Oracle® TimesTen In-Memory Database

PL/SQL Packages Reference

Release 22.1
F35405-03
February 2023

ORACLE

Oracle TimesTen In-Memory Database PL/SQL Packages Reference, Release 22.1

F35405-03

Contents

8 DBMS_UTILITY

9 TT_DB_VERSION

10 TT_STATS

11 UTL_FILE

12 UTL_IDENT

13 UTL_RAW

14 UTL_RECOMP

About This Content

This is a reference document for PL/SQL packages provided with TimesTen.

Audience

This document 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

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Related Resources

TimesTen documentation is available on the TimesTen documentation website.

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

Oracle Database documentation is also available on the Oracle documentation website. This may be especially useful for Oracle Database features that TimesTen supports but does not attempt to fully document, such as OCI and Pro*C/C++.

In particular, the following Oracle Database documents may be of interest.

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

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

Conventions

The following text conventions are used in this document.

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

What's New

This section summarizes new features and functionality of TimesTen Release 22.1.

New features in Release 22.1.1.1.0

- TimesTen 22.1 PL/SQL is based on the PL/SQL implementation for Oracle Database 19c.

1

Introduction to TimesTen-Supplied PL/SQL Packages and Types

A set of PL/SQL packages is supplied with TimesTen. These packages extend database functionality and allow PL/SQL access to SQL features.

This manual documents these public packages, listed with brief descriptions in Summary of TimesTen-Supplied PL/SQL Packages. Packages that are part of the PL/SQL language itself or are for TimesTen or Oracle Database internal use only are not shown here or described in this manual.

This chapter contains these topics:

- Package Overview
- Summary of TimesTen-Supplied PL/SQL Packages

For additional information about PL/SQL and PL/SQL packages, you can refer to the following:

- *Oracle TimesTen In-Memory Database PL/SQL Developer's Guide*
- *Oracle Database PL/SQL Language Reference*
- *Oracle Database PL/SQL Packages and Types Reference*

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
- Displaying the List of TimesTen-Supplied Packages
- 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.

Displaying the List of TimesTen-Supplied Packages

To display the list of packages currently installed in TimesTen, use the system view `ALL_PROCEDURES` for objects owned by `SYS`.

The following example, using `ttIsql`, 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_LOB >
< DBMS_LOCK >
< DBMS_OUTPUT >
< DBMS_PREPROCESSOR >
< DBMS_RANDOM >
< DBMS_SQL >
...
< DBMS_UTILITY >
...
< TT_STATS >
< UTL_FILE >
< UTL_RAW >
< UTL_RECOMP >
< UTL_IDENT >
< TT_DB_VERSION >
19 rows found.
```

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 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 Database user `SYSDBA`. Running as the `ADMIN` user is comparable to running as the Oracle Database user `DBA`.

Referencing Package Contents

To reference the types, items, and subprograms declared in a package specification, use "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

Lists of the PL/SQL packages supplied with TimesTen for public use. These packages run as the invoking user, rather than the package owner.

Note:

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

Table 1-1 Summary of TimesTen-Supplied PL/SQL Packages

Package Name	Description
DBMS_LOB	Provides subprograms to operate on binary and character large objects: BLOBs, CLOBs, and NCLOBs.
DBMS_LOCK	Provides an interface to Lock Management services. TimesTen supports only the `SLEEP` procedure, to suspend the session for a specified duration.
DBMS_OUTPUT	Enables you to send messages from stored procedures and packages.
DBMS_PREPROCESSOR	Provides an interface to print or retrieve the source text of a PL/SQL unit in its post-processed form.
DBMS_RANDOM	Provides a built-in random number generator.
DBMS_SQL	Lets you use dynamic SQL to access the database.
DBMS_UTILITY	Provides various utility routines.
TT_DB_VERSION	Indicates the TimesTen major and minor version numbers.
TT_STATS	Collects snapshots of database metrics and generates reports based on comparisons between snapshots.
UTL_FILE	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.
UTL_IDENT	Indicates in which database or client PL/SQL is running, such as TimesTen or Oracle Database, and server versus client. (Each database or client running PL/SQL has its own copy of this package.)
UTL_RAW	Provides SQL functions for manipulating `RAW` data types.
UTL_RECOMP	Recompiles invalid PL/SQL modules.

Note:

- 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.

2
DBMS_LOB

TimesTen Classic supports LOBs (large objects). The `DBMS_LOB` package provides subprograms to operate on BLOBs, CLOBs, and NCLOBs. You can use `DBMS_LOB` to access and manipulate specific parts of LOBs or complete LOBs.

This chapter contains the following topics:

You can also refer to Large objects (LOBs) in *Oracle TimesTen In-Memory Database PL/SQL Developer's Guide*.

Using DBMS_LOB

Overview

`DBMS_LOB` can read, manipulate, and modify BLOBs, CLOBs, and NCLOBs.

For an overview of LOBs, see Introduction to Large Objects and SecureFiles in *Oracle Database SecureFiles and Large Objects Developer's Guide*.

Security Model

Operations provided by this package are performed under the current calling user, not under the package owner SYS.

Any DBMS_LOB subprogram called from an anonymous PL/SQL block is executed using the privileges of the current user. Any DBMS_LOB subprogram called from a stored procedure is executed using the privileges of the owner of the stored procedure.

When creating the procedure, users can set the AUTHID to indicate whether they want definer's rights or invoker's rights. For example:

```
CREATE PROCEDURE proc1 AUTHID DEFINER ...
```

Or:

```
CREATE PROCEDURE proc1 AUTHID CURRENT_USER ...
```

See Definer's Rights and Invoker's Rights (AUTHID Clause) in *Oracle TimesTen In-Memory Database Security Guide*. For information about the security model pertaining to temporary LOBs, see Operational Notes.

Constants

The DBMS_LOB package uses the constants shown in Table 2-1:

Table 2-1 DBMS_LOB Constants

Constant	Type	Value	Description
CALL	BINARY_INTEGER	12	Create the temporary LOB with call duration.
DEFAULT_CSID	INTEGER	0	This is the default character set ID.
DEFAULT_LANG_CTX	INTEGER	0	This is the default language context.
LOB_READONLY	BINARY_INTEGER	0	Open the specified LOB read-only.
LOB_READWRITE	BINARY_INTEGER	1	Open the specified LOB read/write.
BLOBMAXSIZE	INTEGER	16777216 (16 MB)	Set maximum size of a BLOB in bytes.
CLOBMAXSIZE	INTEGER	4194304 (4 MB)	Set maximum size of a CLOB in bytes.
NO_WARNING	INTEGER	0	Indicates success, no warning message.

Table 2-1 (Cont.) DBMS_LOB Constants

Constant	Type	Value	Description
SESSION	BINARY_INTEGER	10	Create the temporary LOB with session duration. **Note**: In TimesTen, LOB duration cannot extend past the end of the transaction. Temporary LOB contents are destroyed when the corresponding locator is invalidated at the end of the transaction.
TRANSACTION	BINARY_INTEGER	11	Create the temporary LOB with transaction duration.
WARN_INCONVERTIBLE_CHAR	INTEGER	1	Used by the conversion functions to indicate there is an inconvertible character.

Note:

- 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.

Data Types

The DBMS_LOB package uses the data types shown in Table 2-2.

Table 2-2 Data Types Used by DBMS_LOB

Type	Description
BLOB	Source or destination binary LOB
RAW	Source or destination RAW buffer (used with BLOBs)
CLOB	Source or destination character LOB (including NCLOBs)
VARCHAR2	Source or destination character buffer (used with CLOBs and NCLOBs)
INTEGER	Size of a buffer or LOB, offset into a LOB, or amount to access (in bytes for BLOBs or characters for CLOBs or NCLOBs)

The DBMS_LOB package defines no special types.

An NCLOB is a CLOB for holding fixed-width and varying-width, multibyte national character sets.

The clause ANY_CS in the specification of DBMS_LOB subprograms for CLOBs enables the CLOB type to accept a CLOB or NCLOB locator variable as input.

Rules and Limits

General Rules and Limits

- The following rules apply in the specification of subprograms in this package:
  - The *newlen*, *offset*, and *amount* parameters for subprograms operating on BLOBs must be specified in terms of bytes.
  - The *newlen*, *offset*, and *amount* parameters for subprograms operating on CLOBs must be specified in terms of characters.
- A subprogram raises an INVALID_ARGVAL exception if the following restrictions are not followed in specifying values for parameters (unless otherwise specified):
  1. Only positive, absolute offsets from the beginning of LOB data are permitted. Negative offsets from the tail of the LOB are not permitted.
  2. Only positive, nonzero values are permitted for the parameters that represent size and positional quantities, such as *amount*, *offset*, *newlen*, *nth*, and so on. Negative offsets and ranges observed in SQL string functions and operators are not permitted.
  3. The value of *offset*, *amount*, *newlen*, and *nth* must not exceed the value BLOBMAXSIZE (for a BLOB) or CLOBMAXSIZE (for a CLOB or NCLOB) in any DBMS_LOB subprogram. In TimesTen, the maximum BLOB size is 16 MB and the maximum CLOB or NCLOB size is 4 MB.
  4. For CLOBs consisting of fixed-width multibyte characters, the maximum value for these parameters must not exceed (CLOBMAXSIZE/ *character_width_in_bytes*) characters.
- PL/SQL language specifications stipulate an upper limit of 32767 bytes (not characters) for RAW and VARCHAR2 parameters used in DBMS_LOB subprograms. For example, if you declare a variable as follows:

  ```
  charbuf VARCHAR2(3000)
  ```

  Then charbuf can hold 3000 single byte characters or 1500 two-byte fixed width characters. This has an important consequence for DBMS_LOB subprograms for CLOBs and NCLOBs.
- The %CHARSET clause indicates that the form of the parameter with %CHARSET must match the form of the ANY_CS parameter to which it refers.

  For example, in DBMS_LOB subprograms that take a VARCHAR2 buffer parameter, the form of the VARCHAR2 buffer must be appropriate for the form of the character LOB

parameter. If the specified LOB is of type NCLOB, the buffer must contain NCHAR data. If the specified LOB is of type CLOB, the buffer must contain CHAR data.

For DBMS_LOB subprograms that take two-character LOB parameters, both parameters must have the same form. That is, they must both be NCLOBs or they must both be CLOBs.

- If the value of *amount* plus *offset* exceeds the maximum LOB size allowed by the database, then access exceptions are raised. In TimesTen, the maximum BLOB size is 16 MB and the maximum CLOB or NCLOB size is 4 MB.

  Under these input conditions, subprograms such as READ, COMPARE, INSTR, and SUBSTR read until the end of the LOB is reached. For example, for a READ operation on a BLOB, if the user specifies an *offset* value of 3 MB and an amount value of 2 MB on a LOB that is 4 MB, then READ returns only 1 MB (4 MB minus 3 MB).

- Functions with NULL or invalid input values for parameters return NULL. Procedures with NULL values for destination LOB parameters raise exceptions.
- Operations involving patterns as parameters, such as COMPARE, INSTR, and SUBSTR, do not support regular expressions or special matching characters (such as % in the LIKE operator in SQL) in the pattern parameter or substrings.
- The end-of-LOB condition is indicated by the READ procedure using a NO_DATA_FOUND exception. This exception is raised only upon an attempt by the user to read beyond the end of the LOB. The READ buffer for the last read contains 0 bytes.
- Unless otherwise stated, the default value for an *offset* parameter is 1, which indicates the first byte in the BLOB data or the first character in the CLOB or NCLOB data. No default values are specified for the *amount* parameter. You must input the values explicitly.
- You must lock the row containing the destination LOB before calling any subprograms that modify the LOB, such as APPEND, COPY, ERASE, TRIM, or WRITE. These subprograms do not implicitly lock the row containing the LOB.

Maximum LOB Size

The maximum size for LOBs in TimesTen is 16 MB for BLOBs and 4 MB for CLOBs or NCLOBs.

Maximum Buffer Size

The maximum buffer size is 32767 bytes.

For BLOBs, where buffer size is expressed in bytes, the number of bytes cannot exceed 32767.

For CLOBs or NCLOBs, where buffer size is expressed in characters, the number of characters cannot result in a buffer larger than 32767 bytes. For example, if you are using fixed-width, two-byte characters, then specifying 20000 characters is an error (20000*2 = 40000, which is greater than 32767).

Operational Notes

This section discusses how to use LOBS and the DBMS_LOB package, covering these topics:

- LOB Usage Notes
- Persistent LOBs

- Temporary LOBs

LOB Usage Notes

DBMS_LOB subprograms operate based on LOB locators. For the successful completion of DBMS_LOB subprograms, you must provide an input locator that represents a LOB, either a temporary LOB (discussed below) or a persistent LOB that already exists in the database tablespaces.

> **Tip:**
>
> - In TimesTen, LOB locators do not remain valid past the end of the transaction.
> - LOB manipulations through APIs that use LOB locators result in usage of TimesTen temporary space. Any significant number of such manipulations may necessitate a size increase for the TimesTen temporary data partition. See TempSize in *Oracle TimesTen In-Memory Database Reference*.

To use LOBs in your database, you must first use SQL data definition language (DDL) to define the tables that contain columns of type BLOB, CLOB, or NCLOB.

In TimesTen, you can write data into the middle of a LOB only by overwriting previous data. There is no functionality to insert data into the middle of a LOB and move previous data, beginning at that point, toward the end of the LOB correspondingly. Similarly, in TimesTen you can delete data from the middle of a LOB only by overwriting previous data with zeros or null data. There is no functionality to remove data from the middle of a LOB and move previous data, beginning at that point, toward the beginning of the LOB correspondingly. In either case in TimesTen, the size of the LOB does not change, except in the circumstance where from the specified offset there is less space available in the LOB than there is data to write. (In Oracle Database you can use the DBMS_LOB FRAGMENT procedures to insert or delete data, move other data accordingly, and change the size of the LOB. TimesTen does not support those procedures.)

DBMS_LOB procedures and functions are supported for both TimesTen LOBs and passthrough LOBs, which are LOBs in Oracle Database accessed through TimesTen and exposed as TimesTen LOBs. Note, however, that CREATETEMPORARY can only be used to create a temporary LOB in TimesTen. If a temporary passthrough LOB is created using some other mechanism, such as SQL, ISTEMPORARY and FREETEMPORARY can be used on that LOB.

TimesTen does not support DBMS_LOB subprograms intended specifically for BFILEs, SecureFiles (including Database File System features), or inserting or deleting data fragments in the middle of a LOB (FRAGMENT subprograms).

Persistent LOBs

To populate your database table with LOBs after BLOB, CLOB, or NCLOB columns are defined in the table, use the SQL data manipulation language (DML) to initialize or populate the locators in the LOB columns.

Temporary LOBs

TimesTen supports the definition, creation, deletion, access, and update of temporary LOBs. The temporary data partition stores the temporary LOB data. Temporary LOBs are not permanently stored in the database. Their primary purpose is for performing transformations on LOB data from applications.

You can use PL/SQL to create or manipulate a temporary LOB (BLOB, CLOB, or NCLOB).

A temporary LOB is empty when it is created. In TimesTen, all temporary LOBs are deleted at the end of the transaction in which they were created. Also, if a process dies unexpectedly or if the database crashes, temporary LOBs are deleted and the space for temporary LOBs is freed.

There is no support for consistent-read, undo, backup, parallel processing, or transaction management for temporary LOBs. Because consistent-read and rollbacks are not supported for temporary LOBs, you must free the temporary LOB and start over again if you encounter an error.

In PL/SQL, do not use more than one locator for a temporary LOB. Because consistent-read, undo, and versions are not generated for temporary LOBs, there is potentially a performance impact if you assign multiple locators to the same temporary LOB. Semantically, each locator should have its own copy of the temporary LOB. A temporary LOB locator can be passed by reference to other procedures if necessary.

A copy of a temporary LOB is created if the user modifies the temporary LOB while another locator is also pointing to it. The locator on which a modification was performed now points to a new copy of the temporary LOB. Other locators no longer see the same data as the locator through which the modification was made. A deep copy was not incurred by persistent LOBs in these types of situations, because consistent-read snapshots and version pages enable users to see their own versions of the LOB cheaply.

Because temporary LOBs are not associated with any table schema, there are no meanings to the terms in-row and out-of-row for temporary LOBs. Creation of a temporary LOB instance by a user causes the engine to create and return a locator to the LOB data. The PL/SQL `DBMS_LOB` package, as well as other programmatic interfaces, operates on temporary LOBs through these locators just as they do for persistent LOBs.

There is no concept of client-side temporary LOBs. All temporary LOBs reside in the server.

A temporary LOB instance can be accessed and modified using appropriate `DBMS_LOB` functions and procedures, just as for persistent LOBs. To make a temporary LOB persistent, you can use the `COPY` procedure to copy it into a `BLOB`, `CLOB`, or `NCLOB` column (as appropriate) in the database.

When you are finished with a temporary LOB instance, use the `FREETEMPORARY` procedure to free it.

Security is provided through the LOB locator. Only the user who created the temporary LOB can see it. Locators cannot be passed from one user session to another. Even if someone did pass a locator from one session to another, they would not access the temporary LOBs from the original session.

The following notes are specific to temporary LOBs:

- All functions in `DBMS_LOB` return `NULL` if any parameter is `NULL`. All procedures in `DBMS_LOB` raise an exception if the LOB locator is input as `NULL`.

- Operations based on CLOBs do not verify whether the character set IDs (CSIDs) of the parameters (`CLOB` parameters, `VARCHAR2` buffers and patterns, and so on) match. It is the user's responsibility to ensure this.

Exceptions

Table 2-3 DBMS_LOB Exceptions

Exception	Code	Description
ACCESS_ERROR	22925	There was an attempt to write too much data to the LOB. In TimesTen, BLOB size is limited to 16 MB and CLOB or NCLOB size is limited to 4 MB.
BUFFERING_ENABLED	22279	Cannot perform operation with LOB buffering enabled.
CONTENTTYPE_TOOLONG	43859	The length of the `contenttype` string exceeds the defined maximum. Modify the length of the `contenttype` string and retry the operation.
CONTENTTYPEBUF_WRONG	43862	The length of the `contenttype` buffer is less than the defined constant. Modify the length of the `contenttype` buffer and retry the operation.
INVALID_ARGVAL	21560	The argument is expecting a valid non-null value but the argument value passed in is `NULL`, invalid, or out of range.
NO_DATA_FOUND	1403	This is the end-of-LOB indicator for looping read operations. It is not a hard error.
QUERY_WRITE	14553	Cannot perform a LOB write inside a query. (This is not applicable for TimesTen.)
VALUE_ERROR	6502	This is a PL/SQL error for invalid values to subprogram parameters.

DBMS_LOB Subprograms

Table 2-4 summarizes the `DBMS_LOB` subprograms, followed by a full description of each subprogram.

Table 2-4 DBMS_LOB Subprograms

Subprogram	Description
APPEND Procedures	Appends the contents of the source LOB to the destination LOB.
CLOSE Procedures	Closes a previously opened LOB.
COMPARE Functions	Compares two entire LOBs or parts of two LOBs.
CONVERTTOBLOB Procedure	Reads character data from a source CLOB or NCLOB, converts the character data to the specified character set, writes the converted data to a destination BLOB in binary format, and returns the new offsets.

Table 2-4 (Cont.) DBMS_LOB Subprograms

Subprogram	Description
CONVERTTOCLOB Procedure	Takes a source BLOB, converts the binary data in the source to character data using the specified character set, writes the character data to a destination CLOB or NCLOB, and returns the new offsets.
COPY Procedures	Copies all or part of the source LOB to the destination LOB.
CREATETEMPORARY Procedures	Creates a temporary LOB in the temporary data partition.
ERASE Procedures	Erases all or part of a LOB.
FREETEMPORARY Procedures	Frees a temporary LOB in the temporary data partition.
GETCHUNKSIZE Functions	Returns the amount of space used in the LOB chunk to store the LOB value.
GETLENGTH Functions	Returns the length of the LOB value, in bytes for a BLOB or characters for a CLOB.
GET_STORAGE_LIMIT Functions	Returns the storage limit for the LOB type of the specified LOB.
INSTR Functions	Returns the matching position of the *n*th occurrence of the pattern in the LOB.
ISOPEN Functions	Checks to see if the LOB was already opened using the input locator.
ISTEMPORARY Functions	Checks if the locator is pointing to a temporary LOB.
OPEN Procedures	Opens a LOB (persistent or temporary) in the indicated mode, read/write or read-only.
READ Procedures	Reads data from the LOB starting at the specified offset.
SUBSTR Functions	Returns part of the LOB value starting at the specified offset.
TRIM Procedures	Trims the LOB value to the specified length.
WRITE Procedures	Writes data to the LOB from a specified offset.
WRITEAPPEND Procedures	Appends a buffer to the end of a LOB.

APPEND Procedures

This procedure appends the contents of a source LOB to a destination LOB. It appends the complete source LOB. (Do not confuse this with the `WRITEAPPEND` procedure.)

> **Note:**
>
> Also see WRITEAPPEND Procedures.

Syntax

```
DBMS_LOB.APPEND (
   dest_lob IN OUT  NOCOPY BLOB,
   src_lob  IN             BLOB);

DBMS_LOB.APPEND (
   dest_lob IN OUT  NOCOPY CLOB  CHARACTER SET ANY_CS,
   src_lob  IN             CLOB  CHARACTER SET dest_lob%CHARSET);
```

Parameters

Table 2-5 APPEND Procedure Parameters

Parameter	Description
dest_lob	Locator for the LOB to which the data is being appended
src_lob	Locator for the LOB from which the data is being read

Usage Notes

- It is recommended that you enclose write operations to the LOB with `OPEN` and `CLOSE` calls, but not mandatory. If you opened the LOB before performing the operation, however, you must close it before you commit or roll back the transaction.

Exceptions

Table 2-6 APPEND Procedure Exceptions

Exception	Description
`VALUE_ERROR`	Either the source or destination LOB is `NULL`.
`QUERY_WRITE`	Cannot perform a LOB write inside a query. (This is not applicable for TimesTen.)
`BUFFERING_ENABLED`	Cannot perform operation if LOB buffering is enabled on either LOB.

Examples

This example shows use of the `APPEND` procedure.

```
create table t1 (a int, c clob);

insert into t1(a,c) values(1, 'abcde');
1 row inserted.

commit;

declare
  c1 clob;
  c2 clob;
begin
```

```
  c1 := 'abc';
  select c into c2 from t1 where a = 1;
  dbms_output.put_line('c1 before append is ' || c1);
  dbms_output.put_line('c2 before append is ' || c2);
  dbms_lob.append(c1, c2);
  dbms_output.put_line('c1 after append is ' || c1);
  dbms_output.put_line('c2 after append is ' || c2);
  insert into t1 values (2, c1);
end;

c1 before append is abc
c2 before append is abcde
c1 after append is abcabcde
c2 after append is abcde

PL/SQL procedure successfully completed.

select * from t1;
< 1, abcde >
< 2, abcabcde >
2 rows found.
```

(Output is shown after running the commands from a SQL script.)

CLOSE Procedures

This procedure closes a previously opened LOB.

Syntax

```
DBMS_LOB.CLOSE (
   lob_loc    IN OUT NOCOPY BLOB);

DBMS_LOB.CLOSE (
   lob_loc    IN OUT NOCOPY CLOB CHARACTER SET ANY_CS);
```

Parameters

Table 2-7 CLOSE Procedure Parameters

Parameter	Description
lob_loc	Locator for the LOB

Usage Notes

- `CLOSE` requires a round-trip to the server.
- It is not mandatory that you wrap LOB operations inside `OPEN` and `CLOSE` calls. However, if you open a LOB, you must close it before you commit or roll back the transaction.
- It is an error to commit the transaction before closing all LOBs that were opened by the transaction. When the error is returned, the "open" status of the open LOBs is discarded,

but the transaction is successfully committed. Hence, all the changes made to the LOB and non-LOB data in the transaction are committed.

Exceptions

An error is returned if the LOB is not open.

COMPARE Functions

This function compares two entire LOBs or parts of two LOBs.

Syntax

```
DBMS_LOB.COMPARE (
   lob_1            IN BLOB,
   lob_2            IN BLOB,
   amount           IN INTEGER := DBMS_LOB.BLOBMAXSIZE,
   offset_1         IN INTEGER := 1,
   offset_2         IN INTEGER := 1)
  RETURN INTEGER;

DBMS_LOB.COMPARE (
   lob_1            IN CLOB  CHARACTER SET ANY_CS,
   lob_2            IN CLOB  CHARACTER SET lob_1%CHARSET,
   amount           IN INTEGER := DBMS_LOB.CLOBMAXSIZE,
   offset_1         IN INTEGER := 1,
   offset_2         IN INTEGER := 1)
  RETURN INTEGER;
```

Parameters

Table 2-8 COMPARE Function Parameters

Parameter	Description
lob_1	Locator for the first LOB for comparison
lob_2	Locator for the second LOB for comparison
amount	Number of bytes (for BLOBs) or characters (for CLOBs or NCLOBs) to compare
offset_1	Offset in bytes or characters in the first LOB (starting from 1)
offset_2	Offset in bytes or characters in the second LOB (starting from 1)

Return Values

The function returns one of the following:

- 0 (zero) if the data matches exactly over the specified range
- -1 if the first LOB is less than the second
- 1 if the first LOB is greater than the second
- `NULL` if *amount*, *offset_1*, or *offset_2* is an invalid value, outside the range 1 to `BLOBMAXSIZE` or `CLOBMAXSIZE` (as appropriate), inclusive

Usage Notes

- You can only compare LOBs of the same type. For example, you cannot compare a BLOB to a CLOB.
- For fixed-width *n*-byte CLOBs or NCLOBs, if the input amount for `COMPARE` is specified to be greater than `CLOBMAXSIZE/n`, then `COMPARE` matches characters in a range of size that is either `CLOBMAXSIZE/n` or `Max(length(clob1), length(clob2))`, whichever is less.

CONVERTTOBLOB Procedure

This procedure reads character data from a source CLOB or NCLOB, converts the character data to the character set you specify, writes the converted data to a destination BLOB in binary format, and returns the new offsets. You can use this procedure with any combination of persistent or temporary LOBs.

Syntax

```
DBMS_LOB.CONVERTTOBLOB(
  dest_lob       IN OUT     NOCOPY  BLOB,
  src_clob       IN         CLOB CHARACTER SET ANY_CS,
  amount         IN         INTEGER,
  dest_offset    IN OUT     INTEGER,
  src_offset     IN OUT     INTEGER,
  blob_csid      IN         NUMBER,
  lang_context   IN OUT     INTEGER,
  warning        OUT        INTEGER);
```

Parameters

Table 2-9 CONVERTTOBLOB Procedure Parameters

Parameter	Description
dest_lob	Locator for the destination LOB
src_clob	Locator for the source LOB
amount	Number of characters to convert from the source LOB If you want to convert the entire CLOB or NCLOB, pass the constant `CLOBMAXSIZE`. If you pass any other value, it must be less than or equal to the size of the LOB.
dest_offset	`(IN)` Offset in bytes in the destination LOB for the start of the write Specify a value of 1 to start at the beginning of the LOB. `(OUT)` The new offset in bytes after the end of the write
src_offset	`(IN)` Offset in characters in the source LOB for the start of the read `(OUT)` Offset in characters in the source LOB right after the end of the read
blob_csid	Character set ID for the converted BLOB data

Table 2-9 (Cont.) CONVERTTOBLOB Procedure Parameters

Parameter	Description
lang_context	(IN) Language context, such as shift status, for the current conversion (ignored by TimesTen) (OUT) The language context at the time when the current conversion is done (set to 0 by TimesTen) This parameter is not supported by TimesTen.
warning	Warning message This parameter is not supported by TimesTen.

Usage Notes

This section discusses special usage notes for CONVERTTOBLOB.

Preconditions

Before a call to CONVERTTOBLOB, the following preconditions must be met.

- Both the source and destination LOBs must exist.
- If the destination LOB is a persistent LOB, the row must be locked. To lock the row, select the LOB using the FOR UPDATE clause of the SELECT statement.

Constants and Defaults

All parameters are required. You must pass a variable for each OUT or IN OUT parameter. You must pass either a variable or a value for each IN parameter.

Table 2-10 gives a summary of typical values for each parameter. Note that constants are used for some values. These constants are defined in the dbmslob.sql package specification file.

Table 2-10 CONVERTTOBLOB Typical Values

Parameter	Value	Description
amount	CLOBMAXSIZE (IN)	Convert the entire LOB.
dest_offset	1 (IN)	Start from the beginning.
src_offset	1 (IN)	Start from the beginning.
blob_csid	DEFAULT_CSID (IN)	Default character set ID, use same ID as source CLOB.
lang_context	DEFAULT_LANG_CTX (IN)	This is the default language context (ignored by TimesTen).
warning	NO_WARNING (OUT) WARN_INCONVERTIBLE_CHAR (OUT)	This is a warning message (ignored by TimesTen).

General Notes

- You must specify the desired character set ID for the destination BLOB in the *blob_csid* parameter. If you pass a zero value, the database assumes that the desired character set is the same as the source CLOB character set.
- You must specify the offsets for both the source and destination LOBs, and the number of characters to copy from the source LOB. The *amount* and *src_offset* values are in characters and the *dest_offset* is in bytes. To convert the entire LOB, you can specify CLOBMAXSIZE for the *amount* parameter.
- CONVERTTOBLOB gets the source and destination LOBs as necessary before converting and writing the data.

Exceptions

Table 2-11 CONVERTTOBLOB Procedure Exceptions

Exception	Description
VALUE_ERROR	Any of the input parameters is NULL or invalid.
INVALID_ARGVAL	Any of the following is true: *src_offset* < 1 or *src_offset* > CLOBMAXSIZE *dest_offset* < 1 or *dest_offset* > BLOBMAXSIZE *amount* < 1 or *amount* > CLOBMAXSIZE

CONVERTTOCLOB Procedure

This procedure takes a source BLOB, converts the binary data in the source to character data using the character set you specify, writes the character data to a destination CLOB or NCLOB, and returns the new offsets. You can use this procedure with any combination of persistent or temporary LOBs.

Syntax

```
DBMS_LOB.CONVERTTOCLOB(
   dest_lob       IN OUT NOCOPY  CLOB CHARACTER SET ANY_CS,
   src_blob       IN             BLOB,
   amount         IN             INTEGER,
   dest_offset    IN OUT         INTEGER,
   src_offset     IN OUT         INTEGER,
   blob_csid      IN             NUMBER,
   lang_context   IN OUT         INTEGER,
   warning        OUT            INTEGER);
```

Parameters

Table 2-12 CONVERTTOCLOB Procedure Parameters

Parameter	Description
dest_lob	Locator for the destination LOB

Table 2-12 (Cont.) CONVERTTOCLOB Procedure Parameters

Parameter	Description
src_blob	Locator for the source LOB
amount	Number of bytes to convert from the source LOB If you want to convert the entire BLOB, pass the constant `BLOBMAXSIZE`. If you pass any other value, it must be less than or equal to the size of the BLOB.
dest_offset	`(IN)` Offset in characters in the destination LOB for the start of the write Specify a value of 1 to start at the beginning of the LOB. `(OUT)` The new offset in characters after the end of the write This offset always points to the beginning of the first complete character after the end of the write.
src_offset	`(IN)` Offset in bytes in the source LOB for the start of the read `(OUT)` Offset in bytes in the source LOB right after the end of the read
blob_csid	Character set ID for the source BLOB data
lang_context	`(IN)` Language context, such as shift status, for the current conversion (ignored by TimesTen) `(OUT)` Language context at the time when the current conversion is done (set to 0 by TimesTen) This parameter is not supported by TimesTen.
warning	Warning message This parameter is not supported by TimesTen.

Usage Notes

This section discusses special usage notes for `CONVERTTOCLOB`.

Preconditions

Before a call to `CONVERTTOCLOB`, the following preconditions must be met.

- Both the source and destination LOBs must exist.
- If the destination LOB is a persistent LOB, the row must be locked before calling the `CONVERTTOCLOB` procedure. To lock the row, select the LOB using the `FOR UPDATE` clause of the `SELECT` statement.

Constants and Defaults

All parameters are required. You must pass a variable for each `OUT` or `IN OUT` parameter. You must pass either a variable or a value for each `IN` parameter.

Table 2-13 gives a summary of typical values for each parameter. Note that constants are used for some values. These constants are defined in the `dbmslob.sql` package specification file.

Table 2-13 CONVERTTOCLOB Typical Values

Parameter	Value	Description
amount	BLOBMAXSIZE (IN)	Convert the entire LOB.
dest_offset	1 (IN)	Start from the beginning.
src_offset	1 (IN)	Start from the beginning.
blob_csid	DEFAULT_CSID (IN)	Default character set ID, use same ID as destination CLOB.
lang_context	DEFAULT_LANG_CTX (IN)	This is the default language context (ignored by TimesTen).
warning	NO_WARNING (OUT) WARN_INCONVERTIBLE_CHAR (OUT)	This is a warning message (ignored by TimesTen).

General Notes

- You must specify the desired character set ID for the source BLOB in the *blob_csid* parameter. If you pass a zero value, the database assumes that the desired character set is the same as the destination CLOB character set.
- You must specify the offsets for both the source and destination LOBs, and the number of characters to copy from the source LOB. The *amount* and *src_offset* values are in bytes and the *dest_offset* is in characters. To convert the entire LOB, you can specify BLOBMAXSIZE for the *amount* parameter.
- CONVERTTOCLOB gets the source and destination LOBs as necessary before converting and writing the data.

Exceptions

Table 2-14 CONVERTTOCLOB Procedure Exceptions

Exception	Description
VALUE_ERROR	Any of the input parameters is NULL or invalid.
INVALID_ARGVAL	Any of the following is true: *src_offset* < 1 or *src_offset* > BLOBMAXSIZE *dest_offset* < 1 or *dest_offset* > CLOBMAXSIZE *amount* < 1 or *amount* > BLOBMAXSIZE

COPY Procedures

This procedure copies all or part of a source LOB to a destination LOB. You can specify the offsets for both the source and destination LOBs, and the number of bytes or characters to copy.

Syntax

```
DBMS_LOB.COPY (
   dest_lob     IN OUT NOCOPY BLOB,
   src_lob      IN            BLOB,
```

```
   amount      IN            INTEGER,
   dest_offset IN            INTEGER := 1,
   src_offset  IN            INTEGER:= 1);

DBMS_LOB.COPY (
   dest_lob    IN OUT NOCOPY CLOB CHARACTER SET ANY_CS,
   src_lob     IN            CLOB CHARACTER SET dest_lob%CHARSET,
   amount      IN INTEGER,
   dest_offset IN INTEGER := 1,
   src_offset  IN INTEGER := 1);
```

Parameters

Table 2-15 COPY Procedure Parameters

Parameter	Description
dest_lob	Locator for the destination LOB being copied to
src_lob	Locator for the source LOB being copied from
amount	Number of bytes (for BLOBs) or characters (for CLOBs or NCLOBs) to copy
dest_offset	Offset in bytes or characters in the destination LOB for the start of the copy (starting from 1)
src_offset	Offset in bytes or characters in the source LOB for the start of the copy (starting from 1)

Usage Notes

- If the offset you specify in the destination LOB is beyond the end of the data currently in this LOB, then zero-byte fillers (for BLOBs) or spaces (for CLOBs or NCLOBs) are inserted in the destination LOB to reach the offset. If the offset is less than the current length of the destination LOB, then existing data is overwritten.
- It is not an error to specify an amount that exceeds the length of the data in the source LOB. Thus, you can specify a large amount to copy from the source LOB, which copies data from the *src_offset* to the end of the source LOB.
- It is recommended that you enclose write operations to the LOB with `OPEN` and `CLOSE` calls, but not mandatory. However, if you opened the LOB before performing the operation, you must close it before you commit or roll back the transaction.
- In addition to copying from one TimesTen LOB to another, `COPY` can copy from a TimesTen LOB to a passthrough LOB, from a passthrough LOB to a TimesTen LOB, or from one passthrough LOB to another passthrough LOB. An attempt to copy a passthrough LOB to a TimesTen LOB when the passthrough LOB is larger than the TimesTen LOB size limit results in an error.

Exceptions

Maximum LOB size is `BLOBMAXSIZE` for a BLOB or `CLOBMAXSIZE` for a CLOB.

Table 2-16 COPY Procedure Exceptions

Exception	Description
VALUE_ERROR	Any of the input parameters is NULL or invalid.
INVALID_ARGVAL	Any of the following is true: *src_offset* < 1 or *src_offset* > maximum LOB size *dest_offset* < 1 or *dest_offset* > maximum LOB size *amount* < 1 or *amount* > maximum LOB size
QUERY_WRITE	Cannot perform a LOB write inside a query. (This is not applicable for TimesTen.)
BUFFERING_ENABLED	Cannot perform the operation if LOB buffering is enabled on either LOB.

Examples

The examples in this section show how to copy LOBs in PL/SQL, copying between passthrough LOBs (from Oracle Database) and TimesTen LOBs. The first example uses the COPY procedure. The second, as contrast, simply uses INSERT and UPDATE statements, though also uses functionality of the DBMS_LOB package.

Copy CLOBs Using COPY Procedure

This example uses the COPY procedure to first copy a passthrough CLOB from Oracle Database into a TimesTen CLOB, then to copy a TimesTen CLOB into a passthrough CLOB.

```
autocommit 0;
passthrough 0;
DROP TABLE tt_table; CREATE TABLE tt_table (i INT, c CLOB); COMMIT;
passthrough 3;
DROP TABLE ora_table; CREATE TABLE ora_table (i INT, c CLOB); COMMIT;
passthrough 0;
set serveroutput on;

DECLARE
  passthru_clob CLOB;
  tt_clob CLOB;
  clob_length BINARY_INTEGER;
  clob_buffer VARCHAR2(80);

BEGIN
  EXECUTE IMMEDIATE 'call ttoptsetflag(''passthrough'', 1)';

  -- Note that in PL/SQL, passthrough statements must be executed as
  -- dynamic SQL, and SELECT INTO must be used to assign a passthrough LOB.

  -- 1. Copy a passthrough CLOB on Oracle Database to a TimesTen CLOB
  -- On Oracle Database : insert a row with an empty CLOB, get a passthrough
CLOB
  -- handle, and append to the passthrough CLOB.
  EXECUTE IMMEDIATE 'INSERT INTO ora_table VALUES (1, EMPTY_CLOB())';
  EXECUTE IMMEDIATE 'SELECT c FROM ora_table WHERE i = 1 FOR UPDATE'
                    INTO passthru_clob;
```

```
  DBMS_LOB.APPEND(passthru_clob, 'Copy from Oracle Database to
TimesTen');
  clob_length := DBMS_LOB.GETLENGTH(passthru_clob);

  -- On TimesTen: insert a row with an empty CLOB, and get a TimesTen
CLOB handle
  INSERT INTO tt_table VALUES (1, EMPTY_CLOB()) RETURNING c INTO
tt_clob;

  -- Copy the passthrough CLOB on Oracle Database to a TimesTen CLOB
  DBMS_LOB.COPY(tt_clob, passthru_clob, clob_length, 1, 1);

  -- On TimesTen: display the modified TimesTen CLOB
  DBMS_LOB.READ(tt_clob, clob_length, 1, clob_buffer);
  DBMS_OUTPUT.PUT_LINE(clob_buffer);

  -- 2. Copy a TimesTen CLOB to a passthrough CLOB on Oracle Database
  -- On TimesTen: insert a row with LOB data, and get a TimesTen CLOB
handle
  INSERT INTO tt_table VALUES (2, 'Copy from TimesTen to Oracle
Database.')
         RETURNING c INTO tt_clob;
  clob_length := DBMS_LOB.GETLENGTH(tt_clob);

  -- On Oracle Database: insert a row with an empty CLOB, and get a
passthrough
  -- CLOB handle
  EXECUTE IMMEDIATE 'INSERT INTO ora_table VALUES (2, EMPTY_CLOB())';
  EXECUTE IMMEDIATE 'SELECT c FROM ora_table WHERE i = 2 FOR UPDATE'
                    INTO passthru_clob ;

  -- Copy a TimesTen CLOB to a passthrough CLOB on Oracle Database
  DBMS_LOB.COPY(passthru_clob, tt_clob, clob_length, 1, 1);

  -- On Oracle Database: display the modified passthrough CLOB
  DBMS_LOB.READ(passthru_clob, clob_length, 1, clob_buffer);
  DBMS_OUTPUT.PUT_LINE(clob_buffer);

  COMMIT;
  EXECUTE IMMEDIATE 'call ttoptsetflag(''passthrough'', 0)';
END;
```

Copy CLOBs Using INSERT and UPDATE Statements

A passthrough LOB from Oracle Database can be bound to an `INSERT` or `UPDATE` statement executed against a table in TimesTen. You can copy a passthrough LOB to a TimesTen LOB in this way. Similarly, a TimesTen LOB can be bound to a passthrough `INSERT` or `UPDATE` statement executed against a table in Oracle Database. You can copy a TimesTen LOB to a passthrough LOB in this way.

This example shows both of these scenarios.

```
autocommit 0;
passthrough 0;
DROP TABLE tt_table; CREATE TABLE tt_table (i INT, c CLOB); COMMIT;
```

```
passthrough 3;
DROP TABLE ora_table; CREATE TABLE ora_table (i INT, c CLOB); COMMIT;
passthrough 0;
set serveroutput on;

DECLARE
  passthru_clob CLOB;
  tt_clob CLOB;
  clob_length BINARY_INTEGER;
  clob_buffer VARCHAR2(80);

BEGIN
  EXECUTE IMMEDIATE 'call ttoptsetflag(''passthrough'', 1)';

  -- Note that in PL/SQL, passthrough statements must be executed as
  -- dynamic SQL, and SELECT INTO must be used to assign a passthrough LOB.

  -- 1. A TimesTen CLOB is updated with a passthrough CLOB on Oracle Database
  -- On TimesTen: insert a row with a NULL CLOB value
  INSERT INTO tt_table VALUES (1, NULL);

  -- On Oracle Database: insert a row with an empty CLOB, get a passthrough
CLOB
  -- handle
  EXECUTE IMMEDIATE 'INSERT INTO ora_table
                     VALUES (1, ''Copy from Oracle Database to TimesTen'')';
  EXECUTE IMMEDIATE 'SELECT c FROM ora_table WHERE i = 1' INTO
passthru_clob ;

  -- On TimesTen: update the TimesTen CLOB with the passthrough CLOB
  UPDATE tt_table SET c = passthru_clob where i = 1;

  -- On TimesTen: display the modified TimesTen CLOB
  SELECT c INTO tt_clob FROM tt_table WHERE i = 1;
  clob_length := DBMS_LOB.GETLENGTH(tt_clob);
  DBMS_LOB.READ(tt_clob, clob_length, 1, clob_buffer);
  DBMS_OUTPUT.PUT_LINE(clob_buffer);

  -- 2. A passthrough table on Oracle Database is inserted with a TimesTen
CLOB
  -- On TimesTen: insert a row with a CLOB value, and get a TimesTen CLOB
handle
  INSERT INTO tt_table VALUES (2, 'Copy from TimesTen to Oracle Database.')
                       RETURNING c INTO tt_clob;

  -- On Oracle Database: insert a row on Oracle Database with the TimesTen
CLOB
  EXECUTE IMMEDIATE 'INSERT INTO ora_table VALUES (2, :1)' USING tt_clob;

  -- On Oracle Database: display the modified passthrough CLOB
  EXECUTE IMMEDIATE 'SELECT c FROM ora_table WHERE i = 2' INTO
passthru_clob;
  clob_length := DBMS_LOB.GETLENGTH(passthru_clob);
  DBMS_LOB.READ(passthru_clob, clob_length, 1, clob_buffer);
  DBMS_OUTPUT.PUT_LINE(clob_buffer);
```

```
  COMMIT;
  EXECUTE IMMEDIATE 'call ttoptsetflag(''passthrough'', 0)';
END;
```

CREATETEMPORARY Procedures

This procedure creates a temporary BLOB, CLOB, or NCLOB in the temporary data partition.

Use FREETEMPORARY Procedures when you are finished using temporary LOBs.

> **Tip:**
>
> In TimesTen, creation of a temporary LOB results in creation of a database transaction if one is not already in progress. You must execute a commit or rollback to close the transaction.

Syntax

```
DBMS_LOB.CREATETEMPORARY (
   lob_loc IN OUT NOCOPY BLOB,
   cache   IN            BOOLEAN,
   dur     IN            BINARY_INTEGER := DBMS_LOB.SESSION);

DBMS_LOB.CREATETEMPORARY (
   lob_loc IN OUT NOCOPY CLOB CHARACTER SET ANY_CS,
   cache   IN            BOOLEAN,
   dur     IN            BINARY_INTEGER := DBMS_LOB.SESSION);
```

Parameters

Table 2-17 CREATETEMPORARY Procedure Parameters

Parameter	Description
lob_loc	Locator for the temporary LOB It is permissible to specify an `NCLOB` locator instead of a `CLOB` locator. The appropriate character set is used.
cache	Flag indicating whether the LOB should be read into buffer cache
dur	One of two predefined duration values—`SESSION` or `CALL`—that specifies a hint as to when the temporary LOB is cleaned up **Note**: Either setting is permitted, but in TimesTen the duration of a LOB locator does not extend past the end of the transaction.

Usage Notes

- `CREATETEMPORARY` cannot be used to create a temporary passthrough LOB.

ERASE Procedures

This procedure erases all or part of a LOB.

> **Note:**
>
> Also see TRIM Procedures.

Syntax

```
DBMS_LOB.ERASE (
   lob_loc           IN OUT   NOCOPY   BLOB,
   amount            IN OUT   NOCOPY   INTEGER,
   offset            IN                INTEGER := 1);

DBMS_LOB.ERASE (
   lob_loc           IN OUT   NOCOPY   CLOB CHARACTER SET ANY_CS,
   amount            IN OUT   NOCOPY   INTEGER,
   offset            IN                INTEGER := 1);
```

Parameters

Table 2-18 ERASE Procedure Parameters

Parameter	Description
lob_loc	Locator for the LOB
amount	(IN) Number of bytes (for BLOBs) or characters (for CLOBs or NCLOBs) to be erased (OUT) Number of bytes or characters actually erased
offset	Absolute offset (starting from 1) from the beginning of the LOB, in bytes (for BLOBs) or characters (for CLOBs or NCLOBs)

Usage Notes

- When data is erased from the middle of a LOB, zero-byte fillers (for BLOBs) or spaces (for CLOBs or NCLOBs) are written.
- The actual number of bytes or characters erased can differ from the number you specified in the *amount* parameter if the end of the LOB data is reached first. The actual number of characters or bytes erased is returned in the *amount* parameter.
- It is recommended that you enclose write operations to the LOB with OPEN and CLOSE calls, but not mandatory. However, if you opened the LOB before performing the operation, you must close it before you commit or roll back the transaction.

> **Note:**
>
> The length of the LOB does not decrease when a section of the LOB is erased. To decrease the length of a LOB, see TRIM Procedures.

Exceptions

Maximum LOB size is `BLOBMAXSIZE` for a BLOB or `CLOBMAXSIZE` for a CLOB.

Table 2-19 ERASE Procedure Exceptions

Exception	Description
`VALUE_ERROR`	Any input parameter is `NULL`.
`INVALID_ARGVAL`	Any of the following is true: *`amount`* < 1 or *`amount`* > maximum LOB size *`offset`* < 1 or *`offset`* > maximum LOB size
`QUERY_WRITE`	Cannot perform a LOB write inside a query. (This is not applicable for TimesTen.)
`BUFFERING_ENABLED`	Cannot perform operation if LOB buffering is enabled on the LOB.

FREETEMPORARY Procedures

This procedure frees a temporary BLOB, CLOB, or NCLOB in the temporary data partition.

Also refer to CREATETEMPORARY Procedures.

Syntax

```
DBMS_LOB.FREETEMPORARY (
   lob_loc  IN OUT  NOCOPY BLOB);

DBMS_LOB.FREETEMPORARY (
   lob_loc  IN OUT  NOCOPY CLOB CHARACTER SET ANY_CS);
```

Parameters

Table 2-20 FREETEMPORARY Procedure Parameters

Parameter	Description
`lob_loc`	Locator for the LOB

Usage Notes

- After the call to `FREETEMPORARY`, the LOB locator that was freed is marked as invalid.

- If an invalid LOB locator is assigned to another LOB locator through an assignment operation in PL/SQL, then the target of the assignment is also freed and marked as invalid.
- `CREATETEMPORARY` cannot be used to create a temporary passthrough LOB; however, if one is created using some other mechanism, such as SQL, `ISTEMPORARY` and `FREETEMPORARY` can be used on that LOB.

GETCHUNKSIZE Functions

In TimesTen, this function is not supported and simply returns the value 32K for interoperability. This value is not relevant for any performance tuning for a TimesTen application.

Refer to GETCHUNKSIZE Functions in *Oracle Database PL/SQL Packages and Types Reference* if you are interested in Oracle Database functionality.

Syntax

```
DBMS_LOB.GETCHUNKSIZE (
   lob_loc IN BLOB)
  RETURN INTEGER;

DBMS_LOB.GETCHUNKSIZE (
   lob_loc IN CLOB CHARACTER SET ANY_CS)
  RETURN INTEGER;
```

Parameters

Table 2-21 GETCHUNKSIZE Function Parameters

Parameter	Description
lob_loc	Locator for the LOB

Return Values

Returns the value 32K, but applications should not rely on this number for performance tuning.

Exceptions

Table 2-22 GETCHUNKSIZE Procedure Exceptions

Exception	Description
BUFFERING_ENABLED	Cannot perform operation if LOB buffering is enabled on the LOB.

GETLENGTH Functions

This function returns the length of the specified LOB in bytes (for BLOBs) or characters (for CLOBs or NCLOBs).

Syntax

```
DBMS_LOB.GETLENGTH (
   lob_loc    IN  BLOB)
  RETURN INTEGER;

DBMS_LOB.GETLENGTH (
   lob_loc    IN  CLOB   CHARACTER SET ANY_CS)
  RETURN INTEGER;
```

Parameters

Table 2-23 GETLENGTH Function Parameter

Parameter	Description
lob_loc	Locator for the LOB

Return Values

Returns an `INTEGER` value for the length of the LOB in bytes or characters. `NULL` is returned if the value of the input LOB or *lob_loc* is `NULL`.

Usage Notes

- Any zero-byte or space filler in the LOB caused by previous `ERASE` or `WRITE` operations is included in the length count. The length of an empty LOB is 0 (zero).

Exceptions

Table 2-24 GETLENGTH Procedure Exceptions

Exception	Description
`BUFFERING_ENABLED`	Cannot perform operation if LOB buffering is enabled on the LOB.

Examples

The following example shows use of the `GETLENGTH` function.

```
create table t1 (a int, b blob, c clob);

insert into t1(a,b,c) values(1, 0x123451234554321, 'abcde');
1 row inserted.

commit;

declare
```

```
  myblob blob;
  i integer;
begin
  myblob := empty_blob();
  i := dbms_lob.getlength(myblob);
  dbms_output.put_line('Length of BLOB before SELECT: ' || i);
  select b into myblob from t1 where a=1;
  i := dbms_lob.getlength(myblob);
  dbms_output.put_line('Length of BLOB after SELECT: ' || i);
end;

Length of BLOB before SELECT: 0
Length of BLOB after SELECT: 8

PL/SQL procedure successfully completed.
```

(Output is shown after running the commands from a SQL script.)

GET_STORAGE_LIMIT Functions

This function returns the storage limit, in bytes, for the type of specified LOB.

Syntax

```
DBMS_LOB.GET_STORAGE_LIMIT (
   lob_loc  IN CLOB CHARACTER SET ANY_CS)
 RETURN INTEGER;

DBMS_LOB.GET_STORAGE_LIMIT (
   lob_loc  IN BLOB)
 RETURN INTEGER;
```

Parameters

Table 2-25 GET_STORAGE_LIMIT Function Parameters

Parameter	Description
lob_loc	Locator for the LOB

Return Value

In TimesTen, the value returned is simply the maximum storage space, in bytes, for the type of specified LOB. That is 16777216 (16 MB) for a BLOB or 4194304 (4 MB) for a CLOB or NCLOB.

INSTR Functions

This function returns the matching position of the *n*th occurrence of a specified pattern in a specified LOB, starting from a specified offset.

> **Note:**
>
> Also see SUBSTR Functions.

Syntax

```
DBMS_LOB.INSTR (
   lob_loc    IN   BLOB,
   pattern    IN   RAW,
   offset     IN   INTEGER := 1,
   nth        IN   INTEGER := 1)
  RETURN INTEGER;

DBMS_LOB.INSTR (
   lob_loc    IN   CLOB      CHARACTER SET ANY_CS,
   pattern    IN   VARCHAR2  CHARACTER SET lob_loc%CHARSET,
   offset     IN   INTEGER := 1,
   nth        IN   INTEGER := 1)
  RETURN INTEGER;
```

Parameters

Table 2-26 INSTR Function Parameters

Parameter	Description
lob_loc	Locator for the LOB
pattern	Pattern to be tested for The pattern is in `RAW` bytes for BLOBs, or a character string (`VARCHAR2`) for CLOBs or NCLOBs. The maximum size of the pattern is 16383 bytes.
offset	Absolute offset in bytes (for BLOBs) or characters (for CLOBs or NCLOBs), starting from 1, at which the pattern-matching is to start
nth	Occurrence number of the pattern in the LOB, starting from 1

Return Values

The function returns one of the following:

- An `INTEGER` value for the offset of the beginning of the matched pattern, in bytes (for BLOBs) or characters (for CLOBs or NCLOBs)
- 0 (zero) if the pattern is not found

- NULL if any of the input parameters is NULL or invalid or any of the following is true:
  - *offset* < 1 or *offset* > maximum LOB size
  - *nth* < 1 or *nth* > maximum LOB size

  Where maximum LOB size is BLOBMAXSIZE for a BLOB or CLOBMAXSIZE for a CLOB.

Usage Notes

- For a CLOB or NCLOB, the form of the VARCHAR2 buffer for the *pattern* parameter must be appropriate for the type of LOB. If the specified LOB is of type NCLOB, the pattern must contain NCHAR data. If the specified LOB is of type CLOB, the pattern must contain CHAR data.
- Operations that accept RAW or VARCHAR2 parameters for pattern matching, such as INSTR, do not support regular expressions or special matching characters (as with SQL LIKE) in the pattern parameter or substrings.

ISOPEN Functions

This function checks to see if a LOB was already opened using the input locator.

Syntax

```
DBMS_LOB.ISOPEN (
   lob_loc IN BLOB)
  RETURN INTEGER;

DBMS_LOB.ISOPEN (
   lob_loc IN CLOB CHARACTER SET ANY_CS)
  RETURN INTEGER;
```

Parameters

Table 2-27 ISOPEN Function Parameters

Parameter	Description
lob_loc	Locator for the LOB

Return Values

The return value is 1 if the LOB is open, or 0 (zero) if not.

Usage Notes

- The "open" status is associated with the LOB, not with the locator. If any locator is used in opening the LOB, then any other locator for the LOB would also see it as open.
- ISOPEN requires a round-trip, because it must check the state on the server to see if the LOB is open.

ISTEMPORARY Functions

This function determines whether a LOB is temporary.

Syntax

```
DBMS_LOB.ISTEMPORARY (
   lob_loc IN BLOB)
  RETURN INTEGER;

DBMS_LOB.ISTEMPORARY (
   lob_loc IN CLOB CHARACTER SET ANY_CS)
  RETURN INTEGER;
```

Parameters

Table 2-28 ISTEMPORARY Procedure Parameters

Parameter	Description
lob_loc	Locator for the LOB

Return Values

The return value is 1 if the LOB exists and is temporary, 0 (zero) if the LOB does not exist or is not temporary, or `NULL` if the given locator value is `NULL`.

Usage Notes

- When you free a temporary LOB with `FREETEMPORARY`, the LOB locator is not set to `NULL`. Consequently, `ISTEMPORARY` returns 0 (zero) for a locator that has been freed but not explicitly reset to `NULL`.
- `CREATETEMPORARY` cannot be used to create a temporary passthrough LOB; however, if one is created using some other mechanism, such as SQL, `ISTEMPORARY` and `FREETEMPORARY` can be used on that LOB.

OPEN Procedures

This procedure opens a LOB in the indicated mode, read-only or read/write.

Syntax

```
DBMS_LOB.OPEN (
   lob_loc   IN OUT NOCOPY BLOB,
   open_mode IN            BINARY_INTEGER);

DBMS_LOB.OPEN (
   lob_loc   IN OUT NOCOPY CLOB CHARACTER SET ANY_CS,
   open_mode IN            BINARY_INTEGER);
```

Parameters

Table 2-29 OPEN Procedure Parameters

Parameter	Description
lob_loc	Locator for the LOB
open_mode	Mode in which to open, either LOB_READONLY or LOB_READWRITE

Usage Notes

- An error is returned if you try to write to a LOB that was opened as read-only.
- OPEN requires a round-trip to the server and causes execution of other code that relies on the OPEN call.
- It is not mandatory that you wrap LOB operations inside OPEN and CLOSE calls. However, if you open a LOB, you must close it before you commit or roll back the transaction.
- It is an error to commit the transaction before closing all LOBs that were opened in the transaction. When the error is returned, the "open" status of the open LOBs is discarded, but the transaction is successfully committed. Hence, all the changes made to both LOB and non-LOB data in the transaction are committed.

READ Procedures

This procedure reads part of a LOB, starting from a specified absolute offset from the beginning of the LOB, and returns the specified number of bytes (for BLOBs) or characters (for CLOBs or NCLOBs) into the *buffer* parameter.

Syntax

```
DBMS_LOB.READ (
   lob_loc   IN             BLOB,
   amount    IN OUT  NOCOPY INTEGER,
   offset    IN             INTEGER,
   buffer    OUT            RAW);

DBMS_LOB.READ (
   lob_loc   IN             CLOB CHARACTER SET ANY_CS,
   amount    IN OUT  NOCOPY INTEGER,
   offset    IN             INTEGER,
   buffer    OUT            VARCHAR2 CHARACTER SET lob_loc%CHARSET);
```

Parameters

Table 2-30 READ Procedure Parameters

Parameter	Description
lob_loc	Locator for the LOB

Table 2-30 (Cont.) READ Procedure Parameters

Parameter	Description
amount	(IN) Number of bytes (for BLOBs) or characters (for CLOBs or NCLOBs) to read (OUT) Number of bytes or characters actually read
offset	Offset in bytes (for BLOBs) or characters (for CLOBs or NCLOBs) from the start of the LOB (starting from 1)
buffer	Output buffer from the read operation

Usage Notes

- If the input *offset* points past the end of the LOB, then *amount* is set to 0 (zero) and a NO_DATA_FOUND exception is raised.
- For a CLOB or NCLOB, the form of the VARCHAR2 buffer for the *buffer* parameter must be appropriate for the type of LOB. If the specified LOB is of type NCLOB, the buffer must contain NCHAR data. If the specified LOB is of type CLOB, the buffer must contain CHAR data.
- When calling READ from a client, the returned buffer contains data in the client character set. The database converts the LOB value from the server character set to the client character set before it returns the buffer to the user.
- READ gets the LOB, if necessary, before the read.

Exceptions

Maximum LOB size is BLOBMAXSIZE for a BLOB or CLOBMAXSIZE for a CLOB.

Table 2-31 READ Procedure Exceptions

Exception	Description
VALUE_ERROR	Any of *lob_loc*, *amount*, or *offset* is NULL.
INVALID_ARGVAL	Any of the following is true: *amount* < 1 or *amount* > 32767 bytes (or the character equivalent) or the capacity of *buffer* *offset* < 1 or *offset* > maximum LOB size
NO_DATA_FOUND	The end of the LOB is reached and there are no more bytes or characters to read from the LOB. The *amount* parameter has a value of 0 (zero).

SUBSTR Functions

This function returns a specified number of bytes (for a BLOB) or characters (for a CLOB or NCLOB), starting at a specified offset from the beginning of a specified LOB.

> **Note:**
>
> Also see INSTR Functions and READ Procedures.

Syntax

```
DBMS_LOB.SUBSTR (
   lob_loc     IN    BLOB,
   amount      IN    INTEGER := 32767,
   offset      IN    INTEGER := 1)
  RETURN RAW;

DBMS_LOB.SUBSTR (
   lob_loc     IN    CLOB   CHARACTER SET ANY_CS,
   amount      IN    INTEGER := 32767,
   offset      IN    INTEGER := 1)
  RETURN VARCHAR2 CHARACTER SET lob_loc%CHARSET;
```

Parameters

Table 2-32 SUBSTR Function Parameters

Parameter	Description
lob_loc	Locator for the LOB
amount	Number of bytes (for BLOBs) or characters (for CLOBs or NCLOBs) to read
offset	Offset in bytes (for BLOBs) or characters (for CLOBs or NCLOBs) from the start of the LOB (starting from 1)

Return Values

Returns one of the following:

- `RAW` bytes from a BLOB
- `VARCHAR2` characters from a CLOB or NCLOB
- `NULL` if any input parameter is `NULL` or any of the following is true:
  - *amount* < 1 or *amount* > 32767 bytes (or the character equivalent)
  - *offset* < 1 or *offset* > maximum LOB size

  Where maximum LOB size is `BLOBMAXSIZE` for a BLOB or `CLOBMAXSIZE` for a CLOB.

Usage Notes

- For fixed-width *n*-byte CLOBs or NCLOBs, if the input amount for SUBSTR is greater than (32767/*n*), then SUBSTR returns a character buffer of length (32767/*n*) or the length of the CLOB, whichever is less. For CLOBs in a varying-width character set, *n* is the maximum byte-width used for characters in the CLOB.
- For a CLOB or NCLOB, the form of the VARCHAR2 return buffer must be appropriate for the type of LOB. If the specified LOB is of type NCLOB, the buffer must contain NCHAR data. If the specified LOB is of type CLOB, the buffer must contain CHAR data.
- When calling SUBSTR from a client, the returned buffer contains data in the client character set. The database converts the LOB value from the server character set to the client character set before it returns the buffer to the user.
- SUBSTR returns 8191 or more characters based on the characters stored in the LOB. If all characters are not returned because the character byte size exceeds the available buffer, the user should either call SUBSTR with a new offset to read the remaining characters, or call the subprogram in a loop until all the data is extracted.
- SUBSTR get s the LOB, if necessary, before reading.

TRIM Procedures

This procedure trims a LOB to the length you specify in the *newlen* parameter. Specify the new desired data length in bytes for BLOBs or characters for CLOBs or NCLOBs.

> **Note:**
>
> Also see ERASE Procedures and WRITEAPPEND Procedures.

Syntax

```
DBMS_LOB.TRIM (
   lob_loc         IN OUT  NOCOPY BLOB,
   newlen          IN              INTEGER);

DBMS_LOB.TRIM (
   lob_loc         IN OUT  NOCOPY CLOB CHARACTER SET ANY_CS,
   newlen          IN              INTEGER);
```

Parameters

Table 2-33 TRIM Procedure Parameters

Parameter	Description
lob_loc	Locator for the LOB
newlen	Desired trimmed length of the LOB value, in bytes (for BLOBs) or characters (for CLOBs or NCLOBs)

Usage Notes

- If you attempt to trim an empty LOB, no action is taken and `TRIM` returns no error.
- If the new length that you specify in *`newlen`* is greater than the size of the LOB, an exception is raised.
- It is recommended that you enclose write operations to the LOB with `OPEN` and `CLOSE` calls, but not mandatory. However, if you opened the LOB before performing the operation, you must close it before you commit or roll back the transaction.
- `TRIM` gets the LOB, if necessary, before altering the length of the LOB, unless the new length specified is 0 (zero).

Exceptions

Maximum LOB size is `BLOBMAXSIZE` for a BLOB or `CLOBMAXSIZE` for a CLOB.

Table 2-34 TRIM Procedure Exceptions

Exception	Description
`VALUE_ERROR`	The *`lob_loc`* value is `NULL`.
`INVALID_ARGVAL`	Either of the following is true: *`newlen`* < 0 or *`newlen`* > maximum LOB size
`QUERY_WRITE`	Cannot perform a LOB write inside a query. (This is not applicable for TimesTen.)
`BUFFERING_ENABLED`	Cannot perform operation if LOB buffering enabled is enabled on the LOB.

WRITE Procedures

This procedure writes a specified amount of data into a LOB, starting from a specified absolute offset from the beginning of the LOB. The data is written from the *`buffer`* parameter.

`WRITE` replaces (overwrites) any data that already exists in the LOB from the offset through the length you specify.

> **Note:**
>
> Also see COPY Procedures and WRITEAPPEND Procedures.

Syntax

```
DBMS_LOB.WRITE (
   lob_loc  IN OUT NOCOPY  BLOB,
   amount   IN             INTEGER,
   offset   IN             INTEGER,
   buffer   IN             RAW);

DBMS_LOB.WRITE (
```

```
lob_loc  IN OUT  NOCOPY CLOB   CHARACTER SET ANY_CS,
amount   IN             INTEGER,
offset   IN             INTEGER,
buffer   IN             VARCHAR2 CHARACTER SET lob_loc%CHARSET);
```

Parameters

Table 2-35 WRITE Procedure Parameters

Parameter	Description
lob_loc	Locator for the LOB
amount	Number of bytes (for BLOBs) or characters (for CLOBs or NCLOBs) to write
offset	Offset in bytes (for BLOBs) or characters (for CLOBs or NCLOBs) from the start of the LOB for the write operation (starting from 1)
buffer	Input buffer with data for the write

Usage Notes

- There is an error if the specified amount is more than the data in the buffer. If the input amount is less than the data in the buffer, then only *amount* bytes or characters from the buffer are written to the LOB. If the offset you specify is beyond the end of the data currently in the LOB, then zero-byte fillers (for BLOBs) or spaces (for CLOBs or NCLOBs) are inserted into the LOB to reach the offset.
- For a CLOB or NCLOB, the form of the `VARCHAR2` buffer for the *buffer* parameter must be appropriate for the type of LOB. If the specified LOB is of type `NCLOB`, the buffer must contain `NCHAR` data. If the specified LOB is of type `CLOB`, the buffer must contain `CHAR` data.
- When calling `WRITE` from a client, the buffer must contain data in the client character set. The database converts the client-side buffer to the server character set before it writes the buffer data to the LOB.
- It is recommended that you enclose write operations to the LOB with `OPEN` and `CLOSE` calls, but not mandatory. However, if you opened the LOB before performing the operation, you must close it before you commit or roll back the transaction.
- `WRITE` gets the LOB, if necessary, before writing to it, unless the write is specified to overwrite the entire LOB.

Exceptions

Maximum LOB size is `BLOBMAXSIZE` for a BLOB or `CLOBMAXSIZE` for a CLOB.

Table 2-36 WRITE Procedure Exceptions

Exception	Description
`VALUE_ERROR`	Any of *lob_loc*, *amount*, or *offset* is `NULL`, out of range, or invalid.

Table 2-36 (Cont.) WRITE Procedure Exceptions

Exception	Description
INVALID_ARGVAL	Any of the following is true: *amount* < 1 or *amount* > 32767 bytes (or the character equivalent) or capacity of *buffer* *offset* < 1 or *offset* > maximum LOB size
QUERY_WRITE	Cannot perform a LOB write inside a query. (This is not applicable for TimesTen.)
BUFFERING_ENABLED	Cannot perform operation if LOB buffering is enabled on the LOB.

WRITEAPPEND Procedures

This procedure appends a specified amount of data to the end of a LOB. The data is written from the *buffer* parameter. (Do not confuse this with the APPEND procedure.)

> **Note:**
>
> Also see APPEND Procedures, COPY Procedures, and WRITE Procedures.

Syntax

```
DBMS_LOB.WRITEAPPEND (
   lob_loc IN OUT NOCOPY BLOB,
   amount  IN            INTEGER,
   buffer  IN            RAW);

DBMS_LOB.WRITEAPPEND (
   lob_loc IN OUT NOCOPY CLOB CHARACTER SET ANY_CS,
   amount  IN            INTEGER,
   buffer  IN            VARCHAR2 CHARACTER SET lob_loc%CHARSET);
```

Parameters

Table 2-37 WRITEAPPEND Procedure Parameters

Parameter	Description
lob_loc	Locator for the LOB
amount	Number of bytes (for BLOBs) or characters (for CLOBs or NCLOBs) to write
buffer	Input buffer with data for the write

Usage Notes

- There is an error if the input amount is more than the data in the buffer. If the input amount is less than the data in the buffer, then only the *amount* bytes or characters from the buffer are appended to the LOB.
- For a CLOB or NCLOB, the form of the VARCHAR2 buffer for the *buffer* parameter must be appropriate for the type of LOB. If the specified LOB is of type NCLOB, the buffer must contain NCHAR data. If the specified LOB is of type CLOB, the buffer must contain CHAR data.
- When calling WRITEAPPEND from a client, the buffer must contain data in the client character set. The database converts the client-side buffer to the server character set before it writes the buffer data to the LOB.
- It is recommended that you enclose write operations to the LOB with OPEN and CLOSE calls, but not mandatory. However, if you opened the LOB before performing the operation, you must close it before you commit or roll back the transaction.
- WRITEAPPEND gets the LOB, if necessary, before appending to it.

Exceptions

Table 2-38 WRITEAPPEND Procedure Exceptions

Exception	Description
VALUE_ERROR	Any of *lob_loc*, *amount*, or *offset* is null, out of range, or invalid.
INVALID_ARGVAL	Any of the following is true: *amount* < 1 or *amount* > 32767 bytes (or the character equivalent) or capacity of *buffer*
QUERY_WRITE	Cannot perform a LOB write inside a query. (This is not applicable for TimesTen.)
BUFFERING_ENABLED	Cannot perform operation if LOB buffering is enabled on the LOB.

3
DBMS_LOCK

The DBMS_LOCK package provides an interface to Lock Management services. TimesTen supports only the SLEEP subprogram.

This chapter contains the following topics:

- Using DBMS_LOCK
- DBMS_LOCK Subprograms

Using DBMS_LOCK

TimesTen currently implements only the SLEEP subprogram, used to suspend the session for a specified duration.

DBMS_LOCK Subprograms

TimesTen supports only the SLEEP subprogram. Table 3-1 summarizes that subprogram, followed by a full description.

Table 3-1 *DBMS_LOCK Package Subprograms*

Subprogram	Description
SLEEP Procedure	Suspends the session for a specified duration.

SLEEP Procedure

This procedure suspends the session for a specified duration.

Syntax

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

Parameters

Table 3-2 SLEEP Procedure Parameters

Parameter	Description
seconds	Amount of time, in seconds, to suspend the session, where the smallest increment is a hundredth of a second

Usage Notes

- The actual sleep time may be somewhat longer than specified, depending on system activity.
- If the `PLSQL_TIMEOUT` general connection attribute is set to a positive value that is less than this sleep time, the timeout takes effect first. Be sure that either the sleep value is less than the timeout value, or `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);
```

4
DBMS_OUTPUT

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

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

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 can 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:

```
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 by an external tool like `ttIsql` to control whether 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.

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.

Exceptions

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

Table 4-1 DBMS_OUTPUT Exceptions

Exception	Description
`ORU-10027`	Buffer overflow
`ORU-10028`	Line length overflow

Examples

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:

```
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;
   /* Debug line */
   DBMS_OUTPUT.PUT_LINE('Total wages = ' ||
   TO_CHAR(total_wages));
   RETURN total_wages;
END;
```

Assume the user executes the following statements in `ttIsql`:

```
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 = 19000

PL/SQL procedure successfully executed.
```

Data Structures

The `DBMS_OUTPUT` package declares two table types for use with the GET_LINES Procedure.

> **Note:**
>
> - 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

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

DBMS_OUTPUT Subprograms

Table 4-2 summarizes the `DBMS_OUTPUT` subprograms, followed by a full description of each subprogram.

Table 4-2 DBMS_OUTPUT Package Subprograms

Subprogram	Description
DISABLE Procedure	Disables message output.
ENABLE Procedure	Enables message output.
GET_LINE Procedure	Retrieves one line from buffer.
GET_LINES Procedure	Retrieves an array of lines from buffer.
NEW_LINE Procedure	Terminates a line created with `PUT`.
PUT Procedure	Places a partial line in the buffer.
PUT_LINE Procedure	Places a line in the buffer.

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;
```

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

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

Parameters

Table 4-3 ENABLE Procedure Parameters

Parameter	Description
buffer_size	Upper limit, in bytes, for the amount of buffered information Setting *buffer_size* to `NULL` specifies that there should be no limit.

Usage Notes

- It is not necessary to call this procedure when you use `SET SERVEROUTPUT ON` from `ttIsql`. It is called automatically (with `NULL` value for *buffer_size* in the current release).
- 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 20000 for backward compatibility with earlier database versions that did not support unlimited buffering.

GET_LINE Procedure

This procedure retrieves a single line of buffered information.

Syntax

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

Parameters

Table 4-4 GET_LINE Procedure Parameters

Parameter	Description
line	A single line of buffered information, excluding a final newline character You should declare this parameter as `VARCHAR2(32767)` to avoid the risk of "`ORA-06502: PL/SQL: numeric or value error: character string buffer too small`".

Table 4-4 (Cont.) GET_LINE Procedure Parameters

Parameter	Description
status	Call status 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.

Usage Notes

- You can choose to retrieve from the buffer a single line or an array of lines. Call `GET_LINE` to retrieve a single line of buffered information. To reduce the number of calls to the server, call `GET_LINES` 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.DBMSOUTPUT_LINESARRAY,
   numlines    IN OUT INTEGER);
```

Parameters

Table 4-5 GET_LINES Procedure Parameters

Parameter	Description
lines	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.
numlines	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.

Usage Notes

- You can choose to retrieve from the buffer a single line or an array of lines. Call `GET_LINE` to retrieve a single line of buffered information. To reduce the number of trips to the server, call `GET_LINES` 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 `GET_LINE` or `GET_LINES` is called, 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 to `NEW_LINE` 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` version that takes a `NUMBER` input is obsolete. It is supported for legacy reasons only.

Syntax

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

Parameters

Table 4-6 PUT Procedure Parameters

Parameter	Description
a	Item to buffer

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, 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.

Exceptions

Table 4-7 PUT Procedure Exceptions

Exception	Description
ORA-20000, ORU-10027	Buffer overflow, according to the *buffer_size* limit specified in the ENABLE Procedure call
ORA-20000, ORU-10028	Line length overflow, limit of 32767 bytes for each line

PUT_LINE Procedure

This procedure places a line in the buffer.

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

Syntax

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

Parameters

Table 4-8 PUT_LINE Procedure Parameters

Parameter	Description
a	Item to buffer

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.

Exceptions

Table 4-9 PUT_LINE Procedure Exceptions

Exception	Description
`ORA-20000, ORU-10027`	Buffer overflow, according to the *`buffer_size`* limit specified in the ENABLE Procedure call
`ORA-20000, ORU-10028`	Line length overflow, limit of 32767 bytes for each line

5
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
  - Operational notes
- Data Structures
  - Table types
- DBMS_PREPROCESSOR Subprograms

Using DBMS_PREPROCESSOR

- Overview
- Operational 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 that contains the source text of an arbitrary PL/SQL compilation unit
3. Subprograms that take a `VARCHAR2` associative array (index-by table) that 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 compilation unit. A typical usage is to pass the source text of an anonymous block and generate its post-processed source text in the current user schema. The third style can be useful when the source text exceeds the `VARCHAR2` length limit.

Operational Notes

- For subprograms of the first style, the status of the stored PL/SQL unit is not required 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 also appears at the end of the post-processed source text. In some cases, the preprocessing can terminate because of errors. When this happens, the post-processed source text appears to be incomplete and the associated error message can help 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 is not 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 a GET_POST_PROCESSED_SOURCE Function or a PRINT_POST_PROCESSED_SOURCE Procedure, the exception `WRAPPED_INPUT` is raised.

Data Structures

The `DBMS_PREPROCESSOR` package defines a table type.

> **Note:**
>
> - 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 32 KB.

Syntax

```
TYPE source_lines_t IS
    TABLE OF VARCHAR2(32767) INDEX BY BINARY_INTEGER;
```

DBMS_PREPROCESSOR Subprograms

Table 5-1 summarizes the `DBMS_PREPROCESSOR` subprograms, followed by a full description of each subprogram.

Table 5-1 DBMS_PREPROCESSOR Package Subprograms

Subprogram	Description
GET_POST_PROCESSED_SOURCE Function	Returns the post-processed source text.
PRINT_POST_PROCESSED_SOURCE Procedure	Prints post-processed source text.

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:

```
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:

```
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:

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

Parameters

Table 5-2 GET_POST_PROCESSED_SOURCE Function Parameters

Parameter	Description
object_type	One of `PACKAGE`, `PACKAGE BODY`, `PROCEDURE`, or `FUNCTION` (case sensitive)
schema_name	Schema name (case insensitive unless a quoted identifier is used) If `NULL`, use the current schema.
object_name	Name of the object (case insensitive unless a quoted identifier is used)
source	Source text of the compilation unit
source_lines_t	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.

Return Values

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 has blank lines.

Exceptions

Table 5-3 GET_POST_PROCESSED_SOURCE Function Exceptions

Exception	Description
ORA-24234	Insufficient privileges or non-existent object
ORA-24235	Bad value for object type (neither PACKAGE, PACKAGE BODY, PROCEDURE, nor FUNCTION)
ORA-24236	Empty source text
ORA-00931	Missing identifier The object_name value cannot be NULL.
ORA-06502	Numeric or value error: • Character string buffer is too small. • A line is too long (more than 32767 bytes).

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:

```
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:

```
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:

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

Parameters

Table 5-4 PRINT_POST_PROCESSED_SOURCE Procedure Parameters

Parameter	Description
object_type	One of `PACKAGE`, `PACKAGE BODY`, `PROCEDURE`, or `FUNCTION` (case sensitive)
schema_name	Schema name (case insensitive unless a quoted identifier is used) If `NULL`, use current schema.
object_name	Name of the object (case insensitive unless a quoted identifier is used)
source	Source text of the compilation unit
source_lines_t	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.

Usage Notes

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

Exceptions

Table 5-5 PRINT_POST_PROCESSED_SOURCE Procedure Exceptions

Exception	Description
`ORA-24234`	Insufficient privileges or non-existent object
`ORA-24235`	Bad value for object type (neither `PACKAGE`, `PACKAGE BODY`, `PROCEDURE`, nor `FUNCTION`)
`ORA-24236`	Empty source text
`ORA-00931`	Missing identifier The `object_name` value cannot be `NULL`.
`ORA-06502`	Numeric or value error: • Character string buffer is too small. • A line is too long (more than 32767 bytes).

6
DBMS_RANDOM

The DBMS_RANDOM package provides a built-in random number generator.

This chapter contains the following topics:

- Using DBMS_RANDOM
  - Operational notes
- DBMS_RANDOM Subprograms

> **Note:**
>
> DBMS_RANDOM is not intended for cryptography.

Using DBMS_RANDOM

- Operational Notes

Operational Notes

- The RANDOM function produces integers in the range [-2^^31, 2^^31).
- The VALUE function produces numbers in the range [0,1) with 38 digits of precision.

DBMS_RANDOM can be explicitly initialized but does not require initialization before a call to the random number generator. It automatically initializes 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 produces the same result 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 an overload of SEED. To produce different output for every run, simply omit the seed call. Then the system chooses a suitable seed for you.

DBMS_RANDOM Subprograms

Table 6-1 summarizes the DBMS_RANDOM subprograms, followed by a full description of each subprogram.

Table 6-1 *DBMS_RANDOM Package Subprograms*

Subprogram	Description
INITIALIZE Procedure	Initializes the package with a seed value.
NORMAL Function	Returns random numbers in a normal distribution.
RANDOM Function	Generates a random number.
SEED Procedure	Resets the seed.
STRING Function	Gets a random string.
TERMINATE Procedure	Terminates package.
VALUE Function	One version gets a random number greater than or equal to 0 and less than 1, with 38 digits to the right of the decimal point (38-digit precision). The other version gets a random Oracle Database number *x*, where *x* is greater than or equal to a specified lower limit and less than a specified higher limit.

Note:

- The INITIALIZE Procedure, RANDOM Function and TERMINATE Procedure are deprecated. They are included in this release for legacy reasons only.
- 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 is deprecated. Although currently supported, it should not be used. It initializes the random number generator.

Syntax

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

Parameters

Table 6-2 INITIALIZE Procedure Parameters

Parameter	Description
val	Seed number used to generate a random number

Usage Notes

This procedure is obsolete as it simply calls the SEED Procedure.

NORMAL Function

This function returns random numbers in a standard normal distribution.

Syntax

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

Return Value

The random number, a `NUMBER` value

RANDOM Function

This procedure is deprecated. Although currently supported, it should not be used. It generates and returns a random number.

Syntax

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

Return Value

A random `BINARY_INTEGER` value greater than or equal to `-power(2,31)` and less than `power(2,31)`

Usage Notes

See the NORMAL Function and the VALUE Function.

SEED Procedure

This procedure resets the seed used in generating a random number.

Syntax

```
DBMS_RANDOM.SEED (
   val  IN  BINARY_INTEGER);

DBMS_RANDOM.SEED (
   val  IN  VARCHAR2);
```

Parameters

Table 6-3 SEED Procedure Parameters

Parameter	Description
val	Seed number or string used to generate a random number

Usage Notes

The seed can be a string up to length 2000.

STRING Function

This function generates and returns a random string.

Syntax

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

Parameters

Table 6-4 STRING Function Parameters

Parameter	Description
opt	What the returning string looks like: • 'u', 'U' - Returning string is in uppercase alpha characters. • 'l', 'L' - Returning string is in lowercase alpha characters. • 'a', 'A' - Returning string is in mixed-case alpha characters. • 'x', 'X' - Returning string is in uppercase alpha-numeric characters. • 'p', 'P' - Returning string is in any printable characters. Otherwise the returning string is in uppercase alpha characters.
len	Length of the returned string

Return Value

A `VARCHAR2` value with the random string

TERMINATE Procedure

This procedure is deprecated. Although currently supported, it should not be used. It would be called when the user is finished with the package.

Syntax

```
DBMS_RANDOM.TERMINATE;
```

VALUE Function

One version returns 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). The other version returns a random Oracle Database `NUMBER` value `x`, where `x` is greater than or equal to the specified *`low`* value and less than the specified *`high`* value.

Syntax

```
DBMS_RANDOM.VALUE
  RETURN NUMBER;

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

Parameters

Table 6-5 VALUE Function Parameters

Parameter	Description
low	Lower limit of the range in which to generate a random number
high	Upper limit of the range in which to generate a random number

Return Value

A `NUMBER` value that is the generated random number

7

DBMS_SQL

The `DBMS_SQL` package provides an interface for using dynamic SQL to execute data manipulation language (DML) and data definition language (DDL) statements, execute PL/SQL anonymous blocks, and call PL/SQL stored procedures and functions.

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:

> **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*. You can also refer to EXECUTE IMMEDIATE Statement in *Oracle Database PL/SQL Language Reference*.

Using DBMS_SQL

Overview

TimesTen PL/SQL supports 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 functionality 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 (`EXECUTE IMMEDIATE`) is an alternative to `DBMS_SQL` that lets you place dynamic SQL statements, PL/SQL blocks, and PL/SQL procedure and function calls 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, such as there being 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, including the following:

- OCI binds by address, while the `DBMS_SQL` package binds 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 `NULL` is fully supported as a value of a PL/SQL variable.

Security Model

`DBMS_SQL` is owned by `SYS` and 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 (AUTHID Clause) in *Oracle TimesTen In-Memory Database Security Guide*.

Constants

The constants described in Table 7-1 are used with the *`language_flag`* parameter of the PARSE Procedures. For TimesTen, use `NATIVE`.

Table 7-1 DBMS_SQL Constants

Name	Type	Value	Description
V6	INTEGER	0	Specifies Oracle Database version 6 behavior (not applicable for TimesTen).
NATIVE	INTEGER	1	Specifies typical behavior for the database to which the program is connected.
V7	INTEGER	2	Specifies Oracle Database version 7 behavior (not applicable for TimesTen).

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

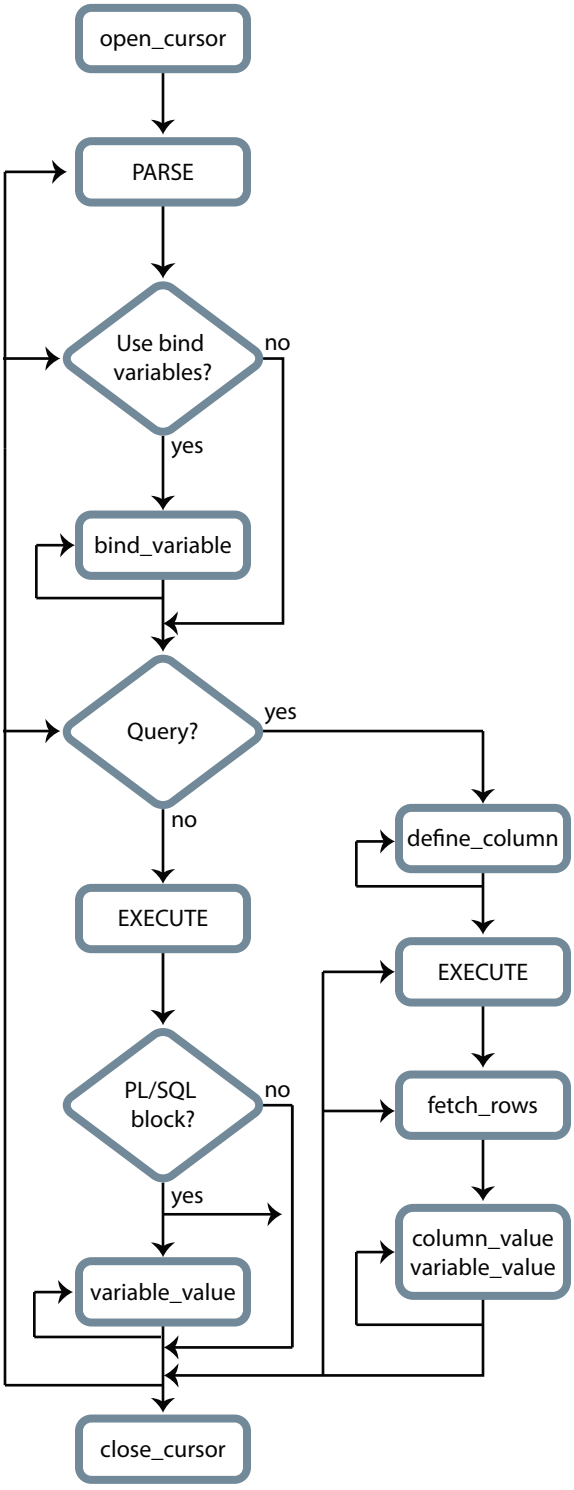
Every SQL statement must be parsed by calling the PARSE Procedures. Parsing the statement checks the statement syntax and associates it with the cursor in your program.

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 timeout can occur if you are 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 timeout occurs after a short time.

The execution flow of `DBMS_SQL` is shown in Figure 7-1 that follows.

Figure 7-1 DBMS_SQL Execution Flow



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 a bind procedure, either the BIND_ARRAY Procedure or the BIND_VARIABLE Procedure, 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 OCI.

> **Note:**
>
> The term "bind parameter" as used in TimesTen developer guides (in keeping with ODBC terminology) is equivalent to the term "bind variable" as used in TimesTen PL/SQL documents (in keeping with Oracle Database PL/SQL terminology).

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 a define procedure (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 rows are fetched 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 cannot retrieve any more 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 a 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 `FETCH_ROWS` 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, call the LAST_ERROR_POSITION Function immediately after a `PARSE` call.

Exceptions

The following table lists the exceptions raised by `DBMS_SQL`.

Table 7-2 Exceptions Raised by DBMS_SQL

Exception	Error Code	Description
`INCONSISTENT_TYPE`	-6562	Raised by the COLUMN_VALUE Procedure or VARIABLE_VALUE Procedure when the type of the given `OUT` parameter (for where to output the requested value) is different from the type of the value.

Examples

This section provides these example procedures that use the `DBMS_SQL` package.

Example 1: Basic

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.

```
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, as follows:

```
Command> begin
            demo(14999);
         end;
         /

PL/SQL procedure successfully completed.

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

Example 2: Copy Between Tables

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.

```
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;

  -- 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;
    RAISE;
END;
```

Following 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, SMAVRIS, 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 >
< 205, Shelley, Higgins, SHIGGINS, 515.123.8080, 1994-06-07 00:00:00, AC_MGR,
12000, <NULL>, 101, 110 >
< 206, William, Gietz, WGIETZ, 515.123.8181, 1994-06-07 00:00:00, AC_ACCOUNT,
8300, <NULL>, 205, 110 >
4 rows found.
```

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, 40 >
< 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 follows:

```
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,
4800, <NULL>, 103, 60 >
< 107, Diana, Lorentz, DLORENTZ, 590.423.5567, 1999-02-07 00:00:00,
IT_PROG,
4200, <NULL>, 103, 60 >
< 204, Hermann, Baer, HBAER, 515.123.8888, 1994-06-07 00:00:00,
PR_REP, 10000,
<NULL>, 101, 70 >
6 rows found.
```

After running the example, only the employee in department 70 remains.

```
Command> select * from employees where department_id>=60 and
department_id<=70;
< 204, Hermann, Baer, HBAER, 515.123.8888, 1994-06-07 00:00:00,
PR_REP, 10000,
<NULL>, 101, 70 >
1 row found.
```

Example 6: Define an Array

This example defines 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

PL/SQL procedure successfully completed.
```

Example 7: Describe Columns

This can be used as a substitute for the `ttIsql DESCRIBE` command by using a `SELECT *` query on the table 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_LINE('col_scale           =    '
                         || rec.col_scale);
    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);

/*
 * Following loop could simply be for j in 1..col_cnt loop.
 * Here we are simply illustrating some PL/SQL table
 * features.
 */
  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;

  DBMS_SQL.CLOSE_CURSOR(c);
END;
```

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
```

```
col_type            =    1
col_maxlen          =    20
col_name            =    FIRST_NAME
col_name_len        =    10
col_schema_name     =    HR
col_schema_name_len =    8
col_precision       =    0
col_scale           =    0
col_null_ok         =    true
...
```

Example 8: RETURNING Clause

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

`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 `BIND_ARRAY` is used. `VARIABLE_VALUE` must be called to get the values in these bind variables.

> **Note:**
>
> This is similar to `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);
```

Examples are shown for single row insert, single row update, multiple row insert, multiple row update, and multiple row delete.

Single row insert

This shows a single row insert.

```
      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.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.
```

Single Row Update

This shows a 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.
```

Multiple Row Insert

This shows a multiple row insert.

```
      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 can be used to run this example in `ttIsql`:

```
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.
```

Multiple Row Update

This shows a multiple row update.

```
      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 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 is in `bnd3`. There is no way of differentiating which rows were updated of each value of `bnd1` and `bnd2`.

The following script can be used to run the example in `ttIsql`:

```
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>
```

Multiple Row Delete

v) This shows a 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 can be used to run the example in `ttIsql`.

```
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:**
>
> `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.

Example 9: PL/SQL Block in Dynamic SQL

You can execute a PL/SQL block in dynamic SQL, using either `DBMS_SQL` or `EXECUTE IMMEDIATE`. This example executes a block using `DBMS_SQL`.

Assume the following procedure:

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

Procedure created.
```

Now execute the procedure in a PL/SQL block using `DBMS_SQL`:

```
Command> declare
            c number;
            r number;
         begin
            c := dbms_sql.open_cursor;
            dbms_sql.parse(c, 'begin foo; end;', dbms_sql.native);
            r := dbms_sql.execute(c);
         end;
         /
test

PL/SQL procedure successfully completed.
```

Data Structures

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

> **Note:**
>
> - 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
- BLOB_TABLE Table Type
- CLOB_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

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

Fields

Table 7-3 DESC_REC Fields

Field	Description
col_type	Type of column
col_max_len	Maximum column length
col_name	Name of column
col_name_len	Length of column name
col_schema_name	Column schema name

Table 7-3 (Cont.) DESC_REC Fields

Field	Description
col_schema_name_len	Length of column schema name
col_precision	Precision of column
col_scale	Scale of column
col_charsetid	Column character set ID
col_charsetform	Column character set form
col_null_ok	Null column flag, `TRUE` if `NULL` is allowable

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`, which is deprecated, because column name values can be greater than 32 characters.

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 7-4 DESC_REC2 Fields

Field	Description
col_type	Type of column
col_max_len	Maximum column length
col_name	Name of column
col_name_len	Length of column name
col_schema_name	Column schema name
col_schema_name_len	Length of column schema name
col_precision	Precision of column

Table 7-4 (Cont.) DESC_REC2 Fields

Field	Description
col_scale	Scale of column
col_charsetid	Column character set ID
col_charsetform	Column character set form
col_null_ok	Null column flag, `TRUE` if `NULL` is allowable

DESC_REC3 Record Type

`DESC_REC3` is the element type of the `DESC_TAB3` table type and the DESCRIBE_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 Database type number for user-defined types), which is not currently used.

Syntax

```
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 7-5 DESC_REC3 Fields

Field	Description
col_type	Type of column
col_max_len	Maximum column length
col_name	Name of column
col_name_len	Length of column name
col_schema_name	Column schema name
col_schema_name_len	Length of column schema name

Table 7-5 (Cont.) DESC_REC3 Fields

Field	Description
col_precision	Precision of column
col_scale	Scale of column
col_charsetid	Column character set ID
col_charsetform	Column character set form
col_null_ok	Null column flag, `TRUE` if `NULL` is allowable
col_type_name	Reserved for future use
col_type_name_len	Reserved for future use

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

```
TYPE binary_float_table IS TABLE OF BINARY_FLOAT INDEX BY BINARY_INTEGER;
```

BLOB_TABLE Table Type

This is a table of `BLOB`.

Syntax

```
TYPE blob_table IS TABLE OF BLOB INDEX BY BINARY_INTEGER;
```

CLOB_TABLE Table Type

This is a table of `CLOB`.

Syntax

```
TYPE clob_table IS TABLE OF CLOB 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

```
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

```
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;
```

INTERVAL_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

```
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

```
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

```
TYPE varchar2s IS TABLE OF VARCHAR2(256) INDEX BY BINARY_INTEGER;
```

DBMS_SQL Subprograms

Table 7-6 summarizes the `DBMS_SQL` subprograms, followed by a full description of each subprogram.

Table 7-6 DBMS_SQL Package Subprograms

Subprogram	Description
BIND_ARRAY Procedure	Binds a given value to a given collection.
BIND_VARIABLE Procedure	Binds a given value to a given variable.
CLOSE_CURSOR Procedure	Closes given cursor and frees memory.
COLUMN_VALUE Procedure	Returns value of the cursor element for a given position in a cursor.
`COLUMN_VALUE_LONG` procedure	Returns a selected part of a `LONG` column that has been defined using `DEFINE_COLUMN_LONG`. **Important**: Because TimesTen does not support the `LONG` data type, attempting to use this procedure in TimesTen results in an `ORA-01018` error at runtime. The `COLUMN_VALUE_LONG` and `DEFINE_COLUMN_LONG` procedures are therefore not documented in this manual.
DEFINE_ARRAY Procedure	Defines a collection to be selected from the given cursor. Used only with `SELECT` statements.
DEFINE_COLUMN Procedure	Defines a column to be selected from the given cursor. Used only with `SELECT` statements.

Table 7-6 (Cont.) DBMS_SQL Package Subprograms

Subprogram	Description
DEFINE_COLUMN_LONG procedure	Defines a LONG column to be selected from the given cursor. Used with SELECT statements. **Important**: Because TimesTen does not support the LONG data type, attempting to use the COLUMN_VALUE_LONG procedure in TimesTen results in an ORA-01018 error at runtime. The COLUMN_VALUE_LONG and DEFINE_COLUMN_LONG procedures are therefore not documented in this manual.
DESCRIBE_COLUMNS Procedure	Describes the columns for a cursor opened and parsed through DBMS_SQL.
DESCRIBE_COLUMNS2 Procedure	Describes the specified column, as an alternative to DESCRIBE_COLUMNS Procedure.
DESCRIBE_COLUMNS3 Procedure	Describes the specified column, as an alternative to DESCRIBE_COLUMNS Procedure.
EXECUTE Function	Executes a given cursor.
EXECUTE_AND_FETCH Function	Executes a given cursor and fetches rows.
FETCH_ROWS Function	Fetches a row from a given cursor.
IS_OPEN Function	Returns TRUE if the given cursor is open.
LAST_ERROR_POSITION Function	Returns byte offset in the SQL statement text where the error occurred.
LAST_ROW_COUNT Function	Returns cumulative count of the number of rows fetched.
LAST_ROW_ID Function	Returns the rowid of last row processed, NULL for TimesTen. TimesTen does not support this feature.
LAST_SQL_FUNCTION_CODE Function	Returns SQL function code for statement.
OPEN_CURSOR Function	Returns cursor ID number of new cursor.
PARSE Procedures	Parses given statement.
TO_CURSOR_NUMBER Function	Takes an opened strongly or weakly typed REF CURSOR and transforms it into a DBMS_SQL cursor number.
TO_REFCURSOR Function	Takes an opened, parsed, and executed cursor 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.
VARIABLE_VALUE Procedure	Returns value of named variable for given cursor.

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

```
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 of the following matching pairs:

```
<bflt_tab>      dbms_sql.Binary_Float_Table
<bdbl_tab>      dbms_sql.Binary_Double_Table
<bl_tab>        dbms_sql.Blob_Table
<cl_tab>        dbms_sql.Clob_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 7-7 BIND_ARRAY Procedure Parameters

Parameter	Description
c	ID number of the cursor where the value is to be bound
name	Name of the collection in the statement
table_variable	Local variable that has been declared as *datatype*
index1	Index for the table element that marks the lower bound of the range
index2	Index for the table element that marks the upper bound of the range

Usage Notes

This section discusses usage notes for the BIND_ARRAY procedure, covering these topics:

- General Notes
- Bulk Array Binds

General 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 work space 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), 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 blob_table     IS TABLE OF BLOB           INDEX BY BINARY_INTEGER;
TYPE clob_table     IS TABLE OF CLOB           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;

<tm_tab>   Time_Table
<tms_tab>  Timestamp_Table
<ids_tab>  Interval_Day_To_Second_Table
<iym_tab>  Interval_Year_To_Month_Table
```

Examples

See Examples .

BIND_VARIABLE Procedure

This procedures 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

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

Where *datatype* can be any of the following types:

```
BINARY_DOUBLE
BINARY_FLOAT
BLOB
CLOB CHARACTER SET ANY_CS
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.

```
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 following syntax:

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

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

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

Parameters

Table 7-8 BIND_VARIABLE Procedure Parameters

Parameter	Description
c	ID number of the cursor where the value is to be bound
name	Name of the variable in the statement
value	Value to bind to the variable in the cursor For `IN` and `IN OUT` variables, the value has the same type as the type of the value being passed in for this parameter.
out_value_size	Maximum expected `OUT` value size, in bytes, for the `VARCHAR2`, `RAW`, `CHAR OUT` or `IN OUT` variable If no size is given, then the length of the current value is used. This parameter must be specified if the *value* parameter is not initialized.

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:

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

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

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

Or:

```
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 .

CLOSE_CURSOR Procedure

This procedure closes a given cursor. The memory allocated to the cursor is released and you can no longer fetch from that cursor.

Syntax

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

Parameters

Table 7-9 CLOSE_CURSOR Procedure Parameters

Parameter	Description
c	(IN) ID number of the cursor to close (OUT) NULL

COLUMN_VALUE Procedure

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

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 of the following types:

```
BINARY_DOUBLE
BINARY_FLOAT
BLOB
CLOB CHARACTER SET ANY_CS
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.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 of these matching pairs:

```
<bdbl_tab>      dbms_sql.Binary_Double_Table
<bflt_tab>      dbms_sql.Binary_Float_Table
<bl_tab>        dbms_sql.Blob_Table
<cl_tab>        dbms_sql.Clob_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
```

Parameters

Table 7-10 COLUMN_VALUE Procedure Parameters (Single Row)

Parameter	Description
c	ID number of the cursor from which you are fetching the values
position	Relative position of the column in the cursor, where the first column in a statement has position 1
value	Value returned from the specified column
column_error	Error code for the column value, if applicable
actual_length	Actual length, before any truncation, of the value in the specified column

Table 7-11 COLUMN_VALUE Procedure Parameters (Bulk)

Parameter	Description
c	ID number of the cursor from which you are fetching the values
position	Relative position of the column in the cursor, where the first column in a statement has position 1
param_name	Local variable that has been declared *table_type* The *param_name* is an `IN OUT NOCOPY` parameter for bulk operations. For bulk operations, the subprogram appends the new elements at the appropriate (implicitly maintained) index. Consider an example where the DEFINE_ARRAY Procedure is used, a batch size (the *cnt* parameter) of 10 rows is specified, and a start index (*lower_bnd*) of 1 is specified. The first call to this subprogram, after calling the FETCH_ROWS Function, populates elements at index `1..10`; the next call populates elements `11..20`; and so on.

Exceptions

```
ORA-06562: Type of out argument must match type of column or bind
variable
```

This exception 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 `DEFINE_COLUMN`.

Examples

See Examples .

DEFINE_ARRAY Procedure

This procedure defines the collection into which the row values are fetched, with a FETCH_ROWS Function call, for a given column. This procedure lets you do batch fetching of rows from a single `SELECT` statement. A single fetch brings several rows into the PL/SQL aggregate object.

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.

```
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 blob_table     IS TABLE OF BLOB           INDEX BY BINARY_INTEGER;
TYPE clob_table     IS TABLE OF CLOB           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;
```

Syntax

```
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 of the following matching pairs:

```
<bflt_tab>      dbms_sql.Binary_Float_Table
<bdbl_tab>      dbms_sql.Binary_Double_Table
<bl_tab>        dbms_sql.Blob_Table
<cl_tab>        dbms_sql.Clob_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.

Parameters

Table 7-12 DEFINE_ARRAY Procedure Parameters

Parameter	Description
c	ID number of the cursor where the array is to be bound
position	Relative position of the column in the array being defined, where the first column in a statement has position 1
table_variable	Local variable that has been declared as *datatype*
cnt	Number of rows that must be fetched
lower_bnd	Lower bound index, the starting point at which results are copied into the collection

Usage Notes

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

Exceptions

```
ORA-29253: Invalid count argument passed to procedure
dbms_sql.define_array
```

This exception is raised if the count (*cnt*) is less than or equal to zero.

Examples

See Examples .

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

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

Where *datatype* can be any of the following types:

```
BINARY_DOUBLE
BINARY_FLOAT
BLOB
CLOB CHARACTER SET ANY_CS
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
Nested table
```

Note that `DEFINE_COLUMN` is overloaded to accept different data types.

The following syntax is also supported for the `DEFINE_COLUMN` procedure:

```
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:

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

DBMS_SQL.DEFINE_COLUMN_RAW (
   c              IN INTEGER,
```

```
   position        IN INTEGER,
   column          IN RAW,
   column_size     IN INTEGER);

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

Parameters

Table 7-13 DEFINE_COLUMN Procedure Parameters

Parameter	Description
c	ID number of the cursor for the row being defined to be selected
position	Relative position of the column in the row being defined, where the first column in a statement has position 1
column	Value of the column being defined The type of this value determines the type for the column being defined.
column_size	Maximum expected size of the column value, in bytes, for columns of type `VARCHAR2`, `CHAR`, and `RAW`

Examples

See Examples .

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 7-14 DESCRIBE_COLUMNS Procedure Parameters

Parameter	Description
c	ID number of the cursor for the columns being described

Table 7-14 (Cont.) DESCRIBE_COLUMNS Procedure Parameters

Parameter	Description
col_cnt	Number of columns in the select list of the query
desc_t	Table to fill in with the description of each of the columns of the query

Examples

See Examples .

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);
```

Parameters

Table 7-15 DESCRIBE_COLUMNS2 Procedure Parameters

Parameter	Description
c	ID number of the cursor for the columns being described
col_cnt	Number of columns in the select list of the query
desc_t	Table to fill in with the description of each of the columns of the query, indexed from 1 to the number of elements in the select list of the query

DESCRIBE_COLUMNS3 Procedure

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

Syntax

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

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

Parameters

Table 7-16 DESCRIBE_COLUMNS3 Procedure Parameters

Parameter	Description
c	ID number of the cursor for the columns being described
col_cnt	Number of columns in the select list of the query
desc_t	Table to fill in with the description of each of the columns of the query, indexed from 1 to the number of elements in the select list of the query

Usage Notes

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

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

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

Parameters

Table 7-17 EXECUTE Function Parameters

Parameter	Description
c	Cursor ID number of the cursor to execute

Return Value

An `INTEGER` value that indicates the number of rows processed

EXECUTE_AND_FETCH Function

This function executes the given cursor and fetches rows. It provides the same functionality as calling `EXECUTE` and then calling `FETCH_ROWS`; however, calling

EXECUTE_AND_FETCH 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;
```

Parameters

Table 7-18 EXECUTE_AND_FETCH Function Parameters

Parameter	Description
c	ID number of the cursor to execute and fetch
exact	TRUE to raise an exception if the number of rows actually matching the query differs from 1 Even if an exception is raised, the rows are still fetched and available.

Return Value

An INTEGER value indicating the number of rows that were fetched

Exceptions

```
ORA-01422: Exact fetch returns more than requested number of rows
```

This exception is raised if the number of rows matching the query is not 1.

FETCH_ROWS Function

This function fetches a row from a given cursor.

A DEFINE_ARRAY Procedure call defines the collection into which the row values are fetched.

A FETCH_ROWS call fetches the specified number of rows, according to the *cnt* parameter of the DEFINE_ARRAY call. When you fetch the rows, they are copied into DBMS_SQL buffers until you execute a COLUMN_VALUE Procedure call, for each column, at which time the rows are copied into the table that was passed as an argument to COLUMN_VALUE. The rows are placed in positions *lower_bnd*, *lower_bnd*+1, *lower_bnd*+2, and so on, according to the *lower_bnd* setting in the DEFINE_ARRAY call. While there are still rows coming in, the user keeps issuing FETCH_ROWS and COLUMN_VALUE calls. You can call FETCH_ROWS repeatedly as long as there are rows remaining to be fetched.

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

Syntax

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

Parameters

Table 7-19 FETCH_ROWS Function Parameters

Parameter	Description
c	ID number of the cursor to fetch

Return Value

An `INTEGER` value indicating the number of rows that were fetched

Examples

See Examples .

IS_OPEN Function

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

Syntax

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

Parameters

Table 7-20 IS_OPEN Function Parameters

Parameter	Description
c	Cursor ID number of the cursor to check

Return Value

`TRUE` for any cursor number that has been opened but not closed, or `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 raises this error, meaning that DBMS_SQL is non-operational for the 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;
```

Return Value

An INTEGER value indicating 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;
```

Return Value

An INTEGER value indicating 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, the value returned is zero.

LAST_ROW_ID Function

This function returns the rowid of the last row processed, but NULL for TimesTen.

TimesTen does not support this feature.

Syntax

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

Return Value

`NULL` for TimesTen

LAST_SQL_FUNCTION_CODE Function

This function returns the SQL function code for the statement.

These codes are listed in the OCI Function Codes table in *Oracle Call Interface Programmer's Guide*.

Syntax

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

Return Value

An `INTEGER` value indicating 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. In TimesTen, however, there is no security enforcement: `security_level=0`.

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 7-21 OPEN_CURSOR Function Parameters

Parameter	Description
security_level	Specifies the level of security protection to enforce on the opened cursor. Only the security level 0 is valid in TimesTen (levels 1 and 2 are not supported). • 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 user ID or roles than at the time the cursor was parsed. This level of security is disabled by default. • Level 1 is not applicable for TimesTen. • Level 2 is not applicable for TimesTen.

Return Value

The cursor ID 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 Procedures

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
- Taking VARCHAR2S, table of VARCHAR2(32767), as an argument
- Taking a CLOB statement as an argument

  You can use the CLOB overload version of the parse procedure to parse a SQL statement larger than 32 KB.

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

```
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);

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

Parameters

Table 7-22 PARSE Procedure Parameters

Parameter	Description
c	ID number of the cursor in which to parse the statement

statement	SQL statement to be parsed
	SQL statements larger than 32 KB can be stored in CLOBs.
	Unlike PL/SQL statements, your SQL statement should not include a final semicolon. For example:
	
	`DBMS_SQL.PARSE(cursor1, 'BEGIN proc; END;', 2);`
	`DBMS_SQL.PARSE(cursor1, 'INSERT INTO tab VALUES(1)',2);`

lb	Lower bound for elements in the statement
ub	Upper bound for elements in the statement
lfflg	`TRUE` to insert a line feed after each element on concatenation
language_flag	Flag to determine how the SQL statement is handled For TimesTen, use the `NATIVE` (or 1) setting, which specifies typical behavior for the database to which the program is connected.

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:

```
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:

```
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

```
ORA-24344: Success with compilation error
```

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

Examples

See Examples .

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

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

Parameters

Table 7-23 TO_CURSOR_NUMBER Function Parameters

Parameter	Description
rc	REF CURSOR to be transformed into a cursor number

Return Value

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

```
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
            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 (by `OPEN_CURSOR`), parsed (by `PARSE`), and executed (by `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 7-24 TO_REFCURSOR Function Parameters

Parameter	Description
cursor_number	Cursor number of the cursor to be transformed into a REF CURSOR

Return Value

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, it is no longer accessible by any `DBMS_SQL` operations.
- After a *cursor_number* is transformed into a REF CURSOR, using `IS_OPEN` results in an error.
- Toggling between REF CURSOR and `DBMS_SQL` cursor number after starting to fetch is not allowed. An error is raised.

Examples

```
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';

  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 of the following types:

```
BINARY_DOUBLE
BINARY_FLOAT
BLOB
CLOB CHARACTER SET ANY_CS
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 any of the following:

```
dbms_sql.Binary_Double_Table
dbms_sql.Binary_Float_Table
dbms_sql.Blob_Table
dbms_sql.Clob_Table
dbms_sql.Date_Table
dbms_sql.Interval_Day_To_Second_Table
dbms_sql.Interval_Year_To_Month_Table
dbms_sql.Number_Table
dbms_sql.Time_Table
dbms_sql.Timestamp_Table
dbms_sql.Varchar2_Table
```

Parameters

Table 7-25 VARIABLE_VALUE Procedure Parameters

Parameter	Description
c	ID number of the cursor from which to get the values
name	Name of the variable for which you are retrieving the value
value	For the single row option, value of the variable for the specified position For the array option, local variable that has been declared *table_type* **Note**: For bulk operations, *value* is an `OUT NOCOPY` parameter.

Exceptions

```
ORA-06562: Type of out argument must match type of column or bind
variable
```

This is raised if the type of the output parameter differs from the type of the value as defined by the `BIND_VARIABLE` call.

Examples

See Examples .

8
DBMS_UTILITY

The DBMS_UTILITY package provides various utility subprograms.

This chapter contains the following topics:

Using DBMS_UTILITY

Security Model

DBMS_UTILITY 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 SYS.

Constants

The DBMS_UTILITY package uses the constants shown in Table 8-1.

Table 8-1 DBMS_UTILITY Constants

Name	Type	Value	Description
INV_ERROR_ON_RESTRICTIONS	BINARY_INTEGER	1	This constant is the only valid value for the *p_option_flags* parameter of the INVALIDATE subprogram.

> **Note:**
>
> - 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.

Data Types

dblink_array

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

Lists of database links would be stored here. (TimesTen does not support dblinks.)

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, including 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.

uncl_array

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

Lists of "*`USER`*"."*`NAME`*"."*`COLUMN`*"@`LINK` should be stored here.

Exceptions

The following table lists the exceptions raised by `DBMS_UTILITY`.

Table 8-2 Exceptions Raised by DBMS_UTILITY

Exception	Error Code	Description
`INV_NOT_EXIST_OR_NO_PRIV`	-24237	Raised by the `INVALIDATE` subprogram when the *`object_id`* argument is `NULL` or invalid, or when the caller does not have `CREATE` privilege on the object being invalidated.
`INV_MALFORMED_SETTINGS`	-24238	Raised by the `INVALIDATE` subprogram if a compiler setting is specified more than once in the *`p_plsql_object_settings`* parameter.
`INV_RESTRICTED_OBJECT`	-24239	Raised by the `INVALIDATE` subprogram when different combinations of conditions pertaining to the *`p_object_id`* parameter are contravened.

DBMS_UTILITY Subprograms

Table 8-3 summarizes the `DBMS_UTILITY` subprograms, followed by a full description of each subprogram.

Table 8-3 ***DBMS_UTILITY Package Subprograms***

Subprogram	Description
CANONICALIZE Procedure	Canonicalizes a given string.
COMMA_TO_TABLE Procedure	Converts a comma-delimited list of names into a PL/SQL table of names.
COMPILE_SCHEMA Procedure	Compiles all procedures, functions, packages, and views in the specified schema.
DB_VERSION Procedure	Returns version information for the database. Returns `NULL` for the compatibility setting because TimesTen does not support the system parameter `COMPATIBLE`.
FORMAT_CALL_STACK Function	Formats the current call stack.
FORMAT_ERROR_BACKTRACE Function	Formats the backtrace from the point of the current error to the exception handler where the error has been caught.
FORMAT_ERROR_STACK Function	Formats the current error stack.
GET_CPU_TIME Function	Returns the current CPU time in hundredths of a second.
GET_DEPENDENCY Procedure	Shows the dependencies on the object passed in.
GET_ENDIANNESS Function	Returns the endianness of your database platform.
GET_HASH_VALUE Function	Computes a hash value for the given string.
GET_SQL_HASH Function	Computes the hash value for a given string using the MD5 algorithm.
GET_TIME Function	Finds out the current time in hundredths of a second.
INVALIDATE Procedure	Invalidates a database object and (optionally) modifies its PL/SQL compiler parameter settings.
IS_BIT_SET Function	Checks the setting of a specified bit in a `RAW` value.
NAME_RESOLVE Procedure	Resolves the given name of the form: `[[a.]b.]c[@dblink]` Where *a*, *b*, and *c* are SQL identifiers and *dblink* is a dblink. **Important**: Do not use `@dblink`. TimesTen does not support dblinks.
NAME_TOKENIZE Procedure	Calls the parser to parse the given name: `'a[.b[.c]][@dblink]"` Where *a*, *b*, and *c* are SQL identifiers and *dblink* 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`. **Important**: Do not use `@dblink`. TimesTen does not support dblinks.

Table 8-3 (Cont.) *DBMS_UTILITY Package Subprograms*

Subprogram	Description
TABLE_TO_COMMA Procedure	Converts a PL/SQL table of names into a comma-delimited list of names.
VALIDATE Procedure	Validates the object described either by owner, name, and namespace or by object ID.

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

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

Parameters

Table 8-4 CANONICALIZE Procedure Parameters

Parameter	Description
name	The string to be canonicalized
canon_name	The canonicalized string
canon_len	The length of the string (in bytes) to canonicalize

Return Value

The first *canon_len* bytes in *canon_name*

Usage Notes

- If the *name* value is `NULL`, the *canon_name* value becomes `NULL`.
- If `name` is a dotted name (such as `a."b".c`), then for each component in the dotted name where the component begins and ends with a double quote, no transformation is performed on that 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 is concatenated in the input position, separated by ".".
- 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.
- Any other character after `a[.b]*` is 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

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

Syntax

```
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 8-5 COMMA_TO_TABLE Procedure Parameters

Parameter	Description
list	Comma-delimited list of names, where a name should have the following format for the first version of the procedure: *a*[.*b*[.*c*]][@*d*] Or the following format for the second version of the procedure: *a*[.*b*]* Where *a*, *b*, *c*, and *d* are simple identifiers (quoted or unquoted).
tablen	Number of tables in the PL/SQL table
tab	PL/SQL table that contains list of names

Return Value

A PL/SQL table 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 8-6 COMPILE_SCHEMA Procedure Parameters

Parameter	Description
schema	Name of the schema
compile_all	TRUE to compile everything within the schema regardless of whether status is VALID FALSE to compile only objects with status INVALID
reuse_settings	Flag to specify whether the session settings in the objects should be reused, or the current session settings should be adopted instead

Usage Notes

- Note that this subprogram is a wrapper for the RECOMP_SERIAL Procedure included with the UTL_RECOMP package.
- After calling this procedure, you should select from view ALL_OBJECTS for items with status 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;
```

Exceptions

Table 8-7 COMPILE_SCHEMA Procedure Exceptions

Exception	Description
ORA-20000	Raised for insufficient privileges for some object in this schema.
ORA-20001	Raised if SYS objects cannot be compiled.
ORA-20002	Raised if maximum iterations exceeded. Some objects may not have been recompiled.

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.

Also see TT_DB_VERSION.

Syntax

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

Parameters

Table 8-8 DB_VERSION Procedure Parameters

Parameter	Description
version	String that represents the internal software version of the database (for example, 22.1.1.1.0) The length of this string is variable and is determined by the database version.
compatibility	Compatibility setting of the database In TimesTen, DB_VERSION returns NULL for the compatibility setting because TimesTen does not support the system parameter COMPATIBLE.

FORMAT_CALL_STACK Function

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

Syntax

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

Return Value

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 (or a null string 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;
/

-- Define and raise an exception to view backtrace.
-- See EXCEPTION_INIT Pragma in Oracle Database PL/SQL Language
Reference.

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. It can be used in exception handlers to look at the full error stack.

Syntax

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

Return Value

The error stack, up to 2000 bytes (or a null string if no error is currently being handled)

GET_CPU_TIME Function

This function returns a measure of current CPU processing time in hundredths of a second. The difference between the times returned from two calls measures the CPU processing time (not the total elapsed time) between those two points.

Also see the GET_TIME Function, which has a different intent.

Syntax

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

Return Value

The number of hundredths of a second of CPU processing time from some arbitrary point

Usage Notes

This subprogram reports cycles (CPU time) used in performing work and is unrelated to clock time or any other fixed reference. It always returns a positive value. The amount of work performed is calculated by measuring the difference between a start point and end point for a particular operation, using a `GET_CPU_TIME` call at each point.

GET_DEPENDENCY Procedure

This procedure shows the dependencies on the object passed in.

Syntax

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

Parameters

Table 8-9 GET_DEPENDENCY Procedure Parameters

Parameter	Description
type	The type of the object For example, if the object is a table, give the type as "TABLE".
schema	The schema name of the object
name	The name of the object

Usage Notes

This procedure uses the DBMS_OUTPUT package to display results, so you must declare `SET SERVEROUTPUT ON` from `ttIsql` to view dependencies. Alternatively, any application that checks the `DBMS_OUTPUT` output buffers can invoke this subprogram and then retrieve the output through `DBMS_OUTPUT` subprograms such as `GET_LINES`.

GET_ENDIANNESS Function

This function indicates the endianness of the database platform.

Syntax

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

Return Value

A `NUMBER` value indicating the endianness of the database platform: 1 for big-endian or 2 for little-endian

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 8-10 GET_HASH_VALUE Function Parameters

Parameter	Description
name	String to be hashed
base	Base value where the returned hash value is to start
hash_size	Desired size of the hash table

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 *hash_size* works best.

GET_SQL_HASH Function

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

Syntax

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

Parameters

Table 8-11 GET_SQL_HASH Procedure Parameters

Parameter	Description
name	String to be hashed
hash	A field to store all 16 bytes of returned hash value
pre10ihash	A field to store a pre-10*g* Oracle Database version hash value

Return Value

A hash value (last four bytes) based on the input string

The MD5 hash algorithm computes a 16-byte hash value, but TimesTen uses only the last four bytes to return a number. The *hash* output parameter gets all 16 bytes.

Example

This example displays the 16-byte hash value (ignoring both the four-byte returned hash value and the pre-10*g* hash value).

```
Command> declare
           ignore_hash_4b    number := 0;
           ignore_pre10ihash number := 0;
           hash_16B          RAW(16);
           query_text        varchar2(255);
         begin
           query_text := 'SELECT * FROM dual';
              -- Calculate the hash of the SQL text
              ignore_hash_4b := DBMS_UTILITY.GET_SQL_HASH(
                                query_text,
                                hash_16B,
                                ignore_pre10ihash);
           dbms_output.put_line('>' || query_text || '< hash is ' ||
hash_16B);
         end;
         /
>SELECT * FROM dual< hash is 462D200E640BC1CBBDFE01B36A231600
```

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

GET_TIME Function

This function returns a measure of current time in hundredths of a second. The difference between the times returned from two calls measures the total elapsed time (not just CPU processing time) between those two points.

Also see the GET_CPU_TIME Function, which has a different intent.

Syntax

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

Return Value

The number of hundredths of a second from the time at which the subprogram is invoked

Usage Notes

Numbers are returned in the range -2,147,483,648 to 2,147,483,647 depending on platform and system, and your application must take the sign of the number into account in determining the interval. For example, for two negative numbers, application logic must allow for the first (earlier) number to be larger than the second (later) number that is closer to zero. By the same token, your application should also allow for the first (earlier) number to be negative and the second (later) number to 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

```
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 8-12 INVALIDATE Procedure Parameters

Parameter	Description
p_object_id	ID number of the object to be invalidated This equals 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 CREATE privilege on the object being invalidated, otherwise the inv_not_exist_or_no_priv exception is raised.
p_plsql_object_settings	Optional parameter that is ignored if the object specified by *p_object_id* is not a PL/SQL object. • If no value is specified for this parameter, the PL/SQL compiler settings are left unchanged, 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. • If a setting is specified more than once, the inv_malformed_settings exception is 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 value. • If a setting is omitted and REUSE SETTINGS was not specified or no value was specified for the parameter in an earlier compilation, then the database obtains the value for that setting from the session environment.
p_option_flags	Optional parameter that defaults to zero (no flags) Only the inv_error_on_restrictions flag is supported (see Constants). With this flag, 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.

Usage Notes

The object type (object_type column from ALL_OBJECTS) of the object that is 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*, the exception inv_restricted_object is raised. If inv_error_on_restrictions is not specified in this situation, then no action is taken.

If the object specified by *p_object_id* is the package specification of STANDARD or DBMS_STANDARD, or the specification or body of DBMS_UTILITY, and if the flag inv_error_on_restrictions is specified in *p_option_flags*, then the exception

inv_restricted_object is raised. If inv_error_on_restrictions is not specified in this situation, then no action is taken.

If the object specified by *p_object_id* is an object type specification and there are tables that depend on the type, and if the flag inv_error_on_restrictions is specified in *p_option_flags*, then the exception inv_restricted_object is raised. If inv_error_on_restrictions is not specified, then no action is taken.

Exceptions

Table 8-13 INVALIDATE Exceptions

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

Examples

This example invalidates a procedure created in the example in FORMAT_ERROR_BACKTRACE Function. 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;
...
< LOG_ERRORS, <NULL>, 296, <NULL>, PROCEDURE, 2009-09-18 12:53:45,
2009-09-18 12
```

```
:58:22, 2009-09-18:12:58:22, VALID, N, N, N, 1, <NULL> >
< P0, <NULL>, 297, <NULL>, PROCEDURE, 2009-09-18 12:53:45, 2009-09-18
12:58:22,
2009-09-18:12:58:22, VALID, N, N, N, 1, <NULL> >
< P1, <NULL>, 298, <NULL>, PROCEDURE, 2009-09-18 12:53:45, 2009-09-18
12:58:22,
2009-09-18:12:58:22, VALID, N, N, N, 1, <NULL> >
< P2, <NULL>, 299, <NULL>, PROCEDURE, 2009-09-18 12:53:45, 2009-09-18
12:58:22,
2009-09-18:12:58:22, VALID, N, N, N, 1, <NULL> >
< P3, <NULL>, 300, <NULL>, PROCEDURE, 2009-09-18 12:53:45, 2009-09-18
12:58:22,
2009-09-18:12:58:22, VALID, N, N, N, 1, <NULL> >
< P4, <NULL>, 301, <NULL>, PROCEDURE, 2009-09-18 12:53:45, 2009-09-18
12:58:22,
2009-09-18:12:58:22, VALID, N, N, N, 1, <NULL> >
< P5, <NULL>, 302, <NULL>, PROCEDURE, 2009-09-18 12:53:45, 2009-09-18
12:58:22,
2009-09-18:12:58:22, VALID, N, N, N, 1, <NULL> >
< TOP_NAIVE, <NULL>, 303, <NULL>, PROCEDURE, 2009-09-18 12:53:45, 2009-09-18
12:
58:22, 2009-09-18:12:58:22, VALID, N, N, N, 1, <NULL> >
< TOP_WITH_LOGGING, <NULL>, 304, <NULL>, PROCEDURE, 2009-09-18 12:53:45,
2009-09
-18 15:19:16, 2009-09-18:15:19:16, VALID, N, N, N, 1, <NULL> >
...
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 marks the `P5` procedure as invalid and sets its `PLSQL_OPTIMIZE_LEVEL` compiler setting to 2. The values of other compiler settings 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';
< P5, <NULL>, 302, <NULL>, PROCEDURE, 2009-09-18 12:53:45, 2009-09-18
12:58:22,
2009-09-18:12:58:22, INVALID, N, N, N, 1, <NULL> >
< TOP_NAIVE, <NULL>, 303, <NULL>, PROCEDURE, 2009-09-18 12:53:45, 2009-09-18
12:
58:22, 2009-09-18:12:58:22, INVALID, N, N, N, 1, <NULL> >
< TOP_WITH_LOGGING, <NULL>, 304, <NULL>, PROCEDURE, 2009-09-18 12:53:45,
2009-09
-18 15:19:16, 2009-09-18:15:19:16, 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;
```

Parameters

Table 8-14 IS_BIT_SET Procedure Parameters

Parameter	Description
r	Source raw
n	Which bit in *r* to check

Return Value

1 if bit *n* in `RAW` *r* is set, where 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[@dblink]
```

Where *a*, *b*, and *c* are SQL identifiers and *dblink* is a dblink, including synonym translation and authorization checking as necessary.

Do not use *@dblink*. TimesTen does not support dblinks.

Syntax

```
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 8-15 NAME_RESOLVE Procedure Parameters

Parameter	Description
name	Name of the object This can be of the form: `[[a.]b.]c[@dblink]` Where *a*, *b*, and *c* are SQL identifiers and *dblink* is a dblink. TimesTen does not support dblinks. No syntax checking is performed on the dblink. If a dblink is specified, or if the name resolves to something with a dblink, then the object is not resolved, but the *schema*, *part1*, *part2*, and *dblink* OUT parameters receive values. The *a*, *b* and *c* entries may be delimited identifiers, and may contain Globalization Support (NLS) characters, either single or multibyte.
context	An integer from 0 to 9, as follows: • 0 - Table • 1 - PL/SQL (for two-part names) • 2 - Sequences • 3 - Trigger (not applicable for TimesTen) • 4 - Java source (not applicable for TimesTen) • 5 - Java resource (not applicable for TimesTen) • 6 - Java class (not applicable for TimesTen) • 7 - Type (not applicable for TimesTen) • 8 - Java shared data (not applicable for TimesTen) • 9 - Index
schema	Schema of the object, *c* If no schema is specified in *name*, then *schema* is determined by resolving the name.
part1	First part of the name The type of this name is specified *part1_type* (synonym or package).
part2	Subprogram name, as applicable, or NULL. 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.
dblink	Not applicable TimesTen does not support dblinks.
part1_type	Type of *part1*, as follows: • 5 - Synonym • 7 - Procedure (top level) • 8 - Function (top level) • 9 - Package
object_number	Object identifier

Exceptions

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

NAME_TOKENIZE Procedure

This procedure calls the parser to parse the input name.

```
"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 dblinks.

Syntax

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

Parameters

Table 8-16 NAME_TOKENIZE Procedure Parameters

Parameter	Description
name	The input name, consisting of SQL identifiers (for example, `scott.foo`)
a	Output for the first token of the name
b	Output for the second token of the name (if applicable)
c	Output for the third token of the name (if applicable)
dblink	Output for the dblink of the name (not applicable for TimesTen)
nextpos	Next position after parsing the input name

Examples

Consider the following script to run in `ttIsql`:

```
declare
   a varchar2(30);
   b varchar2(30);
   c varchar2(30);
   d varchar2(30);
   next integer;

begin
   dbms_utility.name_tokenize('scott.foo', a, b, c, d, next);
   dbms_output.put_line('a: ' || a);
```

```
   dbms_output.put_line('b: ' || b);
   dbms_output.put_line('c: ' || c);
   dbms_output.put_line('d: ' || d);
   dbms_output.put_line('next: ' || next);
end;
/
```

This produces the following output.

```
a: SCOTT
b: FOO
c:
d:
next: 9

PL/SQL procedure successfully completed.
```

TABLE_TO_COMMA Procedure

This procedure 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

```
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 8-17 TABLE_TO_COMMA Procedure Parameters

Parameter	Description
tab	PL/SQL table that contains list of table names
tablen	Number of tables in the PL/SQL table
list	Comma-delimited list of tables

Return Value

A `VARCHAR2` value with 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 8-18 VALIDATE Procedure Parameters

Parameter	Description
object_id	ID number of the object to be validated See INVALIDATE Procedure.
owner	Name of the user who owns the object Same as the OWNER field in ALL_OBJECTS.
objname	Name of the object to be validated Same as the OBJECT_NAME field in ALL_OBJECTS.
namespace	Namespace of the object Same as the *namespace* field in obj$. Equivalent numeric values are as follows: • 1 - Table or procedure • 2 - Body • 3 - Trigger (not applicable for TimesTen) • 4 - Index • 5 - Cluster • 9 - Directory • 10 - Queue • 11 - Replication object group • 12 - Replication propagator • 13 - Java source (not applicable for TimesTen) • 14 - Java resource (not applicable for TimesTen) • 58 - Model (data mining)
edition_name	Reserved for future use

Usage Notes

- Executing VALIDATE on a subprogram also validates subprograms that it references. (See the example below.)
- No errors are raised if the object does not exist, is already valid, or is an object that cannot be validated.

- 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* value.

Examples

This example starts where the `INVALIDATE` example in INVALIDATE Procedure left off. Assume `P5`, `Top_Naive`, and `Top_With_Logging` are invalid, shown as follows:

```
Command> select * from user_objects where status='INVALID';
< P5, <NULL>, 302, <NULL>, PROCEDURE, 2009-09-18 12:53:45, 2009-09-18
12:58:22,
2009-09-18:12:58:22, INVALID, N, N, N, 1, <NULL> >
< TOP_NAIVE, <NULL>, 303, <NULL>, PROCEDURE, 2009-09-18 12:53:45, 2009-09-18
12:
58:22, 2009-09-18:12:58:22, INVALID, N, N, N, 1, <NULL> >
< TOP_WITH_LOGGING, <NULL>, 304, <NULL>, PROCEDURE, 2009-09-18 12:53:45,
2009-09
-18 15:19:16, 2009-09-18:15:19:16, INVALID, N, N, N, 1, <NULL> >
3 rows found.
```

Validating `Top_With_Logging`, for example, also validates `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';
< TOP_NAIVE, <NULL>, 303, <NULL>, PROCEDURE, 2009-09-18 12:53:45, 2009-09-21
11:
14:37, 2009-09-21:11:14:37, INVALID, N, N, N, 1, <NULL> >
1 row found.
```

9
TT_DB_VERSION

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 has boolean variables to indicate which TimesTen major release the package is supplied with.

The package for TimesTen Release 22.1 is as follows:

```
PACKAGE TT_DB_VERSION IS
   VER_LE_1121 CONSTANT BOOLEAN := FALSE;
   VER_LE_1122 CONSTANT BOOLEAN := FALSE;
   VER_LE_1801 CONSTANT BOOLEAN := FALSE;
   VER_LE_2201 CONSTANT BOOLEAN := TRUE;
   ...
END TT_DB_VERSION;
```

Constants

The TT_DB_VERSION package contains boolean constants indicating the current TimesTen release. These are shown in Table 9-1 for TimesTen Release 22.1.

Table 9-1 TT_DB_VERSION Constants

Name	Type	Value	Description
VER_LE_1121	BOOLEAN	FALSE	Boolean that is TRUE if the TimesTen version this package is supplied with is TimesTen 11.2.1 or prior

Table 9-1 (Cont.) TT_DB_VERSION Constants

Name	Type	Value	Description
VER_LE_1122	BOOLEAN	FALSE	Boolean that is TRUE if the TimesTen version this package is supplied with is TimesTen 11*g* Release 2 (11.2.2) or prior.
VER_LE_1801	BOOLEAN	FALSE	Boolean that is TRUE if the TimesTen version this package is supplied with is TimesTen Release 18.1 or prior
VER_LE_2201	BOOLEAN	TRUE	Boolean that is TRUE if the TimesTen version this package is supplied with is TimesTen Release 22.1 or prior

Examples

See Examples in the UTL_IDENT chapter for an example that uses both that package and TT_DB_VERSION for conditional compilation.

10
TT_STATS

The `TT_STATS` package enables you to collect snapshots of TimesTen Classic database metrics (statistics, states, and other information) and generate reports comparing two specified snapshots.

This chapter contains the following topics:

> **Note:**
>
> There is also a `ttStats` utility program. In addition to acting as a convenient front-end for the `TT_STATS` package to collect snapshots and generate reports, the utility can monitor metrics in real-time. See ttStats in *Oracle TimesTen In-Memory Database Reference*.

Using TT_STATS

This section covers the following topics for the `TT_STATS` package:

Overview

The `TT_STATS` package provides features for collecting and comparing snapshots of TimesTen system metrics, according to the capture level. Each snapshot can consist of what TimesTen considers to be basic metrics, typical metrics, or all available metrics.

For those familiar with Oracle Database performance analysis tools, these reports are similar in nature to Oracle Automatic Workload Repository (AWR) reports.

The package includes procedures and functions for the following:

- Capture a snapshot of metrics according to the capture level.
- Generate a report in HTML or plain text showing before and after values of metrics or the differences between those values.

- Show the snapshot ID and timestamp of snapshots currently stored.
- Delete snapshots based on a range of IDs or timestamps.
- Get or set the value of a specified `TT_STATS` configuration parameter.
- Show the values of all configuration parameters.

> **Note:**
>
> The only supported configuration parameters are for the maximum number of snapshots and the maximum total size of snapshots that can be stored.

Security Model

By default, only the instance administrator has privilege to run functions or procedures of the `TT_STATS` PL/SQL package.

Any other user, including an `ADMIN` user, must be granted `EXECUTE` privilege for the `TT_STATS` package by the instance administrator or by an `ADMIN` user, such as in the following example:

```
GRANT EXECUTE ON SYS.TT_STATS TO scott;
```

> **Note:**
>
> Although `ADMIN` users cannot execute the package by default, they can grant themselves privilege to execute it.

Operational Notes

Each metric in the `SYS.SYSTEMSTATS` table has a designated level, and the capture level setting for a snapshot corresponds to those levels. Available levels are `NONE`, `BASIC`, `TYPICAL` (the default, appropriate for most purposes), and `ALL`. See CAPTURE_SNAPSHOT Procedure and Function.

Be aware that the capture level applies only to metrics in the `SYS.SYSTEMSTATS` table, however. For metrics outside of `SYSTEMSTATS`, the same set of data are gathered regardless of the capture level.

> **Note:**
>
> You can also use the `ttStatsConfig` built-in procedure to change the capture level. See ttStatsConfig in *Oracle TimesTen In-Memory Database Reference*.

Snapshots are stored in a number of TimesTen `SYS.SNAPSHOT_xxxxx` system tables. To assist you in minimizing the risk of running out of permanent space, the `TT_STATS` package has configuration parameters to specify the maximum number of snapshots that can be stored and the total size of snapshots stored. In this release, an error is issued if either limit is exceeded, and the snapshot capture would fail. TimesTen provides default limits, but you can alter them through the `SET_CONFIG` procedure. (See SET_CONFIG Procedure.)

Be aware that execution of this package may involve numerous reads and insertions, which may impact database performance during package operations.

TT_STATS Subprograms

Table 10-1 summarizes the `TT_STATS` subprograms, followed by a full description of each subprogram.

Table 10-1 TT_STATS Package Subprograms

Subprogram	Description
CAPTURE_SNAPSHOT Procedure and Function	Takes a snapshot of TimesTen metrics. The function also returns the snapshot ID.
DROP_SNAPSHOTS_RANGE Procedures	Deletes snapshots according to a specified range of snapshot IDs or timestamps.
GENERATE_REPORT_HTML Procedure	Produces a report in HTML format based on the data from two specified snapshots.
GENERATE_REPORT_TEXT Procedure	Produces a report in plain text format based on the data from two specified snapshots.
GET_CONFIG Procedures	Retrieves the value of a specified `TT_STATS` configuration parameter or the values of all configuration parameters.
SET_CONFIG Procedure	Sets a specified value for a specified `TT_STATS` configuration parameter.
SHOW_SNAPSHOTS Procedures	Shows the snapshot IDs and timestamps of all snapshots currently stored in the database.

Note:

The only supported `TT_STATS` configuration parameters are for limits of the number of snapshots and total size of snapshots that can be stored in the permanent memory segment.

CAPTURE_SNAPSHOT Procedure and Function

The procedure captures a snapshot of TimesTen metrics according to the specified capture level, or by default uses what is considered a typical level. The snapshots are stored in TimesTen `SYS.SNAPSHOT_xxxx` system tables.

The function does the same and also returns the ID number of the snapshot.

> **Note:**
>
> - The capture level applies only to metrics from `SYS.SYSTEMSTATS`, as discussed below.
> - There are defined limits for the maximum number of snapshots that can be stored and the maximum total size of all stored snapshots. See SET_CONFIG Procedure.

Syntax

```
TT_STATS.CAPTURE_SNAPSHOT (
   capture_level     IN VARCHAR2 DEFAULTED,
   description       IN VARCHAR2 DEFAULTED);

TT_STATS.CAPTURE_SNAPSHOT (
   capture_level     IN VARCHAR2 DEFAULTED,
   description       IN VARCHAR2 DEFAULTED)
  RETURN BINARY_INTEGER;
```

Parameters

Table 10-2 CAPTURE_SNAPSHOT Procedure Parameters

Parameter	Description
capture_level	The desired level of metrics to capture The following choices are available: • `NONE`: For metrics outside of `SYS.SYSTEMSTATS` only. • `BASIC`: For a minimal basic set of metrics. • `TYPICAL` (default): For a typical set of metrics. This level is appropriate for most purposes. The basic set is a subset of the typical set. • `ALL`: For all available metrics. Use the same level for any two snapshots to be used in a report. **Note**: For metrics outside of `SYS.SYSTEMSTATS`, the same data are gathered regardless of the capture level.
description	An optional description of the snapshot Use this if you want to provide any description or notes for the snapshot, such as to distinguish it from other snapshots.

Usage Notes

- As mentioned above, the capture level applies only to metrics in the `SYS.SYSTEMSTATS` table. For metrics outside of `SYSTEMSTATS`, the same data are gathered regardless of the capture level.
- For `SYSTEMSTATS` metrics, only those within the specified capture level have meaningful accumulated values. `SYSTEMSTATS` metrics outside of the specified level have a value of 0 (zero).

- You can call the procedure or function without specifying the *capture_level* parameter. This results in capture of what is considered a typical level of metrics.

Return Value

The function returns a BINARY_INTEGER value for the ID of the snapshot.

Examples

Capture just the basic metrics:

```
call tt_stats.capture_snapshop('BASIC');
```

Capture the default typical level of metrics:

```
call tt_stats.capture_snapshot;
```

This example uses the function to capture the default typical level of metrics and displays the snapshot ID:

```
declare
   id   number;
begin
   id := tt_stats.capture_snapshot();
   dbms_output.put_line('Snapshot with ID (' || id || ') was captured.');
end;
```

DROP_SNAPSHOTS_RANGE Procedures

This procedure deletes previously captured snapshots of TimesTen metrics according to a specified range of snapshot IDs or timestamps.

> **Note:**
>
> You can use the SHOW_SNAPSHOTS Procedures to display the IDs and timestamps of all currently stored snapshots.

Syntax

```
TT_STATS.DROP_SNAPSHOTS_RANGE (
   snapshot_low      IN BINARY_INTEGER,
   snapshot_high     IN BINARY_INTEGER);

TT_STATS.DROP_SNAPSHOTS_RANGE (
   ts_old            IN TIMESTAMP(6),
   ts_new            IN TIMESTAMP(6));
```

Parameters

Table 10-3 DROP_SNAPSHOTS_RANGE Procedure Parameters

Parameter	Description
snapshot_low	Snapshot ID for the start of the range of snapshots to delete
snapshot_high	Snapshot ID for the end of the range of snapshots to delete
ts_old	Timestamp for the start of the range of snapshots to delete
ts_new	Timestamp for the end of the range of snapshots to delete

Usage Notes

- Specify 0 (zero) for both input parameters to drop all snapshots.
- It is permissible for *snapshot_low* to be greater than *snapshot_high*. The range of snapshots from the lower value through the higher value are still deleted.
- Similarly, it is permissible for *ts_new* to be an older timestamp than *ts_old*.

Examples

This example specifies snapshot IDs, dropping the snapshots with IDs of 1, 2, and 3.

```
call tt_stats.drop_snapshots_range(1,3);
```

GENERATE_REPORT_HTML Procedure

This procedure uses the data from two specified snapshots of TimesTen metrics to produce a report in HTML format with information for each metric, such as rate of change or start and end values.

Reports include a summary of memory usage, connections, and load profile, followed by metrics (as applicable) for SQL statements, transactions, PL/SQL memory, replication, logs and log holds, checkpoints, cache groups, latches, locks, XLA, and TimesTen connection attributes.

For a detailed example of the HTML reports that are produced, see ttStats in *Oracle TimesTen In-Memory Database Reference*.

Also see GENERATE_REPORT_TEXT Procedure.

Note:

- You can use the SHOW_SNAPSHOTS Procedures to display the IDs and timestamps of all currently stored snapshots.
- Use snapshots taken at the same capture level. See CAPTURE_SNAPSHOT Procedure and Function.
- The reports are similar in nature to Oracle Automatic Workload Repository (AWR) reports.
- For `SYSTEMSTATS` metrics, only those within the specified capture level have meaningful accumulated values. `SYSTEMSTATS` metrics outside of the specified level have a value of 0 (zero).

Syntax

```
TT_STATS.GENERATE_REPORT_HTML (
   snapshot_id1     IN  BINARY_INTEGER,
   snapshot_id2     IN  BINARY_INTEGER,
   report           OUT TT_STATS.REPORT_TABLE);
```

Parameters

Table 10-4 GENERATE_REPORT_HTML Procedure Parameters

Parameter	Description
snapshot_id1	ID of the first snapshot to analyze
snapshot_id2	ID of the second snapshot to analyze
report	An associative array (index-by table) containing the HTML-formatted report Each row is of type `VARCHAR2(32767)`. The application can output the report contents line-by-line as desired.

Usage Notes

- You can enter the snapshot IDs in either order. The procedure determines which is the earlier.

GENERATE_REPORT_TEXT Procedure

This procedure analyzes and compares two specified snapshots of TimesTen metrics and produces a report in plain text format with information for each metric, such as rate of change or start and end values.

Reports include a summary of memory usage, connections, and load profile, followed by metrics (as applicable) for SQL statements, transactions, PL/SQL memory, replication, logs and log holds, checkpoints, cache groups, latches, locks, XLA, and TimesTen connection attributes.

Also see GENERATE_REPORT_HTML Procedure.

Note:

- You can use the SHOW_SNAPSHOTS Procedures to display the IDs (and timestamps) of all currently stored snapshots.
- Use snapshots taken at the same capture level. See CAPTURE_SNAPSHOT Procedure and Function.
- The reports are similar in nature to Oracle Automatic Workload Repository (AWR) reports.
- For `SYSTEMSTATS` metrics, only those within the specified capture level have meaningful accumulated values. `SYSTEMSTATS` metrics outside of the specified level have a value of 0 (zero).

Syntax

```
TT_STATS.GENERATE_REPORT_TEXT (
   snapshot_id1     IN  BINARY_INTEGER,
   snapshot_id2     IN  BINARY_INTEGER,
   report           OUT TT_STATS.REPORT_TABLE);
```

Parameters

Table 10-5 GENERATE_REPORT_TEXT Procedure Parameters

Parameter	Description
snapshot_id1	ID of the first snapshot to analyze
snapshot_id2	ID of the second snapshot to analyze
report	An associative array (index-by table) containing the plain-text-formatted report Each row is of type `VARCHAR2(32767)`. The application can output the report contents line-by-line as desired.

Usage Notes

- You can enter the snapshot IDs in either order. The procedure determines which is the earlier.

GET_CONFIG Procedures

Either procedure retrieves the value of a specified `TT_STATS` configuration parameter or the values of all configuration parameters. The version without the `OUT` parameter sends the information to the standard output.

Syntax

```
TT_STATS.GET_CONFIG (
   name      IN   VARCHAR2 DEFAULTED);
```

```
TT_STATS.GET_CONFIG (
   name      IN   VARCHAR2 DEFAULTED,
   params    OUT  TT_STATS.REPORT_TABLE);
```

Parameters

Table 10-6 GET_CONFIG Procedure Parameters

Parameter	Description
name	Name of a TT_STATS configuration parameter whose value you want to retrieve In this release, the following TT_STATS parameters are supported. • MAX_SNAPSHOT_COUNT: This is the maximum number of snapshots that can be stored. • MAX_SNAPSHOT_RETENTION_SIZE: This is the maximum total size of all stored snapshots, in MB. If no parameter name is specified (*name* is empty), the values of all configuration parameters are displayed. Also see SET_CONFIG Procedure.
params	An associative array (index-by table) containing the value of the specified TT_STATS parameter or values of all parameters Each row is of type VARCHAR2(32767).

SET_CONFIG Procedure

This procedure sets a specified value for a specified TT_STATS configuration parameter.

Syntax

```
TT_STATS.SET_CONFIG (
   name      IN  VARCHAR2,
   value     IN  BINARY_INTEGER);
```

Parameters

Table 10-7 SET_CONFIG Procedure Parameters

Parameter	Description
name	Name of the TT_STATS configuration parameter to set In this release, the following TT_STATS parameters are supported: • MAX_SNAPSHOT_COUNT: This is the maximum number of snapshots that can be stored. The default value is 256. • MAX_SNAPSHOT_RETENTION_SIZE: This is the maximum total size of all stored snapshots, in MB. The default value is 256 MB. An error is issued if either limit is exceeded, and the snapshot capture fails. Also see GET_CONFIG Procedures.
value	Value to set for the specified parameter

Usage Notes

- The scope of these settings is global, affecting all connections to the database.

Examples

Specify a limit of 500 stored snapshots:

```
call tt_stats.set_config('MAX_SNAPSHOT_COUNT', 500);
```

SHOW_SNAPSHOTS Procedures

This procedure shows the IDs and timestamps of all snapshots of TimesTen metrics currently stored in the database.

The version without the `OUT` parameter sends the information to the standard output.

Syntax

```
TT_STATS.SHOW_SNAPSHOTS;

TT_STATS.SHOW_SNAPSHOTS (
   resultset    OUT TT_STATS.REPORT_TABLE);
```

Parameters

Table 10-8 SHOW_SNAPSHOTS Procedure Parameters

Parameter	Description
resultset	An associative array (index-by table) with pairs of data showing the ID and timestamp of each currently stored snapshot Each row is of type `VARCHAR2(32767)`.

11
UTL_FILE

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

`UTL_FILE` is limited to the directory *`timesten_home`*`/plsql/utl_file_temp`.

Access does not extend to subdirectories of this directory. In addition, access is subject to file system permission checking. The instance administrator can grant `UTL_FILE` access to specific users as desired. Users can reference this `UTL_FILE` directory by using the string 'UTL_FILE_TEMP' for the location parameter in `UTL_FILE` subprograms. This predefined string is used in the same way as directory object names in Oracle Database.

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

For TimesTen direct connections, the application owner is owner of the file. For client/server connections, the server owner is owner of the file.

`UTL_FILE_DIR` access is not supported in TimesTen.

Tip:

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

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

- The privileges needed to access files are operating system specific. UTL_FILE privileges give you read and write access to all files within the UTL_FILE directory, but not in subdirectories.
- Attempting to apply invalid UTL_FILE options results in unpredictable behavior.

Operational Notes

UTL_FILE is limited to the directory timesten_home/plsql/utl_file_temp. Access does not extend to subdirectories of this directory. In addition, access is subject to file system permission checking. The instance administrator can grant UTL_FILE access to specific users as desired. Users can reference this UTL_FILE directory by using the string 'UTL_FILE_TEMP' for the location parameter in UTL_FILE subprograms. This predefined string is used in the same way as directory object names in Oracle Database.

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 valid 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 GET_LINE specifies the requested number of bytes of character data. The number of bytes actually returned to the user is the least of the following:

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

When data encoded in one character set is read and Globalization Support is informed (such as through 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 Linux or 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, 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, call FCLOSE to complete any output and free resources associated with the file.

Exceptions

This section describes exceptions that are thrown by UTL_FILE subprograms.

> **Note:**
>
> In addition to the exceptions listed here, procedures and functions in UTL_FILE can raise predefined PL/SQL exceptions such as NO_DATA_FOUND or VALUE_ERROR. Refer to Predefined Exceptions in *Oracle Database PL/SQL Language Reference* for information about those.

Table 11-1 UTL_FILE package exceptions

Exception Name	Description
ACCESS_DENIED	Permission to access to the file location is denied.
CHARSETMISMATCH	A file is opened using FOPEN_NCHAR, but later I/O operations use non-NCHAR procedures such as PUTF or GET_LINE. Or a file is opened using FOPEN, but later I/O operations use NCHAR functions such as PUTF_NCHAR or GET_LINE_NCHAR.
DELETE_FAILED	Requested file delete operation failed.
FILE_OPEN	Requested operation failed because the file is open.
INTERNAL_ERROR	There was an unspecified PL/SQL error.
INVALID_FILEHANDLE	File handle is invalid.
INVALID_FILENAME	The *filename* parameter is invalid.
INVALID_MAXLINESIZE	The *max_linesize* value for FOPEN is out of range. It should be within the range 1 to 32767.
INVALID_MODE	The *open_mode* parameter in FOPEN is invalid.
INVALID_OFFSET	Caused by one of the following: • ABSOLUTE_OFFSET is NULL and RELATIVE_OFFSET is NULL. • ABSOLUTE_OFFSET is less than 0. • Either offset caused a seek past the end of the file.
INVALID_OPERATION	File could not be opened or operated on as requested.
INVALID_PATH	File location or name is invalid.

Table 11-1 (Cont.) UTL_FILE package exceptions

Exception Name	Description
LENGTH_MISMATCH	Length mismatch for CHAR or RAW data.
READ_ERROR	Operating system error occurred during the read operation.
RENAME_FAILED	Requested file rename operation failed.
WRITE_ERROR	Operating system error occurred during the write operation.

Examples

Example 1: GET_LINE

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 is 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 is 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 is 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: PUTF

This 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, '\nHello, world!\nI come from %s with %s.\n',
my_world,
                         'greetings for all earthlings');
   utl_file.fflush(handle);
   utl_file.fclose(handle);
end;
```

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: GET_RAW

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.

```
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:

```
hello world!
```

The example produces output as follows:

```
Command> begin
            getraw('u12345.tmp');
         end;
         /
This is the raw data:
68656C6C6F20776F726C64210A

PL/SQL procedure successfully completed.
```

Data Structures

The `UTL_FILE` package defines the following 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.

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

Fields

Table 11-2 FILE_TYPE Fields

Field	Description
id	Indicates the internal file handle number (numeric value).

Table 11-2 (Cont.) FILE_TYPE Fields

Field	Description
datatype	Indicates whether the file is a `CHAR` file, `NCHAR` file, or other (binary).
byte_mode	Indicates whether the file was open as a binary file or as a text file.

Tip:

Oracle Database 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:

- 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.

UTL_FILE Subprograms

Table 11-3 summarizes the `UTL_FILE` subprograms, followed by a full description of each subprogram.

Table 11-3 UTL_FILE Subprograms

Subprogram	Description
FCLOSE Procedure	Closes a file.
FCLOSE_ALL Procedure	Closes all open file handles.
FCOPY Procedure	Copies a contiguous portion of a file to a newly created file.
FFLUSH Procedure	Physically writes all pending output to a file.
FGETATTR Procedure	Reads and returns the attributes of a file.
FGETPOS Function	Returns the current relative offset position (in bytes) within a file, in bytes.
FOPEN Function	Opens a file for input or output.
FOPEN_NCHAR Function	Opens a file in Unicode for input or output.
FREMOVE Procedure	Deletes a file if you have sufficient privileges.

Table 11-3 (Cont.) UTL_FILE Subprograms

Subprogram	Description
FRENAME Procedure	Renames an existing file to a new name, similar to the UNIX `mv` function.
FSEEK Procedure	Adjusts the file pointer forward or backward within the file by the number of bytes specified.
GET_LINE Procedure	Reads text from an open file.
GET_LINE_NCHAR Procedure	Reads text in Unicode from an open file.
GET_RAW Procedure	Reads a `RAW` string value from a file and adjusts the file pointer ahead by the number of bytes read.
IS_OPEN Function	Determines if a file handle refers to an open file.
NEW_LINE Procedure	Writes one or more operating system-specific line terminators to a file.
PUT Procedure	Writes a string to a file.
PUT_LINE Procedure	Writes a line to a file, and so appends an operating system-specific line terminator.
PUT_LINE_NCHAR Procedure	Writes a Unicode line to a file.
PUT_NCHAR Procedure	Writes a Unicode string to a file.
PUT_RAW Procedure	Accepts as input a `RAW` data value and writes the value to the output buffer.
PUTF Procedure	This is equivalent to `PUT` but with formatting.
PUTF_NCHAR Procedure	This is equivalent to `PUT_NCHAR` but with formatting.

FCLOSE Procedure

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

Syntax

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

Parameters

Table 11-4 FCLOSE Procedure Parameters

Parameter	Description
file	Active file handle returned by an `FOPEN` or `FOPEN_NCHAR` call

Exceptions

Refer to Exceptions for information about these exceptions.

```
INVALID_FILEHANDLE
WRITE_ERROR
```

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

Examples

See Examples .

FCLOSE_ALL Procedure

This procedure closes all open file handles for the session. This is useful as an emergency cleanup procedure, such as after a PL/SQL program exits on an exception.

Syntax

```
UTL_FILE.FCLOSE_ALL;
```

Usage Notes

`FCLOSE_ALL` does not alter the state of the open file handles held by the user. Therefore, 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

Refer to 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 11-5 FCOPY Procedure Parameters

Parameters	Description
src_location	Directory location of the source file
src_filename	Source file to be copied
dest_location	Destination directory where the destination file is created
dest_filename	Destination file created from the source file
start_line	Line number at which to begin copying The default is `1` for the first line.
end_line	Line number at which to stop copying The default is `NULL`, signifying end of file.

Exceptions

Refer to Exceptions.

```
INVALID_FILENAME
INVALID_PATH
INVALID_OPERATION
INVALID_OFFSET
READ_ERROR
WRITE_ERROR
```

FFLUSH Procedure

`FFLUSH` physically writes pending data to the file identified by the file handle.

Typically, data 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

```
UTL_FILE.FFLUSH (
   file  IN UTL_FILE.FILE_TYPE);
```

Parameters

Table 11-6 FFLUSH Procedure Parameters

Parameters	Description
file	Active file handle returned by an `FOPEN` or `FOPEN_NCHAR` call

Exceptions

Refer to Exceptions.

```
INVALID_FILEHANDLE
INVALID_OPERATION
WRITE_ERROR
```

Examples

See Examples .

FGETATTR Procedure

This procedure reads and returns the attributes of a file.

Syntax

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

Parameters

Table 11-7 FGETATTR Procedure Parameters

Parameters	Description
location	Location of the source file
filename	Name of the file to be examined
fexists	A `BOOLEAN` for whether the file exists
file_length	Length of the file in bytes, or `NULL` if file does not exist
block_size	File system block size in bytes, or `NULL` if file does not exist

Exceptions

Refer to Exceptions.

```
INVALID_PATH
INVALID_FILENAME
INVALID_OPERATION
READ_ERROR
ACCESS_DENIED
```

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 11-8 FGETPOS Function Parameters

Parameters	Description
file	Active file handle returned by an FOPEN or FOPEN_NCHAR call

Return Value

The relative offset position for an open file, in bytes, or 0 for the beginning of the file

Exceptions

Refer to Exceptions.

```
INVALID_FILEHANDLE
INVALID_OPERATION
READ_ERROR
```

An INVALID_FILEHANDLE exception is raised if the file is not open. An INVALID_OPERATION exception is raised if the file was opened for byte mode operations.

FOPEN Function

This function opens a file. You can specify the maximum line size and have a maximum of 50 files open simultaneously.

Also see FOPEN_NCHAR Function.

Syntax

```
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 11-9 FOPEN Function Parameters

Parameter	Description
location	Directory location of file
filename	File name, including extension (file type), without directory path If a directory path is given as a part of the file name, it is ignored by FOPEN. On Linux or UNIX, the file name cannot end with a slash, "/".
open_mode	Mode in which the file was opened: • r - Read text mode • w - Write text mode • a - Append text mode • rb - Read byte mode • wb - Write byte mode • ab - Append byte mode If you try to open a file specifying 'a' or 'ab' for *open_mode* but the file does not exist, the file is created in WRITE mode.
max_linesize	Maximum number of characters for each line, including the newline character, for this file The minimum value is 1 and the maximum is 32767. If this is unspecified, TimesTen supplies a default value of 1024.

Return Value

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.

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 valid file name on the system, and the directory must be accessible. Any subdirectories of utl_file_temp are not accessible.

Exceptions

Refer to Exceptions.

```
INVALID_PATH
INVALID_MODE
INVALID_OPERATION
INVALID_MAXLINESIZE
```

Examples

See Examples .

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 UTF-8 (depending on the national character set of the database), the contents of the file are always read and written in UTF-8. `UTL_FILE` converts between UTF-8 and `AL16UTF16` as necessary.

Also see FOPEN Function.

Syntax

```
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 11-10 FOPEN_NCHAR Function Parameters

Parameter	Description
location	Directory location of file
filename	File name, including extension
open_mode	Open mode: `r`, `w`, `a`, `rb`, `wb`, or `ab` (as documented for `FOPEN`)
max_linesize	Maximum number of characters for each line, including the newline character, for this file The minimum value is 1 and the maximum is 32767. If this is unspecified, TimesTen supplies a default value of 1024.

Return Value

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.

Exceptions

Refer to Exceptions.

```
INVALID_PATH
INVALID_MODE
```

```
INVALID_OPERATION
INVALID_MAXLINESIZE
```

FREMOVE Procedure

This procedure deletes a file if you have sufficient privileges.

Syntax

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

Parameters

Table 11-11 FREMOVE Procedure Parameters

Parameters	Description
location	Directory location of the file
filename	Name of the file to be deleted

Usage Notes

This procedure does not verify privileges before deleting a file. The operating system verifies file and directory permissions.

Exceptions

Refer to Exceptions.

```
INVALID_PATH
INVALID_FILENAME
INVALID_OPERATION
ACCESS_DENIED
DELETE_FAILED
```

FRENAME 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 11-12 FRENAME Procedure Parameters

Parameters	Description
src_location	Directory location of the source file
src_filename	Source file to be renamed
dest_location	Destination directory of the destination file
dest_filename	New name of the file
overwrite	Whether it is permissible to overwrite an existing file in the destination directory (default `FALSE`)

Usage Notes

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

Exceptions

Refer to Exceptions.

```
INVALID_PATH
INVALID_FILENAME
RENAME_FAILED
ACCESS_DENIED
```

FSEEK Procedure

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

Syntax

```
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 11-13 FSEEK Procedure Parameters

Parameters	Description
file	Active file handle returned by an `FOPEN` or `FOPEN_NCHAR` call
absolute_offset	Absolute location to which to seek, in bytes (default `NULL`)
relative_offset	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`.

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.

Exceptions

Refer to Exceptions.

```
INVALID_FILEHANDLE
INVALID_OPERATION
READ_ERROR
INVALID_OFFSET
```

INVALID_OPERATION is raised if the file was opened for byte-mode operations.
INVALID_OFFSET is raised if the end of the file is reached before the number of bytes specified.

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

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

Parameters

Table 11-14 GET_LINE Procedure Parameters

Parameters	Description
file	Active file handle returned by an FOPEN call
buffer	Data buffer to receive the line read from the file
len	Number of bytes read from the file If NULL (default), TimesTen supplies the value of *max_linesize* from FOPEN.

Usage Notes

- 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. Also see GET_LINE_NCHAR Procedure.

Exceptions

Refer to Exceptions.

```
INVALID_FILEHANDLE
INVALID_OPERATION
READ_ERROR
CHARSETMISMATCH
NO_DATA_FOUND
VALUE_ERROR
```

INVALID_OPERATION is thrown if the file was not opened for read mode (mode r) or was opened for byte-mode operations. CHARSETMISMATCH is thrown if FOPEN_NCHAR was used instead of FOPEN to open the file. NO_DATA_FOUND is thrown if no text was read due to end of file. VALUE_ERROR is thrown if the line does not fit into the buffer. (NO_DATA_FOUND and VALUE_ERROR are predefined PL/SQL exceptions.)

Examples

See Examples .

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 UTF-8 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 performs standard implicit conversion from NVARCHAR2 after the text is read.

Also see GET_LINE Procedure.

Syntax

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

Parameters

Table 11-15 GET_LINE_NCHAR Procedure Parameters

Parameters	Description
file	Active file handle returned by an FOPEN_NCHAR call The file must be open for reading (mode r).
buffer	Data buffer to receive the line read from the file
len	The number of bytes read from the file If NULL (default), TimesTen supplies the value of *max_linesize* from FOPEN_NCHAR.

Exceptions

Refer to Exceptions.

```
INVALID_FILEHANDLE
INVALID_OPERATION
READ_ERROR
CHARSETMISMATCH
NO_DATA_FOUND
VALUE_ERROR
```

INVALID_OPERATION is thrown if the file was not opened for read mode (mode r) or was opened for byte-mode operations. NO_DATA_FOUND is thrown if no text was read due to end of file. VALUE_ERROR is thrown if the line does not fit into the buffer. CHARSETMISMATCH is thrown if the file was opened by FOPEN instead of FOPEN_NCHAR. (NO_DATA_FOUND and VALUE_ERROR are predefined PL/SQL exceptions.)

GET_RAW Procedure

This procedure reads a RAW string value from a file and adjusts the file pointer ahead by the number of bytes read. It ignores line terminators.

Syntax

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

Parameters

Table 11-16 GET_RAW Function Parameters

Parameters	Description
file	Active file handle returned by an FOPEN or FOPEN_NCHAR call
buffer	The RAW data

Table 11-16 (Cont.) GET_RAW Function Parameters

Parameters	Description
len	Number of bytes read from the file If NULL (default), *len* is assumed to be the maximum length of RAW.

Exceptions

Refer to Exceptions.

```
INVALID_FILEHANDLE
INVALID_OPERATION
READ_ERROR
LENGTH_MISMATCH
NO_DATA_FOUND
```

(NO_DATA_FOUND is a predefined PL/SQL exception.)

Examples

See Examples .

IS_OPEN Function

This function tests a file handle to see if it identifies an open file. It reports only whether a file handle represents a file that has been opened, but not yet closed. It does not guarantee you can use the file without error.

Syntax

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

Parameters

Table 11-17 IS_OPEN Function Parameters

Parameter	Description
file	Active file handle returned by an FOPEN or FOPEN_NCHAR call

Return Value

TRUE if the file is open, or FALSE if not

Exceptions

Refer to Exceptions.

```
INVALID_FILEHANDLE
```

NEW_LINE Procedure

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

Syntax

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

Parameters

Table 11-18 NEW_LINE Procedure Parameters

Parameters	Description
file	Active file handle returned by an `FOPEN` or `FOPEN_NCHAR` call
lines	Number of line terminators to be written to the file

Exceptions

Refer to 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 `PUT_LINE` to write a complete line with a line terminator. Also see PUT_NCHAR Procedure.

Syntax

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

Parameters

Table 11-19 PUT Procedure Parameters

Parameters	Description
file	Active file handle returned by an FOPEN_NCHAR call The file must be open for writing (mode w).
buffer	Buffer that contains the text to be written to the file

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

Refer to Exceptions.

```
INVALID_FILEHANDLE
INVALID_OPERATION
WRITE_ERROR
CHARSETMISMATCH
```

INVALID_OPERATION is thrown if the file was not opened using mode w or a (write or append). CHARSETMISMATCH is thrown if FOPEN_NCHAR was used instead of FOPEN to open the file.

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.

Syntax

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

Parameters

Table 11-20 PUT_LINE Procedure Parameters

Parameters	Description
file	Active file handle returned by an FOPEN call
buffer	Text buffer that contains the lines to be written to the file
autoflush	Flag for flushing the buffer to the file system after the write

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

Refer to Exceptions.

```
INVALID_FILEHANDLE
INVALID_OPERATION
WRITE_ERROR
CHARSETMISMATCH
```

INVALID_OPERATION is thrown if the file was opened for byte-mode operations.
CHARSETMISMATCH is thrown if FOPEN_NCHAR was used instead of FOPEN to open the file.

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. Also see PUT_LINE Procedure.

Syntax

```
UTL_FILE.PUT_LINE_NCHAR (
   file    IN UTL_FILE.FILE_TYPE,
   buffer  IN NVARCHAR2);
```

Parameters

Table 11-21 PUT_LINE_NCHAR Procedure Parameters

Parameters	Description
file	Active file handle returned by an FOPEN_NCHAR call The file must be open for writing (mode w).

Table 11-21 (Cont.) PUT_LINE_NCHAR Procedure Parameters

Parameters	Description
buffer	Text buffer that contains the lines to be written to the file

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

Refer to Exceptions.

```
INVALID_FILEHANDLE
INVALID_OPERATION
WRITE_ERROR
CHARSETMISMATCH
```

INVALID_OPERATION is thrown if the file was opened for byte-mode operations. CHARSETMISMATCH is thrown if FOPEN was used instead of FOPEN_NCHAR to open the file.

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 is written in the UTF-8 character set. The expected buffer data type is NVARCHAR2. If a variable of another data type is specified, PL/SQL performs implicit conversion to NVARCHAR2 before writing the text.

Also see PUT Procedure.

Syntax

```
UTL_FILE.PUT_NCHAR (
   file      IN UTL_FILE.FILE_TYPE,
   buffer    IN NVARCHAR2);
```

Parameters

Table 11-22 PUT_NCHAR Procedure Parameters

Parameters	Description
file	Active file handle returned by an FOPEN_NCHAR call
buffer	Buffer that contains the text to be written to the file

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

Refer to Exceptions.

```
INVALID_FILEHANDLE
INVALID_OPERATION
WRITE_ERROR
CHARSETMISMATCH
```

INVALID_OPERATION is thrown if the file was not opened using mode w or a (write or append). CHARSETMISMATCH is thrown if the file was opened by FOPEN instead of FOPEN_NCHAR.

PUT_RAW Procedure

This procedure accepts as input a RAW data value and writes the value to the output buffer.

Syntax

```
UTL_FILE.PUT_RAW (
   file       IN utl_file.file_type,
   buffer     IN RAW,
   autoflush  IN BOOLEAN DEFAULT FALSE);
```

Parameters

Table 11-23 PUT_RAW Procedure Parameters

Parameters	Description
file	Active file handle returned by an FOPEN or FOPEN_NCHAR call
buffer	The RAW data written to the buffer
autoflush	Flag to perform a flush after writing the value to the output buffer (default is FALSE)

Usage Notes

You can request an automatic flush of the buffer by setting *autoflush* 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.

Exceptions

Refer to Exceptions.

```
INVALID_FILEHANDLE
INVALID_OPERATION
WRITE_ERROR
```

PUTF Procedure

This procedure is a formatted `PUT` procedure. It works like a limited `printf()`.

Also see PUTF_NCHAR Procedure.

Syntax

```
UTL_FILE.PUTF (
   file    IN UTL_FILE.FILE_TYPE,
   format  IN VARCHAR2,
   [arg1   IN VARCHAR2  DEFAULT NULL,
   . . .
   arg5    IN VARCHAR2  DEFAULT NULL]);
```

Parameters

Table 11-24 PUTF Procedure Parameters

Parameters	Description
file	Active file handle returned by an `FOPEN` call
format	Format string that can contain text and the formatting characters `\n` and `%s`
arg1..arg5	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, an empty string is substituted for each `%s` for which there is no argument.

Usage Notes

The format string can contain any text, but the character sequences `%s` and `\n` have special meaning.

Character sequence	Meaning
`%s`	Substitute this sequence with the string value of the next argument in the argument list.
`\n`	Substitute with the appropriate platform-specific line terminator.

Exceptions

Refer to Exceptions.

```
INVALID_FILEHANDLE
INVALID_OPERATION
WRITE_ERROR
CHARSETMISMATCH
```

`INVALID_OPERATION` is thrown if the file was opened for byte-mode operations. `CHARSETMISMATCH` is thrown if `FOPEN_NCHAR` was used instead of `FOPEN` to open the file.

Examples

See Examples .

PUTF_NCHAR Procedure

Using `PUTF_NCHAR`, you can write a text file in Unicode instead of in the database character set.

This procedure is the formatted version of the PUT_NCHAR Procedure.

It accepts a format string with formatting elements `\n` and `%s`, and up to five arguments to be substituted for consecutive occurrences 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 performs implicit conversion to `NVARCHAR2` before formatting the text. Formatted text is written in the UTF-8 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 11-25 PUTF_NCHAR Procedure Parameters

Parameters	Description
file	Active file handle returned by an `FOPEN_NCHAR` call The file must be open for reading (mode `r`).
format	Format string that can contain text and the format characters `\n` and `%s`

Table 11-25 (Cont.) PUTF_NCHAR Procedure Parameters

Parameters	Description
arg1..arg5	From one to five operational argument strings Argument strings are substituted, in order, for the `%s` format characters in the format string. If there are more format characters in the format string than there are arguments, an empty string is substituted for each `%s` for which there is no argument.

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

Refer to Exceptions.

```
INVALID_FILEHANDLE
INVALID_OPERATION
WRITE_ERROR
CHARSETMISMATCH
```

`INVALID_OPERATION` is thrown if the file was opened for byte-mode operations. `CHARSETMISMATCH` is thrown if the file was opened by `FOPEN` instead of `FOPEN_NCHAR`.

12
UTL_IDENT

The UTL_IDENT package indicates which database or client PL/SQL is running in, such as TimesTen versus Oracle Database, 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

This section contains topics that relate to using the UTL_IDENT package.

Overview

The UTL_IDENT package indicates whether PL/SQL is running on TimesTen, an Oracle database client, an Oracle database server, or Oracle Forms. Each of these has its own version of UTL_IDENT with appropriate settings for the constants.

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.

```
$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
```

Also see Examples.

Security Model

The UTL_IDENT package runs as the package owner SYS. The public synonym UTL_IDENT and EXECUTE permission on this package are granted to PUBLIC.

Constants

The UTL_IDENT package uses the constants in Table 12-1, shown here with the settings in a TimesTen database.

Table 12-1 UTL_IDENT Constants

Constant	Type	Value	Description
IS_ORACLE_SERVER	BOOLEAN	FALSE	PL/SQL is running in Oracle Database.
IS_ORACLE_CLIENT	BOOLEAN	FALSE	PL/SQL is running in Oracle Client.
IS_ORACLE_FORMS	BOOLEAN	FALSE	PL/SQL is running in Oracle Forms.
IS_TIMESTEN	BOOLEAN	TRUE	PL/SQL is running in TimesTen.

Examples

This example shows output from a script that creates and executes a function IS_CLOB_SUPPORTED that uses the UTL_IDENT and TT_DB_VERSION packages to provide information about the database being used.

The function uses UTL_IDENT to determine whether the database is TimesTen, then uses TT_DB_VERSION to determine the TimesTen version. VER_LE_1121 is TRUE for TimesTen 11.2.1 releases and FALSE for TimesTen 11*g* Release 2 (11.2.2) and higher releases. In the example, because VER_LE_1121 is determined to be FALSE, then it can be assumed that this is a TimesTen 11*g* Release 2 (11.2.2) or higher release (presumably Release 22.1) and therefore LOBs are supported by TimesTen Classic. The example then creates a table with a CLOB column and shows DESCRIBE output of the table.

```
create or replace function is_clob_supported return boolean
as
begin
$if utl_ident.is_oracle_server
$then
 return true;
$elsif utl_ident.is_timesten
$then
 $if tt_db_version.ver_le_1121
 $then
  return false; -- CLOB datatype was introduced in TimesTen 11g
Release 2 (11.2.2)
 $else
  return true;
 $end
```

```
$end
end;

Function created.

show errors;
No errors.

begin
  if is_clob_supported
  then
    execute immediate 'create table mytab (mydata clob)';
  else
    execute immediate 'create table mytab (mydata varchar2(4000000))';
  end if;
end;

PL/SQL procedure successfully completed.

describe mytab;

Table MYSCHEMA.MYTAB:
  Columns:
    MYDATA                          CLOB

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

(Output is shown after running the commands from a SQL script.)

13
UTL_RAW

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

This chapter contains the following topics:

- Using UTL_RAW
  - Overview
  - Operational notes
- UTL_RAW Subprograms

Using UTL_RAW

- Overview
- Operational Notes

Overview

This package is necessary because 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` 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`.

Operational Notes

`UTL_RAW` allows a `RAW` record to be composed of many elements. When the `RAW` data type is used, character set conversion is not 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.

UTL_RAW Subprograms

Table 13-1 summarizes the `UTL_RAW` subprograms, followed by a full description of each subprogram.

Table 13-1 UTL_RAW Package Subprograms

Subprogram	Description
BIT_AND Function	Performs bitwise logical AND of two RAW values and returns the resulting RAW.
BIT_COMPLEMENT Function	Performs bitwise logical COMPLEMENT of a RAW value and returns the resulting RAW.
BIT_OR Function	Performs bitwise logical OR of two RAW values and returns the resulting RAW.
BIT_XOR Function	Performs bitwise logical XOR ("exclusive or") of two RAW values and returns the resulting RAW.
CAST_FROM_BINARY_DOUBLE Function	Returns the RAW binary representation of a BINARY_DOUBLE value.
CAST_FROM_BINARY_FLOAT Function	Returns the RAW binary representation of a BINARY_FLOAT value.
CAST_FROM_BINARY_INTEGER Function	Returns the RAW binary representation of a BINARY_INTEGER value.
CAST_FROM_NUMBER Function	Returns the RAW binary representation of a NUMBER value.
CAST_TO_BINARY_DOUBLE Function	Casts the RAW binary representation of a BINARY_DOUBLE value into a BINARY_DOUBLE.
CAST_TO_BINARY_FLOAT Function	Casts the RAW binary representation of a BINARY_FLOAT value into a BINARY_FLOAT.
CAST_TO_BINARY_INTEGER Function	Casts the RAW binary representation of a BINARY_INTEGER value into a BINARY_INTEGER.
CAST_TO_NUMBER Function	Casts the RAW binary representation of a NUMBER value into a NUMBER.
CAST_TO_NVARCHAR2 Function	Casts a RAW value into an NVARCHAR2 value.
CAST_TO_RAW Function	Casts a VARCHAR2 value into a RAW value.
CAST_TO_VARCHAR2 Function	Casts a RAW value into a VARCHAR2 value.
COMPARE Function	Compares two RAW values.
CONCAT Function	Concatenates up to 12 RAW values into a single RAW.
CONVERT Function	Converts a RAW value from one character set to another and returns the resulting RAW.
COPIES Function	Copies a RAW value a specified number of times and returns the concatenated RAW value.
LENGTH Function	Returns the length in bytes of a RAW value.
OVERLAY Function	Overlays the specified portion of a target RAW value with an overlay RAW value, starting from a specified byte position and proceeding for a specified number of bytes.
REVERSE Function	Reverses a byte-sequence in a RAW value.
SUBSTR Function	Returns a substring of a RAW value for a specified number of bytes from a specified starting position.

Table 13-1 (Cont.) UTL_RAW Package Subprograms

Subprogram	Description
TRANSLATE Function	Translates the specified bytes from an input `RAW` value according to the bytes in a specified translation `RAW` value.
TRANSLITERATE Function	Converts the specified bytes from an input `RAW` value according to the bytes in a specified transliteration `RAW` value.
XRANGE Function	Returns a `RAW` value containing the succession of one-byte encodings beginning and ending with the specified byte-codes.

Note:

- 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

```
UTL_RAW.BIT_AND (
   r1 IN RAW,
   r2 IN RAW)
RETURN RAW;
```

Parameters

Table 13-2 BIT_AND Function Parameters

Parameter	Description
r1	First `RAW` value for `AND` operation
r2	Second `RAW` value for `AND` operation

Return Value

Result of the `AND` operation, or `NULL` if either input value is `NULL`

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;
```

Parameters

Table 13-3 BIT_COMPLEMENT Function Parameters

Parameter	Description
r	RAW value for COMPLEMENT operation

Return Value

Result of the COMPLEMENT operation, or NULL if the input value is NULL

BIT_OR Function

This function performs bitwise logical OR of two supplied RAW values and returns the resulting RAW.

Syntax

```
UTL_RAW.BIT_OR (
   r1 IN RAW,
   r2 IN RAW)
  RETURN RAW;
```

Parameters

Table 13-4 BIT_OR Function Parameters

Parameters	Description
r1	First RAW value for OR operation
r2	Second RAW value for OR operation

Return Value

Result of the OR operation, or NULL if either input value is NULL

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 XOR ("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;
```

Parameters

Table 13-5 BIT_XOR Function Parameters

Parameter	Description
r1	First RAW value for XOR operation
r2	Second RAW value for XOR operation

Return Value

Result of the XOR operation, or NULL if either input value is NULL

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

```
UTL_RAW.CAST_FROM_BINARY_DOUBLE(
   n          IN BINARY_DOUBLE,
   endianess  IN BINARY_INTEGER DEFAULT 1)
RETURN RAW;
```

Parameters

Table 13-6 CAST_FROM_BINARY_DOUBLE Function Parameters

Parameter	Description
n	The BINARY_DOUBLE value
endianess	BINARY_INTEGER value indicating the endianess The function recognizes the defined constants big_endian, little_endian, and machine_endian. The default is big_endian.

Return Value

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
```

- 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.

Endianess	rb0	rb1	rb2	rb3	rb4	rb5	rb6	rb7
big_endian	db0	db1	db2	db3	db4	db5	db6	db7
little_endian	db7	db6	db5	db4	db3	db2	db1	db0

- 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

```
UTL_RAW.CAST_FROM_BINARY_FLOAT(
   n           IN BINARY_FLOAT,
```

```
   endianess  IN BINARY_INTEGER DEFAULT 1)
RETURN RAW;
```

Parameters

Table 13-7 CAST_FROM_BINARY_FLOAT Function Parameters

Parameter	Description
n	The BINARY_FLOAT value
endianess	BINARY_INTEGER value indicating the endianess The function recognizes the defined constants big_endian, little_endian, and machine_endian. The default is big_endian.

Return Value

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.

Endianess	rb0	rb1	rb2	rb3
big_endian	fbo	fb1	fb2	fb3
little_endian	fb3	fb2	fb1	fb0

- 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

```
UTL_RAW.CAST_FROM_BINARY_INTEGER (
   n          IN BINARY_INTEGER
   endianess  IN BINARY_INTEGER DEFAULT 1)
RETURN RAW;
```

Parameters

Table 13-8 CAST_FROM_BINARY_INTEGER Function Parameters

Parameter	Description
n	The BINARY_INTEGER value
endianess	BINARY_INTEGER value indicating the endianess The function recognizes the defined constants big_endian, little_endian, and machine_endian. The default is big_endian.

Return Value

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

```
UTL_RAW.CAST_FROM_NUMBER (
   n  IN NUMBER)
 RETURN RAW;
```

Parameters

Table 13-9 CAST_FROM_NUMBER Function Parameters

Parameter	Description
n	The NUMBER value

Return Value

RAW binary representation of the NUMBER value, or NULL if the input is NULL

CAST_TO_BINARY_DOUBLE Function

This function casts the RAW binary representation of a BINARY_DOUBLE value into a BINARY_DOUBLE value.

Syntax

```
UTL_RAW.CAST_TO_BINARY_DOUBLE (
   r          IN RAW
   endianess  IN BINARY_INTEGER DEFAULT 1)
RETURN BINARY_DOUBLE;
```

Parameters

Table 13-10 CAST_TO_BINARY_DOUBLE Function Parameters

Parameter	Description
r	RAW binary representation of a BINARY_DOUBLE value
endianess	BINARY_INTEGER value indicating the endianess The function recognizes the defined constants big_endian, little_endian, and machine_endian. The default is big_endian.

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.

Endianess	rb0	rb1	rb2	rb3	rb4	rb5	rb6	rb7
big_endian	db0	db1	db2	db3	db4	db5	db6	db7
little_endian	db7	db6	db5	db4	db3	db2	db1	db0

- 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 value.

Syntax

```
UTL_RAW.CAST_TO_BINARY_FLOAT (
   r          IN RAW
   endianess  IN BINARY_INTEGER DEFAULT 1)
RETURN BINARY_FLOAT;
```

Parameters

Table 13-11 CAST_TO_BINARY_FLOAT Function Parameters

Parameter	Description
r	RAW binary representation of a BINARY_FLOAT value
endianess	BINARY_INTEGER value indicating the endianess The function recognizes the defined constants big_endian, little_endian, and machine_endian. The default is big_endian.

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.

Endianess	rb0	rb1	rb2	rb3
big_endian	fbo	fb1	fb2	fb3
little_endian	fb3	fb2	fb1	fb0

- 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` value.

Syntax

```
UTL_RAW.CAST_TO_BINARY_INTEGER (
   r          IN RAW
   endianess  IN BINARY_INTEGER DEFAULT 1)
RETURN BINARY_INTEGER;
```

Parameters

Table 13-12 CAST_TO_BINARY_INTEGER Function Parameters

Parameter	Description
r	`RAW` binary representation of a `BINARY_INTEGER` value
endianess	`BINARY_INTEGER` value indicating the endianess The function recognizes the defined constants `big_endian`, `little_endian`, and `machine_endian`. The default is `big_endian`.

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` value.

Syntax

```
UTL_RAW.CAST_TO_NUMBER (
   r  IN RAW)
 RETURN NUMBER;
```

Parameters

Table 13-13 CAST_TO_NUMBER Function Parameters

Parameter	Description
r	`RAW` binary representation of a `NUMBER` value

Return Value

The NUMBER value, or NULL if the input is NULL

CAST_TO_NVARCHAR2 Function

This function casts a RAW value represented using some number of data bytes into an NVARCHAR2 value with that number of data bytes.

> **Note:**
>
> When casting to NVARCHAR2, the current Globalization Support character set is used for the characters within that NVARCHAR2 value.

Syntax

```
UTL_RAW.CAST_TO_NVARCHAR2 (
   r IN RAW)
RETURN NVARCHAR2;
```

Parameters

Table 13-14 CAST_TO_NVARCHAR2 Function Parameters

Parameter	Description
r	RAW value, without leading length field, to be changed to an NVARCHAR2 value

Return Value

Data converted from the input RAW value, or NULL if the input is NULL

CAST_TO_RAW Function

This function casts a VARCHAR2 value represented using some number of data bytes into a RAW value with that number of data bytes. The data itself is not modified in any way, but its data type is recast to a RAW data type.

Syntax

```
UTL_RAW.CAST_TO_RAW (
   c  IN VARCHAR2)
RETURN RAW;
```

Parameters

Table 13-15 CAST_TO_RAW Function Parameters

Parameter	Description
c	VARCHAR2 value to be changed to a RAW value

Return Values

Data converted from the input VARCHAR2 value, with the same byte-length as the input value but without a leading length field, or NULL if the input is NULL

CAST_TO_VARCHAR2 Function

This function casts a RAW value represented using some number of data bytes into a VARCHAR2 value with that number of data bytes.

> **Note:**
>
> When casting to VARCHAR2, the current Globalization Support character set is used for the characters within that VARCHAR2 value.

Syntax

```
UTL_RAW.CAST_TO_VARCHAR2 (
   r IN RAW)
RETURN VARCHAR2;
```

Parameters

Table 13-16 CAST_TO_VARCHAR2 Function Parameters

Parameter	Description
r	RAW value, without leading length field, to be changed to a VARCHAR2 value

Return Value

Data converted from the input RAW value, or NULL if the input is NULL

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;
```

Parameters

Table 13-17 COMPARE Function Parameters

Parameter	Description
r1	First `RAW` value to be compared **Note**: The value can be `NULL` or zero-length.
r2	Second `RAW` value to be compared **Note**: The value can be `NULL` or zero-length.
pad	Byte to extend whichever of the input values is shorter (default `x'00'`)

Return Value

A `NUMBER` value that equals the position number (numbered from 1) of the first mismatched byte when comparing the two input values, or 0 if the input values are identical or both `NULL`

CONCAT Function

This function concatenates up to 12 `RAW` values into a single `RAW` value. If the concatenated size exceeds 32 KB, an error is returned.

Syntax

```
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;
```

Parameters

Items *r1...r12* are the RAW items to concatenate.

Return Value

RAW value consisting of the concatenated input values

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 database.

Syntax

```
UTL_RAW.CONVERT (
   r            IN RAW,
   to_charset   IN VARCHAR2,
   from_charset IN VARCHAR2)
  RETURN RAW;
```

Parameters

Table 13-18 CONVERT Function Parameters

Parameter	Description
r	RAW byte-string to be converted
to_charset	Name of Globalization Support character set to which the input value is converted
from_charset	Name of Globalization Support character set from which the input value is converted

Return Value

Converted byte-string according to the specified character set

Exceptions

VALUE_ERROR occurs under any of the following circumstances:

- The input byte-string is missing, NULL, or zero-length.
- The *from_charset* or *to_charset* parameter is missing, NULL, or zero-length.

- The *from_charset* or *to_charset* parameter is invalid or unsupported.

COPIES Function

This function returns a specified number of copies of a specified RAW value, concatenated.

Syntax

```
UTL_RAW.COPIES (
   r IN RAW,
   n IN NUMBER)
  RETURN RAW;
```

Parameters

Table 13-19 COPIES Function Parameters

Parameters	Description
r	RAW value to be copied
n	Number of times to copy the RAW value **Note**: This must be a positive value.

Return Value

RAW value copied the specified number of times and concatenated

Exceptions

VALUE_ERROR occurs under any of the following circumstances:

- The value to be copied is missing, NULL, or zero-length.
- The number of times to copy the value is less than or equal to 0.
- The length of the result exceeds the maximum allowable length for a RAW value, which is 32767 bytes.

LENGTH Function

This function returns the length in bytes of a RAW value.

Syntax

```
UTL_RAW.LENGTH (
   r  IN RAW)
RETURN NUMBER;
```

Parameters

Table 13-20 LENGTH Function Parameters

Parameter	Description
r	RAW byte-stream to be measured

Return Value

NUMBER value indicating 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, starting from a specified byte position and proceeding for a specified number of bytes.

Syntax

```
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;
```

Parameters

Table 13-21 OVERLAY Function Parameters

Parameters	Description
overlay_str	Byte-string used to overlay target
target	Target byte-string to be overlaid
pos	Byte position in target at which to start overlay, numbered from 1 (default 1)
len	Number of bytes to overlay (default: length of *overlay_str*)
pad	Pad byte used when len exceeds *overlay_str* length or *pos* exceeds *target* length (default x'00')

Return Value

RAW target byte value overlaid as specified

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 exceed 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

VALUE_ERROR occurs under any of the following circumstances:

- The *overlay_str* is NULL or zero-length.
- The *target* is missing or undefined.
- The length of *target* exceeds the maximum length for a RAW value, 32767 bytes.
- The *len* is less than 0.
- The *pos* is less than or equal to 0.

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

```
UTL_RAW.REVERSE (
   r IN RAW)
  RETURN RAW;
```

Parameters

Table 13-22 REVERSE Function Parameters

Parameter	Description
r	RAW value to reverse

Return Value

RAW value that is the reverse of the input value

Exceptions

VALUE_ERROR occurs if the input value is NULL or zero-length.

SUBSTR Function

This function returns a substring of a RAW value for a specified number of bytes and starting position.

Syntax

```
UTL_RAW.SUBSTR (
   r   IN RAW,
```

```
 pos IN BINARY_INTEGER
[,len IN BINARY_INTEGER DEFAULT NULL])
RETURN RAW;
```

Parameters

Table 13-23 SUBSTR Function Parameters

Parameter	Description
r	RAW byte-string from which the substring is extracted
pos	Byte position at which to begin extraction, either counting forward from the beginning of the input byte-string (positive value) or backward from the end (negative value)
len	Number of bytes, beginning at *pos* and proceeding toward the end of the byte string, to extract (default: to the end of the RAW byte-string)

Return Value

RAW substring beginning at position *pos* for *len* bytes, or NULL if the input is NULL

Usage Notes

If *pos* is positive, SUBSTR counts from the beginning of the RAW byte-string to find the first byte. If *pos* is negative, SUBSTR counts backward from the end of the RAW byte-string. 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 the RAW byte-string.

Exceptions

VALUE_ERROR occurs under any of the following circumstances:

- The *pos* equals 0 or is greater than the length of *r*.
- The *len* is less than or equal to 0.
- The *len* is greater than (length of *r*) minus (*pos*-1).

Examples

Example 1: This example, run in ttIsql, counts backward 15 bytes from the end of the input RAW value for its starting position, then takes a substring of five bytes starting at that point.

```
declare
  sr raw(32767);
  r raw(32767);

begin
  sr       := hextoraw('12365678121256123444343412345678 90ABAA1234');
  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: 123656781212561234443431234567890ABAA1234
return raw: 5612344434

PL/SQL procedure successfully completed.
```

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.

Example 2: This example, run in `ttIsql`, has the same input `RAW` value 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.

```
declare
  sr raw(32767);
  r raw(32767);
begin
  sr       := hextoraw('123656781212561234443431234567890ABAA1234');
  r := UTL_RAW.SUBSTR(sr, -15);
  dbms_output.put_line('source raw: ' || sr);
  dbms_output.put_line('return raw: ' || r);
end;
/
```

Here is the result:

```
source raw: 123656781212561234443431234567890ABAA1234
return raw: 561234443431234567890ABAA1234
```

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 and translate from in the input bytes, and a set of corresponding bytes to translate to.

Whenever a byte in the specified *`from_set`* is found in the input `RAW` value, it is translated to the corresponding byte in the *`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 *`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

```
UTL_RAW.TRANSLATE (
   r        IN RAW,
   from_set IN RAW,
   to_set   IN RAW)
  RETURN RAW;
```

> **Note:**
>
> Be aware that *`to_set`* and *`from_set`* are reversed in the calling sequence compared to `TRANSLITERATE`.

Parameters

Table 13-24 TRANSLATE Function Parameters

Parameter	Description
`r`	`RAW` source byte-string whose bytes are to be translated, as applicable
`from_set`	`RAW` byte-codes that are searched for in the source byte-string Where found, they are translated in the result.
`to_set`	`RAW` byte-codes to translate to Where a *`from_set`* byte is found in the source byte-string, it is translated in the result to the corresponding *`to_set`* byte, as applicable.

Return Value

`RAW` value with the translated byte-string

Usage Notes

- If *`to_set`* is shorter than *`from_set`*, the extra *`from_set`* bytes have no corresponding translation bytes. Bytes from the input `RAW` value that match any such *`from_set`* bytes are not translated or included in the result. They are effectively translated to `NULL`.
- 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 TRANSLITERATE:
>
> - The *from_set* parameter comes before the *to_set* parameter in the calling sequence.
> - Bytes from the source byte-string 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

VALUE_ERROR occurs if the source byte string, *from_set*, or *to_set* is NULL or zero-length.

Examples

Example 1: In this example, run in ttIsql, from_set is x'12AA34' and to_set is x'CD'. Wherever '12' appears in the input RAW value it is replaced by 'CD' in the result. Wherever 'AA' or '34' appears in the input RAW value, because there are no corresponding bytes in *to_set*, those bytes are not included in the result (effectively translated to NULL).

You can compare this to Examples in the TRANSLITERATE section to see how the functions differ.

```
declare
  sr raw(32767);
  from_set raw(32767);
  to_set raw(32767);
  r raw(32767);
begin
  sr       := hextoraw('123656781212561234443434123456789OABAA1234');
  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: 123656781212561234443434123456789OABAA1234
```

```
return raw: CD365678CDCD56CD44CD567890ABCD

PL/SQL procedure successfully completed.
```

The inputs and output are presented in the following, 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 CD 56 CD    44       CD    567890AB    CD
```

Example 2: In this example, run in `ttIsql`, the *`from_set`* is `x'12AA12'` and the *`to_set`* is `x'CDABEF'`. Wherever `'12'` appears in the input RAW it is replaced by `'CD'` in the result. Wherever `'AA'` appears in the input it is 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);
  r raw(32767);
begin
  sr       := hextoraw('123656781212561234443434123456789OABAA1234');
  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 in the `TRANSLITERATE` section.

```
from_set:   12AA12
to_set:     CDABEF
source raw: 123656781212561234443434123456789OABAA1234
return raw: CD365678CDCD56CD34443434CD34567890ABABCD34

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 and convert from in the input bytes, 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 copied as-is to the output RAW value.

Syntax

```
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.

Parameters

Table 13-25 TRANSLITERATE Function Parameters

Parameter	Description
r	RAW source byte-string whose bytes are to be converted, as applicable
to_set	RAW byte-codes to convert to Where a *from_set* byte is found in the source byte-string, 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.
from_set	RAW byte-codes that are searched for in the source byte-string Where found, they are converted in the result. The default is x'00' through x'FF', which results in all bytes in the source byte string being converted in the result.
pad	A "padding" byte used as the conversion value for any byte in the source byte-string for which there is a matching byte in *from_set* that does not have a corresponding byte in *to_set* (default x'00')

Return Value

RAW value with the converted byte-string

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 value 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 the source byte-string 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 differ in functionality only when *to_set* has fewer bytes than *from_set*.

Exceptions

VALUE_ERROR occurs if the source byte-string is NULL or zero-length.

Examples

Example 1: In this example, run in ttIsql, the *from_set* is x'12AA34' and the *to_set* is x'CD'. Wherever '12' appears in the input RAW value it is replaced by 'CD' in the result. Wherever 'AA' or '34' appears in the input RAW value, because there are no corresponding bytes in *to_set*, those bytes are replaced by the *pad* byte, which is not specified and therefore defaults to x'00'.

You can compare this to Examples in the TRANSLATE section to see how the functions differ.

```
declare
  sr raw(32767);
  from_set raw(32767);
  to_set raw(32767);
  r raw(32767);
begin
  sr       := hextoraw('123656781212561234443431234567890ABAA1234');
  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: 123656781212561234443431234567890ABAA1234
return raw: CD365678CDCD56CD00440000CD00567890AB00CD00
```

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

The inputs and output are presented in the following, 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 CD 56 CD 00 44 00 00 CD 00 567890AB 00 CD 00
```

Example 2: This example, run in `ttIsql`, is the same as the preceding example, except *`pad`* is specified to be `x'FF'`.

```
declare
  sr raw(32767);
  from_set raw(32767);
  to_set raw(32767);
  pad raw(32767);
  r raw(32767);
begin
  sr       := hextoraw('123656781212561234443431234567890ABAA1234');
  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: 123656781212561234443431234567890ABAA1234
return raw: CD365678CDCD56CDFF44FFFFCDFF567890ABFFCDFF

PL/SQL procedure successfully completed.
```

Example 3: In this example, run in `ttIsql`, the *`from_set`* is `x'12AA12'` and the *`to_set`* is `x'CDABEF'`. Wherever `'12'` appears in the input `RAW` value it is replaced by `'CD'` in the result. Wherever `'AA'` appears in the input it is 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);
```

```
  r raw(32767);
begin
  sr       := hextoraw('123656781212561234443434123456789ABAA1234');
  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 in the `TRANSLATE` section.

```
from_set:   12AA12
to_set:     CDABEF
source raw: 123656781212561234443434123456789ABAA1234
return raw: CD365678CDCD56CD34443434CD34567890ABABCD34

PL/SQL procedure successfully completed.
```

Example 4: In this example, run in `ttIsql`, *`from_set`* and *`to_set`* are not specified.

```
declare
  sr raw(32767);
  r raw(32767);
begin
  sr       := hextoraw('123656781212561234443434123456789ABAA1234');
  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: 123656781212561234443434123456789ABAA1234
return raw: 00000000000000000000000000000000000000000000

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, 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;
```

Parameters

Table 13-26 XRANGE Function Parameters

Parameters	Description
start_byte	Beginning byte-code value for resulting sequence (default `x'00'`)
end_byte	Ending byte-code value for resulting sequence (default `x'FF'`)

Return Value

`RAW` value containing the succession of one-byte encodings

Examples

The following three examples, run in `ttIsql`, 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.xrange(s,e);
            dbms_output.put_line(r);
         end;
         /
0102030405060708090A

PL/SQL procedure successfully completed.

Command> declare
            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;
         /
EEEFF0F1F2F3F4F5F6F7F8F9FAFBFCFDFEFF00010203040506070809 0A
```

```
PL/SQL procedure successfully completed.

Command> declare
            r raw(32767);
         begin
            r := utl_raw.xrange();
            dbms_output.put_line(r);
         end;
         /
000102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F2021222324252
627
28292A2B2C2D2E2F303132333435363738393A3B3C3D3E3F404142434445464748494A4B4C4D4
E4F
505152535455565758595A5B5C5D5E5F606162636465666768696A6B6C6D6E6F7071727374757
677
78797A7B7C7D7E7F808182838485868788898A8B8C8D8E8F909192939495969798999A9B9C9D9
E9F
A0A1A2A3A4A5A6A7A8A9AAABACADAEAFB0B1B2B3B4B5B6B7B8B9BABBBCBDBEBFC0C1C2C3C4C5C
6C7
C8C9CACBCCCDCECFD0D1D2D3D4D5D6D7D8D9DADBDCDDDEDFE0E1E2E3E4E5E6E7E8E9EAEBECEDE
EEF
F0F1F2F3F4F5F6F7F8F9FAFBFCFDFEFF

PL/SQL procedure successfully completed.
```

14
UTL_RECOMP

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

`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 before operation to 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');
```

UTL_RECOMP Subprograms

Table 14-1 summarizes the `UTL_RECOMP` subprograms, followed by a full description of each subprogram.

Table 14-1 UTL_RECOMP Package Subprograms

Subprogram	Description
RECOMP_PARALLEL Procedure	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.
RECOMP_SERIAL Procedure	Recompiles invalid objects in a given schema or all invalid objects in the database.

Note:

- 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

```
UTL_RECOMP.RECOMP_PARALLEL(
   threads  IN   BINARY_INTEGER DEFAULT NULL,
   schema   IN   VARCHAR2    DEFAULT NULL,
   flags    IN   BINARY_INTEGER DEFAULT 0);
```

Parameters

Table 14-2 RECOMP_PARALLEL Procedure Parameters

Parameter	Description
threads	The number of recompile threads to run in parallel In TimesTen, *threads* is always 1.
schema	The schema in which to recompile invalid objects If NULL, all invalid objects in the database are recompiled.
flags	Flag values intended for internal testing and diagnosability only

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 14-3 RECOMP_SERIAL Procedure Parameters

Parameter	Description
schema	The schema in which to recompile invalid objects If NULL, all invalid objects in the database are recompiled.
flags	Flag values intended for internal testing and diagnosability only