Oracle® TimesTen In-Memory Database

PL/SQL Developer's Guide

Release 11.2.1

E13076-02

August 2009

Oracle TimesTen In-Memory Database PL/SQL Developer's Guide, Release 11.2.1

E13076-02

Contents

Preface

Oracle TimesTen In-Memory Database is a memory-optimized relational database. Deployed in the application tier, TimesTen operates on databases that fit entirely in physical memory using standard SQL interfaces. High availability for the in-memory database is provided through real-time transactional replication.

TimesTen supports a variety of programming interfaces, including ODBC (Open Database Connectivity), OCI (Oracle Call Interface), Oracle Pro*C/C++ (precompiler for embedded SQL and PL/SQL instructions in C or C++ code), JDBC (Java Database Connectivity), and PL/SQL (Oracle procedural language extension for SQL).

This preface covers the following topics:

- Audience
- Related documents
- Conventions
- Documentation Accessibility
- Technical support

Audience

This document is intended for anyone developing or supporting applications that use PL/SQL with TimesTen. Although it provides some overview, you should already be familiar with PL/SQL or have access to more thorough 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 any relevant additional TimesTen developer documentation.

Also see "Audiences for this document" on page 1-3, which goes into more detail.

Related documents

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

`http://www.oracle.com/technology/documentation/timesten_doc.html`

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

`http://www.oracle.com/technology/documentation/database.html`

In particular, these Oracle 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 refers to Windows 2000, Windows XP and Windows Server 2003. The term UNIX refers to Solaris, Linux, HP-UX, and AIX.

This document uses the following text conventions:

Convention	Meaning
italic	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.
`italic monospace`	Italic monospace type indicates a placeholder or a variable in a code example for which you specify or use a particular value. For example: `Driver=`*`install_dir`*`/lib/libtten.sl` Replace *`install_dir`* with the path of your TimesTen installation directory.
[ ]	Square brackets indicate that an item in a command line is optional.
{ }	Curly braces indicated that you must choose one of the items separated by a vertical bar (|) in a command line.
|	A vertical bar (or pipe) separates alternative arguments.
. . .	An ellipsis (. . .) after an argument indicates that you may use more than one argument on a single command line. An ellipsis in a code example indicates that what is shown is only a partial example.
%	The percent sign indicates the UNIX shell prompt.

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

Convention	Meaning
`install_dir`	The path that represents the directory where TimesTen is installed.

Convention	Meaning
`TTinstance`	The instance name for your specific installation of TimesTen. Each installation of TimesTen must be identified at installation time with a unique alphanumeric instance name. This name appears in the installation path.
`bits` or *`bb`*	Two digits, either 32 or 64, that represent either a 32-bit or 64-bit operating system.
`release` or *`rr`*	Numbers that represent a major TimesTen release, with or without dots. For example, 1121 or 11.2.1 represents TimesTen Release 11.2.1.
`DSN`	TimesTen data source name.

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible to all users, including users that are disabled. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Accessibility standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For more information, visit the Oracle Accessibility Program Web site at `http://www.oracle.com/accessibility/`.

Accessibility of Code Examples in Documentation

Screen readers may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, some screen readers may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation

This documentation may contain links to Web sites of other companies or organizations that Oracle does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

Deaf/Hard of Hearing Access to Oracle Support Services

To reach Oracle Support Services, use a telecommunications relay service (TRS) to call Oracle Support at 1.800.223.1711. An Oracle Support Services engineer will handle technical issues and provide customer support according to the Oracle service request process. Information about TRS is available at `http://www.fcc.gov/cgb/consumerfacts/trs.html`, and a list of phone numbers is available at `http://www.fcc.gov/cgb/dro/trsphonebk.html`.

Technical support

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

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

1

Introduction to PL/SQL in the TimesTen Database

Oracle TimesTen In-Memory Database supports PL/SQL (Procedural Language Extension to SQL), a programming language that allows you to integrate procedural constructs with SQL in your TimesTen database. PL/SQL is an integral part of Oracle Database. As such, many of the PL/SQL features present in Oracle 11*g* (11.1.0.7) are also present in TimesTen. In addition, PL/SQL operates in essentially the same way in TimesTen as in Oracle.

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

- Features of PL/SQL in TimesTen
- Sources of documentation for PL/SQL (in this manual and elsewhere)
- Audiences for this document
- About the TimesTen PL/SQL demos

Features of PL/SQL in TimesTen

PL/SQL support in TimesTen allows you to:

- Take full advantage of the PL/SQL programming language.
- Execute PL/SQL from your client applications that use these APIs:
  - JDBC
  - OCI
  - ODBC
  - Pro*C/C++
  - TTClasses
- Execute TimesTen SQL from PL/SQL
- CREATE, ALTER, and 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 TimesTen database to customize your PL/SQL environment.

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

Note: TimesTen PL/SQL is not currently supported in a Windows environment.

Sources of documentation for PL/SQL

The Oracle Database and TimesTen In-Memory Database documentation libraries provide multiple sources of information to help you learn about features and enhancements of PL/SQL in TimesTen.

This guide, the *Oracle TimesTen In-Memory Database PL/SQL Developer's Guide*, is your main source of information for exploring the many features of PL/SQL in TimesTen. Use its examples to assist you, and refer to its descriptions of differences between PL/SQL in the TimesTen database and PL/SQL in Oracle Database.

Table 1–1 lists specific sources of documentation within this guide, as well as relevant information in other Oracle Database and TimesTen documents. Oracle documentation is available through the following URL:

`http://www.oracle.com/technology/documentation/database.html`

Table 1–1 Sources of documentation

Feature/Enhancement	See...
ALTER SESSION statement (for PL/SQL session parameters)	Chapter 7, "PL/SQL Installation and Environment" Also *Oracle TimesTen In-Memory Database SQL Reference*
Connection attributes	Chapter 7, "PL/SQL Installation and Environment" Also *Oracle TimesTen In-Memory Database Reference*
CREATE, ALTER, DROP statements for procedures, functions, packages, and package bodies	Chapter 6, "Examples Using Standalone Subprograms" Also *Oracle TimesTen In-Memory Database SQL Reference*, *Oracle Database SQL Language Reference*
Exception handling and error reporting	Chapter 4, "Errors and Exception Handling" Also *Oracle Database PL/SQL Language Reference*
Use of PL/SQL features from TimesTen supported client applications	*Oracle TimesTen In-Memory Database C Developer's Guide* *Oracle TimesTen In-Memory Database Java Developer's Guide* *Oracle TimesTen In-Memory Database TTClasses Guide*
Execution of TimesTen SQL from PL/SQL	"Use of SQL in PL/SQL programs" on page 2-11 Also *Oracle Database PL/SQL Language Reference*
PL/SQL programming language	All of this document Also *Oracle Database PL/SQL Language Reference*
System tables and views	*Oracle TimesTen In-Memory Database SQL Reference* *Oracle Database Reference*
TimesTen supplied packages	Chapter 9, "TimesTen Supplied PL/SQL Packages" Also *Oracle Database PL/SQL Packages and Types Reference*
TimesTen utilities	*Oracle TimesTen In-Memory Database Reference*

Audiences for this document

There are two primary developer audiences for this document:

- Developers experienced with Oracle Database and Oracle PL/SQL who want to learn how to use PL/SQL in TimesTen. In particular, they want to learn the differences between PL/SQL in Oracle 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 PL/SQL

Developers experienced with Oracle 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 and TimesTen, include the following. Note that TimesTen-specific considerations are discussed at the end of chapters 2, 3, and 4 and throughout chapter 10.

- "How to execute PL/SQL procedures and functions" on page 2-8. This includes a comparison between how you can execute them in TimesTen and in Oracle.
- "Differences in TimesTen: transaction behavior" on page 2-23. This discusses cursor behavior when a transaction ends in TimesTen.
- "Differences in TimesTen: data type considerations" on page 3-9. This includes TimesTen-specific conversions, and types that TimesTen does not support.
- "Differences in TimesTen: exception handing and error behavior" on page 4-6.
- Chapter 7, "PL/SQL Installation and Environment." This includes discussion of TimesTen connection attributes.
- Chapter 9, "TimesTen Supplied PL/SQL Packages." This documents the subset of Oracle PL/SQL packages that TimesTen supports.
- Chapter 10, "TimesTen PL/SQL Support: Reference Summary." This reference chapter provides a thorough and low-level treatment of differences between TimesTen PL/SQL and Oracle PL/SQL.

Developers experienced with TimesTen

Most of this document is geared toward 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 10, "TimesTen PL/SQL Support: Reference Summary," is geared toward differences between TimesTen PL/SQL and Oracle PL/SQL and may be of less interest.

About the TimesTen PL/SQL demos

After you have configured your environment, you can confirm that everything is set up correctly by compiling and running the TimesTen Quick Start demo applications. Refer to the Quick Start welcome page at `install_dir/quickstart.html`, especially the links under Sample Programs, for information about the following:

- Demo schema and setup

  The `build_sampledb` script creates a sample database and demo schema. You must run this before you start using the demos.

- Demo environment and setup

  The `ttquickstartenv` script, a superset of the `ttenv` script generally used for TimesTen setup, sets up the demo environment. You must run this each time you enter a session where you want to compile and run any of the demos.

- Demos and setup

  TimesTen provides demos for PL/SQL in a subdirectory under the `quickstart/sample_code` directory. For instructions on running the demos, see the README file in the subdirectory.

- What the demos do

  A synopsis of each demo is provided.

2 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.

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

Main features include:

- PL/SQL blocks
- PL/SQL variables and constants
- PL/SQL built-in functions and SQL functions
- PL/SQL control structures
- How to execute PL/SQL procedures and functions
- How to pass data between an application and PL/SQL
- Use of SQL in PL/SQL programs
- Use of cursors in PL/SQL programs
- PL/SQL procedures and functions
- PL/SQL packages
- 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 the `ttIsql` utility, 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. Oracle TimesTen In-Memory Database 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 Oracle In-Memory Database Cache: A PL/SQL block cannot be passed through to Oracle.

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 in the TimesTen database 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 (`:=`) (Example 2–2).
- By selecting or fetching values into it (Example 2–3).
- By passing the variable as an OUT or IN OUT parameter to a subprogram and then assigning the value inside the subprogram (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 9, "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; -- Assign 40 to hours_worked
       >   hourly_salary NUMBER := 22.50; -- Assign 22.50 to hourly_salary
       >   bonus NUMBER := 150; -- Assign 150 to bonus
       >   country VARCHAR2(128);
       >   counter NUMBER := 0; -- Assign 0 to counter
       >   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 := 'Theresa';
       >   emp_rec1.last_name := 'Bellchuck';
       >   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 2-16, which creates a subprogram that will remain stored in the user's schema.

PL/SQL built-in functions and SQL functions

The PL/SQL programming language supports built-in functions that you can use in your PL/SQL statements. These built-in functions are included in standard packages that are part of the PL/SQL language.

Oracle TimesTen In-Memory Database supports SQL functions that you invoke in SQL expressions. When using PL/SQL, use the SQL functions in your static and dynamic SQL statements.

In the two examples that follow, the function `RTRIM` is used in both a PL/SQL assignment statement and a static SQL statement, respectively. Even though the function has the same name (`RTRIM`), in the first example the PL/SQL built-in function is invoked, and in the second example the TimesTen SQL function is invoked.

Note: Do not confuse PL/SQL built-in functions with TimesTen built-in SQL functions, which are documented in "Built-In Procedures" in *Oracle TimesTen In-Memory Database Reference*.

Example 2–5 Using the PL/SQL RTRIM built-in function

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

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

PL/SQL procedure successfully completed.
```

Example 2–6 Using the SQL RTRIM function

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 Examplexxxyyyxyxy', 'xy')
       >      INTO tt_var FROM DUAL;
       >    DBMS_OUTPUT.PUT_LINE (tt_var);
       > END;
       > /
RTRIM Example
```

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

Table 2–1 summarizes the distinction between PL/SQL built-in functions and TimesTen SQL functions.

Table 2–1 Distinguishing PL/SQL and TimesTen SQL functions

Function type	Use in...	For more information, see...
PL/SQL built-in functions	PL/SQL statements	"Summary of PL/SQL Built-In Functions" in *Oracle Database PL/SQL Language Reference*
TimesTen SQL functions	Static and dynamic SQL statements	"Expressions" in *Oracle TimesTen In-Memory Database SQL Reference* for information on some SQL functions

PL/SQL control structures

Control structures are among the most important PL/SQL extensions to SQL. Oracle TimesTen In-Memory Database supports the same control structures as Oracle Database.

The types of control structures discussed here are:

- Conditional control
- Iterative control
- CONTINUE statement

Both the TimesTen In-Memory Database and the Oracle Database support the CONTINUE statement.

Conditional control

The IF-THEN-ELSE statement and CASE statement are examples of conditional control. In Example 2–7, the IF-THEN-ELSE statement is used to determine the salary raise of an employee based on the current salary. The CASE statement 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 statements

```
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 was added to the Oracle 11*g* release and is also supported by TimesTen. 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;
       >     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.
```

How to execute PL/SQL procedures and functions

TimesTen supports execution of PL/SQL from client applications using ODBC, OCI, Pro*C/C++, 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, also called subprograms, are PL/SQL blocks that have been defined with a specified name. (See "PL/SQL procedures and functions" on page 2-16 for how to define and create them.)

In TimesTen, a PL/SQL procedure or function must be executed in an anonymous block.

Consider the following function:

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

In TimesTen, you can execute `mytest` as follows:

```
Command> declare
       > n number;
       > begin
       > n := mytest;
       > end;
       > /

PL/SQL procedure successfully completed.
```

In Oracle, you could also execute `mytest` through a CALL statement or from a SQL statement, as follows. These execution mechanisms are not supported in TimesTen.

- In a CALL statement:

  ```
  SQL> variable n number;
  SQL> call mytest() into :n;

  Call completed.
  ```

- In a SELECT statement:

  ```
  SQL> select mytest from dual;

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

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 Arguments" 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` in calling 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 6-1 for the complete example.

See "PL/SQL procedures and functions" on page 2-16 for how to create and define procedures and functions.

See "Binding parameters" 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.

Note: For duplicate parameters, the implementation in PL/SQL in TimesTen is no different than the implementation in PL/SQL in Oracle Database.

IN, OUT, and IN OUT parameter modes

Parameter modes define whether parameters declared in a PL/SQL subprogram 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.

See "Examples using input and output parameters and bind variables" on page 6-1.

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 In-Memory Database Cache

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 "Dynamic SQL in PL/SQL (EXECUTE IMMEDIATE statement)" on page 2-12.
- See "Differences in TimesTen: transaction behavior" on page 2-23 for important information.

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

Example 2–10 below 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 2–10 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 for any of the following:

- Use a DML statement such as INSERT, UPDATE, or DELETE.
- Use a DDL statement such as CREATE or ALTER. (For example, you can use ALTER SESSION to change a PL/SQL connection attribute.)
- Call a TimesTen built-in procedure.

In particular, 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 for dynamic SQL is for DDL, which cannot be executed in static SQL from within PL/SQL.

Note: See "Differences in TimesTen: transaction behavior" on page 2-23 for important information.

Example 2–11 provides an example of EXECUTE IMMEDIATE. For additional examples, see "Examples using EXECUTE IMMEDIATE" on page 5-9.

Example 2–11 Using the EXECUTE IMMEDIATE statement to create a table

Consider a situation where you do not know your table definition at compilation. By using the EXECUTE IMMEDIATE statement, you can create your table at execution time. This example creates 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.

```
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 *)
```

FORALL and BULK COLLECT operations

Bulk binding is a powerful feature used in the execution of SQL statements from PL/SQL, used in moving large amounts of data between SQL and PL/SQL. (Do not confuse this with 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 a FETCH, INSERT, UPDATE, and DELETE operation multiple times. Oracle TimesTen In-Memory Database 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" on page 3-4.)

You can use the %BULK_EXCEPTIONS cursor attribute and the SAVE EXCEPTIONS clause with FORALL statements. SAVE EXCEPTIONS allows a statement to continue executing even when an insert or update statement 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 signal 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 and BULK COLLECT Together" in *Oracle Database PL/SQL Language Reference* for more information on these features.

Example 2–12 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 5-5.

Example 2–12   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 = 100;
< 24000 >
< 17000 >
< 6000 >
3 rows found.
```

Execute procedure and verify results:

```
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.
Command> ROLLBACK;
```

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) as well as 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" on page 3-4. For BULK COLLECT, see "FORALL and BULK COLLECT operations" on page 2-13 and "Examples using FORALL and BULK COLLECT" on page 5-5.

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 5-11.

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

TimesTen PL/SQL with In-Memory Database 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 an IMDB Cache environment, 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 the Oracle database.

In particular, the following points should be made 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 an IMDB 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 In-Memory Database Cache User's Guide* for details of the passthrough facility.)

  With `passthrough=1`, a statement can be passed through to Oracle if any accessed table does not exist in TimesTen. But in PL/SQL, 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, in the current release the TimesTen passthrough facility cannot be used to route PL/SQL blocks from TimesTen to Oracle Database. For example, when using IMDB Cache with `passthrough=3`, all statements executed on a TimesTen connection will be routed to Oracle Database. 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, not the block as a whole.)

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 more than one row 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. This also includes any DDL statement executed within PL/SQL, because the `DDLCommitBehavior` connection must be set to 0 if PL/SQL is enabled, resulting in autocommits of DDL statements.

Example 2–13 shows basic use of a cursor. See "Examples using cursors" on page 5-3 for additional information and examples. Also see "PL/SQL REF CURSORs" on page 3-5.

Example 2–13   Using a cursor to retrieve information about an employee

Declare a cursor `c1` to retrieve 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.
```

PL/SQL procedures and functions

Procedures and functions are PL/SQL blocks that have been defined with a specified name. They are also called subprograms.

Standalone subprograms are created at the database level with the CREATE PROCEDURE or CREATE FUNCTION statements.

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 if you want 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. 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: As is the case with a CREATE statement for any database object, CREATE statements for PL/SQL functions and procedures are not replicated.
- If you use Oracle In-Memory Database Cache: An Oracle-resident PL/SQL procedure or function cannot be called in TimesTen by passthrough. Procedures and functions must be defined in TimesTen to be executable in TimesTen.
- TimesTen does not support non-ASCII or quoted non-uppercase procedure names and function names.
- 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" on page 8-7.

An example of creating a procedure using OUT parameters follows. For additional examples, see Chapter 6, "Examples Using Standalone Subprograms".

Example 2–14 Create a procedure with OUT parameters

```
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.
```

PL/SQL packages

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

For information about PL/SQL packages provided with TimesTen, refer to Chapter 9, "TimesTen Supplied 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, as well as 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; however, while Oracle documentation mentions that you must run a script called `DBMSSTDX.SQL`, this does not apply to TimesTen.

Creating and using packages

To create packages and store them permanently in a TimesTen database, use 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 called *public* objects and can be referenced outside the package, as well as 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, called *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 if you want 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*.

Example 2–15 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 that count of 33 is correct

```
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.
```

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. The `wrap` utility processes an input SQL file and wraps only the PL/SQL units in the file, such as a package specification, package body, function, or procedure.

Consider the following example that creates a file called `wrap_test.sql`, the purpose of which is to create a procedure called `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.

```
$ cat 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;
/

$ wrap iname=wrap_test.sql

PL/SQL Wrapper: Release 11.1.0.7.0- Production on Thu Sep 11 23:27:04 2008

Copyright (c) 1993, 2009, 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
abcd
abcd
abcd
```

```
abcd
abcd
abcd
abcd
7
109 124
88/TJ0ycbC+uGVlIpcLGCFnYCg8wg+nwf/Ydf3QC2vjqNGMUKbgh9iAYckXK5QNfzYzt+o6D
LS+DZ5zkzuVb3jmo7cYSTwk8NxVuvSQPILBOxv6IcXb88echYysoGXS006xKqkF95sO5A7zY
Pko3h+4fFD7wC2PvQxnuyiVWceKJGUJ7wPUWFCHDet1ym181AY0rd7oXR3tVh4h5d3RhLzNM
xKpGTRsHj7Al9eLe4pAutkqgVVDBveT5RrLRnKoGp79VjbFXinShf9huGTE9mnPh2CJgUw==


/
$ ttIsql SampleDatabase

Copyright (c) 1996-2009, Oracle. All rights reserved.
Type ? or "help" for help, type "exit" to quit ttIsql.



connect "DSN=SampleDatabase";
Connection successful: ... PermSize=32;TypeMode=0;PLSQL_MEMORY_ADDRESS=20000000;
(Default setting AutoCommit=1)
Command> @wrap_test.plb

CREATE OR REPLACE PROCEDURE wraptest wrapped
a000000
1
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
7
109 124
88/TJ0ycbC+uGVlIpcLGCFnYCg8wg+nwf/Ydf3QC2vjqNGMUKbgh9iAYckXK5QNfzYzt+o6D
LS+DZ5zkzuVb3jmo7cYSTwk8NxVuvSQPILBOxv6IcXb88echYysoGXS006xKqkF95sO5A7zY
Pko3h+4fFD7wC2PvQxnuyiVWceKJGUJ7wPUWFCHDet1ym181AY0rd7oXR3tVh4h5d3RhLzNM
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
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
7
109 124
88/TJ0ycbC+uGVlIpcLGCFnYCg8wg+nwf/Ydf3QC2vjqNGMUKbgh9iAYckXK5QNfzYzt+o6D
LS+DZ5zkzuVb3jmo7cYSTwk8NxVuvSQPILBOxv6IcXb88echYysoGXS006xKqkF95sO5A7zY
Pko3h+4fFD7wC2PvQxnuyiVWceKJGUJ7wPUWFCHDet1ym181AY0rd7oXR3tVh4h5d3RhLzNM
xKpGTRsHj7Al9eLe4pAutkqgVVDBveT5RrLRnKoGp79VjbFXinShf9huGTE9mnPh2CJgUw==
 >
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 in PL/SQL. This is because when PL/SQL is enabled (`PLSQL` connection attribute is set to 1), the TimesTen `DDLCommitBehavior` connection attribute must be set to 0, for Oracle mode (autocommit DDL).

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. This 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;
/
```

3

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.

Oracle TimesTen In-Memory Database 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:

- 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 make use of rows, columns, and constants in the TimesTen database. These elements are expressed using TimesTen SQL data types.
- Application programs that interact with the TimesTen 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, float, 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*.

Topics 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 CURSOR.

Note: See "Non-supported data types" on page 3-11.

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 NVARCHAR2
BINARY	RAW

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

Data type family	Data type name
BOOLEAN	BOOLEAN **Note**: You cannot bind BOOLEAN types in SQL statements.
DATETIME	DATE TIMESTAMP
INTERVAL	INTERVAL YEAR TO MONTH INTERVAL DAY TO SECONDS
ROWID	ROWID

Note: See "Non-supported data types" on page 3-11.

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 will never be 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 TimesTen database 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 specification" in *Oracle TimesTen In-Memory Database SQL Reference* for additional information about rowids and the ROWID data type, including usage and life.

Note: See "Non-supported data types" on page 3-11.

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.

Oracle TimesTen In-Memory supports the composite data types:

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

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

Note: While TimesTen PL/SQL supports these types, it does not support passing them between PL/SQL and applications written in other languages.

Using collections

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.
```

Collections can be passed between PL/SQL subprograms as parameters, but cannot be returned to applications written in other languages.

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

Using records

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
       > -- NULL does nothing but allows unit to be compiled and tested
       >   NULL;
       > END;
       > /

PL/SQL procedure successfully completed.
```

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 from PL/SQL to your application. Oracle TimesTen In-Memory Database supports OUT REF CURSORs. REF CURSORs can also be passed from PL/SQL to PL/SQL.

This means you can pass REF CURSORs:

- From PL/SQL to PL/SQL: Pass REF CURSORs from one procedure or function to another in any mode (IN, OUT, or IN OUT).

- From PL/SQL to your client API (such as ODBC): Use OUT parameters to pass REF CURSORs from PL/SQL to your application.

In your applications, open the REF CURSOR within PL/SQL and pass the REF CURSOR back to your application so that your application can fetch the result set.

Oracle TimesTen In-Memory Database supports REF CURSORs in ODBC, JDBC, OCI, Pro*C, and TTClasses in either a direct-mode or client/server scenario. REF CURSORs are also discussed in the documentation for those programming interfaces.

Note: Oracle TimesTen In-Memory Database supports one OUT REF CURSOR per statement.

To define a REF CURSOR, perform the same steps as you would in Oracle. First define a REF CURSOR type and 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
       > ...
```

To declare cursor variables as the formal parameters of functions and procedures:

```
Command> DECLARE
       >   TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
       >   PROCEDURE open_emp_cv (emp_cv OUT EmpCurTyp) IS ...
```

Example 3–4 Fetch rows from result set of a dynamic multirow query

This example defines a REF CURSOR, `EmpCurType`, then declares a cursor variable, `emp_cv`, of the type `EmpCurType`. In the executable section of the PL/SQL block, the OPEN...FOR statement associates the cursor variable `emp_cv` with the multirow query, `sql_stmt`. The FETCH statement returns a row from the result set of a multirow query and assigns the values of the select list items to `emp_rec` in the INTO clause. When the last row is processed, the cursor variable is closed.

```
Command> DECLARE
       >   TYPE EmpCurTyp IS REF CURSOR;
       >   emp_cv   EmpCurTyp;
       >   emp_rec  employees%ROWTYPE;
       >   sql_stmt  VARCHAR2 (200);
       >   my_job VARCHAR2 (10) := 'ST_CLERK';
       > BEGIN
       >   sql_stmt := 'SELECT * FROM employees WHERE job_id = :j';
       >   OPEN  emp_cv FOR sql_stmt USING my_job;
       >   LOOP
       >     FETCH emp_cv INTO emp_rec;
       >     EXIT WHEN emp_cv%NOTFOUND;
       >     DBMS_OUTPUT.PUT_LINE (emp_rec.employee_id);
       >   END LOOP;
       >   CLOSE emp_cv;
       > END;
       > /
125
126
127
128
129
130
131
```

```
132
133
134
135
136
137
138
139
140
141
142
143
144

PL/SQL procedure successfully completed.
```

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 3-9.

Conversion between PL/SQL data types

Oracle TimesTen In-Memory Database 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 and 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 3-11.

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

Oracle TimesTen In-Memory Database 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 below shows a sampling of 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 "ODBC to SQL or PL/SQL type mappings" 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 used)
SQL_CHAR	CHAR (bound precision used)
SQL_DATE	DATE
SQL_DECIMAL	NUMBER
SQL_DOUBLE	NUMBER
SQL_FLOAT	BINARY_DOUBLE
SQL_INTEGER	PLS_INTEGER
SQL_NUMERIC	NUMBER
SQL_REAL	BINARY_FLOAT
SQL_REFCURSOR	REF CURSOR
SQL_TIMESTAMP	TIMESTAMP (bound scale used)
SQL_VARCHAR	VARCHAR2 (bound precision used)

Example 3–5 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. Oracle TimesTen In-Memory Database 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 3–6 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

Oracle TimesTen In-Memory Database 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. "Y" indicates that a conversion is possible between the two families. Supported conversions are bi-directional.

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

Type Family	NUMERIC	CHARACTER	BINARY	DATETIME	INTERVAL	ROWID
NUMERIC	Y	Y				
CHARACTER	Y	Y	Y	Y	Y	Y
DATETIME		Y		Y		
TIME		Y				
ROWID		Y				Y
BINARY		Y	Y			Y

Table 3–6 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	Unsigned integer 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	Signed 16-bit integer 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.

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

TimesTen data type	Description
TT_INTEGER	Signed integer in the range -2,147,483,648 to 2,147,483,647. Equivalent to PLS_INTEGER.
TT_BIGINT	Signed 8-byte integer in 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. TimesTen CHAR can hold up to 8300 bytes. TimesTen NCHAR can hold up to 4150 characters (8300 bytes). TimesTen VARCHAR2 can hold up to 4,194,304 bytes. TimesTen NVARCHAR2 can hold up to 2,097,152 characters (4,194,304 bytes).
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	TimesTen BINARY can hold up to 8300 bytes. TimesTen VARBINARY can hold up to 4,194,304 bytes. RAW and LONG RAW can hold up to 32,767 bytes.

Note: See "Non-supported data types" on page 3-11.

Example 3–7 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. In TimesTen:

- 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).

The SQL and PL/SQL TO_DATE and TO_CHAR functions can be used 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 large objects (LOBs), 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.
- TimesTen PL/SQL does not support the TimesTen type TT_DECIMAL.

4 Errors and Exception Handling

This chapter explores 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 "Handling PL/SQL Errors" 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
- Differences in TimesTen: exception handing 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 more than one row, 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.

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	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	Must be declared in the declarative section. 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 4-8.

Table 4–1 Predefined exceptions

Exception name	Oracle 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 WHEN clauses of a CASE statement were selected and there is no ELSE clause.
COLLECTION_IS_NULL	ORA-06531	-6531	Program attempted to apply collection methods other than EXISTS 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	A program attempted to open an already opened cursor.
DUP_VAL_ON_INDEX	ORA-00001	-1	A program attempted to insert duplicate values in a column that is constrained by a unique index.
INVALID_CURSOR	ORA-01001	-1001	Illegal cursor operation.
INVALID_NUMBER	ORA-01722	-1722	Conversion of character string to number failed.
NO_DATA_FOUND	ORA-01403	+100	Single row SELECT 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.
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).

Table 4–1 (Cont.) Predefined exceptions

Exception name	Oracle error number	SQLCODE	Description
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 value ROWID.
TOO_MANY_ROWS	ORA-01422	-1422	Single row SELECT returned more than one row.
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

PL/SQL in TimesTen allows you to define your own exceptions. 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, SQLCODE will return the same error code, but SQLERRM will not necessarily return the same error message. This is also noted in "TimesTen error messages and SQL codes" on page 4-7.

Using the RAISE_APPLICATION_ERROR procedure

Use the RAISE_APPLICATION_ERROR procedure in either the executable section or the exception section of your PL/SQL program or both. 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.
```

Differences in TimesTen: exception handing and error behavior

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

TimesTen PL/SQL transaction and rollback behavior for unhandled exceptions

TimesTen PL/SQL differs from Oracle 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 will roll back to the beginning of the anonymous block. TimesTen will 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 will fail because values must be unique, so there will be an exception and the program will perform a rollback. Running this in TimesTen results in the following.

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

PL/SQL procedure successfully completed.

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

The result is equivalent in Oracle, with the SELECT 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 will indicate 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, there will be a rollback to the beginning of the PL/SQL block, so the SELECT will 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 will be able to 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 will be rolled back.

TimesTen error messages and SQL codes

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

For further information:

- Example 4–2, "Using RAISE statement to trap user-defined exception" 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 (as well as Oracle-specific) information.

Warnings not visible in PL/SQL

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

TimesTen In-Memory Database does have the concept of warnings, but because the TimesTen PL/SQL implementation is based on the Oracle 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 will not see the warning.

Unsupported predefined errors

"Trapping predefined TimesTen errors" on page 4-2 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 error number	SQLCODE	Description
LOGIN_DENIED	ORA-01017	-1017	Invalid user name and password.
NOT_LOGGED_ON	ORA-01012	-1012	A program issued a database call without being connected to the database.
SELF_IS_NULL	ORA-30625	-30625	A program attempted to invoke a MEMBER method, but the instance of the object was not initialized.
TIMEOUT_ON_RESOURCE	ORA-00051	-51	A timeout occurred while the database is waiting for a resource.

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

The TimesTen PL/SQL implementation uses the Oracle SQL parser in compiling PL/SQL programs. As a result, if your program uses Oracle syntax or Oracle built-ins that are not supported by TimesTen, the issue will not be discovered during compilation. A runtime error would occur during program execution, however.

5

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 include:

Example using the SELECT 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 more than one row.

Example 5–1 Using SELECT... INTO to return sum of salaries

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

```
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.
```

Example using the INSERT statement

Oracle TimesTen In-Memory Database supports the TimesTen DML statements INSERT, UPDATE, DELETE, and MERGE. This section has an example of the INSERT statement.

Example 5–2 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 the 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 cursors

Oracle TimesTen In-Memory Database supports cursors, as discussed in "Use of cursors in PL/SQL programs" on page 2-15. 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

Fetching values

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

Example 5–3 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. Refer to "%NOTFOUND Attribute" in *Oracle Database PL/SQL Language Reference* for more information.

```
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 5–4 Fetching values into a record

This is similar to Example 5–3, 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 5–5 shows how to use the %ROWCOUNT cursor attribute as well as the %NOTFOUND cursor attribute (previously shown in Example 5–3 and Example 5–4).

Example 5–5   Using %ROWCOUNT and %NOTFOUND attributes

This example has the same results as Example 5–4, 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. Refer to "SQL%ROWCOUNT Attribute" in *Oracle Database PL/SQL Language Reference* for more information.

```
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 5–6 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 5–5.

```
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 5–7 Using a cursor FOR loop with subqueries

This example illustrates a FOR loop using subqueries. The results are the same as in Example 5–6.

```
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

Oracle TimesTen In-Memory Database supports bulk binding and the FORALL statement and BULK COLLECT feature, as noted in "FORALL and BULK COLLECT operations" on page 2-13.

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 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 5–8.

Example 5–8 Using the FORALL statement with SQL%BULKROWCOUNT

```
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 5–9 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 5–10 uses a cursor to bulk-collect rows from a table.

Example 5–10 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 5–9.

```
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.
```

Examples using EXECUTE IMMEDIATE

TimesTen supports the EXECUTE IMMEDIATE statement, as noted in "Dynamic SQL in PL/SQL (EXECUTE IMMEDIATE statement)" on page 2-12. This section provides additional examples to consider as you develop your PL/SQL applications in TimesTen, including how to use EXECUTE IMMEDIATE to alter a PL/SQL connection attribute or call a TimesTen built-in procedure.

Example 5–11 Using EXECUTE IMMEDIATE to alter PLSCOPE_SETTINGS

This example uses the 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. Refer to "ttConfiguration" in *Oracle TimesTen In-Memory Database Reference* for information about this procedure.

```
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, 32 >
< 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, 32 >
< PLSQL_OPTIMIZE_LEVEL, 3 >
< PLSQL_TIMEOUT, 30 >
...
54 rows found.
```

Example 5–12 Using the EXECUTE IMMEDIATE statement with a single row query

In this example, the function `get_emp` retrieves the employee record into variable `v_emprec`. Execute the function and return the results in `v_emprec`.

```
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.
```

Example 5–13 Using EXECUTE IMMEDIATE with TimesTen specific syntax

Use the EXECUTE IMMEDIATE statement to execute a TimesTen SELECT FIRST *n* statement. This syntax is specific to TimesTen.

```
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.
```

Example 5–14 Using EXECUTE IMMEDIATE to call ttConfiguration

Oracle TimesTen In-Memory Database supports a number of TimesTen specific built-in procedures, such as `ttConfiguration`. You can use the EXECUTE IMMEDIATE statement with CALL to call these built-in procedures.

For example, to call the built-in procedure `ttConfiguration`, create a PL/SQL record type and then SELECT INTO that record type. Because `ttConfiguration` returns more than one row, use BULK COLLECT.

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

```
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(1).name
       >    || ' Value: ' || v_ttConfigs(1).value);
       > end;
       > /
Name: CacheGridEnable Value: 0

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 2-14 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.

```
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.

```
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.
```

Examples using the AUTHID clause

This section runs a script twice with just one change, first defining a PL/SQL procedure with AUTHID CURRENT_USER for invoker's rights, then with AUTHID DEFINER for definer's rights. See "Definer's rights and invoker's rights" on page 8-7 for related information.

The script assumes three users have been created: a tool vendor and two tool users (`brandX` and `brandY`). Each has been granted CREATE SESSION, CREATE PROCEDURE, and CREATE TABLE privileges as necessary. The following setup is also assumed, to allow "`use` *`username`*`;`" syntax to connect to the database as *`username`*:

```
connect adding "uid=toolVendor;pwd=pw" as toolVendor;
connect adding "uid=brandX;pwd=pw" as brandX;
connect adding "uid=brandY;pwd=pw" as brandY;
```

The script does the following:

- Creates the procedure, `printInventoryStatistics`, as the tool vendor.
- Creates a table with the same name, `myInventory`, in each of the three user schemas, populating it with unique data in each case.

- Runs the procedure as each of the tool users.

The different results between the two executions of the script show the difference between invoker's rights and definer's rights.

Here is the script for the invoker's rights execution:

```
use toolVendor;
create table myInventory (name varchar2(100), inventoryCount tt_integer);
insert into myInventory values('butter', 1);

create or replace procedure printInventoryStatistics authid current_user is
 inventoryCount pls_integer;
begin
 select count(*) into inventoryCount from myInventory;
 dbms_output.put_line('Total items in inventory: ' || inventoryCount);
 for currentItem in (select * from myInventory) loop
   dbms_output.put_line(currentItem.name || ' ' || currentItem.inventoryCount);
 end loop;
end;
/
grant execute on printInventoryStatistics to brandX;
grant execute on printInventoryStatistics to brandY;

use brandX;
create table myInventory (name varchar2(100), inventoryCount tt_integer);
insert into myInventory values('toothpaste', 100);
set serveroutput on
execute toolVendor.printInventoryStatistics;

use brandY;
create table myInventory (name varchar2(100), inventoryCount tt_integer);
insert into myInventory values('shampoo', 10);
set serveroutput on
execute toolVendor.printInventoryStatistics;
```

The only difference for the definer's rights execution is the change in the AUTHID clause for the procedure definition:

```
...
create or replace procedure printInventoryStatistics authid definer is
 inventoryCount pls_integer;
begin
 select count(*) into inventoryCount from myInventory;
 dbms_output.put_line('Total items in inventory: ' || inventoryCount);
 for currentItem in (select * from myInventory) loop
   dbms_output.put_line(currentItem.name || ' ' || currentItem.inventoryCount);
 end loop;
end;
/
...
```

Example 5–15   Using AUTHID CURRENT_USER

Following are the results when the procedure is defined with invoker's rights. Note that when the tool users `brandX` and `brandY` run the `printInventoryStatistics` procedure, each sees the data in his own (the invoker's) `myInventory` table.

```
Command> run invoker.sql

use toolVendor;
create table myInventory (name varchar2(100), inventoryCount tt_integer);
```

```
insert into myInventory values('butter', 1);
1 row inserted.

create or replace procedure printInventoryStatistics authid current_user is
 inventoryCount pls_integer;
begin
 select count(*) into inventoryCount from myInventory;
 dbms_output.put_line('Total items in inventory: ' || inventoryCount);
 for currentItem in (select * from myInventory) loop
   dbms_output.put_line(currentItem.name || ' ' || currentItem.inventoryCount);
 end loop;
end;
/

Procedure created.

grant execute on printInventoryStatistics to brandX;
grant execute on printInventoryStatistics to brandY;

use brandX;
create table myInventory (name varchar2(100), inventoryCount tt_integer);
insert into myInventory values('toothpaste', 100);
1 row inserted.
set serveroutput on;

execute toolVendor.printInventoryStatistics;
Total items in inventory: 1
toothpaste 100

PL/SQL procedure successfully completed.

use brandY;
create table myInventory (name varchar2(100), inventoryCount tt_integer);
insert into myInventory values('shampoo', 10);
1 row inserted.
set serveroutput on;

execute toolVendor.printInventoryStatistics;
Total items in inventory: 1
shampoo 10

PL/SQL procedure successfully completed.
```

Use the following to terminate all the connections:

```
Command> disconnect all;
```

Example 5–16 Using AUTHID DEFINER

Following are the results when the procedure is defined with definer's rights. Note that when the tool users `brandX` and `brandY` run `printInventoryStatistics`, each sees the data in the `myInventory` table belonging to the tool vendor (the definer).

```
Command> run definer.sql

use toolVendor;
create table myInventory (name varchar2(100), inventoryCount tt_integer);
insert into myInventory values('butter', 1);
1 row inserted.
```

```
create or replace procedure printInventoryStatistics authid definer is
 inventoryCount pls_integer;
begin
 select count(*) into inventoryCount from myInventory;
 dbms_output.put_line('Total items in inventory: ' || inventoryCount);
 for currentItem in (select * from myInventory) loop
   dbms_output.put_line(currentItem.name || ' ' || currentItem.inventoryCount);
 end loop;
end;
/

Procedure created.

grant execute on printInventoryStatistics to brandX;
grant execute on printInventoryStatistics to brandY;

use brandX;
create table myInventory (name varchar2(100), inventoryCount tt_integer);
insert into myInventory values('toothpaste', 100);
1 row inserted.
set serveroutput on;

execute toolVendor.printInventoryStatistics;
Total items in inventory: 1
butter 1

PL/SQL procedure successfully completed.

use brandY;
create table myInventory (name varchar2(100), inventoryCount tt_integer);
insert into myInventory values('shampoo', 10);
1 row inserted.
set serveroutput on;

execute toolVendor.printInventoryStatistics;
Total items in inventory: 1
butter 1

PL/SQL procedure successfully completed.
```

In this case, it is also instructive to see that although `brandX` and `brandY` can each access the `toolVendor.myInventory` table through the procedure, they cannot access it directly. That is a key use of definer's rights, to allow specific and restricted access to a table or other SQL object through the actions of a procedure.

```
Command> use brandX;
brandx: Command> select * from toolVendor.myInventory;
15100: User BRANDX lacks privilege SELECT on TOOLVENDOR.MYINVENTORY
The command failed.

brandx: Command> use brandY;
brandy: Command> select * from toolVendor.myInventory;
15100: User BRANDY lacks privilege SELECT on TOOLVENDOR.MYINVENTORY
The command failed.
```

Use the following to terminate all the connections:

```
Command> disconnect all;
```

6

Examples Using Standalone Subprograms

This chapter shows how to create and execute standalone procedures and functions. Examples use input and output parameters and bind variables, executing procedures and functions from PL/SQL blocks. The chapter is organized as follows:

Examples using input and output parameters and bind variables

The examples in this section use IN parameters, OUT parameters, IN OUT parameters, and bind variables (host variables).

Example 6–1 Using IN and OUT parameters

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

Example 6–2 Using bind variables to execute a procedure

This example uses bind variables to execute procedure `query_emp` from Example 6–1. (You must create that procedure before completing this example.) Remember to check that data types are compatible.

```
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
```

Example 6–3 Using IN OUT parameters and bind variables

Consider a situation where you want to format a phone number. You decide to use an IN OUT parameter to pass the unformatted phone number to a procedure. After the procedure is executed, the IN OUT parameter contains the formatted phone number value. Procedure `FORMAT_PHONE` in this example accomplishes that, accepting a 10 character string containing digits for a phone number. Bind variable `b_phone_no` first provides the input value passed to `FORMAT_PHONE`, then after execution is used as an output value returning the updated string.

```
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
```

Example querying a system view

This section provides an example that queries a system view.

Example 6–4 Querying system view USER_SOURCE

This example queries the USER_SOURCE system view to examine the source code of procedure `query_emp` from Example 6–1. (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.

Example creating and invoking a standalone function

This section contains an example that executes a standalone function.

Example 6–5 Creating and invoking a standalone function

This example creates and invokes the function `get_sal`, which as one input parameter and returns `salary` as type NUMBER.

```
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> BEGIN
       >   DBMS_OUTPUT.PUT_LINE (get_sal (100));
       > END;
       > /
24000

PL/SQL procedure successfully completed.
```

7

PL/SQL Installation and Environment

Oracle TimesTen In-Memory Database seamlessly integrates PL/SQL with the database. You can conveniently manage PL/SQL in TimesTen and set connection attributes specific to your environment. In little time, you can start using PL/SQL in your TimesTen environment.

The chapter shows you how to manage PL/SQL in your TimesTen database, set connection attributes, and display system-provided packages. It also describes the `ttSrcScan` utility, which you can use to check for PL/SQL features unsupported in TimesTen. The chapter concludes with examples to assist you in your setup procedures.

Topics in this chapter include:

- Confirming that PL/SQL is Installed and Enabled in TimesTen
- PL/SQL connection attributes
- The ttSrcScan utility
- Displaying supplied packages

Confirming that PL/SQL is Installed and Enabled in TimesTen

This section covers the following topics:

- PL/SQL installation and the ttmodinstall utility
- Understanding the PLSQL connection attribute
- Checking that PL/SQL is enabled in a TimesTen database

PL/SQL installation and the ttmodinstall utility

Oracle TimesTen In-Memory Database installs PL/SQL in your instance by default. If you chose not to install PL/SQL during installation, you can use the TimesTen `ttmodinstall` utility to install it later. For more information, see "ttmodinstall" in *Oracle TimesTen In-Memory Database Reference*.

Note: Only the instance administrator can run this utility.

Understanding the PLSQL connection attribute

PL/SQL is enabled in TimesTen through the first connection attribute `PLSQL`. You can set this attribute when you initially create your database or at any first connection afterward. Note that once PL/SQL is enabled (`PLSQL=1`), it cannot be disabled (`PLSQL=0` would have no effect).

If PL/SQL is supported on your platform and enabled at installation time, TimesTen sets `PLSQL=1` by default. You can also set the `PLSQL` connection attribute in the `odbc.ini` file or in your application.

For more information on the `PLSQL` connection attribute, see "PLSQL" in *Oracle TimesTen In-Memory Database Reference*.

Checking that PL/SQL is enabled in a TimesTen database

There are several ways to check the status of PL/SQL in your database:

- Use the `ttVersion` utility to confirm that PL/SQL is installed in your instance. If PL/SQL is enabled, that is indicated as in the following example:

  ```
  $ ttVersion
  TimesTen Release 11.2.1.0.0 (32 bit Linux/x86) (user:4738) 2008-07-04T22:01:57Z
    Instance admin: user
    Instance home directory: /scratch/user...
    Daemon home directory: /scratch/user...
    PL/SQL enabled.
  ```

- Use the `ttStatus` utility to determine if PL/SQL is enabled in your database. In the following example, PL/SQL is enabled in database `plsql1` and is not enabled in database `plsql0`.

  ```
  $ ttstatus
  TimesTen status report as of Wed Jul 16 14:35:31 2008

  Daemon pid 00000 port 0000 instance user
  TimesTen server pid 00000 started on port 0000
  ------------------------------------------------------------------------
  Data store /scratch/user/plsql1
  There are no connections to the data store
  Replication policy  : Manual
  Cache Agent policy  : Manual
  PL/SQL enabled.
  ------------------------------------------------------------------------
  Data store /scratch/user/plsql0
  There are no connections to the data store
  Replication policy  : Manual
  Cache Agent policy  : Manual
  ------------------------------------------------------------------------
  ...
  ```

- Using the `ttIsql` utility, call the `ttConfiguration` built-in procedure to determine the `PLSQL` connection attribute setting for your database. Refer to "ttConfiguration" in *Oracle TimesTen In-Memory Database Reference* for information about this procedure.

  For example:

  ```
  Command> call ttConfiguration;
  < CacheGridEnable, 0 >
  < CacheGridMsgWait, 60 >
  ...
  < PLSQL, 1 >
  ...
  ```

PL/SQL connection attributes

There are several TimesTen connection attributes specific to PL/SQL, as summarized in Table 7–1. 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 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.

Table 7–1 PL/SQL Connection Attributes

Attribute	Summary
PLSQL	First connection attribute Required privilege: Instance administrator Enables PL/SQL in the database. If `PLSQL=1`, PL/SQL is enabled. If `PLSQL=0`, PL/SQL is not enabled. You can enable PL/SQL when your database is initially created or at any first connection. Once PL/SQL is enabled, it cannot be disabled. Default: 1 (for platforms where PL/SQL is supported)
PLSQL_MEMORY_ADDRESS	First connection attribute Required privilege: Instance administrator 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. If a single application simultaneously connects to more than one TimesTen database in direct mode, then you must set different values for each of the databases. Default: Platform-specific value. Refer to "PLSQL_MEMORY_ADDRESS" in *Oracle TimesTen In-Memory Database Reference* for platform-specific information.
PLSQL_MEMORY_SIZE	First connection attribute Required privilege: Instance administrator Determines the size, in megabytes, of memory allocated for the PL/SQL shared memory segment. This is memory used to hold recently executed PL/SQL code and metadata about PL/SQL objects, as opposed to storing runtime data such as database output. There is also related tuning information for this setting in "PLSQL_MEMORY_SIZE" in *Oracle TimesTen In-Memory Database Operations Guide*. Default: 32 megabytes **Important**: Use a value of at least 8 megabytes.

Table 7–1 (Cont.) PL/SQL Connection Attributes

Attribute	Summary
PLSCOPE_SETTINGS	General connection attribute Required privilege: None Controls whether the PL/SQL compiler generates cross-reference information. Possible values are `IDENTIFIERS:NONE` or `IDENTIFIERS:ALL`. You can use the ALTER SESSION statement to change this value within your session. Default: `IDENTIFIERS:NONE`

PLSQL_CCFLAGS	General connection attribute
	Required privilege: None
	This can be used to set inquiry directives to control conditional compilation of PL/SQL units, which allows 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. Possible uses include activating debugging or tracing features, or basing functionality on the version of the database. The following is an example:
	`PLSQL_CCFLAGS='DEBUG:TRUE, PRODUCTION:YES'`
	PL/SQL conditional compilation flags are similar in concept to flags on a C compiler command line, such as the following:
	`% cc -DEBUG=TRUE -DPRODUCTION=YES ...`
	You can use the ALTER SESSION statement to change `PLSQL_CCFLAGS` within your session.
	See "Conditional Compilation" in *Oracle Database PL/SQL Language Reference* for information about this feature. There is also an example of conditional compilation (though not involving `PLSQL_CCFLAGS`) in "UTL_IDENT" on page 9-12.
	Default: null

Table 7–1 (Cont.) PL/SQL Connection Attributes

Attribute	Summary

PLSQL_CONN_MEM_LIMIT	General connection attribute
	
	Required privilege: None
	
	Specifies the maximum amount of 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.
	
	The amount of space consumed by PL/SQL variables is roughly what you might expect comparable variables to consume in other programming languages. As an example, consider a large array of strings:
	
	`type chararr is table of varchar2(32767)`
	`           index by binary_integer;`
	`big_array chararr;`
	
	If 100,000 strings of 100 bytes each are placed into such an array, approximately 12 megabytes of memory is consumed.
	
	Memory consumed by variables in PL/SQL blocks is used while the block executes, then is released. Memory consumed by variables in PL/SQL package specifications or bodies (not within a procedure or function) is used for the lifetime of the package. Memory consumed by variables in a PL/SQL procedure or function, including one defined within a package, is used for the lifetime of the procedure or function. However, in all cases, memory freed by PL/SQL is not returned to the operating system. Instead, it is kept by PL/SQL and reused by future PL/SQL invocations. The memory is freed when the application disconnects from TimesTen.
	
	The `PLSQL_CONN_MEM_LIMIT` value is a number specified in megabytes. A setting of 0 means no limit.
	
	You can use the ALTER SESSION statement to change this value within your session.
	
	Default: 100 megabytes
	
	Note: In `ttPLSQLMemoryStats` output, the related value `CurrentConnectionMemory` indicates how much process heap memory PL/SQL has actually acquired through `malloc()`. (Also see Example 7–4, "View PL/SQL performance statistics".)

PLSQL_OPTIMIZE_LEVEL	General connection attribute Required privilege: None 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. You can use the ALTER SESSION statement to change this value within your session. Default: 2

Table 7–1 (Cont.) PL/SQL Connection Attributes

Attribute	Summary
PLSQL_TIMEOUT	General connection attribute Required privilege: None Controls how long PL/SQL procedures are allowed to run, in seconds, before being terminated. A new value impacts PL/SQL program units currently running. Possible values are 0 (meaning no time limit) or any positive integer. You can use the ALTER SESSION statement to change this value within your session. Default: 30 seconds **Note**: 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.

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

- If PL/SQL is enabled in your database, the value for `DDLCommitBehavior` must be 0. See "DDLCommitBehavior".
- If `LockLevel` 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 `MemoryLock`. See "MemoryLock".

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

Example 7–1 Create a database with PL/SQL default connection attributes

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

Sample `odbc.ini`:

```
[pldef]
Driver=path/libtten.so
Datastore=/scratch/user/pldef
DatabaseCharacterSet=US7ASCII
```

Connect to database `pldef`:

```
$ ttIsql pldef

Copyright (c) 1996-2009, Oracle. All rights reserved.
Type ? or "help" for help, type "exit" to quit ttIsql.
connect "DSN=pldef";
```

```
Connection successful: DSN=pldef;UID=user;DataStore=/scratch/user/pldef;Database
CharacterSet=US7ASCII;ConnectionCharacterSet=US7ASCII;DRIVER=path/libtten.so;
TypeMode=0;
(Default setting AutoCommit=1)
```

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

```
Command> call ttConfiguration;
...
< ConnectionCharacterSet, US7ASCII >
< ConnectionName, pldef >
...
< DataBaseCharacterSet, US7ASCII >
< DataStore, /scratch/user/pldef >
...
< 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, 32 >
< PLSQL_OPTIMIZE_LEVEL, 2 >
< PLSQL_TIMEOUT, 30 >
...
```

Example 7–2 Enable PL/SQL at first connection

This example establishes a first connection to a database that does not yet have PL/SQL enabled, specifying `PLSQL=1`. Because the connection is a first connection, TimesTen enables PL/SQL in the database as a result. The sample `odbc.ini` file also provides settings for `PLSQL_MEMORY_SIZE` and `PLSQL_MEMORY_ADDRESS`.

Sample `odbc.ini`:

```
[plsql0]
Driver=path/libtten.so
Datastore=/scratch/user/plsql0
DatabaseCharacterSet=US7ASCII
PLSQL=0
PLSQL_MEMORY_SIZE=40
PLSQL_MEMORY_ADDRESS=20000000
```

Connect to the `plsql0` database with `PLSQL=1`:

```
$ ttisql

Copyright (c) 1996-2009, Oracle. All rights reserved.
Type ? or "help" for help, type "exit" to quit ttIsql.

Command> connect "DSN=plsql0;PLSQL=1";
Connection successful: DSN=plsql0;UID=user;DataStore=/scratch/user/plsql0;
DatabaseCharacterSet=US7ASCII;ConnectionCharacterSet=US7ASCII;DRIVER=path/libtten.
so;TypeMode=0;PLSQL_MEMORY_SIZE=40;PLSQL_MEMORY_ADDRESS=20000000;
(Default setting AutoCommit=1)
```

Call the `ttConfiguration` built-in procedure to verify `PLSQL=1` as well as PL/SQL settings from `odbc.ini`:

```
Command> call ttConfiguration;
...
```

```
< ConnectionCharacterSet, US7ASCII >
< ConnectionName, plsql0 >
...
< DataBaseCharacterSet, US7ASCII >
< DataStore, /scratch/user/plsql0 >
...
< PLSCOPE_SETTINGS, IDENTIFIERS:NONE >
< PLSQL, 1 >
< PLSQL_CCFLAGS, <NULL> >
< PLSQL_CODE_TYPE, INTERPRETED >
< PLSQL_CONN_MEM_LIMIT, 100 >
< PLSQL_MEMORY_ADDRESS, 0x20000000 >
< PLSQL_MEMORY_SIZE, 40 >
< PLSQL_OPTIMIZE_LEVEL, 2 >
< PLSQL_TIMEOUT, 30 >
...
```

Example 7–3 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_OPTIMIZE_LEVEL`, and `PLSQL_CONN_MEM_LIMIT`. 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_OPTIMIZE_LEVEL=3;

Session altered.

Command> ALTER SESSION SET PLSQL_CONN_MEM_LIMIT=200;

Session altered.

Command> call ttconfiguration;
...
< PLSCOPE_SETTINGS, IDENTIFIERS:ALL >
...
< PLSQL_CONN_MEM_LIMIT, 200 >
...
< PLSQL_OPTIMIZE_LEVEL, 3 >
...
```

Next, the example sets the `PLSQL_TIMEOUT` connection attribute to 20 seconds. When there is an attempt to execute a program that loops indefinitely, sometime after 20 seconds has passed the execution is terminated and TimesTen returns an error.

```
Command> ALTER SESSION SET PLSQL_TIMEOUT = 20;

Session altered.

Command> DECLARE v_timeout NUMBER;
       > BEGIN
       >  LOOP
       >   v_timeout := 0;
       >   EXIT WHEN v_timeout < 0;
       >  END LOOP;
       > END;
       > /
```

```
 8509: PL/SQL execution terminated; PLSQL_TIMEOUT exceeded
The command failed.
```

Example 7–4 View PL/SQL performance statistics

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

```
Command> call ttplsqlmemorystats;
< Gets, 0.000000e+00 >
< GetHits, 0.000000e+00 >
< GetHitRatio, 1.000000 >
< Pins, 0.000000e+00 >
< PinHits, 0.000000e+00 >
< PinHitRatio, 1.000000 >
< Reloads, 0.000000e+00 >
< Invalidations, 0.000000e+00 >
< CurrentConnectionMemory, 0.000000e+00 >
9 rows found.
```

Note: `CurrentConnectionMemory` is related to the `PLSQL_CONN_MEM_LIMIT` connection attribute documented in "PL/SQL connection attributes" on page 7-3, indicating the amount of heap memory that has actually been acquired by PL/SQL.

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 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. You can find the `ttSrcScan` executable in the `quickstart/sample_util` directory in your TimesTen installation.

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. See the README file in the `sample_util` directory for information.

Displaying supplied packages

Oracle TimesTen In-Memory Database installs supplied packages when you enable PL/SQL in your database. To display a list of the packages provided with TimesTen, use the system view ALL_PROCEDURES for objects owned by SYS, as in the following example:

```
Command> select distinct object_name from all_procedures where owner='SYS';
< DBMS_LOCK >
< DBMS_OUTPUT >
< DBMS_PREPROCESSOR >
< DBMS_RANDOM >
< DBMS_SQL >
< DBMS_STANDARD >
< DBMS_SYS_ERROR >
< DBMS_UTILITY >
< PLITBLM >
< STANDARD >
< SYS_STUB_FOR_PURITY_ANALYSIS >
< UTL_FILE >
< UTL_RAW >
< UTL_RECOMP >
< TT_DB_VERSION >
< UTL_IDENT >
16 rows found.
```

For more information on supplied packages in TimesTen, see Chapter 9, "TimesTen Supplied PL/SQL Packages".

Note: As with other ALL_* system views, all users have SELECT privilege for the ALL_PROCEDURES system view.

8 Access Control for PL/SQL Programs

TimesTen has features to control database access with object-level resolution for database objects such as tables, views, materialized views, indexes, sequences, functions, procedures, and packages, for example. You can refer to "Managing Access Control" in *Oracle TimesTen In-Memory Database Operations Guide* for introductory information about TimesTen access control features.

This chapter introduces access control as it relates to PL/SQL users.

Note: Access control is automatically enabled when you install TimesTen. You cannot disable it.

Topics in this chapter include:

- Access control for PL/SQL operations
- Access control for SQL operations
- Definer's rights and invoker's rights
- Additional access control considerations

Access control for PL/SQL operations

This section covers the following topics:

- Required privileges for PL/SQL statements and operations
- Granting and revoking privileges
- Invalidated objects

Required privileges for PL/SQL statements and operations

For PL/SQL users, access control affects the ability to create, alter, drop, or execute PL/SQL procedures and functions, including packages and their member procedures and functions.

You need the CREATE PROCEDURE privilege to create a procedure, function, package definition, or package body if it is being created in your own schema, or CREATE ANY PROCEDURE if it is being created in any schema other than your own. To alter or drop a procedure, function, package definition, or package body, you must be the owner or have the ALTER ANY PROCEDURE privilege or DROP ANY PROCEDURE privilege, respectively.

To execute a procedure or function, you must be the owner, have the EXECUTE privilege for the procedure or function (or for the package to which it belongs, if applicable), or have the EXECUTE ANY PROCEDURE privilege. This is all summarized in Table 8–1.

Important: Be cautious in granting the CREATE ANY PROCEDURE and EXECUTE ANY PROCEDURE privileges. They can be misused, especially in combination with each other.

Table 8–1 Privileges for using PL/SQL procedures and functions

Action	SQL statement or operation	Required Privilege

Create a procedure, function, package definition, or package body.	CREATE [OR REPLACE] PROCEDURE
	CREATE [OR REPLACE] FUNCTION
	CREATE [OR REPLACE] PACKAGE
	CREATE [OR REPLACE] PACKAGE BODY
		CREATE PROCEDURE in user's schema
		Or:
		CREATE ANY PROCEDURE in any other schema

Alter a procedure, function, or package.	ALTER PROCEDURE ALTER FUNCTION ALTER PACKAGE	Ownership of the procedure, function, or package Or: ALTER ANY PROCEDURE
Drop a procedure, function, package definition, or package body.	DROP PROCEDURE DROP FUNCTION DROP PACKAGE DROP PACKAGE BODY	Ownership of the procedure, function, or package Or: DROP ANY PROCEDURE
Execute a procedure or function.	Invoke the procedure or function.	Ownership of the procedure or function, or of the package to which it belongs (if applicable) Or: EXECUTE for the procedure or function, or for the package to which it belongs (if applicable) Or: EXECUTE ANY PROCEDURE

See "SQL Statements" in *Oracle TimesTen In-Memory Database SQL Reference* for the syntax and required privileges of SQL statements discussed in this section.

Notes:

- A user who has been granted privilege to execute a procedure (or function) can execute the procedure even if he or she has no privilege on other procedures that the procedure calls. For example, consider a stored procedure `user2.proc1` that executes procedure `user2.proc2`. If `user1` is granted privilege to execute `proc1` but is not granted privilege to execute `proc2`, he could not run `proc2` directly but could still run `proc1`.
- Privilege to execute a procedure or function allows implicit compilation of the procedure or function if it is invalid or not compiled at the time of execution.
- When CREATE OR REPLACE results in an object (procedure, function, or package) being replaced, there is no effect on privileges that any users had previously been granted on that object. This is as opposed to when there is an explicit DROP and then CREATE to re-create an object, in which case all privileges on the object are revoked.

Granting and revoking privileges

Use the SQL statement GRANT to grant a privilege. Use REVOKE to revoke one.

The following example grants EXECUTE privilege to `user2` for a procedure and a package that `user1` owns:

```
Command> grant execute on user1.myproc to user2;
Command> grant execute on user1.mypkg to user2;
```

This example revokes the privileges:

```
Command> revoke execute on user1.myproc from user2;
Command> revoke execute on user1.mypkg from user2;
```

Example 8–1 Granting of required privileges

This example shows a series of attempted operations by a user, `user1`, as follows:

1. The user attempts each operation before having the necessary privilege. The resulting error is shown.
2. The instance administrator grants the necessary privilege.
3. The user successfully performs the operation.

The `ttIsql` utility is used by `user1` to perform (or attempt) the operations and by the instance administrator to grant privileges.

USER1:

Initially the user does not have permission to create a procedure. That must be granted even in one's own schema.

```
Command> create procedure testproc is
       > begin
       > dbms_output.put_line('user1.testproc called');
       > end;
       > /
15100: User USER1 lacks privilege CREATE PROCEDURE
The command failed.
```

INSTANCE ADMINISTRATOR:

```
Command> grant create procedure to user1;
```

USER1:

Once `user1` can create a procedure in his own schema, he can execute it because he owns it.

```
Command> create procedure testproc is
       > begin
       > dbms_output.put_line('user1.testproc called');
       > end;
       > /

Procedure created.

Command> begin
       > testproc();
       > end;
       > /
user1.testproc called

PL/SQL procedure successfully completed.
```

The user cannot yet create a procedure in another schema, though.

```
Command> create procedure user2.testproc is
       > begin
       > dbms_output.put_line('user2.testproc called');
       > end;
       > /
15100: User USER1 lacks privilege CREATE ANY PROCEDURE
The command failed.
```

INSTANCE ADMINISTRATOR:

```
Command> grant create any procedure to user1;
```

USER1:

Now `user1` can create a procedure in another schema, but he cannot execute it yet because he does not own it or have privilege.

```
Command> create procedure user2.testproc is
       > begin
       > dbms_output.put_line('user2.testproc called');
       > end;
       > /

Procedure created.

Command> begin
       > user2.testproc();
       > end;
       > /
 8503: ORA-06550: line 2, column 7:
PLS-00904: insufficient privilege to access object USER2.TESTPROC
 8503: ORA-06550: line 2, column 1:
PL/SQL: Statement ignored
The command failed.
```

INSTANCE ADMINISTRATOR:

```
Command> grant execute any procedure to user1;
```

USER1:

Now `user1` can execute a procedure in another schema.

```
Command> begin
       > user2.testproc();
       > end;
       > /
user2.testproc called

PL/SQL procedure successfully completed.
```

Invalidated objects

When a privilege on an object is revoked from a user, all of that user's PL/SQL objects that refer to that object are temporarily invalidated. Once the privilege has been restored, a user can explicitly recompile and revalidate an object by executing ALTER PROCEDURE, ALTER FUNCTION, or ALTER PACKAGE, as applicable, on the object. Alternatively, each object will be recompiled and revalidated automatically the next time it is executed.

For example, if `user1` has a procedure `user1.proc0` that calls `user2.proc1`, `proc0` becomes invalid if EXECUTE privilege for `proc1` is revoked from `user1`.

To see if any of your objects are invalid:

```
select * from user_objects where status='INVALID';
```

See "SQL Statements" in *Oracle TimesTen In-Memory Database SQL Reference* for information about the ALTER statements.

Example 8–2 Invalidated object

This example shows a sequence that results in an invalidated object, in this case a PL/SQL procedure, as follows:

1. A user is granted CREATE ANY PROCEDURE privilege, creates a procedure in another user's schema, then creates a procedure in his own schema that calls the procedure in the other user's schema.
2. The user is granted EXECUTE privilege to execute the procedure in the other user's schema.
3. The user executes the procedure in his schema that calls the procedure in the other user's schema.
4. EXECUTE privilege for the procedure in the other user's schema is revoked from the user, invalidating the user's own procedure.
5. EXECUTE privilege for the procedure in the other user's schema is granted to the user again. When he executes his own procedure, it is implicitly recompiled and revalidated.

INSTANCE ADMINISTRATOR:

```
Command> grant create any procedure to user1;
```

USER1:

```
Command> create procedure user2.proc1 is
```

```
       > begin
       > dbms_output.put_line('user2.proc1 is called');
       > end;
       > /

Procedure created.

Command> create procedure user1.proc0 is
       > begin
       > dbms_output.put_line('user1.proc0 is called');
       > user2.proc1;
       > end;
       > /

Procedure created.
```

INSTANCE ADMINISTRATOR:

```
Command> grant execute on user2.proc1 to user1;
```

USER1:

```
Command> begin
       > user1.proc0;
       > end;
       > /
user1.proc0 is called
user2.proc1 is called

PL/SQL procedure successfully completed.
```

And to confirm `user1` has no invalid objects at this time:

```
Command> select * from user_objects where status='INVALID';
0 rows found.
```

INSTANCE ADMINISTRATOR:

Now revoke the EXECUTE privilege from `user1`.

```
Command> revoke execute on user2.proc1 from user1;
```

USER1:

Immediately, `user1.proc0` becomes invalid because `user1` no longer has privilege to execute `user2.proc1`.

```
Command> select * from user_objects where status='INVALID';
< PROC0, <NULL>, 273, <NULL>, PROCEDURE, 2009-06-04 14:51:34, 2009-06-04 14:58:23,
2009-06-04:14:58:23, INVALID, N, N, N, 1, <NULL> >
1 row found.
```

So `user1` can no longer execute the procedure.

```
Command> begin
       > user1.proc0;
       > end;
       > /
 8503: ORA-06550: line 2, column 7:
PLS-00905: object USER1.PROC0 is invalid
 8503: ORA-06550: line 2, column 1:
PL/SQL: Statement ignored
The command failed.
```

INSTANCE ADMINISTRATOR:

Now once again grant EXECUTE privilege on `user2.proc1` to `user1`.

```
Command> grant execute on user2.proc1 to user1;
```

USER1:

The procedure `user1.proc0` is still invalid until it is either explicitly or implicitly recompiled. It is implicitly recompiled when it is executed. (Or ALTER PROCEDURE could be used to explicitly recompile it.)

```
Command> select * from user_objects where status='INVALID';
< PROC0, <NULL>, 273, <NULL>, PROCEDURE, 2009-06-04 14:51:34, 2009-06-04 16:13:00,
2009-06-04:16:13:00, INVALID, N, N, N, 1, <NULL> >
1 row found.
Command> begin
       > user1.proc0;
       > end;
       > /
user1.proc0 is called
user2.proc1 is called

PL/SQL procedure successfully completed.

Command> select * from user_objects where status='INVALID';
0 rows found.
```

Access control for SQL operations

For any query or SQL DML statement executed in an anonymous block, or any SQL DDL statement executed in an EXECUTE IMMEDIATE statement, including all such operations discussed in this document or used in any example, it is assumed that the user has appropriate privilege to execute the statement and access the desired objects. SQL executed in a PL/SQL anonymous block requires the same privilege as when executed directly. For example, if you want to insert rows of data into a table you own, no privilege is required. If you want to insert rows of data into a table you do not own, you must be granted INSERT privilege on that table or granted INSERT ANY TABLE.

Refer to "SQL Statements" in *Oracle TimesTen In-Memory Database SQL Reference* for details SQL statements and their required privileges.

Definer's rights and invoker's rights

When a PL/SQL procedure or function is defined, the optional AUTHID clause of the CREATE FUNCTION or CREATE PROCEDURE statement specifies whether the function or procedure executes with *definer's rights* (AUTHID DEFINER, the default) or *invoker's rights* (AUTHID CURRENT_USER). Similarly, for procedures or functions in a package, the AUTHID clause of the CREATE PACKAGE statement specifies whether each member function or procedure of the package executes with definer's rights or invoker's rights. The AUTHID clause is shown in the syntax documentation for these statements, under "SQL Statements" in *Oracle TimesTen In-Memory Database SQL Reference*.

The AUTHID setting affects the name resolution and privilege checking of SQL statements that a procedure or function issues at runtime. With definer's rights, SQL name resolution and privilege checking operate as though the owner of the procedure or function (the "definer", in whose schema it resides) is running it. With invoker's rights, SQL name resolution and privilege checking simply operate as though the current user (the "invoker") is running it.

Invoker’s rights would be useful in a scenario where you might want to grant broad privileges for a body of code, but would want that code to affect only each user's own objects in his or her own schema.

Definer’s rights would be useful in a situation where you want all users to have access to the same centralized tables or other SQL objects, but only for the specific and limited actions that are executed by the procedure. The users would not have access to the SQL objects otherwise.

See "Examples using the AUTHID clause" on page 5-12 for examples using definer’s and invoker’s rights.

Refer to "Using Invoker's Rights or Definer's Rights (AUTHID Clause)" in *Oracle Database PL/SQL Language Reference* for additional information.

Additional access control considerations

This section covers the following:

- Access control for connections and connection attributes
- Access control for system views and supplied packages
- Access control for built-in procedures relating to PL/SQL

Access control for connections and connection attributes

Note the following when connecting to the database:

- Privilege to connect to a TimesTen data store must be explicitly granted to every user, other than the instance administrator, through the CREATE SESSION privilege. This is a system privilege so must be granted to the user either by the instance administrator or by a user with ADMIN privilege. This can be accomplished either directly or through the PUBLIC role. Refer to "Managing Access Control" in *Oracle TimesTen In-Memory Database Operations Guide* for additional information and examples.
- Required privileges for PL/SQL connection attributes are included in "PL/SQL connection attributes" on page 7-3.

Access control for system views and supplied packages

Note the following regarding access to system views and PL/SQL supplied packages.

- SELECT and EXECUTE privileges on various system tables, system views, PL/SQL functions, PL/SQL procedures, and PL/SQL packages are granted by default to all users through the PUBLIC role, of which all users are a member. This role is documented in "Privileges" in *Oracle TimesTen In-Memory Database SQL Reference*. Use the following command to see the list of these public database objects and the associated privileges:

  ```
  SELECT table_name, privilege FROM sys.all_tab_privs
  WHERE grantee='PUBLIC';
  ```

  All users have SELECT privilege for the ALL_* and USER_* system views.
- EXECUTE ANY PROCEDURE does not apply to supplied packages; however, most are accessible through the PUBLIC role. Access control for PL/SQL packages

provided with TimesTen is noted at the beginning of Chapter 9, "TimesTen Supplied PL/SQL Packages."

Access control for built-in procedures relating to PL/SQL

The `ttPLSQLMemoryStats` built-in procedure, which returns statistics about library cache performance and activity, can be called by any user. This procedure is documented under "ttPLSQLMemoryStats" in *Oracle TimesTen In-Memory Database Reference*. Also see Example 7–4, "View PL/SQL performance statistics".

9 TimesTen Supplied PL/SQL Packages

Oracle TimesTen In-Memory Database supplies a number of 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.

This chapter lists and briefly describes the subprograms that comprise each package. For detailed information on these PL/SQL packages, refer to *Oracle Database PL/SQL Packages and Types Reference* (other than for TT_DB_VERSION, which is TimesTen-specific and is detailed here).

- DBMS_LOCK
- DBMS_OUTPUT
- DBMS_PREPROCESSOR
- DBMS_RANDOM
- DBMS_SQL
- DBMS_UTILITY
- TT_DB_VERSION
- UTL_FILE
- UTL_IDENT
- UTL_RAW
- UTL_RECOMP

Notes:

- The packages STANDARD, DBMS_STANDARD, and PLITBLM are not documented here. Subprograms belonging to these packages are part of the PL/SQL language.
- All users have EXECUTE privilege for packages described in this chapter, other than for UTL_RECOMP as noted in that section.

DBMS_LOCK

The DBMS_LOCK package provides an interface to lock-management services. In the current release, TimesTen supports only the sleep feature.

Table 9–1 describes the supported DBMS_LOCK subprogram.

Table 9–1 DBMS_OUTPUT Subprograms

Subprogram	Description
SLEEP procedure	This procedure suspends the session for a given period of time. Specify the amount of time in seconds. The smallest supported increment is a hundredth of a second. For example: `DBMS_LOCK.SLEEP(1.95);` **Notes:** ■ The actual sleep time may be somewhat longer than specified, depending on system activity. ■ If `PLSQL_TIMEOUT` is set to a positive value that is less than this sleep time, then the timeout will take effect first. Be sure that either the sleep value is less than the timeout value, or `PLSQL_TIMEOUT=0` (no timeout). See "PL/SQL connection attributes" on page 7-3 for information about `PLSQL_TIMEOUT`.

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 9–2 describes the DBMS_OUTPUT subprograms.

Table 9–2 DBMS_OUTPUT Subprograms

Subprogram	Description
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 9–3 describes the DBMS_PREPROCESSOR subprograms.

Table 9–3 DBMS_PREPROCESSOR Subprograms

Subprogram	Description
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 9–4 describes the DBMS_RANDOM subprograms.

Table 9–4 DBMS_RANDOM Subprograms

Subprogram	Description
INITIALIZE procedure	Initializes the package with a seed value.
NORMAL function	Returns random numbers in a normal distribution.
RANDOM procedure	Generates a random number.
SEED procedure	Resets the seed.
STRING function	Gets a random string.
TERMINATE procedure	Terminates the package.
VALUE function	The VALUE function 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). The overload function gets a random NUMBER `x`, where `x` is greater than or equal to LOW and less than HIGH.

DBMS_SQL

The DBMS_SQL package provides an interface for using dynamic SQL to parse data manipulation language (DML) or data definition language (DDL) statements using PL/SQL.

The DBMS_SQL package does not support pre-defined data types and overloads with data types that are not supported in TimesTen, such as LOBs, 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 9–5 describes the DBMS_SQL subprograms.

Table 9–5 DBMS_SQL Subprograms

Subprogram	Description
BIND_ARRAY procedure	Binds a given value to a given collection.
BIND_VARIABLE procedure	Binds a given value to a given variable.
CLOSE_CURSOR procedure	Closes a given cursor and frees memory.
COLUMN_VALUE procedure	Returns the value of the cursor element for a given position in a cursor.
COLUMN_VALUE_LONG procedure	Returns a selected part of a LONG column that has been defined using DEFINE_COLUMN_LONG.
DEFINE_ARRAY procedure	Defines a collection to be selected from the given cursor. Use with SELECT statements.
DEFINE_COLUMN procedure	Defines a column to be selected from the given cursor. Use with SELECT statements.
DEFINE_COLUMN_LONG procedure	Defines a LONG column to be selected from the given cursor. Use with SELECT statements.
DESCRIBE_COLUMNS procedure	Describes the columns for a cursor opened and parsed through the DBMS_SQL package.
DESCRIBE_COLUMNS2 procedure	Describes the specified column. Use as an alternative to DESCRIBE_COLUMNS procedure.
DESCRIBE_COLUMNS3 procedure	Describes the specified column. Use as an alternative to DESCRIBE_COLUMNS procedure.
EXECUTE function	Executes a given cursor.
EXECUTE_AND_FETCH function	Executes a given cursor and fetches rows.
FETCH_ROWS function	Fetches a row from a given cursor.
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 cumulative count of the number of rows fetched.
LAST_ROW_ID function	TimesTen does not support ROWID of the last row operated on by a DML statement. This function returns NULL.
LAST_SQL_FUNCTION_CODE function	Returns the SQL function code for the statement.

Table 9–5 (Cont.) DBMS_SQL Subprograms

Subprogram	Description
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 the OPEN, PARSE, and EXECUTE subprograms), 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 9–6 describes DBMS_UTILITY subprograms.

Table 9–6 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.
DB_VERSION procedure	Returns version information for the TimesTen 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_ENDIANNESS 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	Returns bit setting.
NAME_RESOLVE procedure	Resolves the given name of the form: `[[`*`a`*`.]`*`b`*`.]`*`c`*`[@`*`d`*`]` Where *`a`*, *`b`*, and *`c`* are SQL identifiers and *`d`* is a dblink (database link). Do not use `@`*`d`*. TimesTen does not support dblinks.

Table 9–6 (Cont.) DBMS_UTILITY Subprograms

Subprogram	Description
NAME_TOKENIZE procedure	Calls the parser to parse the given name: `"a [.b [.c ]][@dblink]"` Strips double quotes or converts to uppercase if there are no quotes. Ignores comments and does not perform semantic analysis. Missing values are NULL. Do not use *`@dblink`*. TimesTen does not support dblinks.
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 is a TimesTen-specific package that provides the version number and release number for the Oracle TimesTen In-Memory Database.

Table 9–7 describes the TT_DB_VERSION constants.

The primary use case for the TT_DB_VERSION and UTL_IDENT packages is for conditional compilation. See "UTL_IDENT" on page 9-12 for an example.

Table 9–7 TT_DB_VERSION Constants

Name	Description
VERSION	Equals the major release number of the Oracle TimesTen In-Memory Database. VERSION is of type PLS_INTEGER. For example, for the Oracle TimesTen In-Memory Database, Release 11.2.1.0, TT_DB_VERSION.VERSION equals 1121.
RELEASE	Equals the minor release number of the Oracle TimesTen In-Memory Database product. RELEASE is of type PLS_INTEGER. For example, for the Oracle TimesTen In-Memory Database, Release 11.2.1.0, TT_DB_VERSION.RELEASE equals 0.

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.

Table 9–8 describes the UTL_FILE subprograms.

Table 9–8 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 disk 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 disk 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.
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 client, an Oracle server, or Oracle Forms. Each of these has its own version of UTL_IDENT with appropriate settings for the constants.

Table 9–9 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
```

See Example 9–1 below.

Table 9–9 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.

Example 9–1 Using UTL_IDENT and TT_DB_VERSION

This example uses the UTL_IDENT and TT_DB_VERSION packages to show information about the database being used. For the current release, it displays either "Oracle Database 11.1" or "TimesTen 11.2.1". The conditional compilation trigger character, `$`, identifies code that is processed before the application is compiled.

```
Command> run what_db.sql

create or replace function what_db
return varchar2
as
 dbname varchar2(100);
 version varchar2(100);
begin
$if utl_ident.is_timesten
$then
 dbname := 'TimesTen';
 version := substr(tt_db_version.version, 1, 2) ||
            '.' ||
            substr(tt_db_version.version, 3, 1) ||
            '.' ||
            substr(tt_db_version.version, 4, 1);
$elsif utl_ident.is_oracle_server
$then
 dbname := 'Oracle Database';
 version := dbms_db_version.version || '.' || dbms_db_version.release;
$else
 dbname := 'Non-database environment';
 version := '';
```

```
$end
 return dbname || ' ' || version;
end;
/

Function created.

set serveroutput on;

begin
dbms_output.put_line(what_db());
end;
/

TimesTen 11.2.1

PL/SQL procedure successfully completed.
```

UTL_RAW

The UTL_RAW package provides SQL functions for manipulating RAW data types.

Table 9–10 describes the UTL_RAW subprograms.

Table 9–10 UTL_RAW Subprograms

Subprogram	Description
BIT_AND function	Performs a bitwise logical `and` of the value in RAW `r1` with RAW `r2` and returns the result as RAW.
BIT_COMPLEMENT function	Performs a bitwise logical `complement` of the values in RAW `r` and returns the result as RAW.
BIT_OR function	Performs a bitwise logical `or` of the value in RAW `r1` with RAW `r2` and returns the result as RAW.
BIT_XOR function	Performs a bitwise logical `exclusive or` of the value in RAW `r1` with RAW `r2` and returns the result as RAW.
CAST_FROM_BINARY_DOUBLE function	Returns the binary representation of a BINARY_DOUBLE (in RAW).
CAST_FROM_BINARY_FLOAT function	Returns the binary representation of a BINARY_FLOAT (in RAW).
CAST_FROM_BINARY_INTEGER function	Returns the binary representation of a BINARY_INTEGER (in RAW).
CAST_FROM_NUMBER function	Returns the binary representation of a NUMBER (in RAW).
CAST_TO_BINARY_DOUBLE function	Casts the binary representation of a RAW into a BINARY_DOUBLE.
CAST_TO_BINARY_FLOAT function	Casts the binary representation of a RAW into a BINARY_FLOAT.
CAST_TO_BINARY_INTEGER function	Casts the binary representation of a BINARY_INTEGER (in RAW) into a BINARY_INTEGER.
CAST_TO_NUMBER function	Casts the binary representation of a NUMBER (in RAW) into a NUMBER.
CAST_TO_NVARCHAR2 function	Converts a RAW value represented using `n` data bytes into an NVARCHAR2 value with `n` data bytes.
CAST_TO_RAW function	Converts a VARCHAR2 value represented using `n` data bytes into a RAW with `n` data bytes.
CAST_TO_VARCHAR2 function	Converts a RAW value represented using `n` data bytes into a VARCHAR2 value with `n` data bytes.
COMPARE function	Compares RAW `r1` against RAW `r2`.
CONCAT function	Concatenates up to 12 RAW values into a single RAW value.
CONVERT function	Converts RAW `r` from character set `from_charset` to character set `to_charset` and returns the resulting RAW.
COPIES function	Returns `n` copies of `r` concatenated together.
LENGTH function	Returns the length in bytes of a RAW value `r`.

Table 9–10 (Cont.) UTL_RAW Subprograms

Subprogram	Description
OVERLAY function	Overlays the specified portion of the target RAW with the overlay RAW, starting from byte position `pos` of the target and proceeding for `len` bytes.
REVERSE function	Reverses a byte sequence in RAW value `r` from end to end.
SUBSTR function	Returns `len` bytes, starting at `pos` from RAW `r`.
TRANSLATE function	Translates the bytes in the input RAW `r` according to the bytes in the translation RAWs `from_set` and `to_set`.
TRANSLITERATE function	Converts the bytes in the input RAW `r` according to the bytes in the transliteration RAW values `from_set` and `to_set`.
XRANGE function	Returns a RAW value containing all valid one-byte encodings in succession, beginning with the value `start_byte` and ending with the value `end_byte`.

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 9–11 describes the UTL_RECOMP subprograms.

Important: To use this package, you must be the instance administrator and run it as SYS.UTL_RECOMP.

Table 9–11 UTL_RECOMP Subprograms

Name	Description
RECOMP_PARALLEL procedure	Recompiles invalid objects in a given schema, or all invalid objects in the database, in parallel. **Note**: 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.

10 TimesTen PL/SQL Support: Reference Summary

"PL/SQL Language Elements" in *Oracle Database PL/SQL Language Reference* describes language elements that are supported in Oracle.

The purpose of this chapter is to summarize these language elements and indicate their support in TimesTen. Additional TimesTen support considerations are also presented:

- TimesTen PL/SQL language element support
- Additional support considerations and differences

TimesTen PL/SQL language element support

Table 10–1 lists the same PL/SQL language elements as in "PL/SQL Language Elements" in *Oracle Database PL/SQL Language Reference*, 11g Release 1 (11.1). You can refer to that document for detailed information on the PL/SQL language elements. For each element:

- The first column lists the language element.
- The second column includes a brief description of the element.
- The third column indicates if the element is supported in TimesTen.
- The fourth column provides examples and further comments, and refers to additional discussion and examples earlier in this document or in other documents.

Note: Also refer to Table 10–2, " Additional differences between TimesTen PL/SQL and Oracle PL/SQL" for additional support considerations.

Table 10–1 PL/SQL Language Elements

Element Name	Description	Supported	Example/Comment
Assignment statement	Sets current value of a variable, parameter, or element.	Y	See "PL/SQL variables and constants" on page 2-2.
AUTONOMOUS_ TRANSACTION pragma	Marks a routine as autonomous.	N	TimesTen does not support autonomous transactions.
Block declaration	Basic unit of a PL/SQL source program.	Y	See "PL/SQL blocks" on page 2-2.

Table 10–1 (Cont.) PL/SQL Language Elements

Element Name	Description	Supported	Example/Comment
CASE statement	Evaluates an expression, compares it against several values, and takes action according to the comparison that is TRUE.	Y	See "PL/SQL control structures" on page 2-6.
CLOSE statement	Closes cursor or cursor variable.	Y	See Example 2–13, "Using a cursor to retrieve information about an employee" on page 2-16 (among others).
Collection definition	Specifies a collection, which is an ordered group of elements, all of the same type.	Y	Examples include: associative arrays (index-by tables), nested tables, and varrays. While TimesTen supports these types, it does not support passing them between PL/SQL and applications written in other languages. See "Using collections" on page 3-4.
Collection methods	Built-in subprograms that operate on collections and are called using "dot" notation.	Y	See "Using Collection Methods" in *Oracle Database PL/SQL Language Reference*. Examples include COUNT, DELETE, EXISTS, EXTEND, FIRST, LAST, LIMIT, NEXT, PRIOR, and TRIM.
Comments	Text included within your code for explanatory purposes.	Y	Single-line and multi-line comments are supported.
COMMIT statement	Ends the current transaction and makes permanent all changes performed in the transaction.	Y	TimesTen SQL statement. See "COMMIT" in *Oracle TimesTen In-Memory Database SQL Reference*. **Important**: COMMIT and ROLLBACK statements close all cursors in TimesTen.
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.	Y	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.	Y	See "CONTINUE statement" on page 2-8.
Cursor attributes	Appended to the cursor or cursor variable to return useful information about the execution of a data manipulation statement.	Y	Explicit cursors and cursor variables have four attributes: %FOUND, %ISOPEN, %NOTFOUND, %ROWCOUNT. The implicit cursor (SQL) has additional attributes: %BULK_ROWCOUNT, %BULK_EXCEPTIONS. See "Using the %ROWCOUNT and %NOTFOUND attributes" on page 5-4 and "Using FORALL with SQL%BULK_ROWCOUNT" on page 5-6. Also see "Cursor Attributes" in *Oracle Database PL/SQL Language Reference*.

Table 10–1 (Cont.) PL/SQL Language Elements

Element Name	**Description**	**Supported**	**Example/Comment**
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.	Y	See "Use of cursors in PL/SQL programs" on page 2-15.
Cursor variables (REF CURSORs)	Act as handles to cursors over SQL result sets.	Y	TimesTen supports OUT REF CURSORs, one per statement. See "PL/SQL REF CURSORs" on page 3-5.
DELETE statement	Deletes rows from a table.	Y	TimesTen SQL statement. See "DELETE" in *Oracle TimesTen In-Memory Database SQL Reference*.
EXCEPTION_INIT pragma	Associates a user-defined exception with a TimesTen error number.	Y	See "EXCEPTION_INIT Pragma" in *Oracle Database PL/SQL Language Reference*.
Exception definition	Specifies an exception, which is a runtime error or warning condition. Can be predefined or user-defined.	Y	Predefined conditions are raised implicitly. User-defined exceptions are raised explicitly by the RAISE statement. To handle raised exceptions, write separate routines called "exception handlers". See Chapter 4, "Errors and Exception Handling".
EXECUTE IMMEDIATE statement	Builds and executes a dynamic SQL statement in a single operation.	Y	TimesTen supports this to execute SQL DML and DDL statements, but not to execute PL/SQL. See "Dynamic SQL in PL/SQL (EXECUTE IMMEDIATE statement)" on page 2-12.
EXIT statement	Exits a loop and transfers control to the end of the loop.	Y	See Example 7–3 on page 7-8 (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.	Y	
FETCH statement	Retrieves rows of data from the result set of a multi-row query.	Y	See Example 2–13 on page 2-16 (among others).
FORALL statement	Bulk-binds input collections before sending them to the SQL engine.	Y	See "FORALL and BULK COLLECT operations" on page 2-13.

Table 10–1 (Cont.) PL/SQL Language Elements

Element Name	Description	Supported	Example/Comment
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.	Y	A function or procedure must be executed in an anonymous block. It cannot be executed through a CALL statement or from any other SQL statement in TimesTen. Use the CREATE FUNCTION statement in TimesTen SQL to create stored functions. See "PL/SQL procedures and functions" on page 2-16. Also see "CREATE FUNCTION" in *Oracle TimesTen In-Memory Database SQL Reference*. Also see the table entry below for "Procedure declaration and definition".
GOTO statement	Branches unconditionally to a statement label or block label.	Y	See "GOTO Statement" in *Oracle Database PL/SQL Language Reference*.
IF statement	Executes or skips a sequence of statements depending on the value of the associated boolean expression.	Y	See "Conditional control" on page 2-6.
INLINE pragma	Specifies whether a subprogram call is to be inline.	Y	See "INLINE Pragma" in *Oracle Database PL/SQL Language Reference*.
INSERT statement	Inserts one or more rows of data into a table.	Y	TimesTen SQL statement. See "Example using the INSERT statement" on page 5-2. Also see "INSERT" in *Oracle TimesTen In-Memory Database SQL Reference*.
Literal declaration	Specifies a numeric, character string, or boolean value.	Y	Examples: Numeric literal: 135 String literal: 'TimesTen'
LOCK TABLE statement	Locks database tables in a specified lock mode.	N	TimesTen does not support the LOCK TABLE statement.
LOOP statement	Executes a sequence of statements multiple times. Can be used, for example, in implementing a FOR loop or WHILE loop.	Y	See Example 2–8, "Using a WHILE loop" on page 2-7. Also see "LOOP Statements" in *Oracle Database PL/SQL Language Reference*.
MERGE statement	Allows you to select rows from one or more sources for update or insertion into a target table.	Y	TimesTen SQL statement. See "MERGE" in *Oracle TimesTen In-Memory Database SQL Reference*.
NULL statement	A no-operation statement. Passes control to the next statement without performing any action.	Y	See "NULL Statement" in *Oracle Database PL/SQL Language Reference*. Also, one is used in Example 3–3 on page 3-5.
Object type declaration	Specifies a custom object type, which is created in SQL and stored in the database.	N	Object types are not supported at the database level. For example, CREATE TYPE is not supported.

Table 10–1 (Cont.) PL/SQL Language Elements

Element Name	Description	Supported	Example/Comment
OPEN statement	Executes the query associated with a cursor. Allocates database resources to process the query, and identifies the result set.	Y	See Example 2–13, "Using a cursor to retrieve information about an employee" on page 2-16.
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.	Y	See Example 3–4, "Fetch rows from result set of a dynamic multirow query" on page 3-6.
Package declaration	Specifies a package, which is a database object that groups logically related PL/SQL types, items, and subprograms.	Y	TimesTen SQL statements CREATE PACKAGE and CREATE PACKAGE BODY. See "SQL Statements" in *Oracle TimesTen In-Memory Database SQL Reference* for information about these statements. Also see "PL/SQL packages" on page 2-17.
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.	Y	A procedure or function must be executed in an anonymous block. It cannot be executed through a CALL statement or from any other SQL statement in TimesTen. Use the CREATE PROCEDURE statement in TimesTen SQL to create stored procedures. See "PL/SQL procedures and functions" on page 2-16. Also see "CREATE PROCEDURE" in *Oracle TimesTen In-Memory Database SQL Reference*. Also see 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.	Y	See "Using the RAISE statement" 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).	Y	See "Using records" on page 3-5.
RESTRICT_REFERENCES pragma	Asserts that a subprogram (usually a function) in a package specification or object type specification does not read or write database tables or package variables.	N	TimesTen ignores this.
RETURN statement	Immediately completes the execution of a subprogram and returns control to the invoker. Execution resumes with the statement following the subprogram call.	Y	See "RETURN Statement" in *Oracle Database PL/SQL Language Reference*.

Table 10–1 (Cont.) PL/SQL Language Elements

Element Name	**Description**	**Supported**	**Example/Comment**
RETURNING INTO clause	Specifies the variables in which to store the values returned by the statement to which the clause belongs.	Y	TimesTen clause. See "RETURNING INTO clause" on page 2-14 and "Examples using RETURNING INTO" on page 5-11.
ROLLBACK statement	Undoes database changes made during the current transaction.	Y	TimesTen SQL statement. See "ROLLBACK" in *Oracle TimesTen In-Memory Database SQL Reference*. **Important**: COMMIT and ROLLBACK statements close all cursors in TimesTen.
%ROWTYPE attribute	Provides a record type that represents a row in a database table.	Y	See Example 2–2 on page 2-3.
SAVEPOINT statement	Names and marks the current point in the processing of a transaction.	N	TimesTen does not support savepoints.
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 in Table 10–2 below), this statement retrieves an entire result set at once.	Y	See Example 2–3, "Using SELECT INTO to assign values to variables" on page 2-4. Also see "Querying Data with PL/SQL" in *Oracle Database PL/SQL Language Reference*.
SERIALLY_REUSABLE pragma	Indicates that package state is needed only for the duration of one call to the server.	N	TimesTen does not support the SERIALLY_REUSABLE pragma.
SET TRANSACTION statement	Begins a read-only or read/write transaction.	N	TimesTen does not support the SET TRANSACTION statement.
SQL cursor	Either explicit or implicit, is used to handle the result set of a SELECT statement.	Y	See "Use of cursors in PL/SQL programs" on page 2-15.
SQLCODE function	Returns number code of the most recent exception.	Y	Given the same error condition, error codes returned by the built-in function SQLCODE are the same in TimesTen as in Oracle, although the SQLERRM returns may be different. This is also noted in "TimesTen error messages and SQL codes" on page 4-7.

Table 10–1 (Cont.) PL/SQL Language Elements

Element Name	Description	Supported	Example/Comment
SQLERRM function	Returns the error message associated with the error-number argument.	Y	Given the same error condition, error messages returned by the built-in function SQLERRM are not necessarily the same in TimesTen as in Oracle, although SQLCODE returns are the same. This is also noted in "TimesTen error messages and SQL codes" on page 4-7.
%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.	Y	See "PL/SQL variables and constants" 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.	Y	TimesTen SQL statement. See "UPDATE" in *Oracle TimesTen In-Memory Database SQL Reference*.

Additional support considerations and differences

Table 10–2 provides a summary of additional support considerations for features that are supported in TimesTen. Comments explain differences between PL/SQL in TimesTen and PL/SQL in Oracle (Release 11.1.0.7).

Table 10–2 Additional differences between TimesTen PL/SQL and Oracle PL/SQL

Feature	Comment
ALTER {PROCEDURE | FUNCTION | PACKAGE}	Syntax and semantics are the same as in Oracle. You can refer to information about these statements in "SQL Statements" in *Oracle TimesTen In-Memory Database SQL Reference*.
ALTER SESSION statement	You can use ALTER SESSION to set some PL/SQL connection attributes as discussed in "PL/SQL connection attributes" on page 7-3. For more information on this statement in TimesTen, see "ALTER SESSION" in *Oracle TimesTen In-Memory Database SQL Reference*.
Built-in functions and SQL functions	See "PL/SQL built-in functions and SQL functions" on page 2-5.
BULK COLLECT	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 2-13 and "Examples using FORALL and BULK COLLECT" on page 5-5.
CALL statement	In TimesTen, use the CALL statement to execute TimesTen built-in procedures. In Oracle, use the CALL statement to execute PL/SQL stored procedures and functions.
Connection attributes	Equivalent to initialization parameters in Oracle. See "PL/SQL connection attributes" on page 7-3. Also see "Data Store Attributes" in *Oracle TimesTen In-Memory Database Reference*.

Table 10–2 (Cont.) Additional differences between TimesTen PL/SQL and Oracle PL/SQL

Feature	Comment
CREATE FUNCTION	The CREATE FUNCTION statement is supported in TimesTen, but the AS LANGUAGE, AS EXTERNAL, and PIPELINED clauses are not supported. See "PL/SQL procedures and functions" on page 2-16. Also see "CREATE FUNCTION" in *Oracle TimesTen In-Memory Database SQL Reference*. You are not required to run `DBMSSTDX.SQL` in TimesTen.
CREATE LIBRARY	The CREATE LIBRARY statement is not supported in TimesTen.
CREATE PACKAGE [BODY]	Syntax and semantics are the same as in Oracle. See "PL/SQL packages" on page 2-17. Also see "CREATE PACKAGE" and "CREATE PACKAGE BODY" in *Oracle TimesTen In-Memory Database SQL Reference*. You are not required to run `DBMSSTDX.SQL` in TimesTen.
CREATE PROCEDURE	CREATE PROCEDURE statement is supported in TimesTen, but the AS LANGUAGE and AS EXTERNAL clauses are not supported. See "PL/SQL procedures and functions" on page 2-16. Also see "CREATE PROCEDURE" in *Oracle TimesTen In-Memory Database SQL Reference*. You are not required to run `DBMSSTDX.SQL` in TimesTen.
Data type support	See "Understanding the data type environments" on page 3-1.
DROP { PROCEDURE | FUNCTION | PACKAGE}	Syntax and semantics are the same as in Oracle. You can refer to information about these statements in "SQL Statements" in *Oracle TimesTen In-Memory Database SQL Reference*.
Error reporting	TimesTen applications report errors using Oracle error codes instead of TimesTen error codes. The error messages that accompany the error codes are either TimesTen error messages or Oracle error messages.
Executing PL/SQL from SQL	In TimesTen, you cannot execute PL/SQL from either a static or dynamic SQL statement.
Executing PL/SQL from client applications	Oracle TimesTen In-Memory Database supports ODBC, OCI, Pro*C, TTClasses (a set of TimesTen C++ classes), and JDBC. Refer to documentation for those programming interfaces, as desired.
Native Dynamic SQL execution	In TimesTen, in addition to Oracle functionality, the EXECUTE IMMEDIATE statement can be used to execute TimesTen built-in procedures and TimesTen-specific SQL features (such as SELECT FIRST).
Non-ASCII names	In TimesTen (unlike in Oracle), use of non-ASCII character sets in names of tables, columns, procedures, functions, and other database objects is not supported.

Non-uppercase names	In TimesTen (unlike in Oracle), quoted non-uppercase names of tables, columns, procedures, functions, and other database objects are not supported (such as `lowercase` and `MixedCase`). For example, you cannot have the following:
	
	`create or replace procedure "MixedCase" as`
	`begin`
	`  ...`
	`end;`
	`/`

Table 10–2 (Cont.) Additional differences between TimesTen PL/SQL and Oracle PL/SQL

Feature	Comment
Result cache	Oracle TimesTen In-Memory Database does not support the PL/SQL function result cache.
SOUNDEX SQL function	TimesTen does not support this function (which returns a character string containing the phonetic representation of a `char`).
SQL constructs, misc.	Oracle TimesTen In-Memory Database does not support the CREATE TYPE statement, database links, or triggers.
Supplied packages	Oracle TimesTen In-Memory Database provides a subset of the Oracle PL/SQL supplied packages. See Chapter 9, "TimesTen Supplied PL/SQL Packages".
System tables and views	Oracle TimesTen In-Memory Database supports a subset of the Oracle system tables and views. See "System and Replication Tables" in *Oracle TimesTen In-Memory Database SQL Reference*.
`ttPLSQLMemoryStats` built-in procedure	In TimesTen, use this built-in procedure to return statistics about library cache performance and activity. See "ttPLSQLMemoryStats" in *Oracle TimesTen In-Memory Database Reference*. In Oracle, use the V$LIBRARYCACHE system view to retrieve the same statistical information.

Index