## Contents

### Preface
- Audience ................................................................................................................. ix
- Documentation Accessibility ................................................................................... ix
- Related Documents .................................................................................................. x
- Conventions ............................................................................................................... x

### 1 Introduction to TimesTen-Supplied PL/SQL Packages and Types
- Package overview ..................................................................................................... 1-3
  - Package components .............................................................................................. 1-4
  - Using TimesTen-supplied packages ....................................................................... 1-5
  - Referencing package contents ................................................................................ 1-6
  - Running package examples .................................................................................... 1-7
- Summary of TimesTen-supplied PL/SQL packages .................................................. 1-8

### 2 DBMS_LOCK
- Using DBMS_LOCK .................................................................................................... 2-2
- Summary of DBMS_LOCK subprograms ................................................................... 2-3
  - SLEEP procedure .................................................................................................. 2-4

### 3 DBMS_OUTPUT
- Using DBMS_OUTPUT ............................................................................................... 3-2
  - Overview ................................................................................................................ 3-3
  - Operational notes ................................................................................................... 3-4
  - Exceptions .............................................................................................................. 3-5
  - Rules and limits ....................................................................................................... 3-6
  - Example .................................................................................................................. 3-7
- Data structures ......................................................................................................... 3-8
  - CHARARR table type ............................................................................................. 3-9
  - DBMSOUTPUT_LINESARRAY table type ............................................................... 3-10
- Summary of DBMS_OUTPUT subprograms ............................................................... 3-11
  - DISABLE procedure ............................................................................................. 3-12
  - ENABLE procedure ............................................................................................... 3-13
  - GET_LINE procedure ......................................................................................... 3-14
  - GET_LINES procedure ......................................................................................... 3-15
  - NEW_LINE procedure .......................................................................................... 3-16
4 DBMS_PREPROCESSOR
Using DBMS_PREPROCESSOR .......................................................................................... 4-2
  Overview .................................................................................................................. 4-3
  Operating notes ...................................................................................................... 4-4
Data structures ........................................................................................................... 4-5
  SOURCE_LINES_T table type ............................................................................... 4-6
Summary of DBMS_PREPROCESSOR subprograms ....................................................... 4-7
  GET_POST_PROCESSED_SOURCE function ....................................................... 4-8
  PRINT_POST_PROCESSED_SOURCE procedure .............................................. 4-10

5 DBMS_RANDOM
Using DBMS_RANDOM ................................................................................................. 5-2
  Operational notes ................................................................................................... 5-3
Summary of DBMS_RANDOM subprograms ................................................................... 5-4
  INITIALIZE procedure .......................................................................................... 5-5
  NORMAL function ................................................................................................... 5-6
  RANDOM function ................................................................................................... 5-7
  SEED procedure ...................................................................................................... 5-8
  STRING function ..................................................................................................... 5-9
  TERMINATE procedure ........................................................................................... 5-10
  VALUE function ..................................................................................................... 5-11

6 DBMS_SQL
Using DBMS_SQL ......................................................................................................... 6-2
  Overview .................................................................................................................. 6-3
  Security model ......................................................................................................... 6-4
  Constants .................................................................................................................. 6-5
  Exceptions ................................................................................................................ 6-6
  Operational notes ..................................................................................................... 6-7
  Execution flow ......................................................................................................... 6-7
    OPEN_CURSOR ..................................................................................................... 6-7
    PARSE ................................................................................................................... 6-7
    BIND_VARIABLE or BIND_ARRAY ....................................................................... 6-8
    DEFINE_COLUMN or DEFINE_ARRAY .............................................................. 6-9
    EXECUTE .............................................................................................................. 6-9
    FETCH_ROWS or EXECUTE_AND_FETCH ......................................................... 6-9
    VARIABLE_VALUE or COLUMN_VALUE ......................................................... 6-9
    CLOSE_CURSOR ................................................................................................... 6-9
    Processing queries ............................................................................................... 6-9
    Processing updates, inserts, and deletes ............................................................ 6-9
    Locating errors ..................................................................................................... 6-10
  Examples .................................................................................................................. 6-11
Data structures ........................................................................................................... 6-23
7 DBMS_UTILITY

Using DBMS_UTILITY ................................................................. 7-2
Security model ................................................................. 7-3
Constants ................................................................. 7-4
Types ............................................................................. 7-5
dblink_array ........................................................................ 7-5
9  UTL_FILE

Using UTL_FILE ................................................................. 9-2
  Security model ............................................................. 9-3
  Operational notes ......................................................... 9-4
  Rules and limits ........................................................... 9-5
  Exceptions ...................................................................... 9-6
  Examples ....................................................................... 9-7

8  TT_DB_VERSION

Using TT_DB_VERSION ................................................. 8-2
  Overview ...................................................................... 8-3
  Constants ..................................................................... 8-4
  Examples ...................................................................... 8-5
10 UTL_IDENT

Using UTL_IDENT ................................................................. 10-2
Overview ........................................................................ 10-3
Security model .............................................................. 10-4
Constants ........................................................................ 10-5
Examples ........................................................................ 10-6

11 UTL_RAW

Using UTL_RAW ................................................................. 11-2
Overview ........................................................................ 11-3
Operational notes ......................................................... 11-4
Summary of UTL_RAW subprograms ............................... 11-5
BIT_AND function ....................................................... 11-7
BIT_COMPLEMENT function ....................................... 11-8
BIT_OR function .......................................................... 11-9
BIT_XOR function ....................................................... 11-10
CAST_FROM_BINARY_DOUBLE function .................... 11-11
CAST_FROM_BINARY_FLOAT function ......................... 11-13
CAST_FROM_BINARY_INTEGER function ...................... 11-14
CAST_FROM_NUMBER function .................................... 11-15
CAST_TO_BINARY_DOUBLE function ......................... 11-16
CAST_TO_BINARY_FLOAT function .............................. 11-18
CAST_TO_BINARY_INTEGER function ......................... 11-20
CAST_TO_NUMBER function ........................................ 11-21
CAST_TO_NVARCHAR2 function .................................... 11-22
<table>
<thead>
<tr>
<th>Function Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAST_TO_RAW function</td>
<td>11-23</td>
</tr>
<tr>
<td>CAST_TO_VARCHAR2 function</td>
<td>11-24</td>
</tr>
<tr>
<td>COMPARE function</td>
<td>11-25</td>
</tr>
<tr>
<td>CONCAT function</td>
<td>11-26</td>
</tr>
<tr>
<td>CONVERT function</td>
<td>11-27</td>
</tr>
<tr>
<td>COPIES function</td>
<td>11-28</td>
</tr>
<tr>
<td>LENGTH function</td>
<td>11-29</td>
</tr>
<tr>
<td>OVERLAY function</td>
<td>11-30</td>
</tr>
<tr>
<td>REVERSE function</td>
<td>11-32</td>
</tr>
<tr>
<td>SUBSTR function</td>
<td>11-33</td>
</tr>
<tr>
<td>TRANSLATE function</td>
<td>11-36</td>
</tr>
<tr>
<td>TRANSLITERATE function</td>
<td>11-39</td>
</tr>
<tr>
<td>XRANGE function</td>
<td>11-43</td>
</tr>
</tbody>
</table>

## 12 UTL_RECOMP

### Using UTL_RECOMP
- Overview ........................................... 12-2
- Operational notes .............................. 12-3
- Examples .......................................... 12-5

### Summary of UTL_RECOMP subprograms
- RECOMP_PARALLEL procedure .................. 12-7
- RECOMP_SERIAL procedure ..................... 12-8
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 contains these topics:

- Audience
- Documentation Accessibility
- Related Documents
- Conventions

**Audience**

*Oracle TimesTen In-Memory Database PL/SQL Packages Reference* is a reference for programmers, systems analysts, project managers, and others interested in developing database applications using PL/SQL. This manual assumes a working knowledge of application programming and familiarity with SQL and PL/SQL to access information in relational database systems.

**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/](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](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](http://www.fcc.gov/cgb/dro/trsphonebk.html).

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


*Oracle TimesTen In-Memory Database PL/SQL Developer’s Guide* is especially relevant.

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:


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, Tru64, and AIX.

This document uses the following text conventions:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>italic</em></td>
<td>Italic type indicates terms defined in text, book titles, or emphasis.</td>
</tr>
<tr>
<td><em>monospace</em></td>
<td>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.</td>
</tr>
</tbody>
</table>
TimesTen documentation uses these variables to identify path, file and user names:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>install_dir</strong></td>
<td>The path that represents the directory where TimesTen is installed.</td>
</tr>
<tr>
<td><strong>TTinstance</strong></td>
<td>The instance name for your specific installation of TimesTen. Each installation of TimesTen must be identified at install time with a unique instance name. This name appears in the install path.</td>
</tr>
<tr>
<td><strong>bits or bb</strong></td>
<td>Two digits, either 32 or 64, that represent either the 32-bit or 64-bit operating system.</td>
</tr>
<tr>
<td><strong>release or rr</strong></td>
<td>Numbers that represent a major TimesTen release, with or without dots. For example, 1121 or 11.2.1 represents TimesTen Release 11.2.1.</td>
</tr>
<tr>
<td><strong>DSN</strong></td>
<td>The data source name.</td>
</tr>
</tbody>
</table>
Introduction to TimesTen-Supplied PL/SQL Packages and Types

A set of PL/SQL packages is installed when you enable PL/SQL for the TimesTen In-Memory Database. These packages extend database functionality and allow PL/SQL access to SQL features. To display the list of packages provided with TimesTen, use the system view ALL_PROCEDURES for objects owned by SYS. The following example shows this. As with other ALL_* system views, all users have SELECT privilege for the ALL_PROCEDURES system view.

```
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.
```

This is the list of TimesTen-supplied packages currently installed in the database. It includes internal-use packages and public packages. This manual documents only the public packages, as summarized in "Summary of TimesTen-supplied PL/SQL packages" on page 1-8.

**Notes:** The following packages are for internal use only and are not documented in this manual:

- **STANDARD, PLITBLM, and DBMS_STANDARD:** Programs defined in these packages are part of the PL/SQL language.
- **DBMS_SYS_ERROR:** Defines private PL/SQL subprograms for system error messages arising from DBMS* routines.
- **SYS_STUB_FOR_PURITY_ANALYSIS:** Required for the creation of top-level subprograms.
This chapter contains these topics:

■ Package overview
■ Summary of TimesTen-supplied PL/SQL packages
Package overview

A package is an encapsulated collection of related program objects stored together in the database. Program objects are procedures, functions, variables, constants, cursors, and exceptions.

This section covers the following topics:

- Package components
- Using TimesTen-supplied packages
- Referencing package contents
- Running package examples
Package components

PL/SQL packages have two parts, the specification and the body, although sometimes the body is unnecessary. The specification is the interface to your application; it declares the types, variables, constants, exceptions, cursors, and subprograms available for use. The body fully defines cursors and subprograms, and so implements the specification.

Unlike subprograms, packages cannot be called, parameterized, or nested. However, the formats of a package and a subprogram are similar:

CREATE PACKAGE name AS  -- specification (visible part)
  -- public type and item declarations
  -- subprogram specifications
END [name];

CREATE PACKAGE BODY name AS  -- body (hidden part)
  -- private type and item declarations
  -- subprogram bodies
BEGIN
  -- initialization statements
END [name];

The specification holds public declarations that are visible to your application. The body holds implementation details and private declarations that are hidden from your application. You can debug, enhance, or replace a package body without changing the specification. You can change a package body without recompiling calling programs because the implementation details in the body are hidden from your application.
Using TimesTen-supplied packages

TimesTen-supplied packages are automatically installed when the database is created.

All users have EXECUTE privilege for packages described in this document, other than for UTL_RECOMP and UTL_FILE, as noted in those chapters. To use this package, you must be the instance administrator.

To select from a view defined with a PL/SQL function, you must have SELECT privileges on the view. No separate EXECUTE privileges are needed to select from the view. Instructions on special requirements for packages are documented in the individual chapters.

**Note:** In TimesTen, running as the instance administrator is comparable to running as the Oracle SYSDBA. Running as the ADMIN user is comparable to running as the Oracle DBA.
Referencing package contents

To reference the types, items, and subprograms declared in a package specification, use the dot notation. For example:

package_name.type_name
package_name.item_name
package_name.subprogram_name
Running package examples

In order to see the output from the package examples in this document, first execute the following command in ttIsql:

Command> set serveroutput on
Summary of TimesTen-supplied PL/SQL packages

Table 1–1 lists the PL/SQL server packages supplied with TimesTen for public use. These packages run as the invoking user, rather than the package owner.

---

**Caution:**

- The procedures and functions provided in these packages and their external interfaces are reserved by Oracle and are subject to change.
- Modifying supplied packages can cause internal errors and database security violations. Do not modify supplied packages.

---

**Table 1–1 Summary of TimesTen-supplied PL/SQL packages**

<table>
<thead>
<tr>
<th>Package Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBMS_LOCK</td>
<td>Provides an interface to Lock Management services. TimesTen supports only the SLEEP procedure, to suspend the session for a specified period of time.</td>
</tr>
<tr>
<td>DBMS_OUTPUT</td>
<td>Enables you to send messages from stored procedures and packages.</td>
</tr>
<tr>
<td>DBMS_PREPROCESSOR</td>
<td>Provides an interface to print or retrieve the source text of a PL/SQL unit in its post-processed form.</td>
</tr>
<tr>
<td>DBMS_RANDOM</td>
<td>Provides a built-in random number generator.</td>
</tr>
<tr>
<td>DBMS_SQL</td>
<td>Lets you use dynamic SQL to access the database.</td>
</tr>
<tr>
<td>DBMS_UTILITY</td>
<td>Provides various utility routines.</td>
</tr>
<tr>
<td>TT_DB_VERSION</td>
<td>Indicates the TimesTen major and minor version numbers.</td>
</tr>
<tr>
<td>UTL_FILE</td>
<td>Enables your PL/SQL programs to read and write operating system text files and provides a restricted version of standard operating system stream file I/O.</td>
</tr>
<tr>
<td>UTL_IDENT</td>
<td>Indicates which database or client PL/SQL is running in, such as TimesTen versus Oracle, and server versus client. (Each database or client running PL/SQL has its own copy of this package.)</td>
</tr>
<tr>
<td>UTL_RAW</td>
<td>Provides SQL functions for manipulating RAW data types.</td>
</tr>
<tr>
<td>UTL_RECOMP</td>
<td>Recompiles invalid PL/SQL modules.</td>
</tr>
</tbody>
</table>

---

**Note:** Notes on data types:

- The PLS_INTEGER and BINARY_INTEGER data types are identical. This document uses "BINARY_INTEGER" to indicate data types in reference information (such as for table types, record types, subprogram parameters, or subprogram return values), but may use either in discussion and examples.
- The INTEGER and NUMBER(38) data types are also identical. This document uses "INTEGER" throughout.
The DBMS_LOCK package provides an interface to Lock Management services. In the current release, TimesTen supports only the SLEEP subprogram. This chapter contains the following topics:

- Using DBMS_LOCK
- Summary of DBMS_LOCK subprograms
Using DBMS_LOCK

TimesTen currently implements only the SLEEP subprogram, used to suspend the session for a specified period of time.
Summary of DBMS_LOCK subprograms

In the current release, TimesTen supports only the SLEEP subprogram.

<table>
<thead>
<tr>
<th>Subprogram</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLEEP procedure</td>
<td>Suspends the session for a specified period of time.</td>
</tr>
</tbody>
</table>
SLEEP procedure

This procedure suspends the session for a specified period of time.

Syntax

```
DBMS_LOCK.SLEEP (
    seconds IN NUMBER);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>seconds</td>
<td>Amount of time, in seconds, to suspend the session. The smallest increment is a hundredth of a second.</td>
</tr>
</tbody>
</table>

Usage notes

- The actual sleep time may be somewhat longer than specified, depending on system activity.
- If the PLSQL_TIMEOUT connection attribute is set to a positive value that is less than this sleep time, 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" in Oracle TimesTen In-Memory Database PL/SQL Developer’s Guide for information about PLSQL_TIMEOUT.

Examples

```
DBMS_LOCK.SLEEP(1.95);
```
The DBMS_OUTPUT package enables you to send messages from stored procedures and packages. The package is especially useful for displaying PL/SQL debugging information.

This chapter contains the following topics:

- **Using DBMS_OUTPUT**
  - Overview
  - Operational notes
  - Exceptions
  - Rules and limits
  - Examples
- **Data structures**
  - Table types
- **Summary of DBMS_OUTPUT subprograms**
Using DBMS_OUTPUT

This section contains topics which relate to using the DBMS_OUTPUT package.

- Overview
- Operational notes
- Exceptions
- Rules and limits
- Example
Overview

The `PUT procedure` and `PUT_LINE procedure` in this package enable you to place information in a buffer that can be read by another procedure or package. In a separate PL/SQL procedure or anonymous block, you can display the buffered information by calling the `GET_LINE procedure` and `GET_LINES procedure`.

If the package is disabled, all calls to subprograms are ignored. In this way, you can design your application so that subprograms are available only when a client is able to process the information.
Operational notes

- If you do not call GET_LINE, or if you do not display the messages on your screen in `ttIsql`, the buffered messages are ignored.
- The `ttIsql` utility calls GET_LINES after issuing a SQL statement or anonymous PL/SQL calls.
- Typing `SET SERVEROUTPUT ON` in `ttIsql` has the same effect as the following:
  ```plsql
  DBMS_OUTPUT.ENABLE (buffer_size => NULL);
  ```
  There is no limit on the output.
- You should generally avoid having application code invoke either the `DISABLE procedure` or `ENABLE procedure` because this could subvert the attempt of an external tool like `ttIsql` to control whether or not to display output.

**Note:** Messages sent using `DBMS_OUTPUT` are not actually sent until the sending subprogram completes. There is no mechanism to flush output during the execution of a procedure.
Exceptions

DBMS_OUTPUT subprograms raise the application error ORA-20000, and the output procedures can return the following errors:

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORU-10027</td>
<td>Buffer overflow.</td>
</tr>
<tr>
<td>ORU-10028</td>
<td>Line length overflow.</td>
</tr>
</tbody>
</table>
Rules and limits

- The maximum line size is 32767 bytes.
- The default buffer size is 20000 bytes. The minimum size is 2000 bytes and the maximum is unlimited.
Example

Example: Debugging stored procedures
The DBMS_OUTPUT package is commonly used to debug stored procedures or functions.

This function queries the employees table of the HR schema and returns the total salary for a specified department. The function includes calls to the PUT_LINE procedure:

```sql
CREATE OR REPLACE FUNCTION dept_salary (dnum NUMBER) RETURN NUMBER IS
  CURSOR emp_cursor IS
    select salary, commission_pct from employees where department_id = dnum;
  total_wages NUMBER(11, 2) := 0;
  counter NUMBER(10) := 1;
BEGIN
  FOR emp_record IN emp_cursor LOOP
    emp_record.commission_pct := NVL(emp_record.commission_pct, 0);
    total_wages := total_wages + emp_record.salary + emp_record.commission_pct;
    DBMS_OUTPUT.PUT_LINE('Loop number = ' || counter || '; Wages = '|| TO_CHAR(total_wages)); /* Debug line */
    counter := counter + 1; /* Increment debug counter */
  END LOOP;
  DBMS_OUTPUT.PUT_LINE('Total wages = ' || TO_CHAR(total_wages));
RETURN total_wages;
END;
/
```

Assume the user executes the following statements in ttIsql:

```sql
Command> SET SERVEROUTPUT ON
Command> VARIABLE salary NUMBER;
Command> EXECUTE :salary := dept_salary(20);
```

The user would then see output such as the following:

```
Loop number = 1; Wages = 13000
Loop number = 2; Wages = 19000
Total wages = 190000
```

PL/SQL procedure successfully executed.
Data structures

The DBMS_OUTPUT package declares two collection types for use with the GET_LINES procedure.

---

**Note:** Notes on data types:

- The PLS_INTEGER and BINARY_INTEGER data types are identical. This document uses "BINARY_INTEGER" to indicate data types in reference information (such as for table types, record types, subprogram parameters, or subprogram return values), but may use either in discussion and examples.

- The INTEGER and NUMBER(38) data types are also identical. This document uses "INTEGER" throughout.

---

Table types

- CHARARR table type
- DBMSOUTPUT_LINESARRAY table type
CHARARR table type

This package type is to be used with the GET_LINES procedure to obtain text submitted through the PUT procedure and PUT_LINE procedure.

Syntax

TYPE CHARARR IS TABLE OF VARCHAR2(32767) INDEX BY BINARY_INTEGER;
DBMSOUTPUT_LINESARRAY table type

This package type is to be used with the GET_LINES procedure to obtain text submitted through the PUT procedure and PUT_LINE procedure.

Syntax

```plaintext
TYPE DBMSOUTPUT_LINESARRAY IS
    VARRAY(2147483647) OF VARCHAR2(32767);
```

## Summary of DBMS_OUTPUT subprograms

### Table 3–2  DBMS_OUTPUT package subprograms

<table>
<thead>
<tr>
<th>Subprogram</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DISABLE procedure</td>
<td>Disables message output.</td>
</tr>
<tr>
<td>ENABLE procedure</td>
<td>Enables message output.</td>
</tr>
<tr>
<td>GET_LINE procedure</td>
<td>Retrieves one line from buffer.</td>
</tr>
<tr>
<td>GET_LINES procedure</td>
<td>Retrieves an array of lines from buffer.</td>
</tr>
<tr>
<td>NEW_LINE procedure</td>
<td>Terminates a line created with PUT.</td>
</tr>
<tr>
<td>PUT procedure</td>
<td>Places a line in the buffer.</td>
</tr>
<tr>
<td>PUT_LINE procedure</td>
<td>Places partial line in buffer.</td>
</tr>
</tbody>
</table>
DISABLE procedure

This procedure disables calls to PUT, PUT_LINE, NEW_LINE, GET_LINE, and GET_LINES, and purges the buffer of any remaining information.

As with the ENABLE procedure, you do not need to call this procedure if you are using the SET SERVEROUTPUT ON setting from ttIsql.

Syntax

DBMS_OUTPUT.DISABLE;

Pragmas

pragma restrict_references(disable,WNDS,RNDS);
ENABLE procedure

This procedure enables calls to PUT, PUT_LINE, NEW_LINE, GET_LINE, and GET_LINES. Calls to these procedures are ignored if the DBMS_OUTPUT package is not activated.

Syntax

```sql
DBMS_OUTPUT.ENABLE (
    buffer_size IN INTEGER DEFAULT 20000);
```

Pragmas

```sql
pragma restrict_references(enable,WNDS,RNDS);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>buffer_size</td>
<td>Upper limit, in bytes, the amount of buffered information. Setting buffer_size to NULL specifies that there should be no limit.</td>
</tr>
</tbody>
</table>

Usage notes

- It is not necessary to call this procedure when you use SET SERVEROUTPUT ON from ttIsql.
- If there are multiple calls to ENABLE, then `buffer_size` is the last of the values specified. The maximum size is 1,000,000 and the minimum is 2000 when the user specifies `buffer_size` (NOT NULL).
- NULL is expected to be the usual choice. The default is 20,000 for backwards compatibility with earlier database versions that did not support unlimited buffering.
GET_LINE procedure

This procedure retrieves a single line of buffered information.

Syntax

```sql
DBMS_OUTPUT.GET_LINE (  
    line   OUT VARCHAR2,  
    status OUT INTEGER);  
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>line</td>
<td>Returns a single line of buffered information, excluding a final newline character. You should declare this parameter as VARCHAR2(32767) to avoid the risk of &quot;ORA-06502: PL/SQL: numeric or value error: character string buffer too small&quot;.</td>
</tr>
<tr>
<td>status</td>
<td>If the call completes successfully, then the status returns as 0. If there are no more lines in the buffer, then the status is 1.</td>
</tr>
</tbody>
</table>

Usage notes

- You can choose to retrieve from the buffer a single line or an array of lines. Call the GET_LINE procedure to retrieve a single line of buffered information. To reduce the number of calls to the server, call the GET_LINES procedure to retrieve an array of lines from the buffer.
- You can choose to automatically display this information if you are using ttIsql by using the special SET SERVEROUTPUT ON command.
- After calling GET_LINE or GET_LINES, any lines not retrieved before the next call to PUT, PUT_LINE, or NEW_LINE are discarded to avoid confusing them with the next message.
GET_LINES procedure

This procedure retrieves an array of lines from the buffer.

Syntax

DBMS_OUTPUT.GET_LINES (  
  lines       OUT     DBMS_OUTPUT.CHARARR,  
  numlines    IN OUT  INTEGER);  

DBMS_OUTPUT.GET_LINES (  
  lines       OUT     DBMS_OUTPUT.DBS_OUTPUT_LINESARRAY,  
  numlines    IN OUT INTEGER);  

Parameters

Table 3–5 GET_LINES procedure parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lines</td>
<td>Returns an array of lines of buffered information. The maximum length of each line in the array is 32767 bytes. It is recommended that you use the VARRAY overload version in a 3GL host program to execute the procedure from a PL/SQL anonymous block.</td>
</tr>
<tr>
<td>numlines</td>
<td>Number of lines you want to retrieve from the buffer. After retrieving the specified number of lines, the procedure returns the number of lines actually retrieved. If this number is less than the number of lines requested, then there are no more lines in the buffer.</td>
</tr>
</tbody>
</table>

Usage notes

- You can choose to retrieve from the buffer a single line or an array of lines. Call the GET_LINE procedure to retrieve a single line of buffered information. To reduce the number of calls to the server, call the GET_LINES procedure to retrieve an array of lines from the buffer.

- You can choose to automatically display this information if you are using ttIsq1 by using the special SET SERVEROUTPUT ON command.

- After calling GET_LINE or GET_LINES, any lines not retrieved before the next call to PUT, PUT_LINE, or NEW_LINE are discarded to avoid confusing them with the next message.
NEW_LINE procedure

This procedure puts an end-of-line marker. The GET_LINE procedure and the GET_LINES procedure return "lines" as delimited by "newlines". Every call to the PUT_LINE procedure or NEW_LINE procedure generates a line that is returned by GET_LINE or GET_LINES.

Syntax

DBMS_OUTPUT.NEW_LINE;
PUT procedure

This procedure places a partial line in the buffer.

Note:  The PUT procedure that takes a NUMBER is obsolete. It is supported for legacy reasons only.

Syntax

```plsql
DBMS_OUTPUT.PUT (
   a IN VARCHAR2);
```

Parameters

<table>
<thead>
<tr>
<th>Table 3–6</th>
<th>PUT procedure parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter</td>
<td>Description</td>
</tr>
<tr>
<td>a</td>
<td>Item to buffer.</td>
</tr>
</tbody>
</table>

Exceptions

<table>
<thead>
<tr>
<th>Table 3–7</th>
<th>PUT procedure exceptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error</td>
<td>Description</td>
</tr>
<tr>
<td>ORA-20000, ORU-10027</td>
<td>Buffer overflow, limit of (&lt;\text{buf_limit})&gt; bytes.</td>
</tr>
<tr>
<td>ORA-20000, ORU-10028</td>
<td>Line length overflow, limit of 32767 bytes for each line.</td>
</tr>
</tbody>
</table>

Usage notes

- You can build a line of information piece by piece by making multiple calls to PUT, or place an entire line of information into the buffer by calling PUT_LINE.
- When you call PUT_LINE the item you specify is automatically followed by an end-of-line marker. If you make calls to PUT to build a line, then you must add your own end-of-line marker by calling NEW_LINE. GET_LINE and GET_LINES do not return lines that have not been terminated with a newline character.
- If your lines exceed the line limit, you receive an error message.
- Output that you create using PUT or PUT_LINE is buffered. The output cannot be retrieved until the PL/SQL program unit from which it was buffered returns to its caller.
PUT_LINE procedure

This procedure places a line in the buffer.

---

**Note:** The PUT_LINE procedure that takes a NUMBER is obsolete. It is supported for legacy reasons only.

---

**Syntax**

```sql
DBMS_OUTPUT.PUT_LINE (
    a IN VARCHAR2);
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Item to buffer.</td>
</tr>
</tbody>
</table>

**Exceptions**

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORA-20000, ORU-10027</td>
<td>Buffer overflow, limit of <code>&lt;buf_limit&gt;</code> bytes.</td>
</tr>
<tr>
<td>ORA-20000, ORU-10028</td>
<td>Line length overflow, limit of 32767 bytes for each line.</td>
</tr>
</tbody>
</table>

**Usage notes**

- You can build a line of information piece by piece by making multiple calls to PUT, or place an entire line of information into the buffer by calling PUT_LINE.
- When you call PUT_LINE the item you specify is automatically followed by an end-of-line marker. If you make calls to PUT to build a line, then you must add your own end-of-line marker by calling NEW_LINE. GET_LINE and GET_LINES do not return lines that have not been terminated with a newline character.
- If your lines exceeds the line limit, you receive an error message.
- Output that you create using PUT or PUT_LINE is buffered. The output cannot be retrieved until the PL/SQL program unit from which it was buffered returns to its caller.
DBMS_PREPROCESSOR

The DBMS_PREPROCESSOR package provides an interface to print or retrieve the source text of a PL/SQL unit in its post-processed form.

This package contains the following topics

- **Using DBMS_PREPROCESSOR**
  - Overview
  - Operating notes
- **Data structures**
  - Table types
- **Summary of DBMS_PREPROCESSOR subprograms**
Using DBMS_PREPROCESSOR

- Overview
- Operating notes
Overview

There are three styles of subprograms:

1. Subprograms that take a schema name, a unit type name, and the unit name.

2. Subprograms that take a VARCHAR2 string which contains the source text of an arbitrary PL/SQL compilation unit.

3. Subprograms that take a VARCHAR2 associative array (index-by table) which contains the segmented source text of an arbitrary PL/SQL compilation unit.

Subprograms of the first style are used to print or retrieve the post-processed source text of a stored PL/SQL unit. The user must have the privileges necessary to view the original source text of this unit. The user must also specify the schema in which the unit is defined, the type of the unit, and the name of the unit. If the schema is null, then the current user schema is used. If the status of the stored unit is VALID and the user has the required privilege, then the post-processed source text is guaranteed to be the same as that of the unit the last time it was compiled.

Subprograms of the second or third style are used to generate post-processed source text in the current user schema. The source text is passed in as a single VARCHAR2 string in the second style, or as a VARCHAR2 associative array in the third style. The source text can represent an arbitrary PL/SQL associative array in the third style. The source text can be useful when the source text exceeds the VARCHAR2 length limit.
Operating notes

- For subprograms of the first style, the status of the stored PL/SQL unit does not need to be VALID. Likewise, the source text passed in as a VARCHAR2 string or a VARCHAR2 associative array may contain compile time errors. If errors are found when generating the post-processed source, the error message text will also appear at the end of the post-processed source text. In some cases, the preprocessing can be aborted because of errors. When this happens, the post-processed source text will appear to be incomplete and the associated error message can help to indicate that an error has occurred during preprocessing.

- For subprograms of the second or third style, the source text can represent any arbitrary PL/SQL compilation unit. However, the source text of a valid PL/SQL compilation unit cannot include commonly used prefixes such as CREATE OR REPLACE. In general, the input source should be syntactically prepared in a way as if it were obtained from the ALL_SOURCE view. The following list gives some examples of valid initial syntax for some PL/SQL compilation units.

  anonymous block  (BEGIN | DECLARE) ...
  package          PACKAGE <name> ...
  package body     PACKAGE BODY <name> ...
  procedure        PROCEDURE <name> ...
  function         FUNCTION <name> ...

If the source text represents a named PL/SQL unit that is valid, that unit will not be created after its post-processed source text is generated.

- If the text of a wrapped PL/SQL unit is obtained from the ALL_SOURCE view, the keyword WRAPPED always immediately follows the name of the unit, as in this example:

  PROCEDURE "some proc" WRAPPED
  a000000
  b2
  ...

If such source text is presented to one of the GET_POST_PROCESSED_SOURCE function or to one of the PRINT_POST_PROCESSED_SOURCE procedure, the exception DBMS_PREPROCESSOR.WRAPPED_INPUT is raised.
Data structures

The DBMS_PREPROCESSOR package defines a table type.

Note: Notes on data types:

- The PLS_INTEGER and BINARY_INTEGER data types are identical. This document uses "BINARY_INTEGER" to indicate data types in reference information (such as for table types, record types, subprogram parameters, or subprogram return values), but may use either in discussion and examples.

- The INTEGER and NUMBER(38) data types are also identical. This document uses "INTEGER" throughout.

Table types

SOURCE_LINES_T table type
SOURCE_LINES_T table type

This table type stores lines of post-processed source text. It is used to hold PL/SQL source text both before and after it is processed. It is especially useful in cases in which the amount of text exceeds 32K.

Syntax

```sql
TYPE source_lines_t IS
  TABLE OF VARCHAR2(32767) INDEX BY BINARY_INTEGER;
```
Summary of DBMS_PREPROCESSOR subprograms

Table 4–1 DBMS_PREPROCESSOR package subprograms

<table>
<thead>
<tr>
<th>Subprogram</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET_POST_PROCESSED_SOURCE function</td>
<td>Returns the post-processed source text.</td>
</tr>
<tr>
<td>PRINT_POST_PROCESSED_SOURCE procedure</td>
<td>Prints post-processed source text.</td>
</tr>
</tbody>
</table>
GET_POST_PROCESSED_SOURCE function

This overloaded function returns the post-processed source text. The different functionality of each form of syntax is presented along with the definition.

Syntax

Returns post-processed source text of a stored PL/SQL unit:

```sql
DBMS_PREPROCESSOR.GET_POST_PROCESSED_SOURCE (
    object_type    IN VARCHAR2,
    schema_name    IN VARCHAR2,
    object_name    IN VARCHAR2)
RETURN dbms_preprocessor.source_lines_t;
```

Returns post-processed source text of a compilation unit:

```sql
DBMS_PREPROCESSOR.GET_POST_PROCESSED_SOURCE (
    source        IN VARCHAR2)
RETURN dbms_preprocessor.source_lines_t;
```

Returns post-processed source text of an associative array (index-by table) containing the source text of the compilation unit:

```sql
DBMS_PREPROCESSOR.GET_POST_PROCESSED_SOURCE (
    source        IN dbms_preprocessor.source_lines_t)
RETURN dbms_preprocessor.source_lines_t;
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object_type</td>
<td>Must be one of PACKAGE, PACKAGE BODY, PROCEDURE, or FUNCTION. Case sensitive.</td>
</tr>
<tr>
<td>schema_name</td>
<td>The schema name. Case insensitive unless a quoted identifier is used. If NULL, use current schema.</td>
</tr>
<tr>
<td>object_name</td>
<td>The name of the object. The object_type is always case insensitive. Case insensitive unless a quoted identifier is used.</td>
</tr>
<tr>
<td>source</td>
<td>The source text of the compilation unit.</td>
</tr>
<tr>
<td>source_lines_t</td>
<td>Associative array containing the source text of the compilation unit. The source text is a concatenation of all the non-NULL associative array elements in ascending index order.</td>
</tr>
</tbody>
</table>

Return values

The function returns an associative array containing the lines of the post-processed source text starting from index 1.

Usage notes

- Newline characters are not removed.
- Each line in the post-processed source text is mapped to a row in the associative array.
- In the post-processed source, unselected text will have blank lines.
Exceptions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORA-24234</td>
<td>Insufficient privileges or object does not exist.</td>
</tr>
<tr>
<td>ORA-24235</td>
<td>Bad value for object type. Should be one of PACKAGE, PACKAGE BODY, PROCEDURE, or FUNCTION.</td>
</tr>
<tr>
<td>ORA-24236</td>
<td>The source text is empty.</td>
</tr>
<tr>
<td>ORA-00931</td>
<td>Missing identifier. The object_name should not be NULL.</td>
</tr>
<tr>
<td>ORA-06502</td>
<td>Numeric or value error:</td>
</tr>
<tr>
<td></td>
<td>Character string buffer too small.</td>
</tr>
<tr>
<td></td>
<td>A line is too long (more than 32767 bytes).</td>
</tr>
</tbody>
</table>
PRINT_POST_PROCESSED_SOURCE procedure

This overloaded procedure calls DBMS_OUTPUT.PUT_LINE to let you view post-processed source text. The different functionality of each form of syntax is presented along with the definition.

Syntax

Prints post-processed source text of a stored PL/SQL unit:

```sql
DBMS_PREPROCESSOR.PRINT_POST_PROCESSED_SOURCE (
    object_type    IN VARCHAR2,
    schema_name    IN VARCHAR2,
    object_name    IN VARCHAR2);
```

Prints post-processed source text of a compilation unit:

```sql
DBMS_PREPROCESSOR.PRINT_POST_PROCESSED_SOURCE (
    source        IN VARCHAR2);
```

Prints post-processed source text of an associative array containing the source text of the compilation unit:

```sql
DBMS_PREPROCESSOR.PRINT_POST_PROCESSED_SOURCE (
    source        IN dbms_preprocessor.source_lines_t);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object_type</td>
<td>Must be one of PACKAGE, PACKAGE BODY, PROCEDURE, or FUNCTION. Case sensitive.</td>
</tr>
<tr>
<td>schema_name</td>
<td>The schema name. Case insensitive unless a quoted identifier is used. If NULL, use current schema.</td>
</tr>
<tr>
<td>object_name</td>
<td>The name of the object. The object_type is always case insensitive. Case insensitive unless a quoted identifier is used.</td>
</tr>
<tr>
<td>source</td>
<td>The source text of the compilation unit.</td>
</tr>
<tr>
<td>source_lines_t</td>
<td>Associative array containing the source text of the compilation unit. The source text is a concatenation of all the non-NULL associative array elements in ascending index order.</td>
</tr>
</tbody>
</table>

Exceptions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORA-24234</td>
<td>Insufficient privileges or object does not exist.</td>
</tr>
<tr>
<td>ORA-24235</td>
<td>Bad value for object type. Should be one of PACKAGE, PACKAGE BODY, PROCEDURE, or FUNCTION.</td>
</tr>
<tr>
<td>ORA-24236</td>
<td>The source text is empty.</td>
</tr>
<tr>
<td>ORA-00931</td>
<td>Missing identifier. The object_name should not be NULL.</td>
</tr>
</tbody>
</table>
Usage notes

The associative array may contain holes. NULL elements are ignored when doing the concatenation.

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORA-06502</td>
<td>Numeric or value error:</td>
</tr>
<tr>
<td></td>
<td>- Character string buffer too small.</td>
</tr>
<tr>
<td></td>
<td>- A line is too long (more than 32767 bytes).</td>
</tr>
</tbody>
</table>
The DBMS_RANDOM package provides a built-in random number generator. DBMS_RANDOM is not intended for cryptography.

This chapter contains the following topics:

- Using DBMS_RANDOM
  - Operational notes
- Summary of DBMS_RANDOM subprograms
Using DBMS_RANDOM

- Operational notes
Operational notes

- DBMS_RANDOM.RANDOM produces integers in \([-2^{31}, 2^{31})\).
- DBMS_RANDOM.VALUE produces numbers in \([0,1)\) with 38 digits of precision.

DBMS_RANDOM can be explicitly initialized, but does not need to be initialized before calling the random number generator. It will automatically initialize with the date, user ID, and process ID if no explicit initialization is performed.

If this package is seeded twice with the same seed, then accessed in the same way, it will produce the same results in both cases.

In some cases, such as when testing, you may want the sequence of random numbers to be the same on every run. In that case, you seed the generator with a constant value by calling one of the overloads of DBMS_RANDOM.SEED. To produce different output for every run, simply omit the call to “Seed” and the system will choose a suitable seed for you.
## Summary of DBMS_RANDOM subprograms

<table>
<thead>
<tr>
<th>Subprogram</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INITIALIZE</td>
<td>Initializes the package with a seed value.</td>
</tr>
<tr>
<td>NORMAL</td>
<td>Returns random numbers in a normal distribution.</td>
</tr>
<tr>
<td>RANDOM</td>
<td>Generates a random number.</td>
</tr>
<tr>
<td>SEED</td>
<td>Resets the seed.</td>
</tr>
<tr>
<td>STRING</td>
<td>Gets a random string.</td>
</tr>
<tr>
<td>TERMINATE</td>
<td>Terminates package.</td>
</tr>
<tr>
<td>VALUE</td>
<td>This 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), while the overloaded function gets a random Oracle number ( x ), where ( x ) is greater than or equal to ( low ) and less than ( high ).</td>
</tr>
</tbody>
</table>

**Note:** The INITIALIZE procedure, RANDOM function and the TERMINATE procedure are all obsolete and, while currently supported, are included in this release for legacy reasons only.

**Note:** Notes on data types:

- The PLS_INTEGER and BINARY_INTEGER data types are identical. This document uses "BINARY_INTEGER" to indicate data types in reference information (such as for table types, record types, subprogram parameters, or subprogram return values), but may use either in discussion and examples.

- The INTEGER and NUMBER(38) data types are also identical. This document uses "INTEGER" throughout.
INITIALIZE procedure

This procedure initializes the generator (but see the usage notes).

Syntax

```plaintext
DBMS_RANDOM.INITIALIZE (  
    val  IN  BINARY_INTEGER);
```

Pragmas

```plaintext
PRAGMA restrict_references (initialize, WNDS);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>val</td>
<td>The seed number used to generate a random number.</td>
</tr>
</tbody>
</table>

Usage notes

This procedure is obsolete as it simply calls the SEED procedure on page 5-8.
NORMAL function

This function returns random numbers in a standard normal distribution.

Syntax

```sql
DBMS_RANDOM.NORMAL
RETURN NUMBER;
```

Pragmas

`PRAGMA restrict_references (normal, WNDS);`

Return value

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>A random number.</td>
</tr>
</tbody>
</table>
RANDOM function

This procedure generates and returns a random number.

Syntax

```
DBMS_RANDOM.RANDOM
    RETURN binary_integer;
```

Pragmas

```
PRAGMA restrict_references (random, WNDs);
```

Return value

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary_integer</td>
<td>A random integer greater or equal to -power(2,31) and less than power(2,31).</td>
</tr>
</tbody>
</table>
SEED procedure

This procedure resets the seed.

Syntax

```sql
DBMS_RANDOM.SEED {
    val IN BINARY_INTEGER};

DBMS_RANDOM.SEED {
    val IN VARCHAR2};
```

Pragmas

```sql
PRAGMA restrict_references {seed, WNDS};
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>val</td>
<td>Seed number or string used to generate a random number.</td>
</tr>
</tbody>
</table>

Usage notes

The seed can be a string up to length 2000.
STRING function

This function generates and returns a random string.

**Syntax**

```sql
DBMS_RANDOM.STRING
  opt IN CHAR,
  len IN NUMBER)
RETURN VARCHAR2;
```

**Pragmas**

```sql
PRAGMA restrict_references (string, WNDS);
```

**Parameters**

**Table 5–6  STRING function parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>opt</td>
<td>Specifies what the returning string looks like:</td>
</tr>
<tr>
<td></td>
<td>- 'u', 'U' - returning string in uppercase alpha characters.</td>
</tr>
<tr>
<td></td>
<td>- 'l', 'L' - returning string in lowercase alpha characters.</td>
</tr>
<tr>
<td></td>
<td>- 'a', 'A' - returning string in mixed case alpha characters.</td>
</tr>
<tr>
<td></td>
<td>- 'x', 'X' - returning string in uppercase alpha-numeric characters.</td>
</tr>
<tr>
<td></td>
<td>- 'p', 'P' - returning string in any printable characters.</td>
</tr>
<tr>
<td></td>
<td>Otherwise the returning string is in uppercase alpha characters.</td>
</tr>
<tr>
<td>len</td>
<td>The length of the returning string.</td>
</tr>
</tbody>
</table>

**Return value**

**Table 5–7  STRING function return value**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VARCHAR2</td>
<td>A VARCHAR2 value.</td>
</tr>
</tbody>
</table>
**TERMINATE procedure**

When you are finished with the package, call the TERMINATE procedure (but see Usage notes)

**Syntax**

```
DBMS_RANDOM.TERMINATE;
```

**Usage notes**

This procedure performs no function and, although it is currently supported, it is obsolete and should not be used.
VALUE function

The basic function gets a random number, greater than or equal to 0 and less than 1, with 38 digits to the right of the decimal (38-digit precision). Alternatively, you can get a random Oracle number \( x \), where \( x \) is greater than or equal to \( \text{low} \) and less than \( \text{high} \).

Syntax

```sql
DBMS_RANDOM.VALUE
    RETURN NUMBER;

DBMS_RANDOM.VALUE(
    low  IN  NUMBER,
    high IN  NUMBER)
    RETURN NUMBER;
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>low</td>
<td>The lowest number in a range from which to generate a random number. The number generated may be equal to ( \text{low} ).</td>
</tr>
<tr>
<td>high</td>
<td>The highest number below which to generate a random number. The number generated will be less than ( \text{high} ).</td>
</tr>
</tbody>
</table>

Return value

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUMBER</td>
<td>An Oracle NUMBER value.</td>
</tr>
</tbody>
</table>
The DBMS_SQL package provides an interface to use dynamic SQL to parse any data manipulation language (DML) or data definition language (DDL) statement using PL/SQL. For example, you can enter a DROP TABLE statement from within a stored procedure by using the PARSE procedure supplied with the DBMS_SQL package.

This chapter contains the following topics:

- **Using DBMS_SQL**
  - Overview
  - Security model
  - Constants
  - Exceptions
  - Operational notes
  - Examples

- **Data structures**
  - Record types
  - Table types

- **Summary of DBMS_SQL subprograms**

**Note:** For more information on native dynamic SQL, see "Dynamic SQL in PL/SQL (EXECUTE IMMEDIATE statement)" in *Oracle TimesTen In-Memory Database PL/SQL Developer’s Guide* or see *Oracle Database PL/SQL Language Reference.*
Using DBMS_SQL

- Overview
- Security model
- Constants
- Exceptions
- Operational notes
- Examples
Overview

TimesTen lets you write stored procedures and anonymous PL/SQL blocks that use dynamic SQL. Dynamic SQL statements are not embedded in your source program; rather, they are stored in character strings that are input to, or built by, the program at runtime. This enables you to create more general-purpose procedures. For example, dynamic SQL lets you create a procedure that operates on a table whose name is not known until runtime.

Native dynamic SQL is an alternative to DBMS_SQL that lets you place dynamic SQL statements directly into PL/SQL blocks. In most situations, native dynamic SQL is easier to use and performs better than DBMS_SQL. However, native dynamic SQL itself has certain limitations:

- There is no support for so-called Method 4 (for dynamic SQL statements with an unknown number of inputs or outputs).

Also, there are some tasks that can only be performed using DBMS_SQL.

The ability to use dynamic SQL from within stored procedures generally follows the model of the Oracle Call Interface (OCI). See Oracle Call Interface Programmer’s Guide for information about OCI.

PL/SQL differs somewhat from other common programming languages, such as C. For example, addresses (also called pointers) are not user-visible in PL/SQL. As a result, there are some differences between OCI and the DBMS_SQL package. These differences include the following:

- The OCI uses bind by address, while the DBMS_SQL package uses bind by value.
- With DBMS_SQL you must call VARIABLE_VALUE to retrieve the value of an OUT parameter for an anonymous block, and you must call COLUMN_VALUE after fetching rows to actually retrieve the values of the columns in the rows into your program.
- The current release of the DBMS_SQL package does not provide CANCEL cursor procedures.
- Indicator variables are not required, because NULLs are fully supported as values of a PL/SQL variable.
Security model

DBMS_SQL is a SYS-owned package compiled with AUTHID CURRENT_USER. Any DBMS_SQL subprogram called from an anonymous PL/SQL block is run using the privileges of the current user.

See "Definer’s rights and invoker’s rights" in Oracle TimesTen In-Memory Database PL/SQL Developer’s Guide for information about the AUTHID clause.

Preventing malicious or accidental access of open cursor numbers
An error, ORA-29471, is raised when any DBMS_SQL subprogram is called with a cursor number that does not denote an open cursor. When the error is raised, an alert is issued to the alert log and DBMS_SQL becomes inoperable for the life of the session.

If the actual value for the cursor number in a call to the IS_OPEN function denotes a cursor currently open in the session, the return value is TRUE. If the actual value is NULL, then the return value is FALSE. Otherwise, this raises an ORA-29471 error.

Note that the OPEN_CURSOR function is the only DBMS_SQL subprogram that has no formal parameter for the cursor number; rather, it returns a cursor number. Therefore it is not within the scope of these rules.

Preventing inappropriate use of a cursor
Cursors are protected from security breaches that subvert known existing cursors.

Checks are made when binding and executing. Optionally, checks may be performed for every single DBMS_SQL subprogram call. The check is as follows:

- The current_user is the same on calling the subprogram as it was on calling the most recent parse.
- The enabled roles on calling the subprogram must be a superset of the enabled roles on calling the most recent parse.

Consistent with the use of definer’s rights subprograms, roles do not apply.

If either check fails, an ORA-29470 error is raised.

The mechanism for defining when checks are performed is a new overload for the OPEN_CURSOR subprogram which takes a formal parameter, security_level, with allowed values NULL, 1, and 2.

- When security_level = 1 (or is NULL), the checks are made only when binding and executing.
- When security_level = 2, the checks are always made.
## Constants

The constants described in Table 6–1 are used with the `language_flag` parameter of the PARSE procedure. For TimesTen, use NATIVE.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>V6</td>
<td>INTEGER</td>
<td>0</td>
<td>Specifies Oracle database version 6 behavior (not applicable for TimesTen).</td>
</tr>
<tr>
<td>NATIVE</td>
<td>INTEGER</td>
<td>1</td>
<td>Specifies normal behavior for the database to which the program is connected.</td>
</tr>
<tr>
<td>V7</td>
<td>INTEGER</td>
<td>2</td>
<td>Specifies Oracle database version 7 behavior (not applicable for TimesTen).</td>
</tr>
</tbody>
</table>
Exceptions

inconsistent_type EXCEPTION;
pragma exception_init(inconsistent_type, -6562);

This exception is raised by the COLUMN_VALUE procedure or the VARIABLE_VALUE procedure when the type of the given OUT parameter (for where to put the requested value) is different from the type of the value.
Operational notes

- Execution flow
- Processing queries
- Processing updates, inserts, and deletes
- Locating errors

Execution flow

1. OPEN_CURSOR
2. PARSE
3. BIND_VARIABLE or BIND_ARRAY
4. DEFINE_COLUMN or DEFINE_ARRAY
5. EXECUTE
6. FETCH_ROWS or EXECUTE_AND_FETCH
7. VARIABLE_VALUE or COLUMN_VALUE
8. CLOSE_CURSOR

OPEN_CURSOR
To process a SQL statement, you must have an open cursor. When you call the OPEN_CURSOR function, you receive a cursor ID number for the data structure representing a valid cursor maintained by TimesTen. These cursors are distinct from cursors defined at the precompiler, OCI, or PL/SQL level, and are used only by the DBMS_SQL package.

PARSE
Every SQL statement must be parsed by calling the PARSE procedure. Parsing the statement checks the statement's syntax and associates it with the cursor in your program.

You can parse any DML or DDL statement. DDL statements are run on the parse, which performs the implied commit.

---

**Note:** When parsing a DDL statement to drop a procedure or a package, a deadlock can occur if you’re still using the procedure in question or a procedure in the package in question. After a call to a procedure, that procedure is considered to be in use until execution has returned to the user side. Any such deadlock times out after a short time.

---

The execution flow of DBMS_SQL is shown in Figure 6–1.
**BIND_VARIABLE or BIND_ARRAY**

Many DML statements require that data in your program be input to TimesTen. When you define a SQL statement that contains input data to be supplied at runtime, you must use placeholders in the SQL statement to mark where data must be supplied.

For each placeholder in the SQL statement, you must call one of the bind procedures, the **BIND_ARRAY procedure** on page 6-43 or the **BIND_VARIABLE procedure** on page 6-45, to supply the value of a variable in your program (or the values of an array) to the placeholder. When the SQL statement is subsequently run, TimesTen uses the data that your program has placed in the output and input, or bind, variables.

DBMS_SQL can run a DML statement multiple times — each time with a different bind variable. The **BIND_ARRAY procedure** lets you bind a collection of scalars, each value of which is used as an input variable once for each EXECUTE. This is similar to the array interface supported by the OCI.

---

**Figure 6–1  DBMS_SQL execution flow**

- `open_cursor`
- `PARSE`
- `Use bind variables?`
  - `no`
  - `yes`
    - `bind_variable`
    - `query?`
      - `no`
      - `EXECUTE` (with options)
    - `yes`
      - `PL/SQL block?`
        - `no`
        - `EXECUTE` (with options)
        - `fetch_rows`
        - `column_value`
        - `variable_value`
        - `close_cursor`
        - `define_column`
        - `variable_value`
        - `variable_value`
**DEFINE_COLUMN or DEFINE_ARRAY**
The columns of the row being selected in a SELECT statement are identified by their relative positions as they appear in the select list, from left to right. For a query, you must call one of the define procedures (DEFINE_COLUMN or DEFINE_ARRAY) to specify the variables that are to receive the SELECT values, much the way an INTO clause does for a static query.

Use the DEFINE_ARRAY procedure to define a PL/SQL collection into which you want to fetch rows in a single SELECT statement. DEFINE_ARRAY provides an interface to fetch multiple rows at one fetch. You must call DEFINE_ARRAY before using the COLUMN_VALUE procedure to fetch the rows.

**EXECUTE**
Call the EXECUTE function to run your SQL statement.

**FETCH_ROWS or EXECUTE_AND_FETCH**
The FETCH_ROWS function retrieves the rows that satisfy the query. Each successive fetch retrieves another set of rows, until the fetch is unable to retrieve anymore rows. Instead of calling EXECUTE and then FETCH_ROWS, you may find it more efficient to call EXECUTE_AND_FETCH if you are calling EXECUTE for a single execution.

**VARIABLE_VALUE or COLUMN_VALUE**
For queries, call COLUMN_VALUE to determine the value of a column retrieved by the FETCH_ROWS call. For anonymous blocks containing calls to PL/SQL procedures or DML statements with RETURNING clause, call VARIABLE_VALUE to retrieve the values assigned to the output variables when statements were run.

**CLOSE_CURSOR**
When you no longer need a cursor for a session, close the cursor by calling CLOSE_CURSOR.

If you neglect to close a cursor, then the memory used by that cursor remains allocated even though it is no longer needed.

**Processing queries**
If you are using dynamic SQL to process a query, then you must perform the following steps:

1. Specify the variables that are to receive the values returned by the SELECT statement by calling the DEFINE_COLUMN procedure or the DEFINE_ARRAY procedure.
2. Run your SELECT statement by calling the EXECUTE function.
3. Call the FETCH_ROWS function (or EXECUTE_AND_FETCH) to retrieve the rows that satisfied your query.
4. Call COLUMN_VALUE procedure to determine the value of a column retrieved by the FETCH_ROWS function for your query. If you used anonymous blocks containing calls to PL/SQL procedures, then you must call the VARIABLE_VALUE procedure to retrieve the values assigned to the output variables of these procedures.

**Processing updates, inserts, and deletes**
If you are using dynamic SQL to process an INSERT, UPDATE, or DELETE, then you must perform the following steps.
1. You must first run your INSERT, UPDATE, or DELETE statement by calling the `EXECUTE` function.

2. If statements have the RETURNING clause, then you must call the `VARIABLE_VALUE` procedure to retrieve the values assigned to the output variables.

Locating errors

There are additional functions in the DBMS_SQL package for obtaining information about the last referenced cursor in the session. The values returned by these functions are only meaningful immediately after a SQL statement is run. In addition, some error-locating functions are only meaningful after certain DBMS_SQL calls. For example, you call the `LAST_ERROR_POSITION` function immediately after a `PARSE`. 
Examples

This section provides example procedures that make use of the DBMS_SQL package.

Example 1
This example does not require the use of dynamic SQL because the text of the statement is known at compile time, but it illustrates the basic concept underlying the package.

The demo procedure deletes all employees from a table myemployees (created from the employees table of the HR schema) whose salaries exceed a specified value.

```sql
CREATE OR REPLACE PROCEDURE demo(p_salary IN NUMBER) AS
    cursor_name INTEGER;
    rows_processed INTEGER;
BEGIN
    cursor_name := dbms_sql.open_cursor;
    DBMS_SQL.PARSE(cursor_name, 'DELETE FROM myemployees WHERE salary > :x', DBMS_SQL.NATIVE);
    DBMS_SQL.BIND_VARIABLE(cursor_name, ':x', p_salary);
    rows_processed := DBMS_SQL.EXECUTE(cursor_name);
    DBMS_SQL.CLOSE_CURSOR(cursor_name);
EXCEPTION
    WHEN OTHERS THEN
        DBMS_SQL.CLOSE_CURSOR(cursor_name);
END;
/
```

Create the myemployees table and see how many employees have salaries greater than or equal to $15,000:

Command> create table myemployees as select * from employees;
107 rows inserted.

Command> select * from myemployees where salary>=15000;
< 100, Steven, King, SKING, 515.123.4567, 1987-06-17 00:00:00, AD_PRES, 24000, <NULL>, <NULL>, 90 >
< 101, Neena, Kochhar, NKOCHHAR, 515.123.4568, 1989-09-21 00:00:00, AD_VP, 17000, <NULL>, 100, 90 >
< 102, Lex, De Haan, LDEHAAN, 515.123.4569, 1993-01-13 00:00:00, AD_VP, 17000, <NULL>, 100, 90 >
3 rows found.

Run demo to delete everyone with a salary greater than $14,999 and confirm the results:

Command> begin
> demo(14999);
> end;
> /
PL/SQL procedure successfully completed.

Command> select * from myemployees where salary>=15000;
0 rows found.

Example 2
The following sample procedure is passed the names of a source and a destination table, and copies the rows from the source table to the destination table. This sample
procedure assumes that both the source and destination tables have the following columns:

- **id** of type NUMBER
- **name** of type VARCHAR2(30)
- **birthdate** of type DATE

This procedure does not specifically require the use of dynamic SQL; however, it illustrates the concepts of this package.

```sql
CREATE OR REPLACE PROCEDURE copy (source IN VARCHAR2, destination IN VARCHAR2) IS
    id_var             NUMBER;
    name_var           VARCHAR2(30);
    birthdate_var      DATE;
    source_cursor      INTEGER;
    destination_cursor INTEGER;
    ignore             INTEGER;
BEGIN
    -- Prepare a cursor to select from the source table:
    source_cursor := DBMS_SQL.open_cursor;
    DBMS_SQL.PARSE(source_cursor,
        'SELECT id, name, birthdate FROM ' || source,
        DBMS_SQL.NATIVE);
    DBMS_SQL.DEFINE_COLUMN(source_cursor, 1, id_var);
    DBMS_SQL.DEFINE_COLUMN(source_cursor, 2, name_var, 30);
    DBMS_SQL.DEFINE_COLUMN(source_cursor, 3, birthdate_var);
    ignore := DBMS_SQL.EXECUTE(source_cursor);

    -- Prepare a cursor to insert into the destination table:
    destination_cursor := DBMS_SQL.OPEN_CURSOR;
    DBMS_SQL.PARSE(destination_cursor,
        'INSERT INTO ' || destination ||
        ' VALUES (:id_bind, :name_bind, :birthdate_bind)',
        DBMS_SQL.NATIVE);

    -- Fetch a row from the source table and insert it into the destination table:
    LOOP
        IF DBMS_SQL.FETCH_ROWS(source_cursor)>0 THEN
            -- get column values of the row
            DBMS_SQL.COLUMN_VALUE(source_cursor, 1, id_var);
            DBMS_SQL.COLUMN_VALUE(source_cursor, 2, name_var);
            DBMS_SQL.COLUMN_VALUE(source_cursor, 3, birthdate_var);

            -- Bind the row into the cursor that inserts into the destination table. You could alter this example to require the use of dynamic SQL by inserting an if condition before the bind.
            DBMS_SQL.BIND_VARIABLE(destination_cursor, ':id_bind', id_var);
            DBMS_SQL.BIND_VARIABLE(destination_cursor, ':name_bind', name_var);
            DBMS_SQL.BIND_VARIABLE(destination_cursor, ':birthdate_bind', birthdate_var);
            ignore := DBMS_SQL.EXECUTE(destination_cursor);
        ELSE
            -- No more rows to copy:
            EXIT;
        END IF;
    END LOOP;
END;
```
-- Commit (in TimesTen commit closes cursors automatically):
  COMMIT;

EXCEPTION
  WHEN OTHERS THEN
    IF DBMS_SQL.IS_OPEN(source_cursor) THEN
      DBMS_SQL.CLOSE_CURSOR(source_cursor);
    END IF;
    IF DBMS_SQL.IS_OPEN(destination_cursor) THEN
      DBMS_SQL.CLOSE_CURSOR(destination_cursor);
    END IF;
    RAISE;
  END;
/

Examples 3, 4, and 5: Bulk DML

This series of examples shows how to use bulk array binds (table items) in the SQL
DML statements INSERT, UPDATE, and DELETE.

Here is an example of a bulk INSERT statement that adds three new departments to
the departments table in the HR schema:

DECLARE
  stmt VARCHAR2(200);
  departid_array     DBMS_SQL.NUMBER_TABLE;
  deptname_array     DBMS_SQL.VARCHAR2_TABLE;
  mgrid_array        DBMS_SQL.NUMBER_TABLE;
  locid_array        DBMS_SQL.NUMBER_TABLE;
  c                NUMBER;
  dummy            NUMBER;
BEGIN
  departid_array(1):= 280;
  departid_array(2):= 290;
  departid_array(3):= 300;

  deptname_array(1) := 'Community Outreach';
  deptname_array(2) := 'Product Management';
  deptname_array(3) := 'Acquisitions';

  mgrid_array(1) := 121;
  mgrid_array(2) := 120;
  mgrid_array(3) := 70;

  locid_array(1):= 1500;
  locid_array(2):= 1700;
  locid_array(3):= 2700;

  stmt := 'INSERT INTO departments VALUES(
    :departid_array, :deptname_array, :mgrid_array, :locid_array)';
  c := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(c, stmt, DBMS_SQL.NATIVE);
  DBMS_SQL.BIND_ARRAY(c, ':departid_array', departid_array);
  DBMS_SQL.BIND_ARRAY(c, ':deptname_array', deptname_array);
  DBMS_SQL.BIND_ARRAY(c, ':mgrid_array', mgrid_array);
  DBMS_SQL.BIND_ARRAY(c, ':locid_array', locid_array);
  dummy := DBMS_SQL.EXECUTE(c);
  DBMS_SQL.CLOSE_CURSOR(c);
EXCEPTION WHEN OTHERS THEN
  IF DBMS_SQL.IS_OPEN(c) THEN
    DBMS_SQL.CLOSE_CURSOR(c);
  END IF;
END;
END IF;
RAISE;
END;
/

Here is output from a SELECT statement, showing the new rows:

Command> select * from departments;
< 10, Administration, 200, 1700 >
...< 280, Community Outreach, 121, 1500 >
< 290, Product Management, 120, 1700 >
< 300, Acquisitions, 70, 2700 >
30 rows found.

Here is an example of a bulk UPDATE statement that demonstrates updating salaries for four existing employees in the employees table in the HR schema:

DECLARE
stmt VARCHAR2(200);
empno_array     DBMS_SQL.NUMBER_TABLE;
salary_array    DBMS_SQL.NUMBER_TABLE;
c               NUMBER;
dummy           NUMBER;
BEGIN

empno_array(1):= 203;
empno_array(2):= 204;
empno_array(3):= 205;
empno_array(4):= 206;
salary_array(1) := 7000;
salary_array(2) := 11000;
salary_array(3) := 13000;
salary_array(4) := 9000;
stmt := 'update employees set salary = :salary_array
   WHERE employee_id = :num_array';
c := DBMS_SQL.OPEN_CURSOR;
DBMS_SQL.PARSE(c, stmt, DBMS_SQL.NATIVE);
DBMS_SQL.BIND_ARRAY(c, ':num_array', empno_array);
DBMS_SQL.BIND_ARRAY(c, ':salary_array', salary_array);
dummy := DBMS_SQL.EXECUTE(c);
DBMS_SQL.CLOSE_CURSOR(c);
EXCEPTION WHEN OTHERS THEN
   IF DBMS_SQL.IS_OPEN(c) THEN
      DBMS_SQL.CLOSE_CURSOR(c);
   END IF;
   RAISE;
END;
/

Assume the following entries for the specified employees before running the example, showing salaries of $6500, $10000, $12000, and $8300:

Command> select * from employees where employee_id>=203 and employee_id<=206;
< 203, Susan, Mavris, SNAVRIS, 515.123.7777, 1994-06-07 00:00:00, HR_REP, 6500, <NULL>, 101, 40 >
< 204, Hermann, Baer, HBAER, 515.123.8888, 1994-06-07 00:00:00, PR_REP, 10000, <NULL>, 101, 70 >
Using DBMS_SQL

The following shows the new salaries after running the example:

Command> select * from employees where employee_id>=203 and employee_id<=206;
< 203, Susan, Mavris, SMAVRIS, 515.123.7777, 1994-06-07 00:00:00, HR_REP,
7000, <NULL>, 101, 60 >
< 204, Hermann, Baer, HBAER, 515.123.8888, 1994-06-07 00:00:00, PR_REP,
11000, <NULL>, 101, 70 >
< 205, Shelley, Higgins, SHIGGINS, 515.123.8080, 1994-06-07 00:00:00, AC_MGR,
13000, <NULL>, 101, 110 >
< 206, William, Gietz, WGIETZ, 515.123.8181, 1994-06-07 00:00:00, AC_ACCOUNT,
9000, <NULL>, 205, 110 >
4 rows found.

In a DELETE statement, for example, you could bind in an array in the WHERE clause
and have the statement be run for each element in the array, as in the following example:

DECLARE
   stmt VARCHAR2(200);
   dept_no_array DBMS_SQL.NUMBER_TABLE;
   c NUMBER;
   dummy NUMBER;
begin
   dept_no_array(1) := 60;
   dept_no_array(2) := 70;
   stmt := 'delete from employees where department_id = :dept_array';
   c := DBMS_SQL.OPEN_CURSOR;
   DBMS_SQL.PARSE(c, stmt, DBMS_SQL.NATIVE);
   DBMS_SQL.BIND_ARRAY(c, ':dept_array', dept_no_array, 1, 1);
   dummy := DBMS_SQL.EXECUTE(c);
   DBMS_SQL.CLOSE_CURSOR(c);
EXCEPTION WHEN OTHERS THEN
   IF DBMS_SQL.IS_OPEN(c) THEN
      DBMS_SQL.CLOSE_CURSOR(c);
   END IF;
   RAISE;
END;
/

In this example, only the first element of the array is specified by the BIND_ARRAY
call (lower and upper bounds of the array elements are both set to 1), so only
employees in department 60 are deleted.

Before running the example, there are five employees in department 60 and one in
department 70, where the department number is the last entry in each row:

Command> select * from employees where department_id>=60 and department_id<=70;
< 103, Alexander, Hunold, AHUNOLD, 590.423.4567, 1990-01-03 00:00:00, IT_PROG,
9000, <NULL>, 102, 60 >
< 104, Bruce, Ernst, BERNST, 590.423.4568, 1991-05-21 00:00:00, IT_PROG, 6000,
<NULL>, 103, 60 >
< 105, David, Austin, DAUSTIN, 590.423.4569, 1997-06-25 00:00:00, IT_PROG, 4800,
<NULL>, 103, 60 >
< 106, Valli, Pataballa, VPATABAL, 590.423.4560, 1998-02-05 00:00:00, IT_PROG,
After running the example, only the employee in department 70 remains:

Command> select * from employees where department_id>=60 and department_id<=70;

1 row found.

Example 6: Defining an array

```
CREATE OR REPLACE PROCEDURE BULK_PLSQL(deptid NUMBER) IS
  names    DBMS_SQL.VARCHAR2_TABLE;
  sals     DBMS_SQL.NUMBER_TABLE;
  c        NUMBER;
  r        NUMBER;
  sql_stmt VARCHAR2(32767) :=
    'SELECT last_name, salary FROM employees WHERE department_id = :b1';
BEGIN
  c := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(c, sql_stmt, dbms_sql.native);
  DBMS_SQL.BIND_VARIABLE(c, 'b1', deptid);
  DBMS_SQL.DEFINE_ARRAY(c, 1, names, 5, 1);
  DBMS_SQL.DEFINE_ARRAY(c, 2, sals, 5, 1);
  r := DBMS_SQL.EXECUTE(c);
  LOOP
    r := DBMS_SQL.FETCH_ROWS(c);
    DBMS_SQL.COLUMN_VALUE(c, 1, names);
    DBMS_SQL.COLUMN_VALUE(c, 2, sals);
    EXIT WHEN r != 5;
  END LOOP;
  DBMS_SQL.CLOSE_CURSOR(c);
  -- loop through the names and sals collections
  FOR i IN names.FIRST .. names.LAST LOOP
    DBMS_OUTPUT.PUT_LINE('Name = ' || names(i) || ', salary = ' || sals(i));
  END LOOP;
END;
/
```

For example, for Department 20 in the employees table, this produces the following output:

Command> begin
  > bulk_plsql(20);
  > end;
  > /
Name = Hartstein, salary = 13000
Name = Fay, salary = 6000
Example 7: Describe columns

This can be used as a substitute for the ttIsql DESCRIBE command by using a SELECT * query on the table that you want to describe. This example describes columns of the employees table.

```
DECLARE
  c           NUMBER;
  d           NUMBER;
  col_cnt     INTEGER;
  f           BOOLEAN;
  rec_tab     DBMS_SQL.DESC_TAB;
  col_num    NUMBER;
PROCEDURE print_rec(rec in DBMS_SQL.DESC_REC) IS
BEGIN
  DBMS_OUTPUT.NEW_LINE;
  DBMS_OUTPUT.PUT_LINE('col_type            =    ' || rec.col_type);
  DBMS_OUTPUT.PUT_LINE('col_maxlen          =    ' || rec.col_max_len);
  DBMS_OUTPUT.PUT_LINE('col_name            =    ' || rec.col_name);
  DBMS_OUTPUT.PUT_LINE('col_name_len        =    ' || rec.col_name_len);
  DBMS_OUTPUT.PUT_LINE('col_schema_name     =    ' || rec.col_schema_name);
  DBMS_OUTPUT.PUT_LINE('col_schema_name_len =    ' || rec.col_schema_name_len);
  DBMS_OUTPUT.PUT_LINE('col_precision       =    ' || rec.col_precision);
  DBMS_OUTPUT.PUT('col_null_ok         =    ');
  IF (rec.col_null_ok) THEN
    DBMS_OUTPUT.PUT_LINE('true');
  ELSE
    DBMS_OUTPUT.PUT_LINE('false');
  END IF;
END;
BEGIN
  c := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(c, 'SELECT * FROM employees', DBMS_SQL.NATIVE);
  d := DBMS_SQL.EXECUTE(c);
  DBMS_SQL.DESCRIBE_COLUMNS(c, col_cnt, rec_tab);
  col_num := rec_tab.first;
  IF (col_num IS NOT NULL) THEN
    LOOP
      print_rec(rec_tab(col_num));
      col_num := rec_tab.next(col_num);
      EXIT WHEN (col_num IS NULL);
    END LOOP;
  END IF;
END;
```

PL/SQL procedure successfully completed.
Here is an abbreviated sample of the output, describing columns of the `employees` table, assuming it was run from the `HR` schema. Information from only the first two columns is shown here:

```
col_type            =    2
col_maxlen          =    7
col_name            =    EMPLOYEE_ID
col_name_len        =    11
col_schema_name     =    HR
col_schema_name_len =    8
col_precision       =    6
col_scale           =    0
col_null_ok         =    false
```

**Example 8: RETURNING clause**

With this clause, INSERT, UPDATE, and DELETE statements can return values of expressions. These values are returned in bind variables.

DBMS_SQL.BIND_VARIABLE is used to bind these outbinds if a single row is inserted, updated, or deleted. If multiple rows are inserted, updated, or deleted, then DBMS_SQL.BIND_ARRAY is used. DBMS_SQL.VARIABLE_VALUE must be called to get the values in these bind variables.

---

**Note:** This is similar to DBMS_SQL.VARIABLE_VALUE, which must be called after running a PL/SQL block with an out-bind inside DBMS_SQL.

---

The examples that follow assume a table `tab` has been created:

Command> create table tab (c1 number, c2 number);

i) Single row insert.

```sql
CREATE OR REPLACE PROCEDURE single_Row_insert
(c1 NUMBER, c2 NUMBER, r OUT NUMBER) is
  c NUMBER;
  n NUMBER;
begin
  c := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(c, 'INSERT INTO tab VALUES (:bnd1, :bnd2) ' ||
    'RETURNING c1*c2 INTO :bnd3', DBMS_SQL.NATIVE);
  DBMS_SQL.BIND_VARIABLE(c, 'bnd1', c1);
  DBMS_SQL.INSERT(c);
  DBMS_SQL.BIND_VARIABLE(c, 'bnd3', c);
  DBMS_SQL.CLOSE_CURSOR(c);
  n := c;
end;
/```

...
DBMS_SQL.BIND_VARIABLE(c, 'bnd2', c2);
DBMS_SQL.BIND_VARIABLE(c, 'bnd3', r);
n := DBMS_SQL.EXECUTE(c);
DBMS_SQL.VARIABLE_VALUE(c, 'bnd3', r); -- get value of outbind variable
DBMS_SQL.CLOSE_CURSOR(c);
END;
/

The following runs this example and shows the results. The table was initially empty.

Command> declare r NUMBER;
  > begin
  > single_Row_insert(100,200,r);
  > dbms_output.put_line('Product = ' || r);
  > end;
  > /
Product = 20000

PL/SQL procedure successfully completed.

Command> select * from tab;
< 100, 200 >
1 row found.

ii) Single row update. Note that rownum is an internal variable for row number.

CREATE OR REPLACE PROCEDURE single_Row_update
  (c1 NUMBER, c2 NUMBER, r out NUMBER) IS
    c NUMBER;
    n NUMBER;
BEGIN
  c := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(c, 'UPDATE tab SET c1 = :bnd1, c2 = :bnd2 ' ||
    'WHERE rownum = 1 ' ||
    'RETURNING c1*c2 INTO :bnd3', DBMS_SQL.NATIVE);
  DBMS_SQL.BIND_VARIABLE(c, 'bnd1', c1);
  DBMS_SQL.BIND_VARIABLE(c, 'bnd2', c2);
  DBMS_SQL.BIND_VARIABLE(c, 'bnd3', r);
  n := DBMS_SQL.EXECUTE(c);
  DBMS_SQL.VARIABLE_VALUE(c, 'bnd3', r);-- get value of outbind variable
  DBMS_SQL.CLOSE_CURSOR(c);
END;
/

The following runs this example and shows the results, updating the row that was
inserted in the previous example.

Command> declare r NUMBER;
  > begin
  > single_Row_update(200,300,r);
  > dbms_output.put_line('Product = ' || r);
  > end;
  > /
Product = 60000

PL/SQL procedure successfully completed.

Command> select * from tab;
< 200, 300 >
1 row found.
iii) Multiple row insert.

```sql
CREATE OR REPLACE PROCEDURE multi_Row_insert
   (c1 DBMS_SQL.NUMBER_TABLE, c2 DBMS_SQL.NUMBER_TABLE,
    r OUT DBMS_SQL.NUMBER_TABLE) is
  c NUMBER;
  n NUMBER;
BEGIN
  c := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(c, 'insert into tab VALUES (:bnd1, :bnd2) ' ||
    'RETURNING c1*c2 INTO :bnd3', DBMS_SQL.NATIVE);
  DBMS_SQL.BIND_ARRAY(c, 'bnd1', c1);
  DBMS_SQL.BIND_ARRAY(c, 'bnd2', c2);
  DBMS_SQL.BIND_ARRAY(c, 'bnd3', r);
  n := DBMS_SQL.EXECUTE(c);
  DBMS_SQL.VARIABLE_VALUE(c, 'bnd3', r); -- get value of outbind variable
  DBMS_SQL.CLOSE_CURSOR(c);
END;
/
```

The following script runs this example:

```sql
declare
  c1_array dbms_sql.number_table;
  c2_array dbms_sql.number_table;
  r_array dbms_sql.number_table;
begin
  c1_array(1) := 10;
  c1_array(2) := 20;
  c1_array(3) := 30;
  c2_array(1) := 15;
  c2_array(2) := 25;
  c2_array(3) := 35;
  multi_Row_insert(c1_array,c2_array,r_array);
  dbms_output.put_line('Product for row1 = ' || r_array(1));
  dbms_output.put_line('Product for row2 = ' || r_array(2));
  dbms_output.put_line('Product for row3 = ' || r_array(3));
end;
/
```

Following are the results. The table was initially empty.

Product for row1 = 150
Product for row2 = 500
Product for row3 = 1050

PL/SQL procedure successfully completed.

```
Command> select * from tab;
< 10, 15 >
< 20, 25 >
< 30, 35 >
3 rows found.
```

iv) Multiple row update.

```sql
CREATE OR REPLACE PROCEDURE multi_Row_update
   (c1 NUMBER, c2 NUMBER, r OUT DBMS_SQL.NUMBER_TABLE) IS
  c NUMBER;
  n NUMBER;
BEGIN
  c := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(c, 'update tab set col1 = col1*2, col2 = col2*2 ' ||
    'RETURNING c1*c2 INTO :bnd3', DBMS_SQL.NATIVE);
  DBMS_SQL.BIND_PARAM(c, 'bnd1', c1);
  DBMS_SQL.BIND_PARAM(c, 'bnd2', c2);
  DBMS_SQL.BIND_ARRAY(c, 'bnd3', r);
  n := DBMS_SQL.EXECUTE(c);
  DBMS_SQL.VARIABLE_VALUE(c, 'bnd3', r); -- get value of outbind variable
  DBMS_SQL.CLOSE_CURSOR(c);
END;
/
```

The following script runs this example:

```sql
declare
  c1_array dbms_sql.number_table;
  c2_array dbms_sql.number_table;
  r_array dbms_sql.number_table;
begin
  c1_array(1) := 10;
  c1_array(2) := 20;
  c1_array(3) := 30;
  c2_array(1) := 15;
  c2_array(2) := 25;
  c2_array(3) := 35;
  multi_Row_update(c1_array,c2_array,r_array);
  dbms_output.put_line('Product for row1 = ' || r_array(1));
  dbms_output.put_line('Product for row2 = ' || r_array(2));
  dbms_output.put_line('Product for row3 = ' || r_array(3));
end;
/
```

Following are the results.

Product for row1 = 75
Product for row2 = 250
Product for row3 = 1050

PL/SQL procedure successfully completed.

```
Command> select * from tab;
< 10, 15 >
< 20, 25 >
< 30, 35 >
3 rows found.
```
DBMS_SQL.PARSE(c, 'UPDATE tab SET c1 = :bnd1 WHERE c2 > :bnd2 ' ||
    'RETURNING c1*c2 INTO :bnd3', DBMS_SQL.NATIVE);
DBMS_SQL.BIND_VARIABLE(c, 'bnd1', c1);
DBMS_SQL.BIND_VARIABLE(c, 'bnd2', c2);
DBMS_SQL.BIND_ARRAY(c, 'bnd3', r);
n := DBMS_SQL.EXECUTE(c);
DBMS_OUTPUT.PUT_LINE(n || ' rows updated');
DBMS_SQL.VARIABLE_VALUE(c, 'bnd3', r);-- get value of outbind variable
DBMS_SQL.CLOSE_CURSOR(c);
END;
/

Note: Note that bnd1 and bnd2 can be arrays as well. The value of the expression for all the rows updated will be in bnd3. There is no way of differentiating which rows were updated of each value of bnd1 and bnd2.

The following script runs the example:

declare
  c1 NUMBER;
  c2 NUMBER;
  r_array dbms_sql.number_table;
begin
  c1 := 100;
  c2 := 0;
  multi_Row_update(c1, c2, r_array);
  dbms_output.put_line('Product for row1 = ' || r_array(1));
  dbms_output.put_line('Product for row2 = ' || r_array(2));
  dbms_output.put_line('Product for row3 = ' || r_array(3));
end;
/

Here are the results, updating the rows that were inserted in the previous example. (The report of the number of rows updated is from the example itself. The products are reported by the test script.)

3 rows updated
Product for row1 = 1500
Product for row2 = 2500
Product for row3 = 3500

PL/SQL procedure successfully completed.

Command> select * from tab;
< 100, 15 >
< 100, 25 >
< 100, 35 >
3 rows found.
Command>

v) Multiple row delete.

CREATE OR REPLACE PROCEDURE multi_Row_delete
  (c1_test NUMBER,
   r OUT DBMS_SQL.NUMBER_TABLE) is
  c NUMBER;
n NUMBER;
BEGIN

c := DBMS_SQL.OPEN_CURSOR;
DBMS_SQL.PARSE(c, 'DELETE FROM tab WHERE c1 = :bnd1 ' ||
               'RETURNING c1*c2 INTO :bnd2', DBMS_SQL.NATIVE);
DBMS_SQL.BIND_VARIABLE(c, 'bnd1', c1_test);
DBMS_SQL.BIND_ARRAY(c, 'bnd2', r);
n := DBMS_SQL.EXECUTE(c);
DBMS_OUTPUT.PUT_LINE(n || ' rows deleted');
DBMS_SQL.VARIABLE_VALUE(c, 'bnd2', r);-- get value of outbind variable
DBMS_SQL.CLOSE_CURSOR(c);
END;
/

The following script runs the example.

declare
  c1_test NUMBER;
  r_array dbms_sql.number_table;
begin
  c1_test := 100;
multi_Row_delete(c1_test, r_array);
dbms_output.put_line('Product for row1 = ' || r_array(1));
dbms_output.put_line('Product for row2 = ' || r_array(2));
dbms_output.put_line('Product for row3 = ' || r_array(3));
end;
/

Here are the results, deleting the rows that were updated in the previous example.
(The report of the number of rows deleted is from the example itself. The products are
reported by the test script.)

3 rows deleted
Product for row1 = 1500
Product for row2 = 2500
Product for row3 = 3500

PL/SQL procedure successfully completed.

Command> select * from tab;
0 rows found.

Note: DBMS_SQL.BIND_ARRAY of number_Table internally
binds a number. The number of times statement is run depends on
the number of elements in an inbind array.
Data structures

The DBMS_SQL package defines the following record types and table types.

---

Note: Notes on data types:

- The PLS_INTEGER and BINARY_INTEGER data types are identical. This document uses "BINARY_INTEGER" to indicate data types in reference information (such as for table types, record types, subprogram parameters, or subprogram return values), but may use either in discussion and examples.

- The INTEGER and NUMBER(38) data types are also identical. This document uses "INTEGER" throughout.

---

Record types

- DESC_REC record type
- DESC_REC2 record type
- DESC_REC3 record type

Table types

- BINARY_DOUBLE_TABLE table type
- BINARY_FLOAT_TABLE table type
- DATE_TABLE table type
- DESC_TAB table type
- DESC_TAB2 table type
- DESC_TAB3 table type
- INTERVAL_DAY_TO_SECOND_TABLE table type
- INTERVAL_YEAR_TO_MONTH_TABLE table type
- NUMBER_TABLE table type
- TIME_TABLE table type
- TIMESTAMP_TABLE table type
- VARCHAR2_TABLE table type
- VARCHAR2A table type
- VARCHAR2S table type
DESC_REC record type

**Note:** This type has been deprecated in favor of the DESC_REC2 record type.

This record type holds the describe information for a single column in a dynamic query. It is the element type of the DESC_TAB table type and the DESCRIBE_COLUMNS procedure.

**Syntax**

```sql
TYPE desc_rec IS RECORD (  
  col_type            BINARY_INTEGER := 0,
  col_max_len         BINARY_INTEGER := 0,
  col_name            VARCHAR2(32)   := '',
  col_name_len        BINARY_INTEGER := 0,
  col_schema_name     VARCHAR2(32)   := '',
  col_schema_name_len BINARY_INTEGER := 0,
  col_precision       BINARY_INTEGER := 0,
  col_scale           BINARY_INTEGER := 0,
  col_charsetid       BINARY_INTEGER := 0,
  col_charsetform     BINARY_INTEGER := 0,
  col_null_ok         BOOLEAN        := TRUE);
```

```sql
TYPE desc_tab IS TABLE OF desc_rec INDEX BY BINARY_INTEGER;
```

**Fields**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>col_type</td>
<td>Type of column.</td>
</tr>
<tr>
<td>col_max_len</td>
<td>Maximum column length.</td>
</tr>
<tr>
<td>col_name</td>
<td>Name of column.</td>
</tr>
<tr>
<td>col_name_len</td>
<td>Length of column name.</td>
</tr>
<tr>
<td>col_schema_name</td>
<td>Column schema name.</td>
</tr>
<tr>
<td>col_schema_name_len</td>
<td>Length of column schema name.</td>
</tr>
<tr>
<td>col_precision</td>
<td>Precision of column.</td>
</tr>
<tr>
<td>col_scale</td>
<td>Scale of column.</td>
</tr>
<tr>
<td>col_charsetid</td>
<td>Column character set id.</td>
</tr>
<tr>
<td>col_charsetform</td>
<td>Column character set form.</td>
</tr>
<tr>
<td>col_null_ok</td>
<td>NULL column flag. TRUE if NULL is allowable.</td>
</tr>
</tbody>
</table>
DESC_REC2 record type

DESC_REC2 is the element type of the DESC_TAB2 table type and the DESCRIBE_COLUMNS2 procedure.

This record type is identical to DESC_REC except for the col_name field, which has been expanded to the maximum possible size for VARCHAR2. It is therefore preferred to DESC_REC because column name values can be greater than 32 characters. DESC_REC is deprecated as a result.

Syntax

```
TYPE desc_rec2 IS RECORD (
  col_type            binary_integer := 0,
  col_max_len         binary_integer := 0,
  col_name            varchar2(32767) := '',
  col_name_len        binary_integer := 0,
  col_schema_name     varchar2(32)   := '',
  col_schema_name_len binary_integer := 0,
  col_precision       binary_integer := 0,
  col_scale           binary_integer := 0,
  col_charsetid       binary_integer := 0,
  col_charsetform     binary_integer := 0,
  col_null_ok         boolean        := TRUE);
```

Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>col_type</td>
<td>Type of column.</td>
</tr>
<tr>
<td>col_max_len</td>
<td>Maximum column length.</td>
</tr>
<tr>
<td>col_name</td>
<td>Name of column.</td>
</tr>
<tr>
<td>col_name_len</td>
<td>Length of column name.</td>
</tr>
<tr>
<td>col_schema_name</td>
<td>Column schema name.</td>
</tr>
<tr>
<td>col_schema_name_len</td>
<td>Length of column schema name.</td>
</tr>
<tr>
<td>col_precision</td>
<td>Precision of column.</td>
</tr>
<tr>
<td>col_scale</td>
<td>Scale of column.</td>
</tr>
<tr>
<td>col_charsetid</td>
<td>Column character set ID.</td>
</tr>
<tr>
<td>col_charsetform</td>
<td>Column character set form.</td>
</tr>
<tr>
<td>col_null_ok</td>
<td>NULL column flag. TRUE if NULL is allowable.</td>
</tr>
</tbody>
</table>
DESC_REC3 record type

DESC_REC3 is the element type of the DESC_TAB3 table type and the DESCRIPT_COLUMNS3 procedure.

DESC_REC3 is identical to DESC_REC2 except for two additional fields to hold the type name (type_name) and type name len (type_name_len) of a column in a dynamic query. The col_type_name and col_type_name_len fields are only populated when the col_type field value is 109 (the Oracle type number for user-defined types), which is not currently used.

Syntax

```plsql
TYPE desc_rec3 IS RECORD (
    col_type               binary_integer := 0,
    col_max_len            binary_integer := 0,
    col_name               varchar2(32767) := '',
    col_name_len           binary_integer := 0,
    col_schema_name        varchar2(32) := '',
    col_schema_name_len    binary_integer := 0,
    col_precision          binary_integer := 0,
    col_scale              binary_integer := 0,
    col_charsetid          binary_integer := 0,
    col_charsetform        binary_integer := 0,
    col_null_ok            boolean := TRUE,
    col_type_name          varchar2(32767) := '',
    col_type_name_len      binary_integer := 0);
```

Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>col_type</td>
<td>Type of column.</td>
</tr>
<tr>
<td>col_max_len</td>
<td>Maximum column length.</td>
</tr>
<tr>
<td>col_name</td>
<td>Name of column.</td>
</tr>
<tr>
<td>col_name_len</td>
<td>Length of column name.</td>
</tr>
<tr>
<td>col_schema_name</td>
<td>Column schema name.</td>
</tr>
<tr>
<td>col_schema_name_len</td>
<td>Length of column schema name.</td>
</tr>
<tr>
<td>col_precision</td>
<td>Precision of column.</td>
</tr>
<tr>
<td>col_scale</td>
<td>Scale of column.</td>
</tr>
<tr>
<td>col_charsetid</td>
<td>Column character set ID.</td>
</tr>
<tr>
<td>col_charsetform</td>
<td>Column character set form.</td>
</tr>
<tr>
<td>col_null_ok</td>
<td>NULL column flag. TRUE if NULL is allowable.</td>
</tr>
<tr>
<td>col_type_name</td>
<td>Reserved for future use.</td>
</tr>
<tr>
<td>col_type_name_len</td>
<td>Reserved for future use.</td>
</tr>
</tbody>
</table>
**BINARY_DOUBLE_TABLE table type**

This is a table of BINARY_DOUBLE.

**Syntax**

```
TYPE binary_double_table IS TABLE OF BINARY_DOUBLE INDEX BY BINARY_INTEGER;
```
**BINARY_FLOAT_TABLE table type**

This is a table of BINARY_FLOAT.

**Syntax**

```sql
TYPE binary_float_table IS TABLE OF BINARY_FLOAT INDEX BY BINARY_INTEGER;
```
DATE_TABLE table type

This is a table of DATE.

Syntax

type date_table IS TABLE OF DATE INDEX BY BINARY_INTEGER;
DESC_TAB table type

This is a table of DESC_REC record type.

Syntax

    TYPE desc_tab IS TABLE OF desc_rec INDEX BY BINARY_INTEGER;
DESC_TAB2 table type

This is a table of DESC_REC2 record type.

Syntax

```sql
TYPE desc_tab2 IS TABLE OF desc_rec2 INDEX BY BINARY_INTEGER;
```
DESC_TAB3 table type

This is a table of DESC_REC3 record type.

Syntax

```plsql
TYPE desc_tab3 IS TABLE OF desc_rec3 INDEX BY BINARY_INTEGER;
```
INTERVAL_DAY_TO_SECOND_TABLE table type

This is a table of DSINTERVAL_UNCONSTRAINED.

Syntax

```
TYPE interval_day_to_second_Table IS TABLE OF
  DSINTERVAL_UNCONSTRAINED INDEX BY binary_integer;
```
INTERRUPT_YEAR_TO_MONTH_TABLE table type

This is a table of YMINTERVAL_UNCONSTRAINED.

Syntax

```
TYPE interval_year_to_month_table IS TABLE OF YMINTERVAL_UNCONSTRAINED
   INDEX BY BINARY_INTEGER;
```
NUMBER_TABLE table type

This is a table of NUMBER.

Syntax

```sql
TYPE number_table IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
```
TIME_TABLE table type

This is a table of TIME_UNCONSTRAINED.

Syntax

```
TYPE time_table IS TABLE OF TIME_UNCONSTRAINED INDEX BY BINARY_INTEGER;
```
TIMESTAMP_TABLE table type

This is a table of TIMESTAMP_UNCONSTRAINED.

Syntax

```
TYPE timestamp_table IS TABLE OF TIMESTAMP_UNCONSTRAINED INDEX BY BINARY_INTEGER;
```
VARCHAR2_TABLE table type

This is table of VARCHAR2(2000).

Syntax

```plsql
TYPE varchar2_table IS TABLE OF VARCHAR2(2000) INDEX BY BINARY_INTEGER;
```
VARCHAR2A table type

This is table of VARCHAR2(32767).

Syntax

```
TYPE varchar2a IS TABLE OF VARCHAR2(32767) INDEX BY BINARY_INTEGER;
```
VARCHAR2S table type

This is table of VARCHAR2(256).

Note: This type has been superseded by the VARCHAR2A table type. It is supported only for backward compatibility.

Syntax

```sql
TYPE varchar2s IS TABLE OF VARCHAR2(256) INDEX BY BINARY_INTEGER;
```
## Summary of DBMS_SQL subprograms

### Table 6–5 DBMS_SQL Package Subprograms

<table>
<thead>
<tr>
<th>Subprogram</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIND_ARRAY procedure</td>
<td>Binds a given value to a given collection.</td>
</tr>
<tr>
<td>BIND_VARIABLE procedure</td>
<td>Binds a given value to a given variable.</td>
</tr>
<tr>
<td>CLOSE_CURSOR procedure</td>
<td>Closes given cursor and frees memory.</td>
</tr>
<tr>
<td>COLUMN_VALUE procedure</td>
<td>Returns value of the cursor element for a given position in a cursor.</td>
</tr>
<tr>
<td>COLUMN_VALUE_LONG Procedure</td>
<td>Returns a selected part of a LONG column that has been defined using DEFINE_COLUMN_LONG.</td>
</tr>
<tr>
<td></td>
<td><strong>Important</strong>: Because TimesTen does not support the LONG data type, attempting to use this procedure in TimesTen will result in an ORA-01018 error at runtime.</td>
</tr>
<tr>
<td></td>
<td>The COLUMN_VALUE_LONG and DEFINE_COLUMN_LONG procedures are therefore not documented in this manual.</td>
</tr>
<tr>
<td>DEFINE_ARRAY procedure</td>
<td>Defines a collection to be selected from the given cursor, used only with SELECT statements.</td>
</tr>
<tr>
<td>DEFINE_COLUMN procedure</td>
<td>Defines a column to be selected from the given cursor, used only with SELECT statements.</td>
</tr>
<tr>
<td>DEFINE_COLUMN_LONG Procedure</td>
<td>Defines a LONG column to be selected from the given cursor. Use with SELECT statements.</td>
</tr>
<tr>
<td></td>
<td><strong>Important</strong>: Because TimesTen does not support the LONG data type, attempting to use the COLUMN_VALUE_LONG procedure in TimesTen will result in an ORA-01018 error at runtime.</td>
</tr>
<tr>
<td></td>
<td>The COLUMN_VALUE_LONG and DEFINE_COLUMN_LONG procedures are therefore not documented in this manual.</td>
</tr>
<tr>
<td>DESCRIBE_COLUMNS procedure</td>
<td>Describes the columns for a cursor opened and parsed through DBMS_SQL.</td>
</tr>
<tr>
<td>DESCRIBE_COLUMNS2 procedure</td>
<td>Describes the specified column, as an alternative to DESCRIBE_COLUMNS procedure.</td>
</tr>
<tr>
<td>DESCRIBE_COLUMNS3 procedure</td>
<td>Describes the specified column, as an alternative to DESCRIBE_COLUMNS procedure.</td>
</tr>
<tr>
<td>EXECUTE function</td>
<td>Executes a given cursor.</td>
</tr>
<tr>
<td>EXECUTE_AND_FETCH function</td>
<td>Executes a given cursor and fetches rows.</td>
</tr>
<tr>
<td>FETCH_ROWS function</td>
<td>Fetches a row from a given cursor.</td>
</tr>
<tr>
<td>IS_OPEN function</td>
<td>Returns TRUE if the given cursor is open.</td>
</tr>
<tr>
<td>LAST_ERROR_POSITION function</td>
<td>Returns byte offset in the SQL statement text where the error occurred.</td>
</tr>
<tr>
<td>LAST_ROW_COUNT function</td>
<td>Returns cumulative count of the number of rows fetched.</td>
</tr>
<tr>
<td>Subprogram</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>---------------------------------------------------------------</td>
</tr>
<tr>
<td>LAST_ROW_ID function</td>
<td>Returns ROWID of last row processed.</td>
</tr>
<tr>
<td></td>
<td>TimesTen does not support ROWID of the last row operated on by a DML statement. This function returns NULL.</td>
</tr>
<tr>
<td>LAST_SQL_FUNCTION_CODE function</td>
<td>Returns SQL function code for statement.</td>
</tr>
<tr>
<td>OPEN_CURSOR function</td>
<td>Returns cursor ID number of new cursor.</td>
</tr>
<tr>
<td>PARSE procedure</td>
<td>Parses given statement.</td>
</tr>
<tr>
<td>TO_CURSOR_NUMBER function</td>
<td>Takes an opened strongly or weakly-typed REF CURSOR and transforms it into a DBMS_SQL cursor number.</td>
</tr>
<tr>
<td>TO_REFCURSOR function</td>
<td>Takes an opened, parsed, and executed cursor 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.</td>
</tr>
<tr>
<td>VARIABLE_VALUE procedure</td>
<td>Returns value of named variable for given cursor.</td>
</tr>
</tbody>
</table>
**BIND_ARRAY procedure**

This procedure binds a given value or set of values to a given variable in a cursor, based on the name of the variable in the statement.

**Syntax**

```sql
DBMS_SQL.BIND_ARRAY (
    c                   IN INTEGER,
    name                IN VARCHAR2,
    <table_variable>    IN <datatype>
[],index1              IN INTEGER,
    index2              IN INTEGER)
);
```

Where the `<table_variable>` and its corresponding `<datatype>` can be any one of the following matching pairs:

- `<bflt_tab>`   `dbms_sql.Binary_Float_Table`
- `<bdbl_tab>`   `dbms_sql.Binary_Double_Table`
- `<c_tab>`      `dbms_sql.Varchar2_Table`
- `<d_tab>`      `dbms_sql.Date_Table`
- `<ids_tab>`    `dbms_sql.Interval_Day_to_Second_Table`
- `<iym_tab>`    `dbms_sql.Interval_Year_to_Month_Table`
- `<n_tab>`      `dbms_sql.Number_Table`
- `<tm_tab>`     `dbms_sql.Time_Table`
- `<tms_tab>`    `dbms_sql.Timestamp_Table`

Notice that the BIND_ARRAY procedure is overloaded to accept different data types.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>ID number of the cursor to which you want to bind a value.</td>
</tr>
<tr>
<td>name</td>
<td>Name of the collection in the statement.</td>
</tr>
<tr>
<td>table_variable</td>
<td>Local variable that has been declared as <code>&lt;datatype&gt;</code>.</td>
</tr>
<tr>
<td>index1</td>
<td>Index for the table element that marks the lower bound of the range.</td>
</tr>
<tr>
<td>index2</td>
<td>Index for the table element that marks the upper bound of the range.</td>
</tr>
</tbody>
</table>

**Usage notes**

The length of the bind variable name should be less than or equal to 30 bytes.

For binding a range, the table must contain the elements that specify the range — `tab(index1)` and `tab(index2)` — but the range does not have to be dense. The `index1` value must be less than or equal to `index2`. All elements between `tab(index1)` and `tab(index2)` are used in the bind.

If you do not specify indexes in the bind call, and two different binds in a statement specify tables that contain a different number of elements, then the number of elements actually used is the minimum number between all tables. This is also the case if you specify indexes. The minimum range is selected between the two indexes for all tables.
Not all bind variables in a query have to be array binds. Some can be regular binds and the same value are used for each element of the collections in expression evaluations (and so forth).

**Bulk array binds**

Bulk selects, inserts, updates, and deletes can enhance the performance of applications by bundling many calls into one. The DBMS_SQL package lets you work on collections of data using the PL/SQL table type.

*Table items* are unbounded homogeneous collections. In persistent storage, they are like other relational tables and have no intrinsic ordering. But when a table item is brought into the workspace (either by querying or by navigational access of persistent data), or when it is created as the value of a PL/SQL variable or parameter, its elements are given subscripts that can be used with array-style syntax to get and set the values of elements.

The subscripts of these elements need not be dense, and can be any number including negative numbers. For example, a table item can contain elements at locations -10, 2, and 7 only.

When a table item is moved from transient workspace to persistent storage, the subscripts are not stored; the table item is unordered in persistent storage.

At bind time the table is copied out from the PL/SQL buffers into local DBMS_SQL buffers (the same as for all scalar types) and then the table is manipulated from the local DBMS_SQL buffers. Therefore, if you change the table after the bind call, then that change does not affect the way the execute acts.

**Types for scalar collections**

You can declare a local variable as one of the following table-item types, which are defined as public types in DBMS_SQL.

```
TYPE binary_double_table
   IS TABLE OF BINARY_DOUBLE  INDEX BY BINARY_INTEGER;

TYPE binary_float_table
   IS TABLE OF BINARY_FLOAT   INDEX BY BINARY_INTEGER;

TYPE date_table
   IS TABLE OF DATE           INDEX BY BINARY_INTEGER;

TYPE interval_day_to_second_table
   IS TABLE OF dsinterval_unconstrained
           INDEX BY BINARY_INTEGER;

TYPE interval_year_to_month_table
   IS TABLE OF yminterval_unconstrained
           INDEX BY BINARY_INTEGER;

TYPE number_table   IS TABLE OF NUMBER         INDEX BY BINARY_INTEGER;

TYPE time_table     IS TABLE OF time_unconstrained
           INDEX BY BINARY_INTEGER;

TYPE timestamp_table
   IS TABLE OF timestamp_unconstrained
           INDEX BY BINARY_INTEGER;

TYPE varchar2_table IS TABLE OF VARCHAR2(2000) INDEX BY BINARY_INTEGER;
```

Examples

See "Examples" on page 6-11.
BIND_VARIABLE procedure

This procedure binds a given value or set of values to a given variable in a cursor, based on the name of the variable in the statement.

Syntax

```sql
DBMS_SQL.BIND_VARIABLE (  
    c           IN INTEGER,  
    name        IN VARCHAR2,  
    value       IN <datatype>);  
```

Where `<datatype>` can be any one of the following types:

- BINARY_DOUBLE
- BINARY_FLOAT
- DATE
- INTERVAL DAY TO SECOND(9,9) (DSINTERVAL_UNCONSTRAINED)
- NUMBER
- TIME(9) (TIME_UNCONSTRAINED)
- TIMESTAMP(9) (TIMESTAMP_UNCONSTRAINED)
- VARCHAR2 CHARACTER SET ANY_CS
- INTERVAL YEAR TO MONTH(9) (YMINTERVAL_UNCONSTRAINED)
- VARRAY
- Nested table

Notice that BIND_VARIABLE is overloaded to accept different data types.

The following syntax is also supported for BIND_VARIABLE. The square brackets [] indicate an optional parameter for the BIND_VARIABLE function.

```sql
DBMS_SQL.BIND_VARIABLE (  
    c           IN INTEGER,  
    name        IN VARCHAR2,  
    value       IN VARCHAR2 CHARACTER SET ANY_CS [,out_value_size IN INTEGER]];  
```

To bind CHAR, RAW, and ROWID data, you can use the following variations on the syntax:

```sql
DBMS_SQL.BIND_VARIABLE_CHAR (  
    c           IN INTEGER,  
    name        IN VARCHAR2,  
    value       IN CHAR CHARACTER SET ANY_CS [,out_value_size IN INTEGER]];  
```

```sql
DBMS_SQL.BIND_VARIABLE_RAW (  
    c           IN INTEGER,  
    name        IN VARCHAR2,  
    value       IN RAW [,out_value_size IN INTEGER]];  
```

```sql
DBMS_SQL.BIND_VARIABLE_ROWID (  
    c           IN INTEGER,  
    name        IN VARCHAR2,  
    value       IN ROWID);  
```

Pragmas

```sql
pragma restrict_references(bind_variable,WNDS);  
```
Parameters

Table 6-7  BIND_VARIABLE procedure parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>ID number of the cursor to which you want to bind a value.</td>
</tr>
<tr>
<td>name</td>
<td>Name of the variable in the statement.</td>
</tr>
<tr>
<td>value</td>
<td>Value that you want to bind to the variable in the cursor.</td>
</tr>
<tr>
<td></td>
<td>For IN and IN OUT variables, the value has the same type as the type of the</td>
</tr>
<tr>
<td></td>
<td>value being passed in for this parameter.</td>
</tr>
<tr>
<td>out_value_size</td>
<td>Maximum expected OUT value size, in bytes, for the VARCHAR2, RAW, CHAR OUT</td>
</tr>
<tr>
<td></td>
<td>or IN OUT variable.</td>
</tr>
<tr>
<td></td>
<td>If no size is given, then the length of the current value is used. This</td>
</tr>
<tr>
<td></td>
<td>parameter must be specified if the value parameter is not initialized.</td>
</tr>
</tbody>
</table>

Usage notes

If the variable is an IN or IN OUT variable or an IN collection, then the given bind value must be valid for the variable or array type. Bind values for OUT variables are ignored.

The bind variables or collections of a SQL statement are identified by their names. When binding a value to a bind variable or bind array, the string identifying it in the statement must contain a leading colon, as shown in the following example:

```sql
SELECT last_name FROM employees WHERE salary > :X;
```

For this example, the corresponding bind call would look similar to:

```sql
BIND_VARIABLE(cursor_name, ':X', 3500);
```

Or:

```sql
BIND_VARIABLE (cursor_name, 'X', 3500);
```

The length of the bind variable name should be less than or equal to 30 bytes.

Examples

See "Examples" on page 6-11.
CLOSE_CURSOR procedure

This procedure closes a given cursor.

Syntax

```sql
DBMS_SQL.CLOSE_CURSOR (  
c    IN OUT INTEGER);
```

Pragmas

```sql
pragma restrict_references(close_cursor,RNDS,WNDS);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>For IN, the ID number of the cursor that you want to close. For OUT, the cursor is set to NULL. After you call CLOSE_CURSOR, the memory allocated to the cursor is released and you can no longer fetch from that cursor.</td>
</tr>
</tbody>
</table>
COLUMN_VALUE procedure

This procedure returns the value of the cursor element for a given position in a given cursor. This procedure is used to access the data fetched by calling FETCH_ROWS.

Syntax

```
DBMS_SQL.COLUMN_VALUE (
    c                 IN  INTEGER,
    position          IN  INTEGER,
    value             OUT <datatype>
[,column_error      OUT NUMBER]
[,actual_length     OUT INTEGER]);
```

Where square brackets [ ] indicate optional parameters and <datatype> can be any one of the following types:

- **BINARY_DOUBLE**
- **BINARY_FLOAT**
- **DATE**
- **INTERVAL DAY TO SECOND(9,9) (DSINTERVAL_UNCONSTRAINED)**
- **NUMBER**
- **TIME(9) (TIME_UNCONSTRANDED)**
- **TIMESTAMP(9) (TIMESTAMP_UNCONSTRAINED)**
- **VARCHAR2 CHARACTER SET ANY_CS**
- **INTERVAL YEAR TO MONTH(9) (YMINTERVAL_UNCONSTRAINED)**
- **VARRAY**
- **Nested table**

For variables containing CHAR, RAW, and ROWID data, you can use the following variations on the syntax:

```
DBMS_SQL.COLUMN_VALUE_CHAR (
    c               IN  INTEGER,
    position        IN  INTEGER,
    value           OUT CHAR CHARACTER SET ANY_CS
[,column_error    OUT NUMBER]
[,actual_length   OUT INTEGER]);
```

```
DBMS_SQL.COLUMN_VALUE_RAW (
    c               IN  INTEGER,
    position        IN  INTEGER,
    value           OUT RAW
[,column_error    OUT NUMBER]
[,actual_length   OUT INTEGER]);
```

```
DBMS_SQL.COLUMN_VALUE_ROWID (
    c               IN  INTEGER,
    position        IN  INTEGER,
    value           OUT ROWID
[,column_error    OUT NUMBER]
[,actual_length   OUT INTEGER]);
```

The following syntax enables the COLUMN_VALUE procedure to accommodate bulk operations:

```
DBMS_SQL.COLUMN_VALUE(
    c                 IN             INTEGER,
    position          IN             INTEGER,
    <param_name>      IN OUT NOCOPY  <table_type>);
```
Where the `<param_name>` and its corresponding `<table_type>` can be any one of these matching pairs:

- `bdbl_tab` : `dbms_sql.Binary_Double_Table`
- `bflt_tab` : `dbms_sql.Binary_Float_Table`
- `c_tab` : `dbms_sql.Varchar2_Table`
- `d_tab` : `dbms_sql.Date_Table`
- `ids_tab` : `dbms_sql.Interval_Day_To_Second_Table`
- `iym_tab` : `dbms_sql.Interval_Year_To_Month_Table`
- `n_tab` : `dbms_sql.Number_Table`
- `tm_tab` : `dbms_sql.Time_Table`
- `tms_tab` : `dbms_sql.Timestamp_Table`

### Pragmas

```sql
pragma restrict_references(column_value,RNDS,WNDS);
```

### Parameters

#### Table 6–9 COLUMN_VALUE procedure parameters (single row)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>c</code></td>
<td>ID number of the cursor from which you are fetching the values.</td>
</tr>
<tr>
<td><code>position</code></td>
<td>Relative position of the column in the cursor.</td>
</tr>
<tr>
<td></td>
<td>The first column in a statement has position 1.</td>
</tr>
<tr>
<td><code>value</code></td>
<td>Returns the value at the specified column.</td>
</tr>
<tr>
<td></td>
<td>Oracle raises exception ORA-06562, <code>inconsistent_type</code>, if the type of this output parameter differs from the actual type of the value, as defined by the call to DEFINE_COLUMN.</td>
</tr>
<tr>
<td><code>column_error</code></td>
<td>Returns any error code for the specified column value.</td>
</tr>
<tr>
<td><code>actual_length</code></td>
<td>The actual length, before any truncation, of the value in the specified column.</td>
</tr>
</tbody>
</table>

#### Table 6–10 COLUMN_VALUE procedure parameters (bulk)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>c</code></td>
<td>ID number of the cursor from which you are fetching the values.</td>
</tr>
<tr>
<td><code>position</code></td>
<td>Relative position of the column in the cursor.</td>
</tr>
<tr>
<td></td>
<td>The first column in a statement has position 1.</td>
</tr>
<tr>
<td><code>&lt;param_name&gt;</code></td>
<td>Local variable that has been declared <code>&lt;table_type&gt;</code>.&lt;<code>param_name&gt;</code> is an IN OUT NOCOPY parameter for bulk operations.</td>
</tr>
<tr>
<td></td>
<td>For bulk operations, the subprogram appends the new elements at the appropriate (implicitly maintained) index. For instance if on utilizing the <code>DEFINE_ARRAY</code> procedure a batch size (the <code>cnt</code> parameter) of 10 rows was specified and a start index (<code>lower_bound</code>) of 1 was specified, then the first call to this subprogram after calling the <code>FETCH_ROWS</code> function will populate elements at index <code>1..10</code>, and the next call will populate elements <code>11..20</code>, and so on.</td>
</tr>
</tbody>
</table>

### Exceptions

`INCONSISTENT_TYPE` (ORA-06562) is raised if the type of the given OUT parameter `value` is different from the actual type of the value. This type was the given type when the column was defined by calling procedure `DEFINE_COLUMN`. 
COLUMN_VALUE procedure

Examples

See "Examples" on page 6-11.
**DEFINE_ARRAY procedure**

This procedure defines the collection for column into which you want to fetch rows (with a FETCH_ROWS call). This procedure lets you do batch fetching of rows from a single SELECT statement. A single fetch call brings over a number of rows into the PL/SQL aggregate object.

When you fetch the rows, they are copied into DBMS_SQL buffers until you run a COLUMN_VALUE call, at which time the rows are copied into the table that was passed as an argument to the COLUMN_VALUE call.

**Scalar types for collections**

You can declare a local variable as one of the following table-item types, and then fetch any number of rows into it using DBMS_SQL. These are the same types you can specify for the BIND_ARRAY procedure.

```plaintext
TYPE binary_double_table
     IS TABLE OF BINARY_DOUBLE  INDEX BY BINARY_INTEGER;

TYPE binary_float_table
     IS TABLE OF BINARY_FLOAT   INDEX BY BINARY_INTEGER;

TYPE date_table     IS TABLE OF DATE           INDEX BY BINARY_INTEGER;

TYPE interval_day_to_second_table
     IS TABLE OF dbms_sql.Interval_Day_To_Second_Table
                   INDEX BY BINARY_INTEGER;

TYPE interval_year_to_month_table
     IS TABLE OF dbms_sql.Interval_Year_To_Month_Table
                   INDEX BY BINARY_INTEGER;

TYPE number_table   IS TABLE OF NUMBER         INDEX BY BINARY_INTEGER;

TYPE time_table     IS TABLE OF time_unconstrained
                   INDEX BY BINARY_INTEGER;

TYPE timestamp_table
     IS TABLE OF timestamp_unconstrained
                   INDEX BY BINARY_INTEGER;

TYPE varchar2_table IS TABLE OF VARCHAR2(2000) INDEX BY BINARY_INTEGER;
```

**Syntax**

```plaintext
DBMS_SQL.DEFINE_ARRAY (  
    c           IN INTEGER,  
    position    IN INTEGER,  
    <table_variable>    IN <datatype>  
    cnt         IN INTEGER,  
    lower_bnd   IN INTEGER);  
```

Where `<table_variable>` and its corresponding `<datatype>` can be any one of the following matching pairs:

- `<bflt_tab>` : `dbms_sql.Binary_Float_Table`
- `<bdbl_tab>` : `dbms_sql.Binary_Double_Table`
- `<c_tab>` : `dbms_sql.Varchar2_Table`
- `<d_tab>` : `dbms_sql.Date_Table`
- `<n_tab>` : `dbms_sql.Number_Table`
- `<tm_tab>` : `dbms_sql.Time_Table`
- `<tms_tab>` : `dbms_sql.Timestamp_Table`
- `<ids_tab>` : `dbms_sql.Interval_Day_To_Second_Table`
- `<iym_tab>` : `dbms_sql.Interval_Year_To_Month_Table`

Note that DEFINE_ARRAY is overloaded to accept different data types.
Pragmas

```sql
pragma restrict_references(define_array,RNDS,WNDS);
```

The subsequent FETCH_ROWS call fetches the specified (cnt) number of rows. When the COLUMN_VALUE call is made, these rows are placed in positions `indx`, `indx+1`, `indx+2`, and so on. While there are still rows coming in, the user keeps issuing FETCH_ROWS and COLUMN_VALUE calls. The rows keep accumulating in the table specified as an argument in the COLUMN_VALUE call.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>c</code></td>
<td>ID number of the cursor to which you want to bind an array.</td>
</tr>
<tr>
<td><code>position</code></td>
<td>Relative position of the column in the array being defined.</td>
</tr>
<tr>
<td></td>
<td>The first column in a statement has position 1.</td>
</tr>
<tr>
<td><code>table_variable</code></td>
<td>Local variable that has been declared as <code>&lt;datatype&gt;</code>.</td>
</tr>
<tr>
<td><code>cnt</code></td>
<td>Number of rows that must be fetched.</td>
</tr>
<tr>
<td><code>lower_bnd</code></td>
<td>Results are copied into the collection, starting at this lower bound index.</td>
</tr>
</tbody>
</table>

Usage notes

The count (cnt) must be an integer greater than zero; otherwise an exception is raised. The `indx` can be positive, negative, or zero. A query on which a DEFINE_ARRAY call was issued cannot contain array binds.

Examples

See "Examples" on page 6-11.
**DEFINE_COLUMN procedure**

This procedure defines a column to be selected from the given cursor. This procedure is only used with SELECT cursors.

The column being defined is identified by its relative position in the SELECT list of the statement in the given cursor. The type of the COLUMN value determines the type of the column being defined.

**Syntax**

```sql
DBMS_SQL.DEFINE_COLUMN (
  c              IN INTEGER,
  position       IN INTEGER,
  column         IN <datatype>);
```

Where `<datatype>` can be any one of the following types:

- BINARY_DOUBLE
- BINARY_FLOAT
- DATE
- INTERVAL DAY TO SECOND(9,9) (DSINTERVAL_UNCONSTRAINED)
- NUMBER
- TIME(9) (TIME_UNCONSTRAINED)
- TIMESTAMP(9) (TIMESTAMP_UNCONSTRAINED)
- INTERVAL YEAR TO MONTH(9) (YMINTERVAL_UNCONSTRAINED)
- VARRAY

Note that DEFINE_COLUMN is overloaded to accept different data types.

The following syntax is also supported for the DEFINE_COLUMN procedure:

```sql
DBMS_SQL.DEFINE_COLUMN (
  c              IN INTEGER,
  position       IN INTEGER,
  column         IN VARCHAR2 CHARACTER SET ANY_CS,
  column_size    IN INTEGER);
```

To define columns with CHAR, RAW, and ROWID data, you can use the following variations on the procedure syntax:

```sql
DBMS_SQL.DEFINE_COLUMN_CHAR (
  c              IN INTEGER,
  position       IN INTEGER,
  column         IN CHAR CHARACTER SET ANY_CS,
  column_size    IN INTEGER);
```

```sql
DBMS_SQL.DEFINE_COLUMN_RAW (
  c              IN INTEGER,
  position       IN INTEGER,
  column         IN RAW,
  column_size    IN INTEGER);
```

```sql
DBMS_SQL.DEFINE_COLUMN_ROWID (
  c              IN INTEGER,
  position       IN INTEGER,
  column         IN ROWID);
```
Pragmas

```sql
pragma restrict_references(define_column, RNDS, WNDS);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>ID number of the cursor for the row being defined to be selected.</td>
</tr>
<tr>
<td>position</td>
<td>Relative position of the column in the row being defined.</td>
</tr>
<tr>
<td>column</td>
<td>Value of the column being defined.</td>
</tr>
<tr>
<td>column_size</td>
<td>Maximum expected size of the column value, in bytes, for columns of type VARCHAR2, CHAR, and RAW.</td>
</tr>
</tbody>
</table>

The first column in a statement has position 1.

Examples

See "Examples" on page 6-11.
DESCRIBE_COLUMNS procedure

This procedure describes the columns for a cursor opened and parsed through DBMS_SQL.

Syntax

DBMS_SQL.DESCRIBE_COLUMNS (
    c              IN  INTEGER,
    col_cnt        OUT INTEGER,
    desc_t         OUT DBMS_SQL.DESC_TAB);

DBMS_SQL.DESCRIBE_COLUMNS (
    c              IN  INTEGER,
    col_cnt        OUT INTEGER,
    desc_t         OUT DBMS_SQL.DESC_REC);

Parameters

Table 6–13  DESCRIBE_COLUMNS procedure parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>ID number of the cursor for the columns being described.</td>
</tr>
<tr>
<td>col_cnt</td>
<td>Number of columns in the select list of the query.</td>
</tr>
<tr>
<td>desc_t</td>
<td>Describe table to fill in with the description of each of the columns of the query.</td>
</tr>
</tbody>
</table>

Examples

See "Examples" on page 6-11.
DESCRIBE_COLUMNS2 procedure

This function describes the specified column. This is an alternative to DESCRIBE_COLUMNS procedure.

Syntax

```
DBMS_SQL.DESCRIBE_COLUMNS2 (  
  c              IN  INTEGER,  
  col_cnt        OUT INTEGER,  
  desc_t         OUT DBMS_SQL.DESC_TAB2);
```

```
DBMS_SQL.DESCRIBE_COLUMNS2 (  
  c              IN  INTEGER,  
  col_cnt        OUT INTEGER,  
  desc_t         OUT DBMS_SQL.DESC_REC2);
```

Pragmas

```
PRAGMA RESTRICT_REFERENCES(describe_columns2,WNDS);
```

Parameters

```
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>ID number of the cursor for the columns being described.</td>
</tr>
<tr>
<td>col_cnt</td>
<td>Number of columns in the select list of the query.</td>
</tr>
<tr>
<td>desc_t</td>
<td>Describe table to fill in with the description of each of the columns of the query. This table is indexed from one to the number of elements in the select list of the query.</td>
</tr>
</tbody>
</table>
**DESCRIBE_COLUMNS3 procedure**

This function describes the specified column. This is an alternative to **DESCRIBE_COLUMNS procedure**.

**Syntax**

```sql
DBMS_SQL.DESCRIBE_COLUMNS3 (  
  c              IN  INTEGER,  
  col_cnt        OUT INTEGER,  
  desc_t         OUT DBMS_SQL.DESC_TAB3);
```

```sql
DBMS_SQL.DESCRIBE_COLUMNS3 (  
  c              IN  INTEGER,  
  col_cnt        OUT INTEGER,  
  desc_t         OUT DBMS_SQL.DESC_REC3);
```

**Pragmas**

```sql
PRAGMA RESTRICT_REFERENCES(describe_columns3,WNDS);
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>ID number of the cursor for the columns being described.</td>
</tr>
<tr>
<td>col_cnt</td>
<td>Number of columns in the select list of the query.</td>
</tr>
<tr>
<td>desc_t</td>
<td>Describe table to fill in with the description of each of the columns of the query. This table is indexed from one to the number of elements in the select list of the query.</td>
</tr>
</tbody>
</table>

**Usage notes**

The cursor passed in by the cursor ID has to be opened and parsed, otherwise an "invalid cursor id" error is raised.
EXECUTE function

This function executes a given cursor. This function accepts the ID number of the cursor and returns the number of rows processed. The return value is only valid for INSERT, UPDATE, and DELETE statements; for other types of statements, including DDL, the return value is undefined and should be ignored.

Syntax

```sql
DBMS_SQL.EXECUTE (
    c   IN INTEGER)
RETURN INTEGER;
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>Cursor ID number of the cursor to execute.</td>
</tr>
</tbody>
</table>

Return value

Returns an integer with the number of rows processed.
EXECUTE_AND_FETCH function

This function executes the given cursor and fetches rows. This function provides the same functionality as calling EXECUTE and then calling FETCH_ROWS. Calling EXECUTE_AND_FETCH instead, however, may reduce the number of network round-trips when used against a remote database.

The EXECUTE_AND_FETCH function returns the number of rows actually fetched.

Syntax

```
DBMS_SQL.EXECUTE_AND_FETCH (  
    c IN INTEGER,  
    exact IN BOOLEAN DEFAULT FALSE)  
RETURN INTEGER;
```

Pragmas

```
pragma restrict_references(execute_and_fetch,WNDS);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>ID number of the cursor to execute and fetch.</td>
</tr>
<tr>
<td>exact</td>
<td>Set to TRUE to raise an exception if the number of rows actually matching the query differs from one. Even if an exception is raised, the rows are still fetched and available.</td>
</tr>
</tbody>
</table>

Return value

Returns an integer with the number of rows that were fetched.
FETCH_ROWS function

This function fetches a row from a given cursor. You can call FETCH_ROWS repeatedly as long as there are rows remaining to be fetched. These rows are retrieved into a buffer, and must be read by calling COLUMN_VALUE, for each column, after each call to FETCH_ROWS.

The FETCH_ROWS function accepts the ID number of the cursor to fetch, and returns the number of rows actually fetched.

Syntax

```sql
DBMS_SQL.FETCH_ROWS (c IN INTEGER) RETURN INTEGER;
```

Pragmas

```sql
pragma restrict_references(fetch_rows,WNDS);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>ID number of the cursor to fetch.</td>
</tr>
</tbody>
</table>

Return value

Returns an integer with the number of rows that were fetched.

Examples

See "Examples" on page 6-11.
**IS_OPEN function**

This function checks to see if the given cursor is currently open.

**Syntax**

```sql
DBMS_SQL.IS_OPEN (  
c  IN INTEGER)    
RETURN BOOLEAN;
```

**Pragmas**

```sql
pragma restrict_references(is_open,RNDS,WNDS);
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>Cursor ID number of the cursor to check.</td>
</tr>
</tbody>
</table>

**Return value**

Returns TRUE for any cursor number that has been opened but not closed, and FALSE for a NULL cursor number. Note that the `CLOSE_CURSOR` procedure nulls out the cursor variable passed to it.

**Exceptions**

ORA-29471 DBMS_SQL access denied

This is raised if an invalid cursor ID number is detected. Once a session has encountered and reported this error, every subsequent DBMS_SQL call in the same session will raise this error, meaning that DBMS_SQL is non-operational for this session.
LAST_ERROR_POSITION function

This function returns the byte offset in the SQL statement text where the error occurred. The first character in the SQL statement is at position 0.

Syntax

```
DBMS_SQL.LAST_ERROR_POSITION
RETURN INTEGER;
```

Pragmas

```
pragma restrict_references(last_error_position,RNDS,WNDS);
```

Return value

Returns an integer with the byte offset in the SQL statement text where the error occurred.

Usage notes

Call this function after a PARSE call, before any other DBMS_SQL procedures or functions are called.
LAST_ROW_COUNT function

This function returns the cumulative count of the number of rows fetched.

Syntax

DBMS_SQL.LAST_ROW_COUNT
RETURN INTEGER;

Pragmas

pragma restrict_references(last_row_count,RNDS,WNDS);

Return value

Returns an integer with the cumulative count of the number of rows that were fetched.

Usage notes

Call this function after a FETCH_ROWS or an EXECUTE_AND_FETCH call. If called after an EXECUTE call, then the value returned is zero.
LAST_ROW_ID function

This function returns the rowid of the last row processed.

Because TimesTen does not support rowid of the last row operated on by a DML statement, this function returns NULL.

Syntax

```sql
DBMS_SQL.LAST_ROW_ID
RETURN ROWID;
```

Pragmas

```sql
pragma restrict_references(last_row_id,RNDS,WNDS);
```

Return value

Returns NULL for TimesTen.
LAST_SQL_FUNCTION_CODE function

This function returns the SQL function code for the statement. These codes are listed in the Oracle Call Interface Programmer’s Guide.

Syntax

```
DBMS_SQL.LAST_SQL_FUNCTION_CODE
    RETURN INTEGER;
```

Pragmas

```
pragma restrict_references(last_sql_function_code,RNDS,WNDS);
```

Return value

Returns an integer with the SQL function code for the statement.

Usage notes

Call this function immediately after the SQL statement is run. Otherwise, the return value is undefined.
OPEN_CURSOR function

This procedure opens a new cursor. The second overload takes a security_level parameter to apply fine-grained control to the security of the opened cursor.

When you no longer need this cursor, you must close it explicitly by calling the CLOSE_CURSOR procedure.

Syntax

```
DBMS_SQL.OPEN_CURSOR
   RETURN INTEGER;

DBMS_SQL.OPEN_CURSOR (security_level IN INTEGER)
   RETURN INTEGER;
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>security_level</td>
<td>Specifies the level of security protection to enforce on the opened cursor. Valid security level values are 0, 1, and 2. When a NULL argument value is provided to this overload, as well as for cursors opened using the overload of <code>open_cursor</code> without the security_level parameter, the default security level value 1 will be enforced on the opened cursor.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Level 0 allows all DBMS_SQL operations on the cursor without any security checks. The cursor may be fetched from, and even re-bound and re-executed, by code running with a different effective userid or roles than those in effect at the time the cursor was parsed. This level of security is off by default.</td>
</tr>
<tr>
<td></td>
<td>- Level 1 requires that the effective userid and roles of the caller to DBMS_SQL for bind and execute operations on this cursor must be the same as those of the caller of the most recent parse operation on this cursor.</td>
</tr>
<tr>
<td></td>
<td>- Level 2 requires that the effective userid and roles of the caller to DBMS_SQL for all bind, execute, define, describe, and fetch operations on this cursor must be the same as those of the caller of the most recent parse operation on this cursor.</td>
</tr>
</tbody>
</table>

Pragmas

```
pragma restrict_references(open_cursor,RNDS,WNDS);
```

Return value

This function returns the cursor ID number of the new cursor.

Usage notes

You can use cursors to run the same SQL statement repeatedly or to run a new SQL statement. When a cursor is reused, the contents of the corresponding cursor data area are reset when the new SQL statement is parsed. It is never necessary to close and reopen a cursor before reusing it.
**PARSE procedure**

This procedure parses the given statement in the given cursor. All statements are parsed immediately. In addition, DDL statements are run immediately when parsed.

There are multiple versions of the PARSE procedure:

- Taking a VARCHAR2 statement as an argument
- Taking VARCHAR2A, table of VARCHAR2(32767), as an argument. The VARCHAR2A overload version of the procedure concatenates elements of a PL/SQL table statement and parses the resulting string. You can use this procedure to parse a statement that is longer than the limit for a single VARCHAR2 variable by splitting up the statement.

**Syntax**

```sql
DBMS_SQL.PARSE(
  c                  IN   INTEGER,
  statement          IN   VARCHAR2,
  language_flag      IN   INTEGER);

DBMS_SQL.PARSE(
  c                  IN   INTEGER,
  statement          IN   DBMS_SQL.VARCHAR2A,
  lb                 IN   INTEGER,
  ub                 IN   INTEGER,
  lfflg              IN   BOOLEAN,
  language_flag      IN   INTEGER);

DBMS_SQL.PARSE(
  c                  IN   INTEGER,
  statement          IN   DBMS_SQL.VARCHAR2S,
  lb                 IN   INTEGER,
  ub                 IN   INTEGER,
  lfflg              IN   BOOLEAN,
  language_flag      IN   INTEGER);
```

**Parameters**

**Table 6–21 PARSE procedure parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>ID number of the cursor in which to parse the statement.</td>
</tr>
<tr>
<td>statement</td>
<td>SQL statement to be parsed. Unlike PL/SQL statements, your SQL statement should not include a final semicolon. For example:</td>
</tr>
<tr>
<td></td>
<td>DBMS_SQL.PARSE(cursor1, ‘BEGIN proc; END;’, 2);</td>
</tr>
<tr>
<td></td>
<td>DBMS_SQL.PARSE(cursor1, ‘INSERT INTO tab VALUES(1)’, 2);</td>
</tr>
<tr>
<td>lb</td>
<td>Lower bound for elements in the statement.</td>
</tr>
<tr>
<td>ub</td>
<td>Upper bound for elements in the statement.</td>
</tr>
<tr>
<td>lfflg</td>
<td>If TRUE, then insert a linefeed after each element on concatenation.</td>
</tr>
<tr>
<td>language_flag</td>
<td>Determines how Oracle handles the SQL statement. For TimesTen, use the NATIVE (or 1) setting, which specifies normal behavior for the database to which the program is connected.</td>
</tr>
</tbody>
</table>
Usage notes

- Because client-side code cannot reference remote package variables or constants, you must explicitly use the values of the constants.

  For example, the following code does not compile on the client:

  ```sql
  DBMS_SQL.PARSE(cur_hdl, stmt_str, DBMS_SQL.NATIVE);
  -- uses constant DBMS_SQL.NATIVE
  ```

  The following code works on the client, because the argument is explicitly provided:

  ```sql
  DBMS_SQL.PARSE(cur_hdl, stmt_str, 1); -- compiles on the client
  ```

- The VARCHAR2S type is supported only for backward compatibility. You are advised to use VARCHAR2A instead.

Exceptions

If you create a type, procedure, function, or package that has compilation warnings, an ORA-24344 exception is raised but the object is still created.

Examples

See "Examples" on page 6-11.
TO_CURSOR_NUMBER function

This function takes an opened strongly or weakly-typed REF CURSOR and transforms it into a DBMS_SQL cursor number.

Syntax

````sql
DBMS_SQL.TO_CURSOR_NUMBER(
    rc IN OUT SYS_REFCURSOR
)
RETURN INTEGER;
````

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rc</td>
<td>REF CURSOR to be transformed into a cursor number.</td>
</tr>
</tbody>
</table>

Return value

Returns a DBMS_SQL manageable cursor number transformed from a REF CURSOR.

Usage notes

- The REF CURSOR passed in has to be opened (OPEN_CURSOR).
- Once the REF CURSOR is transformed into a DBMS_SQL cursor number, the REF CURSOR is no longer accessible by any native dynamic SQL operations.
- Toggling between a REF CURSOR and DBMS_SQL cursor number after a fetch has started is not allowed.

Examples

```sql
CREATE OR REPLACE PROCEDURE DO_QUERY1(sql_stmt VARCHAR2) IS
    TYPE CurType IS REF CURSOR;
    src_cur         CurType;
    curid           NUMBER;
    desctab         DBMS_SQL.DESC_TAB;
    colcnt          NUMBER;
    namevar         VARCHAR2(50);
    numvar          NUMBER;
    datevar         DATE;
BEGIN
    -- sql_stmt := 'select * from employees';
    OPEN src_cur FOR sql_stmt;
    -- Switch from native dynamic SQL to DBMS_SQL
    curid := DBMS_SQL.TO_CURSOR_NUMBER(src_cur);
    DBMS_SQL.DESCRIBE_COLUMNS(curid, colcnt, desctab);
    -- Define columns
    FOR i IN 1 .. colcnt LOOP
        IF desctab(i).col_type = 2 THEN
            DBMS_SQL.DEFINE_COLUMN(curid, i, numvar);
        ELSIF desctab(i).col_type = 12 THEN
```

Table 6–22  TO_CURSOR_NUMBER function parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rc</td>
<td>REF CURSOR to be transformed into a cursor number.</td>
</tr>
</tbody>
</table>
DBMS_SQL.DEFINE_COLUMN(curid, i, datevar);
ELSE
  DBMS_SQL.DEFINE_COLUMN(curid, i, namevar, 25);
END IF;
END LOOP;

-- Fetch Rows
WHILE DBMS_SQL.FETCH_ROWS(curid) > 0 LOOP
  FOR i IN 1 .. colcnt LOOP
    IF (desctab(i).col_type = 1) THEN
      DBMS_SQL.COLUMN_VALUE(curid, i, namevar);
    ELSIF (desctab(i).col_type = 2) THEN
      DBMS_SQL.COLUMN_VALUE(curid, i, numvar);
    ELSIF (desctab(i).col_type = 12) THEN
      DBMS_SQL.COLUMN_VALUE(curid, i, datevar);
    END IF;
  END LOOP;
END LOOP;

DBMS_SQL.CLOSE_CURSOR(curid);
END;
/

You could execute this procedure as follows:

Command> begin
  > do_query1('select * from employees');
  > end;
  > /

PL/SQL procedure successfully completed.
TO_REFCURSOR function

This function takes an opened (OPEN_CURSOR), parsed (PARSE), and executed (EXECUTE) cursor 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. This subprogram is only used with SELECT cursors.

Syntax

DBMS_SQL.TO_REFCURSOR(
    cursor_number IN OUT  INTEGER
) RETURN SYS_REFCURSOR;

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cursor_number</td>
<td>Cursor number of the cursor to be transformed into REF CURSOR.</td>
</tr>
</tbody>
</table>

Return value

Returns a PL/SQL REF CURSOR transformed from a DBMS_SQL cursor number.

Usage notes

- The cursor passed in by the cursor_number has to be opened, parsed, and executed; otherwise an error is raised.
- Once the cursor_number is transformed into a REF CURSOR, the cursor_number is no longer accessible by any DBMS_SQL operations.
- After a cursor_number is transformed into a REF CURSOR, using DBMS_SQL.IS_OPEN to check to see if the cursor_number is still open results in an error.
- Toggle between REF CURSOR and DBMS_SQL cursor number after started to fetch is not allowed, and an error is raised.

Examples

```sql
CREATE OR REPLACE PROCEDURE DO_QUERY2(mgr_id NUMBER) IS
    TYPE CurType IS REF CURSOR;
    src_cur CurType;
    curid NUMBER;
    sql_stmt VARCHAR2(200);
    ret INTEGER;
    empnos DBMS_SQL.Number_Table;
    depts DBMS_SQL.Number_Table;

BEGIN
    -- DBMS_SQL.OPEN_CURSOR
    curid := DBMS_SQL.OPEN_CURSOR;

    sql_stmt :=
        'SELECT EMPLOYEE_ID, DEPARTMENT_ID from employees where MANAGER_ID = :b1';
```
TO_REFCURSOR function

```plsql
DBMS_SQL.PARSE(curid, sql_stmt, DBMS_SQL.NATIVE);
DBMS_SQL.BIND_VARIABLE(curid, 'b1', mgr_id);
ret := DBMS_SQL.EXECUTE(curid);

-- Switch from DBMS_SQL to native dynamic SQL
src_cur := DBMS_SQL.TO_REFCURSOR(curid);

-- Fetch with native dynamic SQL
FETCH src_cur BULK COLLECT INTO empnos, depts;

IF empnos.COUNT > 0 THEN
  DBMS_OUTPUT.PUT_LINE('EMPNO DEPTNO');
  DBMS_OUTPUT.PUT_LINE('----- ------');
  -- Loop through the empnos and depts collections
  FOR i IN 1 .. empnos.COUNT LOOP
    DBMS_OUTPUT.PUT_LINE(empnos(i) || '   ' || depts(i));
  END LOOP;
END IF;

-- Close cursor
CLOSE src_cur;
END;
/
```

The following example executes this procedure for a manager ID of 103.

```
Command> begin
  > do_query2(103);
  > end;
  > /
EMPNO DEPTNO
----- ------
104   60
105   60
106   60
107   60

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

This procedure returns the value of the named variable for a given cursor. It is used to return the values of bind variables inside PL/SQL blocks or of DML statements with a RETURNING clause.

Syntax

DBMS_SQL.VARIABLE_VALUE (c               IN  INTEGER,  
name            IN  VARCHAR2,  
value           OUT NOCOPY <datatype>);

Where <datatype> can be any one of the following types:

BINARY_DOUBLE
BINARY_FLOAT
DATE
INTERVAL DAY TO SECOND(9,9) (DSINTERVAL_UNCONSTRAINED)
NUMBER
TIME(9) (TIME_UNCONSTRAINED)
TIMESTAMP(9) (TIMESTAMP_UNCONSTRAINED)
VARCHAR2 CHARACTER SET ANY_CS
INTERVAL YEAR TO MONTH(9) (YMINTERVAL_UNCONSTRAINED)
VARRAY

Nested table

For variables containing CHAR, RAW, and ROWID data, you can use the following variations on the syntax:

DBMS_SQL.VARIABLE_VALUE_CHAR (c               IN  INTEGER,  
name            IN  VARCHAR2,  
value           OUT CHAR CHARACTER SET ANY_CS);

DBMS_SQL.VARIABLE_VALUE_RAW (c               IN  INTEGER,  
name            IN  VARCHAR2,  
value           OUT RAW);

DBMS_SQL.VARIABLE_VALUE_ROWID (c               IN  INTEGER,  
name            IN  VARCHAR2,  
value           OUT ROWID);

The following syntax enables the VARIABLE_VALUE procedure to accommodate bulk operations:

DBMS_SQL.VARIABLE_VALUE (c               IN  INTEGER,  
name            IN  VARCHAR2,  
value           OUT NOCOPY <table_type>);

For bulk operations, <table_type> can be one of the following:

dbms_sql.Binary_Double_Table
dbms_sql.Binary_Float_Table
dbms_sql.Date_Table
dbms_sql.Interval_Day_To_Second_Table
dbms_sql.Interval_Year_To_Month_Table
VARIABLE_VALUE procedure

- dbms_sql.Number_Table
- dbms_sql.Time_Table
- dbms_sql.Timestamp_Table
- dbms_sql.Varchar2_Table

Pragmas

```
pragma restrict_references(variable_value,RNDS,WNDS);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>ID number of the cursor from which to get the values.</td>
</tr>
<tr>
<td>name</td>
<td>Name of the variable for which you are retrieving the value.</td>
</tr>
<tr>
<td>value</td>
<td>For the single row option, this is the value of the variable for the specified position. Oracle raises the exception ORA-06562, inconsistent_type, if the type of this output parameter differs from the actual type of the value, as defined by the call to BIND_VARIABLE. For the array option, this is the local variable that has been declared &lt;table_type&gt;. For bulk operations, value is an OUT NOCOPY parameter.</td>
</tr>
</tbody>
</table>

Examples

See "Examples" on page 6-11.
The DBMS_UTILITY package provides various utility subprograms.

This chapter contains the following topics:

- Using DBMS_UTILITY
  - Security model
  - Constants
  - Types
  - Exceptions
- Summary of DBMS_UTILITY subprograms
Using DBMS_UTILITY

- Security model
- Constants
- Types
- Exceptions
Security model

DBMS_UTIL runs with the privileges of the calling user for the NAME_RESOLVE procedure and the COMPILE_SCHEMA procedure. This is necessary so that the SQL works correctly.

The package does not run as SYSC.
The DBMS_UTILITY package uses the constants shown in Table 7–1.

### Table 7–1 DBMS_UTILITY constants

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INV_ERROR_ON_RESTRICTIONS</td>
<td>BINARY_INTEGER</td>
<td>1</td>
<td>This constant is the only legal value for the p_option_flags parameter of the INVALIDATE subprogram.</td>
</tr>
</tbody>
</table>

**Note:** Notes on data types:

- The PLS_INTEGER and BINARY_INTEGER data types are identical. This document uses "BINARY_INTEGER" to indicate data types in reference information (such as for table types, record types, subprogram parameters, or subprogram return values), but may use either in discussion and examples.

- The INTEGER and NUMBER(38) data types are also identical. This document uses "INTEGER" throughout.
Types

- dblink_array
- index_table_type
- instance_record
- lname_array
- name_array
- number_array
- uncl_array

**dblink_array**

TYPE dblink_array IS TABLE OF VARCHAR2(128) INDEX BY BINARY_INTEGER;

Lists of database links should be stored here.

**index_table_type**

TYPE index_table_type IS TABLE OF BINARY_INTEGER INDEX BY BINARY_INTEGER;

The order in which objects should be generated is returned here.

**instance_record**

TYPE instance_record IS RECORD (  
    inst_number   NUMBER,
    inst_name     VARCHAR2(60));

TYPE instance_table IS TABLE OF instance_record INDEX BY BINARY_INTEGER;

The list of active instance number and instance name.  
The starting index of instance_table is 1; instance_table is dense.

**lname_array**

TYPE lname_array IS TABLE OF VARCHAR2(4000) INDEX BY BINARY_INTEGER;

Lists of Long NAME should be stored here, it includes fully qualified attribute names.

**name_array**

TYPE name_array IS TABLE OF VARCHAR2(30) INDEX BY BINARY_INTEGER;

Lists of NAME should be stored here.

**number_array**

TYPE number_array IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;

The order in which objects should be generated is returned here for users.

**uncl_array**

TYPE uncl_array IS TABLE OF VARCHAR2(227) INDEX BY BINARY_INTEGER;

Lists of "USER"."NAME"."COLUMN"@LINK should be stored here.
The following table lists the exceptions raised by DBMS_UTILITY.

<table>
<thead>
<tr>
<th>Exception</th>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INV_NOT_EXIST_OR_NO_PRIV</td>
<td>-24237</td>
<td>Raised by the INVALIDATE subprogram when the object_id argument is NULL or invalid, or when the caller does not have CREATE privileges on the object being invalidated.</td>
</tr>
<tr>
<td>INV_MALFORMED_SETTINGS</td>
<td>-24238</td>
<td>Raised by the INVALIDATE subprogram if a compiler setting is specified more than once in the p_plsql_object_settings parameter.</td>
</tr>
<tr>
<td>INV_RESTRICTED_OBJECT</td>
<td>-24239</td>
<td>Raised by the INVALIDATE subprogram when different combinations of conditions pertaining to the p_object_id parameter are contravened.</td>
</tr>
</tbody>
</table>
### Summary of DBMS_UTILITY subprograms

<table>
<thead>
<tr>
<th>Subprogram</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CANONICALIZE procedure</strong></td>
<td>Canonicalizes a given string.</td>
</tr>
<tr>
<td><strong>COMMA_TO_TABLE procedure</strong></td>
<td>Converts a comma-delimited list of names into a PL/SQL table of names.</td>
</tr>
<tr>
<td><strong>COMPILE_SCHEMA procedure</strong></td>
<td>Compiles all procedures, functions, packages, and views in the specified schema.</td>
</tr>
<tr>
<td><strong>DB_VERSION procedure</strong></td>
<td>Returns version information for the database. Returns NULL for the compatibility setting because TimesTen does not support the system parameter COMPATIBLE.</td>
</tr>
<tr>
<td><strong>FORMAT_CALL_STACK function</strong></td>
<td>Formats the current call stack.</td>
</tr>
<tr>
<td><strong>FORMAT_ERROR_BACKTRACE function</strong></td>
<td>Formats the backtrace from the point of the current error to the exception handler where the error has been caught.</td>
</tr>
<tr>
<td><strong>FORMAT_ERROR_STACK function</strong></td>
<td>Formats the current error stack.</td>
</tr>
<tr>
<td><strong>GET_CPU_TIME function</strong></td>
<td>Returns the current CPU time in hundredths of a second.</td>
</tr>
<tr>
<td><strong>GET_DEPENDENCY procedure</strong></td>
<td>Shows the dependencies on the object passed in.</td>
</tr>
<tr>
<td><strong>GET_ENDIANNESS function</strong></td>
<td>Returns the endianness of your database platform.</td>
</tr>
<tr>
<td><strong>GET_HASH_VALUE function</strong></td>
<td>Computes a hash value for the given string.</td>
</tr>
<tr>
<td><strong>GET_SQL_HASH function</strong></td>
<td>Computes the hash value for a given string using the MD5 algorithm.</td>
</tr>
<tr>
<td><strong>GET_TIME function</strong></td>
<td>Finds out the current time in hundredths of a second.</td>
</tr>
<tr>
<td><strong>INVALIDATE procedure</strong></td>
<td>Invalidates a database object and (optionally) modifies its PL/SQL compiler parameter settings.</td>
</tr>
<tr>
<td><strong>IS_BIT_SET function</strong></td>
<td>Returns bit setting.</td>
</tr>
<tr>
<td><strong>NAME_RESOLVE procedure</strong></td>
<td>Resolves the given name of the form <code>[[a.]b.[c][@dblink]]</code>, where a, b, and c are SQL identifiers and d is a dblink. Do not use @d. TimesTen does not support database links.</td>
</tr>
<tr>
<td><strong>NAME_TOKENIZE procedure</strong></td>
<td>Calls the parser to parse the given name <code>&quot;a[,b[,c]@dblink]&quot;</code>, where a, b, and c are SQL identifiers and d is a 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 database links.</td>
</tr>
<tr>
<td><strong>TABLE_TO_COMMA procedure</strong></td>
<td>Converts a PL/SQL table of names into a comma-delimited list of names.</td>
</tr>
<tr>
<td><strong>VALIDATE procedure</strong></td>
<td>Validates the object described either by owner, name, and namespace or by object ID.</td>
</tr>
</tbody>
</table>
CANONICALIZE procedure

This procedure canonicalizes the given string. The procedure handles a single reserved or key word (such as "table"), and strips off white spaces for a single identifier. For example, "table" becomes TABLE.

Syntax

```sql
DBMS_UTILITY.CANONICALIZE(
    name           IN    VARCHAR2,
    canon_name     OUT   VARCHAR2,
    canon_len      IN    BINARY_INTEGER);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>The string to be canonicalized.</td>
</tr>
<tr>
<td>canon_name</td>
<td>The canonicalized string.</td>
</tr>
<tr>
<td>canon_len</td>
<td>The length of the string (in bytes) to canonicalize.</td>
</tr>
</tbody>
</table>

Return value

Returns the first `canon_len` bytes in `canon_name`.

Usage notes

- If `name` is NULL, `canon_name` becomes NULL.
- If `name` is not a dotted name, and if `name` begins and ends with a double quote, remove both quotes. Alternatively, convert to upper case with NLS_UPPER. Note that this case does not include a name with special characters, such as a space, but is not doubly quoted.
- If `name` is a dotted name (such as `a."b".c`), for each component in the dotted name in the case in which the component begins and ends with a double quote, no transformation will be performed on this component. Alternatively, convert to upper case with NLS_UPPER and apply begin and end double quotes to the capitalized form of this component. In such a case, each canonicalized component will be concatenated together in the input position, separated by ".".
- Any other character after `a[.b]*` will be ignored.
- The procedure does not handle cases like ‘A B.’

Examples

- `a` becomes `A`.
- "a" becomes `a`.
- "a".b becomes "a"."B".
- "a".b,c.f becomes "a"."B" with".c.f" ignored.
COMMA_TO_TABLE procedure

These procedures converts a comma-delimited list of names into a PL/SQL table of names. The second version supports fully qualified attribute names.

Syntax

```plaintext
DBMS_UTILITY.COMMA_TO_TABLE (  
    list   IN  VARCHAR2,  
    tablen OUT BINARY_INTEGER,  
    tab    OUT dbms_utility.uncl_array);

DBMS_UTILITY.COMMA_TO_TABLE (  
    list   IN  VARCHAR2,  
    tablen OUT BINARY_INTEGER,  
    tab    OUT dbms_utility.lname_array);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
</table>
| list      | Comma-delimited list of names, where a name should have the following format for the first overloading: 
            a [. b [. c ]][ @ d ]  
            And should have the following format for the second overloading: 
            a [. b]*  
            Where a, b, c, d are simple identifiers (quoted or unquoted). |
| tablen    | Number of tables in the PL/SQL table. |
| tab       | PL/SQL table which contains list of names. |

Return value

A PL/SQL table is returned with values 1..n, and n+1 is NULL.

Usage notes

The list must be a non-empty comma-delimited list. Anything other than a comma-delimited list is rejected. Commas inside double quotes do not count. Entries in the comma-delimited list cannot include multibyte characters. The values in tab are copied from the original list, with no transformations.
COMPILE_SCHEMA procedure

This procedure compiles all procedures, functions, packages, and views in the specified schema.

Syntax

```
DBMS_UTILITY.COMPILE_SCHEMA (
    schema          IN VARCHAR2,
    compile_all     IN BOOLEAN DEFAULT TRUE,
    reuse_settings  IN BOOLEAN DEFAULT FALSE);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>schema</td>
<td>Name of the schema.</td>
</tr>
<tr>
<td>compile_all</td>
<td>If TRUE, will compile everything within the schema regardless of whether it is VALID.</td>
</tr>
<tr>
<td></td>
<td>If FALSE, will compile only INVALID objects.</td>
</tr>
<tr>
<td>reuse_settings</td>
<td>Indicates whether the session settings in the objects should be reused, or whether the current session settings should be adopted instead.</td>
</tr>
</tbody>
</table>

Exceptions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORA-20000</td>
<td>Insufficient privileges for some object in this schema.</td>
</tr>
<tr>
<td>ORA-20001</td>
<td>Cannot recompile SYS objects.</td>
</tr>
<tr>
<td>ORA-20002</td>
<td>Maximum iterations exceeded. Some objects may not have been recompiled.</td>
</tr>
</tbody>
</table>

Usage notes

- Note that this subprogram is a wrapper for RECOMP_SERIAL procedure included with the UTL_RECOMP package.
- After calling this procedure, you should select from view ALL_OBJECTS for items with status of INVALID to see if all objects were successfully compiled.
- To see the errors associated with invalid objects, you can use the ttIsql show errors command:
  
  Command> show errors [{FUNCTION | PROCEDURE | PACKAGE | PACKAGE BODY} [schema.]name];

Examples:

Command> show errors function foo;
Command> show errors procedure fred.bar;
Command> show errors package body emp_actions;
DB_VERSION procedure

This procedure returns version information for the database.

Returns NULL for the compatibility setting because TimesTen does not support the system parameter COMPATIBLE.

Syntax

```sql
DBMS_UTILITY.DB_VERSION (
    version       OUT VARCHAR2,
    compatibility OUT VARCHAR2);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>version</td>
<td>A string which represents the internal software version of the database (for example, 7.1.0.0.0). The length of this string is variable and is determined by the database version.</td>
</tr>
<tr>
<td>compatibility</td>
<td>The compatibility setting of the database.</td>
</tr>
<tr>
<td></td>
<td>In TimesTen, DB_VERSION returns NULL for the compatibility setting because TimesTen does not support the system parameter COMPATIBLE.</td>
</tr>
</tbody>
</table>
FORMAT_CALL_STACK function

This function formats the current call stack. This can be used on any stored procedure to access the call stack. This can be useful for debugging.

Syntax

```
DBMS_UTILITY.FORMAT_CALL_STACK
RETURN VARCHAR2;
```

Pragmas

```
pragma restrict_references(format_call_stack,WNDS);
```

Return value

This returns the call stack, up to 2000 bytes.
FORMAT_ERROR_BACKTRACE function

This procedure displays the call stack at the point where an exception was raised, even if the procedure is called from an exception handler in an outer scope. The output is similar to the output of the SQLERRM function, but not subject to the same size limitation.

Syntax

DBMS_UTILITY.FORMAT_ERROR_BACKTRACE
RETURN VARCHAR2;

Return value

The backtrace string. A NULL string is returned if no error is currently being handled.

Examples

Script format_error_backtrace.sql
Execute the following script from ttIsql, using the run command.

CREATE OR REPLACE PROCEDURE Log_Errors ( i_buff in varchar2 ) IS
  g_start_pos integer := 1;
  g_end_pos  integer;

  FUNCTION Output_One_Line RETURN BOOLEAN IS
  BEGIN
    g_end_pos := Instr ( i_buff, Chr(10), g_start_pos );
    CASE g_end_pos > 0
    WHEN true THEN
      DBMS_OUTPUT.PUT_LINE ( Substr ( i_buff, g_start_pos,
               g_end_pos-g_start_pos ) );
      g_start_pos := g_end_pos+1;
      RETURN TRUE;
    WHEN FALSE THEN
      DBMS_OUTPUT.PUT_LINE ( Substr ( i_buff, g_start_pos,
               (Length(i_buff)-g_start_pos)+1 ) );
      RETURN FALSE;
    END CASE;
  END Output_One_Line;

  BEGIN
    WHILE Output_One_Line() LOOP NULL;
  END LOOP;
  END Log_Errors;
/

CREATE OR REPLACE PROCEDURE P0 IS
  e_01476 EXCEPTION; pragma exception_init ( e_01476, -1476 );
BEGIN
  RAISE e_01476;
END P0;
/
Show Errors

CREATE OR REPLACE PROCEDURE P1 IS
BEGIN
  P0();
END P1;
/
SHOW ERRORS

CREATE OR REPLACE PROCEDURE P2 IS
BEGIN
  P1();
END P2;
/
SHOW ERRORS

CREATE OR REPLACE PROCEDURE P3 IS
BEGIN
  P2();
END P3;
/
SHOW ERRORS

CREATE OR REPLACE PROCEDURE P4 IS
BEGIN
  P3();
END P4;
/
CREATE OR REPLACE PROCEDURE P5 IS
BEGIN
  P4();
END P5;
/
SHOW ERRORS

CREATE OR REPLACE PROCEDURE Top_Naive IS
BEGIN
  P5();
END Top_Naive;
/
SHOW ERRORS

CREATE OR REPLACE PROCEDURE Top_With_Logging IS
  -- NOTE: SqlErrm in principle gives the same info as Format_Error_Stack.
  -- But SqlErrm is subject to some length limits,
  -- while Format_Error_Stack is not.
BEGIN
  P5();
  EXCEPTION
  WHEN OTHERS THEN
    Log_Errors ( 'Error_Stack...' || Chr(10) ||
                  DBMS_UTILITY.FORMAT_ERROR_STACK() );
    Log_ Errors ( 'Error_Backtrace...' || Chr(10) ||
                  DBMS_UTILITY.FORMAT_ERROR_BACKTRACE() );
    DBMS_OUTPUT.PUT_LINE ( '----------' );
END Top_With_Logging;
/
SHOW ERRORS

**Execute Top_Naive**

This shows the results of executing the Top_Naive procedure that is created in the script, assuming user SCOTT ran the script and executed the procedure.
Command> set serveroutput on
Command> begin
  > Top_Naive();
  > end;
  > /
8507: ORA-01476: divisor is equal to zero
8507: ORA-06512: at "SCOTT.P0", line 4
8507: ORA-06512: at "SCOTT.P1", line 3
8507: ORA-06512: at "SCOTT.P2", line 3
8507: ORA-06512: at "SCOTT.P3", line 3
8507: ORA-06512: at "SCOTT.P4", line 3
8507: ORA-06512: at "SCOTT.P5", line 3
8507: ORA-06512: at "SCOTT.TOP_NAIVE", line 3
8507: ORA-06512: at line 2
The command failed.

This output shows the call stack at the point where an exception was raised. It shows the backtrace error message as the call stack unwound, starting at the unhandled exception ORA-01476 raised at SCOTT.P0 line 4, back to SCOTT.Top_Naive line 3.

**Execute Top_With_Logging**

This shows the results of executing the Top_With_Logging() procedure that is created in the script, assuming user SCOTT ran the script and executed the procedure.

Command> begin
  > Top_With_Logging();
  > end;
  > /
Error_Stack...
ORA-01476: divisor is equal to zero
Error_Backtrace...
ORA-06512: at "SCOTT.P0", line 4
ORA-06512: at "SCOTT.P1", line 3
ORA-06512: at "SCOTT.P2", line 3
ORA-06512: at "SCOTT.P3", line 3
ORA-06512: at "SCOTT.P4", line 3
ORA-06512: at "SCOTT.P5", line 3
ORA-06512: at "SCOTT.TOP_WITH_LOGGING", line 6
-----------
PL/SQL procedure successfully completed.

This output shows the call stack at the point where an exception was raised. It shows the backtrace error message as the call stack unwound, starting at the unhandled exception ORA-01476 raised at SCOTT.P0 line 4, back to SCOTT.Top_With_Logging line 6.

**ORA-06512 information**

*Oracle Database Error Messages* provides the following information about the ORA-06512 error:

ORA-06512: at stringline string
  Cause: Backtrace message as the stack is unwound by unhandled exceptions.
  Action: Fix the problem causing the exception or write an exception handler for this condition. Or you may need to contact your application administrator or DBA.
FORMAT_ERROR_STACK function

This function formats the current error stack. This can be used in exception handlers to look at the full error stack.

Syntax

```
DBMS_UTILITY.FORMAT_ERROR_STACK
RETURN VARCHAR2;
```

Return value

This returns the error stack, up to 2000 bytes. A NULL string is returned if no error is currently being handled.
GET_CPU_TIME function

This function returns the current CPU time in hundredths of a second. The returned CPU time is the number of hundredths of a second from some arbitrary epoch.

Syntax

```
DBMS_UTILITY.GET_CPU_TIME
RETURN NUMBER;
```

Return value

Time is the number of hundredths of a second from some arbitrary epoch.

Usage notes

This subprogram reports cycles (CPU time) used in performing work, and is unrelated to clock time or any other fixed reference. Since the base line is rubricator, and the relationship between work performed and the number generated is operating system specific, the amount of work performed is calculated by measuring the difference between a start point and end point for a particular operation.
GET_DEPENDENCY procedure

This procedure shows the dependencies on the object passed in.

Syntax

```sql
DBMS_UTILITY.GET_DEPENDENCY
type IN VARCHAR2,
schema IN VARCHAR2,
name IN VARCHAR2);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>The type of the object. For example, if the object is a table give the type as &quot;TABLE&quot;.</td>
</tr>
<tr>
<td>schema</td>
<td>The schema name of the object.</td>
</tr>
<tr>
<td>name</td>
<td>The name of the object.</td>
</tr>
</tbody>
</table>

Usage notes

This procedure uses the DBMS_OUTPUT package to display results, so you must declare SET SERVEROUTPUT ON if you wish to view dependencies. Alternatively, any application that checks the DBMS_OUTPUT output buffers can invoke this subprogram and then retrieve the output by means of DBMS_OUTPUT subprograms such as GET_LINES.
**GET_ENDIANNESS function**

This function gets the endianness of the database platform.

**Syntax**

```sql
DBMS_UTILITY.GET_ENDIANNESS
RETURN NUMBER;
```

**Return value**

The endianness of the database platform.
GET_HASH_VALUE function

This function computes a hash value for the given string.

Syntax

```
DBMS_UTILITY.GET_HASH_VALUE (  
    name      IN  VARCHAR2,
    base      IN  NUMBER,
    hash_size IN  NUMBER)    
RETURN NUMBER;
```

Parameters

```
Table 7–10  GET_HASH_VALUE function parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String to be hashed.</td>
</tr>
<tr>
<td>base</td>
<td>Base value where the returned hash value is to start.</td>
</tr>
<tr>
<td>hash_size</td>
<td>Desired size of the hash table.</td>
</tr>
</tbody>
</table>
```

Pragmas

```
pragma restrict_references(get_hash_value, WNDS, RNDS, WNPS, RNPS);
```

Return value

A hash value based on the input string. For example, to get a hash value on a string where the hash value should be between 1000 and 3047, use 1000 as the base value and 2048 as the hash_size value. Using a power of 2 for the hash_size parameter works best.
GET_SQL_HASH function

This function computes a hash value for the given string using MD5 algorithm.

Syntax

```sql
DBMS_UTILITY.GET_SQL_HASH (
    name          IN   VARCHAR2,
    hash          OUT  RAW,
    pre10ihash    OUT  NUMBER)
RETURN NUMBER;
```

Pragmas

```sql
pragma restrict_references(GET_SQL_HASH, WNDS, RNDS, WNPS, RNPS);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String to be hashed.</td>
</tr>
<tr>
<td>hash</td>
<td>An optional field to store all 16 bytes of returned hash value.</td>
</tr>
<tr>
<td>pre10ihash</td>
<td>An optional field to store a pre-Oracle 10g database version hash value.</td>
</tr>
</tbody>
</table>

Return value

A hash value (last 4 bytes) based on the input string. The MD5 hash algorithm computes a 16 byte hash value, but we only return the last 4 bytes so that we can return an actual number. One could use an optional RAW parameter to get all 16 bytes and to store the pre-10g hash value of 4 bytes in the pre10ihash optional parameter.
GET_TIME function

This function determines the current time in hundredths of a second and is primarily used for determining elapsed time. The subprogram is called twice, at the beginning and end of a process. The first (earlier) number is subtracted from the second (later) number to determine the time elapsed.

Syntax

```
DBMS_UTILITY.GET_TIME
RETURN NUMBER;
```

Return value

Time is the number of hundredths of a second from the point in time at which the subprogram is invoked.

Usage notes

Numbers are returned in the range -2147483648 to 2147483647 depending on platform and system, and your application must take the sign of the number into account in determining the interval. For instance, in the case of two negative numbers, application logic must allow that the first (earlier) number will be larger than the second (later) number which is closer to zero. By the same token, your application should also allow that the first (earlier) number be negative and the second (later) number be positive.
INVALIDATE procedure

This procedure invalidates a database object and (optionally) modifies its PL/SQL compiler parameter settings. It also invalidates any objects that directly or indirectly depend on the object being invalidated.

Syntax

```sql
DBMS_UTILITY.INVALIDATE (
    p_object_id              IN  NUMBER,
    p_plsql_object_settings  IN  VARCHAR2 DEFAULT NULL,
    p_option_flags           BINARY_INTEGER DEFAULT 0);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>p_object_id</td>
<td>ID number of object to be invalidated. This is the same as the value of the OBJECT_ID column from ALL_OBJECTS. If the p_object_id argument is NULL or invalid then the exception inv_not_exist_or_no_priv is raised. The caller of this procedure must have the CREATE privilege on the object being invalidated, otherwise the inv_not_exist_or_no_priv exception is raised.</td>
</tr>
<tr>
<td>p_plsql_object_settings</td>
<td>This optional parameter is ignored if the object specified by p_object_id is not a PL/SQL object. If no value is specified for this parameter then the PL/SQL compiler settings are left unchanged, that is, equivalent to REUSE_SETTINGS. If a value is provided, it must specify the values of the PL/SQL compiler settings separated by one or more spaces. Each setting can be specified only once else inv_malformed_settings exception will be raised. The setting values are changed only for the object specified by p_object_id and do not affect dependent objects that may be invalidated. The setting names and values are case insensitive. If a setting is omitted and REUSE_SETTINGS is specified, then if a value was specified for the compiler setting in an earlier compilation of this library unit, TimesTen uses that earlier value. If a setting is omitted and REUSE_SETTINGS was not specified or no value has been specified for the parameter in an earlier compilation, then the database will obtain the value for that setting from the session environment.</td>
</tr>
<tr>
<td>p_option_flags</td>
<td>This parameter is optional and defaults to zero (no flags). Option flags supported by invalidate.</td>
</tr>
<tr>
<td></td>
<td>- inv_error_on_restrictions (see &quot;Constants&quot; on page 7-4): The subprogram imposes various restrictions on the objects that can be invalidated. For example, the object specified by p_object_id cannot be a table. By default, invalidate quietly returns on these conditions (and does not raise an exception). If the caller sets this flag, the exception inv_restricted_object is raised.</td>
</tr>
</tbody>
</table>
Exceptions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INV_NOT_EXIST_OR_NO_PRIV</td>
<td>Raised when the object_id argument is NULL or invalid, or when the caller does not have CREATE privilege on the object being invalidated.</td>
</tr>
<tr>
<td>INV_MALFORMED_SETTINGS</td>
<td>Raised if a compiler setting is specified more than once in the p_plsql_object_settings parameter.</td>
</tr>
<tr>
<td>INV_RESTRICTED_OBJECT</td>
<td>Raised when different combinations of conditions pertaining to the p_object_id parameter are contravened.</td>
</tr>
</tbody>
</table>

Usage notes

The object type (object_type column from ALL_OBJECTS) of the object specified by p_object_id must be a PROCEDURE, FUNCTION, PACKAGE, PACKAGE BODY, LIBRARY, OPERATOR, or SYNONYM. If the object is not one of these types and the flag inv_error_on_restrictions is specified in p_option_flags then the exception inv_restricted_object is raised, else no action is taken.

If the object specified by p_object_id is the package specification of STANDARD, DBMS_STANDARD, or specification or body of DBMS_UTILITY and the flag inv_error_on_restrictions is specified in p_option_flags then the exception inv_restricted_object is raised, else no action is taken.

If the object specified by p_object_id is an object type specification and there exist tables which depend on the type and the flag inv_error_on_restrictions is specified in p_option_flags then the exception inv_restricted_object is raised, else no action is taken.

Examples

This example invalidates one of the procedures created in the example in "FORMAT_ERROR_BACKTRACE function" on page 7-13. From examining user_objects, you can see information for the procedures created in that example. The following describes user_objects then queries its contents.

Command> describe user_objects;

View SYS.USER_OBJECTS:

Columns:

- OBJECT_NAME VARCHAR2 (30) INLINE
- SUBOBJECT_NAME VARCHAR2 (30) INLINE
- OBJECT_ID TT_BIGINT NOT NULL
- DATA_OBJECT_ID TT_BIGINT
- OBJECT_TYPE VARCHAR2 (17) INLINE NOT NULL
- CREATED DATE NOT NULL
- LAST_DDL_TIME DATE NOT NULL
- TIMESTAMP VARCHAR2 (78) INLINE NOT NULL
- STATUS VARCHAR2 (7) INLINE NOT NULL
- TEMPORARY VARCHAR2 (1) INLINE NOT NULL
- GENERATED VARCHAR2 (1) INLINE NOT NULL
- SECONDARY VARCHAR2 (1) INLINE NOT NULL
- NAMESPACE TT_INTEGER NOT NULL
- EDITION_NAME VARCHAR2 (30) INLINE
1 view found.

Command> select * from user_objects;
...
...
20 rows found.

To invalidate the P5 procedure, for example, specify object_id 302 in the INVALIDATE call:

Command> begin
> dbms_utility.invalidate(302, 'PLSQL_OPTIMIZE_LEVEL=2 REUSE SETTINGS');
> end;
> /

This will mark the P5 procedure invalid and set its PLSQL_OPTIMIZE_LEVEL compiler setting to 2. The values of other compiler settings will remain unchanged because REUSE SETTINGS is specified. Note that in addition to P5 being invalidated, any PL/SQL objects that refer to that object are invalidated. Given that Top_With_Logging and Top_Naive call P5, here are the results of the INVALIDATE call, querying for all user objects that are now invalid:

Command> select * from user_objects where status='INVALID';
< TOP_NAIVE, <NULL>, 303, <NULL>, PROCEDURE, 2009-09-18 12:53:45, 2009-09-18 12:58:22, INVALID, N, N, N, 1, <NULL> >
3 rows found.

A user can explicitly recompile and revalidate an object by calling the VALIDATE procedure discussed later in this chapter, or by executing ALTER PROCEDURE, ALTER FUNCTION, or ALTER PACKAGE, as applicable, on the object. Alternatively, each object is recompiled and revalidated automatically the next time it is executed.
IS_BIT_SET function

This function checks the bit setting for the given bit in the given RAW value.

Syntax

```
DBMS_UTILITY.IS_BIT_SET (
    r       IN RAW,
    n       IN NUMBER)
RETURN NUMBER;
```

Pragmas

```
pragma restrict_references (IS_BIT_SET, WNDS, RNDS, WNPS, RNPS);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>Source raw.</td>
</tr>
<tr>
<td>n</td>
<td>Which bit in r to check.</td>
</tr>
</tbody>
</table>

Return value

This function returns 1 if bit \( n \) in RAW \( r \) is set. Bits are numbered high to low with the lowest bit being bit number 1.
NAME_RESOLVE procedure

This 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, including synonym translation and authorization checking as necessary.

Do not use @d. TimesTen does not support database links.

Syntax

```sql
DBMS_UTILITY.NAME_RESOLVE (
    name          IN  VARCHAR2,
    context       IN  NUMBER,
    schema        OUT VARCHAR2,
    part1         OUT VARCHAR2,
    part2         OUT VARCHAR2,
    dblink        OUT VARCHAR2,
    part1_type    OUT NUMBER,
    object_number OUT NUMBER);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the object. This can be of the form [[a.]b.]c[@d]], where a, b, and c are SQL identifiers and d is a dblink. No syntax checking is performed on the dblink. If a dblink is specified, or if the name resolves to something with a dblink, then object is not resolved, but the schema, part1, part2, and dblink OUT parameters are filled in. The a, b and c entries may be delimited identifiers, and may contain Globalization Support (NLS) characters, either single or multibyte.</td>
</tr>
<tr>
<td>context</td>
<td>Must be an integer between 0 and 9.</td>
</tr>
<tr>
<td></td>
<td>■ 0 - table</td>
</tr>
<tr>
<td></td>
<td>■ 1 - PL/SQL (for 2 part names)</td>
</tr>
<tr>
<td></td>
<td>■ 2 - sequences</td>
</tr>
<tr>
<td></td>
<td>■ 3 - trigger (not applicable for TimesTen)</td>
</tr>
<tr>
<td></td>
<td>■ 4 - Java source (not applicable for TimesTen)</td>
</tr>
<tr>
<td></td>
<td>■ 5 - Java resource (not applicable for TimesTen)</td>
</tr>
<tr>
<td></td>
<td>■ 6 - Java class (not applicable for TimesTen)</td>
</tr>
<tr>
<td></td>
<td>■ 7 - type (not applicable for TimesTen)</td>
</tr>
<tr>
<td></td>
<td>■ 8 - Java shared data (not applicable for TimesTen)</td>
</tr>
<tr>
<td></td>
<td>■ 9 - index</td>
</tr>
<tr>
<td>schema</td>
<td>Schema of the object: c. If no schema is specified in name, then the schema is determined by resolving the name.</td>
</tr>
<tr>
<td>part1</td>
<td>First part of the name. The type of this name is specified part1_type (synonym or package).</td>
</tr>
<tr>
<td>part2</td>
<td>If this is non-NULL, then this is a subprogram name. If part1 is non-NULL, then the subprogram is within the package indicated by part1. If part1 is NULL, the subprogram is a top-level subprogram.</td>
</tr>
</tbody>
</table>
Table 7–15 (Cont.) NAME_RESOLVE procedure parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dblink</td>
<td>If this is non-NULL, then a database link was specified as part of <code>name</code>,</td>
</tr>
<tr>
<td></td>
<td>or <code>name</code> was a synonym that resolved to something with a database link.</td>
</tr>
<tr>
<td></td>
<td>In this case, if further name translation is desired, then you must call</td>
</tr>
<tr>
<td></td>
<td>the DBMS_UTILITY.NAME_RESOLVE procedure on this remote node.</td>
</tr>
<tr>
<td></td>
<td>TimesTen does not support database links.</td>
</tr>
<tr>
<td>part1_type</td>
<td>Type of <code>part1</code> is:</td>
</tr>
<tr>
<td></td>
<td>■ 5 - synonym</td>
</tr>
<tr>
<td></td>
<td>■ 7 - procedure (top level)</td>
</tr>
<tr>
<td></td>
<td>■ 8 - function (top level)</td>
</tr>
<tr>
<td></td>
<td>■ 9 - package</td>
</tr>
<tr>
<td>object_number</td>
<td>Object identifier.</td>
</tr>
</tbody>
</table>

Exceptions

All errors are handled by raising exceptions. A wide variety of exceptions are possible, based on the various syntax error that are possible when specifying object names.
NAME_TOKENIZE procedure

This procedure calls the parser to parse the given name as "a [.b [.c]] [@dblink] ", where a, b, and c are SQL identifiers and dblink is a dblink. It strips double quotes, or converts to uppercase if there are no quotes. It ignores comments of all sorts, and does no semantic analysis. Missing values are left as NULL.

Do not use @dblink. TimesTen does not support database links.

Syntax

```sql
DBMS_UTILITY.NAME_TOKENIZE (  
    name IN VARCHAR2,  
    a OUT VARCHAR2,  
    b OUT VARCHAR2,  
    c OUT VARCHAR2,  
    dblink OUT VARCHAR2,  
    nextpos OUT BINARY_INTEGER);
```

Parameters

For each of a, b, c, and dblink, tell where the following token starts in anext, bnext, cnext, dnext respectively.
TABLE_TO_COMMA procedure

These procedures converts a PL/SQL table of names into a comma-delimited list of names. This takes a PL/SQL table, 1..n, terminated with n+1 being NULL. The second version supports fully qualified attribute names.

Syntax

```sql
DBMS_UTILITY.TABLE_TO_COMMA (  
    tab    IN  dbms_utility.uncl_array,
    tablen OUT BINARY_INTEGER,
    list   OUT VARCHAR2);

DBMS_UTILITY.TABLE_TO_COMMA (  
    tab    IN  dbms_utility.lname_array,
    tablen OUT BINARY_INTEGER,
    list   OUT VARCHAR2);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tab</td>
<td>PL/SQL table which contains list of table names.</td>
</tr>
<tr>
<td>tablen</td>
<td>Number of tables in the PL/SQL table.</td>
</tr>
<tr>
<td>list</td>
<td>Comma-delimited list of tables.</td>
</tr>
</tbody>
</table>

Return value

A comma-delimited list and the number of elements found in the table.
VALIDATE procedure

Validates the object described either by owner, name, and namespace or by object ID.

Syntax

DBMS_UTILITY.VALIDATE(
    object_id     IN  NUMBER);

DBMS_UTILITY.VALIDATE(
    owner          IN  VARCHAR2,
    objname        IN  VARCHAR2,
    namespace      NUMBER,
    edition_name   VARCHAR2 := NULL;

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>Name of the user who owns the object. Same as the OWNER field in ALL_OBJECTS.</td>
</tr>
<tr>
<td>objname</td>
<td>Name of the object to be validated. Same as the OBJECT_NAME field in ALL_OBJECTS.</td>
</tr>
<tr>
<td>namespace</td>
<td>Namespace of the object. Same as the namespace field in obj$. Equivalent numeric values are as follows:</td>
</tr>
<tr>
<td></td>
<td>- 1 - table or procedure</td>
</tr>
<tr>
<td></td>
<td>- 2 - bod</td>
</tr>
<tr>
<td></td>
<td>- 3 - trigger (not applicable for TimesTen)</td>
</tr>
<tr>
<td></td>
<td>- 4 - index</td>
</tr>
<tr>
<td></td>
<td>- 5 - cluster</td>
</tr>
<tr>
<td></td>
<td>- 8 - LOB (not applicable for TimesTen)</td>
</tr>
<tr>
<td></td>
<td>- 9 - directory</td>
</tr>
<tr>
<td></td>
<td>- 10 - queue</td>
</tr>
<tr>
<td></td>
<td>- 11 - replication object group</td>
</tr>
<tr>
<td></td>
<td>- 12 - replication propagator</td>
</tr>
<tr>
<td></td>
<td>- 13 - Java source (not applicable for TimesTen)</td>
</tr>
<tr>
<td></td>
<td>- 14 - Java resource (not applicable for TimesTen)</td>
</tr>
<tr>
<td></td>
<td>- 58 - model (data mining)</td>
</tr>
<tr>
<td>edition_name</td>
<td>Reserved for future use.</td>
</tr>
</tbody>
</table>

Usage notes

- Executing VALIDATE on a subprogram will also validate subprograms that it references. (See the example below.)
- No errors are raised if the object does not exist or is already valid or is an object that cannot be validated.
- If the object being validated is not actually in the specified edition, the subprogram automatically switches into the edition in which the object is actual prior to
validation. That is, a call to VALIDATE will not actualize the object in the specified edition.

- The INVALIDATE procedure invalidates a database object and optionally changes its PL/SQL compiler parameter settings. The object to be invalidated is specified by its object_id. The subprogram automatically switches to the edition in which the object is actual prior to invalidation. That is, a call to INVALIDATE will not actualize the object in the current edition.

Examples

This example starts where the INVALIDATE example in "INVALIDATE procedure" on page 7-23 left off. Assume P5, Top_Naive, and Top_With_Logging are invalid, shown as follows:

```
Command> select * from user_objects where status='INVALID';
<TOP_NAIVE, <NULL>, 303, <NULL>, PROCEDURE, 2009-09-18 12:53:45, 2009-09-18 12:58:22, INVALID, N, N, 1, <NULL> >
3 rows found.
```

Validating Top_With_Logging, for example, will also validate P5, because it calls P5 (leaving only Top_Naive invalid):

```
Command> begin
> dbms_utility.validate(304);
> end;
> /

PL/SQL procedure successfully completed.
```

```
Command> select * from user_objects where status='INVALID';
1 row found.
```
The TT_DB_VERSION package indicates the TimesTen version numbers.

This chapter contains the following topics

- **Using TT_DB_VERSION**
  - Overview
  - Constants
  - Examples
Using TT_DB_VERSION

- Overview
- Constants
- Examples
Overview

The TT_DB_VERSION package indicates the TimesTen major release (VERSION) and minor release (RELEASE) version numbers. For any TimesTen 11.2.1.x release, the VERSION value is 1121. Depending on the minor release, the RELEASE value would be 0, 1, 2, etc.

The package for Oracle TimesTen In-Memory Database Release 11.2.1.4.0 is as follows:

```sql
PACKAGE TT_DB_VERSION IS
  VERSION CONSTANT PLS_INTEGER := 1121; -- major release number
  RELEASE CONSTANT PLS_INTEGER := 4;  -- minor release number
END TT_DB_VERSION;
```
The TT_DB_VERSION package contains different constant values for different TimesTen releases. The Oracle TimesTen In-Memory Database Release 11.2.1.4.0 version of the TT_DB_VERSION package has the values shown in Table 8–1.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VERSION</td>
<td>BINARY_INTEGER</td>
<td>1121</td>
<td>Current major release of the Oracle TimesTen In-Memory Database.</td>
</tr>
<tr>
<td>RELEASE</td>
<td>BINARY_INTEGER</td>
<td>4</td>
<td>Current minor release of the Oracle TimesTen In-Memory Database.</td>
</tr>
</tbody>
</table>

Note:  Notes on data types:

- The PLS_INTEGER and BINARY_INTEGER data types are identical. This document uses "BINARY_INTEGER" to indicate data types in reference information (such as for table types, record types, subprogram parameters, or subprogram return values), but may use either in discussion and examples.

- The INTEGER and NUMBER(38) data types are also identical. This document uses "INTEGER" throughout.
Examples

See "Examples" on page 10-6 in the UTL_IDENT chapter for an example that uses both packages for conditional compilation.
With the UTL_FILE package, PL/SQL programs can read and write operating system text files. UTL_FILE provides a restricted version of operating system stream file I/O. This chapter contains the following topics:

- **Using UTL_FILE**
  - Security model
  - Rules and limits
  - Exceptions
  - Examples

- **Data structures**

- **Summary of UTL_FILE subprograms**
Using UTL_FILE

- Security model
- Operational notes
- Rules and limits
- Exceptions
- Examples
Security model

In the current TimesTen release, UTL_FILE is restricted to a single directory object called UTL_FILE_TEMP. This directory object points to the $install_dir/plsql/utl_file_temp directory. Access does not extend to subdirectories of this directory. In addition, access is subject to file system permission checking.

You cannot use UTL_FILE with a link, which could be used to circumvent desired access limitations. Specifying a link as the file name will cause FOPEN to fail with an error.

In TimesTen direct mode, the application owner is owner of the file. In Client/Server mode, the server owner is owner of the file.

UTL_FILE_DIR access is not supported in TimesTen.

---

**Important:**

- Users do not have execute permission on UTL_FILE by default. To use UTL_FILE in TimesTen, an ADMIN user or instance administrator must explicitly GRANT EXECUTE permission on it, such as in the following example:

  ```sql
  GRANT EXECUTE ON SYS.UTL_FILE TO scott;
  ```

- The privileges needed to access files in a directory object are operating system specific. UTL_FILE directory object privileges give you read and write access to all files within the specified directory.

- Attempting to apply invalid UTL_FILE options will cause unpredictable results.
Operational notes

The file location and file name parameters are supplied to the FOPEN function as separate strings, so that the file location can be checked against the utl_file_temp directory. Together, the file location and name must represent a legal file name on the system, and the directory must be accessible. Any subdirectories of utl_file_temp are not accessible.

UTL_FILE implicitly interprets line terminators on read requests, thereby affecting the number of bytes returned on a GET_LINE call. For example, the len parameter of UTL_FILE.GET_LINE specifies the requested number of bytes of character data. The number of bytes actually returned to the user will be the least of the following:

- GET_LINE len parameter
- Number of bytes until the next line terminator character
- The max_linesize parameter specified by UTL_FILE.FOPEN

The FOPEN max_linesize parameter must be a number in the range 1 and 32767. If unspecified, TimesTen supplies a default value of 1024. The GET_LINE len parameter must be a number in the range 1 and 32767. If unspecified, TimesTen supplies the default value of max_linesize. If max_linesize and len are defined to be different values, then the lesser value takes precedence.

UTL_FILE.GET_RAW ignores line terminators and returns the actual number of bytes requested by the GET_RAW len parameter.

When data encoded in one character set is read and Globalization Support is told (such as by means of NLS_LANG) that it is encoded in another character set, the result is indeterminate. If NLS_LANG is set, it should be the same as the database character set.
**Rules and limits**

Operating system-specific parameters, such as C-shell environment variables under UNIX, cannot be used in the file location or file name parameters.

UTL_FILE I/O capabilities are similar to standard operating system stream file I/O (OPEN, GET, PUT, CLOSE) capabilities, but with some limitations. For example, you call the FOPEN function to return a file handle, which you use in subsequent calls to GET_LINE or PUT to perform stream I/O to a file. When file I/O is done, you call FCLOSE to complete any output and free resources associated with the file.
Exceptions

<table>
<thead>
<tr>
<th>Exception Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INVALID_PATH</td>
<td>File location is invalid.</td>
</tr>
<tr>
<td>INVALID_MODE</td>
<td>The open_mode parameter in FOPEN is invalid.</td>
</tr>
<tr>
<td>INVALID_FILEHANDLE</td>
<td>File handle is invalid.</td>
</tr>
<tr>
<td>INVALID_OPERATION</td>
<td>File could not be opened or operated on as requested.</td>
</tr>
<tr>
<td>READ_ERROR</td>
<td>Operating system error occurred during the read operation.</td>
</tr>
<tr>
<td>WRITE_ERROR</td>
<td>Operating system error occurred during the write operation.</td>
</tr>
<tr>
<td>INTERNAL_ERROR</td>
<td>Unspecified PL/SQL error.</td>
</tr>
<tr>
<td>CHARSETMISMATCH</td>
<td>A file is opened using FOPEN_NCHAR, but later I/O operations</td>
</tr>
<tr>
<td></td>
<td>use nonchar functions such as PUTF or GET_LINE.</td>
</tr>
<tr>
<td>FILE_OPEN</td>
<td>The requested operation failed because the file is open.</td>
</tr>
<tr>
<td>INVALID_MAXLINESIZE</td>
<td>The max_linesize value for FOPEN is invalid. It should be</td>
</tr>
<tr>
<td></td>
<td>within the range 1 to 32767.</td>
</tr>
<tr>
<td>INVALID_FILENAME</td>
<td>The filename parameter is invalid.</td>
</tr>
<tr>
<td>ACCESS_DENIED</td>
<td>Permission to access to the file location is denied.</td>
</tr>
<tr>
<td>INVALID_OFFSET</td>
<td>Causes of the INVALID_OFFSET exception. One of the following:</td>
</tr>
<tr>
<td></td>
<td>■ ABSOLUTE_OFFSET is NULL and RELATIVE_OFFSET is NULL.</td>
</tr>
<tr>
<td></td>
<td>■ ABSOLUTE_OFFSET is less than 0.</td>
</tr>
<tr>
<td></td>
<td>■ Either offset caused a seek past the end of the file.</td>
</tr>
<tr>
<td>DELETE_FAILED</td>
<td>The requested file delete operation failed.</td>
</tr>
<tr>
<td>RENAME_FAILED</td>
<td>The requested file rename operation failed.</td>
</tr>
</tbody>
</table>

Procedures in UTL_FILE can also raise predefined PL/SQL exceptions such as NO_DATA_FOUND or VALUE_ERROR.
Examples

Example 1
This example reads from a file using the GET_LINE procedure.

```
DECLARE
    V1 VARCHAR2(32767);
    F1 UTL_FILE.FILE_TYPE;
BEGIN
    -- In this example MAX_LINESIZE is less than GET_LINE's length request
    -- so number of bytes returned will be 256 or less if a line terminator is seen.
    F1 := UTL_FILE.FOPEN('utl_file_temp','u12345.tmp','R',256);
    UTL_FILE.GET_LINE(F1,V1,32767);
    DBMS_OUTPUT.PUT_LINE('Get line: ' || V1);
    UTL_FILE.FCLOSE(F1);

    -- In this example, FOPEN's MAX_LINESIZE is NULL and defaults to 1024,
    -- so number of bytes returned will be 1024 or less if line terminator is seen.
    F1 := UTL_FILE.FOPEN('utl_file_temp','u12345.tmp','R');
    UTL_FILE.GET_LINE(F1,V1,32767);
    DBMS_OUTPUT.PUT_LINE('Get line: ' || V1);
    UTL_FILE.FCLOSE(F1);

    -- GET_LINE doesn't specify a number of bytes, so it defaults to
    -- same value as FOPEN's MAX_LINESIZE which is NULL and defaults to 1024.
    -- So number of bytes returned will be 1024 or less if line terminator is seen.
    F1 := UTL_FILE.FOPEN('utl_file_temp','u12345.tmp','R');
    UTL_FILE.GET_LINE(F1,V1);
    DBMS_OUTPUT.PUT_LINE('Get line: ' || V1);
    UTL_FILE.FCLOSE(F1);
END;
```

Consider the following test file, u12345.tmp, in the utl_file_temp directory:

```
This is line 1.
This is line 2.
This is line 3.
This is line 4.
This is line 5.
```

The example results in the following output, repeatedly getting the first line only:

```
Get line: This is line 1.
Get line: This is line 1.
Get line: This is line 1.
```

PL/SQL procedure successfully completed.

Example 2
This example appends content to the end of a file using the PUTF procedure.

```
declare
    handle utl_file.file_type;
    my_world  varchar2(4) := 'Zork';
begin
    handle := utl_file.fopen('utl_file_temp','u12345.tmp','a');
    utl_file.putf(handle, '
Hello, world!
I come from %s with %s.
greetings for all earthlings');
    utl_file.fflush(handle);
    utl_file.fclose(handle);
```

This appends the following to file u12345.tmp in the utl_file_temp directory:

Hello, world!
I come from Zork with greetings for all earthlings.

Example 3
This procedure gets raw data from a specified file using the GET_RAW procedure. It exits when it reaches the end of the data, through its handling of No_Data_Found in the EXCEPTION processing.

```sql
CREATE OR REPLACE PROCEDURE getraw(n IN VARCHAR2) IS
  h       UTL_FILE.FILE_TYPE;
  Buf     RAW(32767);
  Amnt    CONSTANT BINARY_INTEGER := 32767;
BEGIN
  h := UTL_FILE.FOPEN('utl_file_temp', n, 'r', 32767);
  LOOP
    BEGIN
      UTL_FILE.GET_RAW(h, Buf, Amnt);
      -- Do something with this chunk
      DBMS_OUTPUT.PUT_LINE('This is the raw data:');
      DBMS_OUTPUT.PUT_LINE(Buf);
      EXCEPTION WHEN No_Data_Found THEN
        EXIT;
    END;
  END LOOP;
  UTL_FILE.FCLOSE (h);
END;
/
```

Consider the following content in file u12345.tmp in the utl_file_temp directory:

```text
hello world!
```

The example produces output as follows:

```text
Command> begin
    > getraw('u12345.tmp');
    > end;
    > /
This is the raw data:
68656C6C6F20776F726C64210A
PL/SQL procedure successfully completed.
```
Data structures

The UTL_FILE package defines a record type.

Record types

- FILE_TYPE record type
FILE_TYPE record type

The contents of FILE_TYPE are private to the UTL_FILE package. You should not reference or change components of this record.

```plsql
TYPE file_type IS RECORD (
  id        BINARY_INTEGER,
  datatype  BINARY_INTEGER,
  byte_mode BOOLEAN);
```

**Fields**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>A numeric value indicating the internal file handle number.</td>
</tr>
<tr>
<td>datatype</td>
<td>Indicates whether the file is a CHAR file, NCHAR file, or other (binary).</td>
</tr>
<tr>
<td>byte_mode</td>
<td>Indicates whether the file was open as a binary file or as a text file.</td>
</tr>
</tbody>
</table>

---

**Important:** Oracle does not guarantee the persistence of FILE_TYPE values between database sessions or within a single session. Attempts to clone file handles or use dummy file handles may have indeterminate outcomes.

---

**Note:** Notes on data types:

- The PLS_INTEGER and BINARY_INTEGER data types are identical. This document uses "BINARY_INTEGER" to indicate data types in reference information (such as for table types, record types, subprogram parameters, or subprogram return values), but may use either in discussion and examples.

- The INTEGER and NUMBER(38) data types are also identical. This document uses "INTEGER" throughout.
### Summary of UTL_FILE subprograms

<table>
<thead>
<tr>
<th>Subprogram</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCLOSE procedure</td>
<td>Closes a file.</td>
</tr>
<tr>
<td>FCLOSE_ALL procedure</td>
<td>Closes all open file handles.</td>
</tr>
<tr>
<td>FCOPY procedure</td>
<td>Copies a contiguous portion of a file to a newly created file.</td>
</tr>
<tr>
<td>FFLUSH procedure</td>
<td>Physically writes all pending output to a file.</td>
</tr>
<tr>
<td>FGETATTR procedure</td>
<td>Reads and returns the attributes of a disk file.</td>
</tr>
<tr>
<td>FGETPOS function</td>
<td>Returns the current relative offset position (in bytes) within a file, in bytes.</td>
</tr>
<tr>
<td>FOPEN function</td>
<td>Opens a file for input or output.</td>
</tr>
<tr>
<td>FOPEN_NCHAR function</td>
<td>Opens a file in Unicode for input or output.</td>
</tr>
<tr>
<td>FREMOVE procedure</td>
<td>Deletes a disk file, assuming that you have sufficient privileges.</td>
</tr>
<tr>
<td>FRENAMER procedure</td>
<td>Renames an existing file to a new name, similar to the UNIX <code>mv</code> function.</td>
</tr>
<tr>
<td>FSEEK procedure</td>
<td>Adjusts the file pointer forward or backward within the file by the number of bytes specified.</td>
</tr>
<tr>
<td>GET_LINE procedure</td>
<td>Reads text from an open file.</td>
</tr>
<tr>
<td>GET_LINE_NCHAR procedure</td>
<td>Reads text in Unicode from an open file.</td>
</tr>
<tr>
<td>GET_RAW function</td>
<td>Reads a RAW string value from a file and adjusts the file pointer ahead by the number of bytes specified.</td>
</tr>
<tr>
<td>IS_OPEN function</td>
<td>Determines if a file handle refers to an open file.</td>
</tr>
<tr>
<td>NEW_LINE procedure</td>
<td>Writes one or more operating system-specific line terminators to a file.</td>
</tr>
<tr>
<td>PUT procedure</td>
<td>Writes a string to a file.</td>
</tr>
<tr>
<td>PUT_LINE procedure</td>
<td>Writes a line to a file, and so appends an operating system-specific line terminator.</td>
</tr>
<tr>
<td>PUT_LINE_NCHAR procedure</td>
<td>Writes a Unicode line to a file.</td>
</tr>
<tr>
<td>PUT_NCHAR procedure</td>
<td>Writes a Unicode string to a file.</td>
</tr>
<tr>
<td>PUT_RAW procedure</td>
<td>Accepts as input a RAW data value and writes the value to the output buffer.</td>
</tr>
<tr>
<td>PUTF procedure</td>
<td>Equivalent to PUT but with formatting.</td>
</tr>
<tr>
<td>PUTF_NCHAR procedure</td>
<td>Equivalent to PUT_NCHAR but with formatting.</td>
</tr>
</tbody>
</table>
FCLOSE procedure

This procedure closes an open file identified by a file handle.

Syntax

```sql
UTL_FILE.FCLOSE (
    file IN OUT UTL_FILE.FILE_TYPE);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>file</td>
<td>Active file handle returned by an FOPEN or FOPEN_NCHAR call.</td>
</tr>
</tbody>
</table>

Usage notes

If there is buffered data yet to be written when FCLOSE runs, then you may receive a WRITE_ERROR exception when closing a file.

Examples

See "Examples" on page 9-7.

Exceptions

- WRITE_ERROR
- INVALID_FILEHANDLE
FCLOSE_ALL procedure

This procedure closes all open file handles for the session. This should be used as an emergency cleanup procedure, for example, when a PL/SQL program exits on an exception.

Syntax

UTL_FILE.FCLOSE_ALL;

Usage notes

Note: FCLOSE_ALL does not alter the state of the open file handles held by the user. This means that an IS_OPEN test on a file handle after an FCLOSE_ALL call still returns TRUE, even though the file has been closed. No further read or write operations can be performed on a file that was open before an FCLOSE_ALL.

Exceptions

WRITE_ERROR
FCOPY procedure

This procedure copies a contiguous portion of a file to a newly created file. By default, the whole file is copied if the start_line and end_line parameters are omitted. The source file is opened in read mode. The destination file is opened in write mode. A starting and ending line number can optionally be specified to select a portion from the center of the source file for copying.

Syntax

```
UTL_FILE.FCOPY (  
    src_location    IN VARCHAR2,  
    src_filename    IN VARCHAR2,  
    dest_location   IN VARCHAR2,  
    dest_filename   IN VARCHAR2,  
    start_line      IN BINARY_INTEGER DEFAULT 1,  
    end_line        IN BINARY_INTEGER DEFAULT NULL);  
```

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>src_location</td>
<td>Directory location of the source file.</td>
</tr>
<tr>
<td>src_filename</td>
<td>Source file to be copied.</td>
</tr>
<tr>
<td>dest_location</td>
<td>Destination directory where the destination file is created.</td>
</tr>
<tr>
<td>dest_filename</td>
<td>Destination file created from the source file.</td>
</tr>
<tr>
<td>start_line</td>
<td>Line number at which to begin copying. The default is 1 for the first line.</td>
</tr>
<tr>
<td>end_line</td>
<td>Line number at which to stop copying. The default is NULL, signifying end of file.</td>
</tr>
</tbody>
</table>
FFLUSH procedure

FFLUSH physically writes pending data to the file identified by the file handle. Normally, data being written to a file is buffered. The FFLUSH procedure forces the buffered data to be written to the file. The data must be terminated with a newline character.

Flushing is useful when the file must be read while still open. For example, debugging messages can be flushed to the file so that they can be read immediately.

Syntax

```plsql
UTL_FILE.FFLUSH (
    file  IN UTL_FILE.FILE_TYPE);
invalid_maxlinesize  EXCEPTION;
```

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>file</td>
<td>Active file handle returned by an FOPEN or FOPEN_NCHAR call.</td>
</tr>
</tbody>
</table>

Examples

See "Examples" on page 9-7.

Exceptions

- INVALID_FILEHANDLE
- INVALID_OPERATION
- WRITE_ERROR
FGETATTR procedure

This procedure reads and returns the attributes of a disk file.

Syntax

```
UTIL_FILE.FGETATTR(
    location    IN VARCHAR2,
    filename    IN VARCHAR2,
    fexists     OUT BOOLEAN,
    file_length OUT NUMBER,
    blocksize   OUT BINARY_INTEGER);
```

Parameters

```
<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>location</td>
<td>Location of the source file.</td>
</tr>
<tr>
<td>filename</td>
<td>The name of the file to be examined.</td>
</tr>
<tr>
<td>fexists</td>
<td>A BOOLEAN for whether or not the file exists.</td>
</tr>
<tr>
<td>file_length</td>
<td>The length of the file in bytes. NULL if file does not exist.</td>
</tr>
<tr>
<td>blocksize</td>
<td>The file system block size in bytes. NULL if the file does not exist.</td>
</tr>
</tbody>
</table>
```
FGETPOS function

This function returns the current relative offset position within a file, in bytes.

Syntax

```
UTL_FILE.FGETPOS (
    file   IN utl_file.file_type)
RETURN BINARY_INTEGER;
```

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>file</td>
<td>Active file handle returned by an FOPEN or FOPEN_NCHAR call.</td>
</tr>
</tbody>
</table>

Return value

FGETPOS returns the relative offset position for an open file, in bytes. It raises an exception if the file is not open. It returns 0 for the beginning of the file.

Usage notes

If the file was opened for byte mode operations, an INVALID_OPERATION exception is raised.
FOPEN function

This function opens a file. You can specify the maximum line size and have a maximum of 50 files open simultaneously. See also "FOPEN_NCHAR function" on page 9-20.

Syntax

```sql
UTL_FILE.FOPEN (  
    location     IN VARCHAR2,  
    filename     IN VARCHAR2,  
    open_mode    IN VARCHAR2,  
    max_linesize IN BINARY_INTEGER DEFAULT 1024)  
RETURN utl_file.file_type;
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>location</td>
<td>Directory location of file.</td>
</tr>
<tr>
<td>filename</td>
<td>File name, including extension (file type), without directory path. If a</td>
</tr>
<tr>
<td></td>
<td>directory path is given as a part of the filename, it is ignored by FOPEN.</td>
</tr>
<tr>
<td></td>
<td>On UNIX, the filename cannot end with a slash, &quot;.&quot;.</td>
</tr>
<tr>
<td>open_mode</td>
<td>Specifies how the file is opened. Modes include:</td>
</tr>
<tr>
<td></td>
<td>- r - read text</td>
</tr>
<tr>
<td></td>
<td>- w - write text</td>
</tr>
<tr>
<td></td>
<td>- a - append text</td>
</tr>
<tr>
<td></td>
<td>- rb - read byte mode</td>
</tr>
<tr>
<td></td>
<td>- wb - write byte mode</td>
</tr>
<tr>
<td></td>
<td>- ab - append byte mode</td>
</tr>
<tr>
<td></td>
<td>If you try to open a file specifying 'a' or 'ab' for open_mode but the</td>
</tr>
<tr>
<td></td>
<td>file does not exist, the file is created in WRITE mode.</td>
</tr>
<tr>
<td>max_linesize</td>
<td>Maximum number of characters for each line, including the newline character,</td>
</tr>
<tr>
<td></td>
<td>for this file (minimum value 1, maximum value 32767). If unspecified,</td>
</tr>
<tr>
<td></td>
<td>TimesTen supplies a default value of 1024.</td>
</tr>
</tbody>
</table>

Return value

FOPEN returns a file handle, which must be passed to all subsequent procedures that operate on that file. The specific contents of the file handle are private to the UTL_FILE package, and individual components should not be referenced or changed by the UTL_FILE user.

<table>
<thead>
<tr>
<th>Return</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>file_type</td>
<td>Handle to open file.</td>
</tr>
</tbody>
</table>
Usage notes

The file location and file name parameters are supplied to the FOPEN function as separate strings, so that the file location can be checked against the utl_file_temp directory. Together, the file location and name must represent a legal filename on the system, and the directory must be accessible. Any subdirectories of utl_file_temp are not accessible.

Examples

See "Examples" on page 9-7.

Exceptions

INVALID_PATH: File location or name was invalid.
INVALID_MODE: The open_mode string was invalid.
INVALID_OPERATION: File could not be opened as requested.
INVALID_MAXLINESIZE: Specified max_linesize is too large or too small.
FOPEN_NCHAR function

This function opens a file in national character set mode for input or output, with the maximum line size specified. You can have a maximum of 50 files open simultaneously. With this function, you can read or write a text file in Unicode instead of in the database character set.

Even though the contents of an NVARCHAR2 buffer may be AL16UTF16 or UTF8 (depending on the national character set of the database), the contents of the file are always read and written in UTF8. UTL_FILE converts between UTF8 and AL16UTF16 as necessary.

See also “FOPEN function” on page 9-18.

Syntax

```sql
UTL_FILE.FOPEN_NCHAR (  
  location     IN VARCHAR2,  
  filename     IN VARCHAR2,  
  open_mode    IN VARCHAR2,  
  max_linesize IN BINARY_INTEGER DEFAULT 1024)  
RETURN utl_file.file_type;
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>location</td>
<td>Directory location of file.</td>
</tr>
<tr>
<td>filename</td>
<td>File name, including extension.</td>
</tr>
<tr>
<td>open_mode</td>
<td>Open mode (r, w, a, rb, wb, or ab).</td>
</tr>
<tr>
<td>max_linesize</td>
<td>Maximum number of characters for each line, including the newline character, for this file. The minimum value is 1. The maximum is 32767.</td>
</tr>
</tbody>
</table>

Return value

FOPEN_NCHAR returns a file handle, which must be passed to all subsequent procedures that operate on that file. The specific contents of the file handle are private to the UTL_FILE package, and individual components should not be referenced or changed by the UTL_FILE user.

<table>
<thead>
<tr>
<th>Return</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>file_type</td>
<td>Handle to open file.</td>
</tr>
</tbody>
</table>
FREMOVE procedure

This procedure deletes a disk file, assuming that you have sufficient privileges.

Syntax

```
UTL_FILE.FREMOVE (
    location IN VARCHAR2,
    filename IN VARCHAR2);
```

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>location</td>
<td>The directory location of the file.</td>
</tr>
<tr>
<td>filename</td>
<td>The name of the file to be deleted.</td>
</tr>
</tbody>
</table>

Usage notes

The FREMOVE procedure does not verify privileges before deleting a file. The operating system verifies file and directory permissions. An exception is returned on failure.
FRENNAME procedure

This procedure renames an existing file.

Syntax

```
UTL_FILE.FRENAME (    
    src_location  IN VARCHAR2,    
    src_filename  IN VARCHAR2,    
    dest_location IN VARCHAR2,    
    dest_filename IN VARCHAR2,    
    overwrite     IN BOOLEAN DEFAULT FALSE);
```

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>src_location</td>
<td>The directory location of the source file.</td>
</tr>
<tr>
<td>src_filename</td>
<td>The source file to be renamed.</td>
</tr>
<tr>
<td>dest_location</td>
<td>The destination directory of the destination file.</td>
</tr>
<tr>
<td>dest_filename</td>
<td>The new name of the file.</td>
</tr>
<tr>
<td>overwrite</td>
<td>Whether it is permissible to overwrite an existing file in the destination directory. The default is FALSE.</td>
</tr>
</tbody>
</table>

Usage notes

Permission on both the source and destination directories must be granted.
FSEEK procedure

This procedure adjusts the file pointer forward or backward within the file by the number of bytes specified.

Syntax

```sql
UTL_FILE.FSEEK (
    file            IN OUT utl_file.file_type,
    absolute_offset IN     BINARY_INTEGER DEFAULT NULL,
    relative_offset IN     BINARY_INTEGER DEFAULT NULL);
```

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>file</td>
<td>Active file handle returned by an FOPEN or FOPEN_NCHAR call.</td>
</tr>
<tr>
<td>absolute_offset</td>
<td>The absolute location to which to seek, in bytes. Default is NULL.</td>
</tr>
<tr>
<td>relative_offset</td>
<td>The number of bytes to seek forward or backward. Use a positive integer to seek forward, a negative integer to see backward, or 0 for the current position. Default is NULL.</td>
</tr>
</tbody>
</table>

Usage notes

- Using FSEEK, you can read previous lines in the file without first closing and reopening the file. You must know the number of bytes by which you want to navigate.
- If the beginning of the file is reached before the number of bytes specified, then the file pointer is placed at the beginning of the file. If the end of the file is reached before the number of bytes specified, then an INVALID_OFFSET error is raised.
- If the file was opened for byte mode operations, an INVALID_OPERATION exception is raised.
GET_LINE procedure

This procedure reads text from the open file identified by the file handle and places the
text in the output buffer parameter. Text is read up to, but not including, the line
terminator, or up to the end of the file, or up to the end of the len parameter. It cannot
exceed the max_linesize specified in FOPEN.

Syntax

```sql
UTL_FILE.GET_LINE (  
  file        IN  UTL_FILE.FILE_TYPE,  
  buffer      OUT VARCHAR2,  
  len         IN  BINARY_INTEGER DEFAULT NULL);
```

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>file</td>
<td>Active file handle returned by an FOPEN call. If the file is not open for reading (mode r), an INVALID_OPERATION exception is raised.</td>
</tr>
<tr>
<td>buffer</td>
<td>Data buffer to receive the line read from the file.</td>
</tr>
<tr>
<td>len</td>
<td>The number of bytes read from the file. Default is NULL. If NULL, TimesTen supplies the value of max_linesize.</td>
</tr>
</tbody>
</table>

Usage notes

If the line does not fit in the buffer, a VALUE_ERROR exception is raised. If no text
was read due to end of file, the NO_DATA_FOUND exception is raised. If the file was
opened for byte mode operations, the INVALID_OPERATION exception is raised.

Because the line terminator character is not read into the buffer, reading blank lines
returns empty strings.

The maximum size of the buffer parameter is 32767 bytes unless you specify a
smaller size in FOPEN.

If unspecified, TimesTen supplies a default value of 1024. See also "GET_LINE_NCHAR procedure" on page 9-25.

Examples

See "Examples" on page 9-7.

Exceptions

```
INVALID_FILEHANDLE
INVALID_OPERATION
READ_ERROR
NO_DATA_FOUND
VALUE_ERROR
```
GET_LINE_NCHAR procedure

This procedure reads text from the open file identified by the file handle and places the text in the output buffer parameter. With this function, you can read a text file in Unicode instead of in the database character set.

The file must be opened in national character set mode, and must be encoded in the UTF8 character set. The expected buffer data type is NVARCHAR2. If a variable of another data type such as NCHAR or VARCHAR2 is specified, PL/SQL will perform standard implicit conversion from NVARCHAR2 after the text is read.

See also "GET_LINE procedure" on page 9-24.

Syntax

```
UTL_FILE.GET_LINE_NCHAR (
    file    IN  UTL_FILE.FILE_TYPE,
    buffer  OUT NVARCHAR2,
    len     IN  BINARY_INTEGER DEFAULT NULL);
```

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>file</td>
<td>Active file handle returned by an FOPEN_NCHAR call. The file must be open for reading (mode r). If the file was opened by FOPEN instead of FOPEN_NCHAR, a CHARSETMISMATCH exception is raised.</td>
</tr>
<tr>
<td>buffer</td>
<td>Data buffer to receive the line read from the file.</td>
</tr>
<tr>
<td>len</td>
<td>The number of bytes read from the file. Default is NULL. If NULL, TimesTen supplies the value of max_linesize.</td>
</tr>
</tbody>
</table>
GET_RAW function

This function reads a RAW string value from a file and adjusts the file pointer ahead by the number of bytes read. UTL_FILE.GET_RAW ignores line terminators and returns the actual number of bytes requested by the GET_RAW len parameter.

Syntax

```
UTL_FILE.GET_RAW (  
    file    IN  utl_file.file_type,  
    buffer  OUT NOCOPY RAW,  
    len     IN  BINARY_INTEGER DEFAULT NULL);
```

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>file</td>
<td>Active file handle returned by an FOPEN or FOPEN_NCHAR call.</td>
</tr>
<tr>
<td>buffer</td>
<td>The RAW data.</td>
</tr>
<tr>
<td>len</td>
<td>The number of bytes read from the file. Default is NULL. If NULL, len is assumed to be the maximum length of RAW.</td>
</tr>
</tbody>
</table>

Examples

See "Examples" on page 9-7.
IS_OPEN function

This function tests a file handle to see if it identifies an open file. IS_OPEN reports only whether a file handle represents a file that has been opened, but not yet closed. It does not guarantee that there will be no operating system errors when you attempt to use the file handle.

Syntax

UTL_FILE.IS_OPEN (  
    file IN UTL_FILE.FILE_TYPE)  
RETURN BOOLEAN;

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>file</td>
<td>Active file handle returned by an FOPEN or FOPEN_NCHAR call.</td>
</tr>
</tbody>
</table>

Return value

TRUE or FALSE.
NEW_LINE procedure

This procedure writes one or more line terminators to the file identified by the input file handle. This procedure is separate from PUT because the line terminator is a platform-specific character or sequence of characters.

Syntax

```sql
UTL_FILE.NEW_LINE (  
    file    IN UTL_FILE.FILE_TYPE,  
    lines   IN BINARY_INTEGER := 1);
```

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>file</td>
<td>Active file handle returned by an FOPEN or FOPEN_NCHAR call.</td>
</tr>
<tr>
<td>lines</td>
<td>Number of line terminators to be written to the file.</td>
</tr>
</tbody>
</table>

Exceptions

- INVALID_FILEHANDLE
- INVALID_OPERATION
- WRITE_ERROR
PUT procedure

PUT writes the text string stored in the buffer parameter to the open file identified by the file handle. The file must be open for write operations. No line terminator is appended by PUT. Use NEW_LINE to terminate the line or use PUT_LINE to write a complete line with a line terminator. Also see "PUT_NCHAR procedure" on page 9-32.

Syntax

```sql
UTL_FILE.PUT (  
    file IN UTL_FILE.FILE_TYPE,  
    buffer IN VARCHAR2);
```

Parameters

Table 9–21 PUT procedure parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>file</td>
<td>Active file handle returned by an FOPEN_NCHAR call. The file must be open for writing.</td>
</tr>
<tr>
<td>buffer</td>
<td>Buffer that contains the text to be written to the file. If the file was not opened using mode w or a, an INVALID_OPERATION exception is raised.</td>
</tr>
</tbody>
</table>

Usage notes

The maximum size of the buffer parameter is 32767 bytes unless you specify a smaller size in FOPEN. If unspecified, TimesTen supplies a default value of 1024. The sum of all sequential PUT calls cannot exceed 32767 without intermediate buffer flushes.

Exceptions

INVALID_FILEHANDLE
INVALID_OPERATION
WRITE_ERROR
PUT_LINE procedure

This procedure writes the text string stored in the buffer parameter to the open file identified by the file handle. The file must be open for write operations. PUT_LINE terminates the line with the platform-specific line terminator character or characters. Also see "PUT_LINE_NCHAR procedure" on page 9-31.

Syntax

```sql
UTL_FILE.PUT_LINE (  
    file      IN UTL_FILE.FILE_TYPE,  
    buffer    IN VARCHAR2,  
    autoflush IN BOOLEAN DEFAULT FALSE);
```

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>file</td>
<td>Active file handle returned by an FOPEN call.</td>
</tr>
<tr>
<td>buffer</td>
<td>Text buffer that contains the lines to be written to the file.</td>
</tr>
<tr>
<td>autoflush</td>
<td>Flushes the buffer to disk after the write.</td>
</tr>
</tbody>
</table>

Usage notes

- The maximum size of the `buffer` parameter is 32767 bytes unless you specify a smaller size in FOPEN. If unspecified, TimesTen supplies a default value of 1024. The sum of all sequential PUT calls cannot exceed 32767 without intermediate buffer flushes.
- If the file was opened for byte mode operations, an INVALID_OPERATION exception is raised.

Exceptions

- INVALID_FILEHANDLE
- INVALID_OPERATION
- WRITE_ERROR
PUT_LINE_NCHAR procedure

This procedure writes the text string stored in the buffer parameter to the open file identified by the file handle. With this function, you can write a text file in Unicode instead of in the database character set. This procedure is equivalent to the PUT_NCHAR procedure, except that the line separator is appended to the written text. See also “PUT_LINE procedure” on page 9-30.

Syntax

```sql
UTL_FILE.PUT_LINE_NCHAR (
    file    IN UTL_FILE.FILE_TYPE,
    buffer  IN NVARCHAR2);
```

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>file</td>
<td>Active file handle returned by an FOPEN_NCHAR call. The file must be open for writing.</td>
</tr>
<tr>
<td>buffer</td>
<td>Text buffer that contains the lines to be written to the file.</td>
</tr>
</tbody>
</table>

Usage notes

- The maximum size of the buffer parameter is 32767 bytes unless you specify a smaller size in FOPEN. If unspecified, TimesTen supplies a default value of 1024. The sum of all sequential PUT calls cannot exceed 32767 without intermediate buffer flushes.
- If the file was opened for byte mode operations, an INVALID_OPERATION exception is raised.
PUT_NCHAR procedure

This procedure writes the text string stored in the buffer parameter to the open file identified by the file handle.

With this function, you can write a text file in Unicode instead of in the database character set. The file must be opened in the national character set mode. The text string will be written in the UTF8 character set. The expected buffer data type is NVARCHAR2. If a variable of another data type is specified, PL/SQL will perform implicit conversion to NVARCHAR2 before writing the text.

See also "PUT procedure" on page 9-29.

Syntax

UTL_FILE.PUT_NCHAR (  
  file      IN UTL_FILE.FILE_TYPE,  
  buffer    IN NVARCHAR2);  

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>file</td>
<td>Active file handle returned by an FOPEN_NCHAR call. If the file was opened by FOPEN instead of FOPEN_NCHAR, a CHARSETMISMATCH exception is raised.</td>
</tr>
<tr>
<td>buffer</td>
<td>Buffer that contains the text to be written to the file. If the file was not opened using mode w or mode a, an INVALID_OPERATION exception is raised.</td>
</tr>
</tbody>
</table>

Usage notes

The maximum size of the buffer parameter is 32767 bytes unless you specify a smaller size in FOPEN. If unspecified, TimesTen supplies a default value of 1024. The sum of all sequential PUT calls cannot exceed 32767 without intermediate buffer flushes.
PUT_RAW procedure

This function accepts as input a RAW data value and writes the value to the output buffer.

Syntax

```plaintext
UTL_FILE.PUT_RAW (  
    file       IN utl_file.file_type,  
    buffer     IN RAW,  
    autoflush  IN BOOLEAN DEFAULT FALSE);  
```

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>file</td>
<td>Active file handle returned by an FOPEN or FOPEN_NCHAR call.</td>
</tr>
<tr>
<td>buffer</td>
<td>The RAW data written to the buffer.</td>
</tr>
<tr>
<td>autoflush</td>
<td>If TRUE, performs a flush after writing the value to the output buffer. The default is FALSE.</td>
</tr>
</tbody>
</table>

Usage notes

You can request an automatic flush of the buffer by setting the third argument to TRUE.

The maximum size of the `buffer` parameter is 32767 bytes unless you specify a smaller size in FOPEN. If unspecified, TimesTen supplies a default value of 1024. The sum of all sequential PUT calls cannot exceed 32767 without intermediate buffer flushes.
PUTF procedure

This procedure is a formatted PUT procedure. It works like a limited printf(). Also see "PUTF_NCHAR procedure" on page 9-35.

Syntax

```sql
UTL_FILE.PUTF(
    file    IN UTL_FILE.FILE_TYPE,
    format  IN VARCHAR2,
    [arg1   IN VARCHAR2  DEFAULT NULL,
    . . .
    arg5    IN VARCHAR2  DEFAULT NULL]);
```

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>file</td>
<td>Active file handle returned by an FOPEN call.</td>
</tr>
<tr>
<td>format</td>
<td>Format string that can contain text as well as the formatting characters \n and %s.</td>
</tr>
<tr>
<td>arg1..arg5</td>
<td>From one to five operational argument strings. Argument strings are substituted, in order, for the %s formatters in the format string. If there are more formatters in the format parameter string than there are arguments, then an empty string is substituted for each %s for which there is no argument.</td>
</tr>
</tbody>
</table>

Usage notes

- If the file was opened for byte mode operations, an INVALID_OPERATION exception is raised.
- The format string can contain any text, but the character sequences %s and \n have special meaning.

<table>
<thead>
<tr>
<th>Character sequence</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>%s</td>
<td>Substitute this sequence with the string value of the next argument in the argument list.</td>
</tr>
<tr>
<td>\n</td>
<td>Substitute with the appropriate platform-specific line terminator.</td>
</tr>
</tbody>
</table>

Examples

See "Examples" on page 9-7.

Exceptions

- INVALID_FILEHANDLE
- INVALID_OPERATION
- WRITE_ERROR
PUTF_NCHAR procedure

This procedure is a formatted version of a PUT_NCHAR procedure. Using
PUTF_NCHAR, you can write a text file in Unicode instead of in the database
character set. It accepts a format string with formatting elements \n and %s, and up to
five arguments to be substituted for consecutive instances of %s in the format string.
The expected data type of the format string and the arguments is NVARCHAR2.
If variables of another data type are specified, PL/SQL will perform implicit
conversion to NVARCHAR2 before formatting the text. Formatted text is written in the
UTF8 character set to the file identified by the file handle. The file must be opened in
the national character set mode.

Syntax

UTL_FILE.PUTF_NCHAR (
    file    IN UTL_FILE.FILE_TYPE,
    format  IN NVARCHAR2,
    [arg1   IN NVARCHAR2  DEFAULT NULL,
    . . .
    arg5    IN NVARCHAR2  DEFAULT NULL]);

Parameters

Table 9–27  PUTF_NCHAR procedure parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>file</td>
<td>Active file handle returned by an FOPEN_NCHAR call. The file must be open for reading (mode r). If the file was opened by FOPEN instead of FOPEN_NCHAR, a CHARSETMISMATCH exception is raised.</td>
</tr>
<tr>
<td>format</td>
<td>Format string that can contain text as well as the formatting characters \n and %s.</td>
</tr>
<tr>
<td>arg1..arg5</td>
<td>From one to five operational argument strings. Argument strings are substituted, in order, for the %s formatters in the format string. If there are more formatters in the format parameter string than there are arguments, then an empty string is substituted for each %s for which there is no argument.</td>
</tr>
</tbody>
</table>

Usage notes

- The maximum size of the buffer parameter is 32767 bytes unless you specify a smaller size in FOPEN. If unspecified, TimesTen supplies a default value of 1024. The sum of all sequential PUT calls cannot exceed 32767 without intermediate buffer flushes.
- If the file was opened for byte mode operations, an INVALID_OPERATION exception is raised.
The UTL_IDENT package indicates which database or client PL/SQL is running in, such as TimesTen versus Oracle, and server versus client. Each database or client running PL/SQL has its own copy of this package.

This chapter contains the following topics:

- Using UTL_IDENT
  - Overview
  - Security model
  - Constants
  - Examples
Using UTL_IDENT

This section contains topics that relate to using the UTL_IDENT package.

- Overview
- Security model
- Constants
- Examples
Overview

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.

The primary use case for the UTL_IDENT package is for conditional compilation, resembling the following, of PL/SQL packages that are supported by Oracle Database, TimesTen, or clients such as Oracle Forms.

```sql
$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
```
The UTL_IDENT package runs as the package owner SYS. The public synonym UTL_IDENT and EXECUTE permission on this package is granted to PUBLIC.
Constants

The UTL_IDENT package uses the constants shown in Table 10–1, which indicates the settings for TimesTen.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS_ORACLE_SERVER</td>
<td>BOOLEAN</td>
<td>FALSE</td>
<td>Stipulates if Oracle Database or not.</td>
</tr>
<tr>
<td>IS_ORACLE_CLIENT</td>
<td>BOOLEAN</td>
<td>FALSE</td>
<td>Stipulates if Oracle Client or not.</td>
</tr>
<tr>
<td>IS_ORACLE_FORMS</td>
<td>BOOLEAN</td>
<td>FALSE</td>
<td>Stipulates if Oracle Forms or not.</td>
</tr>
<tr>
<td>IS_TIMESTEN</td>
<td>BOOLEAN</td>
<td>TRUE</td>
<td>Stipulates if TimesTen or not.</td>
</tr>
</tbody>
</table>
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 11.2" 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.
```
The UTL_RAW package provides SQL functions for manipulating RAW data types. This chapter contains the following topics:

- Using UTL_RAW
  - Overview
  - Operational notes
- Summary of UTL_RAW subprograms
Using UTL_RAW

- Overview
- Operational notes
Overview

This package is necessary because normal SQL functions do not operate on RAW values, and PL/SQL does not allow overloading between a RAW and a CHAR data type. UTL_RAW also includes subprograms that convert various COBOL number formats to and from RAW formats.

UTL_RAW is not specific to the database environment and may be used in other environments. For this reason, the prefix UTL has been given to the package, instead of DBMS.
UTL_RAW allows a RAW "record" to be composed of many elements. By using the RAW data type, character set conversion will not be performed, keeping the RAW value in its original format when being transferred through remote procedure calls.

With the RAW functions, you can manipulate binary data that was previously limited to the `hextoraw` and `rawtohex` SQL functions.

Functions returning RAW values do so in hexadecimal encoding.
### Summary of UTL_RAW subprograms

**Table 11–1  UTL_RAW Package Subprograms**

<table>
<thead>
<tr>
<th>Subprogram</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT_AND function</td>
<td>Performs bitwise logical &quot;and&quot; of two RAW values and returns the resulting RAW.</td>
</tr>
<tr>
<td>BIT_COMPLEMENT function</td>
<td>Performs bitwise logical &quot;complement&quot; of a RAW value and returns the resulting RAW.</td>
</tr>
<tr>
<td>BIT_OR function</td>
<td>Performs bitwise logical &quot;or&quot; of two RAW values and returns the resulting RAW.</td>
</tr>
<tr>
<td>BIT_XOR function</td>
<td>Performs bitwise logical &quot;exclusive or&quot; of two RAW values and returns the resulting RAW.</td>
</tr>
<tr>
<td>CAST_FROM_BINARY_DOUBLE function</td>
<td>Returns the RAW binary representation of a BINARY_DOUBLE value.</td>
</tr>
<tr>
<td>CAST_FROM_BINARY_FLOAT function</td>
<td>Returns the RAW binary representation of a BINARY_FLOAT value.</td>
</tr>
<tr>
<td>CAST_FROM_BINARY_INTEGER function</td>
<td>Returns the RAW binary representation of a BINARY_INTEGER value.</td>
</tr>
<tr>
<td>CAST_FROM_NUMBER function</td>
<td>Returns the RAW binary representation of a NUMBER value.</td>
</tr>
<tr>
<td>CAST_TO_BINARY_DOUBLE function</td>
<td>Casts the RAW binary representation of a BINARY_DOUBLE value into a BINARY_DOUBLE.</td>
</tr>
<tr>
<td>CAST_TO_BINARY_FLOAT function</td>
<td>Casts the RAW binary representation of a BINARY_FLOAT value into a BINARY_FLOAT.</td>
</tr>
<tr>
<td>CAST_TO_BINARY_INTEGER function</td>
<td>Casts the RAW binary representation of a BINARY_INTEGER value into a BINARY_INTEGER.</td>
</tr>
<tr>
<td>CAST_TO_NUMBER function</td>
<td>Casts the RAW binary representation of a NUMBER value into a NUMBER.</td>
</tr>
<tr>
<td>CAST_TO_NVARCHAR2 function</td>
<td>Converts a RAW value represented using (n) data bytes into an NVARCHAR2 value with (n) data bytes.</td>
</tr>
<tr>
<td>CAST_TO_RAW function</td>
<td>Converts a VARCHAR2 value represented using (n) data bytes into a RAW value with (n) data bytes.</td>
</tr>
<tr>
<td>CAST_TO_VARCHAR2 function</td>
<td>Converts a RAW value represented using (n) data bytes into a VARCHAR2 value with (n) data bytes.</td>
</tr>
<tr>
<td>COMPARE function</td>
<td>Compares two RAW values.</td>
</tr>
<tr>
<td>CONCAT function</td>
<td>Concatenates up to 12 RAW values into a single RAW.</td>
</tr>
<tr>
<td>CONVERT function</td>
<td>Converts a RAW value from one character set to another and returns the resulting RAW.</td>
</tr>
<tr>
<td>COPIES function</td>
<td>Copies a RAW value a specified number of times and returns the concatenated RAW value.</td>
</tr>
<tr>
<td>LENGTH function</td>
<td>Returns the length in bytes of a RAW value.</td>
</tr>
<tr>
<td>OVERLAY function</td>
<td>Overlays the specified portion of a target RAW value with an overlay RAW value, starting from a specified byte position and proceeding for a specified number of bytes.</td>
</tr>
<tr>
<td>REVERSE function</td>
<td>Reverses a byte-sequence in a RAW value.</td>
</tr>
</tbody>
</table>
Table 11–1  (Cont.)  UTL_RAW Package Subprograms

<table>
<thead>
<tr>
<th>Subprogram</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUBSTR function</td>
<td>Returns a substring of a RAW value for a specified number of bytes from a specified starting position.</td>
</tr>
<tr>
<td>TRANSLATE function</td>
<td>Translates the specified bytes from an input RAW value according to the bytes in a specified translation RAW value.</td>
</tr>
<tr>
<td>TRANSLITERATE function</td>
<td>Converts the specified bytes from an input RAW value according to the bytes in a specified transliteration RAW value.</td>
</tr>
<tr>
<td>X RANGE function</td>
<td>Returns a RAW value containing the succession of one-byte encodings beginning and ending with the specified byte-codes.</td>
</tr>
</tbody>
</table>

Note: Notes on data types:

- The PLS_INTEGER and BINARY_INTEGER data types are identical. This document uses "BINARY_INTEGER" to indicate data types in reference information (such as for table types, record types, subprogram parameters, or subprogram return values), but may use either in discussion and examples.

- The INTEGER and NUMBER(38) data types are also identical. This document uses "INTEGER" throughout.
BIT_AND function

This function performs bitwise logical "and" of two supplied RAW values and returns the resulting RAW.

Syntax

```plaintext
UTL_RAW.BIT_AND (
    r1 IN RAW,
    r2 IN RAW)
RETURN RAW;
```

Pragmas

```
pragma restrict_references(bit_and, WNDS, RNDS, WNPS, RNPS);
```

Parameters

Table 11–2 BIT_AND function parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>RAW value to &quot;and&quot; with r2.</td>
</tr>
<tr>
<td>r2</td>
<td>RAW value to &quot;and&quot; with r1.</td>
</tr>
</tbody>
</table>

Return value

Table 11–3 BIT_AND function return value

<table>
<thead>
<tr>
<th>Return</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAW</td>
<td>Contains the &quot;and&quot; result of r1 and r2.</td>
</tr>
<tr>
<td>NULL</td>
<td>If either r1 or r2 is NULL.</td>
</tr>
</tbody>
</table>

Usage notes

If r1 and r2 differ in length, the operation is terminated after the last byte of the shorter of the two RAW values, and the unprocessed portion of the longer RAW value is appended to the partial result. The resulting length equals that of the longer of the two input values.
**BIT_COMPLEMENT function**

This function performs bitwise logical “complement” of the supplied RAW value and returns the resulting RAW. The result length equals the input RAW length.

**Syntax**

```
UTL_RAW.BIT_COMPLEMENT (    
    r IN RAW    
) RETURN RAW;
```

**Pragmas**

```
pragma restrict_references(bit_complement, WNDS, RNDS, WNPS, RNPS);
```

**Parameters**

**Table 11–4 BIT_COMPLEMENT function parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>RAW value for “complement” operation.</td>
</tr>
</tbody>
</table>

**Return value**

**Table 11–5 BIT_COMPLEMENT function return value**

<table>
<thead>
<tr>
<th>Return</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAW</td>
<td>The “complement” of r.</td>
</tr>
<tr>
<td>NULL</td>
<td>If r is NULL.</td>
</tr>
</tbody>
</table>
**BIT_OR function**

This function performs bitwise logical "or" of two supplied RAW values and returns the resulting RAW.

**Syntax**

```sql
UTL_RAW.BIT_OR (  
  r1 IN RAW,  
  r2 IN RAW)  
RETURN RAW;
```

**Pragmas**

```sql
pragma restrict_references(bit_or, WNDS, RNDS, WNPS, RNPS);
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>RAW value to &quot;or&quot; with r2.</td>
</tr>
<tr>
<td>r2</td>
<td>RAW value to &quot;or&quot; with r1.</td>
</tr>
</tbody>
</table>

**Return value**

<table>
<thead>
<tr>
<th>Return</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAW</td>
<td>Contains the &quot;or&quot; result of r1 and r2.</td>
</tr>
<tr>
<td>NULL</td>
<td>If either r1 or r2 is NULL.</td>
</tr>
</tbody>
</table>

**Usage notes**

If r1 and r2 differ in length, the operation is terminated after the last byte of the shorter of the two RAW values, and the unprocessed portion of the longer RAW value is appended to the partial result. The resulting length equals that of the longer of the two input values.
BIT_XOR function

This function performs bitwise logical "exclusive or" of two supplied RAW values and returns the resulting RAW.

Syntax

```
UTL_RAW.BIT_XOR (
  r1 IN RAW,
  r2 IN RAW)
RETURN RAW;
```

Pragmas

```
pragma restrict_references(bit_xor, WNDS, RNDS, WNPS, RNPS);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>RAW value to &quot;xor&quot; with r2.</td>
</tr>
<tr>
<td>r2</td>
<td>RAW value to &quot;xor&quot; with r1.</td>
</tr>
</tbody>
</table>

Return value

<table>
<thead>
<tr>
<th>Return</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAW</td>
<td>Contains the &quot;xor&quot; result of r1 and r2.</td>
</tr>
<tr>
<td>NULL</td>
<td>If either r1 or r2 is NULL.</td>
</tr>
</tbody>
</table>

Usage notes

If r1 and r2 differ in length, the operation is terminated after the last byte of the shorter of the two RAW values, and the unprocessed portion of the longer RAW value is appended to the partial result. The resulting length equals that of the longer of the two input values.
CAST_FROM_BINARY_DOUBLE function

This function returns the RAW binary representation of a BINARY_DOUBLE value.

Syntax

```sql
UTL_RAW.CAST_FROM_BINARY_DOUBLE(
    n          IN BINARY_DOUBLE,
    endianess  IN BINARY_INTEGER DEFAULT 1)
RETURN RAW;
```

Pragmas

```sql
pragma restrict_references(cast_from_binary_double, WNDS, RNDS, WNPS, RNPS);
```

Parameters

**Table 11–10 CAST_FROM_BINARY_DOUBLE function parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>The BINARY_DOUBLE value.</td>
</tr>
<tr>
<td>endianess</td>
<td>A BINARY_INTEGER value indicating the endianess. The function recognizes the defined constants big_endian (1), little_endian (2), and machine_endian (3). The default is big_endian.</td>
</tr>
</tbody>
</table>

Return value

The RAW binary representation of the BINARY_DOUBLE value, or NULL if the input is NULL.

Usage notes

- An eight-byte BINARY_DOUBLE value maps to the IEEE 754 double-precision format as follows:
  
  byte 0: bit 63 ~ bit 56  
  byte 1: bit 55 ~ bit 48  
  byte 2: bit 47 ~ bit 40  
  byte 3: bit 39 ~ bit 32  
  byte 4: bit 31 ~ bit 24  
  byte 5: bit 23 ~ bit 16  
  byte 6: bit 15 ~ bit 8  
  byte 7: bit 7 ~ bit 0  

- The parameter `endianess` specifies how the bytes of the BINARY_DOUBLE value are mapped to the bytes of the RAW value. In the following matrix, `rb0` to `rb7` refer to the bytes of the RAW and `db0` to `db7` refer to the bytes of the BINARY_DOUBLE.

<table>
<thead>
<tr>
<th>rb0</th>
<th>rb1</th>
<th>rb2</th>
<th>rb3</th>
<th>rb4</th>
<th>rb5</th>
<th>rb6</th>
<th>rb7</th>
</tr>
</thead>
<tbody>
<tr>
<td>big_endian</td>
<td>db0</td>
<td>db1</td>
<td>db2</td>
<td>db3</td>
<td>db4</td>
<td>db5</td>
<td>db6</td>
</tr>
<tr>
<td>little_endian</td>
<td>db7</td>
<td>db6</td>
<td>db5</td>
<td>db4</td>
<td>db3</td>
<td>db2</td>
<td>db1</td>
</tr>
</tbody>
</table>

- When `machine_endian` is specified, the eight bytes of the BINARY_DOUBLE argument are copied straight across into the RAW return value. The effect is the
same as if the user specified big_endian on a big-endian system or little_endian on a little-endian system.
CAST_FROM_BINARY_FLOAT function

This function returns the RAW binary representation of a BINARY_FLOAT value.

Syntax

```plaintext
UTL_RAW.CAST_FROM_BINARY_FLOAT(
  n          IN BINARY_FLOAT,
  endianess  IN BINARY_INTEGER DEFAULT 1)
RETURN RAW;
```

Pragmas

```plaintext
pragma restrict_references(cast_from_binary_float, WND5, RNDS, WNPS, RNPS);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>The BINARY_FLOAT value.</td>
</tr>
<tr>
<td>endianess</td>
<td>A BINARY_INTEGER value indicating the endianess. The function recognizes</td>
</tr>
<tr>
<td></td>
<td>the defined constants <code>big_endian</code> (1), <code>little_endian</code> (2), and <code>machine_endian</code> (3). The default is <code>big_endian</code>.</td>
</tr>
</tbody>
</table>

Return value

The RAW binary representation of the BINARY_FLOAT value, or NULL if the input is NULL.

Usage notes

- A four-byte BINARY_FLOAT value maps to the IEEE 754 single-precision format as follows:

  - byte 0: bit 31 ~ bit 24
  - byte 1: bit 23 ~ bit 16
  - byte 2: bit 15 ~ bit 8
  - byte 3: bit 7 ~ bit 0

- The parameter `endianess` specifies how the bytes of the BINARY_FLOAT value are mapped to the bytes of the RAW value. In the following matrix, `rb0` to `rb3` refer to the bytes of the RAW and `fb0` to `fb3` refer to the bytes of the BINARY_FLOAT.

<table>
<thead>
<tr>
<th>endianess</th>
<th>rb0</th>
<th>rb1</th>
<th>rb2</th>
<th>rb3</th>
</tr>
</thead>
<tbody>
<tr>
<td>big_endian</td>
<td>fb0</td>
<td>fb1</td>
<td>fb2</td>
<td>fb3</td>
</tr>
<tr>
<td>little_endian</td>
<td>fb3</td>
<td>fb2</td>
<td>fb1</td>
<td>fb0</td>
</tr>
</tbody>
</table>

- When `machine_endian` is specified, the four bytes of the BINARY_FLOAT argument are copied straight across into the RAW return value. The effect is the same as if the user specified `big_endian` on a big-endian system or `little_endian` on a little-endian system.
CAST_FROM_BINARY_INTEGER function

This function returns the RAW binary representation of a BINARY_INTEGER value.

Syntax

```plsql
UTL_RAW.CAST_FROM_BINARY_INTEGER (  
n          IN BINARY_INTEGER  
endianess  IN BINARY_INTEGER DEFAULT 1)  
RETURN RAW;
```

Pragmas

`pragma restrict_references(cast_from_binary_integer, WNDs, RNDS, WNPS, RNPS);`

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>The BINARY_INTEGER value.</td>
</tr>
<tr>
<td>endianess</td>
<td>A BINARY_INTEGER value indicating the endianess. The function recognizes the defined constants <code>big_endian</code> (1), <code>little_endian</code> (2), and <code>machine_endian</code> (3). The default is <code>big_endian</code>.</td>
</tr>
</tbody>
</table>

Return value

The RAW binary representation of the BINARY_INTEGER value, or NULL if the input is NULL.
CAST_FROM_NUMBER function

This function returns the RAW binary representation of a NUMBER value.

Syntax

```sql
UTL_RAW.CAST_FROM_NUMBER ( 
    n  IN NUMBER) 
RETURN RAW;
```

Pragma

```sql
pragma restrict_references(cast_from_number, WNDS, RNDS, WNPS, RNPS);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>The NUMBER value.</td>
</tr>
</tbody>
</table>

Return value

The RAW binary representation of the NUMBER value, or NULL if the input is NULL.
This function casts the RAW binary representation of a BINARY_DOUBLE value into a BINARY_DOUBLE.

**Syntax**

```plsql
UTL_RAW.CAST_TO_BINARY_DOUBLE (
   r          IN RAW
   endianess  IN BINARY_INTEGER DEFAULT 1)
RETURN BINARY_DOUBLE;
```

**Pragmas**

```plsql
pragma restrict_references(cast_to_binary_double, WNDS, RNDS, WNPS, RNPS);
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>The RAW binary representation of a BINARY_DOUBLE.</td>
</tr>
<tr>
<td>endianess</td>
<td>A BINARY_INTEGER value indicating the endianess. The function recognizes the defined constants <code>big_endian</code> (1), <code>little_endian</code> (2), and <code>machine_endian</code> (3). The default is <code>big_endian</code>.</td>
</tr>
</tbody>
</table>

**Return value**

The BINARY_DOUBLE value, or NULL if the input is NULL.

**Usage notes**

- If the RAW argument is more than eight bytes, only the first eight bytes are used and the rest of the bytes are ignored. If the result is -0, +0 is returned. If the result is NaN, the value BINARY_DOUBLE_NAN is returned.

- An eight-byte BINARY_DOUBLE value maps to the IEEE 754 double-precision format as follows:
  - byte 0: bit 63 ~ bit 56
  - byte 1: bit 55 ~ bit 48
  - byte 2: bit 47 ~ bit 40
  - byte 3: bit 39 ~ bit 32
  - byte 4: bit 31 ~ bit 24
  - byte 5: bit 23 ~ bit 16
  - byte 6: bit 15 ~ bit 8
  - byte 7: bit  7 ~ bit  0

- The parameter `endianess` specifies how the bytes of the BINARY_DOUBLE value are mapped to the bytes of the RAW value. In the following matrix, `rb0` to `rb7` refer to the bytes in RAW and `db0` to `db7` refer to the bytes in BINARY_DOUBLE.

<table>
<thead>
<tr>
<th>rb0</th>
<th>rb1</th>
<th>rb2</th>
<th>rb3</th>
<th>rb4</th>
<th>rb5</th>
<th>rb6</th>
<th>rb7</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>db0</code></td>
<td><code>db1</code></td>
<td><code>db2</code></td>
<td><code>db3</code></td>
<td><code>db4</code></td>
<td><code>db5</code></td>
<td><code>db6</code></td>
<td><code>db7</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rb0</th>
<th>rb1</th>
<th>rb2</th>
<th>rb3</th>
<th>rb4</th>
<th>rb5</th>
<th>rb6</th>
<th>rb7</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>big_endian</code></td>
<td><code>db0</code></td>
<td><code>db1</code></td>
<td><code>db2</code></td>
<td><code>db3</code></td>
<td><code>db4</code></td>
<td><code>db5</code></td>
<td><code>db6</code></td>
</tr>
</tbody>
</table>
When `machine_endian` is specified, the eight bytes of the RAW argument are copied straight across into the BINARY_DOUBLE return value. The effect is the same as if the user specified `big_endian` on a big-endian system or `little_endian` on a little-endian system.

**Exceptions**

If the RAW argument is less than eight bytes, a VALUE_ERROR exception is raised.
CAST_TO_BINARY_FLOAT function

This function casts the RAW binary representation of a BINARY_FLOAT value into a BINARY_FLOAT.

Syntax

```
UTL_RAW.CAST_TO_BINARY_FLOAT (  
  r          IN RAW  
  endianess  IN BINARY_INTEGER DEFAULT 1)  
RETURN BINARY_FLOAT;
```

Pragmas

```
pragma restrict_references(cast_to_binary_float, WDNS, RNDS, WNPS, RNPS);
```

Parameters

```
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>The RAW binary representation of a BINARY_FLOAT.</td>
</tr>
<tr>
<td>endianess</td>
<td>A BINARY_INTEGER value indicating the endianess. The function recognizes the defined constants big_endian (1), little_endian (2), and machine_endian (3). The default is big_endian.</td>
</tr>
</tbody>
</table>
```

Return value

The BINARY_FLOAT value, or NULL if the input is NULL.

Usage notes

- If the RAW argument is more than four bytes, only the first four bytes are used and the rest of the bytes are ignored. If the result is -0, +0 is returned. If the result is NaN, the value BINARY_FLOAT_NAN is returned.
- A four-byte BINARY_FLOAT value maps to the IEEE 754 single-precision format as follows:
  - byte 0: bit 31 ~ bit 24
  - byte 1: bit 23 ~ bit 16
  - byte 2: bit 15 ~ bit  8
  - byte 3: bit  7 ~ bit  0
- The parameter endianess specifies how the bytes of the BINARY_FLOAT value are mapped to the bytes of the RAW value. In the following matrix, rb0 to rb3 refer to the bytes in RAW and fb0 to fb3 refer to the bytes in BINARY_FLOAT.

<table>
<thead>
<tr>
<th></th>
<th>rb0</th>
<th>rb1</th>
<th>rb2</th>
<th>rb3</th>
</tr>
</thead>
<tbody>
<tr>
<td>big_endian</td>
<td>fb0</td>
<td>fb1</td>
<td>fb2</td>
<td>fb3</td>
</tr>
<tr>
<td>little_endian</td>
<td>fb3</td>
<td>fb2</td>
<td>fb1</td>
<td>fb0</td>
</tr>
</tbody>
</table>
- When machine_endian is specified, the four bytes of the RAW argument are copied straight across into the BINARY_FLOAT return value. The effect is the
same as if the user specified `big_endian` on a big-endian system or `little_endian` on a little-endian system.

**Exceptions**

If the RAW argument is less than four bytes, a VALUE_ERROR exception is raised.
CAST_TO_BINARY_INTEGER function

This function casts the RAW binary representation of a BINARY_INTEGER value into a BINARY_INTEGER.

Syntax

```
UTL_RAW.CAST_TO_BINARY_INTEGER (  
    r  IN RAW  
    endianess  IN BINARY_INTEGER DEFAULT 1)  
RETURN BINARY_INTEGER;
```

Pragmas

```
pragma restrict_references(cast_to_binary_integer, WNDS, RNDS, WNPS, RNPS);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>The RAW binary representation of a BINARY_INTEGER.</td>
</tr>
<tr>
<td>endianess</td>
<td>A BINARY_INTEGER value indicating the endianess. The function recognizes the defined constants <code>big_endian</code> (1), <code>little_endian</code> (2), and <code>machine_endian</code> (3). The default is <code>big_endian</code>.</td>
</tr>
</tbody>
</table>

Return value

The BINARY_INTEGER value, or NULL if the input is NULL.
CAST_TO_NUMBER function

This function casts the RAW binary representation of a NUMBER value into a NUMBER.

Syntax

```
UTL_RAW.CAST_TO_NUMBER (r IN RAW)
RETURN NUMBER;
```

Pragmas

```
pragma restrict_references(cast_to_number, WNDS, RNDS, WNPS, RNPS);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>The RAW binary representation of a NUMBER.</td>
</tr>
</tbody>
</table>

Return value

The NUMBER value, or NULL if the input is NULL.
CAST_TO_NVARCHAR2 function

This function converts a RAW value represented using n data bytes into an NVARCHAR2 value with n data bytes.

---

**Note:** When casting to NVARCHAR2, the current Globalization Support character set is used for the characters within that NVARCHAR2 value.

---

**Syntax**

```sql
UTL_RAW.CAST_TO_NVARCHAR2 (r IN RAW) RETURN NVARCHAR2;
```

**Pragmas**

```sql
pragma restrict_references(cast_to_NVARCHAR2, WNDS, RNDS, WNPS, RNPS);
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>RAW value, without leading length field, to be changed to an NVARCHAR2 value.</td>
</tr>
</tbody>
</table>

**Return value**

<table>
<thead>
<tr>
<th>Return</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVARCHAR2</td>
<td>Containing the data converted from the input RAW.</td>
</tr>
<tr>
<td>NULL</td>
<td>If r is NULL.</td>
</tr>
</tbody>
</table>
CAST_TO_RAW function

This function converts a VARCHAR2 value represented using n data bytes into a RAW value with n data bytes. The data is not modified in any way. Only its data type is recast to a RAW data type.

Syntax

```
UTL_RAW.CAST_TO_RAW (  
    c  IN VARCHAR2)  
RETURN RAW;
```

Pragmas

```
pragma restrict_references(cast_to_raw, WNDS, RNDS, WNPS, RNPS);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>VARCHAR2 value to be changed to a RAW value.</td>
</tr>
</tbody>
</table>

Return values

<table>
<thead>
<tr>
<th>Return</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAW</td>
<td>Containing the data converted from the input VARCHAR2, with the same byte-length as the input value but without a leading length field.</td>
</tr>
<tr>
<td>NULL</td>
<td>If c is NULL.</td>
</tr>
</tbody>
</table>
CAST_TO_VARCHAR2 function

This function converts a RAW value represented using n data bytes into a VARCHAR2 value with n data bytes.

Note: When casting to VARCHAR2, the current Globalization Support character set is used for the characters within that VARCHAR2 value.

Syntax

UTIL_RAW.CAST_TO_VARCHAR2 (
    r IN RAW)
RETURN VARCHAR2;

Pragmas

pragma restrict_references(cast_to_VARCHAR2, WND, RND, WNPS, RNPS);

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>RAW value, without leading length field, to be changed to a VARCHAR2 value.</td>
</tr>
</tbody>
</table>

Return value

<table>
<thead>
<tr>
<th>Return</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VARCHAR2</td>
<td>Containing the data converted from the input RAW.</td>
</tr>
<tr>
<td>NULL</td>
<td>If r is NULL.</td>
</tr>
</tbody>
</table>
COMPARE function

This function compares two RAW values. If they differ in length, then the shorter is extended on the right according to the optional pad parameter.

Syntax

```
UTL_RAW.COMPARE ( 
  r1  IN RAW, 
  r2  IN RAW, 
  pad IN RAW DEFAULT NULL) 
RETURN NUMBER;
```

Pragmas

```
pragma restrict_references(compare, WNDS, RNDS, WNPS, RNPS);
```

Parameters

**Table 11–24 COMPARE function parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>First RAW value to be compared. It can be NULL or zero-length.</td>
</tr>
<tr>
<td>r2</td>
<td>Second RAW value to be compared. It can be NULL or zero-length.</td>
</tr>
<tr>
<td>pad</td>
<td>Byte to extend whichever of r1 or r2 is shorter. This is optional. The default is x'00'.</td>
</tr>
</tbody>
</table>

Return value

**Table 11–25 COMPARE function return value**

<table>
<thead>
<tr>
<th>Return</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUMBER</td>
<td>Equals 0 if the RAW byte-strings are identical or both NULL, or equals the position (numbered from 1) of the first mismatched byte.</td>
</tr>
</tbody>
</table>
**CONCAT function**

This function concatenates up to 12 RAW values into a single RAW value. If the concatenated size exceeds 32K, an error is returned.

**Syntax**

```sql
UTL_RAW.CONCAT (
    r1  IN RAW DEFAULT NULL,
    r2  IN RAW DEFAULT NULL,
    r3  IN RAW DEFAULT NULL,
    r4  IN RAW DEFAULT NULL,
    r5  IN RAW DEFAULT NULL,
    r6  IN RAW DEFAULT NULL,
    r7  IN RAW DEFAULT NULL,
    r8  IN RAW DEFAULT NULL,
    r9  IN RAW DEFAULT NULL,
    r10 IN RAW DEFAULT NULL,
    r11 IN RAW DEFAULT NULL,
    r12 IN RAW DEFAULT NULL)
RETURN RAW;
```

**Pragmas**

```sql
pragma restrict_references(concat, WNDS, RNDS, WNPS, RNPS);
```

**Parameters**

Items `r1...r12` are the RAW items to concatenate.

**Return value**

<table>
<thead>
<tr>
<th>Return</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAW</td>
<td>Containing the items concatenated.</td>
</tr>
</tbody>
</table>

**Exceptions**

There is an error if the sum of the lengths of the inputs exceeds the maximum allowable length for a RAW value, which is 32767 bytes.
CONVERT function

This function converts a RAW value from one character set to another and returns the resulting RAW value.

Both character sets must be supported character sets defined to the TimesTen database.

Syntax

```sql
UTL_RAW.CONVERT (r IN RAW, to_charset IN VARCHAR2, from_charset IN VARCHAR2)
RETURN RAW;
```

Pragmas

```sql
pragma restrict_references(convert, WNDS, RNDS, WNPS, RNPS);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>RAW byte-string to be converted.</td>
</tr>
<tr>
<td>to_charset</td>
<td>Name of Globalization Support character set to which <code>r</code> is converted.</td>
</tr>
<tr>
<td>from_charset</td>
<td>Name of Globalization Support character set in which <code>r</code> is supplied.</td>
</tr>
</tbody>
</table>

Return value

<table>
<thead>
<tr>
<th>Return</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAW</td>
<td>Byte-string <code>r</code> converted according to the specified character sets.</td>
</tr>
</tbody>
</table>

Exceptions

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VALUE_ERROR</td>
<td>Occurs under any of the following circumstances:</td>
</tr>
<tr>
<td></td>
<td>■ If <code>r</code> is missing, NULL, or zero-length.</td>
</tr>
<tr>
<td></td>
<td>■ If <code>from_charset</code> or <code>to_charset</code> is missing, NULL, or zero-length.</td>
</tr>
<tr>
<td></td>
<td>■ If <code>from_charset</code> or <code>to_charset</code> is invalid or unsupported.</td>
</tr>
</tbody>
</table>
COPIES function

This function returns a specified number of copies of a specified RAW value, concatenated together.

Syntax

```
UTL_RAW.COPIES (
    r IN RAW,
    n IN NUMBER
) RETURN RAW;
```

Pragmas

```
pragma restrict_references(copies, WDNS, RNDS, WNPS, RNPS);
```

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>RAW value to be copied.</td>
</tr>
<tr>
<td>n</td>
<td>Number of times to copy the RAW value (must be positive).</td>
</tr>
</tbody>
</table>

Return value

This returns the RAW value copied n times and concatenated.

Exceptions

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VALUE_ERROR</td>
<td>Occurs under any of the following circumstances:</td>
</tr>
<tr>
<td></td>
<td>- If r is missing, NULL, or zero-length.</td>
</tr>
<tr>
<td></td>
<td>- If n &lt;= 0.</td>
</tr>
<tr>
<td></td>
<td>- If the length of the result exceeds the maximum allowable length for a RAW value, which is 32767 bytes.</td>
</tr>
</tbody>
</table>
LENGTH function

This function returns the length in bytes of a RAW value.

Syntax

```
UTL_RAW.LENGTH (
    r  IN RAW)
RETURN NUMBER;
```

Pragmas

```
pragma restrict_references(length, WND, RNDS, WNPS, RNPS);
```

Parameters

| Table 11–32  LENGTH function parameters |
|-----------------|----------------------------------------|
| **Parameter**   | **Description**                        |
| r               | The RAW byte-stream to be measured.     |

Return value

| Table 11–33  LENGTH function return value |
|-----------------|----------------------------------------|
| **Return**      | **Description**                        |
| NUMBER          | The length of the RAW value, in bytes.  |
OVERLAY function

This function overlays the specified portion of a target RAW value with an overlay RAW value, starting from a specified byte position and proceeding for a specified number of bytes.

Syntax

```sql
UTL_RAW.OVERLAY (    overlay_str IN RAW,    target IN RAW,    pos IN BINARY_INTEGER DEFAULT 1,    len IN BINARY_INTEGER DEFAULT NULL,    pad IN RAW DEFAULT NULL)    RETURN RAW;
```

Pragmas

```sql
pragma restrict_references(overlay, WNSD, RNDS, WNPS, RNPS);
```

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>overlay_str</td>
<td>Byte-string used to overlay target.</td>
</tr>
<tr>
<td>target</td>
<td>Target byte-string to be overlaid.</td>
</tr>
<tr>
<td>pos</td>
<td>Byte position in target at which to start overlay (numbered from 1). This is optional. The default is 1.</td>
</tr>
<tr>
<td>len</td>
<td>The number of bytes to overlay. This is optional. The default is the length of overlay_str.</td>
</tr>
<tr>
<td>pad</td>
<td>Pad byte used when len exceeds overlay_str length or pos exceeds target length. This is optional. The default is x'00'.</td>
</tr>
</tbody>
</table>

Return value

<table>
<thead>
<tr>
<th>Return</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAW</td>
<td>The target byte value overlaid as specified.</td>
</tr>
</tbody>
</table>

Usage notes

If `overlay_str` has less than `len` bytes, then it is extended to `len` bytes using the pad byte. If `overlay_str` exceeds `len` bytes, then the extra bytes in `overlay_str` are ignored. If `len` bytes beginning at position `pos` of `target` exceeds the length of `target`, then `target` is extended to contain the entire length of `overlay_str`.

If `len` is specified, it must be greater than or equal to 0. If `pos` is specified, it must be greater than or equal to 1. If `pos` exceeds the length of `target`, then `target` is padded with `pad` bytes to position `pos`, and `target` is further extended with `overlay_str` bytes.
## Exceptions

### Table 11–36  OVERLAY function exceptions

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VALUE_ERROR</td>
<td>Occurs under any of the following circumstances:</td>
</tr>
<tr>
<td></td>
<td>- If <code>overlay_str</code> is NULL or zero-length.</td>
</tr>
<tr>
<td></td>
<td>- If <code>target</code> is missing or undefined.</td>
</tr>
<tr>
<td></td>
<td>- If the length of <code>target</code> exceeds the maximum length for a RAW value,</td>
</tr>
<tr>
<td></td>
<td>32767 bytes.</td>
</tr>
<tr>
<td></td>
<td>- If <code>len &lt; 0</code>.</td>
</tr>
<tr>
<td></td>
<td>- If <code>pos &lt;= 0</code>.</td>
</tr>
</tbody>
</table>
REVERSE function

This function reverses a RAW byte-sequence from end to end. For example, x'0102F3' would be reversed to x'F30201', and 'xyz' would be reversed to 'zyx'. The result length is the same as the input length.

Syntax

```sql
UTL_RAW.REVERSE (r IN RAW)
RETURN RAW;
```

Pragmas

```sql
pragma restrict_references(reverse, WNDS, RNDS, WNPS, RNPS);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>RAW value to reverse.</td>
</tr>
</tbody>
</table>

Return value

<table>
<thead>
<tr>
<th>Return</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAW</td>
<td>Containing the reverse of r.</td>
</tr>
</tbody>
</table>

Exceptions

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VALUE_ERROR</td>
<td>Occurs if r is NULL or zero-length.</td>
</tr>
</tbody>
</table>
SUBSTR function

This function returns a substring of a RAW value for a specified number of bytes and starting position.

Syntax

```sql
UTL_RAW.SUBSTR (    r   IN RAW,
    pos IN BINARY_INTEGER,
    len IN BINARY_INTEGER DEFAULT NULL)    RETURN RAW;
```

Pragmas

```sql
pragma restrict_references(substr, WDNS, RNDS, WNPS, RNPS);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>The RAW byte-string from which the substring is extracted.</td>
</tr>
<tr>
<td>pos</td>
<td>The byte position in <code>r</code> at which to begin extraction, either counting forward from the beginning of <code>r</code> (positive value) or backward from the end of <code>r</code> (negative value).</td>
</tr>
<tr>
<td>len</td>
<td>The number of bytes, beginning at <code>pos</code> and proceeding toward the end of <code>r</code>, to extract from <code>r</code>. This is optional. The default is to the end of <code>r</code>.</td>
</tr>
</tbody>
</table>

Return value

<table>
<thead>
<tr>
<th>Return</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAW</td>
<td>The RAW substring beginning at <code>pos</code> for <code>len</code> bytes.</td>
</tr>
<tr>
<td>NULL</td>
<td>If the input parameter was NULL.</td>
</tr>
</tbody>
</table>

Usage notes

If `pos` is positive, SUBSTR counts from the beginning of `r` to find the first byte. If `pos` is negative, SUBSTR counts backward from the end of `r`. The value of `pos` cannot equal 0.

A specified value of `len` must be positive. If `len` is omitted, SUBSTR returns all bytes to the end of `r`. 
Exceptions

### Example 1
This example counts backward 15 bytes from the end of the input RAW for its starting position, then takes a substring of five bytes starting at that point.

```plsql
declare
    sr raw(32767);
    r raw(32767);
begin
    sr       := hextoraw('12365678121256123444341234567890ABAA1234');
    r := UTL_RAW.SUBSTR(sr, -15, 5);
    dbms_output.put_line('source raw: ' || sr);
    dbms_output.put_line('return raw: ' || r);
end;
/```

The result is as follows:

```
source raw: 12365678121256123444341234567890ABAA1234
return raw: 5612344434
```

The substring starts at the 15th byte from the end.

### Example 2
This example has the same input RAW and starting point as the preceding example, but because `len` is not specified the substring is taken from the starting point to the end of the input.

```plsql
declare
    sr raw(32767);
    r raw(32767);
begin
    sr       := hextoraw('12365678121256123444341234567890ABAA1234');
    r := UTL_RAW.SUBSTR(sr, -15);
    dbms_output.put_line('source raw: ' || sr);
    dbms_output.put_line('return raw: ' || r);
end;
/```

Here the input and output are presented, for purposes of this discussion, in a way that gives a clearer indication of the functionality:

```
source raw: 12 36 56 78 12 12 56 12 34 44 34 34 12 34 56 78 90 AB AA 12 34
return raw: 56 12 34 44 34
```

The substring starts at the 15th byte from the end.

### Table 11–42 SUBSTR function exceptions

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VALUE_ERROR</td>
<td>Occurs under any of the following circumstances:</td>
</tr>
<tr>
<td></td>
<td>• If <code>pos</code> equals 0 or is greater than the length of <code>r</code>.</td>
</tr>
<tr>
<td></td>
<td>• If <code>len</code> is less than or equal to 0.</td>
</tr>
<tr>
<td></td>
<td>• If <code>len</code> is greater than (length of <code>r</code>) minus (pos-1).</td>
</tr>
</tbody>
</table>
Here is the result:

source raw: 12365678121256123444341234567890ABAA1234
return raw: 56123444341234567890ABAA1234

Here the input and output are presented, for purposes of this discussion, in a way that gives a clearer indication of the functionality:

source raw: 12 36 56 78 12 12 56 12 34 44 34 34 12 34 56 78 90 AB AA 12 34
return raw: 56 12 34 44 34 34 12 34 56 78 90 AB AA 12 34
TRANSLATE function

This function performs a byte-by-byte translation of a RAW value, given an input set of bytes, a set of bytes to search for in the input bytes and translate from, and a set of corresponding bytes to translate to. Whenever a byte in the specified \textit{from\_set} is found in the input RAW value, it is translated to the corresponding byte in the \textit{to\_set} for the output RAW value, or it is simply not included in the output RAW value if there is no corresponding byte in \textit{to\_set}. Any bytes in the input RAW value that do not appear in \textit{from\_set} are simply copied as-is to the output RAW value.

Syntax

\begin{verbatim}
UTIL_RAW.TRANSLATE ( 
  r        IN RAW,  
  from_set IN RAW,  
  to_set   IN RAW)  
RETURN RAW;
\end{verbatim}

\textbf{Note: } Be aware that \textit{to\_set} and \textit{from\_set} are reversed in the calling sequence compared to TRANSLITERATE.

Pragmas

\begin{verbatim}
pragma restrict_references(translate, WNDS, RNDS, WNPS, RNPS);
\end{verbatim}

Parameters

\begin{table}[h]
\centering
\caption{TRANSLATE function parameters}
\begin{tabular}{|l|p{1.5in}|}
\hline
Parameter & Description \\
\hline
\textbf{r} & RAW source byte-string whose bytes are to be translated, as applicable. \\
\textbf{from\_set} & RAW byte-codes that are searched for in \textit{r}. Where found, they are translated in the result. \\
\textbf{to\_set} & RAW byte-codes to translate to. Where a \textit{from\_set} byte is found in \textit{r}, it is translated in the result to the corresponding \textit{to\_set} byte, as applicable. \\
\hline
\end{tabular}
\end{table}

Return value

\begin{table}[h]
\centering
\caption{TRANSLATE function return value}
\begin{tabular}{|l|p{1.5in}|}
\hline
Return & Description \\
\hline
\textbf{RAW} & Translated byte-string. \\
\hline
\end{tabular}
\end{table}

Usage notes

\begin{itemize}
\item If \textit{to\_set} is shorter than \textit{from\_set}, the extra \textit{from\_set} bytes have no corresponding translation bytes. Bytes from the input RAW that match any such \textit{from\_set} bytes are not translated or included in the result. They are effectively translated to NULL.
\item If \textit{to\_set} is longer than \textit{from\_set}, the extra \textit{to\_set} bytes are ignored.
\item If a byte value is repeated in \textit{from\_set}, the repeated occurrence is ignored.
\end{itemize}
Note: Differences from TRANSLITERATE:

- The from_set parameter comes before the to_set parameter in the calling sequence.
- Bytes from r that appear in from_set but have no corresponding values in to_set are not translated or included in the result.
- The resulting RAW value may be shorter than the input RAW value.

Note that TRANSLATE and TRANSLITERATE only differ in functionality when to_set has fewer bytes than from_set.

Exceptions

Table 11–45  TRANSLATE function exceptions

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
</table>
| VALUE_ERROR | Occurs if r, from_set, or to_set is NULL or zero-length.

Examples

Example 1

In this example, from_set is x'12AA34' and to_set is x'CD'. Wherever '12' appears in the input RAW it will be replaced by 'CD' in the result. Wherever 'AA' or '34' appears in the input RAW, because there are no corresponding bytes in to_set those bytes will simply not be included in the result (effectively translated to NULL).

You can compare this to "Examples" on page 11-40 in the TRANSLITERATE section to see how the functions differ.

```plsql
declare
    sr raw(32767);
    from_set raw(32767);
    to_set raw(32767);
    r raw(32767);
begin
    sr := hextoraw('12365678121256123444341234567890ABAA1234');
    from_set := hextoraw('12AA34');
    to_set := hextoraw('CD');
    dbms_output.put_line('from_set: ' || from_set);
    dbms_output.put_line('to_set:     ' || to_set);
    r := UTL_RAW.TRANSLATE(sr, from_set, to_set);
    dbms_output.put_line('source raw: ' || sr);
    dbms_output.put_line('return raw: ' || r);
end;
/
```

The result is as follows:

```
from_set:  12AA34
to_set:    CD
source raw: 12365678121256123444341234567890ABAA1234
return raw: CD365678CD56CD44CD567890ABCD
```

PL/SQL procedure successfully completed.
Here the inputs and output are presented, for purposes of this discussion, in a way that gives a clearer indication of the functionality:

| from_set:   | 12  AA 34 |
| to_set:     | CD        |
| source raw: | 12 365678 12 12 56 12 34 44 34 12 12 34 567890AB AA 12 34 |
| return raw: | CD 365678 CD CD 56 CD 44 CD 567890AB CD |

### Example 2

In this example, the from_set is `x'12AA12'` and the to_set is `x'CDABEF'`. Wherever `12` appears in the input RAW it will be replaced by `CD` in the result. Wherever `AA` appears in the input it will be replaced by `AB` in the result. The second `12` in from_set is ignored, and therefore the corresponding byte in to_set is ignored as well.

```plsql
declare
  sr raw(32767);
  from_set raw(32767);
  to_set raw(32767);
  r raw(32767);
begin
  sr := hextoraw('12365678121256123444341234567890ABAA1234');
  from_set := hextoraw('12AA12');
  to_set := hextoraw('CDABEF');
  dbms_output.put_line('from_set: ' || from_set);
  dbms_output.put_line('to_set: ' || to_set);
  r := UTL_RAW.TRANSLATE(sr, from_set, to_set);
  dbms_output.put_line('source raw: ' || sr);
  dbms_output.put_line('return raw: ' || r);
end;
/
```

The result is as follows. Note this is the same behavior as for TRANSLITERATE with the same input RAW, from_set, and to_set, as shown in "Examples" on page 11-40.

| from_set:   | 12AA12 |
| to_set:     | CDABEF |
| source raw: | 12365678121256123444341234567890ABAA1234 |
| return raw: | CD365678CDCD56CD34443434CD34567890AB ABCD34 |

PL/SQL procedure successfully completed.
TRANSLITERATE function

This function performs a byte-by-byte transliteration of a RAW value, given an input set of bytes, a set of bytes to search for in the input bytes and convert from, and a set of corresponding bytes to convert to. Whenever a byte in the specified from_set is found in the input RAW value, it is converted to the corresponding byte in the to_set for the output RAW value, or it is converted to the specified “padding” byte if there is no corresponding byte in to_set. Any bytes in the input RAW value that do not appear in from_set are simply copied as-is to the output RAW value.

Syntax

```sql
UTL_RAW.TRANSLITERATE ( 
    r        IN RAW,
    to_set   IN RAW DEFAULT NULL,
    from_set IN RAW DEFAULT NULL,
    pad      IN RAW DEFAULT NULL) 
RETURN RAW;
```

Note: Be aware that to_set and from_set are reversed in the calling sequence compared to TRANSLATE.

Pragmas

```sql
pragma restrict_references(transliterate, WNDS, RNDS, WNPS, RNPS);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>RAW source byte-string whose bytes are to be converted, as applicable.</td>
</tr>
<tr>
<td>to_set</td>
<td>RAW byte-codes to convert to. Where a from_set byte is found in r, it is converted in the result to the corresponding to_set byte, as applicable. This defaults to a NULL string effectively extended with pad to the length of from_set, as necessary.</td>
</tr>
<tr>
<td>from_set</td>
<td>RAW byte-codes that are searched for in r. Where found, they are converted in the result. The default is x’00’ through x’FF’, which results in all bytes in r being converted in the result.</td>
</tr>
<tr>
<td>pad</td>
<td>A “padding” byte that is used as the conversion value for any byte in r for which there is a matching byte in from_set that does not have a corresponding byte in to_set. The default is x’00’.</td>
</tr>
</tbody>
</table>

Return value

<table>
<thead>
<tr>
<th>Return</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAW</td>
<td>Converted byte-string.</td>
</tr>
</tbody>
</table>
**Usage notes**

- If `to_set` is shorter than `from_set`, the extra `from_set` bytes have no corresponding conversion bytes. Bytes from the input RAW that match any such `from_set` bytes are converted in the result to the pad byte instead.
- If `to_set` is longer than `from_set`, the extra `to_set` bytes are ignored.
- If a byte value is repeated in `from_set`, the repeated occurrence is ignored.

---

**Note:** Differences from TRANSLATE:

- The `to_set` parameter comes before the `from_set` parameter in the calling sequence.
- Bytes from `r` that appear in `from_set` but have no corresponding values in `to_set` are replaced by pad in the result.
- The resulting RAW value always has the same length as the input RAW value.

Note that TRANSLATE and TRANSLITERATE only differ in functionality when `to_set` has fewer bytes than `from_set`.

---

**Exceptions**

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VALUE_ERROR</td>
<td>Occurs if <code>r</code> is NULL or zero-length.</td>
</tr>
</tbody>
</table>

**Examples**

**Example 1**

In this example, the `from_set` is `x'12AA34'` and the `to_set` is `x'CD'`. Wherever '12' appears in the input RAW it will be replaced by 'CD' in the result. Wherever 'AA' or '34' appears in the input RAW, because there are no corresponding bytes in `to_set` those bytes will be replaced by the pad byte, which is not specified and therefore defaults to `x'00'`.

You can compare this to "Examples" on page 11-37 in the TRANSLATE section to see how the functions differ.

```sql
declare
    sr          RAW(32767);
    from_set    RAW(32767);
    to_set      RAW(32767);
    r           RAW(32767);
begin
    sr := hextoraw('12365678121256123444341234567890ABAA1234');
    from_set := hextoraw('12AA34');
    to_set := hextoraw('CD');
    dbms_output.put_line('from_set: ' || from_set);
    dbms_output.put_line('to_set: ' || to_set);
    r := UTL_RAW.TRANSLITERATE(sr, to_set, from_set);
    dbms_output.put_line('source raw: ' || sr);
    dbms_output.put_line('return raw: ' || r);
end;
```
The result is as follows:

```
from_set:   12AA34
to_set:     CD
source raw: 1236567812125612344434341234567890ABAA1234
return raw: CD365678CDD56CD00440000CD00567890AB00CD00
```

PL/SQL procedure successfully completed.

Here the inputs and output are presented, for purposes of this discussion, in a way that gives a clearer indication of the functionality:

```
from_set:   12 AA 34
to_set:     CD
source raw: 12 365678 12 12 56 12 34 44 34 34 12 34 567890AB AA 12 34
return raw: CD 365678 CD 56 CD 00 44 00 00 CD 00 567890AB 00 CD 00
```

Example 2
This is the same as the preceding example, except pad is specified to be \texttt{x'FF'}.

```
declare
    sr raw(32767);
    from_set raw(32767);
    to_set raw(32767);
    pad raw(32767);
    r raw(32767);
begin
    sr       := hextoraw('1236567812125612344434341234567890ABAA1234');
    from_set := hextoraw('12AA34');
    to_set   := hextoraw('CD');
    pad      := hextoraw('FF');
    dbms_output.put_line('from_set:   ' || from_set);
    dbms_output.put_line('to_set:     ' || to_set);
    r := UTL_RAW.TRANSLITERATE(sr, to_set, from_set, pad);
    dbms_output.put_line('source raw: ' || sr);
    dbms_output.put_line('return raw: ' || r);
end;
/
```

The result is as follows. 'AA' and '34' are replaced by 'FF' instead of '00'.

```
from_set:   12AA34
to_set:     CD
source raw: 1236567812125612344434341234567890ABAA1234
return raw: CD365678CDD56CDFF44FFFFFCDDF567890ABFPCDDF
```

PL/SQL procedure successfully completed.

Example 3
In this example, the from_set is \texttt{x'12AA12'} and the to_set is \texttt{x'CDABEF'}. Wherever '12' appears in the input RAW it will be replaced by 'CD' in the result. Wherever 'AA' appears in the input it will be replaced by 'AB' in the result. The second '12' in from_set is ignored, and therefore the corresponding byte in to_set is ignored as well.

```
declare
    sr raw(32767);
    from_set raw(32767);
    to_set raw(32767);
begin
    sr       := hextoraw('1236567812125612344434341234567890ABAA1234');
    from_set := hextoraw('12AA12');
    to_set   := hextoraw('CDABEF');
    r := UTL_RAW.TRANSLITERATE(sr, to_set, from_set, pad);
    dbms_output.put_line('from_set:   ' || from_set);
    dbms_output.put_line('to_set:     ' || to_set);
    r := UTL_RAW.TRANSLITERATE(sr, to_set, from_set, pad);
    dbms_output.put_line('from_set:   ' || from_set);
    dbms_output.put_line('to_set:     ' || to_set);
    r := UTL_RAW.TRANSLITERATE(sr, to_set, from_set, pad);
end;
/
```

The result is as follows:

```
from_set:   12AA12
to_set:     CDABEF
source raw: 1236567812125612344434341234567890ABAA1234
return raw: CD365678CDD56CDABEF44FFFFFCDDF567890ABFPCDDF
```

PL/SQL procedure successfully completed.
r raw(32767);
begin
sr := hextoraw('1236567812125612344434341234567890ABAA1234');
from_set := hextoraw('12AA12');
to_set := hextoraw('CDABEF');
dbms_output.put_line('from_set: ' || from_set);
dbms_output.put_line('to_set: ' || to_set);
r := UTL_RAW.TRANSLITERATE(sr, to_set, from_set);
dbms_output.put_line('source raw: ' || sr);
dbms_output.put_line('return raw: ' || r);
end;
/

The result is as follows. Note this is the same behavior as for TRANSLATE with the same input RAW, from_set, and to_set, as shown in "Examples" on page 11-37.

from_set: 12AA12
to_set: CDABEF
source raw: 1236567812125612344434341234567890ABAA1234
return raw: CD365678CDCD56CD34443434CD34567890ABABCD34

PL/SQL procedure successfully completed.

Example 4
In this example, from_set and to_set are not specified.

declare
sr raw(32767);
r raw(32767);
begin
sr := hextoraw('1236567812125612344434341234567890ABAA1234');
r := UTL_RAW.TRANSLITERATE(sr);
dbms_output.put_line('source raw: ' || sr);
dbms_output.put_line('return raw: ' || r);
end;
/

The result is as follows. According to the from_set and to_set defaults, all bytes are replaced by x'00'.

source raw: 1236567812125612344434341234567890ABAA1234
return raw: 000000000000000000000000000000000000000000

PL/SQL procedure successfully completed.
XRANGE function

This function returns a RAW value containing the succession of one-byte encodings beginning and ending with the specified byte-codes. The specified byte-codes must be single-byte RAW values. If the start_byte value is greater than the end_byte value, then the succession of resulting bytes begins with start_byte, wraps through x'FF' back to x'00', then ends at end_byte.

Syntax

UTL_RAW.XRANGE {
    start_byte IN RAW DEFAULT NULL,
    end_byte   IN RAW DEFAULT NULL)
RETURN RAW;

Pragmas

pragma restrict_references(xrange, WNDS, RNDS, WNPS, RNPS);

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>start_byte</td>
<td>Beginning byte-code value for resulting sequence. The default is x'00'.</td>
</tr>
<tr>
<td>end_byte</td>
<td>Ending byte-code value for resulting sequence. The default is x'FF'.</td>
</tr>
</tbody>
</table>

Return value

<table>
<thead>
<tr>
<th>Return</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAW</td>
<td>Containing succession of one-byte encodings.</td>
</tr>
</tbody>
</table>

Examples

The following three examples show the results where start_byte is less than end_byte, start_byte is greater than end_byte, and default values are used.

Command> declare
>   r raw(32767);
>   s raw(32767);
>   e raw(32767);
> begin
>   s := hextoraw('1');
>   e := hextoraw('A');
>   r := utl_raw.xrangexrange(s,e);
>   dbms_output.put_line(r);
> end;
> /
0102030405060708090A
PL/SQL procedure successfully completed.
>    r raw(32767);
>    s raw(32767);
>    e raw(32767);
>    begin
>    s := hextoraw('EE');
>    e := hextoraw('A');
>    r := utl_raw.xrange(s,e);
>    dbms_output.put_line(r);
>    end;
>    /
EEEFF0F1F2F3F4F5F6F7F8F9FAFBFCFDFE0F000102030405060708090A

PL/SQL procedure successfully completed.

Command> declare
>    r raw(32767);
>    begin
>    r := utl_raw.xrange();
>    dbms_output.put_line(r);
>    end;
>    /
000102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F2021222324252627
28292A2B2C2D2E2F303132333435363738393A3B3C3D3E3F404142434445464748494A4B4C4D4E4F
505152535455565758595A5B5C5D5E5F606162636465666768696A6B6C6D6E6F7071727374757677
78797A7B7C7D7E7F808182838485868788898A8B8C8D8E8F90919293949596979899A9B9C9D9E9F
A0A1A2A3A4A5A6A7A8A9AAABACADAEAFB0B1B2B3B4B5B6B7B8B9BABBBCBDBEDEFC0C1C2C3C4C5C6C7
C8C9CACCDCDCECDFD0D1D2D3D4D5D6D7D8D9DADEDDDEDFE0E1E2E3E4E5E6E7E8E9EABECCDEEEEE
F0F1F2F3F4F5F6F7F8F9FAFBFCFDFE0F

PL/SQL procedure successfully completed.
The UTL_RECOMP package recompiles invalid PL/SQL modules, invalid views, index types, and operators in a database.

This chapter contains the following topics:

- **Using UTL_RECOMP**
  - Overview
  - Operational notes
  - Examples
- **Summary of UTL_RECOMP subprograms**
Using UTL_RECOMP

- Overview
- Operational notes
- Examples
Overview

UTL_RECOMP is particularly useful after a major-version upgrade that typically invalidates all PL/SQL objects. Although invalid objects are recompiled automatically on use, it is useful to run this prior to operation because this will either eliminate or minimize subsequent latencies due to on-demand automatic recompilation at runtime.
Operational notes

- This package must be run using ttIsql.
- To use this package, you must be the instance administrator and run it as SYS.UTL_RECOMP.
- This package expects the following packages to have been created with VALID status:
  - STANDARD (standard.sql)
  - DBMS_STANDARD (dbmsstdx.sql)
  - DBMS_RANDOM (dbmsrand.sql)
- There should be no other DDL on the database while running entries in this package. Not following this recommendation may lead to deadlocks.
- 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.
Examples

Recompile all objects sequentially:

Command> EXECUTE SYS.UTL_RECOMP.RECOMP_SERIAL();

Recompile objects in schema SCOTT sequentially:

Command> EXECUTE SYS.UTL_RECOMP.RECOMP_SERIAL('SCOTT');
Summary of UTL_RECOMP subprograms

Table 12–1  UTL_RECOMP Package Subprograms

<table>
<thead>
<tr>
<th>Subprogram</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RECOMP_PARALLEL</td>
<td>Recompiles invalid objects in a given schema, or all invalid objects in the database, in parallel. As noted earlier, in TimesTen 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.</td>
</tr>
<tr>
<td>RECOMP_SERIAL</td>
<td>Recompiles invalid objects in a given schema or all invalid objects in the database.</td>
</tr>
</tbody>
</table>

Note: Notes on data types:

- The PLS_INTEGER and BINARY_INTEGER data types are identical. This document uses "BINARY_INTEGER" to indicate data types in reference information (such as for table types, record types, subprogram parameters, or subprogram return values), but may use either in discussion and examples.

- The INTEGER and NUMBER(38) data types are also identical. This document uses "INTEGER" throughout.
RECOMP_PARALLEL procedure

This procedure uses the information exposed in the DBA_Dependencies view to recompile invalid objects in the database, or in a given schema, in parallel.

In TimesTen, the threads value is always 1 regardless of how it is set. As a result, there is no effective difference between RECOMP_PARALLEL and RECOMP_SERIAL.

Syntax

```sql
UTL_RECOMP.RECOMP_PARALLEL(
    threads  IN   BINARY_INTEGER DEFAULT NULL,
    schema   IN   VARCHAR2    DEFAULT NULL,
    flags    IN   BINARY_INTEGER DEFAULT 0);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>threads</td>
<td>The number of recompile threads to run in parallel. If NULL, use the value of 'job_queue_processes'. In TimesTen, threads is always 1.</td>
</tr>
<tr>
<td>schema</td>
<td>The schema in which to recompile invalid objects. If NULL, all invalid objects in the database are recompiled.</td>
</tr>
<tr>
<td>flags</td>
<td>Flag values are intended for internal testing and diagnosability only.</td>
</tr>
</tbody>
</table>
RECOMP_SERIAL procedure

This procedure recompiles invalid objects in a given schema or all invalid objects in the database.

Syntax

```
UTL_RECOMP.RECOMP_SERIAL(
    schema   IN   VARCHAR2    DEFAULT NULL,
    flags    IN   BINARY_INTEGER DEFAULT 0);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>schema</td>
<td>The schema in which to recompile invalid objects. If NULL, all invalid objects in the database are recompiled.</td>
</tr>
<tr>
<td>flags</td>
<td>Flag values are intended for internal testing and diagnosability only.</td>
</tr>
</tbody>
</table>
anonymous PL/SQL blocks and dynamic SQL, 6-3
arrays
  BIND_ARRAY procedure, 6-8
  bulk DML using DBMS_SQL, 6-44

BINARY_DOUBLE_TABLE table type, 6-27
BINARY_FLOAT_TABLE table type, 6-28
BIND_ARRAY procedure, 6-43
BIND_VARIABLE procedure, 6-45
BIT_AND function, 11-7
BIT_COMPLEMENT function, 11-8
BIT_OR function, 11-9
BIT_XOR function, 11-10

BINARY_DOUBLE_TABLE table type, 6-27
BINARY_FLOAT_TABLE table type, 6-28
BIND_ARRAY procedure, 6-43
BIND_VARIABLE procedure, 6-45
BIT_AND function, 11-7
BIT_COMPLEMENT function, 11-8
BIT_OR function, 11-9
BIT_XOR function, 11-10

CAST_FROM_BINARY_DOUBLE function, 11-11
CAST_FROM_BINARY_FLOAT function, 11-13
CAST_FROM_BINARY_INTEGER function, 11-14
CAST_FROM_NUMBER function, 11-15
CAST_TO_BINARY_DOUBLE function, 11-16
CAST_TO_BINARY_FLOAT function, 11-18
CAST_TO_BINARY_INTEGER function, 11-20
CAST_TO_NUMBER function, 11-21
CAST_TO_NVARCHAR2 function, 11-22
CAST_TO_RAW function, 11-23
CAST_TO_VARCHAR2 function, 11-24
CHARARR table type, 3-9
CLOSE_CURSOR procedure, 6-47
collections
  scalar types for collections, 6-44
table items, 6-44
COLUMN_VALUE procedure, 6-48
COLUMN_VALUE_LONG procedure (not supported), 6-41
COMMA_TO_TABLE procedure, 7-9
COMPARE function, 11-25
COMPILE_SCHEMA procedure, 7-10
CONCAT function, 11-26
CONCAT function, 11-27
COPIES function, 11-28
cursors, DBMS_SQL package, 6-7

DATE_TABLE table type, 6-29
DB_VERSION procedure, 7-11
DBMS_LOCK package
  subprograms summary, 2-3
  using, 2-2
DBMS_OUTPUT package
  data structures, 3-8
  example, 3-7
  subprograms summary, 3-11
  using, 3-2
DBMS_PREPROCESSOR package
  data structures, 4-5
  subprogram summary, 4-7
  using, 4-2
DBMS_RANDOM package
  subprogram summary, 5-4
  using, 5-2
DBMS_SQL package
  data structures, 6-23
  examples, 6-11, 6-69, 6-71
  subprograms summary, 6-41
  using, 6-2
DBMSUTILITY package
  examples, 7-13, 7-24, 7-32
  subprograms summary, 7-7
  using, 7-2
DBMS_OUTPUT_LINESARRAY table type, 3-10
DEFINE_ARRAY procedure, 6-51
DEFINE_COLUMN procedure, 6-53
DEFINE_COLUMN_LONG procedure (not supported), 6-41
DESC_REC2 record type, 6-25
DESC_REC3 record type, 6-26
DESC_RESC record type, 6-24
DESC_TAB table type, 6-30
DESC_TAB2 table type, 6-31
DESC_TAB3 table type, 6-32
DESCRIBE_COLUMNS procedure, 6-55
DESCRIBE_COLUMNS2 procedure, 6-56
DESCRIBE_COLUMNS3 procedure, 6-57
DISABLE procedure, 3-12
dynamic SQL
anonymous blocks and, 6-3
DBMS_SQL functions, using, 6-2
execution flow in, 6-7

E
ENABLE procedure, 3-13
examples
DBMS_OUTPUT, 3-7
DBMS_SQL, 6-11
DBMS_SQL.TO_CURSOR_NUMBER, 6-69
DBMS_SQL.TO_REFCURSOR, 6-71
DBMS.Utility.FORMAT_ERROR_BACKTRACE, 7-13
DBMS.Utility.INVALIDATE, 7-24
DBMS.Utility.VALIDATE, 7-32
TT_DB_VERSION, 10-6
UTIL_FILE, 9-7
UTIL_IDENT, 10-6
EXECUTE function, 6-58
EXECUTE_AND_FETCH function, 6-59
execution flow in dynamic SQL, 6-7

F
FCLOSE procedure, 9-12
FCLOSE_ALL procedure, 9-13
FCOPY procedure, 9-14
FETCH_ROWS function, 6-60
FFLUSH procedure, 9-15
FGETATTR procedure, 9-16
FGETPOS function, 9-17
FILE_TYPE record type, 9-10
FOPEN function, 9-18
FOPEN_NCHAR function, 9-20
FORMAT_CALL_STACK function, 7-12
FORMAT_ERROR_BACKTRACE function, 7-13
FORMAT_ERROR_STACK function, 7-16
FREMOVE procedure, 9-21
FRENAME procedure, 9-22
FSEEK procedure, 9-23

G
GET_CPU_TIME function, 7-17
GET_DEPENDENCY procedure, 7-18
GET_ENDIANNESS function, 7-19
GET_HASH_VALUE function, 7-20
GET_LINE procedure, 3-14, 9-24
GET_LINE_NCHAR procedure, 9-25
GET_LINES procedure, 3-15
GET_POST_PROCESSED_SOURCE function, 4-8
GET_RAW function, 9-26
GET_SQL_HASH function, 7-21
GET_TIME function, 7-22

I
INITIALIZE procedure, 5-5
INTERVAL_DAY_TO_SECOND_TABLE table type, 6-33
INTERVAL_YEAR_TO_MONTH_TABLE table type, 6-34
INVALIDATE procedure, 7-23
IS_BIT_SET function, 7-26
IS_OPEN function, 6-61, 9-27

L
LAST_ERROR_POSITION function, 6-62
LAST_ROW_COUNT function, 6-63
LAST_ROW_ID function, 6-64
LAST_SQL_FUNCTION_CODE function, 6-65
LENGTH function, 11-29

N
NAME_RESOLVE procedure, 7-27
NAME_TOKENIZE procedure, 7-29
NEW_LINE procedure, 3-16, 9-28
NORMAL function, 5-6
NUMBER_TABLE table type, 6-35

O
OPEN_CURSOR function, 6-66
OVERLAY function, 11-30
overview of packages
package components, 1-4
referencing package contents, 1-6
summary of TimesTen packages, 1-8
using TimesTen packages, 1-5

P
PARSE procedure, 6-67
PRINT_POST_PROCESSED_SOURCE
procedure, 4-10
PUT, 9-30
PUT procedure, 3-17, 9-29
PUT_LINE procedure, 3-18, 9-30
PUT_LINE_NCHAR procedure, 9-31
PUT_NCHAR procedure, 9-32
PUT_RAW procedure, 9-33
PUTF procedure, 9-34
PUTF_NCHAR procedure, 9-35

R
RANDOM function, 5-7
RECOMP_PARALLEL procedure, 12-7
RECOMP_SERIAL procedure, 12-8
referencing package contents, 1-6
REVERSE function, 11-32

S
SEED procedure, 5-8
SLEEP procedure, 2-4
SOURCE_LINES_T table type, 4-6
STRING function, 5-9
SUBSTR function, 11-33
summary of TimesTen packages,  1-8

T

TABLE_TO_COMMA procedure,  7-30
tables, table items as arrays,  6-44
TERMINATE procedure,  5-10
TIME_TABLE table type,  6-36
TIMESTAMP_TABLE table type,  6-37
TO_CURSOR_NUMBER function,  6-69
TO_REFCURSOR function,  6-71
TRANSLATE function,  11-36
TRANSLITERATE function,  11-39
TT_DB_VERSION package
  constants,  8-4
  example,  10-6
  using,  8-2

U

using TimesTen packages,  1-5
UTL_FILE package
  data structures,  9-9
  examples,  9-7
  subprograms summary,  9-11
  using,  9-2
UTL_IDENT package
  constants,  10-5
  example,  10-6
  using,  10-2
UTL_RAW package
  subprograms summary,  11-5
  using,  11-2
UTL_RECOMP package
  subprograms summary,  12-6
  using,  12-2

V

VALIDATE procedure,  7-31
VALUE function,  5-11
VARCHAR2_TABLE table type,  6-38
VARCHAR2A table type,  6-39
VARCHAR2S table type,  6-40
VARIABLE_VALUE procedure,  6-73

X

XRANGE function,  11-43