDL statements, such as `CREATE` statements for PL/SQL functions, procedures, and packages, are not replicated. See "Creating a new PL/SQL object in an existing active standby pair" and "Adding a PL/SQL object to an existing classic replication scheme" in *Oracle TimesTen In-Memory Database Replication Guide* for steps to address this.
- If you use TimesTen Cache with TimesTen Classic: A PL/SQL procedure or function resident in Oracle Database cannot be called in TimesTen by passthrough. Procedures and functions must be defined in TimesTen to be executable in TimesTen. (Also see "[TimesTen PL/SQL with TimesTen Cache](#)" on page 2-23.)
- PL/SQL and database object names: TimesTen does not support non-ASCII or quoted non-uppercase names of PL/SQL objects (procedures, functions, and packages). Also, trailing spaces in the quoted names of PL/SQL objects are not supported. In addition, trailing spaces in the quoted names of objects such as tables and views that are passed to PL/SQL are silently removed.
- Definer's rights or invoker's rights determines access to SQL objects used by a PL/SQL procedure or function. For information, refer to "Definer's rights and invoker's rights (AUTHID clause)" in *Oracle TimesTen In-Memory Database Security Guide*.
- See "[Showing errors in ttlsq](#)" on page 4-6 for how to get information when you encounter errors in compiling a procedure or function.

## Executing procedures and functions

TimesTen supports execution of PL/SQL from client applications using ODBC, OCI, Pro\*C/C++, ODP.NET, JDBC, or TimesTen TTClasses (for C++).

As noted earlier, a block is the basic unit of a PL/SQL source program. Anonymous blocks were also discussed earlier. By contrast, procedures and functions are PL/SQL blocks that have been defined with a specified name. See "[PL/SQL procedures and functions](#)" on page 2-8 for how to define and create them.

---

**Important:** TimesTen Scaleout does not support user-defined PL/SQL procedures and functions.

---

In TimesTen Classic, a PL/SQL procedure or function that is standalone (created with `CREATE PROCEDURE` or `CREATE FUNCTION`) or part of a package can be executed using an anonymous block or a `CALL` statement. (See "`CALL`" in *Oracle TimesTen In-Memory Database SQL Reference* for details about `CALL` syntax.)

Consider the following function:

```
create or replace function mytest return number is
begin
    return 1;
end;
/
```

In TimesTen, you can execute `mytest` in either of the following ways.

- In an anonymous block:

```
Command> variable n number;
Command> begin
           :n := mytest();
           end;
           /
```

PL/SQL procedure successfully completed.

```
Command> print n;
N                : 1
```

- In a CALL statement:

```
Command> variable n number;
Command> call mytest() into :n;
Command> print n;
N                : 1
```

In Oracle Database, you could also execute `mytest` through a SQL statement, as follows. This execution mechanism is *not* supported in TimesTen.

- In a SELECT statement:

```
SQL> select mytest from dual;
```

```
MYTEST
-----
      1
```

---

---

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

---

---

### **Example 2–10 Create and execute a procedure with OUT parameters**

This example creates a procedure that uses OUT parameters, executes the procedure in an anonymous block, then displays the OUT values. The procedure takes an employee ID as input then outputs the salary and job ID for the employee.

```
Command> CREATE OR REPLACE PROCEDURE get_employee
         (p_empid in employees.employee_id%TYPE,
          p_sal OUT employees.salary%TYPE,
          p_job OUT employees.job_id%TYPE) IS
BEGIN
  SELECT salary, job_id
  INTO p_sal, p_job
  FROM employees
  WHERE employee_id = p_empid;
END;
/
```

Procedure created.

```
Command> VARIABLE v_salary NUMBER;
Command> VARIABLE v_job VARCHAR2(15);
Command> BEGIN
```

```

        GET_EMPLOYEE (120, :v_salary, :v_job);
    END;
/

```

PL/SQL procedure successfully completed.

```

Command> PRINT
V_SALARY          : 8000
V_JOB             : ST_MAN

```

```

Command> SELECT salary, job_id FROM employees WHERE employee_id = 120;
< 8000, ST_MAN >
1 row found.

```

---



---

**Note:** Instead of using the anonymous block shown in the preceding example, you could use a CALL statement:

```

Command> CALL GET_EMPLOYEE(120, :v_salary, :v_job);

```

---



---

### Example 2-11 Create and call a function

This example creates a function that returns the salary of the employee whose employee ID is specified as input, then calls the function and displays the result that was returned.

```

Command> CREATE OR REPLACE FUNCTION get_sal
        (p_id employees.employee_id%TYPE) RETURN NUMBER IS
        v_sal employees.salary%TYPE := 0;
    BEGIN
        SELECT salary INTO v_sal FROM employees
            WHERE employee_id = p_id;
        RETURN v_sal;
    END get_sal;
/

```

Function created.

```

Command> variable n number;
Command> call get_sal(100) into :n;
Command> print n;
N                : 24000

```

---



---

**Note:** Instead of using the CALL statement shown in the preceding example, you could use an anonymous block:

```

Command> begin
        :n := get_sal(100);
    end;
/

```

---



---

## Using synonyms for procedures and functions

TimesTen supports private and public synonyms (aliases) for database objects, including PL/SQL procedures, functions, and packages. Synonyms are often used to mask object names and object owners or to simplify SQL statements.

Create a private synonym for procedure foo in your schema as follows:

```

CREATE SYNONYM synfoo FOR foo;

```

Create a public synonym as follows:

```
CREATE PUBLIC SYNONYM pubfoo FOR foo;
```

A private synonym exists in the schema of a specific user and shares the same namespace as database objects such as tables, views, and sequences. A private synonym cannot have the same name as a table or other object in the same schema.

A public synonym does not belong to any particular schema, is accessible to all users, and can have the same name as any private object.

To use a synonym you must have appropriate privileges to access the underlying object. For required privileges to create or drop a synonym, see "Privileges for PL/SQL statements and operations" in *Oracle TimesTen In-Memory Database Security Guide*.

For general information about synonyms, see "Understanding synonyms" in *Oracle TimesTen In-Memory Database Operations Guide*. For information about the `CREATE SYNONYM` and `DROP SYNONYM` statements, see "SQL Statements" in *Oracle TimesTen In-Memory Database SQL Reference*.

### **Example 2–12 Use a synonym for a procedure**

In the following example, `USER1` creates a procedure in his schema and creates a public synonym for it. Then `USER2` executes the procedure through the public synonym.

Assume the following:

- `USER1` has been granted `CREATE SESSION`, `CREATE PROCEDURE`, and `CREATE PUBLIC SYNONYM` privileges.
- `USER2` has been granted `CREATE SESSION` and `EXECUTE ANY PROCEDURE` privileges.
- Both users have connected to the database.
- `USER2` employs the `SET SERVEROUTPUT ON` setting.

#### **USER1:**

```
Command> create or replace procedure test is
begin
  dbms_output.put_line('Running the test');
end;
/
```

Procedure created.

```
Command> create public synonym pubtest for test;
```

Synonym created.

#### **USER2:**

```
Command> begin
  pubtest;
end;
/
```

Running the test

PL/SQL procedure successfully completed.

## **PL/SQL packages**

This section discusses how to create and use PL/SQL packages in TimesTen Classic.

---

For information about PL/SQL packages provided with TimesTen, refer to [Chapter 7, "TimesTen Supplied PL/SQL Packages."](#)

---

**Important:** TimesTen Scaleout does not support user-defined PL/SQL packages.

---

## Package concepts

A package is a database object that groups logically related PL/SQL types, variables, and subprograms. You specify the package and then define its body in separate steps.

The package specification is the interface to the package, declaring the public types, variables, constants, exceptions, cursors, and subprograms that are visible outside the immediate scope of the package. The body defines the objects declared in the specification, queries for the cursors, code for the subprograms, and private objects that are not visible to applications outside the package.

TimesTen stores the package specification separately from the package body in the database. Other schema objects that call or reference public program objects depend only on the package specification, not on the package body.

---

**Note:** The syntax for creating packages and package bodies is the same as in Oracle Database; however, while Oracle Database documentation mentions that you must run a script named `DBMSSTDX.SQL`, this does not apply to TimesTen.

---

## Creating and using packages

In TimesTen Classic, create packages and store them permanently in a TimesTen database by using the `CREATE PACKAGE` and `CREATE PACKAGE BODY` statements.

To create a new package, do the following:

1. Create the package specification with the `CREATE PACKAGE` statement.

You can declare program objects in the package specification. Such objects are referred to as *public* objects and can be referenced outside the package, and by other objects in the package.

Optionally use `CREATE OR REPLACE PACKAGE` if you want the package specification to be replaced if it already exists.

2. Create the package body with the `CREATE PACKAGE BODY` (or `CREATE OR REPLACE PACKAGE BODY`) statement.

You can declare and define program objects in the package body.

- You must define public objects declared in the package specification.
- You can declare and define additional package objects, referred to as *private* objects. Private objects are declared in the package body rather than in the package specification, so they can be referenced only by other objects in the package. They cannot be referenced outside the package.

Use `ALTER PACKAGE` to explicitly compile the member procedures and functions of a package or modify the compilation options.

For more information on the `CREATE PACKAGE`, `CREATE PACKAGE BODY`, and `ALTER PACKAGE` statements, see "SQL Statements" in *Oracle TimesTen In-Memory Database SQL Reference*.

**Notes:**

- If you use replication: PL/SQL DDL statements, such as CREATE statements for PL/SQL functions, procedures, and packages, are not replicated. See "Creating a new PL/SQL object in an existing active standby pair" and "Adding a PL/SQL object to an existing classic replication scheme" in *Oracle TimesTen In-Memory Database Replication Guide* for steps to address this.
- See "Showing errors in ttlsq" on page 4-6 for how to get information when you encounter errors in compiling a package.

**Example 2–13 Create and use a package**

Consider the case where you want to add a row to the employees tables when you hire a new employee and delete a row from the employees table when an employee leaves your company. The following example creates two procedures to accomplish these tasks and bundles the procedures in a package. The package also contains a function to return the count of employees with a salary greater than that of a specific employee. The example then executes the function and procedures and verifies the results.

```
Command> CREATE OR REPLACE PACKAGE emp_actions AS
    PROCEDURE hire_employee (employee_id NUMBER,
        last_name VARCHAR2,
        first_name VARCHAR2,
        email VARCHAR2,
        phone_number VARCHAR2,
        hire_date DATE,
        job_id VARCHAR2,
        salary NUMBER,
        commission_pct NUMBER,
        manager_id NUMBER,
        department_id NUMBER);
    PROCEDURE remove_employee (emp_id NUMBER);
    FUNCTION num_above_salary (emp_id NUMBER) RETURN NUMBER;
END emp_actions;
/
```

Package created.

```
Command> -- Package body:
CREATE OR REPLACE PACKAGE BODY emp_actions AS
-- Code for procedure hire_employee:
    PROCEDURE hire_employee (employee_id NUMBER,
        last_name VARCHAR2,
        first_name VARCHAR2,
        email VARCHAR2,
        phone_number VARCHAR2,
        hire_date DATE,
        job_id VARCHAR2,
        salary NUMBER,
        commission_pct NUMBER,
        manager_id NUMBER,
        department_id NUMBER) IS
    BEGIN
        INSERT INTO employees VALUES (employee_id,
            last_name,
            first_name,
            email,
```

```

        phone_number,
        hire_date,
        job_id,
        salary,
        commission_pct,
        manager_id,
        department_id);
    END hire_employee;
-- Code for procedure remove_employee:
    PROCEDURE remove_employee (emp_id NUMBER) IS
    BEGIN
        DELETE FROM employees WHERE employee_id = emp_id;
    END remove_employee;
-- Code for function num_above_salary:
    FUNCTION num_above_salary (emp_id NUMBER) RETURN NUMBER IS
        emp_sal NUMBER(8,2);
        num_count NUMBER;
    BEGIN
        SELECT salary INTO emp_sal FROM employees
        WHERE employee_id = emp_id;
        SELECT COUNT(*) INTO num_count FROM employees
        WHERE salary > emp_sal;
        RETURN num_count;
    END num_above_salary;
END emp_actions;
/

```

Package body created.

```

Command> BEGIN
    /* call function to return count of employees with salary
       greater than salary of employee with employee_id = 120
    */
    DBMS_OUTPUT.PUT_LINE
        ('Number of employees with higher salary: ' ||
         TO_CHAR(emp_actions.num_above_salary(120)));
END;
/

```

Number of employees with higher salary: 33

PL/SQL procedure successfully completed.

Verify the count of 33.

```

Command> SELECT salary FROM employees WHERE employee_id = 120;
< 8000 >
1 row found.

```

```

Command> SELECT COUNT (*) FROM employees WHERE salary > 8000;
< 33 >
1 row found.

```

Now add an employee and verify results. Then, remove the employee and verify that the employee was deleted from the employees table.

```

Command> BEGIN
    emp_actions.hire_employee(300,
        'Belden',
        'Enrique',
        'EBELDEN',
        '555.111.2222',

```

```
'31-AUG-04',  
'AC_MGR',  
9000,  
.1,  
101,  
110);  
END;  
/
```

PL/SQL procedure successfully completed.

```
Command> SELECT * FROM employees WHERE employee_id = 300;  
< 300, Belden, Enrique, EBELDEN, 555.111.2222, 2004-08-31 00:00:00, AC_MGR, 9000,  
.1, 101, 110 >  
1 row found.  
Command> BEGIN  
    emp_actions.remove_employee (300);  
END;  
/
```

PL/SQL procedure successfully completed.

```
Command> SELECT * FROM employees WHERE employee_id = 300;  
0 rows found.
```

## Using synonyms for packages

TimesTen supports private and public synonyms (aliases) for database objects, including PL/SQL procedures, functions, and packages. Synonyms are often used to mask object names and object owners or to simplify SQL statements.

To create a private synonym for package foopkg in your schema:

```
CREATE SYNONYM synfoopkg FOR foopkg;
```

To create a public synonym for foopkg:

```
CREATE PUBLIC SYNONYM pubfoopkg FOR foopkg;
```

Also see "[Using synonyms for procedures and functions](#)" on page 2-11 in this document and "[Privileges for PL/SQL statements and operations](#)" in *Oracle TimesTen In-Memory Database Security Guide*.

---

---

**Note:** You cannot create synonyms for individual member subprograms of a package.

This is valid:

```
create or replace public synonym pubtestpkg for testpkg;
```

This is not valid:

```
create or replace public synonym pubtestproc for testpkg.testproc;
```

---

---

## How to pass data between an application and PL/SQL

This section covers the following topics for passing data between an application and PL/SQL:

- [Using bind variables from an application](#)

- [IN, OUT, and IN OUT parameter modes](#)

Refer to "Bind Variables" in *Oracle Database PL/SQL Language Reference* for additional information.

## Using bind variables from an application

You can use `:var` notation for bind variables to be passed between your application (such as a C or Java application) and PL/SQL. The term *bind variable* (or sometimes *host variable*) is used equivalently to how the term *parameter* has historically been used in TimesTen, and bind variables from an application would correspond to the parameters declared in a PL/SQL procedure or function specification.

Here is a simple example using `ttIsql` to call a PL/SQL procedure that retrieves the name and salary of the employee corresponding to a specified employee ID. In this example, `ttIsql` essentially acts as the calling application, and the name and salary are output from PL/SQL:

```
Command> VARIABLE b_name VARCHAR2 (25)
Command> VARIABLE b_sal NUMBER

Command> BEGIN
    query_emp (171, :b_name, :b_sal);
END;
/
```

PL/SQL procedure successfully completed.

```
Command> PRINT b_name
B_NAME           : Smith
Command> PRINT b_sal
B_SAL            : 7400
```

See ["Examples using input and output parameters and bind variables"](#) on page 5-3 for the complete example.

See ["PL/SQL procedures and functions"](#) on page 2-8 for how to create and define procedures and functions.

See ["Binding parameters and executing statements"](#) in *Oracle TimesTen In-Memory Database C Developer's Guide* and ["Preparing SQL statements and setting input parameters"](#) in *Oracle TimesTen In-Memory Database Java Developer's Guide* for additional information and examples for those languages.

---

**Notes:**

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

```
SELECT 'x' FROM DUAL WHERE :a = :b;
```

You could address the issue as follows, for example:

```
SELECT 'x' from DUAL WHERE CAST(:a as VARCHAR2(10)) =  
                           CAST(:b as VARCHAR2(10));
```

- For duplicate parameters, the implementation in PL/SQL in TimesTen is no different than the implementation in PL/SQL in Oracle Database.
  - The term "bind parameter" as used in TimesTen developer guides (in keeping with ODBC terminology) is equivalent to the term "bind variable" as used in TimesTen PL/SQL documents (in keeping with Oracle Database PL/SQL terminology).
- 

## IN, OUT, and IN OUT parameter modes

Parameter modes define whether parameters declared in a PL/SQL subprogram (procedure or function) specification are used for input, output, or both. The three parameter modes are `IN` (the default), `OUT`, and `IN OUT`.

An `IN` parameter lets you pass a value to the subprogram being invoked. Inside the subprogram, an `IN` parameter acts like a constant and cannot be assigned a value. You can pass a constant, literal, initialized variable, or expression as an `IN` parameter.

An `OUT` parameter returns a value to the caller of a subprogram. Inside the subprogram, an `OUT` parameter acts like a variable. You can change its value and reference the value after assigning it.

An `IN OUT` parameter passes an initial value to a subprogram and returns an updated value to the caller. It can be assigned a value and its value can be read. Typically, an `IN OUT` parameter is a string buffer or numeric accumulator that is read inside the subprogram and then updated. The actual parameter that corresponds to an `IN OUT` formal parameter must be a variable, not a constant or an expression.

---

**Note:** TimesTen supports the binding of associative arrays (but not varrays or nested tables) as `IN`, `OUT`, or `IN OUT` parameters. See ["Using associative arrays from applications"](#) on page 3-6.

---

See ["Examples using input and output parameters and bind variables"](#) on page 5-3.

## Use of SQL in PL/SQL programs

PL/SQL is tightly integrated with the TimesTen database through the SQL language. This section covers use of the following SQL features in PL/SQL.

- [Static SQL in PL/SQL for queries and DML statements](#)

- [Dynamic SQL in PL/SQL \(EXECUTE IMMEDIATE statement\)](#)
- [FORALL and BULK COLLECT operations](#)
- [RETURNING INTO clause](#)
- [TimesTen PL/SQL with TimesTen Cache](#)
- [Large objects \(LOBs\)](#)

## Static SQL in PL/SQL for queries and DML statements

From within PL/SQL, you can execute the following as static SQL:

- DML statements: INSERT, UPDATE, DELETE, and MERGE
- Queries: SELECT
- Transaction control: COMMIT and ROLLBACK

---



---

### Notes:

- You must use dynamic SQL to execute DDL statements in PL/SQL. See the next section, "[Dynamic SQL in PL/SQL \(EXECUTE IMMEDIATE statement\)](#)".
  - See "[Differences in TimesTen: transaction behavior](#)" on page 2-31 for details about how TimesTen transaction behavior differs from Oracle Database behavior.
- 
- 

For information on these SQL statements, refer to "SQL Statements" in *Oracle TimesTen In-Memory Database SQL Reference*.

[Example 2-14](#) shows how to execute a query. For additional examples using TimesTen SQL in PL/SQL, see [Chapter 5, "Examples Using TimesTen SQL in PL/SQL"](#).

### **Example 2-14 Retrieving data with SELECT...INTO**

Use the SELECT... INTO statement to retrieve exactly one row of data. TimesTen returns an error for any query that returns no rows or multiple rows.

This example retrieves hire\_date and salary for the employee with employee\_id=100 from the employees table of the HR schema.

Command> run selectinto.sql

```

DECLARE
  v_emp_hiredate employees.hire_date%TYPE;
  v_emp_salary   employees.salary%TYPE;

BEGIN
  SELECT hire_date, salary
  INTO   v_emp_hiredate, v_emp_salary
  FROM   employees
  WHERE  employee_id = 100;
  DBMS_OUTPUT.PUT_LINE(v_emp_hiredate || ' ' || v_emp_salary);
END;
/

```

1987-06-17 24000

PL/SQL procedure successfully completed.

## Dynamic SQL in PL/SQL (EXECUTE IMMEDIATE statement)

You can use native dynamic SQL, through the `EXECUTE IMMEDIATE` statement, to accomplish any of the following:

- Execute a DML statement such as `INSERT`, `UPDATE`, or `DELETE`.
- Execute a DDL statement such as `CREATE` or `ALTER`. For example, you can use `ALTER SESSION` to change a PL/SQL first connection attribute.
- Execute a PL/SQL anonymous block.
- Call a PL/SQL stored procedure or function.
- Call a TimesTen built-in procedure. (See "Built-In Procedures" in *Oracle TimesTen In-Memory Database Reference*.)

One use case is if you do not know the full text of your SQL statement until execution time. For example, during compilation you may not know the name of the column to use in the `WHERE` clause of your `SELECT` statement. In such a situation, you can use the `EXECUTE IMMEDIATE` statement.

Another use case is for DDL, which cannot be executed in static SQL from within PL/SQL.

To call a TimesTen built-in procedure that returns a result set, create a record type and use `EXECUTE IMMEDIATE` with `BULK COLLECT` to fetch the results into an array.

[Example 2–15](#) provides a set of brief examples of `EXECUTE IMMEDIATE`. For additional examples, see "[Examples using EXECUTE IMMEDIATE](#)" on page 5-13.

For more information, see "EXECUTE IMMEDIATE Statement" in *Oracle Database PL/SQL Language Reference*.

---



---

### Notes:

- See "[Differences in TimesTen: transaction behavior](#)" on page 2-31 for important information.
  - As a DDL statement is being parsed to drop a procedure or a package, a timeout occurs if the procedure, or a procedure in the package, is still in use. After a call to a procedure, that procedure is considered to be in use until execution has returned to the user side. Any such deadlock times out after a short time.
  - You can also use the `DBMS_SQL` package for dynamic SQL. See "[DBMS\\_SQL](#)" on page 7-8.
- 
- 

### **Example 2–15 Using EXECUTE IMMEDIATE to execute PL/SQL**

This provides a set of examples using PL/SQL in `EXECUTE IMMEDIATE` statements. (The examples are independent of each other.)

Create a table and execute a DML statement on it within a PL/SQL block, specifying the input parameter through a `USING` clause. Then select the table to see the result.

```
Command> create table t(i int);
Command> declare
    i number := 1;
begin
    execute immediate 'begin insert into t values(:j);end;' using i;
end;
/
```

PL/SQL procedure successfully completed.

```
Command> select * from t;
< 1 >
1 row found.
```

Create a PL/SQL procedure `foo` then execute it in a PL/SQL block, specifying the input parameter through a `USING` clause:

```
Command> create or replace procedure foo(message varchar2) is
begin
    dbms_output.put_line(message);
end;
/
```

Procedure created.

```
Command> begin
    execute immediate 'begin foo(:b);end;' using 'hello';
end;
/
hello
```

PL/SQL procedure successfully completed.

Create a PL/SQL procedure `myprint` then execute it through a `CALL` statement, specifying the input parameter through a `USING` clause:

```
Command> declare
    a number := 1;
begin
    execute immediate 'call myprint(:b)' using a;
end;
/
```

myprint procedure got number 1

PL/SQL procedure successfully completed.

### Usage notes for dynamic SQL

Code that is executed through `EXECUTE IMMEDIATE` generally shares the same environment as the outer PL/SQL block, as in Oracle Database. In particular, be aware of the following. (These points apply to using `DBMS_SQL` as well as `EXECUTE IMMEDIATE`.)

- SQL and PL/SQL executed through `EXECUTE IMMEDIATE` run in the same transaction as the outer block.
- Any exception raised during execution of an `EXECUTE IMMEDIATE` statement is propagated to the outer block. Therefore, any errors on the error stack when the `EXECUTE IMMEDIATE` statement is executed are visible inside the outer block. This is useful for procedures such as `DBMS_UTILITY.FORMAT_ERROR_STACK`.
- Errors on the error stack before execution of a PL/SQL block in an `EXECUTE IMMEDIATE` statement are visible inside the block, for example by using `DBMS_UTILITY.FORMAT_ERROR_STACK`.
- The execution environment in which an `EXECUTE IMMEDIATE` statement executes is the same as for the outer block. PL/SQL and TimesTen parameters, `REF CURSOR` state, and package state from the `EXECUTE IMMEDIATE` statement are visible inside the outer block.

## FORALL and BULK COLLECT operations

*Bulk binding* is a powerful feature used in the execution of SQL statements from PL/SQL to move large amounts of data between SQL and PL/SQL. (This is different from binding parameters from an application program to PL/SQL.) With bulk binding, you bind arrays of values in a single operation rather than using a loop to perform `FETCH`, `INSERT`, `UPDATE`, and `DELETE` operations multiple times. TimesTen supports bulk binding, which can result in significant performance improvement.

Use the `FORALL` statement to bulk-bind input collections before sending them to the SQL engine. Use `BULK COLLECT` to bring back batches of results from SQL. You can bulk-collect into any type of PL/SQL collection, such as a varray, nested table, or associative array (index-by table). For additional information on collections, refer to ["Using collections in PL/SQL"](#) on page 3-5.

You can use the `%BULK_EXCEPTIONS` cursor attribute and the `SAVE EXCEPTIONS` clause with `FORALL` statements. `SAVE EXCEPTIONS` allows an `UPDATE`, `INSERT`, or `DELETE` statement to continue executing after it issues an exception (for example, a constraint error). Exceptions are collected into an array that you can examine using `%BULK_EXCEPTIONS` after the statement has executed. When you use `SAVE EXCEPTIONS`, if exceptions are encountered during the execution of the `FORALL` statement, then all rows in the collection are processed. When the statement finishes, an error is issued to indicate that at least one exception occurred. If you do not use `SAVE EXCEPTIONS`, then when an exception is issued during a `FORALL` statement, the statement returns the exception immediately and no other rows are processed.

Refer to ["Using FORALL Statement and BULK COLLECT Clause Together"](#) in *Oracle Database PL/SQL Language Reference* for more information on these features.

[Example 2-16](#) shows basic use of bulk binding and the `FORALL` statement. For more information and examples on bulk binding, see ["Examples using FORALL and BULK COLLECT"](#) on page 5-9.

### **Example 2-16 Using the FORALL statement**

In the following example, the PL/SQL program increases the salary for employees with IDs 100, 102, 104, or 110. The `FORALL` statement bulk-binds the collection.

```
Command> CREATE OR REPLACE PROCEDURE raise_salary (p_percent NUMBER) IS
    TYPE numlist_type IS TABLE OF NUMBER
        INDEX BY BINARY_INTEGER;
    v_id numlist_type; -- collection
BEGIN
    v_id(1) := 100; v_id(2) := 102; v_id(3) := 104; v_id(4) := 110;
    -- bulk-bind the associative array
    FORALL i IN v_id.FIRST .. v_id.LAST
        UPDATE employees
            SET salary = (1 + p_percent/100) * salary
            WHERE employee_id = v_id(i);
END;
/
```

Procedure created.

Find out salaries before executing the `raise_salary` procedure:

```
Command> SELECT salary FROM employees WHERE employee_id = 100 OR employee_id =
102 OR employee_id = 104 OR employee_id = 110;
< 24000 >
< 17000 >
< 6000 >
```

3 rows found.

Execute the procedure and verify results as follows.

```
Command> EXECUTE raise_salary (10);
```

PL/SQL procedure successfully completed.

```
Command> SELECT salary FROM employees WHERE employee_id = 100 or employee_id =
102 OR employee_id = 104 OR employee_id = 100;
< 26400 >
< 18700 >
< 6600 >
3 rows found.
```

## RETURNING INTO clause

You can use a `RETURNING INTO` clause, sometimes referred to as *DML returning*, with an `INSERT`, `UPDATE`, or `DELETE` statement to return specified columns or expressions, optionally including rowids, from rows that were affected by the action. This eliminates the need for a subsequent `SELECT` statement and separate round trip, in case, for example, you want to confirm what was affected or want the rowid after an insert or update.

A `RETURNING INTO` clause can be used with dynamic SQL (with `EXECUTE IMMEDIATE`) or static SQL.

Through the PL/SQL `BULK COLLECT` feature, the clause can return items from a single row into either a set of parameters or a record, or can return columns from multiple rows into a PL/SQL collection such as a varray, nested table, or associative array (index-by table). Parameters in the `INTO` part of the clause must be output only, not input/output. For information on collections, refer to ["Using collections in PL/SQL"](#) on page 3-5. For `BULK COLLECT`, see ["FORALL and BULK COLLECT operations"](#) on page 2-22 and ["Examples using FORALL and BULK COLLECT"](#) on page 5-9.

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

Also see ["Examples using RETURNING INTO"](#) on page 5-16.

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

## TimesTen PL/SQL with TimesTen Cache

When PL/SQL programs execute SQL statements, the SQL statements are processed by TimesTen in the same manner as when SQL is executed from applications written in other programming languages. All standard behaviors of TimesTen SQL apply. In a TimesTen Cache environment in TimesTen Classic, this includes the ability to use all cache features from PL/SQL. When PL/SQL accesses tables in cache groups, the normal rules for those tables apply. For example, issuing a `SELECT` statement against a cache instance in a dynamic cache group may cause the instance to be automatically loaded into TimesTen from Oracle Database.

In particular, be aware of the following points about this functionality.

- When you use static SQL in PL/SQL, any tables accessed must exist in TimesTen or the PL/SQL will not compile successfully. In the following example, `ABC` must exist in TimesTen.

```
begin
  insert into abc values(1, 'Y');
end;
```

- In a TimesTen Cache environment, there is the capability to use the TimesTen passthrough facility to automatically route SQL statements from TimesTen to Oracle Database. (See "Setting a passthrough level" in *Oracle TimesTen Application-Tier Database Cache User's Guide* for details of the passthrough facility.)

With `passthrough=1`, a statement can be passed through to Oracle Database if any accessed table does not exist in TimesTen. In PL/SQL, however, the statement would have to be executed using dynamic SQL.

Updating the preceding example, the following TimesTen PL/SQL block could be used to access ABC in Oracle Database with `passthrough=1`:

```
begin
  execute immediate 'insert into abc values(1, 'Y')';
end;
```

In this case, TimesTen PL/SQL can compile the block because the SQL statement is not examined at compile time.

- While PL/SQL can be executed in TimesTen, the TimesTen passthrough facility cannot be used to route PL/SQL blocks from TimesTen to Oracle Database. For example, when using TimesTen Cache with `passthrough=3`, statements executed on a TimesTen connection are routed to Oracle Database in most circumstances. In this scenario, you may not execute PL/SQL blocks from your application program, because TimesTen would attempt to forward them to Oracle Database, which is not supported. (In the `passthrough=1` example, it is just the SQL statement being routed to Oracle Database, not the block as a whole.)

---

---

**Important:** PL/SQL procedures and functions can use any of the following cache operations with either definer's rights or invoker's rights:

- Loading or refreshing a cache group with `commit every n rows`
- DML on AWT cache groups
- DML on non-propagated cache groups (user managed cache groups without `PROPAGATE` enabled)
- `SELECT` on cache group tables that do not invoke passthrough or dynamic load
- `UNLOAD CACHE GROUP`

PL/SQL procedures or functions that use any of the following cache operations must use invoker's rights (`AUTHID CURRENT_USER`): passthrough, dynamic loading of a cache group, loading or refreshing a cache group using `WITH ID`, DDL on cache groups, DML on SWT cache groups, or `FLUSH CACHE GROUP`.

See "Definer's rights and invoker's rights (`AUTHID` clause)" in *Oracle TimesTen In-Memory Database Security Guide*.

---

---

## Large objects (LOBs)

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

PL/SQL language features support LOBs in TimesTen as they do in Oracle Database, unless noted otherwise.

This section provides a brief overview of LOBs and discusses their use in PL/SQL, covering the following topics.

- [About LOBs](#)
- [LOB locators](#)
- [Temporary LOBs](#)
- [Differences between TimesTen LOBs and Oracle Database LOBs](#)
- [Using LOBs](#)
- [PL/SQL package support for LOBs](#)

---



---

**Note:** TimesTen Scaleout supports LOBs as PL/SQL variables but not as database columns.

---



---

You can also refer to the following:

- "LOB data types" in *Oracle TimesTen In-Memory Database SQL Reference* for additional information about LOBs in TimesTen
- *Oracle Database SecureFiles and Large Objects Developer's Guide* for general information about programming with LOBs (but not specific to TimesTen functionality)

### About LOBs

A LOB is a large binary object (BLOB) or character object (CLOB or NCLOB). In TimesTen, a BLOB can be up to 16 MB and a CLOB or NCLOB up to 4 MB. LOBs in TimesTen have essentially the same functionality as in Oracle Database, except as noted otherwise. (See "[Differences between TimesTen LOBs and Oracle Database LOBs](#)" on page 2-26.)

LOBs may be either persistent or temporary. A persistent LOB exists in a LOB column in the database. A temporary LOB exists only within an application.

### LOB locators

In PL/SQL, a LOB consists of a LOB locator and a LOB value. The locator is an opaque structure that acts as a handle to the value. When an application uses a LOB in an operation such as passing a LOB as a parameter, it is passing the locator, not the actual value.

---



---

**Important:** LOB manipulations through APIs that use LOB locators result in usage of TimesTen temporary space. Any significant number of such manipulations may necessitate a size increase for the TimesTen temporary data region. See "TempSize" in *Oracle TimesTen In-Memory Database Reference*.

---



---

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

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

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

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

### Temporary LOBs

A PL/SQL block can create a temporary LOB explicitly, for its own use. In TimesTen, the lifetime of such a LOB does not extend past the end of the transaction in which it is created (as is the case with the lifetime of any LOB locator in TimesTen).

A temporary LOB may also be created implicitly by TimesTen. For example, if a `SELECT` statement selects a LOB concatenated with an additional string of characters, TimesTen implicitly creates a temporary LOB to contain the concatenated data. Note that a temporary LOB is a server-side object. TimesTen has no concept of client-side LOBs.

Temporary LOBs are stored in the TimesTen temporary data region.

See "CREATETEMPORARY procedures" in *Oracle TimesTen In-Memory Database PL/SQL Packages Reference* for how to create temporary LOBs.

### Differences between TimesTen LOBs and Oracle Database LOBs

Be aware of the following:

- A key difference between the TimesTen LOB implementation and the Oracle Database implementation is that in TimesTen, LOB locators do not remain valid past the end of the transaction. All LOB locators are invalidated after a commit or rollback, whether explicit or implicit. This includes after any DDL statement.
- TimesTen does not support BFILES, SecureFiles, array reads and writes for LOBs, or callback functions for LOBs.
- In TimesTen, the `DBMS_LOB FRAGMENT` procedures are not supported, so you can write data into the middle of a LOB only by overwriting previous data. There is no

functionality to insert data into the middle of a LOB and move previous data, beginning at that point, higher in the LOB correspondingly. Similarly, in TimesTen you can delete data from the middle of a LOB only by overwriting previous data with zeros or null data. There is no functionality to remove data from the middle of a LOB and move previous data, beginning at that point, lower in the LOB correspondingly. In either case in TimesTen, the size of the LOB does not change, except in the circumstance where from the specified offset there is less space available in the LOB than there is data to write. (In Oracle Database there is functionality for either mode, either overwriting and not changing the size of the LOB, or inserting or deleting and changing the size of the LOB.)

- TimesTen does not support binding arrays of LOBs.
- TimesTen does not support batch processing of LOBs.
- Relevant to BLOBs, there are differences in the usage of hexadecimal literals in TimesTen. See the description of *HexadecimalLiteral* in "Constants" in *Oracle TimesTen In-Memory Database SQL Reference*.

## Using LOBs

The following shows basic use of a CLOB. Assume a table defined and populated as follows, with a BLOB column (not used here) and a CLOB column:

```
Command> create table t1 (a int, b blob, c clob);
Command> insert into t1(a,b,c) values(1, 0x123451234554321, 'abcde');
1 row inserted.
Command> commit;
```

Select a CLOB from the table and display it:

```
Command> declare
    myclob clob;
begin
    select c into myclob from t1 where a=1;
    dbms_output.put_line('CLOB selected from table t1 is: ' || myclob);
end;
/
CLOB selected from table t1 is: abcde
```

PL/SQL procedure successfully completed.

The following tries to display the temporary CLOB again after a `commit` statement has ended the transaction, showing that the LOB locator becomes invalid from that point:

```
Command> declare
    myclob clob;
begin
    select c into myclob from t1 where a=1;
    dbms_output.put_line('CLOB selected from table t1 is: ' || myclob);
    commit;
    dbms_output.put_line('CLOB after end of transaction is: ' || myclob);
end;
/
1806: invalid LOB locator specified
8507: ORA-06512: at line 8
CLOB selected from table t1 is: abcde
The command failed.
```

## PL/SQL package support for LOBs

TimesTen supports subprograms of the `DBMS_LOB` package for manipulation of LOB data.

See ["DBMS\\_LOB"](#) on page 7-2 in this document for a list and descriptions of these subprograms. See ["DBMS\\_LOB"](#) in *Oracle TimesTen In-Memory Database PL/SQL Packages Reference* for further information.

## Passthrough LOBs

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

- TimesTen LOB size limitations do not apply to storage of passthrough LOBs, but do apply to binding. Also, if a passthrough LOB is copied to a TimesTen LOB, such as through `DBMS_LOB.COPY`, the size limit applies to the copy.  
  
An attempt to copy a passthrough LOB to a TimesTen LOB when the passthrough LOB is larger than the TimesTen LOB size limit results in an error.
- As with TimesTen local LOBs, a locator for a passthrough LOB does not remain valid past the end of the transaction.

See ["DBMS\\_LOB"](#) on page 7-2 in this document for information about `DBMS_LOB` support for passthrough LOBs.

## Use of cursors in PL/SQL programs

A cursor, either explicit or implicit, is used to handle the result set of a `SELECT` statement. As a programmer, you can declare an explicit cursor to manage queries that return multiple rows of data. PL/SQL declares and opens an implicit cursor for any `SELECT` statement that is not associated with an explicit cursor.

---

---

**Important:** Be aware that in TimesTen, any operation that ends your transaction closes all cursors associated with the connection. This includes any `COMMIT` or `ROLLBACK` statement and any DDL statement. This results in autocommits of DDL statements. See ["Differences in TimesTen: transaction behavior"](#) on page 2-31 for additional information.

---

---

[Example 2-17](#) shows basic use of a cursor. See ["Examples using cursors"](#) on page 5-6 for additional information and examples. Also see ["PL/SQL REF CURSORS"](#) on page 3-7.

### **Example 2-17 Using a cursor to retrieve information about an employee**

Declare a cursor `c1` to retrieve the last name, salary, hire date, and job class for the employee whose employee ID is 120.

```
Command> DECLARE
          CURSOR c1 IS
              SELECT last_name, salary, hire_date, job_id FROM employees
              WHERE employee_id = 120;
          --declare record variable that represents a row
          --fetched from the employees table
          employee_rec c1%ROWTYPE;
BEGIN
```

```

-- open the explicit cursor
-- and use it to fetch data into employee_rec
OPEN c1;
FETCH c1 INTO employee_rec;
DBMS_OUTPUT.PUT_LINE('Employee name: ' || employee_rec.last_name);
CLOSE c1;
END;
/
Employee name: Weiss

PL/SQL procedure successfully completed.

```

## Wrapping PL/SQL source code

*Wrapping* is the process of hiding PL/SQL source code. You can wrap PL/SQL source code with the `wrap` utility, which processes an input SQL file and wraps only the PL/SQL units in the file, such as a package specifications, package bodies, functions, and procedures.

Consider the following example, which uses a file `wrap_test.sql` to define a procedure named `wraptest`. It then uses the `wrap` utility to process `wrap_test.sql`. The procedure is created with the source code hidden, and executes successfully. As a final step, the `ALL_OBJECTS` view is queried to see the wrapped source code.

Here are the contents of `wrap_test.sql`:

```

CREATE OR REPLACE PROCEDURE wraptest IS
  TYPE emp_tab IS TABLE OF employees%ROWTYPE INDEX BY PLS_INTEGER;
  all_emps emp_tab;
BEGIN
  SELECT * BULK COLLECT INTO all_emps FROM employees;
  FOR i IN 1..10
  LOOP
    DBMS_OUTPUT.PUT_LINE('Emp Id: ' || all_emps(i).employee_id);
  END LOOP;
END;
/

```

In the example that follows, "%" is the UNIX prompt, "Command>" is the `ttIsql` prompt, and user input is shown in bold.

```
% wrap iname=wrap_test.sql
```

```
PL/SQL Wrapper: Release 12.1.0.2.0- Production on Wed Sep 14 12:59:27 2019
```

```
Copyright (c) 1993, 2019, Oracle. All rights reserved.
```

```
Processing wrap_test.sql to wrap_test.plb
```

```
% cat wrap_test.plb
```

```

CREATE OR REPLACE PROCEDURE wraptest wrapped
a000000
1
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd

```



```

Command> SET SERVEROUTPUT ON
Command> BEGIN
           wraptest();
           END;
           /
Emp Id: 100
Emp Id: 101
Emp Id: 102
Emp Id: 103
Emp Id: 104
Emp Id: 105
Emp Id: 106
Emp Id: 107
Emp Id: 108
Emp Id: 109

PL/SQL procedure successfully completed.

Command> SELECT text FROM all_source WHERE name = 'WRAPTEST';

< PROCEDURE wraptest wrapped
a000000
1
abcd
7
124 12c
YZ6L0v2ntFaqtW8hSJD5IHIYccwg+nwNfZqfHQcv/9kMJyznwdLh8FepNXpWS1fzVBDkTke
LWlhFdFzCMfmmJ5GGrCwrqngEhfRpg7ck5Dzsf7sDlnQeE3QGmb/yu9Dec1+JO2kOM1x3dq
BuC7fR2f5sjDtBeDXiGCC0kJ5QBVregtoBckZNO9MoiWS4w0jF6T1CPY0Aoi/KUwxC8S8I8n
amF5xGQDCYTDajs77orIGEqTX747k0YAO+r1e9adGUsVgZK1ONcTM/+Wit+LYKi7b03eJxdB
+aaKn/Lh

>
1 row found.

```

## Differences in TimesTen: transaction behavior

In TimesTen, any operation that ends your transaction closes all cursors associated with the connection. This includes the following:

- Any COMMIT or ROLLBACK statement
- Any DDL statement

For example, consider the following scenario, where you want to recompile a set of procedures. This would not work, because the first time `ALTER PROCEDURE` is executed, the cursor (`pnamecurs`) would be closed:

```
declare
  cursor pnamecurs is select * from all_objects where object_name like 'MYPROC%';
begin
  for rec in pnamecurs loop
    execute immediate 'alter procedure ' || rec.object_name || ' compile';
  end loop;
end;
```

Instead, you can do something like the following, which fetches all the procedure names into an internal table then executes `ALTER PROCEDURE` on them with no active cursor.

```
declare
  cursor pnamecurs is select * from all_objects where object_name like 'MYPROC%';
  type tbl is table of c%rowtype index by binary_integer;
  myprocs tbl;

begin
  open pnamecurs;
  fetch pnamecurs bulk collect into myprocs;
  close pnamecurs;
  for i in 1..myprocs.count loop
    execute immediate 'alter procedure ' || myprocs(i).object_name || ' compile';
  end loop;
end;
```

---



---

## Data Types in PL/SQL in TimesTen

This chapter focuses on the range of data types available to you for manipulating data in PL/SQL, TimesTen SQL, and your application programs.

TimesTen supports PL/SQL data types and the interactions between PL/SQL data types, TimesTen data types, and client application program data types. Data type conversions and data type mappings are supported.

See the end of the chapter for TimesTen-specific considerations.

Topics in this chapter include the following:

- [Understanding the data type environments](#)
- [Understanding and using PL/SQL data types](#)
- [Data type conversion](#)
- [Differences in TimesTen: data type considerations](#)

### Understanding the data type environments

There are three distinct environments to consider when discussing data types:

- PL/SQL programs that contain variables and constants that use PL/SQL data types
- TimesTen SQL statements that use database rows, columns, and constants  
These elements are expressed using TimesTen SQL data types.
- Application programs that interact with the database and the PL/SQL programming language

Application programs are written in programming languages such as C and Java and contain variables and constants that use data types from these programming languages.

[Table 3–1](#) summarizes the environments and gives examples of data types for each environment.

**Table 3–1** Summarizing the data type environments

Environment	Data type examples
PL/SQL programs	NUMBER, PLS_INTEGER, VARCHAR2, STRING, DATE, TIMESTAMP
TimesTen SQL statements	TT_BIGINT, TT_INTEGER, BINARY_FLOAT, VARCHAR2, DATE, TIMESTAMP
Application programs	int, double, String

## Understanding and using PL/SQL data types

This section describes the PL/SQL data types that are supported in PL/SQL programs. It does not describe the data types supported in TimesTen SQL statements. For information on data types supported in TimesTen SQL statements, see "Data Types" in *Oracle TimesTen In-Memory Database SQL Reference*.

The following topics are covered in this section:

- [PL/SQL data type categories](#)
- [Predefined PL/SQL scalar data types](#)
- [PL/SQL composite data types](#)
- [PL/SQL REF CURSORS](#)

For additional information see "PL/SQL Data Types" in *Oracle Database PL/SQL Language Reference*.

### PL/SQL data type categories

In a PL/SQL block, every constant, variable, and parameter has a data type. PL/SQL provides predefined data types and subtypes and lets you define your own PL/SQL subtypes.

[Table 3–2](#) lists the categories of the predefined PL/SQL data types.

**Table 3–2** *Predefined PL/SQL data type categories*

Data type category	Description
Scalar	Single values with no internal components
Composite	Internal components that are either scalar or composite
Reference	Pointers to other data items such as REF CURSORS

---

**Note:** See "[Non-supported data types](#)" on page 3-13.

---

### Predefined PL/SQL scalar data types

Scalar data types store single values with no internal components. [Table 3–3](#) lists predefined PL/SQL scalar data types of interest, grouped by data type families.

**Table 3–3** *Predefined PL/SQL scalar data types*

Data type family	Data type name
NUMERIC	NUMBER
	PLS_INTEGER
	BINARY_FLOAT
	BINARY_DOUBLE
CHARACTER	CHAR[ACTER]
	VARCHAR2
	NCHAR (national character CHAR)
	NVARCHAR2 (national character VARCHAR2)
BINARY	RAW

**Table 3–3 (Cont.) Predefined PL/SQL scalar data types**

Data type family	Data type name
BOOLEAN	BOOLEAN <b>Note:</b> You cannot bind BOOLEAN types in SQL statements.
DATETIME	DATE TIMESTAMP
INTERVAL	INTERVAL YEAR TO MONTH INTERVAL DAY TO SECONDS
ROWID	ROWID
LOB	BLOB (binary LOB) CLOB (character LOB) NCLOB (national character LOB)

---

**Note:** See "[Non-supported data types](#)" on page 3-13.

---

**Example 3–1 Declaring PL/SQL variables**

```

Command> DECLARE
    v_emp_job      VARCHAR2 (9);
    v_count_loop  BINARY_INTEGER := 0;
    v_dept_total_sal NUMBER (9,2) := 0;
    v_orderdate   DATE := SYSDATE + 7;
    v_valid       BOOLEAN NOT NULL := TRUE;
    ...

```

**PLS\_INTEGER and BINARY\_INTEGER data types**

The PLS\_INTEGER and BINARY\_INTEGER data types are identical and are used interchangeably in this document.

The PLS\_INTEGER data type stores signed integers in the range -2,147,483,648 through 2,147,483,647 represented in 32 bits. It has the following advantages over the NUMBER data type and subtypes:

- PLS\_INTEGER values require less storage.
- PLS\_INTEGER operations use hardware arithmetic, so they are faster than NUMBER operations, which use library arithmetic.

For efficiency, use PLS\_INTEGER values for all calculations that fall within its range. For calculations outside the PLS\_INTEGER range, use INTEGER, a predefined subtype of the NUMBER data type.

See "PLS\_INTEGER and BINARY\_INTEGER Data Types" in *Oracle Database PL/SQL Language Reference* for additional information.

---

**Note:** When a calculation with two PLS\_INTEGER data types overflows the PLS\_INTEGER range, an overflow exception is raised even if the result is assigned to a NUMBER data type.

---

### **SIMPLE\_INTEGER data type**

`SIMPLE_INTEGER` is a predefined subtype of the `PLS_INTEGER` data type that has the same range as `PLS_INTEGER` (-2,147,483,648 through 2,147,483,647) and has a `NOT NULL` constraint. It differs from `PLS_INTEGER` in that it does not overflow.

You can use `SIMPLE_INTEGER` when the value is never null and overflow checking is unnecessary. Without the overhead of checking for null values and overflow, `SIMPLE_INTEGER` provides better performance than `PLS_INTEGER`.

See "SIMPLE\_INTEGER Subtype of PLS\_INTEGER" in *Oracle Database PL/SQL Language Reference* for additional information.

### **ROWID data type**

Each row in a table has a unique identifier known as its *rowid*.

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

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

### **LOB data types**

The LOB (large object) type family includes `CLOB` (character LOBs), `NCLOB` (national character LOBs), and `BLOB` (binary LOBs).

A LOB consists of a LOB locator and a LOB value. The locator acts as a handle to the value. When an application selects a LOB or passes a LOB as a parameter, for example, it is using the locator, not the actual value.

LOBs may be either persistent or temporary. A persistent LOB exists in the database, in a particular row of a LOB column. A temporary LOB is used internally within a program, but could then be inserted into a LOB column in the database to become a persistent LOB.

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

Also see "[Large objects \(LOBs\)](#)" on page 2-25 for usage information.

## **PL/SQL composite data types**

Composite types have internal components that can be manipulated individually, such as the elements of an array, record, or table.

TimesTen supports the following composite data types:

- Associative array (index-by table)
- Nested table
- Varray
- Record

Associative arrays, nested tables, and varrays are also referred to as *collections*.

The following sections discuss the use of composite data types:

- [Using collections in PL/SQL](#)
- [Using records in PL/SQL](#)

- [Using associative arrays from applications](#)

See "PL/SQL Collections and Records" in *Oracle Database PL/SQL Language Reference* for additional information.

### Using collections in PL/SQL

You can declare collection data types similar to arrays, sets, and hash tables found in other languages. A collection is an ordered group of elements, all of the same type. Each element has a unique subscript that determines its position in the collection.

In PL/SQL, array types are known as *varrays* (variable size arrays), set types are known as *nested tables*, and hash table types are known as *associative arrays* or *index-by tables*. These are all collection types.

#### Example 3–2 Using a PL/SQL collection type

This example declares collection type `staff_list` as a table of `employee_id`, then uses the collection type in a loop and in the `WHERE` clause of the `SELECT` statement.

```
Command> DECLARE
    TYPE staff_list IS TABLE OF employees.employee_id%TYPE;
    staff staff_list;
    lname employees.last_name%TYPE;
    fname employees.first_name%TYPE;
BEGIN
    staff := staff_list(100, 114, 115, 120, 122);
    FOR i IN staff.FIRST..staff.LAST LOOP
        SELECT last_name, first_name INTO lname, fname FROM employees
            WHERE employees.employee_id = staff(i);
        DBMS_OUTPUT.PUT_LINE (TO_CHAR(staff(i)) ||
            ': ' || lname || ', ' || fname );
    END LOOP;
END;
/
100: King, Steven
114: Raphaely, Den
115: Khoo, Alexander
120: Weiss, Matthew
122: Kaufling, Payam
```

PL/SQL procedure successfully completed.

Any collections can be passed between PL/SQL subprograms as parameters. In TimesTen, however, only associative arrays can be passed between PL/SQL and applications written in other languages. (See "[Using associative arrays from applications](#)" below.)

You can use collections to move data in and out of TimesTen tables using bulk SQL.

### Using records in PL/SQL

Records are composite data structures that have fields with different data types. You can pass records to subprograms with a single parameter. You can also use the `%ROWTYPE` attribute to declare a record that represents a row in a table or a row from a query result set, without specifying the names and types for the fields, as shown in [Example 2–2](#) on page 2-3.

#### Example 3–3 Declaring a record type

Declare various record types.

```

Command> DECLARE
        TYPE timerec IS RECORD (hours SMALLINT, minutes SMALLINT);
        TYPE meetin_typ IS RECORD (
            date_held DATE,
            duration timerec, -- nested record
            location VARCHAR2(20),
            purpose VARCHAR2(50));
        BEGIN
            ...
        END;
    /

```

## Using associative arrays from applications

Associative arrays, formerly known as index-by tables or PL/SQL tables, are supported as IN, OUT, or IN OUT bind parameters in TimesTen PL/SQL, such as from an OCI, Pro\*C/C++, or JDBC application. This enables arrays of data to be passed efficiently between an application and the database.

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

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

```

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

```

See [Example 3-4](#) below and "[Using associative arrays](#)" on page 5-5 for examples.

Also see "Associative array bindings in TimesTen OCI" and "Associative array bindings in TimesTen Pro\*C/C++" in *Oracle TimesTen In-Memory Database C Developer's Guide*, and "Binding associative arrays" in *Oracle TimesTen In-Memory Database Java Developer's Guide*.

For general information about associative arrays, see "Associative Arrays" in *Oracle Database PL/SQL Language Reference*.

---



---

**Notes:** Note the following restrictions in TimesTen:

- The following types are not supported in binding associative arrays: LOBs, REF CURSORS, TIMESTAMP, ROWID.
  - Associative array binding is not allowed in passthrough statements.
  - General bulk binding of arrays is not supported in TimesTen programmatic APIs. Varrays and nested tables are not supported as bind parameters.
- 
- 

The following example does some simple manipulation of an associative array, effectively binding it from ttIsql and printing the array.

**Example 3-4 Using an associative array from tllsql**

```

Command> var lngvc[1000] varchar2(30);
Command> declare
    TYPE VARCHARARRTYP IS TABLE OF VARCHAR2(30) INDEX BY BINARY_INTEGER;
    x VARCHARARRTYP;
begin
    x := :lngvc;
    x ( 1 ) := 'One';
    x ( 10 ) := 'Ten';
    :lngvc := x;
end;
/

```

PL/SQL procedure successfully completed.

```

Command> print lngvc;
LNGVC                : ARRAY [ 1000 ] (Current Size 10)
LNGVC[1] : One
LNGVC[2] : <NULL>
LNGVC[3] : <NULL>
LNGVC[4] : <NULL>
LNGVC[5] : <NULL>
LNGVC[6] : <NULL>
LNGVC[7] : <NULL>
LNGVC[8] : <NULL>
LNGVC[9] : <NULL>
LNGVC[10] : Ten

```

**PL/SQL REF CURSORS**

A REF CURSOR is a handle to a cursor over a SQL result set that can be passed as a parameter between PL/SQL and an application. TimesTen supports OUT REF CURSORS, from PL/SQL to the application. The application would open the REF CURSOR within PL/SQL, pass it from there through the applicable API, and fetch the result set.

TimesTen supports REF CURSORS in ODBC, JDBC, ODP.NET, OCI, Pro\*C/C++, and TTCclasses for either direct connections or client/server connections. REF CURSORS are also discussed in the following TimesTen documents:

- "Working with REF CURSORS" in *Oracle TimesTen In-Memory Database C Developer's Guide*
- "Working with REF CURSORS" in *Oracle TimesTen In-Memory Database Java Developer's Guide*
- "Working with REF CURSORS" in *Oracle TimesTen In-Memory Database TTCclasses Guide*

---

**Note:** TimesTen supports one OUT REF CURSOR per statement.

---

You can define a REF CURSOR in PL/SQL in TimesTen as you would in Oracle Database. (See "Cursor Variables" in *Oracle Database PL/SQL Language Reference*.) It is typical to use REF CURSOR as a metatype, where you define a "strong" (specific) REF CURSOR type tailored to your data, then declare a cursor variable of that type. For example:

```

Command> DECLARE
    TYPE DeptCurTyp IS REF CURSOR RETURN departments%ROWTYPE;

```

```
dept_cv DeptCurTyp; -- declare cursor variable
...
```

**Example 3-5 Use a REF CURSOR as an output parameter**

This example creates a procedure GET\_EMP in a package FOO\_PACK to retrieve information about employees from the employees table. The procedure declares a REF CURSOR type cursor\_out, then uses that type for the output parameter.

First specify the package definition, REF CURSOR type, and procedure definition.

```
create or replace package foo_pack is
  type cursor_out is ref cursor;
  procedure get_emp (results out cursor_out);
end foo_pack;
```

Then specify the package body and procedure implementation.

```
create or replace package body foo_pack as
  procedure get_emp (results out cursor_out) is
  begin
    open results for select employee_id, last_name from employees
                    where employee_id < 110 order by last_name;
  end get_emp;
end foo_pack;
```

Declare a REF CURSOR variable for the output, execute the procedure, and display the results. Note that outside of PL/SQL, you can declare only "weak" (generic) REF CURSORS.

```
Command> var proc_result refcursor;
Command> exec foo_pack.get_emp(:proc_result);
```

PL/SQL procedure successfully completed.

```
Command> print proc_result;
PROC_RESULT          :
< 105, Austin >
< 102, De Haan >
< 104, Ernst >
< 109, Faviet >
< 108, Greenberg >
< 103, Hunold >
< 100, King >
< 101, Kochhar >
< 107, Lorentz >
< 106, Pataballa >
10 rows found.
```

Alternatively, you could declare a weakly typed REF CURSOR variable in FOO\_PACK:

```
create or replace package foo_pack is
  procedure get_emp (results out sys_refcursor);
end foo_pack;

create or replace package body foo_pack as
  procedure get_emp (results out sys_refcursor) is
  begin
    open results for select employee_id, last_name from employees
                    where employee_id < 110 order by last_name;
  end get_emp;
end foo_pack;
```

## Data type conversion

This section covers the following data type conversions:

- [Conversion between PL/SQL data types](#)
- [Conversion between application data types and PL/SQL or SQL data types](#)

Also see type conversion information under "[Differences in TimesTen: data type considerations](#)" on page 3-10.

### Conversion between PL/SQL data types

TimesTen supports implicit and explicit conversions between PL/SQL data types.

Consider this example: The variable `v_sal_hike` is of type `VARCHAR2`. When calculating the total salary, PL/SQL first converts `v_sal_hike` to `NUMBER` then performs the operation. The result is of type `NUMBER`. PL/SQL uses implicit conversion to obtain the correct result.

```
Command> DECLARE
          v_salary NUMBER (6) := 6000;
          v_sal_hike VARCHAR2(5) := '1000';
          v_total_salary v_salary%TYPE;
BEGIN
v_total_salary := v_salary + v_sal_hike;
DBMS_OUTPUT.PUT_LINE (v_total_salary);
end;
/
7000
```

PL/SQL procedure successfully completed.

---

**Note:** Also see "[Date and timestamp formats: NLS\\_DATE\\_FORMAT and NLS\\_TIMESTAMP\\_FORMAT](#)" on page 3-12.

---

### Conversion between application data types and PL/SQL or SQL data types

TimesTen supports data type conversions between application program data types and PL/SQL data types, and between application program data types and TimesTen SQL data types. For SQL, the conversions are the same whether SQL is invoked by your PL/SQL program or is invoked directly by your application.

As an example, [Table 3-4](#) shows a few representative data type mappings from an application using the ODBC API to PL/SQL program data types. For more information about ODBC-to-PL/SQL type mappings, refer to "Determination of parameter type assignments and type conversions" in *Oracle TimesTen In-Memory Database C Developer's Guide*.

**Table 3-4** *Sampling of ODBC SQL to PL/SQL type mapping*

ODBC type	PL/SQL type
SQL_BINARY	RAW (Bound precision is used.)
SQL_CHAR	CHAR (Bound precision is used.)
SQL_DATE	DATE
SQL_DECIMAL	NUMBER
SQL_DOUBLE	NUMBER

**Table 3–4 (Cont.) Sampling of ODBC SQL to PL/SQL type mapping**

ODBC type	PL/SQL type
SQL_FLOAT	BINARY_DOUBLE
SQL_INTEGER	PLS_INTEGER
SQL_REFCURSOR	REF CURSOR
SQL_TIMESTAMP	TIMESTAMP (Bound scale is used.)
SQL_VARCHAR	VARCHAR2 (Bound precision is used.)

**Example 3–6 ODBC to PL/SQL data type conversions**

Consider a scenario where your C program uses the ODBC API and your goal is to bind your C variable of type VARCHAR2 to a PL/SQL variable of type NUMBER. TimesTen performs the implicit conversion for you.

```
Command> VARIABLE c_var VARCHAR2 (30) := '961';
Command> DECLARE v_var NUMBER;
          BEGIN
              v_var := :c_var;
              DBMS_OUTPUT.PUT_LINE (v_var);
          END;
          /
961
```

PL/SQL procedure successfully completed.

**Example 3–7 ODBC to TimesTen SQL data type conversions**

This example creates a table with a column of type TT\_BIGINT and uses PL/SQL to invoke the TimesTen SQL INSERT statement. A bind variable of type SQL\_VARCHAR is used in the INSERT statement. The conversions are the same as the conversions that would occur if your application invoked the INSERT statement directly.

```
Command> CREATE TABLE conversion_test2 (Col1 TT_BIGINT);
Command> VARIABLE v_var VARCHAR2 (100) := '1000';
Command> BEGIN
          INSERT INTO conversion_test2 VALUES (:v_var);
        END;
        /
```

PL/SQL procedure successfully completed.

```
Command> SELECT * FROM conversion_test2;
< 1000 >
1 row found.
```

## Differences in TimesTen: data type considerations

This section covers the following TimesTen-specific considerations regarding data type support and type conversions:

- [Conversion between PL/SQL and TimesTen SQL data types](#)
- [Date and timestamp formats: NLS\\_DATE\\_FORMAT and NLS\\_TIMESTAMP\\_FORMAT](#)
- [Non-supported data types](#)

## Conversion between PL/SQL and TimesTen SQL data types

TimesTen supports conversions between PL/SQL data types and TimesTen SQL data types.

Table 3–5 shows supported data type conversions, with PL/SQL types along the top and SQL types down the left side. The data types are grouped by data type families, with columns referring to PL/SQL type families and rows referring to TimesTen type families. "Yes" indicates that a conversion is possible between the two families. Supported conversions are bidirectional.

**Table 3–5 Supported conversions between PL/SQL and TimesTen SQL data types**

Type Family	NUMERIC	CHARACTER	BINARY	DATETIME	INTERVAL	ROWID
NUMERIC	Yes	Yes	No	No	No	No
CHARACTER	Yes	Yes	Yes	Yes	Yes	Yes
DATETIME	No	Yes	No	Yes	No	No
TIME	No	Yes	No	No	No	No
ROWID	No	Yes	No	No	No	Yes
BINARY	No	Yes	Yes	No	No	Yes

Table 3–6 that follows summarizes the TimesTen data types and suggestions for PL/SQL type mappings.

**Table 3–6 Data type usage and sizes**

TimesTen data type	Description
TT_TINYINT	This is for unsigned integers ranging from 0 to 255. Numeric overflows can occur if you insert a value with type PL/SQL NUMBER or PL/SQL PLS_INTEGER (or BINARY_INTEGER) into a TT_TINYINT column.
TT_SMALLINT	This is for signed 16-bit integers in the range -32,768 to 32,767. Numeric overflows can occur if you insert a value with type PL/SQL NUMBER or PL/SQL PLS_INTEGER (or BINARY_INTEGER) into a TT_SMALLINT column.
TT_INTEGER	This is for signed integers in the range -2,147,483,648 to 2,147,483,647. This is equivalent to PLS_INTEGER.
TT_BIGINT	This is for signed eight-byte integers in the range -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. Use PL/SQL NUMBER. A PL/SQL PLS_INTEGER (or BINARY_INTEGER) variable could overflow.
NUMBER, BINARY_FLOAT, BINARY_DOUBLE	Use when floating point precision is required.
Character types	All PL/SQL character types can hold up to 32,767 bytes of data. <ul style="list-style-type: none"> <li>■ TimesTen CHAR can hold up to 8300 bytes.</li> <li>■ TimesTen NCHAR can hold up to 4150 characters (8300 bytes).</li> <li>■ TimesTen VARCHAR2 can hold up to 4,194,304 bytes.</li> <li>■ TimesTen NVARCHAR2 can hold up to 2,097,152 characters (4,194,304 bytes).</li> </ul>

**Table 3–6 (Cont.) Data type usage and sizes**

TimesTen data type	Description
Datetime, interval, and time types	Use the TO_CHAR and TO_DATE built-in functions when you require a format that is different than the default format used when converting these types to and from character types.
Binary types	<ul style="list-style-type: none"> <li>■ TimesTen BINARY can hold up to 8300 bytes.</li> <li>■ TimesTen VARBINARY can hold up to 4,194,304 bytes.</li> <li>■ RAW and LONG RAW can hold up to 32,767 bytes.</li> </ul>

---

**Note:** See "[Non-supported data types](#)" on page 3-13.

---

**Example 3–8 Conversions between TimesTen SQL data types and PL/SQL data types**

Consider the case where you have a table with two columns. Col1 has a data type of TT\_INTEGER and Col2 has a data type of NUMBER. In your PL/SQL program, you declare two variables: v\_var1 of type PLS\_INTEGER and v\_var2 of type VARCHAR2. The goal is to SELECT the row of data from your table into the two PL/SQL variables.

Data type conversions occur when you execute the SELECT statement. Col1 is converted from a TimesTen SQL TT\_INTEGER type into a PLS\_INTEGER type. Col2 is converted from a TimesTen SQL NUMBER type into a PL/SQL VARCHAR2 type. The query executes successfully.

```
Command> CREATE TABLE test_conversion (Col1 TT_INTEGER, Col2 NUMBER);
Command> INSERT INTO test_conversion VALUES (100, 20);
1 row inserted.
```

```
Command> DECLARE
    v_var1 PLS_INTEGER;
    v_var2 VARCHAR2 (100);
BEGIN
    SELECT Col1, Col2 INTO v_var1, v_var2 FROM test_conversion;
    DBMS_OUTPUT.PUT_LINE (v_var1);
    DBMS_OUTPUT.PUT_LINE (v_var2);
END;
/

100
20
```

PL/SQL procedure successfully completed.

**Date and timestamp formats: NLS\_DATE\_FORMAT and NLS\_TIMESTAMP\_FORMAT**

TimesTen does not support user-specified NLS\_DATE\_FORMAT and NLS\_TIMESTAMP\_FORMAT settings.

- NLS\_DATE\_FORMAT is always 'yyyy-mm-dd'.
- NLS\_TIMESTAMP\_FORMAT is always 'yyyy-mm-dd hh:mi:ss.ff6' (fractional seconds to six decimal places).

You can use the SQL and PL/SQL TO\_DATE and TO\_CHAR functions to specify other desired formats. See "Expressions" in *Oracle TimesTen In-Memory Database SQL Reference* for details of these functions.

## Non-supported data types

Note the following non-support of data types:

- PL/SQL data type categories: PL/SQL in TimesTen does not support Internet data types (`XMLType`, `URIType`, `HttpURIType`) or "Any" data types (`AnyType`, `AnyData`, `AnyDataSet`).
- PL/SQL scalar data types: TimesTen does not support the PL/SQL data types `TIMESTAMP WITH [LOCAL] TIME ZONE` and `UROWID`.



---

---

## Errors and Exception Handling

This chapter describes the flexible error trapping and error handling you can use in your PL/SQL programs.

For more information on error-handling and exceptions in PL/SQL, see "PL/SQL Error Handling" in *Oracle Database PL/SQL Language Reference*.

See the end of this chapter for TimesTen-specific considerations.

The following topics are covered:

- [Understanding exceptions](#)
- [Trapping exceptions](#)
- [Retrying after transient errors \(PL/SQL\)](#)
- [Showing errors in ttlsq](#)
- [Differences in TimesTen: exception handling and error behavior](#)

### Understanding exceptions

This section provides an overview of exceptions in PL/SQL programming, covering the following topics:

- [About exceptions](#)
- [Exception types](#)

### About exceptions

An exception is a PL/SQL error that is raised during program execution, either implicitly by TimesTen or explicitly by your program. Handle an exception by trapping it with a handler or propagating it to the calling environment.

For example, if your `SELECT` statement returns multiple rows, TimesTen returns an error (exception) at runtime. As the following example shows, you would see TimesTen error 8507, then the associated ORA error message. (ORA messages, originally defined for Oracle Database, are similarly implemented by TimesTen.)

```
Command> DECLARE
  v_lname VARCHAR2 (15);
BEGIN
  SELECT last_name INTO v_lname
  FROM employees
  WHERE first_name = 'John';
  DBMS_OUTPUT.PUT_LINE ('Last name is : ' || v_lname);
END;
```

```

/

8507: ORA-01422: exact fetch returns more than requested number of rows
8507: ORA-06512: at line 4
The command failed.

```

You can handle such exceptions in your PL/SQL block so that your program completes successfully. For example:

```

Command> DECLARE
    v_lname VARCHAR2 (15);
BEGIN
    SELECT last_name INTO v_lname
    FROM employees
    WHERE first_name = 'John';
    DBMS_OUTPUT.PUT_LINE ('Last name is : ' || v_lname);
EXCEPTION
    WHEN TOO_MANY_ROWS THEN
    DBMS_OUTPUT.PUT_LINE (' Your SELECT statement retrieved multiple
    rows. Consider using a cursor. ');
END;
/

Your SELECT statement retrieved multiple rows. Consider using a cursor.

PL/SQL procedure successfully completed.

```

## Exception types

There are three types of exceptions:

- Predefined exceptions are error conditions that are defined by PL/SQL.
- Non-predefined exceptions include any standard TimesTen errors.
- User-defined exceptions are exceptions specific to your application.

In TimesTen, these three types of exceptions are used in the same way as in Oracle Database.

Exception	Description	How to handle
Predefined TimesTen error	One of approximately 20 errors that occur most often in PL/SQL code	You are not required to declare these exceptions. They are predefined by TimesTen. TimesTen implicitly raises the error.
Non-predefined TimesTen error	Any other standard TimesTen error	These must be declared in the declarative section of your application. TimesTen implicitly raises the error and you can use an exception handler to catch the error.
User-defined error	Error defined and raised by the application	These must be declared in the declarative section. The developer raises the exception explicitly.

## Trapping exceptions

This section describes how to trap predefined TimesTen errors or user-defined errors.

## Trapping predefined TimesTen errors

Trap a predefined TimesTen error by referencing its predefined name in your exception-handling routine. PL/SQL declares predefined exceptions in the `STANDARD` package.

[Table 4-1](#) lists predefined exceptions supported by TimesTen, the associated ORA error numbers and `SQLCODE` values, and descriptions of the exceptions.

Also see "[Unsupported predefined errors](#)" on page 4-10.

**Table 4-1** Predefined exceptions

Exception name	Oracle Database error number	SQLCODE	Description
ACCESS_INTO_NULL	ORA-06530	-6530	Program attempted to assign values to the attributes of an uninitialized object.
CASE_NOT_FOUND	ORA-06592	-6592	None of the choices in the <code>WHEN</code> clauses of a <code>CASE</code> statement is selected and there is no <code>ELSE</code> clause.
COLLECTION_IS_NULL	ORA-06531	-6531	Program attempted to apply collection methods other than <code>EXISTS</code> to an uninitialized nested table or varray, or program attempted to assign values to the elements of an uninitialized nested table or varray.
CURSOR_ALREADY_OPENED	ORA-06511	-6511	Program attempted to open an already opened cursor.
DUP_VAL_ON_INDEX	ORA-00001	-1	Program attempted to insert duplicate values in a column that is constrained by a unique index.
INVALID_CURSOR	ORA-01001	-1001	There is an illegal cursor operation.
INVALID_NUMBER	ORA-01722	-1722	Conversion of character string to number failed.
NO_DATA_FOUND	ORA-01403	+100	Single row <code>SELECT</code> returned no rows or your program referenced a deleted element in a nested table or an uninitialized element in an associative array (index-by table).
PROGRAM_ERROR	ORA-06501	-6501	PL/SQL has an internal problem.
ROWTYPE_MISMATCH	ORA-06504	-6504	Host cursor variable and PL/SQL cursor variable involved in an assignment statement have incompatible return types.
STORAGE_ERROR	ORA-06500	-6500	PL/SQL ran out of memory or memory was corrupted.
SUBSCRIPT_BEYOND_COUNT	ORA-06533	-6533	A program referenced a nested table or varray using an index number larger than the number of elements in the collection.

**Table 4–1 (Cont.) Predefined exceptions**

Exception name	Oracle Database error number	SQLCODE	Description
SUBSCRIPT_OUTSIDE_LIMIT	ORA-06532	-6532	A program referenced a nested table or varray element using an index number that is outside the legal range (for example, -1).
SYS_INVALID_ROWID	ORA-01410	-1410	The conversion of a character string into a universal rowid failed because the character string does not represent a ROWID value.
TOO_MANY_ROWS	ORA-01422	-1422	Single row <code>SELECT</code> returned multiple rows.
VALUE_ERROR	ORA-06502	-6502	An arithmetic, conversion, truncation, or size constraint error occurred.
ZERO_DIVIDE	ORA-01476	-1476	A program attempted to divide a number by zero.

**Example 4–1 Using the ZERO\_DIVIDE predefined exception**

In this example, a PL/SQL program attempts to divide by 0. The `ZERO_DIVIDE` predefined exception is used to trap the error in an exception-handling routine.

```

Command> DECLARE v_invalid PLS_INTEGER;
          BEGIN
              v_invalid := 100/0;
          EXCEPTION
              WHEN ZERO_DIVIDE THEN
                  DBMS_OUTPUT.PUT_LINE ('Attempt to divide by 0');
          END;
          /

```

Attempt to divide by 0

PL/SQL procedure successfully completed.

**Trapping user-defined exceptions**

You can define your own exceptions in PL/SQL in TimesTen, and you can raise user-defined exceptions explicitly with either the PL/SQL `RAISE` statement or the `RAISE_APPLICATION_ERROR` procedure.

**Using the RAISE statement**

The `RAISE` statement stops normal execution of a PL/SQL block or subprogram and transfers control to an exception handler. `RAISE` statements can raise predefined exceptions, or user-defined exceptions whose names you decide.

**Example 4–2 Using RAISE statement to trap user-defined exception**

In this example, the department number 500 does not exist, so no rows are updated in the `departments` table. The `RAISE` statement is used to explicitly raise an exception and display an error message, returned by the `SQLERRM` built-in function, and an error code, returned by the `SQLCODE` built-in function. Use the `RAISE` statement by itself within an exception handler to raise the same exception again and propagate it back to the calling environment.

```

Command> DECLARE
        v_deptno NUMBER := 500;
        v_name VARCHAR2 (20) := 'Testing';
        e_invalid_dept EXCEPTION;
BEGIN
        UPDATE departments
        SET department_name = v_name
        WHERE department_id = v_deptno;
        IF SQL%NOTFOUND THEN
                RAISE e_invalid_dept;
        END IF;
ROLLBACK;
EXCEPTION
        WHEN e_invalid_dept THEN
                DBMS_OUTPUT.PUT_LINE ('No such department');
                DBMS_OUTPUT.PUT_LINE (SQLERRM);
                DBMS_OUTPUT.PUT_LINE (SQLCODE);
END;
/
No such department
User-Defined Exception
1

```

PL/SQL procedure successfully completed.

The command succeeded.

---



---

**Note:** Given the same error condition in TimesTen and Oracle Database, SQLCODE returns the same error code, but SQLERRM does not necessarily return the same error message. This is also noted in ["TimesTen error messages and SQL codes"](#) on page 4-9.

---



---

### Using the RAISE\_APPLICATION\_ERROR procedure

Use the RAISE\_APPLICATION\_ERROR procedure in the executable section or exception section (or both) of your PL/SQL program. TimesTen reports errors to your application so you can avoid returning unhandled exceptions.

Use an error number between -20,000 and -20,999. Specify a character string up to 2,048 bytes for your message.

#### **Example 4-3 Using the RAISE\_APPLICATION\_ERROR procedure**

This example attempts to delete from the employees table where last\_name=Patterson. The RAISE\_APPLICATION\_ERROR procedure raises the error, using error number -20201.

```

Command> DECLARE
        v_last_name employees.last_name%TYPE := 'Patterson';
BEGIN
        DELETE FROM employees WHERE last_name = v_last_name;
        IF SQL%NOTFOUND THEN
                RAISE_APPLICATION_ERROR (-20201, v_last_name || ' does not exist');
        END IF;
END;
/
8507: ORA-20201: Patterson does not exist
8507: ORA-06512: at line 6
The command failed.

```

## Retrying after transient errors (PL/SQL)

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

- ORA-57005: Transient transaction failure due to unavailability of resource. Roll back the transaction and try it again.

---



---

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

---



---

Using the PL/SQL pragma `EXCEPTION_INIT`, declare an exception to correspond to this error, and retry the transaction when the exception is encountered.

Here is an example:

```
declare
  retry_stmt exception;
  pragma exception_init(retry_stmt, -57007);
  retry_txn exception;
  pragma exception_init(retry_txn, -57005);
begin
  -- Execute SQL
exception
when retry_stmt then
  -- Re-execute the failing SQL statement
when retry_txn then
  -- Re-execute the failing transaction
end;
```

## Showing errors in ttlsql

You can use the `show errors` command in `ttIsql` to see details about errors you encounter in executing anonymous blocks or compiling packages, procedures, or functions. This is shown in [Example 4-4](#).

### **Example 4-4** *ttlsql show errors command*

Again consider [Example 2-13](#) on page 2-14. Assume the same package specification shown there, which declares the procedures and functions `hire_employee`, `remove_employee`, and `num_above_salary`. But instead of the body definition shown there, consider the following, which defines `hire_employee` and `num_above_salary` but not `remove_employee`:

```
CREATE OR REPLACE PACKAGE BODY emp_actions AS
-- Code for procedure hire_employee:
  PROCEDURE hire_employee (employee_id NUMBER,
    last_name VARCHAR2,
    first_name VARCHAR2,
    email VARCHAR2,
    phone_number VARCHAR2,
    hire_date DATE,
    job_id VARCHAR2,
    salary NUMBER,
    commission_pct NUMBER,
    manager_id NUMBER,
    department_id NUMBER) IS
  BEGIN
```

```

INSERT INTO employees VALUES (employee_id,
    last_name,
    first_name,
    email,
    phone_number,
    hire_date,
    job_id,
    salary,
    commission_pct,
    manager_id,
    department_id);
END hire_employee;
-- Code for function num_above_salary:
FUNCTION num_above_salary (emp_id NUMBER) RETURN NUMBER IS
    emp_sal NUMBER(8,2);
    num_count NUMBER;
BEGIN
    SELECT salary INTO emp_sal FROM employees
    WHERE employee_id = emp_id;
    SELECT COUNT(*) INTO num_count FROM employees
    WHERE salary > emp_sal;
    RETURN num_count;
END num_above_salary;
END emp_actions;
/

```

Attempting this body definition after the original package specification results in the following:

Warning: Package body created with compilation errors.

To get more information, run `ttIsql` and use the command `show errors`. In this example, `show errors` provides the following:

```

Command> show errors;
Errors for PACKAGE BODY EMP_ACTIONS:

LINE/COL ERROR
-----
13/13      PLS-00323: subprogram or cursor 'REMOVE_EMPLOYEE' is declared in a
package specification and must be defined in the package body

```

## Differences in TimesTen: exception handling and error behavior

You should be aware of some error-related behaviors that differ between TimesTen PL/SQL and Oracle Database PL/SQL:

- [TimesTen PL/SQL transaction and rollback behavior for unhandled exceptions](#)
- [TimesTen error messages and SQL codes](#)
- [Warnings not visible in PL/SQL](#)
- [Unsupported predefined errors](#)
- [Possibility of runtime errors after clean compile \(use of Oracle Database SQL parser\)](#)
- [Use of TimesTen expressions at runtime](#)

## TimesTen PL/SQL transaction and rollback behavior for unhandled exceptions

TimesTen PL/SQL differs from Oracle Database PL/SQL in a scenario where an application executes PL/SQL in the middle of a transaction, and an unhandled exception occurs during execution of the PL/SQL. Oracle Database rolls back to the beginning of the anonymous block. TimesTen does not roll back.

An application should always handle any exception that results from execution of a PL/SQL block, as in the following example, run with autocommit disabled:

```
create table mytable (num int not null primary key);
set serveroutput on

insert into mytable values(1);
begin
  insert into mytable values(2);
  insert into mytable values(1);
exception
  when dup_val_on_index then
    dbms_output.put_line('oops:' || sqlerrm);
    rollback;
end;

select * from mytable;

commit;
```

The second INSERT fails because values must be unique, so an exception occurs and the program performs a rollback. Running this in TimesTen results in this:

```
oops:TT0907: Unique constraint (MYTABLE) violated at Rowid <BMUFVUAAABQAAAADjq>

select * from mytable;
0 rows found.
```

The result is equivalent in Oracle Database, with the SELECT results showing no rows.

Now consider a TimesTen example where the exception is not handled, again run with autocommit disabled:

```
create table mytable (num int not null primary key);
set serveroutput on

insert into mytable values(1);
begin
  insert into mytable values(2);
  insert into mytable values(1);
end;

select * from mytable;

commit;
```

In TimesTen, the SELECT query indicates execution of the first two inserts:

```
907: Unique constraint (MYTABLE) violated at Rowid <BMUFVUAAABQAAAADjq>
8507: ORA-06512: at line 3
The command failed.

select * from mytable;
< 1 >
< 2 >
```

2 rows found.

If you execute this in Oracle Database, there is a rollback to the beginning of the PL/SQL block, so the results of the `SELECT` indicate execution of only the first insert:

```
ORA-00001: unique constraint (SYSTEM.SYS_C004423) violated
ORA-06512: at line 3
```

```
NUM
-----
1
```

---

---

#### Notes:

- If there is an unhandled exception in a PL/SQL block, TimesTen leaves the transaction open only to allow the application to assess its state and determine appropriate action.
  - An application in TimesTen should not execute a PL/SQL block while there are uncommitted changes in the current transaction, unless those changes together with the PL/SQL operations really do constitute a single logical unit of work and the application can determine appropriate action. Such action, for example, might consist of a rollback to the beginning of the transaction.
  - If autocommit is enabled and an unhandled exception occurs in TimesTen, the entire transaction is rolled back.
- 
- 

## TimesTen error messages and SQL codes

Given the same error condition, TimesTen does not guarantee that the error message returned by TimesTen is the same as the message returned by Oracle Database, although the SQL code is the same. Therefore, the information returned by the `SQLERRM` function may be different, but that returned by the `SQLCODE` function is the same.

For further information:

- [Example 4-2](#) on page 4-4 uses `SQLERRM` and `SQLCODE`.
- Refer to "Warnings and Errors" in *Oracle TimesTen In-Memory Database Error Messages and SNMP Traps* for information about specific TimesTen error messages.
- Refer to "SQLERRM Function" and "SQLCODE Function" in *Oracle Database PL/SQL Language Reference* for general information.

## Warnings not visible in PL/SQL

Oracle Database does not have the concept of runtime warnings, so Oracle Database PL/SQL does not support warnings.

TimesTen does have the concept of warnings, but because the TimesTen PL/SQL implementation is based on the Oracle Database PL/SQL implementation, TimesTen PL/SQL does not support warnings.

As a result, in TimesTen you could execute a SQL statement and see a resulting warning, but if you execute the same statement through PL/SQL you would not see the warning.

## Unsupported predefined errors

"[Trapping predefined TimesTen errors](#)" on page 4-3 lists predefined exceptions supported by TimesTen, the associated ORA error numbers and SQLCODE values, and descriptions of the exceptions.

[Table 4-2](#) notes predefined exceptions that are *not* supported by TimesTen.

**Table 4-2 Predefined exceptions not supported by TimesTen**

Exception name	Oracle Database error number	SQLCODE	Description
LOGIN_DENIED	ORA-01017	-1017	User name or password is invalid.
NOT_LOGGED_ON	ORA-01012	-1012	Program issued a database call without being connected to the database.
SELF_IS_NULL	ORA-30625	-30625	Program attempted to invoke a MEMBER method, but the object was not initialized.
TIMEOUT_ON_RESOURCE	ORA-00051	-51	Timeout occurred while the database was waiting for a resource.

## Possibility of runtime errors after clean compile (use of Oracle Database SQL parser)

The TimesTen PL/SQL implementation uses the Oracle Database SQL parser in compiling PL/SQL programs. (This is discussed in "[PL/SQL in TimesTen versus PL/SQL in Oracle Database](#)" on page 1-3.) As a result, if your program uses Oracle Database syntax or built-in procedures that are not supported by TimesTen, the issue is not discovered during compilation. A runtime error occurs during program execution, however.

## Use of TimesTen expressions at runtime

TimesTen SQL includes several constructs that are not present in Oracle Database SQL. The PL/SQL language does not include these constructs. To use TimesTen-specific SQL from PL/SQL, execute the SQL statements using the `EXECUTE IMMEDIATE` statement. This avoids compilation errors.

For lists of TimesTen-specific SQL and expressions, see "Compatibility Between TimesTen and Oracle Databases" in *Oracle TimesTen Application-Tier Database Cache User's Guide*.

For more information about `EXECUTE IMMEDIATE`, refer to "[Dynamic SQL in PL/SQL \(EXECUTE IMMEDIATE statement\)](#)" on page 2-20.

---

---

## Examples Using TimesTen SQL in PL/SQL

This chapter provides additional examples to further explore the tight integration of TimesTen SQL in PL/SQL:

- [Examples using the SELECT...INTO statement in PL/SQL](#)
- [Example using the INSERT statement](#)
- [Examples using input and output parameters and bind variables](#)
- [Examples using cursors](#)
- [Examples using FORALL and BULK COLLECT](#)
- [Examples using EXECUTE IMMEDIATE](#)
- [Examples using RETURNING INTO](#)
- [Example querying a system view](#)

### Examples using the SELECT...INTO statement in PL/SQL

Use the `SELECT . . . INTO` statement to retrieve exactly one row of data. TimesTen returns an error for any query that returns no rows or multiple rows.

The section provides the following examples:

- [Using SELECT... INTO to return sum of salaries](#)
- [Using SELECT...INTO to query another user's table](#)

### Using SELECT... INTO to return sum of salaries

This example uses a `SELECT . . . INTO` statement to calculate the sum of salaries for all employees in the department where `department_id` is 60.

#### **Example 5-1 Using SELECT... INTO to return sum of salaries**

```
Command> DECLARE
          v_sum_sal  NUMBER (10,2);
          v_dept_no  NUMBER NOT NULL := 60;
BEGIN
  SELECT SUM(salary) -- aggregate function
  INTO v_sum_sal FROM employees
  WHERE department_id = v_dept_no;
  DBMS_OUTPUT.PUT_LINE ('Sum is ' || v_sum_sal);
END;
/
Sum is 28800
```

PL/SQL procedure successfully completed.

## Using SELECT...INTO to query another user's table

This example provides two users, USER1 and USER2, to show one user employing SELECT...INTO to query another user's table.

### **Example 5-2 Using SELECT...INTO to query another user's table**

The following privileges are assumed:

```
grant create session to user1;
grant create session to user2;
grant create table to user1;
grant select on user1.test to user2;
```

#### **USER1:**

```
Command> create table test(name varchar2(20), id number);
Command> insert into test values('posey', 363);
1 row inserted.
```

#### **USER2:**

```
Command> declare
    targetid number;
begin
    select id into targetid from user1.test where name='posey';
    dbms_output.put_line('Target ID is ' || targetid);
end;
/
Target ID is 363
```

PL/SQL procedure successfully completed.

## Example using the INSERT statement

TimesTen supports the TimesTen DML statements INSERT, UPDATE, DELETE, and MERGE. This section has an example of the INSERT statement.

### **Example 5-3 Using the INSERT statement in PL/SQL**

This example uses the AS SELECT query clause to create table emp\_copy, sets AUTOCOMMIT off, creates a sequence to increment employee\_id, and uses the INSERT statement in PL/SQL to insert a row of data in table emp\_copy.

```
Command> CREATE TABLE emp_copy AS SELECT * FROM employees;
107 rows inserted.
Command> SET AUTOCOMMIT OFF;

Command> CREATE SEQUENCE emp_copy_seq
START WITH 207
INCREMENT BY 1;

Command> BEGIN
    INSERT INTO emp_copy
        (employee_id, first_name, last_name, email, hire_date, job_id,
         salary)
    VALUES (emp_copy_seq.NEXTVAL, 'Parker', 'Cores', 'PCORES', SYSDATE,
            'AD_ASST', 4000);
```

```
END;
/
```

PL/SQL procedure successfully completed.

Continuing, the example confirms the row was inserted, then rolls back the transaction.

```
Command> SELECT * FROM EMP_COPY WHERE first_name = 'Parker';
< 207, Parker, Cores, PCORES, <NULL>, 2008-07-19 21:49:55, AD_ASST, 4000,
<NULL>, <NULL>, <NULL> >
1 row found.
Command> ROLLBACK;
Command> SELECT * FROM emp_copy WHERE first_name = 'Parker';
0 rows found.
```

Now INSERT is executed again, then the transaction is rolled back in PL/SQL. Finally, the example verifies that TimesTen did not insert the row.

```
Command> BEGIN
      INSERT INTO emp_copy
      (employee_id, first_name, last_name, email, hire_date, job_id,
       salary)
      VALUES (emp_copy_seq.NEXTVAL, 'Parker', 'Cores', 'PCORES', SYSDATE,
              'AD_ASST',4000);
      ROLLBACK;
      END;
/
```

PL/SQL procedure successfully completed.

```
Command> SELECT * FROM emp_copy WHERE first_name = 'Parker';
0 rows found.
```

## Examples using input and output parameters and bind variables

The following examples in this section use IN, OUT, and IN OUT parameters, including bind variables (host variables) from outside PL/SQL:

- [Using IN and OUT parameters](#)
- [Using IN OUT parameters](#)
- [Using associative arrays](#)

### Using IN and OUT parameters

This example creates a procedure `query_emp` to retrieve information about an employee, passes the `employee_id` value 171 to the procedure, and retrieves the name and salary into two OUT parameters.

#### **Example 5–4 Using IN and OUT parameters**

```
Command> CREATE OR REPLACE PROCEDURE query_emp
      (p_id IN employees.employee_id%TYPE,
       p_name OUT employees.last_name%TYPE,
       p_salary OUT employees.salary%TYPE) IS
      BEGIN
      SELECT last_name, salary INTO p_name, p_salary
      FROM employees
      WHERE employee_id = p_id;
```

```

        END query_emp;
    /

Procedure created.

Command> -- Execute the procedure
DECLARE
    v_emp_name employees.last_name%TYPE;
    v_emp_sal   employees.salary%TYPE;
BEGIN
    query_emp (171, v_emp_name, v_emp_sal);
    DBMS_OUTPUT.PUT_LINE (v_emp_name || ' earns ' ||
        TO_CHAR (v_emp_sal, '$999,999.00'));
END;
/
Smith earns      $7,400.00

PL/SQL procedure successfully completed.

```

## Using IN OUT parameters

Consider a situation where you want to format a phone number. This example takes a 10-character string containing digits for a phone number and passes this unformatted string to a procedure as an IN OUT parameter. After the procedure is executed, the IN OUT parameter contains the formatted phone number value.

### **Example 5-5 Using IN OUT parameters**

```

Command> CREATE OR REPLACE PROCEDURE format_phone
    (p_phone_no IN OUT VARCHAR2 ) IS
BEGIN
    p_phone_no := '(' || SUBSTR (p_phone_no,1,3) ||
        ') ' || SUBSTR (p_phone_no,4,3) ||
        '-' || SUBSTR (p_phone_no,7);
END format_phone;
/

```

Procedure created.

Create the bind variable, execute the procedure, and verify the results.

```

Command> VARIABLE b_phone_no VARCHAR2 (15);
Command> EXECUTE :b_phone_no := '8006330575';

```

PL/SQL procedure successfully completed.

```

Command> PRINT b_phone_no;
B_PHONE_NO          : 8006330575
Command> BEGIN
    format_phone (:b_phone_no);
END;
/

```

PL/SQL procedure successfully completed.

```

Command> PRINT b_phone_no
B_PHONE_NO          : (800) 633-0575

```

## Using associative arrays

This example uses `ttIsql` to bind a `NUMBER` array and a `VARCHAR2` array to corresponding `OUT` associative arrays in a PL/SQL procedure.

See "[Using associative arrays from applications](#)" on page 3-6 for related information.

### **Example 5-6 Binding to an associative array from `ttIsql`**

Assume the following SQL setup.

```
DROP TABLE FOO;

CREATE TABLE FOO (CNUM INTEGER,
                  CVC2 VARCHAR2(20));

INSERT INTO FOO VALUES ( null,
                        'VARCHAR 1');
INSERT INTO FOO VALUES (-102,
                        null);
INSERT INTO FOO VALUES ( 103,
                        'VARCHAR 3');
INSERT INTO FOO VALUES (-104,
                        'VARCHAR 4');
INSERT INTO FOO VALUES ( 105,
                        'VARCHAR 5');
INSERT INTO FOO VALUES ( 106,
                        'VARCHAR 6');
INSERT INTO FOO VALUES ( 107,
                        'VARCHAR 7');
INSERT INTO FOO VALUES ( 108,
                        'VARCHAR 8');

COMMIT;
```

Assume the following PL/SQL package definition. This includes the `INTEGER` associative array type `NUMARRTYP` and the `VARCHAR2` associative array type `VCHARRTYP`, used for output associative arrays `c1` and `c2`, respectively, in the definition of procedure `P1`.

```
CREATE OR REPLACE PACKAGE PKG1 AS
  TYPE NUMARRTYP IS TABLE OF INTEGER INDEX BY BINARY_INTEGER;
  TYPE VCHARRTYP IS TABLE OF VARCHAR2(20) INDEX BY BINARY_INTEGER;

  PROCEDURE P1(c1 OUT NUMARRTYP,c2 OUT VCHARRTYP);

END PKG1;
/

CREATE OR REPLACE PACKAGE BODY PKG1 AS

  CURSOR CUR1 IS SELECT CNUM, CVC2 FROM FOO;

  PROCEDURE P1(c1 OUT NUMARRTYP,c2 OUT VCHARRTYP) IS
  BEGIN
    IF NOT CUR1%ISOPEN THEN
      OPEN CUR1;
    END IF;
    FOR i IN 1..8 LOOP
      FETCH CUR1 INTO c1(i), c2(i);
      IF CUR1%NOTFOUND THEN
```

```
        CLOSE CUR1;
        EXIT;
    END IF;
END LOOP;
END P1;

END PKG1;
```

Now `ttIsql` calls `PKG1.P1`, binds arrays to the `P1` output associative arrays, and prints the contents of those associative arrays.

```
Command> var c1[10] number;
Command> var c2[10] varchar2(20);
Command> print;
C1                : ARRAY [ 10 ] (Current Size 0)
C2                : ARRAY [ 10 ] (Current Size 0)
Command> BEGIN PKG1.P1(:c1, :c2); END; /
```

PL/SQL procedure successfully completed.

```
Command> print
C1                : ARRAY [ 10 ] (Current Size 8)
C1[1] : <NULL>
C1[2] : -102
C1[3] : 103
C1[4] : -104
C1[5] : 105
C1[6] : 106
C1[7] : 107
C1[8] : 108
C2                : ARRAY [ 10 ] (Current Size 8)
C2[1] : VARCHAR  1
C2[2] : <NULL>
C2[3] : VARCHAR  3
C2[4] : VARCHAR  4
C2[5] : VARCHAR  5
C2[6] : VARCHAR  6
C2[7] : VARCHAR  7
C2[8] : VARCHAR  8
```

## Examples using cursors

TimesTen supports cursors, as discussed in ["Use of cursors in PL/SQL programs"](#) on page 2-28. Use a cursor to handle the result set of a `SELECT` statement.

Examples in this section cover the following:

- [Fetching values](#)
- [Using the %ROWCOUNT and %NOTFOUND attributes](#)
- [Using cursor FOR loops](#)

See "Explicit Cursor Attributes" in *Oracle Database PL/SQL Language Reference* for information about the cursor attributes used in these examples.

### Fetching values

This section provides examples of how to fetch values from a cursor, including how to fetch the values into a record.

**Example 5-7 Fetching values from a cursor**

The following example uses a cursor to select `employee_id` and `last_name` from the `employees` table where `department_id` is 30. Two variables are declared to hold the fetched values from the cursor, and the `FETCH` statement retrieves rows one at a time in a loop to retrieve all rows. Execution stops when there are no remaining rows in the cursor, illustrating use of the `%NOTFOUND` cursor attribute.

`%NOTFOUND` yields `TRUE` if an `INSERT`, `UPDATE`, or `DELETE` statement affected no rows, or a `SELECT INTO` statement returned no rows.

```
Command> DECLARE
          CURSOR c_emp_cursor IS
              SELECT employee_id, last_name FROM employees
              WHERE department_id = 30;
          v_empno employees.employee_id%TYPE;
          v_lname employees.last_name%TYPE;
BEGIN
  OPEN c_emp_cursor;
  LOOP
    FETCH c_emp_cursor INTO v_empno, v_lname;
    EXIT WHEN c_emp_cursor%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE (v_empno || ' ' || v_lname);
  END LOOP;
  CLOSE c_emp_cursor;
END;
/
```

```
114 Raphaely
115 Khoo
116 Baida
117 Tobias
118 Himuro
119 Colmenares
```

**Example 5-8 Fetching values into a record**

This is similar to [Example 5-7](#) above, with the same results, but fetches the values into a PL/SQL record instead of PL/SQL variables.

```
Command> DECLARE
          CURSOR c_emp_cursor IS
              SELECT employee_id, last_name FROM employees
              WHERE department_id = 30;
          v_emp_record c_emp_cursor%ROWTYPE;
BEGIN
  OPEN c_emp_cursor;
  LOOP
    FETCH c_emp_cursor INTO v_emp_record;
    EXIT WHEN c_emp_cursor%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE (v_emp_record.employee_id || ' ' |
                          v_emp_record.last_name);
  END LOOP;
  CLOSE c_emp_cursor;
END;
/
```

```
114 Raphaely
115 Khoo
116 Baida
117 Tobias
118 Himuro
```

119 Colmenares

PL/SQL procedure successfully completed.

## Using the %ROWCOUNT and %NOTFOUND attributes

[Example 5-9](#) shows how to use the %ROWCOUNT cursor attribute as well as the %NOTFOUND cursor attribute shown in [Example 5-7](#) and [Example 5-8](#).

### **Example 5-9 Using %ROWCOUNT and %NOTFOUND attributes**

This example has the same results as [Example 5-8](#), but illustrating the %ROWCOUNT cursor attribute as well as the %NOTFOUND attribute for exit conditions in the loop.

%ROWCOUNT yields the number of rows affected by an INSERT, UPDATE, or DELETE statement or returned by a SELECT...INTO or FETCH...INTO statement.

```
Command> DECLARE
        CURSOR c_emp_cursor IS
            SELECT employee_id, last_name FROM employees
            WHERE department_id = 30;
        v_emp_record c_emp_cursor%ROWTYPE;
BEGIN
OPEN c_emp_cursor;
LOOP
    FETCH c_emp_cursor INTO v_emp_record;
    EXIT WHEN c_emp_cursor%ROWCOUNT > 10 OR c_emp_cursor%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE (v_emp_record.employee_id || ' ' ||
        v_emp_record.last_name);
    END LOOP;
    CLOSE c_emp_cursor;
END;
/
114 Raphaely
115 Khoo
116 Baida
117 Tobias
118 Himuro
119 Colmenares
```

PL/SQL procedure successfully completed.

## Using cursor FOR loops

PL/SQL in TimesTen supports cursor FOR loops, as shown in the following examples.

### **Example 5-10 Using a cursor FOR loop**

In this example, PL/SQL implicitly declares emp\_record. No OPEN and CLOSE statements are necessary. The results are the same as in [Example 5-9](#) above.

```
Command> DECLARE
        CURSOR c_emp_cursor IS
            SELECT employee_id, last_name FROM employees
            WHERE department_id = 30;
BEGIN
    FOR emp_record IN c_emp_cursor
    LOOP
        DBMS_OUTPUT.PUT_LINE (emp_record.employee_id || ' ' ||
            emp_record.last_name);
    END LOOP;
```

```

        END;
    /
114 Raphaely
115 Khoo
116 Baida
117 Tobias
118 Himuro
119 Colmenares

PL/SQL procedure successfully completed.

```

### **Example 5–11 Using a cursor FOR loop with subqueries**

This example illustrates a FOR loop using subqueries. The results are the same as in [Example 5–9](#) on page 5-8 and [Example 5–10](#) above.

```

Command> BEGIN
        FOR emp_record IN (SELECT employee_id, last_name FROM
                           employees WHERE department_id = 30)
        LOOP
            DBMS_OUTPUT.PUT_LINE (emp_record.employee_id || ' ' ||
                                   emp_record.last_name);
        END LOOP;
    END;
    /
114 Raphaely
115 Khoo
116 Baida
117 Tobias
118 Himuro
119 Colmenares

PL/SQL procedure successfully completed.

```

## **Examples using FORALL and BULK COLLECT**

TimesTen supports bulk binding and the FORALL statement and BULK COLLECT feature, as noted in "[FORALL and BULK COLLECT operations](#)" on page 2-22.

Examples in this section cover the following:

- [Using FORALL with SQL%BULK\\_ROWCOUNT](#)
- [Using BULK COLLECT INTO with queries](#)
- [Using BULK COLLECT INTO with cursors](#)
- [Using SAVE EXCEPTIONS with BULK COLLECT](#)

### **Using FORALL with SQL%BULK\_ROWCOUNT**

The %BULK\_ROWCOUNT cursor attribute is a composite structure designed for use with the FORALL statement.

The attribute acts like an associative array (index-by table). Its *i*th element stores the number of rows processed by the *i*th execution of the INSERT statement. If the *i*th execution affects no rows, then %BULK\_ROWCOUNT(*i*) returns zero.

This is demonstrated in [Example 5–12](#).

**Example 5–12 Using the FORALL statement with SQL%BULK\_ROWCOUNT**

```

Command> DECLARE
        TYPE num_list_type IS TABLE OF NUMBER
            INDEX BY BINARY_INTEGER;
        v_nums num_list_type;
BEGIN
        v_nums (1) := 1;
        v_nums (2) := 3;
        v_nums (3) := 5;
        v_nums (4) := 7;
        v_nums (5) := 11;
        FORALL i IN v_nums.FIRST .. v_nums.LAST
            INSERT INTO num_table (n) VALUES (v_nums (i));
        FOR i IN v_nums.FIRST .. v_nums.LAST
        LOOP
            DBMS_OUTPUT.PUT_LINE ('Inserted ' ||
                SQL%BULK_ROWCOUNT (i) || ' row (s)' ||
                ' on iteration ' || i);
        END LOOP;
    END;
/
Inserted 1 row (s) on iteration 1
Inserted 1 row (s) on iteration 2
Inserted 1 row (s) on iteration 3
Inserted 1 row (s) on iteration 4
Inserted 1 row (s) on iteration 5

```

PL/SQL procedure successfully completed.

**Using BULK COLLECT INTO with queries**

Use BULK COLLECT with the SELECT statement in PL/SQL to retrieve rows without using a cursor.

**Example 5–13 Using BULK COLLECT INTO with queries**

This example selects all rows from the departments table for a specified location into a nested table, then uses a FOR LOOP to output data.

```

Command> CREATE OR REPLACE PROCEDURE get_departments (p_loc NUMBER) IS
        TYPE dept_tab_type IS
        TABLE OF departments%ROWTYPE;
        v_depts dept_tab_type;
BEGIN
        SELECT * BULK COLLECT INTO v_depts
        FROM departments
        where location_id = p_loc;
        FOR i IN 1 .. v_depts.COUNT
        LOOP
            DBMS_OUTPUT.PUT_LINE (v_depts(i).department_id
                || ' ' || v_depts (i).department_name);
        END LOOP;
    END;
/

```

Procedure created.

The following executes the procedure and verifies the results:

```

Command> EXECUTE GET_DEPARTMENTS (1700);

```

```

10 Administration
30 Purchasing
90 Executive
100 Finance
110 Accounting
120 Treasury
130 Corporate Tax
140 Control And Credit
150 Shareholder Services
160 Benefits
170 Manufacturing
180 Construction
190 Contracting
200 Operations
210 IT Support
220 NOC
230 IT Helpdesk
240 Government Sales
250 Retail Sales
260 Recruiting
270 Payroll

```

PL/SQL procedure successfully completed.

```

Command> SELECT department_id, department_name FROM departments WHERE
           location_id = 1700;
< 10, Administration >
< 30, Purchasing >
< 90, Executive >
< 100, Finance >
< 110, Accounting >
< 120, Treasury >
< 130, Corporate Tax >
< 140, Control And Credit >
< 150, Shareholder Services >
< 160, Benefits >
< 170, Manufacturing >
< 180, Construction >
< 190, Contracting >
< 200, Operations >
< 210, IT Support >
< 220, NOC >
< 230, IT Helpdesk >
< 240, Government Sales >
< 250, Retail Sales >
< 260, Recruiting >
< 270, Payroll >
21 rows found.

```

## Using BULK COLLECT INTO with cursors

[Example 5-14](#) uses a cursor to bulk-collect rows from a table.

### **Example 5-14 Using BULK COLLECT INTO with cursors**

This example uses a cursor to bulk-collect rows from the departments table with a specified location\_id. value. Results are the same as in [Example 5-13](#) above.

```

Command> CREATE OR REPLACE PROCEDURE get_departments2 (p_loc NUMBER) IS
           CURSOR cur_dept IS

```

```

        SELECT * FROM departments
        WHERE location_id = p_loc;
    TYPE dept_tab_type IS TABLE OF cur_dept%ROWTYPE;
    v_depts dept_tab_type;
BEGIN
    OPEN cur_dept;
    FETCH cur_dept BULK COLLECT INTO v_depts;
    CLOSE cur_dept;
    FOR i IN 1 .. v_depts.COUNT
    LOOP
        DBMS_OUTPUT.PUT_LINE (v_depts (i).department_id
        || ' ' || v_depts (i).department_name );
    END LOOP;
END;
/

```

Procedure created.

Command> EXECUTE GET\_DEPARTMENTS2 (1700);

```

10 Administration
30 Purchasing
90 Executive
100 Finance
110 Accounting
120 Treasury
130 Corporate Tax
140 Control And Credit
150 Shareholder Services
160 Benefits
170 Manufacturing
180 Construction
190 Contracting
200 Operations
210 IT Support
220 NOC
230 IT Helpdesk
240 Government Sales
250 Retail Sales
260 Recruiting
270 Payroll

```

PL/SQL procedure successfully completed.

## Using SAVE EXCEPTIONS with BULK COLLECT

SAVE EXCEPTIONS allows an UPDATE, INSERT, or DELETE statement to continue executing after it issues an exception. When the statement finishes, an error is issued to signal that at least one exception occurred. Exceptions are collected into an array that you can examine using %BULK\_EXCEPTIONS after the statement has executed.

### **Example 5–15 Using SAVE EXCEPTIONS with BULK COLLECT**

In this example, PL/SQL raises predefined exceptions because some new values are too large for the job\_id column. After the FORALL statement, SQL%BULK\_EXCEPTIONS.COUNT returns 2, and the contents of SQL%BULK\_EXCEPTIONS are (7, 01401) and (13, 01401), indicating the error number and the line numbers where the error was detected. To get the error message, the negative of SQL%BULK\_EXCEPTIONS(i).ERROR\_CODE is passed to the error-reporting function SQLERRM (which expects a negative number).

The following script is executed using ttIsql:

```
-- create a temporary table for this example
CREATE TABLE emp_temp AS SELECT * FROM employees;

DECLARE
    TYPE empid_tab IS TABLE OF employees.employee_id%TYPE;
    emp_sr empid_tab;
-- create an exception handler for ORA-24381
errors NUMBER;
dml_errors EXCEPTION;
PRAGMA EXCEPTION_INIT(dml_errors, -24381);

BEGIN
    SELECT employee_id
        BULK COLLECT INTO emp_sr FROM emp_temp
        WHERE hire_date < '1994-12-30';
-- add '_SR' to the job_id of the most senior employees
FORALL i IN emp_sr.FIRST..emp_sr.LAST SAVE EXCEPTIONS
    UPDATE emp_temp SET job_id = job_id || '_SR'
        WHERE emp_sr(i) = emp_temp.employee_id;
-- If any errors occurred during the FORALL SAVE EXCEPTIONS,
-- a single exception is raised when the statement completes.

EXCEPTION
-- Figure out what failed and why
WHEN dml_errors THEN
    errors := SQL%BULK_EXCEPTIONS.COUNT;
    DBMS_OUTPUT.PUT_LINE
        ('Number of statements that failed: ' || errors);
    FOR i IN 1..errors LOOP
        DBMS_OUTPUT.PUT_LINE('Error #' || i || ' occurred during ' ||
            'iteration #' || SQL%BULK_EXCEPTIONS(i).ERROR_INDEX);
        DBMS_OUTPUT.PUT_LINE('Error message is ' ||
            SQLERRM(-SQL%BULK_EXCEPTIONS(i).ERROR_CODE));
    END LOOP;
END;
/

DROP TABLE emp_temp;
```

Results are as follows:

```
Number of statements that failed: 2
Error #1 occurred during iteration #7
Error message is ORA-01401: inserted value too large for column
Error #2 occurred during iteration #13
Error message is ORA-01401: inserted value too large for column

PL/SQL procedure successfully completed.
```

## Examples using EXECUTE IMMEDIATE

TimesTen supports the EXECUTE IMMEDIATE statement, as noted in "[Dynamic SQL in PL/SQL \(EXECUTE IMMEDIATE statement\)](#)" on page 2-20. This section provides these additional examples to consider as you develop your PL/SQL applications in TimesTen:

- [Using EXECUTE IMMEDIATE to create a table](#)
- [Using EXECUTE IMMEDIATE with a single row query](#)

- [Using EXECUTE IMMEDIATE to alter a connection attribute](#)
- [Using EXECUTE IMMEDIATE to call a TimesTen built-in procedure](#)
- [Using EXECUTE IMMEDIATE with TimesTen-specific syntax](#)

## Using EXECUTE IMMEDIATE to create a table

Consider a situation where you do not know your table definition at compilation. By using an EXECUTE IMMEDIATE statement, you can create your table at execution time. This example shows a procedure that creates a table using the EXECUTE IMMEDIATE statement. The procedure is executed with the table name and column definitions passed as parameters, then creation of the table is verified.

### **Example 5–16 Using EXECUTE IMMEDIATE to create a table**

```
Command> CREATE OR REPLACE PROCEDURE create_table
      (p_table_name VARCHAR2, p_col_specs VARCHAR2) IS
BEGIN
      EXECUTE IMMEDIATE 'CREATE TABLE ' || p_table_name
      || ' (' || p_col_specs || ')';
END;
/
```

Procedure created.

Execute the procedure and verify the table is created.

```
Command> BEGIN
      create_table ('EMPLOYEES_NAMES', 'id NUMBER (4)
      PRIMARY KEY, name VARCHAR2 (40)');
END;
/
```

PL/SQL procedure successfully completed.

```
Command> DESCRIBE employees_names;
```

```
Table USER.EMPLOYEES_NAMES:
Columns:
  *ID                NUMBER (4) NOT NULL
  NAME               VARCHAR2 (40) INLINE
```

```
1 table found.
(primary key columns are indicated with *)
```

## Using EXECUTE IMMEDIATE with a single row query

In this example, the function get\_emp retrieves an employee record. The function is executed and returns the results in v\_emprec.

### **Example 5–17 Using EXECUTE IMMEDIATE with a single row query**

```
Command> CREATE OR REPLACE FUNCTION get_emp (p_emp_id NUMBER)
      RETURN employees%ROWTYPE IS
      v_stmt VARCHAR2 (200);
      v_emprec employees%ROWTYPE;
BEGIN
      v_stmt:= 'SELECT * FROM EMPLOYEES ' ||
      'WHERE employee_id = :p_emp_id';
      EXECUTE IMMEDIATE v_stmt INTO v_emprec USING p_emp_id;
```

```

        RETURN v_emprec;
    END;
/

```

Function created.

```

Command> DECLARE
    v_emprec employees%ROWTYPE := GET_EMP (100);
BEGIN
    DBMS_OUTPUT.PUT_LINE ('Employee: ' || v_emprec.last_name);
END;
/
Employee: King

```

PL/SQL procedure successfully completed.

## Using EXECUTE IMMEDIATE to alter a connection attribute

This example uses an EXECUTE IMMEDIATE statement with ALTER SESSION to alter the PLSQL\_OPTIMIZE\_LEVEL setting, calling the ttConfiguration built-in procedure before and after to verify the results. (The next example calls ttConfiguration from inside an EXECUTE IMMEDIATE statement.) Refer to "ttConfiguration" in *Oracle TimesTen In-Memory Database Reference* for information about this procedure.

### Example 5-18 Using EXECUTE IMMEDIATE to alter PLSCOPE\_SETTINGS

```

Command> call ttconfiguration;
...
< PLSCOPE_SETTINGS, IDENTIFIERS:NONE >
< PLSQL, 1 >
< PLSQL_CCFLAGS, <NULL> >
< PLSQL_CODE_TYPE, INTERPRETED >
< PLSQL_CONN_MEM_LIMIT, 100 >
< PLSQL_MEMORY_ADDRESS, 0x10000000 >
< PLSQL_MEMORY_SIZE, 128 >
< PLSQL_OPTIMIZE_LEVEL, 2 >
< PLSQL_TIMEOUT, 30 >
...
54 rows found.

Command> begin
    execute immediate 'alter session set PLSQL_OPTIMIZE_LEVEL=3';
end;
/
PL/SQL procedure successfully completed.

```

```

Command> call ttconfiguration;
...
< PLSCOPE_SETTINGS, IDENTIFIERS:NONE >
< PLSQL, 1 >
< PLSQL_CCFLAGS, <NULL> >
< PLSQL_CODE_TYPE, INTERPRETED >
< PLSQL_CONN_MEM_LIMIT, 100 >
< PLSQL_MEMORY_ADDRESS, 0x10000000 >
< PLSQL_MEMORY_SIZE, 128 >
< PLSQL_OPTIMIZE_LEVEL, 3 >
< PLSQL_TIMEOUT, 30 >
...
54 rows found.

```

## Using EXECUTE IMMEDIATE to call a TimesTen built-in procedure

In PL/SQL, you can use an `EXECUTE IMMEDIATE` statement with `CALL` syntax to call a TimesTen built-in procedure.

For example, to call the built-in procedure `ttConfiguration` and return its output result set, create a PL/SQL record type then use `EXECUTE IMMEDIATE` with `BULK COLLECT` to fetch the result set into an array.

For more information on TimesTen built-in procedures, see "Built-In Procedures" in *Oracle TimesTen In-Memory Database Reference*.

### Example 5-19 Using EXECUTE IMMEDIATE to call ttConfiguration

```
Command> DECLARE
    TYPE ttConfig_record IS RECORD
        (name varchar2(255), value varchar2 (255));
    TYPE ttConfig_table IS TABLE OF ttConfig_record;
    v_ttConfigs ttConfig_table;
BEGIN
    EXECUTE IMMEDIATE 'CALL ttConfiguration'
        BULK COLLECT into v_ttConfigs;
    DBMS_OUTPUT.PUT_LINE ('Name: ' || v_ttConfigs(7).name
        || ' Value: ' || v_ttConfigs(7).value);
END;
/
Name: CommitBufferSizeMax Value: 10

PL/SQL procedure successfully completed.
```

## Using EXECUTE IMMEDIATE with TimesTen-specific syntax

This example uses an `EXECUTE IMMEDIATE` statement to execute a TimesTen `SELECT FIRST n` statement. This syntax is specific to TimesTen.

### Example 5-20 Using EXECUTE IMMEDIATE with TimesTen-specific syntax

```
Command> DECLARE v_empid NUMBER;
BEGIN
    EXECUTE IMMEDIATE 'SELECT FIRST 1 employee_id FROM employees'
        INTO v_empid;
    DBMS_OUTPUT.PUT_LINE ('Employee id: ' || v_empid);
END;
/
Employee id: 100

PL/SQL procedure successfully completed.
```

## Examples using RETURNING INTO

This section includes the following two examples using the `RETURNING INTO` clause:

- [Using the RETURNING INTO clause with a record](#)
- [Using BULK COLLECT INTO with the RETURNING INTO clause](#)

See "[RETURNING INTO clause](#)" on page 2-23 for an overview.

## Using the RETURNING INTO clause with a record

The following example uses `ttIsql` to run a SQL script that uses a `RETURNING INTO` clause to return data into a record. The example gives a raise to a specified employee, returns his name and new salary into a record, then outputs the data from the record. For reference, the original salary is shown before running the script.

### **Example 5–21 Using the RETURNING INTO clause with a record**

```
Command> SELECT SALARY, LAST_NAME FROM EMPLOYEES WHERE EMPLOYEE_ID = 100;
< 24000, King >
1 row found.
```

```
Command> run ReturnIntoWithRecord.sql;
```

```
CREATE TABLE emp_temp AS SELECT * FROM employees;
107 rows inserted.
```

```
DECLARE
  TYPE EmpRec IS RECORD (last_name employees.last_name%TYPE,
                        salary employees.salary%TYPE);
  emp_info EmpRec;
  emp_id NUMBER := 100;
BEGIN
  UPDATE emp_temp SET salary = salary * 1.1
    WHERE employee_id = emp_id
    RETURNING last_name, salary INTO emp_info;
  DBMS_OUTPUT.PUT_LINE
    ('Just gave a raise to ' || emp_info.last_name ||
    ', who now makes ' || emp_info.salary);
  ROLLBACK;
END;
/
```

```
Just gave a raise to King, who now makes 26400
```

```
PL/SQL procedure successfully completed.
```

## Using BULK COLLECT INTO with the RETURNING INTO clause

The following example uses `ttIsql` to run a SQL script that uses a `RETURNING INTO` clause with `BULK COLLECT` to return data into nested tables, a type of PL/SQL collection. The example deletes all the employees from a specified department, then, using one nested table for employee IDs and one for last names, outputs the employee ID and last name of each deleted employee. For reference, the IDs and last names of employees in the department are also displayed before execution of the script.

### **Example 5–22 Using BULK COLLECT INTO with the RETURNING INTO clause**

```
Command> select employee_id, last_name from employees where department_id=30;
< 114, Raphaely >
< 115, Khoo >
< 116, Baida >
< 117, Tobias >
< 118, Himuro >
< 119, Colmenares >
6 rows found.
```

```
Command> run ReturnIntoWithBulkCollect.sql;
```

```
CREATE TABLE emp_temp AS SELECT * FROM employees;
```

```

107 rows inserted.

DECLARE
  TYPE NumList IS TABLE OF employees.employee_id%TYPE;
  enums NumList;
  TYPE NameList IS TABLE OF employees.last_name%TYPE;
  names NameList;
BEGIN
  DELETE FROM emp_temp WHERE department_id = 30
    RETURNING employee_id, last_name
    BULK COLLECT INTO enums, names;
  DBMS_OUTPUT.PUT_LINE
    ('Deleted ' || SQL%ROWCOUNT || ' rows:');
  FOR i IN enums.FIRST .. enums.LAST
  LOOP
    DBMS_OUTPUT.PUT_LINE
      ('Employee #' || enums(i) || ': ' || names(i));
  END LOOP;
END;
/
Deleted 6 rows:
Employee #114: Raphaely
Employee #115: Khoo
Employee #116: Baida
Employee #117: Tobias
Employee #118: Himuro
Employee #119: Colmenares

PL/SQL procedure successfully completed.

```

## Example querying a system view

This section provides an example that queries a system view.

### **Example 5-23 Querying system view USER\_SOURCE**

This example queries the USER\_SOURCE system view to examine the source code of procedure query\_emp from [Example 5-4](#) on page 5-3. (You must create that procedure before completing this example.)

```

Command> SELECT SUBSTR (text, 1, LENGTH(text)-1)
          FROM user_source
          WHERE name = 'QUERY_EMP' AND type = 'PROCEDURE';

```

This produces the following output:

```

< PROCEDURE query_emp >
< (p_id IN employees.employee_id%TYPE, >
< p_name OUT employees.last_name%TYPE, >
< p_salary OUT employees.salary%TYPE) IS >
< BEGIN >
< SELECT last_name, salary INTO p_name, p_salary >
< FROM employees >
< WHERE employee_id = p_id; >
< END query_emp; >
9 rows found.

```

---

**Note:** As with other USER\_\* system views, all users have SELECT privilege for the USER\_SOURCE system view.

---

---



---

## PL/SQL Environment

The chapter shows you how to manage PL/SQL in your TimesTen database. It also describes the `ttSrcScan` utility, which you can use to check for PL/SQL features unsupported in TimesTen.

Topics in this chapter include:

- [PL/SQL connection attributes](#)
- [PL/SQL database configuration parameters](#)
- [The `ttSrcScan` utility](#)

### PL/SQL connection attributes

There are several TimesTen connection attributes specific to PL/SQL, as summarized in [Table 6–1](#) that follows. For additional information on these connection attributes, see "PL/SQL first connection attributes" and "PL/SQL general connection attributes" in *Oracle TimesTen In-Memory Database Reference*.

The table also notes any required access control privileges and whether each connection attribute is a first connection attribute or a general connection attribute. First connection attributes are set when the database is first loaded, and persist for all connections. Only the instance administrator can load a database with changes to first connection attribute settings. A general connection attribute setting applies to one connection only, and requires no special privilege.

This section includes an example using `ALTER SESSION` to change general connection attribute values. Also see "ALTER SESSION" in *Oracle TimesTen In-Memory Database SQL Reference*.

**Table 6–1 PL/SQL Connection Attributes**

Attribute	Summary
PLSCOPE_SETTINGS	<p>General connection attribute</p> <p>Required privilege: None</p> <p>Controls whether the PL/SQL compiler generates cross-reference information. Possible values are <code>IDENTIFIERS:NONE</code> or <code>IDENTIFIERS:ALL</code>.</p> <p>You can use the <code>ALTER SESSION</code> statement to change this value within your session.</p> <p>Also see "PLSCOPE_SETTINGS" in <i>Oracle TimesTen In-Memory Database Reference</i></p>

**Table 6–1 (Cont.) PL/SQL Connection Attributes**

Attribute	Summary
PLSQL_CCFLAGS	<p>General connection attribute</p> <p>Required privilege: None</p> <p>Use this to set inquiry directives to control conditional compilation of PL/SQL units, which enables you to customize the functionality of a PL/SQL program depending on conditions that are checked. This is especially useful when applications may be deployed to multiple database environments. For example, to activate debugging features:</p> <pre data-bbox="776 541 1094 562">PLSQL_CCFLAGS= 'DEBUG:TRUE'</pre> <p>PL/SQL conditional compilation flags are similar in concept to flags on a C compiler command line, such as the following:</p> <pre data-bbox="776 674 1019 695">% cc -DEBUG=TRUE ...</pre> <p>You can use the <code>ALTER SESSION</code> statement to change <code>PLSQL_CCFLAGS</code> within your session.</p> <p>See "Conditional Compilation" in <i>Oracle Database PL/SQL Language Reference</i> for information about this feature.</p> <p>Also see "PLSQL_CCFLAGS" in <i>Oracle TimesTen In-Memory Database Reference</i>.</p>
PLSQL_CONN_MEM_LIMIT	<p>General connection attribute</p> <p>Required privilege: None</p> <p>Specifies the maximum amount of PL/SQL shared memory (process heap memory) that PL/SQL can allocate for the current connection. (Note that this memory is not actually allocated until needed.) This is memory used for runtime data, such as large PL/SQL collections, as opposed to cached executable code. This limit setting protects other parts of your application, such as C or Java components, when PL/SQL might otherwise take all available runtime memory.</p> <p>The <code>PLSQL_CONN_MEM_LIMIT</code> value is a number specified in megabytes. A setting of 0 means no limit.</p> <p>You can use the <code>ALTER SESSION</code> statement to change this value within your session.</p> <p>Also see "PLSQL_CONN_MEM_LIMIT" in <i>Oracle TimesTen In-Memory Database Reference</i>.</p> <p><b>Note:</b> In <code>ttPLSQLMemoryStats</code> output, the related value <code>CurrentConnectionMemory</code> indicates how much process heap memory PL/SQL has actually acquired through <code>malloc()</code>. (Also see <a href="#">Example 6–3</a> on page 6-6.)</p>

**Table 6–1 (Cont.) PL/SQL Connection Attributes**

Attribute	Summary
PLSQL_MEMORY_ADDRESS	<p>First connection attribute</p> <p>Required privilege: Instance administrator</p> <p>Specifies the virtual address, as a hexadecimal value, at which the PL/SQL shared memory segment is loaded into each process that uses the TimesTen direct drivers. This memory address must be identical in all connections to a given database and in all processes that connect to that database.</p> <p>If a single application simultaneously makes direct connections to multiple databases, then you must set different values for each of the databases.</p> <p>Refer to "PLSQL_MEMORY_ADDRESS" in <i>Oracle TimesTen In-Memory Database Reference</i> for platform-specific information.</p>
PLSQL_MEMORY_SIZE	<p>First connection attribute</p> <p>Required privilege: Instance administrator</p> <p>Determines the size, in megabytes, of memory allocated for the PL/SQL shared memory segment, which is shared by all connections. This memory is used to hold recently executed PL/SQL code and metadata about PL/SQL objects.</p> <p>Refer to "PLSQL_MEMORY_SIZE" in <i>Oracle TimesTen In-Memory Database Reference</i> for information about calculating the PL/SQL memory size and for platform-specific values and tuning information.</p>
PLSQL_OPEN_CURSORS	<p>First connection attribute</p> <p>Required privilege: Instance administrator</p> <p>Specifies the maximum number of PL/SQL cursors that can be open in a session at one time.</p> <p>Note that this attribute has the same functionality as OPEN_CURSORS in Oracle Database.</p> <p>Also see "PLSQL_OPEN_CURSORS" in <i>Oracle TimesTen In-Memory Database Reference</i>.</p> <p><b>Note:</b> Alternatively, you can use the <code>ttDBConfig</code> parameter <code>PLSQL_OPEN_CURSORS</code>. See "<a href="#">PL/SQL database configuration parameters</a>" on page 6-7.</p>
PLSQL_OPTIMIZE_LEVEL	<p>General connection attribute</p> <p>Required privilege: None</p> <p>Specifies the optimization level used to compile PL/SQL library units. The higher the setting, the more effort the compiler makes to optimize PL/SQL library units. Possible values are 0, 1, 2, or 3.</p> <p>You can use the <code>ALTER SESSION</code> statement to change this value within your session.</p> <p>Also see "PLSQL_OPTIMIZE_LEVEL" in <i>Oracle TimesTen In-Memory Database Reference</i>.</p>

**Table 6–1 (Cont.) PL/SQL Connection Attributes**

Attribute	Summary
PLSQL_SESSION_CACHED_CURSORS	<p>General connection attribute</p> <p>Required privilege: None</p> <p>Specifies the number of session cursors to cache.</p> <p>This attribute has the same functionality as <code>SESSION_CACHED_CURSORS</code> in Oracle Database.</p> <p>You can use the <code>ALTER SESSION</code> statement to change this value within your session.</p> <p>Also see "PLSQL_SESSION_CACHED_CURSORS" in <i>Oracle TimesTen In-Memory Database Reference</i>.</p>
PLSQL_TIMEOUT	<p>General connection attribute</p> <p>Required privilege: None</p> <p>Controls how long PL/SQL program units are allowed to run, in seconds, before being terminated. A new value impacts PL/SQL programs currently running. Possible values are 0 (meaning no time limit) or any positive integer.</p> <p>You can use the <code>ALTER SESSION</code> statement to change this value within your session.</p> <p>Also see "PLSQL_TIMEOUT" in <i>Oracle TimesTen In-Memory Database Reference</i>.</p> <p>Be aware of TimesTen SQL query timeout settings, as discussed in "Setting a timeout duration for SQL statements" in <i>Oracle TimesTen In-Memory Database C Developer's Guide</i>, and the TimesTen <code>TTC_Timeout</code> setting (relevant for client/server), discussed in "TTC_Timeout" in <i>Oracle TimesTen In-Memory Database Reference</i>.</p> <p>If you use TimesTen Client/Server, <code>PLSQL_TIMEOUT</code> should be significantly less than <code>TTC_Timeout</code>, and cannot be 0 (for no timeout) if <code>TTC_Timeout</code> is greater than 0. For details, see the <code>TTC_Timeout</code> documentation referenced above.</p> <p>For additional information about the relationship between timeout values, see "Choose SQL and PL/SQL timeout values" in <i>Oracle TimesTen In-Memory Database Operations Guide</i>.</p> <p><b>Note:</b> The frequency with which PL/SQL programs check execution time against this timeout value is variable. It is possible for programs to run significantly longer than the timeout value before being terminated.</p>

---

**Notes:** There are additional TimesTen connection attributes you should consider for PL/SQL. For more information about them, refer to the indicated sections in *Oracle TimesTen In-Memory Database Reference*.

- If the `LockLevel` general connection attribute is set to 1 (database-level locking), certain PL/SQL internal functions cannot be performed. Therefore, set `LockLevel` to 0 for your connection. You can then use the `ttLockLevel` built-in procedure to selectively switch to database-level locking for those transactions that require it. See "LockLevel" and "ttLockLevel".
  - The PL/SQL shared memory segment is not subject to the `MemoryLock` first connection attribute. See "MemoryLock".
- 

The rest of this section provides some examples for setting and altering PL/SQL connection attributes.

---

**Note:** The following example applies to TimesTen Classic. In TimesTen Scaleout, you set connection attributes when you define a connectable and you are not allowed to manually edit configuration files such as `odbc.ini`. See "Connecting to a database" in *Oracle TimesTen In-Memory Database Scaleout User's Guide*.

---

#### **Example 6-1 Create a database with PL/SQL default connection attributes**

This example defines a database `pldef` without specifying PL/SQL connection attributes. (Be aware that only an instance administrator can create a database.)

Sample `odbc.ini` entry:

```
[pldef]
Driver=/mypath/install/lib/libtten.so
DataStore=/mypath/install/info/DemoDataStore/pldef
DatabaseCharacterSet=US7ASCII
```

Connect to database `pldef`:

```
% ttisql pldef
```

```
Copyright (c) 1996-2011, Oracle. All rights reserved.
Type ? or "help" for help, type "exit" to quit ttIsql.
```

```
connect "DSN=pldef";
Connection successful:
DSN=pldef;UID=myuser;DataStore=/mypath/install/info/DemoDataStore/pldef;
DatabaseCharacterSet=US7ASCII;ConnectionCharacterSet=US7ASCII;
DRIVER=/mypath/install/lib/libtten.so;
(Default setting AutoCommit=1)
```

Call the `ttConfiguration` built-in procedure to display settings, which shows you the default PL/SQL settings:

```
Command> call ttconfiguration;
...
< PLScope_Settings, IDENTIFIERS:NONE >
...
< PLSQL_CCFLAGS, <NULL> >
```

```
...
< PLSQL_CONN_MEM_LIMIT, 100 >
< PLSQL_MEMORY_ADDRESS, 0x0000005000000000 >
< PLSQL_MEMORY_SIZE, 128 >
< PLSQL_OPTIMIZE_LEVEL, 2 >
< PLSQL_TIMEOUT, 30 >
...
< PLSQL_SESSION_CACHED_CURSORS, 50 >
...
88 rows found.
```

### **Example 6–2 Use ALTER SESSION to change attribute settings**

This example uses ALTER SESSION statements to alter PL/SQL connection attributes, changing the settings of PLScope\_SETTINGS, PLSQL\_CONN\_MEM\_LIMIT, and PLSQL\_OPTIMIZE\_LEVEL. It then calls the ttConfiguration built-in procedure to display the new values.

```
Command> ALTER SESSION SET PLScope_SETTINGS = "IDENTIFIERS:ALL";
```

```
Session altered.
```

```
Command> ALTER SESSION SET PLSQL_CONN_MEM_LIMIT=200;
```

```
Session altered.
```

```
Command> ALTER SESSION SET PLSQL_OPTIMIZE_LEVEL=3;
```

```
Session altered.
```

```
Command> call ttconfiguration;
```

```
...
< DataStore, /mypath/install/info/DemoDataStore/pldef >
...
< PLScope_SETTINGS, IDENTIFIERS:ALL >
...
< PLSQL_CONN_MEM_LIMIT, 200 >
...
< PLSQL_OPTIMIZE_LEVEL, 3 >
...
< UID, MYUSER >
61 rows found.
```

### **Example 6–3 View PL/SQL performance statistics**

The ttPLSQLMemoryStats built-in procedure returns statistics about PL/SQL library cache performance and activity. This example shows some sample output. Refer to "ttPLSQLMemoryStats" in *Oracle TimesTen In-Memory Database Reference* for information about this procedure.

```
Command> call ttplsqlmemorystats;
< Gets, 5.000000 >
< GetHits, 0.000000e+00 >
< GetHitRatio, 0.000000e+00 >
< Pins, 4.000000 >
< PinHits, 0.000000e+00 >
< PinHitRatio, 0.000000e+00 >
< Reloads, 0.000000e+00 >
< Invalidations, 0.000000e+00 >
< CurrentConnectionMemory, 0.000000e+00 >
< DeferredCleanups, 0.000000e+00 >
```

10 rows found.

---



---

**Note:** CurrentConnectionMemory is related to the PLSQL\_CONN\_MEM\_LIMIT connection attribute documented in "[PL/SQL connection attributes](#)" on page 6-1, indicating the amount of heap memory that has actually been acquired by PL/SQL.

---



---

## PL/SQL database configuration parameters

You can use the ttDBConfig built-in procedure to display or set the value of the database configuration parameter PLSQL\_OPEN\_CURSORS.

This parameter specifies the maximum number of PL/SQL cursors that can be open in a session at one time. Use it to prevent a session from opening an excessive number of cursors.

To retrieve the current value:

```
call ttDbconfig('PLSQL_OPEN_CURSORS');
< PLSQL_OPEN_CURSORS, 50 >
1 row found.
```

To set the value to 75:

```
call ttDbconfig('PLSQL_OPEN_CURSORS', '75');
< PLSQL_OPEN_CURSORS, 75 >
1 row found.
```

Refer to "ttDBConfig" in *Oracle TimesTen In-Memory Database Reference* for information about this built-in procedure and additional information about PLSQL\_OPEN\_CURSORS.

---



---

**Note:** Alternatively, you can use the PLSQL\_OPEN\_CURSORS connection attribute. See "[PL/SQL connection attributes](#)" on page 6-1.

---



---

## The ttSrcScan utility

If you have an existing PL/SQL program and want to see whether it uses PL/SQL features that TimesTen does not support, you can use the ttSrcScan command line utility to scan your program for unsupported functions, packages, types, type codes, attributes, modes, and constants. This is a standalone utility that can be run without TimesTen or Oracle Database being installed and runs on any platform supported by TimesTen. It reads source code files as input and creates HTML and text files as output. If the utility finds unsupported items, they are logged and alternatives are suggested. Specify an input file or directory for the program to be scanned and an output directory for the ttSrcScan reports. Other options are available as well.

The ttSrcScan utility is available on the Oracle Technology Network site. See the README file for information.



---

---

## TimesTen Supplied PL/SQL Packages

TimesTen supplies public PL/SQL packages, listed immediately below, to extend database functionality and provide PL/SQL access to SQL features. TimesTen installs these packages automatically for your use. Packages that are part of the PL/SQL language itself or are for TimesTen or Oracle Database internal use only are not shown here or described in this chapter.

This chapter lists and briefly describes the subprograms that comprise each package. For details on these PL/SQL packages, refer to *Oracle TimesTen In-Memory Database PL/SQL Packages Reference*.

- [DBMS\\_LOB](#)
- [DBMS\\_LOCK](#)
- [DBMS\\_OUTPUT](#)
- [DBMS\\_PREPROCESSOR](#)
- [DBMS\\_RANDOM](#)
- [DBMS\\_SQL](#)
- [DBMS\\_UTILITY](#)
- [TT\\_DB\\_VERSION](#)
- [TT\\_STATS](#)
- [UTL\\_FILE](#)
- [UTL\\_IDENT](#)
- [UTL\\_RAW](#)
- [UTL\\_RECOMP](#)

## DBMS\_LOB

The `DBMS_LOB` package provides subprograms to operate on BLOBs, CLOBs, and NCLOBs, including temporary LOBs, in TimesTen Classic.

---



---

### Notes:

- TimesTen does not support `DBMS_LOB` subprograms intended specifically for BFILES, SecureFiles (including Database File System features), or inserting or deleting data fragments in the middle of a LOB.
- `DBMS_LOB` procedures and functions are supported for both TimesTen LOBs and passthrough LOBs, which are LOBs in Oracle Database accessed through TimesTen and exposed as TimesTen LOBs. Note, however, that `CREATETEMPORARY` can only be used to create a temporary LOB in TimesTen. If a temporary passthrough LOB is created using some other mechanism, such as SQL, `ISTEMPORARY` and `FREETEMPORARY` can be used on that LOB.

As with TimesTen local LOBs, a locator for a passthrough LOB does not remain valid past the end of the transaction.

In addition to copying from one TimesTen LOB to another, `COPY` can copy from a TimesTen LOB to a passthrough LOB, from a passthrough LOB to a TimesTen LOB, or from one passthrough LOB to another passthrough LOB. An attempt to copy a passthrough LOB to a TimesTen LOB when the passthrough LOB is larger than the TimesTen LOB size limit results in an error. ("COPY procedures" in *Oracle TimesTen In-Memory Database PL/SQL Packages Reference* provides examples for copying LOBs.)

See "[Passthrough LOBs](#)" on page 2-28 for related information.

---



---

[Table 7-1](#) describes the supported `DBMS_LOB` subprograms.

**Table 7-1** *DBMS\_LOB subprograms*

Subprogram	Description
APPEND procedures	Appends the contents of the source LOB to the destination LOB.
CLOSE procedures	Closes a previously opened LOB.
COMPARE functions	Compares two entire LOBs or parts of two LOBs.
CONVERTTOBLOB procedure	Reads character data from a source CLOB or NCLOB instance, converts the character data to the specified character set, writes the converted data to a destination BLOB instance in binary format, and returns the new offsets.
CONVERTTOCLOB procedure	Takes a source BLOB instance, converts the binary data in the source instance to character data using the specified character set, writes the character data to a destination CLOB or NCLOB instance, and returns the new offsets.
COPY procedures	Copies all or part of the source LOB to the destination LOB.

**Table 7-1 (Cont.) DBMS\_LOB subprograms**

<b>Subprogram</b>	<b>Description</b>
CREATETEMPORARY procedures	Creates a temporary LOB in the temporary data region. Any of the durations supported by Oracle Database is permitted (SESSION, TRANSACTION, or CALL). In TimesTen, however, LOB duration cannot extend past the end of the transaction.
ERASE procedures	Erases all or part of a LOB.
FREETEMPORARY procedures	Frees a temporary LOB in the temporary data region.
GET_STORAGE_LIMIT functions	Returns the storage limit for the LOB type of the specified LOB.
GETCHUNKSIZE functions	In TimesTen, this simply returns the value 32 KB for interoperability. Do not rely on this value for performance tuning.
GETLENGTH functions	Returns the length of the LOB value, in bytes for a BLOB or characters for a CLOB or NCLOB.
INSTR functions	Returns the matching position of the <i>n</i> th occurrence of the pattern in the LOB.
ISOPEN functions	Checks to see if the LOB was already opened using the input locator.
ISTEMPORARY functions	Checks whether the locator is pointing to a temporary LOB.
OPEN procedures	Opens a LOB (persistent or temporary) in the indicated mode, read/write or read-only. <b>Note:</b> Opening a LOB is similar conceptually, but not technically, to opening a file. Opening a LOB is more like a hint regarding resources to be required.
READ procedures	Reads data from the LOB starting at the specified offset.
SUBSTR functions	Returns part of the LOB value starting at the specified offset.
TRIM procedures	Trims the LOB value to the specified shorter length.
WRITE procedures	Writes data to the LOB from a specified offset.
WRITEAPPEND procedures	Writes a buffer to the end of a LOB.

## DBMS\_LOCK

---

The `DBMS_LOCK` package provides an interface to lock-management services. In the current release, TimesTen supports only the sleep feature.

[Table 7-2](#) describes the supported `DBMS_LOCK` subprogram.

**Table 7-2** *DBMS\_LOCK subprograms*

Subprogram	Description
<code>SLEEP</code> procedure	<p>This procedure suspends the session for a given duration. Specify the amount of time in seconds. The smallest supported increment is a hundredth of a second. For example:</p> <pre>DBMS_LOCK.SLEEP(1.95);</pre> <p><b>Notes:</b></p> <ul style="list-style-type: none"><li>▪ The actual sleep time may be somewhat longer than specified, depending on system activity.</li><li>▪ If <code>PLSQL_TIMEOUT</code> is set to a positive value that is less than this sleep time, the timeout takes effect first. Be sure that either the sleep value is less than the timeout value, or <code>PLSQL_TIMEOUT=0</code> (no timeout). See "<a href="#">PL/SQL connection attributes</a>" on page 6-1 for information about <code>PLSQL_TIMEOUT</code>.</li></ul>

---

---

## DBMS\_OUTPUT

The DBMS\_OUTPUT package enables you to send messages from stored procedures and packages. The package is useful for displaying PL/SQL debugging information.

[Table 7-3](#) describes the DBMS\_OUTPUT subprograms.

**Table 7-3 DBMS\_OUTPUT subprograms**

<b>Subprogram</b>	<b>Description</b>
DISABLE procedure	Disables message output.
ENABLE procedure	Enables message output.
GET_LINE procedure	Retrieves one line from the buffer.
GET_LINES procedure	Retrieves an array of lines from the buffer.
NEW_LINE procedure	Terminates a line created with PUT.
PUT procedure	Places a line in the buffer.
PUT_LINE procedure	Places a partial line in the buffer.

## DBMS\_PREPROCESSOR

The DBMS\_PREPROCESSOR package provides an interface to print or retrieve the source text of a PL/SQL unit after processing of conditional compilation directives.

[Table 7-4](#) describes the DBMS\_PREPROCESSOR subprograms.

**Table 7-4** *DBMS\_PREPROCESSOR subprograms*

<b>Subprogram</b>	<b>Description</b>
GET_POST_PROCESSED_SOURCE function	Returns post-processed source text.
PRINT_POST_PROCESSED_SOURCE procedure	Prints post-processed source text.

---

## DBMS\_RANDOM

The DBMS\_RANDOM package provides a built-in random number generator.

[Table 7-5](#) describes the DBMS\_RANDOM subprograms.

**Table 7-5 DBMS\_RANDOM subprograms**

Subprogram	Description
INITIALIZE procedure	Initializes the package with a seed value (deprecated).
NORMAL function	Returns random numbers in a normal distribution.
RANDOM procedure	Generates a random number (deprecated).
SEED procedure	Resets the seed.
STRING function	Gets a random string.
TERMINATE procedure	Terminates the package (deprecated).
VALUE function	There are two overloaded versions. In the first, it gets a random number greater than or equal to 0 and less than 1, with 38 digits to the right of the decimal point (38-digit precision). In the second, it gets a random number within specified low and high limits.

## DBMS\_SQL

The `DBMS_SQL` package provides an interface for using dynamic SQL to accomplish any of the following:

- Execute data manipulation language (DML) and data definition language (DDL) statements.
- Execute PL/SQL anonymous blocks.
- Call PL/SQL stored procedures and functions.

This package does not support pre-defined data types and overloads with data types that are not supported in TimesTen, such as `UROWID`, time zone features, ADT, database-level collections, and edition overloads. For more information on the supported data types in TimesTen PL/SQL, see "[Understanding the data type environments](#)" on page 3-1.

[Table 7-6](#) describes the `DBMS_SQL` subprograms.

**Table 7-6** *DBMS\_SQL subprograms*

Subprogram	Description
<code>BIND_ARRAY</code> procedure	Binds a given value to a given collection.
<code>BIND_VARIABLE</code> procedure	Binds a given value to a given variable.
<code>CLOSE_CURSOR</code> procedure	Closes a given cursor and frees memory.
<code>COLUMN_VALUE</code> procedure	Returns the value of the cursor element for a given position in a cursor.
<code>COLUMN_VALUE_LONG</code> procedure	Returns a selected part of a <code>LONG</code> column that has been defined using <code>DEFINE_COLUMN_LONG</code> . <b>Important:</b> Because TimesTen does not support the <code>LONG</code> data type, attempting to use this procedure in TimesTen results in an <code>ORA-01018</code> error at runtime.
<code>DEFINE_ARRAY</code> procedure	Defines a collection to be selected from the given cursor. Use with <code>SELECT</code> statements.
<code>DEFINE_COLUMN</code> procedure	Defines a column to be selected from the given cursor. Use with <code>SELECT</code> statements.
<code>DEFINE_COLUMN_LONG</code> procedure	Defines a <code>LONG</code> column to be selected from the given cursor. Use with <code>SELECT</code> statements. <b>Important:</b> Because TimesTen does not support the <code>LONG</code> data type, attempting to use the <code>COLUMN_VALUE_LONG</code> procedure in TimesTen results in an <code>ORA-01018</code> error at runtime. <code>DEFINE_COLUMN_LONG</code> would be used with <code>COLUMN_VALUE_LONG</code> .
<code>DESCRIBE_COLUMNS</code> procedure	Describes the columns for a cursor opened and parsed through the <code>DBMS_SQL</code> package.
<code>DESCRIBE_COLUMNS2</code> procedure	Describes the specified column. Use as an alternative to <code>DESCRIBE_COLUMNS</code> procedure.
<code>DESCRIBE_COLUMNS3</code> procedure	Describes the specified column. Use as an alternative to <code>DESCRIBE_COLUMNS</code> procedure.
<code>EXECUTE</code> function	Executes a given cursor.
<code>EXECUTE_AND_FETCH</code> function	Executes a given cursor and fetches rows.
<code>FETCH_ROWS</code> function	Fetches a row from a given cursor.

**Table 7–6 (Cont.) DBMS\_SQL subprograms**

<b>Subprogram</b>	<b>Description</b>
IS_OPEN function	Returns TRUE if a given cursor is open.
LAST_ERROR_POSITION function	Returns the byte offset in the SQL statement text where the error occurred.
LAST_ROW_COUNT function	Returns a cumulative count of the number of rows fetched.
LAST_ROW_ID function	Returns NULL. TimesTen does not support ROWID of the last row operated on by a DML statement.
LAST_SQL_FUNCTION_CODE function	Returns the SQL function code for the statement.
OPEN_CURSOR function	Returns the cursor ID number of a new cursor.
PARSE procedures	Parses a given statement.
TO_CURSOR_NUMBER function	Takes an opened (by OPEN) strongly or weakly typed REF CURSOR and transforms it into a DBMS_SQL cursor number.
TO_REFCURSOR function	Takes an opened, parsed, and executed cursor (by OPEN, PARSE, and EXECUTE) and transforms or migrates it into a PL/SQL manageable REF CURSOR (a weakly typed cursor) that can be consumed by PL/SQL native dynamic SQL and switched to use native dynamic SQL.
VARIABLE_VALUE procedures	Returns value of a named variable for a given cursor.

---

## DBMS\_UTILITY

The DBMS\_UTILITY package provides a variety of utility subprograms.

Subprograms are not supported (and not listed here) for features that TimesTen does not support.

Table 7-7 describes DBMS\_UTILITY subprograms.

**Table 7-7 DBMS\_UTILITY subprograms**

Subprogram	Description
CANONICALIZE procedure	Canonicalizes a given string.
COMMA_TO_TABLE procedure	Converts a comma-delimited list of names into an associative array (index-by table) of names.
COMPILE_SCHEMA	Compiles all procedures, functions, packages, and views in the specified database schema.
DB_VERSION procedure	Returns version information for the database. The procedure returns NULL for the compatibility setting because TimesTen does not support the system parameter COMPATIBLE.
FORMAT_CALL_STACK function	Formats the current call stack.
FORMAT_ERROR_BACKTRACE function	Formats the backtrace from the point of the current error to the exception handler where the error is caught.
FORMAT_ERROR_STACK function	Formats the current error stack.
GET_CPU_TIME function	Returns the current CPU time in hundredths of a second.
GET_DEPENDENCY procedure	Shows the dependencies on the objects passed in.
GET_ENDIANNESSESS function	Returns the endianness of your database platform.
GET_HASH_VALUE function	Computes a hash value for a given string.
GET_SQL_HASH function	Computes the hash value for a given string using the MD5 algorithm.
GET_TIME function	Returns the current time in hundredths of a second.
INVALIDATE procedure	Invalidates a database object and optionally modifies the PL/SQL compiler parameter settings for the object.
IS_BIT_SET function	Checks the setting of a specified bit in a RAW value.
NAME_RESOLVE procedure	Resolves the given name of the following form: [[a.]b.]c[@dblink] Where <i>a</i> , <i>b</i> , and <i>c</i> are SQL identifiers and <i>dblink</i> is a dblink (database link). Do not use @dblink. TimesTen does not support dblinks.

**Table 7-7 (Cont.) DBMS\_UTILITY subprograms**

Subprogram	Description
NAME_TOKENIZE procedure	<p data-bbox="927 260 1438 317">Calls the parser to parse the given name of the following form:</p> <pre data-bbox="927 327 1208 354">"a [.b [.c ]][@dblink]"</pre> <p data-bbox="927 365 1438 470">Strips double quotes or converts to uppercase if there are no quotes. Ignores comments and does not perform semantic analysis. Missing values are NULL.</p> <p data-bbox="927 480 1438 533">Do not use <i>@dblink</i>. TimesTen does not support dblinks.</p>
TABLE_TO_COMMA procedures	Converts an associative array (index-by table) of names into a comma-delimited list of names.
VALIDATE procedure	Validates the object described by either owner, name and namespace, or object ID.

## TT\_DB\_VERSION

The TT\_DB\_VERSION package contains boolean constants indicating the current TimesTen release.

[Table 7–8](#) describes the TT\_DB\_VERSION constants.

The primary use case for the TT\_DB\_VERSION and UTL\_IDENT packages is for conditional compilation.

**Table 7–8 TT\_DB\_VERSION constants**

Name	Description
VER_LE_1121	Boolean that is TRUE if this package ships with TimesTen Release 11.2.1 or prior. FALSE for TimesTen 11g Release 2 (11.2.2) or higher.
VER_LE_1122	Boolean that is TRUE if this package ships with TimesTen 11g Release 2 (11.2.2) or prior. FALSE for TimesTen Release 18.1.
VER_LE_181	Boolean that is TRUE if this package ships with TimesTen Release 18.1 or prior.

See "Examples" in *Oracle TimesTen In-Memory Database PL/SQL Packages Reference* for an example that uses TT\_DB\_VERSION and UTL\_IDENT.

---

## TT\_STATS

The TT\_STATS package provides features for collecting and comparing snapshots of TimesTen system metrics, according to the capture level. Each snapshot can consist of what TimesTen considers to be basic metrics, typical metrics, or all available metrics.

For those familiar with Oracle Database performance analysis tools, these reports are similar in nature to Oracle Automatic Workload Repository (AWR) reports.

**Table 7–9 TT\_STATS subprograms**

Subprogram	Description
CAPTURE_SNAPSHOT procedure and function	Takes a snapshot of TimesTen metrics. The function also returns the snapshot ID.
DROP_SNAPSHOTS_RANGE function	Deletes snapshots according to a specified range of snapshot IDs or timestamps.
GENERATE_REPORT_HTML procedure	Produces a report in HTML format based on the data from two specified snapshots.
GENERATE_REPORT_TEXT procedure	Produces a report in plain text format based on the data from two specified snapshots.
GET_CONFIG function	Retrieves the value of a specified TT_STATS configuration parameter or the values of all configuration parameters.
SET_CONFIG procedure	Sets a specified value for a specified TT_STATS configuration parameter.
SHOW_SNAPSHOTS function	Shows the snapshot IDs and timestamps of all snapshots currently stored in the database.

---

## UTL\_FILE

The UTL\_FILE package enables PL/SQL programs the ability to read and write operating system text files.

In the current release, this package is restricted to access of a pre-defined temporary directory only. Refer to the *Oracle TimesTen In-Memory Database Release Notes* for details.

---

**Note:** Users do not have execute permission on UTL\_FILE by default. To use UTL\_FILE in TimesTen, an ADMIN user or instance administrator must explicitly grant EXECUTE permission on it, such as in the following example:

```
GRANT EXECUTE ON SYS.UTL_FILE TO scott;
```

---

Table 7–10 describes the UTL\_FILE subprograms.

**Table 7–10 UTL\_FILE subprograms**

Subprogram	Description
FCLOSE procedure	Closes a file.
FCLOSE_ALL procedure	Closes all file handles.
FCOPY procedure	Copies a contiguous portion of a file to a newly created file.
FFLUSH procedure	Physically writes all pending output to a file.
FGETATTR procedure	Reads and returns the attributes of a file.
FGETPOS procedure	Returns the current relative offset position (in bytes) within a file.
FOPEN function	Opens a file for input or output.
FOPEN_NCHAR function	Opens a file in Unicode for input or output.
FREMOVE procedure	With sufficient privilege, deletes a file.
FRENAME procedure	Renames an existing file to a new name (similar to the UNIX mv command).
FSEEK procedure	Adjusts the file pointer forward or backward within the file by the number of bytes specified.
GET_LINE procedure	Reads text from an open file.
GET_LINE_NCHAR procedure	Reads text in Unicode from an open file.
GET_RAW function	Reads a RAW string value from a file and adjusts the file pointer ahead by the number of bytes read.
IS_OPEN function	Determines if a file handle refers to an open file.
NEW_LINE procedure	Writes one or more operating system-specific line terminators to a file.
PUT procedure	Writes a string to a file.
PUT_LINE procedure	Writes a line to a file and appends an operating system-specific line terminator.
PUT_LINE_NCHAR procedure	Writes a Unicode line to a file.
PUT_NCHAR procedure	Writes a Unicode string to a file.

**Table 7-10 (Cont.) UTL\_FILE subprograms**

<b>Subprogram</b>	<b>Description</b>
PUT_RAW function	Accepts as input a RAW data value and writes the value to the output buffer.
PUTF procedure	This is similar to the PUT procedure, but with formatting.
PUTF_NCHAR procedure	This is similar to the PUT_NCHAR procedure, but with formatting. Writes a Unicode string to a file with formatting.

## UTL\_IDENT

The UTL\_IDENT package indicates whether PL/SQL is running on TimesTen, an Oracle database client, an Oracle database server, or Oracle Forms. Each of these has its own version of UTL\_IDENT with appropriate settings for the constants.

Table 7-11 shows the UTL\_IDENT settings for TimesTen.

The primary use case for the UTL\_IDENT package is for conditional compilation, resembling the following:

```
$if utl_ident.is_oracle_server $then
    [...Run code supported for Oracle Database...]
$elsif utl_ident.is_timesten $then
    [...code supported for TimesTen Database...]
$end
```

**Table 7-11 UTL\_IDENT Constants**

Name	Description
IS_ORACLE_CLIENT	BOOLEAN set to FALSE
IS_ORACLE_SERVER	BOOLEAN set to FALSE
IS_ORACLE_FORMS	BOOLEAN set to FALSE
IS_TIMESTEN	BOOLEAN set to TRUE

See "Examples" in *Oracle TimesTen In-Memory Database PL/SQL Packages Reference* for an example that uses TT\_DB\_VERSION and UTL\_IDENT.

## UTL\_RAW

The UTL\_RAW package provides SQL functions for manipulating RAW data types.

Table 7–12 describes the UTL\_RAW subprograms.

**Table 7–12 UTL\_RAW subprograms**

Subprogram	Description
BIT_AND function	Performs bitwise logical "and" of two RAW values and returns the resulting RAW.
BIT_COMPLEMENT function	Performs bitwise logical "complement" of a RAW value and returns the resulting RAW.
BIT_OR function	Performs bitwise logical "or" of two RAW values and returns the resulting RAW.
BIT_XOR function	Performs bitwise logical "exclusive or" of two RAW values and returns the resulting RAW.
CAST_FROM_BINARY_DOUBLE function	Returns the RAW binary representation of a BINARY_DOUBLE value.
CAST_FROM_BINARY_FLOAT function	Returns the RAW binary representation of a BINARY_FLOAT value.
CAST_FROM_BINARY_INTEGER function	Returns the RAW binary representation of a BINARY_INTEGER value.
CAST_FROM_NUMBER function	Returns the RAW binary representation of a NUMBER value.
CAST_TO_BINARY_DOUBLE function	Casts the RAW binary representation of a BINARY_DOUBLE value into a BINARY_DOUBLE.
CAST_TO_BINARY_FLOAT function	Casts the RAW binary representation of a BINARY_FLOAT value into a BINARY_FLOAT.
CAST_TO_BINARY_INTEGER function	Casts the RAW binary representation of a BINARY_INTEGER value into a BINARY_INTEGER.
CAST_TO_NUMBER function	Casts the RAW binary representation of a NUMBER value into a NUMBER.
CAST_TO_NVARCHAR2 function	Casts a RAW value represented using <i>n</i> data bytes into an NVARCHAR2 value with <i>n</i> data bytes.
CAST_TO_RAW function	Casts a VARCHAR2 value represented using <i>n</i> data bytes into a RAW with <i>n</i> data bytes.
CAST_TO_VARCHAR2 function	Casts a RAW value represented using <i>n</i> data bytes into a VARCHAR2 value with <i>n</i> data bytes.
COMPARE function	Compares two RAW values.
CONCAT function	Concatenates up to 12 RAW values into a single RAW value.
CONVERT function	Converts a RAW value from one character set to another and returns the resulting RAW.
COPIES function	Copies a RAW value a specified number of times and returns the concatenated RAW value.
LENGTH function	Returns the length in bytes of a RAW value.

**Table 7-12 (Cont.) UTL\_RAW subprograms**

<b>Subprogram</b>	<b>Description</b>
OVERLAY function	Overlays the specified portion of a target RAW value with an overlay RAW value, starting from a specified byte position and proceeding for a specified number of bytes.
REVERSE function	Reverses a byte-sequence in a RAW value.
SUBSTR function	Returns a substring of a RAW value for a specified number of bytes from a specified starting position.
TRANSLATE function	Translates the specified bytes from an input RAW value according to the bytes in a specified translation RAW value.
TRANSLITERATE function	Converts the specified bytes from an input RAW value according to the bytes in a specified transliteration RAW value.
XRANGE function	Returns a RAW value containing the succession of one-byte encodings beginning and ending with the specified byte-codes.

---

## UTL\_RECOMP

The UTL\_RECOMP package recompiles invalid PL/SQL modules. This is particularly useful after a major-version upgrade that typically invalidates all PL/SQL objects.

Table 7–13 describes the UTL\_RECOMP subprograms.

---

**Important:** To use this package, you must be the instance administrator and specify `SYS.UTL_RECOMP`.

---

**Table 7–13** *UTL\_RECOMP subprograms*

Name	Description
RECOMP_PARALLEL procedure	Recompiles invalid objects in a given schema, or all invalid objects in the database, in parallel.  <b>Note:</b> Because TimesTen does not support DBMS_SCHEDULER, the number of recompile threads to run in parallel is always 1, regardless of what the user specifies. Therefore there is no effective difference between RECOMP_PARALLEL and RECOMP_SERIAL in TimesTen.
RECOMP_SERIAL procedure	Recompiles invalid objects in a given schema, or all invalid objects in the database, serially.



---

---

## TimesTen PL/SQL Support: Reference Summary

This chapter lists PL/SQL language elements and features supported in TimesTen. (In the Oracle Database documentation, many of these features are covered in "PL/SQL Language Elements" in *Oracle Database PL/SQL Language Reference*.)

Note that these are *not* supported in TimesTen:

- **Features:** autonomous transactions, database links (dblinks), result cache, savepoints, triggers, user-defined objects or collections

While TimesTen does not support triggers, you can achieve similar functionality using XLA. See "XLA and TimesTen Event Management" in *Oracle TimesTen In-Memory Database C Developer's Guide*.

- **Statements:** CREATE LIBRARY, CREATE TYPE, LOCK TABLE, SAVEPOINT, SET TRANSACTION
- **Pragmas:** AUTONOMOUS\_TRANSACTIONS, RESTRICT\_REFERENCES, SERIALLY\_REUSABLE
- **SYSTIMESTAMP:** TimesTen cannot support this because the return type, TIMESTAMP WITH TIME ZONE, is not supported. As an alternative, you can use SELECT tt\_sysdate FROM dual using dynamic SQL:

```
declare
  ts timestamp;
begin
  execute immediate
    'select tt_sysdate from dual'
    into ts;
  -- ts has millisecond resolution
end;
```

**Table 8–1 PL/SQL language element and feature support in TimesTen**

Feature name	Description	Example/comment
ALTER {PROCEDURE  FUNCTION   PACKAGE} statements	Recompiles a PL/SQL procedure, function, or package.	Syntax and semantics are the same as in Oracle Database.  For information about these statements, see "SQL Statements" in <i>Oracle TimesTen In-Memory Database SQL Reference</i> .
ALTER SESSION statement	Changes session parameters dynamically.	In TimesTen, you can use ALTER SESSION to set some PL/SQL connection attributes as discussed in "PL/SQL connection attributes" on page 6-1.  For more information on this statement in TimesTen, see "ALTER SESSION" in <i>Oracle TimesTen In-Memory Database SQL Reference</i> .
Assignment statement	Sets current value of a variable, parameter, or element.	See "PL/SQL variables and constants" on page 2-2.
Block declaration	Declares a block, the basic unit of a PL/SQL source program.	See "PL/SQL blocks" on page 2-1.
BULK COLLECT clause	Select multiple rows.	This clause can be used with the SELECT statement in PL/SQL to retrieve rows without using a cursor. See "FORALL and BULK COLLECT operations" on page 2-22 and "Examples using FORALL and BULK COLLECT" on page 5-9.
CALL statement	Executes a routine from within SQL.	In TimesTen, use the CALL statement to execute PL/SQL stored procedures and functions, as in Oracle Database, or TimesTen built-in procedures. (For TimesTen built-in procedures, use EXECUTE IMMEDIATE if CALL is inside PL/SQL.) See "Executing procedures and functions" on page 2-9 and Example 5–19, "Using EXECUTE IMMEDIATE to call ttConfiguration" on page 5-16.
CASE statement	Evaluates an expression, compares it against several values, and takes action according to the comparison that is true.	See "PL/SQL control structures" on page 2-5.
CLOSE statement	Closes a cursor or cursor variable.	See Example 2–17, "Using a cursor to retrieve information about an employee" on page 2-28 (among others).
Collection definition	Specifies a collection, which is an ordered group of elements that are all of the same type.	Examples include associative arrays (index-by tables or PL/SQL tables), nested tables, and varrays.  TimesTen supports all three of these collection types in PL/SQL programs, but supports only associative arrays as bound parameters between PL/SQL and applications written in other languages (such as OCI or JDBC).  See "Using collections in PL/SQL" on page 3-5.
Collection methods	Built-in subprograms that operate on collections and are called using "dot" notation.	See "Collection Methods" in <i>Oracle Database PL/SQL Language Reference</i> . Examples include COUNT, DELETE, EXISTS, EXTEND, FIRST, LAST, LIMIT, NEXT, PRIOR, and TRIM.

**Table 8–1 (Cont.) PL/SQL language element and feature support in TimesTen**

Feature name	Description	Example/comment
Comments	Text included within your code for explanatory purposes.	Single-line and multi-line comments are supported.
COMMIT statement	Ends the current transaction and makes permanent all changes performed in the transaction.	See "COMMIT" in <i>Oracle TimesTen In-Memory Database SQL Reference</i> . <b>Important:</b> COMMIT and ROLLBACK statements close all cursors in TimesTen.
Connection attributes	Equivalent to initialization parameters in Oracle Database.	See "PL/SQL connection attributes" on page 6-1. Also see "PL/SQL first connection attributes" and "PL/SQL general connection attributes" in <i>Oracle TimesTen In-Memory Database Reference</i> .
Constant and variable declarations	Specify constants and variables to be used in PL/SQL code, in the declarative part of any PL/SQL block, subprogram, or package.	See "PL/SQL variables and constants" on page 2-2.
CONTINUE statement	Exits the current iteration of a loop and transfers control to the next iteration.	See "CONTINUE statement" on page 2-7.
CREATE FUNCTION statement	Creates a PL/SQL function.	CREATE FUNCTION is supported in TimesTen Classic, but the AS LANGUAGE, AS EXTERNAL, and PIPELINED clauses are not supported. The ACCESSIBLE BY clause is supported. See "PL/SQL procedures and functions" on page 2-8. Also see "CREATE FUNCTION" in <i>Oracle TimesTen In-Memory Database SQL Reference</i> . You are not required to run DBMSSTDx.SQL in TimesTen.
CREATE PACKAGE statement CREATE PACKAGE BODY statement	These statements are used together to create a PL/SQL package definition and package body.	CREATE PACKAGE is supported in TimesTen Classic. Syntax and semantics are the same as in Oracle Database. The ACCESSIBLE BY clause is supported. See "PL/SQL packages" on page 2-12. Also see "CREATE PACKAGE" and "CREATE PACKAGE BODY" in <i>Oracle TimesTen In-Memory Database SQL Reference</i> . You are not required to run DBMSSTDx.SQL in TimesTen.
CREATE PROCEDURE statement	Creates a PL/SQL procedure.	CREATE PROCEDURE is supported in TimesTen Classic, but the AS LANGUAGE and AS EXTERNAL clauses are not supported. The ACCESSIBLE BY clause is supported. See "PL/SQL procedures and functions" on page 2-8. Also see "CREATE PROCEDURE" in <i>Oracle TimesTen In-Memory Database SQL Reference</i> . <b>Note:</b> You are not required to run DBMSSTDx.SQL in TimesTen.
CURRENT_DATE function	Returns the current date in the session time zone.	In TimesTen, this returns the current date in UTC (universal time). TimesTen does not support local time zones.

**Table 8–1 (Cont.) PL/SQL language element and feature support in TimesTen**

Feature name	Description	Example/comment
Cursor attributes	Appended to the cursor or cursor variable to return useful information about the execution of a data manipulation statement.	Explicit cursors and cursor variables have four attributes: %FOUND, %ISOPEN, %NOTFOUND, and %ROWCOUNT.  The implicit cursor (SQL) has additional attributes: %BULK_ROWCOUNT and %BULK_EXCEPTIONS.  See "Using the %ROWCOUNT and %NOTFOUND attributes" on page 5-8 and "Using FORALL with SQL%BULK_ROWCOUNT" on page 5-9. Also see "Named Cursor Attribute" in <i>Oracle Database PL/SQL Language Reference</i> .
Cursor declaration	Declares a cursor. To execute a multi-row query, TimesTen opens an unnamed work area that stores processing information. A cursor lets you name the work area, access the information, and process the rows individually.	See "Use of cursors in PL/SQL programs" on page 2-28.
Cursor variables (REF CURSORS)	Act as handles to cursors over SQL result sets.	TimesTen supports OUT REF CURSORS, one per statement.  See "PL/SQL REF CURSORS" on page 3-7.
DELETE statement	Deletes rows from a table.	See "DELETE" in <i>Oracle TimesTen In-Memory Database SQL Reference</i> .
DROP { PROCEDURE   FUNCTION   PACKAGE } statement	Removes a PL/SQL procedure, function, or package, as specified.	Syntax and semantics are the same as in Oracle Database.  You can refer to information about these statements in "SQL Statements" in <i>Oracle TimesTen In-Memory Database SQL Reference</i> .
Error reporting	(This is self-explanatory.)	TimesTen applications report errors using Oracle Database error codes instead of TimesTen error codes. The error messages that accompany the error codes are either TimesTen error messages or Oracle Database error messages.
EXCEPTION_INIT pragma	Associates a user-defined exception with a TimesTen error number.	See "EXCEPTION_INIT Pragma" in <i>Oracle Database PL/SQL Language Reference</i> .
Exception definition	Specifies an exception, which is a runtime error or warning condition. Can be predefined or user-defined.	Predefined conditions are raised implicitly. User-defined exceptions are raised explicitly by the RAISE statement. To handle raised exceptions, write separate routines called <i>exception handlers</i> .  See Chapter 4, "Errors and Exception Handling".

**Table 8–1 (Cont.) PL/SQL language element and feature support in TimesTen**

Feature name	Description	Example/comment
EXECUTE IMMEDIATE statement	Builds and executes a dynamic SQL statement.	TimesTen supports this as Oracle Database does to execute a SQL DML or DDL statement, execute a PL/SQL anonymous block, or call a PL/SQL stored procedure or function. See <a href="#">"Dynamic SQL in PL/SQL (EXECUTE IMMEDIATE statement)"</a> on page 2-20.  In TimesTen, the EXECUTE IMMEDIATE statement can also be used to execute TimesTen built-in procedures and TimesTen-specific SQL features (such as SELECT FIRST).
EXIT statement	Exits a loop and transfers control to the end of the loop.	See <a href="#">Example 6–2, "Use ALTER SESSION to change attribute settings"</a> on page 6-6 (among others).
Expression definition	Specifies an expression, which is a combination of operands (variables, constants, literals, operators, and so on) and operators. The simplest expression is a single variable.	See "Expressions" in <i>Oracle Database PL/SQL Language Reference</i> .
FETCH statement	Retrieves rows of data from the result set of a multi-row query.	See <a href="#">Example 2–17, "Using a cursor to retrieve information about an employee"</a> on page 2-28 (among others).
FORALL statement	Bulk-binds input collections before sending them to the SQL engine.	See <a href="#">"FORALL and BULK COLLECT operations"</a> on page 2-22.
Function declaration and definition	Specifies a subprogram or stored program that can be declared and defined in a PL/SQL block or package and returns a single value.	In TimesTen, a stored function or procedure can be executed in an anonymous block or through a CALL statement, but not from any other SQL statement. See <a href="#">"Executing procedures and functions"</a> on page 2-9.  In TimesTen Classic, use the CREATE FUNCTION statement in TimesTen SQL to create stored functions. See <a href="#">"PL/SQL procedures and functions"</a> on page 2-8. Also see "CREATE FUNCTION" in <i>Oracle TimesTen In-Memory Database SQL Reference</i> .  Also refer to the table entry below for "Procedure declaration and definition".
GOTO statement	Branches unconditionally to a statement label or block label.	See "GOTO Statement" in <i>Oracle Database PL/SQL Language Reference</i> .
IF statement	Executes or skips a sequence of statements depending on the value of the associated boolean expression.	See <a href="#">"Conditional control"</a> on page 2-6.
INLINE pragma	Specifies whether a subprogram call is to be inline.	See "INLINE Pragma" in <i>Oracle Database PL/SQL Language Reference</i> .
INSERT statement	Inserts one or more rows of data into a table.	See <a href="#">"Example using the INSERT statement"</a> on page 5-2. Also see "INSERT" in <i>Oracle TimesTen In-Memory Database SQL Reference</i> .
Literal declaration	Specifies a numeric, character string, or boolean value.	Examples: Numeric literal: 135 String literal: 'TimesTen'

**Table 8–1 (Cont.) PL/SQL language element and feature support in TimesTen**

Feature name	Description	Example/comment
LOOP statement	Executes a sequence of statements multiple times. Can be used, for example, in implementing a FOR loop or WHILE loop.	See <a href="#">Example 2–8, "Using a WHILE loop"</a> on page 2-7. Also see "Basic LOOP Statement" in <i>Oracle Database PL/SQL Language Reference</i> .
MERGE statement	Allows you to select rows from one or more sources for update or insertion into a target table.	See "MERGE" in <i>Oracle TimesTen In-Memory Database SQL Reference</i> .
Native dynamic SQL execution	Processes most dynamic SQL statements through the EXECUTE IMMEDIATE statement.	See the EXECUTE IMMEDIATE entry above.
NULL statement	This is a no-operation statement. Control is passed to the next statement without any action.	See "NULL Statement" in <i>Oracle Database PL/SQL Language Reference</i> .
OPEN statement	Executes the query associated with a cursor. Allocates database resources to process the query, and identifies the result set.	See <a href="#">Example 2–17, "Using a cursor to retrieve information about an employee"</a> on page 2-28.
OPEN FOR statement	Executes the SELECT statement associated with a cursor variable (REF CURSOR). Positions the cursor variable before the first row in the result set.	See "OPEN FOR Statement" in <i>Oracle Database PL/SQL Language Reference</i> .
Package declaration	Specifies a package, which is a database object that groups logically related PL/SQL types, items, and subprograms.	In TimesTen Classic, use SQL statements CREATE PACKAGE and CREATE PACKAGE BODY. See " <a href="#">PL/SQL packages</a> " on page 2-12. Also see "SQL Statements" in <i>Oracle TimesTen In-Memory Database SQL Reference</i> for information about the CREATE statements.
Procedure declaration and definition	Specifies a subprogram or stored program that can be declared and defined in a PL/SQL block or package and performs a specific action.	In TimesTen, a stored procedure or function can be executed in an anonymous block or through a CALL statement, but not from any other SQL statement. See " <a href="#">Executing procedures and functions</a> " on page 2-9.  In TimesTen Classic, use the CREATE PROCEDURE statement in TimesTen SQL to create stored procedures. See " <a href="#">PL/SQL procedures and functions</a> " on page 2-8. Also see "CREATE PROCEDURE" in <i>Oracle TimesTen In-Memory Database SQL Reference</i> .  Also refer to the table entry above for "Function declaration and definition".
RAISE statement	Stops normal execution of a PL/SQL block or subprogram and transfers control to an exception handler.	See " <a href="#">Using the RAISE statement</a> " on page 4-4.
Record definition	Defines a record, which is a composite variable that stores data values of different types (similar to a database row).	See " <a href="#">Using records in PL/SQL</a> " on page 3-5.
RETURN statement	Immediately completes the execution of a subprogram and returns control to the invoker. Execution resumes with the statement following the subprogram call.	See "RETURN Statement" in <i>Oracle Database PL/SQL Language Reference</i> .

**Table 8–1 (Cont.) PL/SQL language element and feature support in TimesTen**

<b>Feature name</b>	<b>Description</b>	<b>Example/comment</b>
RETURNING INTO clause	Specifies the variables in which to store the values returned by the statement to which the clause belongs.	See <a href="#">"RETURNING INTO clause"</a> on page 2-23 and <a href="#">"Examples using RETURNING INTO"</a> on page 5-16.
ROLLBACK statement	Undoes database changes made during the current transaction.	See <a href="#">"ROLLBACK"</a> in <i>Oracle TimesTen In-Memory Database SQL Reference</i> . <b>Important:</b> COMMIT and ROLLBACK statements close all cursors in TimesTen.
%ROWTYPE attribute	Provides a record type that represents a row in a database table.	See <a href="#">Example 2–2, "Assigning values to variables with the assignment operator"</a> on page 2-3.
SELECT INTO statement	Retrieves values from one row of a table (SELECT) and then stores the values in either variables or a record. With the BULK COLLECT clause (discussed above), this statement retrieves an entire result set.	See <a href="#">Example 2–3, "Using SELECT INTO to assign values to variables"</a> on page 2-4. Also see <a href="#">"Processing Query Result Sets"</a> in <i>Oracle Database PL/SQL Language Reference</i> .
SOUNDEX SQL function	Returns a character string containing the phonetic representation of a char.	See <a href="#">"SOUNDEX"</a> in <i>Oracle TimesTen In-Memory Database SQL Reference</i> .
SQL cursor	Either explicit or implicit, handles the result set of a SELECT statement.	See <a href="#">"Use of cursors in PL/SQL programs"</a> on page 2-28.
SQLCODE function	Returns number code of the most recent exception.	Given the same error condition, error codes returned by the built-in function SQLCODE are the same in TimesTen as in Oracle Database, although the SQLERRM returns may be different.  This is also noted in <a href="#">"TimesTen error messages and SQL codes"</a> on page 4-9.
SQLERRM function	Returns the error message associated with the error-number argument.	Given the same error condition, error messages returned by the built-in function SQLERRM are not necessarily the same in TimesTen as in Oracle Database, although SQLCODE returns are the same.  This is also noted in <a href="#">"TimesTen error messages and SQL codes"</a> on page 4-9.
Supplied packages	These are PL/SQL packages supplied with the database.	TimesTen provides a subset of the Oracle Database PL/SQL supplied packages.  See <a href="#">Chapter 7, "TimesTen Supplied PL/SQL Packages"</a> .
System tables and views	These are tables and views provided with the database for administrative purposes.	TimesTen supports a subset of the Oracle Database system tables and views.  See <a href="#">"System Tables and Views"</a> in <i>Oracle TimesTen In-Memory Database System Tables and Views Reference</i> .

---

**Table 8–1 (Cont.) PL/SQL language element and feature support in TimesTen**

<b>Feature name</b>	<b>Description</b>	<b>Example/comment</b>
ttPLSQLMemoryStats built-in procedure	Returns statistics about library cache performance and activity.	See "ttPLSQLMemoryStats" in <i>Oracle TimesTen In-Memory Database Reference</i> .  <b>Note:</b> In Oracle Database, the V\$LIBRARYCACHE system view provides the same statistical information.
%TYPE attribute	Lets you use the data type of a field, record, nested table, database column, or variable in your own declarations, rather than hardcoding the data type. Particularly useful when declaring variables, fields, and parameters that refer to database columns.	See " <a href="#">PL/SQL variables and constants</a> " on page 2-2.
UPDATE statement	Updates the values of one or more columns in all rows of a table or in rows that satisfy a search condition.	See "UPDATE" in <i>Oracle TimesTen In-Memory Database SQL Reference</i> .

## Symbols

---

%BULK\_EXCEPTIONS attribute, 2-22, 5-12  
%BULK\_ROWCOUNT attribute, 5-9  
%FOUND attribute, 8-4  
%ISOPEN attribute, 8-4  
%NOTFOUND attribute, 5-7, 8-4  
%ROWCOUNT attribute, 5-8, 8-4  
%ROWTYPE attribute, 2-4  
%TYPE attribute, 2-2

## A

---

ALTER SESSION, 6-6  
anonymous blocks, 2-1  
array binding  
    associative array binding, 3-6  
    bulk binding, 2-22  
associative arrays (index-by tables)  
    binding, 3-6  
    example, 5-5  
    using, 3-5  
audiences for this document, 1-4

## B

---

BINARY\_INTEGER type, 3-3  
bind parameter--see parameters  
bind variables--see parameters  
blocks, 2-1  
built-in functions (TimesTen)  
    calling via EXECUTE IMMEDIATE, 2-20, 5-16  
bulk binding  
    %BULK\_EXCEPTIONS attribute, 2-22, 5-12  
    BULK COLLECT INTO with cursors, 5-11  
    BULK COLLECT INTO with queries, 5-10  
    BULK COLLECT INTO with RETURNING  
        INTO, 5-17  
    examples, 5-9  
    FORALL statement, 2-22  
    overview, 2-22  
    SAVE EXCEPTIONS clause, 2-22, 5-12

## C

---

cache features, use from PL/SQL, 2-23  
CALL statement, calling functions and

    procedures, 2-9  
CASE statement, 2-6  
collections, 3-5  
compilation  
    conditional compilation, use of PLSQL\_  
        CCFLAGS, 6-2  
    conditional, use of UTL\_IDENT and TT\_DB\_  
        VERSION, 7-16  
    DBMS\_UTILITY.COMPILE\_SCHEMA, 7-10  
    DBMS\_UTILITY.INVALIDATE, optionally modify  
        compiler parameter settings, 7-10  
    PLSCOPE\_SETTINGS for compilation  
        cross-reference information, 6-1  
    PLSQL\_OPTIMIZE\_LEVEL for optimization  
        level, 6-3  
    UTL\_RECOMP package to recompile invalid  
        modules, 7-19  
components of PL/SQL, overview, 1-2  
composite data types, 3-4  
conditional control, 2-6  
connection attributes  
    first connection attributes, 6-1  
    general connection attributes, 6-1  
    PLSCOPE\_SETTINGS attribute, 6-1  
    PLSQL\_CCFLAGS attribute, 6-2  
    PLSQL\_CONN\_MEM\_LIMIT attribute, 6-2  
    PLSQL\_MEMORY\_ADDRESS attribute, 6-3  
    PLSQL\_MEMORY\_SIZE attribute, 6-3  
    PLSQL\_OPEN\_CURSORS attribute, 6-3  
    PLSQL\_OPTIMIZE\_LEVEL attribute, 6-3  
    PLSQL\_SESSION\_CACHED\_CURSORS  
        attribute, 6-4  
    PLSQL\_TIMEOUT attribute, 6-4  
constants and variables, 2-2  
CONTINUE statement, 2-7  
conversion--see type conversion  
cursors  
    closed at end of transaction, 2-31  
    cursor attributes, 8-4  
    cursor FOR loop, example, 5-8  
    examples, 5-6  
    REF CURSORS, 3-7  
    use in PL/SQL, 2-28

## D

---

### data types

- associative arrays (index-by tables), 3-5
  - categories, 3-2
  - collections, 3-5
  - composite data types, 3-4
  - conversion between application types and PL/SQL or SQL types, 3-9
  - conversion between PL/SQL and SQL, 3-11
  - conversion between PL/SQL types, 3-9
  - differences in TimesTen, 3-10
  - index-by tables (associative arrays), 3-5
  - nested tables, 3-5
  - non-supported types, 3-13
  - overview of what is supported, 3-1
  - PLS\_INTEGER and BINARY\_INTEGER, 3-3
  - PL/SQL types, 3-2
  - records, 3-5
  - REF CURSORS, 3-7
  - ROWID, 3-4
  - scalar types, 3-2
  - SIMPLE\_INTEGER, 3-4
  - type environments, 3-1
  - varrays, 3-5
- DBMS\_LOB package, 7-2
- DBMS\_LOCK package, 7-4
- DBMS\_OUTPUT package, 7-5
- DBMS\_PREPROCESSOR package, 7-6
- DBMS\_RANDOM package, 7-7
- DBMS\_SQL package, 7-8
- DBMS\_UTILITY package, 7-10
- DDL statements, 2-20
- demos--see sample applications
- differences in TimesTen
- data type considerations, 3-10
  - exception handling and behavior, 4-7
  - execution of PL/SQL from SQL, 1-4
  - PL/SQL language element and feature support, 8-1
  - SQL statements in PL/SQL blocks, 1-3
  - transaction behavior, 2-31
- DML returning, 2-23, 5-16
- DML statements, 2-19
- duplicate parameters, 2-18
- dynamic SQL
- DBMS\_SQL package, 7-8
  - EXECUTE IMMEDIATE examples, 5-13
  - EXECUTE IMMEDIATE usage, 2-20

## E

---

### errors

- error messages, differences vs. Oracle, 4-9
- exception types, 4-2
- RAISE statement, 4-4
- RAISE\_APPLICATION\_ERROR procedure, 4-5
- show errors in ttiSql, 4-6
- SQLCODE built-in function, 4-4, 4-9
- SQLERRM built-in function, 4-4, 4-9
- transaction and rollback behavior, differences vs.

- Oracle, 4-8
- transient (retry), 4-6
- trapping predefined exceptions, 4-3
- trapping user-defined exceptions, 4-4
- understanding exceptions, 4-1
- warnings (not supported), 4-9

### examples

- bind variables, 5-3
  - bulk binding, 5-9
  - cursor FOR loop, 5-8
  - cursors, 5-6
  - dynamic SQL, 5-13
  - FETCH statement, 5-6
  - INSERT statement, 5-2
  - query a system view, 5-18
  - RETURNING INTO, 5-16
  - SELECT statement, 5-1
- exceptions--see errors
- EXECUTE IMMEDIATE statement
- examples, 5-13
  - usage, 2-20

## F

---

- features, overview, 1-1
- FETCH statement, example, 5-6
- first connection attributes, 6-1
- FOR loop, 2-6
- FORALL statement, 2-22, 5-9
- functions
- basic usage and example, 2-8
  - SQL functions, from PL/SQL, 2-5
  - supported ways to execute, 2-9

## G

---

- general connection attributes, 6-1

## I

---

- IF-THEN-ELSE statement, 2-6
- IN OUT parameters, 2-18
- IN parameters, 2-18
- index-by tables (associative arrays)
- binding, 3-6
  - using, 3-5
- INSERT statement, example, 5-2
- integer types
- BINARY\_INTEGER, 3-3
  - PLS\_INTEGER, 3-3
  - SIMPLE\_INTEGER, 3-4
- iterative control, 2-6

## L

---

- language elements and features, support, 8-1
- LOBs
- DBMS\_LOB package, 7-2
  - overview, 2-25
  - using in PL/SQL, 2-25

## N

---

nested tables, 3-5  
NLS\_DATE\_FORMAT, 3-12  
NLS\_TIMESTAMP\_FORMAT, 3-12

## O

---

operations of PL/SQL, overview, 1-2  
OUT parameters, 2-18  
overview  
    components and operations, 1-2  
    features, 1-1

## P

---

packages  
    concepts, 2-13  
    creating and using, 2-13  
    TimesTen-supplied packages, 7-1  
parameters  
    binding, 2-17  
    duplicate parameters, 2-18  
    examples using bind variables, 5-3  
    IN, 2-18  
    IN OUT, 2-18  
    OUT, 2-18  
PLS\_INTEGER type, 3-3  
PLSCOPE\_SETTINGS connection attribute, 6-1  
PLSQL\_CCFLAGS connection attribute, 6-2  
PLSQL\_CONN\_MEM\_LIMIT connection attribute, 6-2  
PLSQL\_MEMORY\_ADDRESS connection attribute, 6-3  
PLSQL\_MEMORY\_SIZE connection attribute, 6-3  
PLSQL\_OPEN\_CURSORS connection attribute, 6-3  
PLSQL\_OPTIMIZE\_LEVEL connection attribute, 6-3  
PLSQL\_SESSION\_CACHED\_CURSORS connection attribute, 6-4  
PLSQL\_TIMEOUT connection attribute, 6-4  
predefined exceptions  
    not supported by TimesTen, 4-10  
    supported by TimesTen, 4-3  
procedures  
    basic usage and example, 2-8  
    supported ways to execute, 2-9  
programming features  
    conditional control, 2-6  
    continue, 2-7  
    iterative control, 2-6  
public objects, 2-13

## Q

---

queries, 2-19  
Quick Start sample applications, 1-5

## R

---

RAISE statement (exceptions), 4-4  
RAISE\_APPLICATION\_ERROR procedure, 4-5

records, 3-5  
REF CURSORS, 3-7  
replication, PL/SQL DDL, not supported, 2-9  
RETURNING INTO clause, 2-23, 5-16  
ROWID type, 3-4

## S

---

sample applications, 1-5  
samples--see examples  
SAVE EXCEPTIONS clause, 2-22, 5-12  
SELECT statement, 2-19  
SELECT statement, example, 5-1  
show errors, ttlsql, 4-6  
SIMPLE\_INTEGER type, 3-4  
sleep functionality, 7-4  
SQL  
    DDL statements, 2-20  
    dynamic SQL, 2-20  
    static SQL, 2-19  
SQL functions, from PL/SQL, 2-5  
SQLCODE built-in function, 4-4, 4-9  
SQLERRM built-in function, 4-4, 4-9  
standalone subprograms (procedures and functions), 2-8  
static SQL, 2-19  
stored functions  
    basic usage and example, 2-8  
    supported ways to execute, 2-9  
stored procedures  
    basic usage and example, 2-8  
    supported ways to execute, 2-9  
subprograms  
    basic usage and example, 2-8  
    supported ways to execute, 2-9  
supplied packages and system views  
    system view, querying, 5-18  
synonyms  
    for packages, 2-16  
    for procedures and functions, 2-11

## T

---

TimesTen Application-Tier Database Cache  
    (TimesTen Cache), use from PL/SQL, 2-23  
TimesTen built-in functions  
    calling via EXECUTE IMMEDIATE, 2-20, 5-16  
TO\_CHAR function, 3-12  
TO\_DATE function, 3-12  
transaction and rollback behavior, differences vs.  
    Oracle, 4-8  
transaction behavior, 2-31  
trapping exceptions  
    predefined exceptions, 4-3  
    user-defined exceptions, 4-4  
TT\_DB\_VERSION package, 7-12  
TT\_DECIMAL data type (unsupported), 3-12  
TT\_STATS package, 7-13  
ttlsql, show errors, 4-6  
ttPLSQLMemoryStats built-in procedure, 6-6, 8-8

ttSrcScan utility (check for unsupported features), 6-7

type conversion

between application types and PL/SQL or SQL types, 3-9

between PL/SQL and SQL, 3-11

between PL/SQL types, 3-9

differences in TimesTen, 3-10

## **U**

---

unsupported features, check with ttSrcScan, 6-7

UTL\_FILE package, 7-14

UTL\_IDENT package, 7-16

UTL\_RAW package, 7-17

UTL\_RECOMP package, 7-19

## **V**

---

variables and constants, 2-2

varrays, 3-5

## **W**

---

warnings (not supported), 4-9

WHILE loop, 2-6

wrapping PL/SQL source code, 2-29