



# ILP PROGRAM - ORACLE APPLICATIONS

Tata Consultancy Services

## Oracle SQL and PL-SQL Study Guide

Author: [MD. Nazmul Hoda \(TCS\)](#)  
Creation Date: Apr 15, 2015  
Last Updated: Apr 15, 2015  
Document Ref: [ILP/ORACLE/SQL-PLSQL/01](#)  
Version: DRAFT 1A

### Approvals:

<Approver 1>

Santanu Sarkar (TCS)

---

<Approver 2>

Shubho Chakraborty(TCS)

---

## Document Control

### Change Record

Date	Author	Version	Change Reference
15-Apr-15	MD. Nazmul Hoda	Draft 1A	No Previous Document

### Reviewers

Name	Position

### Distribution

Copy No.	Name	Location
1	Library Master	Project Library
2		Project Manager
3		
4		

#### Note To Holders:

If you receive an electronic copy of this document and print it out, please write your name on the equivalent of the cover page, for document control purposes.

If you receive a hard copy of this document, please write your name on the front cover, for document control purposes.

## Contents

### Table of Contents

<b>Document Control</b> .....	2
How to use this manual .....	8
1. Understanding the architecture.....	9
3. Writing Basic SQL Statements .....	14
3. Writing Basic SQL Statements .....	14
Basic Oracle Objects .....	18
TRANSACTIONS IN ORACLE .....	23
OVERVIEW OF PL/SQL .....	25
Block Structure .....	25
PL/SQL Architecture .....	26
PL/SQL DATATYPES.....	26
Predefined Datatypes.....	26
Number Types: .....	27
NUMBER Subtypes .....	28
Character Types: .....	28
Boolean Type:.....	29
Datetime and Interval Types:.....	29
Datatype Conversion.....	33
Explicit Conversion .....	33
Implicit Conversion .....	33
PL/SQL CONTROL STRUCTURES.....	35
Conditional Control: IF and CASE Statements.....	36
CASE Statement .....	38
Iterative Control: LOOP and EXIT Statements.....	39

---

Sequential Control: GOTO and NULL Statements .....	44
UNDERSTANDING SQL*PLUS.....	47
Overview of SQL * Plus.....	47
Manipulating Commands.....	50
Setting Up Your SQL*Plus Environment.....	51
Communicating with the User .....	52
Using Bind Variables.....	52
ABOUT CURSORS.....	54
Overview of Implicit Cursors .....	54
Overview of Explicit Cursors .....	55
Declaring a Cursor .....	56
Opening a Cursor.....	56
Fetching with a Cursor .....	56
Closing a Cursor.....	57
Using Cursor FOR Loops .....	57
Using Subqueries Instead of Explicit Cursors .....	57
Parameterized Cursors.....	58
SELECT FOR UPDATE Cursors .....	61
Cursor Variables .....	62
Using a Cursor Variable .....	64
Cursor Variables Assignment .....	67
Dynamism in Using Cursor Variables.....	68
HANDLING PL/SQL ERRORS .....	71
Overview of PL/SQL Error Handling .....	71
Advantages of PL/SQL Exceptions .....	72
Predefined PL/SQL Exceptions .....	72
Defining Your Own PL/SQL Exceptions .....	76
Defining Your Own Error Messages: Procedure RAISE_APPLICATION_ERROR.....	77

---

How PL/SQL Exceptions Are Raised .....	78
How PL/SQL Exceptions Propagate .....	79
Retrieving the Error Code and Error Message: SQLCODE and SQLERRM .....	81
SUBPROGRAMS (PROCEDURES & FUNCTIONS) .....	83
Understanding PL/SQL Procedures .....	83
How to create stored procedure: .....	85
Understanding PL/SQL Functions .....	86
Actual Versus Formal Subprogram Parameters .....	87
Positional Versus Named Notation for Subprogram Parameter .....	88
Specifying Subprogram Parameter Modes .....	88
Passing Large Data Structures with the NOCOPY Compiler Hint .....	90
Performance Improvement of NOCOPY .....	91
Using Default Values for Subprogram Parameters .....	93
PL/SQL PACKAGES .....	94
What Is a PL/SQL Package? .....	94
Advantages of PL/SQL Packages .....	95
Understanding The Package Spec .....	96
Referencing Package Contents .....	96
Understanding The Package Body .....	96
Overloading Packaged Subprograms .....	96
DATABASE TRIGGER .....	98
Objectives .....	98
Different Types of Triggers .....	98
Guidelines for Designing Triggers .....	98
Creating DML Triggers .....	99
Using OLD and NEW Qualifiers .....	103
INSTEAD OF Triggers .....	104
DROP TRIGGER Syntax .....	107

Trigger Test Cases.....	107
Testing Triggers .....	107
After Row and After Statement Triggers.....	108
COLLECTIONS AND RECORDS.....	111
Objectives .....	111
Composite Data Types.....	111
RECORD and TABLE Data Types.....	111
PL/SQL Records .....	111
The %ROWTYPE Attribute .....	113
INDEX BY Tables .....	115
Creating a index by table .....	115
INDEX BY Table Structure.....	116
Referencing an INDEX BY Table Syntax: .....	116
Using INDEX BY Table Methods .....	117
Summary .....	118
BULK BINDING.....	119
Objectives .....	119
Tuning PL/SQL Performance with Bulk Binds.....	119
Reducing Loop Overhead for Collections with Bulk Binds .....	120
How Do Bulk Binds Improve Performance?.....	121
Using the FORALL Statement .....	122
Counting Rows Affected by FORALL Iterations with the %BULK_ROWCOUNT Attribute .....	123
Handling FORALL Exceptions with the %BULK_EXCEPTIONS Attribute .....	123
Retrieving Query Results into Collections with the BULK COLLECT Clause.....	125
Limiting the Rows for a Bulk FETCH Operation with the LIMIT Clause .....	126
Restrictions on BULK COLLECT .....	127
Using FORALL and BULK COLLECT Together .....	128
Summary .....	128

DBMS_OUTPUT .....	129
About the DBMS_OUTPUT Package .....	129
NATIVE DYNAMIC SQL .....	131
What Is Dynamic SQL? .....	131
The Need for Dynamic SQL .....	131
Using the EXECUTE IMMEDIATE Statement.....	132
Backward Compatibility of the USING Clause .....	134
Specifying Parameter Modes .....	134
Using the OPEN-FOR Statements.....	135
Tips and Traps for Dynamic SQL.....	137
Summary .....	139
UTL_FILES .....	140
What Is the UTL_FILE Package? .....	140
File Processing.....	140
The UTL_FILE Package: Procedures and Functions.....	141
SQL*LOADER .....	150
SQL*Loader Basics.....	150
Log File and Logging Information .....	154
Conventional Path Load versus Direct Path Load.....	154
USER MANAGEMENT .....	156
Introduction to Privileges.....	156
Introduction to Roles.....	157
IMPORT & EXPORT UTILITY .....	160
Introduction .....	160

## How to use this manual



Video1: Script: Vid1-Introduction to the chapter and its content – Face recording.

---

This video will introduce the material covered in this pdf, the goals,

1. How this document is organized
2. What is the purpose of this document
3. What will you achieve after going through the document and related videos
4. How to read this document
5. How does it relate to the work you will be doing on real project
6. Reference to other reading materials for further references

---

This manual has been organized as a step by step guide to teach how to write SQL and PL/SQL statements. The target audience is new comes to Oracle SQL and PL/SQL. It assumes that the reader is new to Oracle programming and concepts. After completing this course, you will be able to develop SQL statements and write programs using PL/SQL. The examples used in this guide have been tested on the following version of Oracle:

1. Oracle Database 10g Enterprise Edition Release 10.2.0.1.0
2. PL/SQL Release 10.2.0.1.0

This manual is organized to be read in a serial fashion and follow the instructions given in the document as it is. Practical examples are given in each section to guide you through every step. The tables referred here are common (shared) tables used by different batches, so care should be taken not to delete or update the rows which does not belong to you, this may create problem for the other batches. At the end of the course, you should delete the data you have created.

There are several symbols used to designate particular sections, which are described below:



- Describes the purpose of the section.



- Notes relevant to the section above

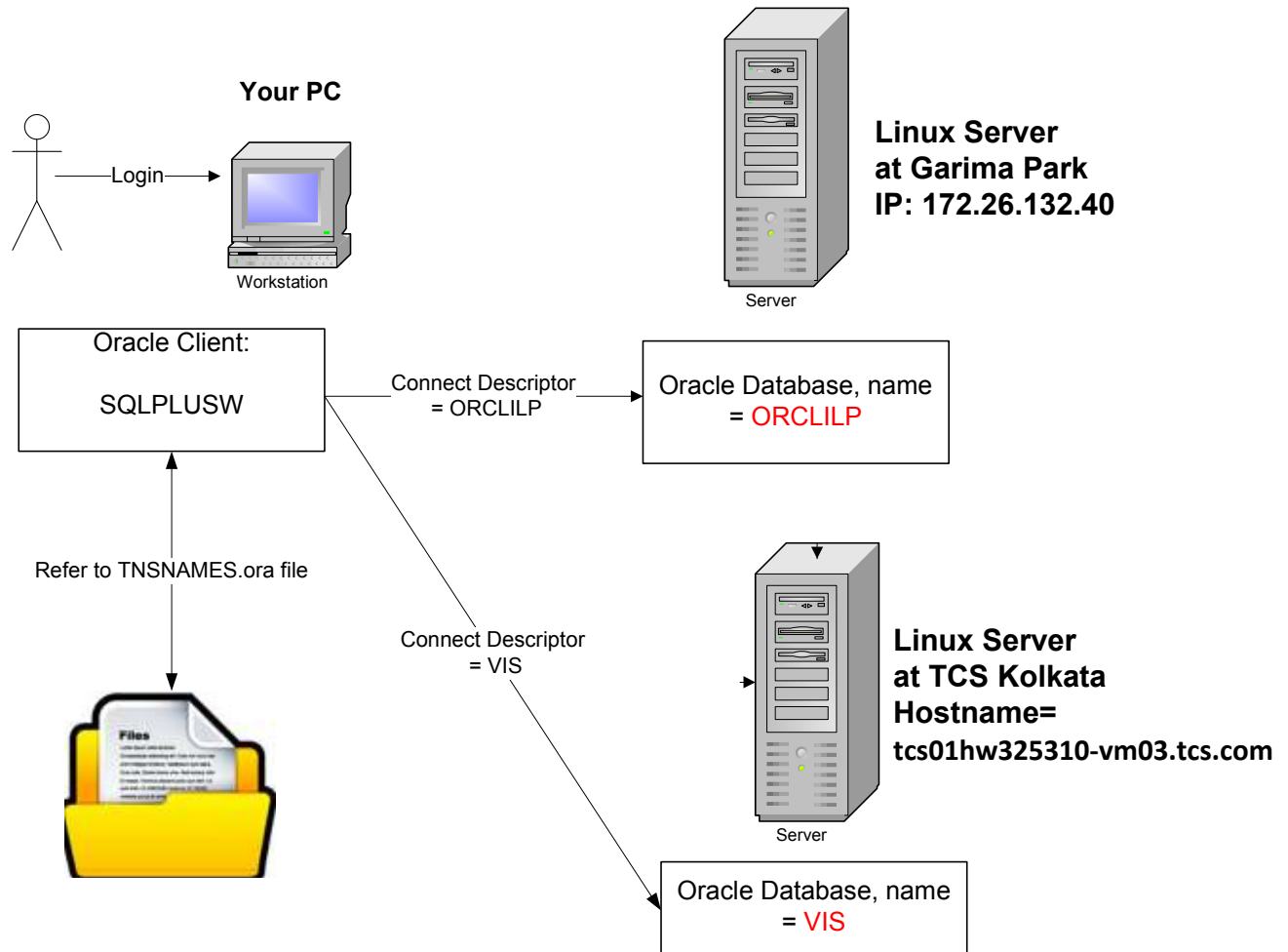


- This denotes the task to be completed by the audience on his own PC. The layout of the output has to be followed as it is. For any confusion, the faculty should be contacted.



## 1. Understanding the architecture

Before we start using sql , you need to setup your PC.. Even before that, let us understand the architecture:



Note the following in the above diagram:

1. The Oracle Database is sitting on the linux server with IP address : 172.26.132.40.
2. Oracle client is sitting on your PC ( includes the sqlplusw tool)
3. Your PC and the server are connected by TCS network.

4. The TNSNAMES.ora file in your PC has details of 'How to connect to the database' This is called connect identifier.
5. In the TNSNAMES.ora , we need to put the details of the database, so that your PC can locate the database and the server and make a database connection. This connection is made over TNS (Transparent Network Substrate) . This is a protocol, which oracle client and the database understands.

Let us see how to configure the entries in the TNSNAMES.ora file:

But before that, we need to locate the file in your PC. To do this, go to the command prompt ( Start > Run > cmd), and issue the following command:

C:\Users\Mohammed> tnsping ORCLILP



*Note: Here, ORCLILP is the name of the database, and with the help of the tnsping utility, we are trying to locate the database. The tnsping utility is provided by the oracle client software, which has already been installed on your PC by the TCS IS team.*

Look at the below screenshot carefully

```
C:\Users\Mohammed>tnsping ORCLILP

TNS Ping Utility for 32-bit Windows: Version 10.2.0.1.0 - Production on 19-APR-2015 09:26:43

Copyright (c) 1997, 2005, Oracle. All rights reserved.

Used parameter files:
C:\Nazmul\ORACLE_10201_HOME\network\admin\sqlnet.ora

Used TNSNAMES adapter to resolve the alias
Attempting to contact (DESCRIPTION = (ADDRESS = (PROTOCOL = TCP)(HOST = TCSAUSLT749)(PORT = 1521)) (CONNECT_DATA = (SERVER = DEDICATED) (SERVICE_NAME = orcl)))
OK (30 msec)
```

#### Observation:

– The utility tells that, it is looking at the file sqlnet.ora in the directory 'C:\Nazmul\ORACLE\_10201\_HOME\network\admin\sqlnet.ora' . SO your tnsnames.ora file will also be located in the same directory.

Let us open the file ( preferably in wordpad) and analyze its content:

```
ORCLIP =
(DESCRIPTION =
  (ADDRESS = (PROTOCOL = TCP)(HOST = 172.26.132.40)(PORT = 1521))
  (CONNECT_DATA =
    (SERVER = DEDICATED)
    (SERVICE_NAME = orclilp)
  )
)
```

```
VIS=
  (DESCRIPTION=
    (ADDRESS=(PROTOCOL=tcp)(HOST=tcs01hw325310-vm03.tcs.com)(PORT=1521))
    (CONNECT_DATA=
      (SERVICE_NAME=VIS)
      (INSTANCE_NAME=VIS)
    )
  )
```

Note that this file has reference to two strings – **ORCLIP** and **vis**. These are called the 'Connect Descriptor' . All the connections you make to the database from your PC first refers to this file. This connect descriptor entry in this file will tell the oracle client on your PC , how to reach the database.

Let us see the information it gathers:



### **TASK1: Understand the connection information gathered by your oracle client:**

Go to command prompt and issue the following command:

```
C:\Users\Mohammed>tnsping ORCLIP
```

```
TNS Ping Utility for 32-bit Windows: Version 10.2.0.1.0 - Production on 23-APR-2015 06:52:53
Copyright (c) 1997, 2005, Oracle. All rights reserved.
Used parameter files:
C:\Nazmul\ORACLE_10201_HOME\network\admin\sqlnet.ora
Used TNSNAMES adapter to resolve the alias
Attempting to contact (DESCRIPTION = (ADDRESS = (PROTOCOL = TCP)(HOST = TCSAUSLT749)(PORT = 1521)) (CONNECT_DATA = (SERVER = DEDICATED) (SERVICE_NAME = orcl)))
OK (520 msec)
```

The task is to answer the following, from the information retrieved above:

1. What is the name of the 'Connect String'
2. How does your PC know, the name of the server
3. What is the port on which database is listening to your connection request
4. Can we have multiple connect strings for the same database?

Please check with your faculty for the correct answers.

## 2. Setting up the environment to start using SQL

If your tnsnames.ora file does not contain the above two entries, please cut paste it from the value demonstrated in the above example. Now you are all set to connect to these two databases.

Now let us login to the sqlplus.

Start, type sqlplusw, select the sqlplus option, it will ask for the username and password. When you have joined ILP, your database userid and password has been created, please check with your faculty, if you do not have one.



*Note: You can also connect to the ORCLILP database directly from the server( through putty login), All of you will have your Linux ID create on the server 172.26.132.40. You can type the command sqlplus from the server as well. But there is an essential difference between these two connections. The sqlplusw from your PC is a TNS connection from your PC to the database hopping through the TCS network, while the sqlplus connection made from putty is a direct connection to the database from the server itself, so there is no network involved in between, and your Oracle client( the tnsnames.ora on your PC) does not have any role in this.*



### **TASK2: Understand local vs remote connection**

Rename the tnsnames.ora file on your PC to tnsnames\_old.ora. and try the following:

1. Use sqlplus to connect to the ORCLILP database
2. Use putty session to connect to the ORCLILP database

Explain why one connection is successful, while other fails.



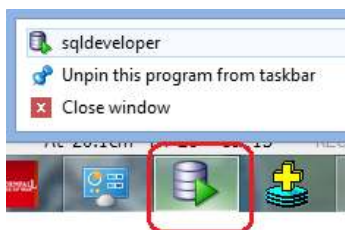
### **TASK3: Creating multiple connect descriptor for one database/**

Make changes in your tnsnames.ora file, so that you can login the ORCLILP database by using:

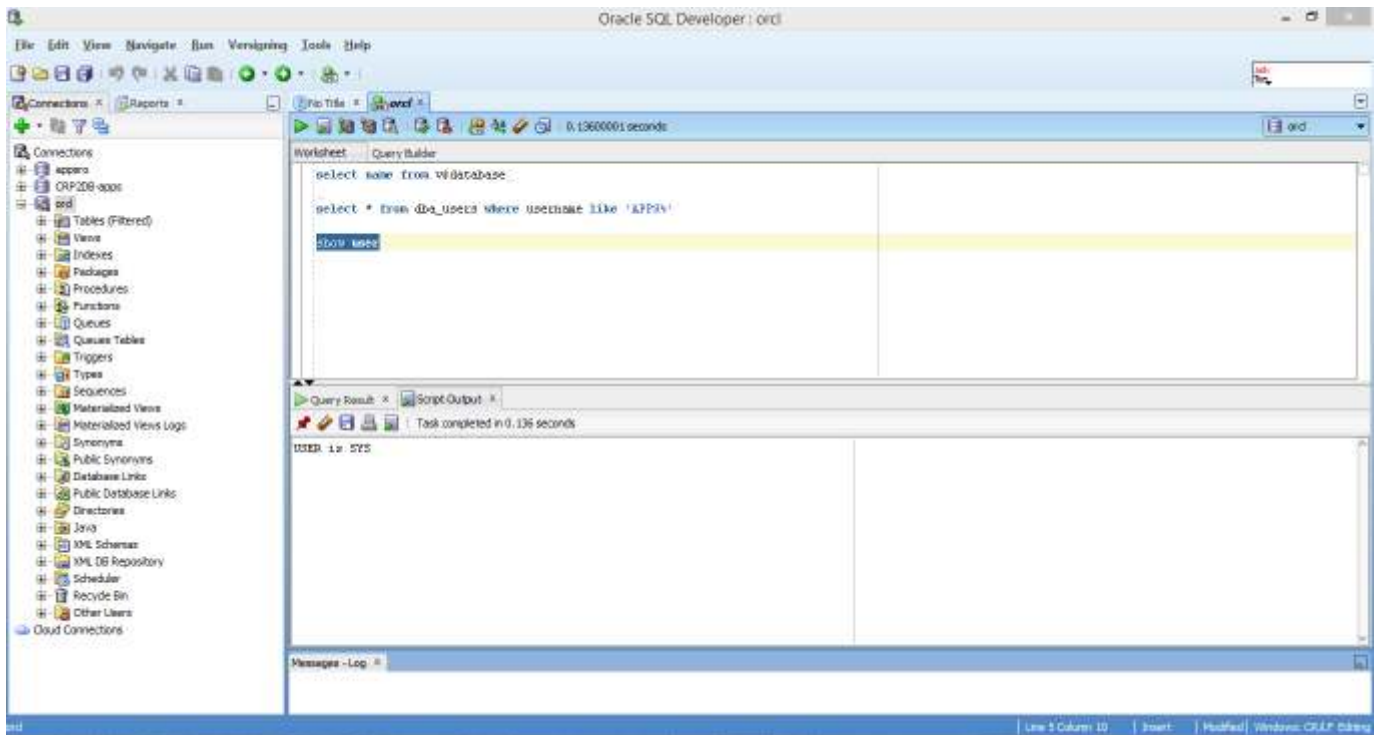
1. ORCLILP
2. ORCLILP\_DUMMY

SQLplusw and sqlplus is not a very user friendly tool, So I will be using a tool called 'SQL Developer'. This is a freeware and already have been installed in your PC. Check with your faculty, if you face issues running the tool.

To access it, type sqldeveloper in windows start, the icon looks like the below:



This will open a GUI tool looking like this:



Here , you can type your sql command, and see the output in another window, where you can scroll and see the data ( unlike split data in sqlplus – which is difficult to read)

### 3. Writing Basic SQL Statements

Now that you are able to successfully login to database using sqlplus/sqlplusw and sqldeveloper, let us learn some basics of sql.

**Note: You will use the ORCLILP connection to do the entire practical in your SQL and PL/SQL course, and not use the VIS database.**

I will be using a database user 'naz' for the following examples, you can use your own user. We need to setup a new connection in sqldeveloper for this. To do this:

Click File > New connection:

Type the credentials as below, and hit the 'test' button.

The screenshot shows the 'New / Select Database Connection' dialog box. On the left, there is a table with existing connections:

Connection Name	Connection Details
appsro	appsro@//localhost...
CRP2DB-apps	apps@//172.27.45...
ord	sys@//localhost:15...

On the right, the configuration fields are as follows:

- Connection Name: NAZ
- Username: naz
- Password: (masked with dots)
- ☒ Save Password
- Tab: Oracle (selected), Access
- Connection Type: Basic (dropdown)
- Role: default (dropdown)
- Hostname: localhost
- Port: 1521
- ☒ SID: ord
- ☐ Service name: (empty field)
- ☐ OS Authentication
- ☐ Kerberos Authentication
- ☐ Proxy Connection

At the bottom, there are buttons: Help, Save, Clear, Test (highlighted), Connect, and Cancel. The status at the bottom left says 'Status : Success'.

Now hit the 'connect' button, it will give you a fresh window.



*Note: The credentials like – Hostname etc, will be different for you. Use your knowledge( information from tnsnames.ora to fill up the details appropriately), or check with your faculty..*

Let us first create a table, issue the following command:

```
SQL> create table t_fuel_bonus (  
Purchase_Amt number,  
Bonus_item varchar2(100)  
)  
/
```



To run the statement, you need to select, and press the small green 'Play' button, as shown in the diagram above. You will see the message, in the output window.

**table T\_FUEL\_BONUS created.**

To see what data is there in the table, run the command:

```
SQL>select * from T_FUEL_BONUS
```



Note the 'All Rows Fetches: 0 in 0.028 Seconds' - meaning, there is no data in the table.

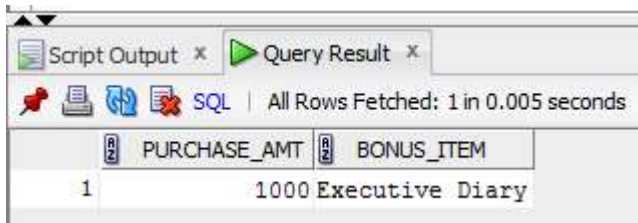
Let us insert some data into this table:

```
SQL>insert into T_FUEL_BONUS values( 1000,'Executive Dairy')
```

*1 rows inserted.*

Let us see the data again:

```
SQL>select * from T_FUEL_BONUS
```



	PURCHASE_AMT	BONUS_ITEM
1		1000 Executive Diary



#### **TASK4: Understanding the commit**

You must be sure, that you have inserted one row in the table, really ? Let us make an experiment to see this.

Make a nbew sqlplusw connection and issue the same select statement.

You will be surprised, that you cannot see any data, while you can see still the data in the original sqldeveloper window where you have inserted the row.

This is because oracle maintains 'read consistency'. Whenever you make a connection to the database, oracle assigns a oracle session for you, and all transactions made in this session are available to this session only, unless you issue a **COMMIT** statement.

Now issue a commit in the first session, and check if the data is available in another session, you can see the data now.



#### **TASK5: Auto commit**

Repeat the above experiment with a table creation.

In the first session , issue:

```
SQL>create table test ( name char(30));
```

*table TEST created.*

Do not issue a COMMIT, go to the second session, and issue:

```
SQL>desc test
```

```
Name Null Type
-----
NAME      CHAR (30)
```

Wow ! I can see the table from other session also !

This is because DDL stataments areauto commit.



**DDL stands for Data Definition Language** (used to create or alter objects), so the create table statement was a DDL statement , which does not require a commit, if you still issue a commit, it will not harm anyone, but it is not of any use, as the commit has already been done automatically.

**DML stands for Data Manipulation language** (used to play around with data – inset, update, delete etc). Such statement required explicit commit.

Now let us drop the table temp:

```
SQL>Drop table test;
```

```
table TEST dropped.
```

## Basic Oracle Objects

Now that we have created a table, let us see what are the other database objects.

The database schema is a collection of logical-structure objects, known as schema objects, that define how you see the database's data. These schema objects consist of structures such as tables, clusters, indexes, views, stored procedures, database triggers, and sequences.

We have created the table called T\_FUEL\_BONUS. We created the table while we were connected to the database with user Naz. So **Naz** is the schema. The table has been created under the schema Naz, and Naz is the owner of this table.

All the database objects are created under a schema or user.

To check what user you are currently logged in, issue the following command:

```
SQL>Show user
```

```
USER is NAZ
```

Let us learn another very useful query to find out the whereabouts of an object:

```
select owner, object_type, status from all_objects where object_name='T_FUEL_BONUS'
```

	OWNER	OBJECT_TYPE	STATUS
1	SYS	TABLE	VALID
2	NAZ	TABLE	VALID



*Note: There are two tables in the database with name 'T\_FUEL\_BONUS' . One is owned by user SYS and the other by NAZ. They are both in valid ( mean Healthy in general). This is a very important concept and confuses many starters in Oracle technology.*

Now we will learn other types of database objects.

### Table

A table, which consists of a tablename and rows and columns of data, is the basic logical storage unit in the Oracle database. Columns are defined by name and data type. Here is how to create a table:

```
create table t_fuel_bonus (  
Purchase_Amt number,  
Bonus_item varchar2(100)  
)  
/
```

Note the column type, bit of explanation of the data types:

1. Number – can store number
2. Varchar2(100) – A character type , but the length can vary. If the size is defined as 100, the length of data can vary from 0 to 100.
3. Char(100) – Fixed length character. Value cannot be less than 100 characters in size – this is not used in the above example.

Below is the list of available data types. You can use them as per your requirement, but for general purpose, the above three are sufficient.

Code	Datatype	Description
1	<b>VARCHAR2</b> ( <i>size</i> [BYTE   CHAR])	Variable-length character string having maximum length <i>size</i> bytes or characters. Maximum <i>size</i> is 4000 bytes or characters, and minimum is 1 byte or 1 character. You must specify <i>size</i> for <b>VARCHAR2</b> .  <b>BYTE</b> indicates that the column will have byte length semantics; <b>CHAR</b> indicates that the column will have character semantics.
1	<b>NVARCHAR2</b> ( <i>size</i> )	Variable-length Unicode character string having maximum length <i>size</i> characters. The number of bytes can be up to two times <i>size</i> for <b>AL16UTF16</b> encoding and three times <i>size</i> for <b>UTF8</b> encoding. Maximum <i>size</i> is determined by the national character set definition, with an upper limit of 4000 bytes. You must specify <i>size</i> for <b>NVARCHAR2</b> .
2	<b>NUMBER</b> ( <i>precision</i> [, <i>scale</i> ])	Number having precision <i>p</i> and scale <i>s</i> . The precision <i>p</i> can range from 1 to 38. The scale <i>s</i> can range from -84 to 127.
8	<b>LONG</b>	Character data of variable length up to 2 gigabytes, or $2^{31}-1$ bytes. Provided for backward compatibility.
12	<b>DATE</b>	Valid date range from January 1, 4712 BC to December 31, 9999 AD. The default format is determined explicitly by the <b>NLS_DATE_FORMAT</b> parameter or implicitly by the <b>NLS_TERRITORY</b> parameter. The size is fixed at 7 bytes. This datatype contains the datetime fields <b>YEAR</b> , <b>MONTH</b> , <b>DAY</b> , <b>hour</b> , <b>MINUTE</b> , and <b>SECOND</b> . It does not have fractional seconds or a time zone.
21	<b>BINARY_FLOAT</b>	32-bit floating point number. This datatype requires 5 bytes, including the length byte.
22	<b>BINARY_DOUBLE</b>	64-bit floating point number. This datatype requires 9 bytes, including the length byte.
180	<b>TIMESTAMP</b> ( <i>fractional_seconds</i> )	Year, month, and day values of date, as well as hour, minute, and second values of time, where <i>fractional_seconds_precision</i> is the number of digits in the fractional part of the <b>SECOND</b> datetime field. Accepted values of <i>fractional_seconds_precision</i> are 0 to 9. The default is 6. The default format is determined explicitly by the <b>NLS_DATE_FORMAT</b> parameter or implicitly by the <b>NLS_TERRITORY</b> parameter. The size varies from 7 to 11 bytes, depending on the precision. This datatype contains the datetime fields <b>YEAR</b> , <b>MONTH</b> , <b>DAY</b> , <b>hour</b> , <b>MINUTE</b> , and <b>SECOND</b> . It contains fractional seconds but does not have a time zone.
181	<b>TIMESTAMP</b> ( <i>fractional_seconds</i> ) WITH <b>TIME ZONE</b>	All values of <b>TIMESTAMP</b> as well as time zone displacement value, where <i>fractional_seconds_precision</i> is the number of digits in the fractional part of the <b>SECOND</b> datetime field. Accepted values are 0 to 9. The default is 6. The default format is determined explicitly by the <b>NLS_DATE_FORMAT</b> parameter or implicitly by the <b>NLS_TERRITORY</b> parameter. The size is fixed at 13 bytes. This datatype contains the datetime fields <b>YEAR</b> , <b>MONTH</b> , <b>DAY</b> , <b>hour</b> , <b>MINUTE</b> , <b>SECOND</b> , <b>TIMEZONE_HOUR</b> , and <b>TIMEZONE_MINUTE</b> . It has fractional seconds and an explicit time zone.
231	<b>TIMESTAMP</b> ( <i>fractional_seconds</i> ) WITH <b>LOCAL TIME ZONE</b>	All values of <b>TIMESTAMP WITH TIME ZONE</b> , with the following exceptions:

Code	Datatype	Description
		<ul style="list-style-type: none"> <li>Data is normalized to the database time zone when it is stored in the database.</li> <li>When the data is retrieved, users see the data in the session time zone.</li> </ul> <p>The default format is determined explicitly by the <b>NLS_DATE_FORMAT</b> parameter or implicitly by the <b>NLS_TERRITORY</b> parameter. The sizes varies from 7 to 11 bytes, depending on the precision.</p>
182	<b>INTERVAL YEAR</b> [( <i>year_precision</i> )] <b>TO MONTH</b>	Stores a period of time in years and months, where <i>year_precision</i> is the number of digits in the <b>YEAR</b> datetime field. Accepted values are 0 to 9. The default is 2. The size is fixed at 5 bytes.
183	<b>INTERVAL DAY</b> [( <i>day_precision</i> )] <b>TO SECOND</b> [( <i>fractional_seconds</i> )]	Stores a period of time in days, hours, minutes, and seconds, where <ul style="list-style-type: none"> <li><i>day_precision</i> is the maximum number of digits in the <b>DAY</b> datetime field. Accepted values are 0 to 9. The default is 2.</li> <li><i>fractional_seconds_precision</i> is the number of digits in the fractional part of the <b>SECOND</b> field. Accepted values are 0 to 9. The default is 6.</li> </ul> <p>The size is fixed at 11 bytes.</p>
23	<b>RAW</b> ( <i>size</i> )	Raw binary data of length <i>size</i> bytes. Maximum <i>size</i> is 2000 bytes. You must specify <i>size</i> for a <b>RAW</b> value.
24	<b>LONG RAW</b>	Raw binary data of variable length up to 2 gigabytes.
69	<b>ROWID</b>	Base 64 string representing the unique address of a row in its table. This datatype is primarily for values returned by the <b>ROWID</b> pseudocolumn.
208	<b>UROWID</b> [( <i>size</i> )]	Base 64 string representing the logical address of a row of an index-organized table. The optional <i>size</i> is the size of a column of type <b>UROWID</b> . The maximum size and default is 4000 bytes.
96	<b>CHAR</b> [( <i>size</i> [ <b>BYTE</b>   <b>CHAR</b> ])]	Fixed-length character data of length <i>size</i> bytes. Maximum <i>size</i> is 2000 bytes or characters. Default and minimum <i>size</i> is 1 byte.  <b>BYTE</b> and <b>CHAR</b> have the same semantics as for <b>VARCHAR2</b> .
96	<b>NCHAR</b> [( <i>size</i> )]	Fixed-length character data of length <i>size</i> characters. The number of bytes can be up to two times <i>size</i> for <b>AL16UTF16</b> encoding and three times <i>size</i> for <b>UTF8</b> encoding. Maximum <i>size</i> is determined by the national character set definition, with an upper limit of 2000 bytes. Default and minimum <i>size</i> is 1 character.
112	<b>CLOB</b>	A character large object containing single-byte or multibyte characters. Both fixed-width and variable-width character sets are supported, both using the database character set. Maximum size is (4 gigabytes - 1) * (database block size).
112	<b>NCLOB</b>	A character large object containing Unicode characters. Both fixed-width and variable-width character sets are supported, both using the database national character set. Maximum size is (4 gigabytes - 1) * (database block size). Stores national character set data.
113	<b>BLOB</b>	A binary large object. Maximum size is (4 gigabytes - 1) * (database block size).

Code	Datatype	Description
114	BFILE	Contains a locator to a large binary file stored outside the database. Enables byte stream I/O access to external LOBs residing on the database server. Maximum size is 4 gigabytes.

#### • Index

An index is a structure created to help retrieve data more quickly and efficiently (just as the index in this book allows you to find a particular section more quickly). An index is declared on a column or set of columns. Access to the table based on the value of the indexed column(s) (as in a WHERE clause) will use the index to locate the table data.

```
CREATE INDEX t_fuel_bonus_INDIX ON t_fuel_bonus (BONUS_ITEM);
```



*Note: Oracle table and column names are not case sensitive, .i.e. for oracle , t\_fuel\_bonus or T\_FUEL\_BONUS are the same table.*

#### • View

A view is a window into one or more tables. A view does not store any data; it presents table data. A view can be queried, updated, and deleted as a table without restriction. Views are typically used to simplify the user's perception of data access by providing limited information from one table, or a set of information from several tables transparently. Views can also be used to prevent some data from being accessed by the user or to create a join from multiple tables.

```
CREATE VIEW Big_bunus_view
AS SELECT *
FROM t_fuel_bonus
WHERE Purchase_Amt > 20000
```

In the above example, we have created a view on the table t\_fuel\_bonus which holds all the bonus details for purchase amounts > 20000



#### **TASK6: View**

Insert some data into the table t\_fuel\_bonus, which has Purchase Amount > 20000 and some data with less value.

Now select the data from the table and then select the data from the view. The data from the view should show only the rows with > 20000 value.



#### **TASK7: View**

Describe the view:

```
SQL> Desc Big_bunus_view
```

Insert some data into the view – use the same insert statement as if it was a table, then check , if the data has gone into the table and the view both.

1 • **Stored procedure**--A stored procedure is a predefined SQL query that is stored in the data dictionary. Stored procedures are designed to allow more efficient queries. Using stored procedures, you can reduce the amount of information that must be passed to the RDBMS and thus reduce network traffic and improve performance.

• **Database trigger**--A database trigger is a procedure that is run automatically when an event occurs. This procedure, which is defined by the administrator or developer, triggers, or is run whenever this event occurs. This procedure could be an insert, a deletion, or even a selection of data from a table.

• **Sequence**--The Oracle sequence generator is used to automatically generate a unique sequence of numbers in cache. By using the sequence generator you can avoid the steps necessary to create this sequence on your own such as locking the record that has the last value of the sequence, generating a new value, and then unlocking the record.

```
CREATE SEQUENCE my_seq  
MINVALUE 1  
MAXVALUE 9999999  
START WITH 1  
INCREMENT BY 1  
CACHE 20;
```

To use the sequence, you can use my\_seq.NEXT\_VAL.

• **Synonym** -- A synonym is an alias for one of the following objects:

- ◆ table
- ◆ object table
- ◆ view
- ◆ object view
- ◆ sequence
- ◆ stored procedure
- ◆ stored function
- ◆ package
- ◆ materialized view
- ◆ java class
- ◆ user defined object type
- ◆ another synonym

The object does not need to exist at the time of its creation CREATE SYNONYM offices FOR hr.locations;

## TRANSACTIONS IN ORACLE

A transaction is a logical unit of work that contains one or more SQL statements. A transaction is an atomic unit; the effects of all the SQL statements in a transaction can be either all committed (applied to the database) or all rolled back (undone from the database).

A transaction begins with the first executable SQL statement. A transaction ends when it is committed or rolled back, either explicitly (with a COMMIT or ROLLBACK statement) or implicitly (when a DDL statement is issued).

A SQL statement that "executes successfully" is different from a "committed" transaction. Executing successfully means that a single statement was parsed and found to be a valid SQL construction, and that the entire statement executed without error as an atomic unit (for example, all rows of a multirow update are changed). However, until the transaction that contains the statement is committed, the transaction can be rolled back, and all of the changes of the statement can be undone.

A statement, rather than a transaction, executes successfully. Committing means that a user has said either explicitly or implicitly "make the changes in this transaction permanent". The changes made by the SQL statement(s) of your transaction become permanent and visible to other users only after your transaction has been committed. Only other users' transactions that started after yours will see the committed changes. If at any time during execution a SQL statement causes an error, all effects of the statement are rolled back. The effect of the rollback is as if that statement were never executed. This is a statement-level rollback.

A SQL statement that fails causes the loss only of any work it would have performed itself; it does not cause the loss of any work that preceded it in the current transaction. If the statement is a DDL statement, the implicit commit that immediately preceded it is not undone.

A transaction in Oracle begins when the first executable SQL statement is encountered. An executable SQL statement is a SQL statement that generates calls to an instance, including DML and DDL statements. A transaction ends when any of the following occurs:

- You issue a COMMIT or ROLLBACK (without a SAVEPOINT clause) statement.
- You execute a DDL statement (such as CREATE, DROP, RENAME, ALTER). If the current transaction contains any DML statements, Oracle first commits the transaction, and then executes and commits the DDL statement as a new, single statement transaction.
- A user disconnects from Oracle. (The current transaction is committed.)
- A user process terminates abnormally. (The current transaction is rolled back.)

After one transaction ends, the next executable SQL statement automatically starts the following transaction.

Note: Applications should always explicitly commit or roll back transactions before program termination.

In the below example, the company decides to give mobile phone as gift to all fuel transactions between Rs 20000 and 30000. We might have hundreds of such row in the table, but this update can be achieved with a single transaction as below:

Let us create some rows into the table for testing purpose.

```
into T_FUEL_BONUS values( 1000,'Executive Dairy');
into T_FUEL_BONUS values( 2000,'Executive Dairy');
into T_FUEL_BONUS values( 3000,'Executive Dairy');
into T_FUEL_BONUS values( 20000,'Executive Dairy');
into T_FUEL_BONUS values( 21000,'Executive Dairy');
into T_FUEL_BONUS values( 25000,'Executive Dairy');
into T_FUEL_BONUS values( 30000,'Executive Dairy');
```

```
update t_fuel_bonus
set bonus_item = 'Mobile Phone'
where
PURCHASE_AMT between 20000 and 30000;
/
commit;
```



### **TASK8: DML Statement practice**

Write sql statements for the following:

1. update bonus\_item to null , for all purchase less than Rs 500
2. The entry clerk made a mistake while entering few rows, he entered bounus\_item as 'Executive Dairy' , the correct spelling should be 'Executive Diary', write an update statement to fix this error



## OVERVIEW OF PL/SQL

Oracle first introduced Procedural Language/Structured Query Language (PL/SQL) in version 6.0 of its relational database management system (RDBMS). As its RDBMS evolved, Oracle made developmental changes to the PL/SQL language by introducing new features and enhancing existing features. As of Oracle9i, the version of PL/SQL is PL/SQL 9.2 or 9.0 depending on whether it is Oracle9i Release 2 (9.2.x) or Oracle9i Release 1 (9.0.x). In this book, I refer to both versions collectively as PL/SQL 9i.

PL/SQL incorporates third-generation language (3GL) structures otherwise unavailable in Structured Query Language (SQL). SQL is a fourth-generation language (4GL), meaning it uses constructs and elements that specify "what to do" without having to specify "how to do it." It's a major language for the Oracle RDBMS (as well as for other RDBMSs), and it's used for data definition, database querying, and data manipulation and control. However, there are situations that demand the use of 3GL constructs, such as conditional or iterative execution of SQL and the like. This kind of logic and control flow can be achieved only in a 3GL language such as Java, C++, or C. To accommodate 3GL features, Oracle designed and implemented the PL/SQL language as a procedural extension to SQL. PL/SQL is integrated with SQL, and both SQL and PL/SQL serve as the major database server-side languages, with each language complementing the other. You can also use PL/SQL in client-side environments.

### Block Structure

PL/SQL is a block-structured language. That is, the basic units (procedures, functions, and anonymous blocks) that make up a PL/SQL program are logical blocks, which can contain any number of nested sub-blocks. Typically, each logical block corresponds to a problem or subproblem to be solved. Thus, PL/SQL supports the divide-and-conquer approach to problem solving called stepwise refinement.

A block (or sub-block) lets you group logically related declarations and statements. That way, you can place declarations close to where they are used. The declarations are local to the block and cease to exist when the block completes. As Figure below shows, a PL/SQL block has three parts: a declarative part, an executable part, and an exception-handling part. (In PL/SQL, a warning or error condition is called an exception.) Only the executable part is required. The order of the parts is logical. First comes the declarative part, in which items can be declared. Once declared, items can be manipulated in the executable part. Exceptions raised during execution can be dealt with in the exception-handling part.

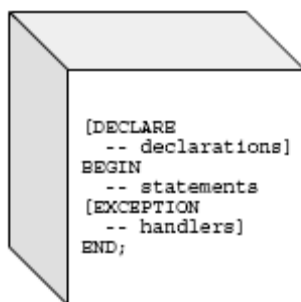
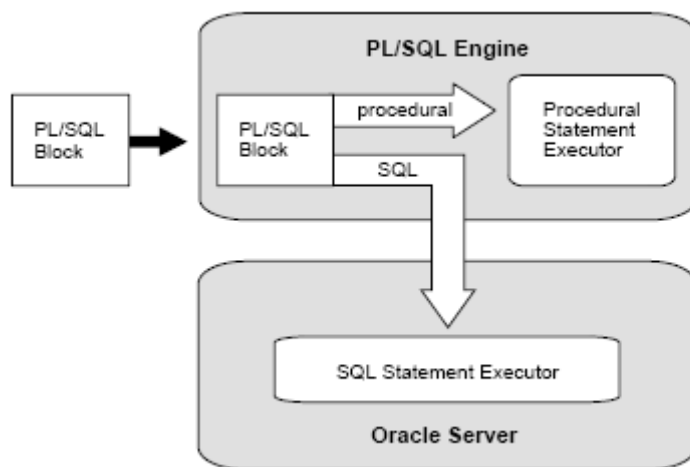


Figure - Block Structure

You can nest sub-blocks in the executable and exception-handling parts of a PL/SQL block or subprogram but not in the declarative part. Also, you can define local subprograms in the declarative part of any block. However, you can call local subprograms only from the block in which they are defined.

## PL/SQL Architecture

The PL/SQL compilation and run-time system is a technology, not an independent product. Think of this technology as an engine that compiles and executes PL/SQL blocks and subprograms. The engine can be installed in an Oracle server or in an application development tool such as Oracle Forms or Oracle Reports. The engine executes procedural statements but sends SQL statements to the SQL Statement Executor in the Oracle server. Figure shows the PL/SQL engine processing an anonymous block.



## PL/SQL DATATYPES

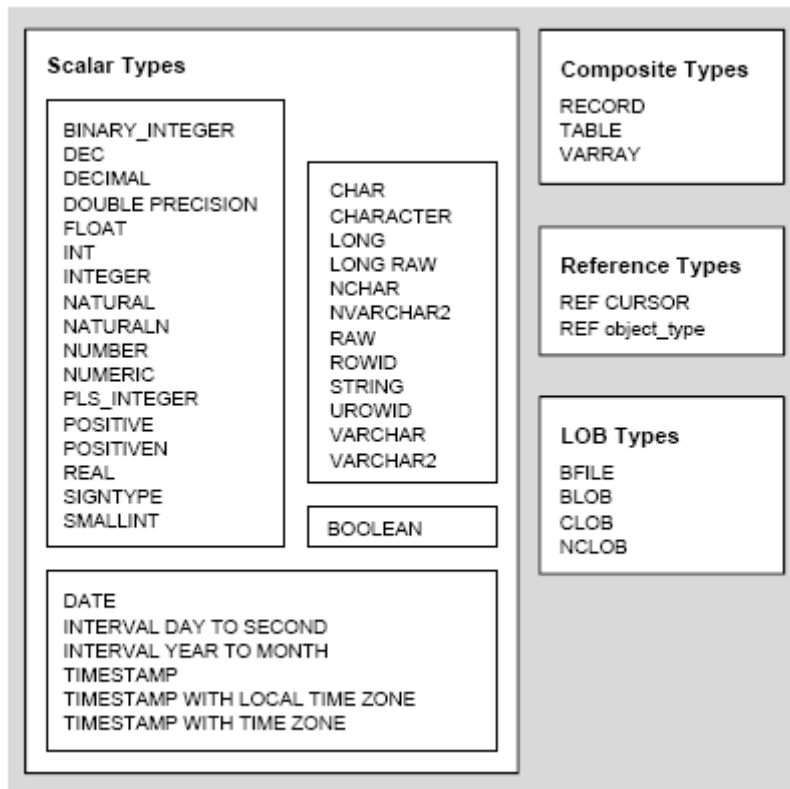
Every constant, variable, and parameter has a datatype (or type), which specifies a storage format, constraints, and valid range of values. PL/SQL provides a variety of predefined datatypes. For instance, you can choose from integer, floating point, character, Boolean, date, collection, reference, and LOB types. In addition, PL/SQL lets you define your own subtypes.

### Predefined Datatypes

A scalar type has no internal components. A composite type has internal components that can be manipulated individually. A reference type holds values, called pointers, that designate other program items. A LOB type holds values, called lob locators, that specify the location of large objects (graphic images for example) stored out-of-line.

Figure below shows the predefined datatypes available for your use. The scalar types fall into four families, which store number, character, Boolean, and date/time data, respectively.

Figure - Built-in Datatypes



## Number Types:

Number types let you store numeric data (integers, real numbers, and floating-point numbers), represent quantities, and do calculations.

### BINARY\_INTEGER

You use the BINARY\_INTEGER datatype to store signed integers. Its magnitude range is  $-2^{31}$  ..  $2^{31}$ .

### NUMBER

You use the NUMBER datatype to store fixed-point or floating-point numbers. Its magnitude range is  $1E-130$  ..  $10E125$ . You can specify precision, which is the total number of digits, and scale, which is the number of digits to the right of the decimal point. The syntax follows:

NUMBER[(precision,scale)]

To declare fixed-point numbers, for which following form: you must specify scale, use the

NUMBER(precision,scale)

To declare floating-point numbers, for which you cannot specify precision or scale because the decimal point can "float" to any position, use the following form:

NUMBER

To declare integers, which have no decimal point, use this form:

NUMBER(*precision*) -- same as NUMBER(*precision*,0)

## NUMBER Subtypes

You can use the following NUMBER subtypes for compatibility with ANSI/ISO and IBM types or when you want a more descriptive name:

DEC  
DECIMAL  
DOUBLE PRECISION  
FLOAT  
INTEGER  
INT  
NUMERIC  
REAL  
SMALLINT

Use the subtypes DEC, DECIMAL, and NUMERIC to declare fixed-point numbers with a maximum precision of 38 decimal digits.

Use the subtypes DOUBLE PRECISION and FLOAT to declare floating-point numbers with a maximum precision of 126 binary digits, which is roughly equivalent to 38 decimal digits. Or, use the subtype REAL to declare floating-point numbers with a maximum precision of 63 binary digits, which is roughly equivalent to 18 decimal digits. Use the subtypes INTEGER, INT, and SMALLINT to declare integers with a maximum precision of 38 decimal digits.

### PLS\_INTEGER

You use the PLS\_INTEGER datatype to store signed integers. Its magnitude range is  $-2^{31}$  ..  $2^{31}$ .

Although PLS\_INTEGER and BINARY\_INTEGER have the same magnitude range, they are not fully compatible. When a PLS\_INTEGER calculation overflows, an exception is raised. However, when a BINARY\_INTEGER calculation overflows, no exception is raised if the result is assigned to a NUMBER variable.

Because of this small semantic difference, you might want to continue using BINARY\_INTEGER in old applications for compatibility. In new applications, always use PLS\_INTEGER for better performance.

## Character Types:

Character types let you store alphanumeric data, represent words and text, and manipulate character strings.

### CHAR

You use the CHAR datatype to store fixed-length character data. The CHAR datatype takes an optional parameter that lets you specify a maximum size up to 32767 bytes. The syntax follows:

CHAR[(maximum\_size [CHAR | BYTE] )]

If you do not specify a maximum size, it defaults to 1.

### VARCHAR2

You use the VARCHAR2 datatype to store variable-length character data. The VARCHAR2 datatype takes a required parameter that specifies a maximum size up to 32767 bytes. The syntax follows:

```
VARCHAR2(maximum_size [CHAR | BYTE])
```

If one value in a comparison has datatype VARCHAR2 and the other value has datatype CHAR, nonblank-padding semantics are used. But, remember, when you assign a character value to a CHAR variable, if the value is shorter than the declared length of the variable, PL/SQL blank-pads the value to the declared length. So, given the declarations

```
last_name1 VARCHAR2(10) := 'STAUB';
last_name2 CHAR(10) := 'STAUB'; -- PL/SQL blank-pads value
```

the following IF condition is false because the value of last\_name2 includes five trailing blanks:

```
IF last_name1 = last_name2 THEN ...
```

### LONG and LONG RAW

You use the LONG datatype to store variable-length character strings. The LONG datatype is like the VARCHAR2 datatype, except that the maximum size of a LONG value is 32760 bytes.

You use the LONG RAW datatype to store binary data or byte strings. LONG RAW data is like LONG data, except that LONG RAW data is not interpreted by PL/SQL. The maximum size of a LONG RAW value is 32760 bytes. LONG columns can store text, arrays of characters, or even short documents. You can reference LONG columns in UPDATE, INSERT, and (most) SELECT statements, but not in expressions, SQL function calls, or certain SQL clauses such as WHERE, GROUP BY, and CONNECT BY.

### ROWID

Internally, every database table has a ROWID pseudocolumn, which stores binary values called rowids. Each rowid represents the storage address of a row. A physical rowid identifies a row in an ordinary table. For example, in SQL\*Plus (which implicitly converts rowids into character strings), the query

```
SQL> SELECT rowid, ename FROM emp WHERE empno = 7788;
```

might return the following row:

```
ROWID                ENAME
-----
AAAAQcAABAAADFNAAH  SCOTT
```

## Boolean Type:

### BOOLEAN

You use the BOOLEAN datatype to store the logical values TRUE, FALSE, and NULL (which stands for a missing, unknown, or inapplicable value). Only logic operations are allowed on BOOLEAN variables.

## Datetime and Interval Types:

The datatypes in this section let you store and manipulate dates, times, and intervals (periods of time). A variable that has a date/time datatype holds values called *datetimes*; a variable that has an interval datatype holds values called *intervals*. A datetime or interval consists of fields, which determine its value. The following list shows the valid values for each field:

Field Name	Valid Datetime Values	Valid Interval Values
YEAR	-4712 to 9999 (excluding year 0)	Any nonzero integer
MONTH	01 to 12	0 to 11
DAY	01 to 31 (limited by the values of MONTH and YEAR, according to the rules of the calendar	Any nonzero integer

Field Name	Valid Datetime Values	Valid Interval Values
	for the locale)	
HOUR	00 to 23	0 to 23
MINUTE	00 to 59	0 to 59
SECOND	00 to 59.9(n), where 9(n) is the precision of time fractional seconds	0 to 59.9(n), where 9(n) is the precision of interval fractional seconds
TIMEZONE_HOUR	-12 to 14 (range accommodates daylight savings time changes)	Not applicable
TIMEZONE_MINUTE	00 to 59	Not applicable
TIMEZONE_REGION	Found in the view V\$TIMEZONE_NAMES.	Not applicable.
TIMEZONE_ABBR	Found in the view V\$TIMEZONE_NAMES.	Not applicable.

## DATE

You use the DATE datatype to store fixed-length datetimes, which include the time of day in seconds since midnight. The date portion defaults to the first day of the current month; the time portion defaults to midnight. The date function SYSDATE returns the current date and time.

**Tip:** To compare dates for equality, regardless of the time portion of each date, use the function result `TRUNC(date_variable)` in comparisons, `GROUP BY` operations, and so on.

You can add and subtract dates. In arithmetic expressions, PL/SQL interprets integer literals as days. For instance, `SYSDATE + 1` is tomorrow.

## **TIMESTAMP**

The datatype `TIMESTAMP`, which extends the datatype `DATE`, stores the year, month, day, hour, minute, and second. The syntax is:

`TIMESTAMP[(precision)]`

where the optional parameter *precision* specifies the number of digits in the fractional part of the seconds field. You cannot use a symbolic constant or variable to specify the precision; you must use an integer literal in the range 0 .. 9. The default is 6. In the following example, you declare a variable of type `TIMESTAMP`, then assign a literal value to it:

```
DECLARE
checkout TIMESTAMP(3);
BEGIN
checkout := '1999-06-22 07:48:53.275';
...
END;
```

In this example, the fractional part of the seconds field is 0.275.



## Datatype Conversion

PL/SQL supports both explicit and implicit (automatic) datatype conversion. This section explains the difference and suggests when and where to use each type. One of the most frequent conversions is a DATE format to a CHAR format, and the reverse. We'll use this as our example for this discussion.

### Explicit Conversion

Oracle has provided built-in functions to use for converting one datatype to another. The most commonly used conversion functions are the TO\_DATE function, which allows you to convert a CHAR value to a DATE value, and the TO\_CHAR function, which allows you to convert from a DATE value to a CHAR value. An explicit conversion occurs when you compare a DATE value to a CHAR value and use the function TO\_DATE or TO\_CHAR to convert the DATE into a CHAR or the CHAR into a DATE. The following example shows how to accomplish this conversion:

```
Select count(1)
From pa_projects_all
Where start_date >= to_date('01-JAN-2000','DD-MON-YYYY') AND
completion_date <= to_date('01-JAN-2001','DD-MON-YYYY')
/
```

This converts the values '01-JAN-2000' and '01-JAN-2001' into a DATE datatype of format 'ddmon-yyyy' and compares it to the value of the start\_date column. If a match is made, then the counter is incremented and the results will show how many rows exist that meet the criteria.

### Implicit Conversion

When you create code that requires conversion of datatypes and you do not use the conversion function, PL/SQL attempts to convert the datatype of a value implicitly. With this feature, you can use literals, variables, and parameters of one datatype while another datatype is expected. In order to accomplish an implicit conversion, you need only compare one datatype to another. The following two examples accomplish the same task and return the same results, though one is an explicit conversion and the other is implicit. First, the explicit conversion example:

```
Select count(1)
From pa_projects_all
Where start_date >= to_date('01-JAN-2000','DD-MON-YYYY') AND
completion_date <= to_date('01-JAN-2001','DD-MON-YYYY')
/
COUNT(1)
-----
958
```

The result in this case is the number of projects that started on or before January 1, 2000, and have a completion date on or before the January 1, 2001. As you can see, we actually converted one of the dates and left the other as a CHAR. The implicit conversion yields the same result, as follows:

```
SELECT count(1)
FROM pa_projects_all
WHERE start_date >= '01-JAN-2000' AND
```

```
completion_date <= '01-JAN-2001'  
/  
COUNT(1)  
-----  
958
```

You can see here that Oracle will convert the data implicitly or allow you to convert the data yourself. Both scenarios return the same results. It is your responsibility to ensure that values are convertible. For instance, PL/SQL can convert the CHAR value '01-JAN-2000' to a DATE value, but it cannot convert the CHAR value TOMORROW' to a DATE value. Similarly, PL/SQL cannot convert a VARCHAR2 value containing alphabetic characters to a NUMBER value. be easier to maintain.

## DATE Values

When you consider what Oracle achieves when it converts a DATE column into a VARCHAR2, you'll realize that the implicit conversion gets the same result that you would get. Oracle must convert the internal value to a character value. PL/SQL does exactly what you would do in your code and uses the function TO\_CHAR, with the default date format set up in NLS (National Language Support). In order to convert the data to time or any other format, you must make the call TO\_CHAR with the desired format mask.

Likewise, Oracle changes the VARCHAR2 into the proper date mask for conversion into a DATE value. Again, keep in mind that Oracle uses the default NLS mask to accomplish the conversion. PL/SQL calls the function TO\_DATE, which expects the default date format. To insert dates in other formats, you must call TO\_DATE with a format mask.

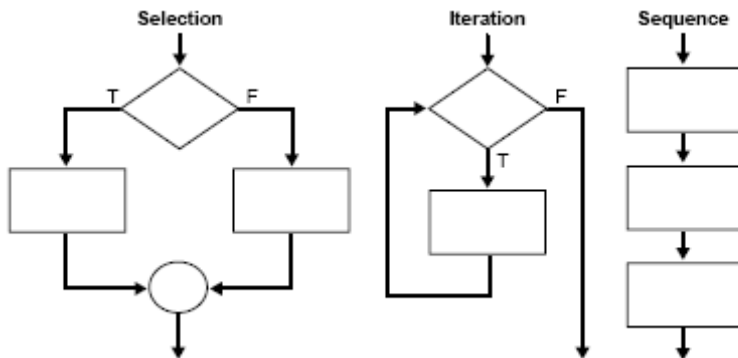
## PL/SQL CONTROL STRUCTURES

This section discusses the following topics:

- Overview of PL/SQL Control Structures
- Conditional Control: IF and CASE Statements
- Iterative Control: LOOP and EXIT Statements
- Sequential Control: GOTO and NULL Statements

### Overview of PL/SQL Control Structures

According to the structure theorem, any computer program can be written using the basic control structures shown in Figure. They can be combined in any way necessary to deal with a given problem.



The selection structure tests a condition, and then executes one sequence of statements instead of another, depending on whether the condition is true or false. A condition is any variable or expression that returns a Boolean value (TRUE or FALSE). The iteration structure executes a sequence of statements repeatedly as long as a condition holds true. The sequence structure simply executes a sequence of statements in the order in which they occur.

## Conditional Control: IF and CASE Statements

IF statement lets you execute a sequence of statements conditionally. There are three forms of IF statements: IF-THEN, IF-THEN-ELSE, and IF-THEN-ELSIF. The CASE statement is a compact way to evaluate a single condition and choose between many alternative actions.

### IF-THEN Statement

The simplest form of IF statement associates a condition with a sequence of statements enclosed by the keywords THEN and END IF (not ENDIF), as follows:

```
IF condition THEN
sequence_of_statements
END IF;
```

The sequence of statements is executed only if the condition is true. If the condition is false or null, the IF statement does nothing. In either case, control passes to the next statement. An example follows:

```
IF sales > quota THEN

    compute_bonus(empid);

    UPDATE payroll SET pay = pay + bonus WHERE empno = emp_id;

END IF;
```

You might want to place brief IF statements on a single line, as in

```
IF x > y THEN high := x; END IF;
```

### IF-THEN-ELSE Statement

The second form of IF statement adds the keyword ELSE followed by an alternative sequence of statements, as follows:

```
IF condition THEN
    sequence_of_statements1
ELSE
    sequence_of_statements2
END IF;
```

The sequence of statements in the ELSE clause is executed only if the condition is false or null. Thus, the ELSE clause ensures that a sequence of statements is executed. In the following example, the first UPDATE statement is executed when the condition is true, but the second UPDATE statement is executed when the condition is false or null:

```
IF trans_type = 'CR' THEN
    UPDATE accounts SET balance = balance + credit WHERE ...
ELSE
    UPDATE accounts SET balance = balance - debit WHERE ...
END IF;
```

The THEN and ELSE clauses can include IF statements. That is, IF statements can be nested, as the following example shows:

```
IF trans_type = 'CR' THEN
    UPDATE accounts SET balance = balance + credit WHERE ...
ELSE
    IF new_balance >= minimum_balance THEN
        UPDATE accounts SET balance = balance - debit WHERE ...
    ELSE
        RAISE insufficient_funds;
    END IF;
END IF;
```

### **IF-THEN-ELSIF Statement**

Sometimes you want to select an action from several mutually exclusive alternatives. The third form of IF statement uses the keyword ELSIF (not ELSEIF) to introduce additional conditions, as follows:

```
IF condition1 THEN
    sequence_of_statements1
ELSIF condition2 THEN
    sequence_of_statements2
ELSE
    sequence_of_statements3
END IF;
```

If the first condition is false or null, the ELSIF clause tests another condition. An IF statement can have any number of ELSIF clauses; the final ELSE clause is optional. Conditions are evaluated one by one from top to bottom. If any condition is true, its associated sequence of statements is executed and control passes to the next statement. If all conditions are false or null, the sequence in the ELSE clause is executed. Consider the following example:

```
BEGIN
...
IF sales > 50000 THEN
    bonus := 1500;
ELSIF sales > 35000 THEN
    bonus := 500;
ELSE
    bonus := 100;
END IF;
    INSERT INTO payroll VALUES (emp_id, bonus, ...);
END;
```

## CASE Statement

Like the IF statement, the CASE statement selects one sequence of statements to execute. However, to select the sequence, the CASE statement uses a selector rather than multiple Boolean expressions. To compare the IF and CASE statements, consider the following code that outputs descriptions of school grades:

```
IF grade = 'A' THEN
    dbms_output.put_line('Excellent');
ELSIF grade = 'B' THEN
    dbms_output.put_line('Very Good');
ELSIF grade = 'C' THEN
    dbms_output.put_line('Good');
ELSIF grade = 'D' THEN
    dbms_output.put_line('Fair');
ELSIF grade = 'F' THEN
    dbms_output.put_line('Poor');
ELSE
    dbms_output.put_line('No such grade');
END IF;
```

Notice the five Boolean expressions. In each instance, we test whether the same variable, grade, is equal to one of five values: 'A', 'B', 'C', 'D', or 'F'. Let us rewrite the preceding code using the CASE statement, as follows:

CASE grade

```
    WHEN 'A' THEN dbms_output.put_line('Excellent');
    WHEN 'B' THEN dbms_output.put_line('Very Good');
    WHEN 'C' THEN dbms_output.put_line('Good');
    WHEN 'D' THEN dbms_output.put_line('Fair');
    WHEN 'F' THEN dbms_output.put_line('Poor');
ELSE dbms_output.put_line('No such grade');
END CASE;
```

The CASE statement is more readable and more efficient. So, when possible, rewrite lengthy IFTHEN-ELSIF statements as CASE statements.

## Iterative Control: LOOP and EXIT Statements

LOOP statements let you execute a sequence of statements multiple times. There are three forms of LOOP statements: LOOP, WHILE-LOOP, and FOR-LOOP.

### LOOP

The simplest form of LOOP statement is the basic (or infinite) loop, which encloses a sequence of statements between the keywords LOOP and END LOOP, as follows:

```
LOOP
    sequence_of_statements
END LOOP;
```

With each iteration of the loop, the sequence of statements is executed, then control resumes at the top of the loop. If further processing is undesirable or impossible, you can use an EXIT statement to complete the loop. You can place one or more EXIT statements anywhere inside a loop, but nowhere outside a loop. There are two forms of EXIT statements: EXIT and EXIT-WHEN.

### EXIT

The EXIT statement forces a loop to complete unconditionally. When an EXIT statement is encountered, the loop completes immediately and control passes to the next statement. An example follows:

```
LOOP
    ...
    IF credit_rating < 3 THEN
        ...
        EXIT; -- exit loop immediately
    END IF;
END LOOP;
```

-- control resumes here

The next example shows that you cannot use the EXIT statement to complete a PL/SQL block:

```
BEGIN
...
    IF credit_rating < 3 THEN
        ...
        EXIT; -- not allowed
    END IF;
END;
```

Remember, the EXIT statement must be placed inside a loop. To complete a PL/SQL block before its normal end is reached, you can use the RETURN statement.

## EXIT-WHEN

The EXIT-WHEN statement lets a loop complete conditionally. When the EXIT statement is encountered, the condition in the WHEN clause is evaluated. If the condition is true, the loop completes and control passes to the next statement after the loop. An example follows:

```
LOOP
    FETCH c1 INTO ...
    EXIT WHEN c1%NOTFOUND; -- exit loop if condition is true
    ...
END LOOP;

CLOSE c1;
```

Until the condition is true, the loop cannot complete. So, a statement inside the loop must change the value of the condition. In the last example, if the FETCH statement returns a row, the condition is false. When the FETCH statement fails to return a row, the condition is true, the loop completes, and control passes to the CLOSE statement. The EXIT-WHEN statement replaces a simple IF statement. For example, compare the following statements:

```
IF count > 100 THEN
    EXIT;
END IF;
```

These statements are logically equivalent, but the EXIT-WHEN statement is easier to read and understand.

## Loop Labels

Like PL/SQL blocks, loops can be labeled. The label, an undeclared identifier enclosed by double angle brackets, must appear at the beginning of the LOOP statement, as follows:

```
<<label_name>>
LOOP
    sequence_of_statements
END LOOP;
```

Optionally, the label name can also appear at the end of the LOOP statement, as the following example shows:

```
<<my_loop>>
LOOP
    ...
END LOOP my_loop;
```



When you nest labeled loops, use ending label names to improve readability. With either form of EXIT statement, you can complete not only the current loop, but any enclosing loop. Simply label the enclosing loop that you want to complete. Then, use the label in an EXIT statement, as follows:

```
<<outer>>
LOOP
    ...
    LOOP
    ..
    EXIT outer WHEN ... -- exit both loops
    END LOOP;
    ...
END LOOP outer;
```

Every enclosing loop up to and including the labeled loop is exited.

## WHILE-LOOP

The WHILE-LOOP statement associates a condition with a sequence of statements enclosed by the keywords LOOP and END LOOP, as follows:

```
WHILE condition LOOP
    sequence_of_statements
END LOOP;
```

Before each iteration of the loop, the condition is evaluated. If the condition is true, the sequence of statements is executed, then control resumes at the top of the loop. If the condition is false or null, the loop is bypassed and control passes to the next statement. An example follows:

```
WHILE total <= 25000 LOOP
    ...
    SELECT sal INTO salary FROM emp WHERE ...
    total := total + salary;
END LOOP;
```

The number of iterations depends on the condition and is unknown until the loop completes. The condition is tested at the top of the loop, so the sequence might execute zero times. In the last example, if the initial value of total is larger than 25000, the condition is false and the loop is bypassed.

Some languages have a LOOP UNTIL or REPEAT UNTIL structure, which tests the condition at the bottom of the loop instead of at the top. Therefore, the sequence of statements is executed at least once. PL/SQL has no such structure, but you can easily build one, as follows:

```
LOOP
    sequence_of_statements
    EXIT WHEN boolean_expression;
END LOOP;
```

To ensure that a WHILE loop executes at least once, use an initialized Boolean variable in the condition, as follows:

```
done := FALSE;
WHILE NOT done LOOP
    sequence_of_statements
    done := boolean_expression;
END LOOP;
```

A statement inside the loop must assign a new value to the Boolean variable. Otherwise, you have an infinite loop. For example, the following LOOP statements are logically equivalent:

```
WHILE TRUE LOOP
    .....
END LOOP;
```

## FOR-LOOP

Whereas the number of iterations through a WHILE loop is unknown until the loop completes, the number of iterations through a FOR loop is known before the loop is entered. FOR loops iterate over a specified range of integers. The range is part of an iteration scheme, which is enclosed by the keywords FOR and LOOP. A double dot (..) serves as the range operator. The syntax follows:

```
FOR counter IN [REVERSE] lower_bound..higher_bound LOOP
    sequence_of_statements
END LOOP;
```

The range is evaluated when the FOR loop is first entered and is never re-evaluated. As the next example shows, the sequence of statements is executed once for each integer in the range. After each iteration, the loop counter is incremented.

```
FOR i IN 1..3 LOOP -- assign the values 1,2,3 to i
    sequence_of_statements -- executes three times
END LOOP;
```

The following example shows that if the lower bound equals the higher bound, the sequence of statements is executed once:

```
FOR i IN 3..3 LOOP -- assign the value 3 to i
    sequence_of_statements -- executes one time
END LOOP;
```

By default, iteration proceeds upward from the lower bound to the higher bound. However, as the example below shows, if you use the keyword REVERSE, iteration proceeds downward from the higher bound to the lower bound. After each iteration, the loop counter is decremented. Nevertheless, you write the range bounds in ascending (not descending) order.

```
FOR i IN REVERSE 1..3 LOOP -- assign the values 3,2,1 to i
    sequence_of_statements -- executes three times
END LOOP;
```

The bounds of a loop range can be literals, variables, or expressions but must evaluate to numbers. Otherwise, PL/SQL raises the predefined exception VALUE\_ERROR. The lower bound need not be 1, as the examples below show. However, the loop counter increment (or decrement) must be 1.

j IN -5..5

### Dynamic Ranges

PL/SQL lets you determine the loop range dynamically at run time, as the following example shows:

```
SELECT COUNT(empno) INTO emp_count FROM emp;
FOR i IN 1..emp_count LOOP
...
END LOOP;
```

The value of emp\_count is unknown at compile time; the SELECT statement returns the value at run time.

What happens if the lower bound of a loop range evaluates to a larger integer than the upper bound? As the next example shows, the sequence of statements within the loop is not executed and control passes to the next statement:

```
-- limit becomes 1
FOR i IN 2..limit LOOP
    sequence_of_statements -- executes zero times
END LOOP;
-- control passes here
```

### Scope Rules

The loop counter is defined only within the loop. You cannot reference it outside the loop. After the loop is exited, the loop counter is undefined, as the following example shows:

```
FOR ctr IN 1..10 LOOP
...
END LOOP;
sum := ctr - 1; -- not allowed
```

You need not explicitly declare the loop counter because it is implicitly declared as a local variable of type INTEGER.

## Using the EXIT Statement

The EXIT statement lets a FOR loop complete prematurely. For example, the following loop normally executes ten times, but as soon as the FETCH statement fails to return a row, the loop completes no matter how many times it has executed:

```
FOR j IN 1..10 LOOP
    FETCH c1 INTO emp_rec;
    EXIT WHEN c1%NOTFOUND;
    ...
END LOOP;
```

Suppose you must exit from a nested FOR loop prematurely. You can complete not only the current loop, but any enclosing loop. Simply label the enclosing loop that Sequential Control: GOTO and NULL Statements you want to complete. Then, use the label in an EXIT statement to specify which FOR loop to exit, as follows:

```
<<outer>>
FOR i IN 1..5 LOOP
    ...
    FOR j IN 1..10
        LOOP
            FETCH c1 INTO emp_rec;
            EXIT outer WHEN c1%NOTFOUND; -- exit both FOR loops
            ...
        END LOOP;
    END LOOP outer;
```

control passes here

## Sequential Control: GOTO and NULL Statements

Unlike the IF and LOOP statements, the GOTO and NULL statements are not crucial to PL/SQL programming. The structure of PL/SQL is such that the GOTO statement is seldom needed. Occasionally, it can simplify logic enough to warrant its use. The NULL statement can improve readability by making the meaning and action of conditional statements clear.

### GOTO Statement

The GOTO statement branches to a label unconditionally. The label must be unique within its scope and must precede an executable statement or a PL/SQL block. When executed, the GOTO statement transfers control to the labeled statement or block. In the following example, you go to an executable statement farther down in a sequence of statements:

```
BEGIN
    ...
    GOTO insert_row;
    ...
    <<insert_row>>
    INSERT INTO emp VALUES ...
END;
```

In the next example, you go to a PL/SQL block farther up in a sequence of statements:

```
BEGIN
...
    <<update_row>>
    BEGIN
        UPDATE emp SET ...
    END;
...
GOTO update_row;
...
END;
```

The label end\_loop in the following example is not allowed because it does not precede an executable statement:

```
DECLARE
done BOOLEAN;
BEGIN
    ...
    FOR i IN 1..50
    LOOP
        IF done THEN
            GOTO end_loop;
        END IF;
        ...
        <<end_loop>> -- not allowed
    END LOOP; -- not an executable statement
END;
```

To debug the last example, just add the NULL statement, as follows:

```
FOR i IN 1..50 LOOP
IF done THEN
GOTO end_loop;
END IF;
...
<<end_loop>>
NULL; -- an executable statement
END LOOP;
```

## NULL Statement

The NULL statement does nothing other than pass control to the next statement. In a conditional construct, the NULL statement tells readers that a possibility has been considered, but no action is necessary. In the following example, the NULL statement shows that no action is taken for unnamed exceptions:

```
EXCEPTION
  WHEN ZERO_DIVIDE THEN
    ROLLBACK;
  WHEN VALUE_ERROR THEN
    INSERT INTO errors VALUES ...
    COMMIT;
  WHEN OTHERS THEN
    NULL;
END;
```

In IF statements or other places that require at least one executable statement, the NULL statement to satisfy the syntax. In the following example, the NULL statement emphasizes that only top-rated employees get bonuses:

```
IF rating > 90 THEN
  compute_bonus(emp_id);
ELSE
  NULL;
END IF;
```

## UNDERSTANDING SQL\*PLUS

### Overview of SQL \* Plus

You can use the SQL\*Plus program in conjunction with the SQL database language and its procedural language extension, PL/SQL. The SQL database language allows you to store and retrieve data in Oracle. PL/SQL allows you to link several SQL commands through procedural logic. SQL\*Plus enables you to execute SQL commands and PL/SQL blocks, and to perform many additional tasks as well. Through SQL\*Plus, you can

- ▣ enter, edit, store, retrieve, and run SQL commands and PL/SQL blocks
- ▣ format, perform calculations on, store, print and create web output of query results
- ▣ list column definitions for any table
- ▣ access and copy data between SQL databases
- ▣ send messages to and accept responses from an end user
- ▣ perform database administration

### What You Need to Run SQL\*Plus

Oracle and SQL\*Plus can run on many different kinds of computers. Before you can begin using SQL\*Plus, both Oracle and SQL\*Plus must be installed on your computer. When you start SQL\*Plus, you will need a username that identifies you as an authorized Oracle user and a password that proves you are the legitimate owner of your username.

### Learning SQL\*Plus Basics

#### Starting SQL\*Plus

1. Enter the command SQLPLUS/SQLPLUSW (if you are in Windows) and press Return. This is an operating system command that starts SQL\*Plus. Some operating systems expect you to enter commands in lowercase letters. If your system expects lowercase, enter the SQLPLUS command in lowercase.

SQLPLUS

SQL\*Plus displays its version number, the current date, and copyright information, and prompts you for your username (the text displayed on your system may differ slightly):

SQL\*Plus: Release 9.0.1.0.0 Production on Thu June 14 16:29:01 2001  
(c) Copyright 1996, 2001 Oracle Corporation. All rights reserved.

Enter username:

2. Enter your username and press Return. SQL\*Plus displays the prompt "Enter password:".
3. Enter your password and press Return again. For your protection, your password does not appear on the screen.
4. The process of entering your username and password is called logging in. SQL\*Plus displays the version of Oracle to which you connected and the versions of available tools such as PL/SQL. Next, SQL\*Plus displays the SQL\*Plus command prompt:  
SQL>  
The command prompt indicates that SQL\*Plus is ready to accept your commands.

## Leaving SQL\*Plus

When you are done working with SQL\*Plus and wish to return to the operating system, enter the EXIT command at the SQL\*Plus command prompt.

## Entering and Executing Commands

You can enter three kinds of commands at the command prompt:

- ▣ SQL commands, for working with information in the database
- ▣ PL/SQL blocks, also for working with information in the database
- ▣ SQL\*Plus commands, for formatting query results, setting options, and editing and storing SQL commands and PL/SQL blocks

## The SQL Buffer

The area where SQL\*Plus stores your most recently entered SQL command or PL/SQL block is called the SQL buffer. The command or block remains there until you enter another. If you want to edit or re-run the current SQL command or PL/SQL block, you may do so without re-entering it.

## Running SQL Commands

The SQL command language enables you to manipulate data in the database. In this example, you will enter and execute a SQL command to display the employee number, name, job, and salary of each employee in the EMP\_DETAILS\_VIEW view.

1. At the command prompt, enter the first line of the command:

```
SELECT EMPLOYEE_ID, LAST_NAME, JOB_ID, SALARY
```

If you make a mistake, use Backspace to erase it and re-enter. When you are done, press Return to move to the next line.

2. SQL\*Plus will display a “2”, the prompt for the second line. Enter the second line of the command:

```
FROM EMP_DETAILS_VIEW WHERE SALARY > 12000;
```

The semicolon (;) means that this is the end of the command. Press Return. SQL\*Plus processes the command and displays the results on the screen:

EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
100	King	AD_PRES	\$24,000
101	Kochhar	AD_VP	\$17,000



## Running PL/SQL Blocks

You can also use PL/SQL subprograms (called blocks) to manipulate data in the database. To enter a PL/SQL subprogram in SQL\*Plus, you need to be in PL/SQL mode. You are placed in PL/SQL mode when

- ▣ You type DECLARE or BEGIN at the SQL\*Plus command prompt. After you enter PL/SQL mode in this way, type the remainder of your PL/SQL subprogram.
- ▣ You type a SQL command (such as CREATE FUNCTION) that creates a stored procedure. After you enter PL/SQL mode in this way, type the stored procedure you want to create. Execute the current subprogram by issuing a RUN or slash (/) command. Likewise, to execute a SQL CREATE command that creates a stored procedure, you must also enter RUN or slash (/). A semicolon (;) will not execute these SQL commands as it does other SQL commands.

## Running SQL\*Plus Commands

You can use SQL\*Plus commands to manipulate SQL commands and PL/SQL blocks and to format and print query results. SQL\*Plus treats SQL\*Plus commands differently than SQL commands or PL/SQL blocks. To speed up command entry, you can abbreviate many SQL\*Plus commands to one or a few letters. For e.g. to view the structure of a table, you can enter the following SQL\*Plus Command

```
SQL> DESC EMP
```

It will produce the following output

Name	Null?	Type
-----		
EMPNO	NOT NULL	NUMBER
ENAME		VARCHAR2 (10)
JOB		VARCHAR2 (9)
MGR		NUMBER (4)
HIREDATE		DATE
SAL		NUMBER (7, 2)
COMM		NUMBER (7, 2)
DEPTNO		NUMBER (2)

## Stopping a Command while it is Running

Suppose you have displayed the first page of a 50 page report and decide you do not need to see the rest of it. Press Cancel, the system's interrupt character, which is usually CTRL+C. SQL\*Plus stops the display and returns to the command prompt.

## Manipulating Commands

### Saving Commands for Later Use

To save the current SQL command or PL/SQL block for later use, enter the SAVE command. Follow the command with a file name:

```
SQL>SAVE file_name
```

SQL\*Plus adds the .SQL extension to the filename to identify it as a SQL query file. If you wish to save the command or block under a name with a different file extension, type a period at the end of the filename, followed by the extension.

### Retrieving Command Files

If you want to place the contents of a command file in the buffer, you must retrieve the command from the file in which it is stored. You can retrieve a command file using the SQL\*Plus command GET. Just as you can save a query from the buffer to a file with the SAVE command, you can retrieve a query from a file to the buffer with the GET command:

```
SQL>GET file_name
```

When appropriate to the operating system, SQL\*Plus adds a period and the extension SQL to the filename unless you type a period at the end of the filename followed by a different extension.

### Running Command Files

The START command retrieves a command file and runs the command(s) it contains. Use START to run a command file containing SQL commands, PL/SQL blocks, and SQL\*Plus commands. You can have many commands in the file. Follow the START command with the name of the file:

```
SQL>START file_name
```

You can also use the @ ("at" sign) command to run a command file:

```
@SALES
```

## Setting Up Your SQL\*Plus Environment

You may wish to set up your SQL\*Plus environment in a particular way (such as showing the current time as part of the SQL\*Plus command prompt) and then reuse those settings with each session. You can do this through a host operating system file called LOGIN with the file extension SQL (also called your User Profile). The exact name of this file is system dependent; see the Oracle installation and user's guide provided for your operating system for the precise name.

You can add any SQL commands, PL/SQL blocks, or SQL\*Plus commands to this file; when you start SQL\*Plus, it automatically searches for your LOGIN file (first in your local directory and then on a system-dependent path) and runs the commands it finds there.

You may wish to add some of the following commands to the LOGIN file:

**SET LINESIZE** Followed by a number, sets the number of characters as page width of the query results.

**SET NUMFORMAT** Followed by a number format (such as \$99,999), sets the default format for displaying numbers in query results.

**SET PAGESIZE** Followed by a number, sets the number of lines per page.

**SET PAUSE** Followed by ON, causes SQL\*Plus to pause at the beginning of each page of output ( SQL\*Plus continues scrolling after you enter Return). Followed by text, sets the text to be displayed each time SQL\*Plus pauses (you must also set PAUSE to ON).

**SET TIME** Followed by ON, displays the current time before each command prompt.

### Writing Interactive Commands

The following features of SQL\*Plus make it possible for you to set up command files that allow end user input:

- ▣ defining user variables
- ▣ substituting values in commands
- ▣ prompting for values

### Defining User Variables

You can define variables, called user variables, for repeated use in a single command file by using the SQL\*Plus **DEFINE** command.

To define a user variable **L\_NAME** and give it the value "SMITH", enter the following command:

```
DEFINE L_NAME = SMITH
```

To confirm the variable definition, enter **DEFINE** followed by the variable name:

```
DEFINE L_NAME
```

To delete a user variable, use the SQL\*Plus command **UNDEFINE** followed by the variable name.

## Using Substitution Variables

Suppose you want to write a query like the one in SALES (see Example 3–7) to list the employees with various jobs, not just those whose job is SA\_MAN. You could do that by editing a different CHAR value into the WHERE clause each time you run the command, but there is an easier way. By using a substitution variable in place of the value SA\_MAN in the WHERE clause, you can get the same results you would get if you had written the values into the command itself.

A substitution variable is a user variable name preceded by one or two ampersands (&). When SQL\*Plus encounters a substitution variable in a command, SQL\*Plus executes the command as though it contained the value of the substitution variable, rather than the variable itself. For example,

```
SQL> SELECT EMPNO, ENAME, JOB
FROM EMP1
WHERE DEPTNO = &DEPTNO;
```

Enter value for deptno: 30

```
old 3: WHERE DEPTNO = &DEPTNO
new 3: WHERE DEPTNO = 30
```

```
EMPNO  ENAME  JOB
-----  -
7499  ALLEN  SALESMAN
7521  WARD   SALESMAN
7654  MARTIN SALESMAN
7698  BLAKE  MANAGER
7844  TURNER SALESMAN
7900  JAMES  CLERK
8003  SOUMEN MANAGER
7 rows selected.
SQL>
```

## Communicating with the User

Three SQL\*Plus commands—PROMPT, ACCEPT, and PAUSE—help you communicate with the end user. These commands enable you to send messages to the screen and receive input from the user, including a simple Return. You can also use PROMPT and ACCEPT to customize the prompts for values SQL\*Plus automatically generates for substitution variables.

```
ACCEPT MYTITLE PROMPT 'Title: '
```

## Using Bind Variables

Bind variables are variables you create in SQL\*Plus and then reference in PL/SQL or SQL. If you create a bind variable in SQL\*Plus, you can use the variable as you would a declared variable in your PL/SQL subprogram and then access the variable from SQL\*Plus. You can use bind variables for such things as storing return codes or debugging your PL/SQL subprograms.

Because bind variables are recognized by SQL\*Plus, you can display their values in SQL\*Plus or reference them in PL/SQL subprograms that you run in SQL\*Plus. You create bind variables in SQL\*Plus with the VARIABLE command. For example

```
VARIABLE ret_val NUMBER
```

This command creates a bind variable named `ret_val` with a datatype of `NUMBER`. You reference bind variables in PL/SQL by typing a colon (`:`) followed immediately by the name of the variable. For example

```
:ret_val := 1;
```

To display the value of a bind variable in SQL\*Plus, you use the SQL\*Plus `PRINT` command. For example:

```
PRINT RET_VAL
```

```
RET_VAL
```

```
-----
```

```
4
```

This command displays a bind variable named `ret_val`.

## ABOUT CURSORS

Cursor is a mechanism by which you can assign a name to a "select statement" and manipulate the information within that SQL statement. PL/SQL uses two types of cursors: implicit and explicit. PL/SQL declares a cursor implicitly for all SQL data manipulation statements, including queries that return only one row. However, for queries that return more than one row, you must declare an explicit cursor, use a cursor FOR loop, or use the BULK COLLECT clause.

While dealing with cursors, you may need to determine the status of your cursor. The following is a list of the cursor attributes that you can use.

### Attribute Explanation

%ISOPEN - Returns TRUE if the cursor is open, FALSE if the cursor is closed.

%FOUND - Returns INVALID\_CURSOR if cursor is declared, but not open; or if cursor has been closed.

- Returns NULL if cursor is open, but fetch has not been executed.
- Returns TRUE if a successful fetch has been executed.
- Returns FALSE if no row was returned.

%NOTFOUND - Returns INVALID\_CURSOR if cursor is declared, but not open; or if cursor has been closed.

- Return NULL if cursor is open, but fetch has not been executed.
- Returns FALSE if a successful fetch has been executed.
- Returns TRUE if no row was returned.

%ROWCOUNT - Returns INVALID\_CURSOR if cursor is declared, but not open; or if cursor has been closed.

- Returns the number of rows fetched.

## Overview of Implicit Cursors

Implicit Cursors get created automatically every time you use an INSERT, UPDATE, DELETE or SELECT command. It doesn't need to be declared. Normally we use to assign the output of a SELECT command to one or more PL/SQL variables. It is used if query returns one and only one record.

For e.g the following syntax in a PL/SQL program will create an implicit cursor

```
SELECT field1, field2, ...  
INTO variable1, variable2, ...  
FROM tablename  
WHERE search_condition_that_will_return_a_single_record;
```

### Overview of Implicit Cursor Attributes

Implicit cursor attributes return information about the execution of an INSERT, UPDATE, DELETE, or SELECT INTO statement. The values of the cursor attributes always refer to the most recently executed SQL statement. Before Oracle opens the SQL cursor, the implicit cursor attributes yield NULL.

**%FOUND Attribute: Has a DML Statement Changed Rows?**

Until a SQL data manipulation statement is executed, %FOUND yields NULL. Thereafter, %FOUND yields TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows, or a SELECT INTO statement returned one or more rows. Otherwise, %FOUND yields FALSE. In the following example, you use %FOUND to insert a row if a delete succeeds:

```
DELETE FROM emp WHERE empno = my_empno;  
IF SQL%FOUND THEN delete
```

Succeeded

```
INSERT INTO new_emp VALUES (my_empno, my_ename, ...);
```

**%ISOPEN Attribute: Always FALSE for Implicit Cursors**

Oracle closes the SQL cursor automatically after executing its associated SQL statement. As a result, %ISOPEN always yields FALSE.

**%NOTFOUND Attribute: Has a DML Statement Failed to Change Rows?**

%NOTFOUND is the logical opposite of %FOUND. %NOTFOUND yields TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, %NOTFOUND yields FALSE.

**%ROWCOUNT Attribute: How Many Rows Affected So Far?**

%ROWCOUNT yields the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement. %ROWCOUNT yields 0 if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. In the following example, you use %ROWCOUNT to take action if more than ten rows have been deleted:

```
DELETE FROM emp WHERE ...  
IF SQL%ROWCOUNT > 10 THEN more  
than 10 rows were deleted  
...  
END IF;
```

If a SELECT INTO statement returns more than one row, PL/SQL raises the predefined exception TOO\_MANY\_ROWS and %ROWCOUNT yields 1, not the actual number of rows that satisfy the query.

## Overview of Explicit Cursors

The set of rows returned by a query can consist of zero, one, or multiple rows, depending on how many rows meet your search criteria. When a query returns multiple rows, you can explicitly declare a cursor to process the rows. Moreover, you can declare a cursor in the declarative part of any PL/SQL block, subprogram, or package.

You use three commands to control a cursor: OPEN, FETCH, and CLOSE. First, you initialize the cursor with the OPEN statement, which identifies the result set. Then, you can execute FETCH repeatedly until all rows have been retrieved, or you can use the BULK COLLECT clause to fetch all rows at once. When the last row has been processed, you release the cursor with the CLOSE statement.

## Declaring a Cursor

Forward references are not allowed in PL/SQL. So, you must declare a cursor before referencing it in other statements. When you declare a cursor, you name it and associate it with a specific query using the syntax

```
CURSOR cursor_name [(parameter[, parameter]...)]  
[RETURN return_type] IS select_statement;
```

where return\_type must represent a record or a row in a database table, and parameter stands for the following syntax:

```
cursor_parameter_name [IN] datatype [{:= | DEFAULT} expression]
```

For example, you might declare cursors named c1 and c2, as follows:

```
DECLARE  
CURSOR c1 IS SELECT empno, ename, job, sal FROM emp  
WHERE sal > 2000;  
CURSOR c2 RETURN dept%ROWTYPE IS  
SELECT * FROM dept WHERE deptno = 10;
```

## Opening a Cursor

Opening the cursor executes the query and identifies the result set, which consists of all rows that meet the query search criteria. For cursors declared using the FOR UPDATE clause, the OPEN statement also locks those rows. An example of the OPEN statement follows:

```
DECLARE  
CURSOR c1 IS SELECT ename, job FROM emp WHERE sal < 3000;  
...  
BEGIN  
OPEN c1;  
...  
END;
```

Rows in the result set are not retrieved when the OPEN statement is executed. Rather, the FETCH statement retrieves the rows.

## Fetching with a Cursor

The FETCH statement retrieves the rows in the result set one at a time. Each fetch retrieves the current row and then advances the cursor to the next row in the result set. An example follows:

```
FETCH c1 INTO my_empno, my_ename, my_deptno;
```

For each column value returned by the query associated with the cursor, there must be a corresponding, type-compatible variable in the INTO list. Typically, you use the FETCH statement in the following way:



```
LOOP
FETCH c1 INTO my_record;
EXIT WHEN c1%NOTFOUND;
process
data record
END LOOP;
```

## Closing a Cursor

The CLOSE statement disables the cursor, and the result set becomes undefined. Once a cursor is closed, you can reopen it. Any other operation on a closed cursor raises the predefined exception INVALID\_CURSOR.

## Using Cursor FOR Loops

In most situations that require an explicit cursor, you can simplify coding by using a cursor FOR loop instead of the OPEN, FETCH, and CLOSE statements. A cursor FOR loop implicitly declares its loop index as a %ROWTYPE record, opens a cursor, repeatedly fetches rows of values from the result set into fields in the record, and closes the cursor when all rows have been processed.

Consider the PL/SQL block below, which computes results from an experiment, then stores the results in a temporary table. The FOR loop index c1\_rec is implicitly declared as a record. Its fields store all the column values fetched from the cursor c1. Dot notation is used to reference individual fields.

```
DECLARE
result temp.col1%TYPE;
CURSOR c1 IS
SELECT n1, n2, n3 FROM data_table WHERE exper_num = 1;
BEGIN
FOR c1_rec IN c1 LOOP
/* calculate and store the results */
result := c1_rec.n2 / (c1_rec.n1 + c1_rec.n3);
INSERT INTO temp VALUES (result, NULL, NULL);
END LOOP;
COMMIT;
END;
```

When the cursor FOR loop is entered, the cursor name cannot belong to a cursor already opened by an OPEN statement or enclosing cursor FOR loop. Before each iteration of the FOR loop, PL/SQL fetches into the implicitly declared record. The record is defined only inside the loop. You cannot refer to its fields outside the loop. The sequence of statements inside the loop is executed once for each row that satisfies the query associated with the cursor. When you leave the loop, the cursor is closed automatically—even if you use an EXIT or GOTO statement to leave the loop prematurely or an exception is raised inside the loop.

## Using Subqueries Instead of Explicit Cursors

You need not declare a cursor because PL/SQL lets you substitute a subquery. The following cursor FOR loop calculates a bonus, then inserts the result into a database table:

```

DECLARE
bonus REAL;
BEGIN
    FOR emp_rec IN (SELECT empno, sal, comm FROM emp) LOOP
        bonus := (emp_rec.sal * 0.05) + (emp_rec.comm * 0.25);
        INSERT INTO bonuses VALUES (emp_rec.empno, bonus);
    END LOOP;
    COMMIT;
END;

```

## Parameterized Cursors

An explicit cursor can take parameters and return a data set for a specific parameter value. This eliminates the need to define multiple cursors and hard-code a value in each cursor. In the following code, the cursor example presented to illustrate parameterized cursors:

```

DECLARE

CURSOR csr_org(p_hrc_code NUMBER) IS
    SELECT h.hrc_descr, o.org_short_name
    FROM org_tab o, hrc_tab h
    WHERE o.hrc_code = h.hrc_code
    AND h.hrc_code = p_hrc_code
    ORDER by 2;

v_org_rec csr_org%ROWTYPE;

BEGIN
    OPEN csr_org(1);
    dbms_output.put_line('Organization Details with Hierarchy 1');
    dbms_output.put_line("");
    dbms_output.put_line(rpad('Hierarchy',20,' ')||' '|| rpad('Organization',30,' '));
    dbms_output.put_line(rpad(",20,")||"||rpad(",30,");
    LOOP
        FETCH csr_org INTO v_org_rec;
        EXIT WHEN csr_org%NOTFOUND;
        dbms_output.put_line(rpad(v_org_rec.hrc_descr,20,' ')||' '||
            rpad(v_org_rec.org_short_name,30,' '));
    END LOOP;
    CLOSE csr_org;

    OPEN csr_org(2);

    dbms_output.put_line('Organization Details with Hierarchy 2');
    dbms_output.put_line("");
    dbms_output.put_line(rpad('Hierarchy',20,' ')||' '||rpad('Organization',30,"));
    dbms_output.put_line(rpad(",20,")||"||rpad(",30,");
    LOOP
        FETCH csr_org INTO v_org_rec;
        EXIT WHEN csr_org%NOTFOUND;
        dbms_output.put_line(rpad(v_org_rec.hrc_descr,20,' ')||' '||
            rpad(v_org_rec.org_short_name,30,' '));
    END LOOP;

```

```
CLOSE csr_org;
END;
```

Here's the output of this program:

```
Organization Details with Hierarchy 1
-----
Hierarchy Organization
-----
CEO/COO      Office of CEO ABC Inc.
CEO/COO      Office of CEO DataPro Inc.
CEO/COO      Office of CEO XYZ Inc.

Organization Details with Hierarchy 2
-----
Hierarchy Organization
-----
VP           Office of VP Mktg ABC Inc.
VP           Office of VP Sales ABC Inc.
VP           Office of VP Tech ABC Inc.

PL/SQL procedure successfully completed.
```

You define the cursor parameters immediately after the cursor name by including the name of the parameter and its data type within parentheses. These are referred to as the formal parameters. The actual parameters (i.e., the actual data values for the formal parameters) are passed via the OPEN statement as shown in the previous example. Notice how the same cursor is used twice with different values of the parameters in each case. You can rewrite the same example using a cursor FOR LOOP. In this case, the actual parameters are passed via the cursor name referenced in the cursor FOR LOOP. Here's the code:

```
DECLARE
CURSOR csr_org(p_hrc_code NUMBER) IS
    SELECT h.hrc_descr, o.org_short_name
    FROM org_tab o, hrc_tab h
    WHERE o.hrc_code = h.hrc_code
    AND h.hrc_code = p_hrc_code
    ORDER by 2;
v_org_rec csr_org%ROWTYPE;
BEGIN
dbms_output.put_line('Organization Details with Hierarchy 1');
dbms_output.put_line("");
dbms_output.put_line(rpad('Hierarchy',20,' ')||' '||rpad('Organization',30,''));
dbms_output.put_line(rpad("",20,"")||' '||rpad("",30,""));
FOR idx in csr_org(1) LOOP
    dbms_output.put_line(rpad(idx.hrc_descr,20,' ')||' '||rpad(idx.org_short_name,30,' '));
END LOOP;
dbms_output.put_line('Organization Details with Hierarchy 2');
dbms_output.put_line("");
dbms_output.put_line(rpad('Hierarchy',20,' ')||' '||
rpad('Organization',30,''));dbms_output.put_line(rpad("",20,"")||' '||rpad("",30,""));

FOR idx in csr_org(2)
LOOP
    dbms_output.put_line(rpad(idx.hrc_descr,20,' ')||' '||rpad(idx.org_short_name,30,' '));
END LOOP;
```

END;

The output of this program is the same as the output of the earlier one. Parameterized cursors are very useful in processing nested cursor loops in which an inner cursor is opened with data values passed to it from an outer opened cursor

## SELECT FOR UPDATE Cursors

You use SELECT FOR UPDATE cursors for updating the rows retrieved by a cursor. This is often required when there's a need to modify each row retrieved by a cursor without having to re-fetch that row. More often, SELECT FOR UPDATE cursors are required to update a column of the table defined in the cursor SELECT using a complex formula.

### Defining a SELECT FOR UPDATE Cursor

A SELECT FOR UPDATE cursor is defined using the FOR UPDATE OF clause in the cursor SELECT statement, as follows:

```
DECLARE
CURSOR csr_1 IS
SELECT * FROM sec_hrc_tab FOR UPDATE OF hrc_descr;
BEGIN
/* . . . Open the cursor and process the resultset . . . */
END;
```

**NOTE** Notice how the column name to be updated is specified in the FOR UPDATE OF clause. If no column name is specified in the FOR UPDATE OF clause, any column of the underlying cursor table can be modified.

### Using a SELECT FOR UPDATE Cursor

Once you've defined a SELECT FOR UPDATE cursor, you use the WHERE CURRENT OF clause to process the rows returned by it. You can use this clause in an UPDATE or DELETE statement. It has the following syntax:

*WHERE CURRENT of cursor\_name;*

where cursor\_name is the name of the cursor defined with a FOR UPDATE clause. The following is a complete example of using SELECT FOR UPDATE cursors.

```
DECLARE
CURSOR csr_1 IS
SELECT * FROM sec_hrc_tab FOR UPDATE OF hrc_descr;
v_hrc_descr VARCHAR2(20);

BEGIN
FOR idx IN csr_1
LOOP
    v_hrc_descr := UPPER(idx.hrc_descr);
    UPDATE sec_hrc_tab
    SET hrc_descr = v_hrc_descr
    WHERE CURRENT OF csr_1;
END LOOP;
COMMIT;
END;
/
```

This program updates the `hrc_descr` column of each row retrieved by `csr_1` with its value converted to uppercase. The mechanism of `SELECT FOR UPDATE` cursors works as follows:

1. The `SELECT FOR UPDATE` cursor puts a lock on the rows retrieved by the cursor. If it's unable to obtain a lock because some other session has placed a lock on the specific rows, it waits until it can get a lock. A `COMMIT` or `ROLLBACK` in the corresponding session frees the locks held by other sessions.
2. For each row identified by the cursor, the cursor updates the specified column of that row. That is, it keeps track of the current row and updates it, and then fetches the subsequent row and updates it. It does this without scanning the same table again. This is unlike an ordinary `UPDATE` or `DELETE` statement inside the loop, where the cursor scans the updated table again to determine the current row to be modified.

## Cursor Variables

An explicit cursor once declared was associated with a specific query—only the one specific query that was known at compile time. In this way, the cursor declared was static and couldn't be changed at runtime. It always pointed to the same work area until the execution of the program completed. However, you may sometimes want to have a variable that can point to different work areas depending on runtime conditions. PL/SQL 2.2 onward offers this facility by means of cursor variables.

A cursor variable is a single PL/SQL variable that you can associate with different queries at runtime. The same variable can point to different work areas. In this way, cursor variables and cursors are analogous to PL/SQL variables and constants, but from a cursor perspective. A cursor variable acts like a pointer that holds the address of a specific work area defined by the query it's pointing to.

### Why Use Cursor Variables?

The primary advantage of using cursor variables is their capability to pass resultsets between stored subprograms. Before cursor variables, this wasn't possible. Now, with cursor variables, the work area that a cursor variable points to remains accessible as long as the variable points to it. Hence, you can point a cursor variable to a work area by opening a cursor for it, and then any application such as Pro\*C, an Oracle client, or another server application can fetch from the corresponding resultset.

Another advantage of cursor variables is their introduction of a sort of dynamism, in that a single cursor variable can be associated with multiple queries at runtime.

### **Defining a Cursor Variable**

Defining a cursor variable consists of defining a pointer of type REF CURSOR and defining a variable of this type. These steps are outlined in the following sections.

### **Defining a Pointer of Type CURSOR**

In PL/SQL, a pointer is declared using the syntax REF type. The keyword REF implies that the new type so defined is a pointer to the defined type. PL/SQL offers two types of REF types: CURSOR and an object type. So, the definition of a cursor variable involves the definition of a REF CURSOR first, as shown here:

```
TYPE rc IS REF CURSOR;
```

### **Defining a Variable of Type REF CURSOR**

Once you've defined a REF CURSOR type, the next step is to declare a variable of this type. Here's the code for this:

```
v_rc rc;
```

So the complete declaration of a cursor variable is as follows:

```
TYPE rc IS REF CURSOR;  
v_rc rc;
```

This code suggests that rc is a pointer of type CURSOR and v\_rc (in fact, any variable) defined of type rc points to a SQL cursor.

### **Strong and Weak REF CURSOR Types**

The REF CURSOR type defined earlier is called a weak REF CURSOR type. This is because it doesn't dictate the return type of the cursor. Hence, it can point to any SELECT query with any number of columns. Weak cursor types are available in PL/SQL 2.3 and higher versions.

PL/SQL lets you define a strong REF CURSOR having a return type using the following syntax:

```
TYPE ref_type_name IS REF CURSOR RETURN return_type;
```

Here, `ref_type_name` is the name of the new pointer name and `return_type` is a record type of either `%ROWTYPE` or a user-defined record type. For example, you can declare strong REF CURSORS as follows:

```
TYPE rc IS REF CURSOR RETURN hrc_tab%ROWTYPE;
v_rc rc;

or

TYPE hrc_rec IS RECORD (hrc_code NUMBER, hrc_name VARCHAR2(20)); TYPE
rc IS REF CURSOR RETURN hrc_rec;
```

In the case of a strong REF CURSOR, the query that's associated with it should be type-compatible one to one with the return type of the corresponding REF CURSOR.

## Using a Cursor Variable

Once you've defined a cursor variable, you can use it to associate it with a query. Here are the steps:

1. Allocate memory.
2. Open the cursor variable for a query.
3. Fetch the results into a PL/SQL record or individual PL/SQL variables.
4. Close the cursor variable.

The following sections provide more detail about each step in the process.

### Allocate Memory

Once you declare a cursor variable in PL/SQL, the PL/SQL engine in PL/SQL 2.3 and higher versions automatically allocates memory for storage of rows. Prior to PL/SQL 2.3, a host environment was needed to explicitly allocate memory to a cursor variable.

### Opening the Cursor Variable

Once you've defined a cursor variable, you have to open it for a multirow query, either with an arbitrary number of columns in the case of a weak REF CURSOR or with a type-compatible query in the case of a strong REF CURSOR. Opening the cursor variable identifies the associated query, executes it, and also identifies the resultset.

You open a cursor variable using the OPEN-FOR statement. Here's the syntax: OPEN

```
{cursor_variable_name | :host_cursor_variable_name} FOR
{ select_query
| dynamic_string [USING bind_variable[, bind_variable] . . . ] };
```

where `cursor_variable_name` is the name of the declared cursor variable and `select_query` is the SELECT query associated with the cursor variable. Also, `host_cursor_variable_name` is the name of the cursor variable declared in a PL/SQL host environment (such as Pro\*C), and `bind_variable` represents the name of a PL/SQL



bind variable. `dynamic_string` represents a dynamic SQL string instead of a hard-coded SELECT statement. You open cursor variables for dynamic strings using native dynamic SQL.

Here's an example that illustrates opening the cursor variable for the previously declared weak cursor variable `v_rc`:

```
DECLARE
TYPE rc IS REF CURSOR;
v_rc rc; BEGIN
OPEN v_rc FOR SELECT * FROM hrc_tab;
/* . . . FETCH the results and process the resultset */
END;
```

### Fetching the Results into a PL/SQL Record or Individual PL/SQL Variables

The next step is to fetch the cursor variable into a PL/SQL record or individual variables. This retrieves individual rows of data into the PL/SQL variables for processing. You fetch a cursor variable using the `FETCH` statement, which has three forms. Here's the syntax:

```
FETCH cursor_variable_name INTO var1, var2, . . . , varN;
or
FETCH cursor_variable_name INTO table_name%ROWTYPE;
or
FETCH cursor__variable_name INTO record_name;
```

Here, `var1`, `var2`, and `varN` represent PL/SQL variables having data types identical to the cursor variable query. `table_name%ROWTYPE` represents a PL/SQL record type with attributes implicitly defined as the column names of the table identified by `table_name`, which are identical to the cursor variable `SELECT`. In this case, you need to explicitly define the record type. Lastly, `record_name` is a variable of a PL/SQL record type that's explicitly defined. In this case also, the number and data types of the individual attributes of the record should exactly match the columns in the cursor variable `SELECT`.

Here's an example that extends the previous example of `v_rc` to fetching rows:

```
DECLARE
TYPE rc IS REF CURSOR;
v_rc rc;
hrc_rec hrc_tab%ROWTYPE;
BEGIN
OPEN v_rc FOR SELECT * FROM hrc_tab; LOOP
END;
/
FETCH v_rc INTO hrc_rec; EXIT WHEN v_rc%NOTFOUND;
```

```
/* . . . Process the individual records */ END LOOP;
```

The number and data types of the individual variables should exactly match the columns list in the cursor variable's associated SELECT statement. If the cursor is fetched into a record type (either table\_name%ROWTYPE or record\_name), the number and data type of each attribute in the record should exactly match the columns list of the cursor variable associated SELECT statement. If this isn't the case, then PL/SQL raises an error at compile time if the cursor variable is strongly typed, and a predefined exception called ROWTYPE\_MISMATCH at runtime if the cursor variable is weakly typed.

### Closing the Cursor Variable

Once the processing of the rows is completed, you can close the cursor variable. Closing the cursor variable frees the resources allocated to the query but doesn't necessarily free the storage of the cursor variable itself. The cursor variable is freed when the variable is out of scope. You close a cursor using the CLOSE statement. Here's the syntax:

```
CLOSE cursor_variable_name;
```

Here's a complete example of using the v\_rc cursor, involving all the steps previously covered:

```
DECLARE
TYPE rc is REF CURSOR;
v_rc rc;
hrc_rec hrc_tab%ROWTYPE;
BEGIN
    OPEN v_rc FOR SELECT * from hrc_tab;
    dbms_output.put_line('Hierarchy Details');
    dbms_output.put_line("");
    dbms_output.put_line('Code||' '||rpad('Description',20,''));
    dbms_output.put_line(rpad("",4,"")||"||rpad("",20,""));
    LOOP
        FETCH v_rc INTO hrc_rec;
        EXIT WHEN v_rc%NOTFOUND;
        dbms_output.put_line(to_char(hrc_rec.hrc_code)||"||rpad(hrc_rec.hrc_descr,20,''));
    END LOOP;
    CLOSE v_rc;
END;
/
```

Here's the output of this program:

```
Hierarchy Details
----- Code
Description
-----
1    CEO/COO
2    VP
3    Director
4    Manager
5    Analyst
```

PL/SQL procedure successfully completed.

This code is similar to the code used for static cursors, except that it uses cursor variables instead of cursors.

## Cursor Variables Assignment

One way to make a cursor variable point to a query work area is to open a query for the cursor variable. You saw this earlier. Here, I describe a second way to make a cursor variable point to a query work area. Simply assign the cursor variable to an already OPENed cursor variable. Here's an example of cursor variable assignment:

```
DECLARE
TYPE rc IS REF CURSOR;
v_rc1 rc;
v_rc2 rc;
hrc_rec hrc_tab%ROWTYPE;
BEGIN
    OPEN v_rc1 FOR SELECT * FROM hrc_tab;
    dbms_output.put_line('Hierarchy
Details');
    dbms_output.put_line("");
    dbms_output.put_line('Code'||
'||rpad('Description',20,""));
    dbms_output.put_line(rpad("",4,"")||"||rpa
d("",20,""));
    /* Assign v_rc1 to v_rc2 */
    v_rc2 := v_rc1;
    LOOP
        /* Fetch from the second cursor
variable, i.e., v_rc2 */
        FETCH v_rc2 INTO hrc_rec;EXIT
        WHEN v_rc2%NOTFOUND;

        dbms_output.put_line(to_char(hrc_rec.hrc
_code)||"||rpad(hrc_rec.hrc_descr,20,""));
    END LOOP;
    CLOSE v_rc2;
```

```
END;
```

```
/
```

The output of this program is the same as the output of the earlier example without the assignment. Note that closing `v_rc2` also closes `v_rc1` and vice versa. However, if the source cursor variable is strongly typed, the target cursor variable must be of the same type as the source cursor variable. This restriction doesn't apply if the source cursor variable is weakly typed. Here's an example that illustrates this concept

```
DECLARE
```

```
    TYPE rc1 is REF CURSOR RETURN hrc_tab%ROWTYPE;
```

```
    TYPE rc2 is REF CURSOR RETURN hrc_tab%ROWTYPE;
```

```
    TYPE rc is REF CURSOR;
```

```
    v_rc1 rc1; v_rc2 rc2; v_rc3 rc;
```

```
    v_rc4 rc;
```

```
    hrc_rec hrc_tab%ROWTYPE;
```

```
BEGIN
```

```
    OPEN v_rc1 FOR SELECT * from hrc_tab;
```

```
    /* Assign v_rc1 to v_rc2 */
```

```
    v_rc2 := v_rc1; — This causes type error.
```

```
    v_rc3 := v_rc1; —This succeeds.
```

```
    v_rc4 := v_rc3; — This succeeds.
```

```
    /* . . . FETCH and process . . . */
```

```
END;
```

```
/
```

## Dynamism in Using Cursor Variables

The real use of cursor variables is when you have a need to open multiple queries using the same cursor variable or to dynamically assign different queries to the same cursor variable depending on runtime conditions. I discuss two examples in the following sections that illustrate the dynamism involved in using cursor variables.

To open multiple queries using the same cursor variable, use this code:

```
DECLARE
```

```
    TYPE rc is REF CURSOR;
```

```
    v_rc rc;
```

```
    hrc_rec hrc_tab%ROWTYPE;
```

```
    v_hrc_descr VARCHAR2(20);
```

```
    v_org_short_name VARCHAR2(30);
```

```
BEGIN
```

```
    OPEN v_rc FOR SELECT * from hrc_tab;
```

```

        dbms_output.put_line('Hierarchy Details');
        dbms_output.put_line("");
        dbms_output.put_line('Code||' '||rpad('Description',20,""));
        dbms_output.put_line(rpad(",4,")||"||rpad(",20,"));
LOOP
    FETCH v_rc INTO hrc_rec;
    EXIT WHEN v_rc%NOTFOUND;
    dbms_output.put_line(to_char(hrc_rec.hrc_code)||"||rpad(hrc_rec.hrc_descr,20,"));
END LOOP;

OPEN v_rc FOR SELECT h.hrc_descr, o.org_short_name
FROM org_tab o, hrc_tab h
WHERE o.hrc_code = h.hrc_code;

dbms_output.put_line('Hierarchy and Organization Details');
dbms_output.put_line("");
dbms_output.put_line(rpad('Hierarchy',20,' ')||' '||rpad('Description',30,""));
dbms_output.put_line(rpad(",20,")||"||rpad(",30,"));
LOOP
    FETCH v_rc INTO v_hrc_descr, v_org_short_name;
    EXIT WHEN v_rc%NOTFOUND;
    dbms_output.put_line(rpad(v_hrc_descr,20,' ')||"||rpad(v_org_short_name,30,"));
END LOOP;
CLOSE v_rc;
END;
```

Here's the output of this program:

Hierarchy Details    Code    Description

1		CEO/COO
2		VP
3		Director
4		Manager
5		Analyst

Hierarchy and Organization Details

Hierarchy	Description
CEO/COO	Office of CEO ABC Inc.
CEO/COO	Office of CEO XYZ Inc.
CEO/COO	Office of CEO DataPro Inc.
VP	Office of VP Sales ABC Inc.
VP	Office of VP Mktg ABC Inc.
VP	Office of VP Tech ABC Inc.

The following points are worth noting:

- The same cursor variable v\_rc is used to point to two different queries.
- After you open v\_rc for the first query and fetch the results, v\_rc isn't closed. It's simply reopened for a second query and a new resultset is identified.



## HANDLING PL/SQL ERRORS

Run-time errors arise from design faults, coding mistakes, hardware failures, and many other sources. Although you cannot anticipate all possible errors, you can plan to handle certain kinds of errors meaningful to your PL/SQL program.

With PL/SQL, a mechanism called exception handling lets you "bulletproof" your program so that it can continue operating in the presence of errors.

### Overview of PL/SQL Error Handling

In PL/SQL, a warning or error condition is called an exception. Exceptions can be internally defined (by the run-time system) or user defined. Examples of internally defined exceptions include division by zero and out of memory. Some common internal exceptions have predefined names, such as `ZERO_DIVIDE` and `STORAGE_ERROR`. The other internal exceptions can be given names.

You can define exceptions of your own in the declarative part of any PL/SQL block, subprogram, or package. For example, you might define an exception named `insufficient_funds` to flag overdrawn bank accounts. Unlike internal exceptions, user-defined exceptions must be given names.

When an error occurs, an exception is raised. That is, normal execution stops and control transfers to the exception-handling part of your PL/SQL block or subprogram. Internal exceptions are raised implicitly (automatically) by the run-time system. User-defined exceptions must be raised explicitly by `RAISE` statements, which can also raise predefined exceptions.

To handle raised exceptions, you write separate routines called exception handlers. After an exception handler runs, the current block stops executing and the enclosing block resumes with the next statement. If there is no enclosing block, control returns to the host environment.

In the example below, you calculate and store a price-to-earnings ratio for a company with ticker symbol XYZ. If the company has zero earnings, the predefined exception `ZERO_DIVIDE` is raised. This stops normal execution of the block and transfers control to the exception handlers. The optional `OTHERS` handler catches all exceptions that the block does not name specifically.

```
DECLARE
  pe_ratio NUMBER(3,1);
BEGIN
  SELECT price / earnings INTO pe_ratio FROM stocks
  WHERE symbol = 'XYZ'; --might cause divisionbyzero error

  INSERT INTO stats (symbol, ratio) VALUES ('XYZ', pe_ratio);
  COMMIT;

EXCEPTION -- exception handlers begin
WHEN ZERO_DIVIDE THEN -- handles 'division by zero' error
  INSERT INTO stats (symbol, ratio) VALUES ('XYZ', NULL);
  COMMIT;
...
WHEN OTHERS THEN -- handles all other errors
  ROLLBACK;
END; -- exception handlers and block end here
```

## Advantages of PL/SQL Exceptions

Using exceptions for error handling has several advantages. Without exception handling, every time you issue a command, you must check for execution errors:

```
BEGIN SELECT ...  
  check for 'no data found' error  
SELECT ...  
  check for 'no data found' error  
SELECT ...  
  check for 'no data found' error
```

Error processing is not clearly separated from normal processing; nor is it robust. If you neglect to code a check, the error goes undetected and is likely to cause other, seemingly unrelated errors. With exceptions, you can handle errors conveniently without the need to code multiple checks, as follows:

```
BEGIN SELECT ... SELECT  
  ... SELECT ...  
  ... EXCEPTION  
WHEN NO_DATA_FOUND THEN catches all 'no data found' errors
```

Exceptions improve readability by letting you isolate error-handling routines. If the exception is ever raised in that block (or any sub-block), you can be sure it will be handled.

## Predefined PL/SQL Exceptions

An internal exception is raised automatically if your PL/SQL program violates an Oracle rule or exceeds a system-dependent limit. PL/SQL predefines some common Oracle errors as exceptions. For example, PL/SQL raises the predefined exception `NO_DATA_FOUND` if a `SELECT INTO` statement returns no rows.

You can use the pragma `EXCEPTION_INIT` to associate exception names with other Oracle error codes that you can anticipate. To handle unexpected Oracle errors, you can use the `OTHERS` handler. Within this handler, you can call the functions `SQLCODE` and `SQLERRM` to return the Oracle error code and message text. Once you know the error code, you can use it with pragma `EXCEPTION_INIT` and write a handler specifically for that error.

PL/SQL declares predefined exceptions globally in package `STANDARD`.



Exception	Oracle Error	SQLCODE Value
ACCESS_INTO_NULL	ORA-06530	-6530
CASE_NOT_FOUND	ORA-06592	-6592
COLLECTION_IS_NULL	ORA-06531	-6531
CURSOR_ALREADY_OPEN	ORA-06511	-6511
DUP_VAL_ON_INDEX	ORA-00001	-1
INVALID_CURSOR	ORA-01001	-1001
INVALID_NUMBER	ORA-01722	-1722
LOGIN_DENIED	ORA-01017	-1017
NO_DATA_FOUND	ORA-01403	+100
NOT_LOGGED_ON	ORA-01012	-1012
PROGRAM_ERROR	ORA-06501	-6501
ROWTYPE_MISMATCH	ORA-06504	-6504

You need not declare them yourself. You can write handlers for predefined exceptions using the names in the following list:

Exception	Oracle Error	SQLCODE Value
SELF_IS_NULL	ORA-30625	-30625
STORAGE_ERROR	ORA-06500	-6500
SUBSCRIPT_BEYOND_COUNT	ORA-06533	-6533
SUBSCRIPT_OUTSIDE_LIMIT	ORA-06532	-6532
SYS_INVALID_ROWID	ORA-01410	-1410
TIMEOUT_ON_RESOURCE	ORA-00051	-51
TOO_MANY_ROWS	ORA-01422	-1422
VALUE_ERROR	ORA-06502	-6502
ZERO_DIVIDE	ORA-01476	-1476

Brief descriptions of the predefined exceptions follow:

Exception	Raised when ...
ACCESS_INTO_NULL	A program attempts to assign values to the attributes of an uninitialized object.
CASE_NOT_FOUND	None of the choices in the WHEN clauses of a CASE statement is selected, and there is no ELSE clause.
COLLECTION_IS_NULL	A program attempts to apply collection methods other than EXISTS to an uninitialized nested table or varray, or the program attempts to assign values to the elements of an uninitialized nested table or varray.
CURSOR_ALREADY_OPEN	A program attempts to open an already open cursor. A cursor must be closed before it can be reopened. A cursor FOR loop automatically opens the cursor to which it refers, so your program cannot open that cursor inside the loop.
DUP_VAL_ON_INDEX	A program attempts to store duplicate values in a database column that is constrained by a unique index.
INVALID_CURSOR	A program attempts a cursor operation that is not allowed, such as closing an unopened cursor.
INVALID_NUMBER	In a SQL statement, the conversion of a character string into a number fails because the string does not represent a valid number. (In procedural statements, VALUE_ERROR is raised.) This exception is also raised when the LIMIT-

Exception	Raised when ...
	clause expression in a bulk FETCH statement does not evaluate to a positive number.
LOGIN_DENIED	A program attempts to log on to Oracle with an invalid username or password.
NO_DATA_FOUND	<p>A SELECT INTO statement returns no rows, or your program references a deleted element in a nested table or an uninitialized element in an index-by table.</p> <p>Because this exception is used internally by some SQL functions to signal that they are finished, you should not rely on this exception being propagated if you raise it within a function that is called as part of a query.</p>
NOT_LOGGED_ON	A program issues a database call without being connected to Oracle.
PROGRAM_ERROR	PL/SQL has an internal problem.
ROWTYPE_MISMATCH	The host cursor variable and PL/SQL cursor variable involved in an assignment have incompatible return types. For example, when an open host cursor variable is passed to a stored subprogram, the return types of the actual and formal parameters must be compatible.
SELF_IS_NULL	A program attempts to call a MEMBER method, but the instance of the object type has not been initialized. The built-in parameter SELF points to the object, and is always the first parameter passed to a MEMBER method.
STORAGE_ERROR	PL/SQL runs out of memory or memory has been corrupted.
SUBSCRIPT_BEYOND_COUNT	A program references a nested table or varray element using an index number larger than the number of elements in the collection.
SUBSCRIPT_OUTSIDE_LIMIT	A program references a nested table or varray element using an index number (-1 for example) that is outside the legal range.

SYS_INVALID_ROWID	The conversion of a character string into a universal rowid fails because the character string does not represent a valid rowid.
-------------------	--

Exception	Raised when ...
TIMEOUT_ON_RESOURCE	A time-out occurs while Oracle is waiting for a resource.
TOO_MANY_ROWS	A SELECT INTO statement returns more than one row.
VALUE_ERROR	An arithmetic, conversion, truncation, or size-constraint error occurs. For example, when your program selects a column value into a character variable, if the value is longer than the declared length of the variable, PL/SQL aborts the assignment and raises VALUE_ERROR. In procedural statements, VALUE_ERROR is raised if the conversion of a character string into a number fails. (In SQL statements, INVALID_NUMBER is raised.)
ZERO_DIVIDE	A program attempts to divide a number by zero.

## Defining Your Own PL/SQL Exceptions

PL/SQL lets you define exceptions of your own. Unlike predefined exceptions, user-defined exceptions must be declared and must be raised explicitly by RAISE statements.

### Declaring PL/SQL Exceptions

Exceptions can be declared only in the declarative part of a PL/SQL block, subprogram, or package. You declare an exception by introducing its name, followed by the keyword EXCEPTION. In the following example, you declare an exception named past\_due:

```
DECLARE
    past_due EXCEPTION;
```

Exception and variable declarations are similar. But remember, an exception is an error condition, not a data item. Unlike variables, exceptions cannot appear in assignment statements or SQL statements. However, the same scope rules apply to variables and exceptions.

### Associating a PL/SQL Exception with a Number: Pragma EXCEPTION\_INIT

To handle error conditions (typically ORA- messages) that have no predefined name, you must use the OTHERS handler or the pragma EXCEPTION\_INIT. A **pragma** is a compiler directive that is processed at compile time, not at run time.

In PL/SQL, the pragma `EXCEPTION_INIT` tells the compiler to associate an exception name with an Oracle error number. That lets you refer to any internal exception by name and to write a specific handler for it. When you see an **error stack**, or sequence of error messages, the one on top is the one that you can trap and handle.

You code the pragma `EXCEPTION_INIT` in the declarative part of a PL/SQL block, subprogram, or package using the syntax

```
PRAGMA EXCEPTION_INIT(exception_name, Oracle_error_number);
```

where `exception_name` is the name of a previously declared exception and the number is a negative value corresponding to an ORA- error number. The pragma must appear somewhere after the exception declaration in the same declarative section, as shown in the following example:

```
DECLARE
deadlock_detected EXCEPTION;
PRAGMA EXCEPTION_INIT(deadlock_detected, 60); BEGIN
... -- Some operation that causes an ORA00060 error
EXCEPTION
WHEN deadlock_detected THEN
    -- handle the error
END;
```

### Defining Your Own Error Messages: Procedure `RAISE_APPLICATION_ERROR`

The procedure `RAISE_APPLICATION_ERROR` lets you issue user-defined ORAerror messages from stored subprograms. That way, you can report errors to your application and avoid returning unhandled exceptions.

To call `RAISE_APPLICATION_ERROR`, use the syntax

```
raise_application_error(error_number, message[, {TRUE | FALSE}]);
```

where `error_number` is a negative integer in the range -20000 .. -20999 and `message` is a character string up to 2048 bytes long. If the optional third parameter is `TRUE`, the error is placed on the stack of previous errors. If the parameter is `FALSE` (the default), the error replaces all previous errors.

An application can call `raise_application_error` only from an executing stored subprogram (or method). When called, `raise_application_error` ends the subprogram and returns a user-defined error number and message to the application. The error number and message can be trapped like any Oracle error.

In the following example, you call `raise_application_error` if an employee's salary is missing:

```
CREATE PROCEDURE raise_salary (emp_id NUMBER, amount NUMBER) AS curr_sal NUMBER;
BEGIN
    SELECT sal INTO curr_sal FROM emp WHERE empno = emp_id;
    IF curr_sal IS NULL THEN
        raise_application_error(20101, 'Salary is missing');
    ELSE
        UPDATE emp SET sal = curr_sal + amount WHERE empno = emp_id;
    END IF;
END raise_salary;
```

The calling application gets a PL/SQL exception, which it can process using the error-reporting functions `SQLCODE` and `SQLERRM` in an `OTHERS` handler.

## How PL/SQL Exceptions Are Raised

Internal exceptions are raised implicitly by the run-time system, as are user-defined exceptions that you have associated with an Oracle error number using `EXCEPTION_INIT`. However, other user-defined exceptions must be raised explicitly by `RAISE` statements.

PL/SQL blocks and subprograms should raise an exception only when an error makes it undesirable or impossible to finish processing. You can place `RAISE` statements for a given exception anywhere within the scope of that exception. In the following example, you alert your PL/SQL block to a user-defined exception named `out_of_stock`:

```
DECLARE
    out_of_stock EXCEPTION;
    number_on_hand NUMBER(4);
BEGIN
    IF number_on_hand < 1 THEN
        RAISE out_of_stock;
    END IF;
EXCEPTION
    WHEN out_of_stock THEN
        -- handle the error
END;
```

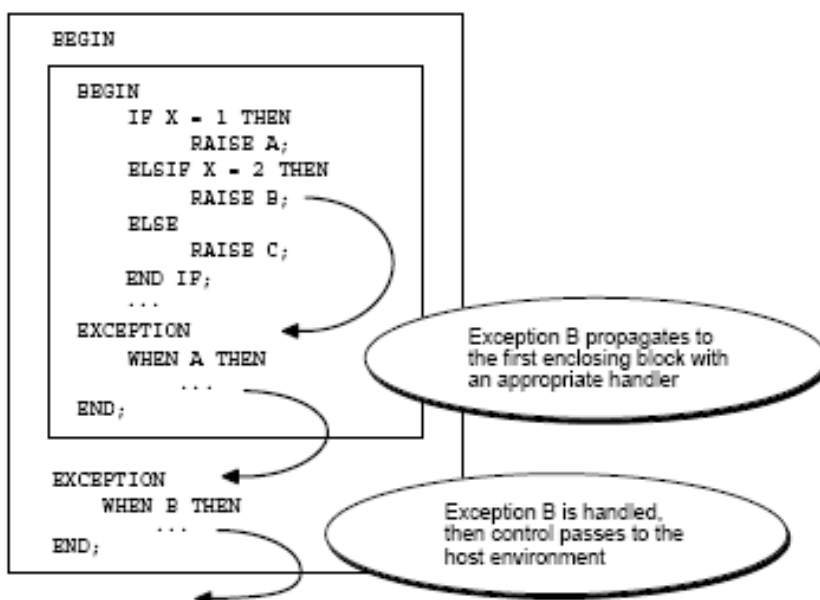
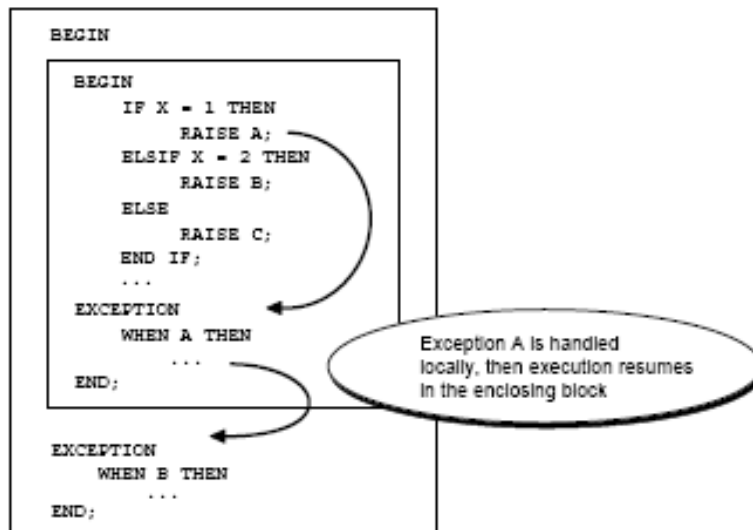
You can also raise a predefined exception explicitly. That way, an exception handler written for the predefined exception can process other errors, as the following example shows:

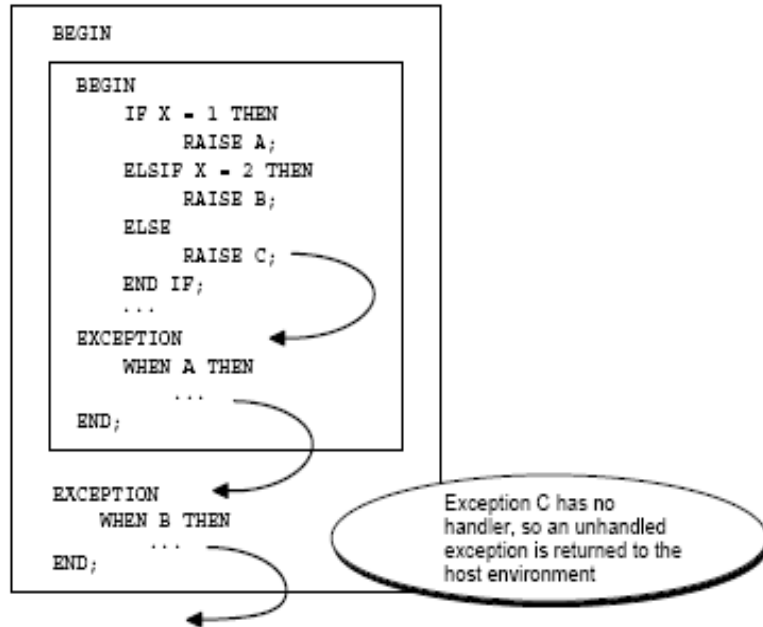
```
DECLARE
    acct_type INTEGER := 7;
BEGIN
    IF acct_type NOT IN (1, 2, 3) THEN
        RAISE INVALID_NUMBER; -- raise predefined exception
    END IF;
EXCEPTION
    WHEN INVALID_NUMBER THEN
        ROLLBACK;
END;
```

## How PL/SQL Exceptions Propagate

When an exception is raised, if PL/SQL cannot find a handler for it in the current block or subprogram, the exception propagates. That is, the exception reproduces itself in successive enclosing blocks until a handler is found or there are no more

blocks to search. In the latter case, PL/SQL returns an unhandled exception error to the host environment.





An exception can propagate beyond its scope, that is, beyond the block in which it was declared. Consider the following example:

```

BEGIN
  ...
  DECLARE -- subblock begins
    past_due EXCEPTION;
  BEGIN
    ...
    IF ... THEN
      RAISE past_due;
    END IF;
  END; -- subblock ends
EXCEPTION
  ...
  WHEN OTHERS THEN
    ROLLBACK;
END;

```

Because the block in which exception `past_due` was declared has no handler for it, the exception propagates to the enclosing block. But, according to the scope rules, enclosing blocks cannot reference exceptions declared in a sub-block. So, only an `OTHERS` handler can catch the exception. If there is no handler for a user-defined exception, the calling application gets the following error:

ORA06510: PL/SQL: unhandled userdefined exception



## Retrieving the Error Code and Error Message: SQLCODE and SQLERRM

In an exception handler, you can use the built-in functions SQLCODE and SQLERRM to find out which error occurred and to get the associated error message. For internal exceptions, SQLCODE returns the number of the Oracle error. The number that SQLCODE returns is negative unless the Oracle error is no data found, in which case SQLCODE returns +100. SQLERRM returns the corresponding error message. The message begins with the Oracle error code.

For user-defined exceptions, SQLCODE returns +1 and SQLERRM returns the message: User- Defined Exception. unless you used the pragma EXCEPTION\_INIT to associate the exception name with an Oracle error number, in which case SQLCODE returns that error number and SQLERRM returns the corresponding error message. The maximum length of an Oracle error message is 512 characters including the error code, nested messages, and message inserts such as table and column names.

If no exception has been raised, SQLCODE returns zero and SQLERRM returns the message: ORA-0000: normal, successful completion. You can pass an error number to SQLERRM, in which case SQLERRM returns the message associated with that error number. Make sure you pass negative error numbers to SQLERRM. In the following example, you pass positive numbers and so get unwanted results:

```
DECLARE

BEGIN
    err_msg VARCHAR2(100);

    /* Get all Oracle error messages. */ FOR err_num
    IN 1..9999 LOOP
        err_msg := SQLERRM(err_num); wrong; should be err_num
        INSERT INTO errors VALUES (err_msg); END LOOP;
    END;
```

Passing a positive number to SQLERRM always returns the message user-defined exception unless you pass +100, in which case SQLERRM returns the message no data found. Passing a zero to SQLERRM always returns the message normal, successful completion.

You cannot use SQLCODE or SQLERRM directly in a SQL statement. Instead, you must assign their values to local variables, then use the variables in the SQL statement, as shown in the following example:

```
DECLARE
    err_num NUMBER;
    err_msg VARCHAR2(100);

BEGIN
    ...
EXCEPTION
    WHEN OTHERS THEN
```

```
err_num := SQLCODE;  
err_msg := SUBSTR(SQLERRM, 1, 100);  
INSERT INTO errors VALUES (err_num, err_msg);  
END;
```

## SUBPROGRAMS (PROCEDURES & FUNCTIONS)

Subprograms are named PL/SQL blocks that can take parameters and be invoked. PL/SQL has two types of subprograms called procedures and functions. Whether you use a function or procedure is often just a matter of programming style. Functions have the advantage of being able to be imbedded in a larger statement. Generally we use a procedure to perform an action and a function to compute a value.

How easy and efficient would it be if the code were shareable? PL/SQL provides you with a mechanism, the stored subprogram, that permits you to share code between applications. A stored subprogram has the major advantage of being stored in the database and it's therefore shareable.

A stored subprogram is a named PL/SQL block that's stored in the database. PL/SQL 9i supports three types of stored subprograms:

- Procedures
- Functions
- Packages

### Understanding PL/SQL Procedures

A procedure is a subprogram that performs a specific action. You write procedures using the syntax:

```
[CREATE [OR REPLACE]] PROCEDURE procedure_name  
[(parameter_name1 parameter_mode datatype,  
... ..  
parameter_nameN parameter_mode datatype)]  
{IS | AS}  
[local declarations] BEGIN  
executable statements  
[EXCEPTION  
exception handlers] END  
[name];
```

where parameter stands for the following syntax:

The CREATE clause lets you create standalone procedures, which are stored in an Oracle database. You can execute the CREATE PROCEDURE statement interactively from SQL\*Plus or from a program using native dynamic SQL.

Procedure\_name is the name of the procedure being created, parameter1 through parameterN are the names of the procedure parameters, parameter\_mode is one of [{IN | OUT [NOCOPY] | IN OUT [NOCOPY]}], and datatype is the data type of the associated parameter. Procedures that take no parameters are written without parentheses.

A procedure has two parts: the specification (spec for short) and the body. The procedure spec begins with the keyword PROCEDURE and ends with the procedure name or a parameter list. Parameter declarations are optional.

The procedure body begins with the keyword IS (or AS) and ends with the keyword END followed by an optional procedure name. The procedure body has three parts: a declarative part, an executable part, and an optional exception-handling part.

The declarative part contains local declarations, which are placed between the keywords IS and BEGIN. The keyword DECLARE, which introduces declarations in an anonymous PL/SQL block, is not used. The executable part contains statements, which are placed between the keywords BEGIN and EXCEPTION (or END). At least one statement must appear in the executable part of a procedure. The NULL statement meets this requirement. The exception-handling part contains exception handlers, which are placed between the keywords EXCEPTION and END.

Consider the procedure raise\_salary, which increases the salary of an employee by a given amount:

```
PROCEDURE raise_salary (emp_id INTEGER, amount REAL) IS
current_salary REAL;
salary_missing EXCEPTION;

BEGIN
SELECT sal
INTO current_salary
FROM emp
WHERE empno = emp_id;
IF current_salary IS NULL THEN RAISE
    salary_missing;
ELSE
    UPDATE emp
    SET sal = sal + amount
    WHERE empno = emp_id; END IF;
EXCEPTION
WHEN NO_DATA_FOUND THEN INSERT INTO
    emp_audit
    VALUES (emp_id, 'No such number'); WHEN
salary_missing THEN
    INSERT INTO emp_audit
        VALUES (emp_id, 'Salary is null');
END raise_salary;
```

When called, this procedure accepts an employee number and a salary increase amount. It uses the employee number to select the current salary from the emp database table. If the employee number is not found or if the current salary is null, an exception is raised. Otherwise, the salary is updated.

A procedure is called as a PL/SQL statement. For example, you might call the procedure raise\_salary as follows:

```
raise_salary(emp_id, amount);
```

The following points are worth noting:

- The signature or the header of the procedure is
- `PROCEDURE raise_salary (emp_id INTEGER, amount REAL)`
- This item is called a signature or header because it identifies the procedure uniquely and is used to execute the procedure or call it from another procedure or a PL/SQL block, or from the client side. Here, the name of the procedure is raise\_salary and it takes two input parameters named emp\_id and amount. The emp\_id parameter is an integer parameter and is of IN mode ( default is IN mode). The parameter amount is of type real and is of IN mode. Note that the length of the parameters isn't specified. This length is determined when the actual data values are passed when the procedure is executed or called.
- The keyword IS follows the signature.
- The local declaration section is

```
current_salary REAL;
```

- The procedure body is the code between BEGIN and END.
- The procedure is called as an executable statement in the PL/SQL block.

## How to create stored procedure:

1. Save the content of the procedure raise\_salary in a file, e.g c:\raise\_salary.sql.
2. Log into to SQL\*PLUS session and type

```
SQL> @ c:\raise_salary.sql.
```

3. In the case of compilation errors, you'll see the message  
Warning: Procedure created with compilation errors.  
You can see the corresponding errors by using the command  
SQL> show errors

in interactive tools such as SQL\*Plus.

4. In the case of successful compilation, you'll see the message Procedure created successfully.
5. This will create an object of type PROCEDURE named raise\_salary in oracle database.

You can call it from a PL/SQL block as follows, once you've created the raise\_salary procedure:

```
BEGIN  
    raise_salary(204589, 50000); END;  
/
```

Here's the output of the preceding program:

```
SQL> BEGIN  
2      raise_salary(204589, 50000);  
3      END;  
4      /
```

PL/SQL procedure successfully completed.

## Understanding PL/SQL Functions

A function is a subprogram that computes a value. Functions are very similar to PL/SQL procedures except for the following differences:

- Functions return a value.
- Functions are used as part of an expression. You write

functions using the syntax:

```
CREATE [OR REPLACE] FUNCTION function_name
[(parameter_name1 parameter_mode datatype,
... ..
parameter_nameN parameter_mode datatype)] RETURN
datatype
{IS | AS}
    [local declaration section] BEGIN
executable section
[EXCEPTION
    exception handling section] END
[function_name];
```

The CREATE clause lets you create standalone functions, which are stored in an Oracle database. You can execute the CREATE FUNCTION statement interactively from SQL\*Plus or from a program using native dynamic SQL.

Function\_name is the name of the function being created, parameter1 through parameterN are the names of the function parameters, parameter\_mode is one of [{IN | OUT [NOCOPY] | IN OUT [NOCOPY]}], and datatype is the data type of the associated parameter. Functions that take no parameters are written without parentheses.

Like a procedure, a function has two parts: the spec and the body. The function spec begins with the keyword FUNCTION and ends with the RETURN clause, which specifies the datatype of the return value. Parameter declarations are optional. You can have more than one RETURN statement in a function, but only one will be executed. One RETURN statement is required even if you return nothing or ignore what is returned.

The function body begins with the keyword IS (or AS) and ends with the keyword END followed by an optional function name. The function body has three parts: a declarative part, an executable part, and an optional exception-handling part.

The declarative part contains local declarations, which are placed between the keywords IS and BEGIN. The keyword DECLARE is not used. The executable part contains statements, which are placed between the keywords BEGIN and EXCEPTION (or END). One or more RETURN statements must appear in the executable part of a function. The exception-handling part contains exception handlers, which are placed between the keywords EXCEPTION and END.

Consider the the emptype function, which determines what type of employee it is:

```
CREATE OR REPLACE FUNCTION emptype (paytype CHAR)

    RETURN VARCHAR2 IS BEGIN
IF paytype = 'H' THEN RETURN
    'Hourly';
ELSIF paytype = 'S' THEN RETURN
    'Salaried';
ELSIF paytype = 'E' THEN RETURN
    'Executive';
```

ELSE

```

RETURN 'Invalid Type';

        END IF; EXCEPTION
        WHEN OTHERS THEN
            RETURN 'Error Encountered'; END
emptytype;
/

```

The above function is titled `emptytype`. It uses parameters called `paytype` of type `CHAR`. When called, this function accepts `pay type` as input parameter. The function returns a value of type `VARCHAR2`, which will be `Hourly`, `Salaried`, `Executive`, `Invalid Type`, or `Error Encountered`. When you begin the function's statements, you use `IF...ELSIF` to determine text to return. If an exception occurs, the function stops processing and returns the value `Error Encountered`. The function is then terminated by calling the `END` statement followed by the function name.

### Using the RETURN Statement

The `RETURN` statement immediately completes the execution of a subprogram and returns control to the caller. Execution then resumes with the statement following the subprogram call. (Do not confuse the `RETURN` statement with the `RETURN` clause in a function spec, which specifies the datatype of the return value.)

In procedures, a `RETURN` statement cannot return a value, and therefore cannot contain an expression. The statement simply returns control to the caller before the normal end of the procedure is reached.

However, in functions, a `RETURN` statement must contain an expression, which is evaluated when the `RETURN` statement is executed. The resulting value is assigned to the function identifier, which acts like a variable of the type specified in the `RETURN` clause. Observe how the function `balance` returns the balance of a specified bank account:

```

FUNCTION balance (acct_id INTEGER) RETURN REAL IS
acct_bal REAL; BEGIN
SELECT bal INTO acct_bal
FROM accts
WHERE acct_no = acct_id; RETURN
acct_bal;
END balance;

```

## Actual Versus Formal Subprogram Parameters

The variables declared in a subprogram spec and referenced in the subprogram body are *formal* parameters. For example, the following procedure declares two formal parameters named `emp_id` and `amount`:

```

PROCEDURE raise_salary (emp_id INTEGER, amount REAL) IS BEGIN
    UPDATE emp
    SET sal = sal + amount
    WHERE empno = emp_id; END
raise_salary;

```

Subprograms pass information using *parameters*. The variables or expressions referenced in the parameter list of a subprogram call are *actual* parameters. For example, the following procedure call lists two actual parameters named `emp_num` and `in_amount`:

```
raise_salary(emp_num, in_amount);
```

A good programming practice is to use different names for actual and formal parameters.

## Positional Versus Named Notation for Subprogram Parameter

When calling a subprogram, you can write the actual parameters using either positional or named notation or both. That is, you can indicate the association between an actual and formal parameter by position or name or both.

So, given the declarations

```
DECLARE
acct INTEGER;
amt REAL;
PROCEDURE credit_acct (acct_no INTEGER, amount REAL) IS ...
```

you can call the procedure `credit_acct` in four logically equivalent ways:

```
BEGIN
credit_acct(acct, amt); -- positional notation credit_acct(amount =>
amt, acct_no => acct); -- named notation credit_acct(acct_no => acct,
amount => amt); -- named notation credit_acct(acct, amount => amt); --
mixed notation
```

### Using Positional Notation

The first procedure call uses positional notation. The PL/SQL compiler associates the first actual parameter, `acct`, with the first formal parameter, `acct_no`. And, the compiler associates the second actual parameter, `amt`, with the second formal parameter, `amount`.

### Using Named Notation

The second procedure call uses named notation. An arrow (`=>`) serves as the association operator, which associates the formal parameter to the left of the arrow with the actual parameter to the right of the arrow. The third procedure call also uses named notation and shows that you can list the parameter pairs in any order. So, you need not know the order in which the formal parameters are listed.

### Using Mixed Notation

The fourth procedure call shows that you can mix positional and named notation. In this case, the first parameter uses positional notation, and the second parameter uses named notation. Positional notation must precede named notation. The reverse is not allowed. For example, the following procedure call is illegal:

```
credit_acct(acct_no => acct, amt); -- illegal
```

## Specifying Subprogram Parameter Modes

You use parameter modes to define the behavior of formal parameters. The three parameter modes, `IN` (the default), `OUT`, and `IN OUT`, can be used with any subprogram.

### Using the IN Mode

An `IN` parameter lets you pass values to the subprogram being called. Inside the subprogram, an `IN` parameter acts like a constant. Therefore, it cannot be assigned a value.

For example, the following assignment statement causes a compilation error:

```
PROCEDURE insert_employee ( in_emp_id IN EMPLOYEE.emp_id%TYPE,
                           in_firstname IN EMPLOYEE.emp_first_name%TYPE,
```



```

                in_lastname IN EMPLOYEE.emp_last_name%TYPE )
IS
    v_count NUMBER;
BEGIN
    SELECT count(1)
    INTO v_count
    FROM EMPLOYEE
    WHERE emp_id = in_emp_id;
    in_firstname := 'Mr.' || in_firstname; -- causes compilation error
    IF v_count = 0
    THEN
        INSERT INTO EMPLOYEE( emp_id, emp_first_name, emp_last_name)
        VALUES ( in_emp_id, in_firstname, in_lastname);
        COMMIT;
    END IF;
EXCEPTION
WHEN OTHERS
THEN
    ROLLBACK;
    DBMS_OUTPUT.PUT_LINE( 'Error occurred while insert employee');
END insert_employee;

```

### Using the OUT Mode

An OUT parameter lets you return values to the caller of a subprogram. Inside the subprogram, an OUT parameter acts like a variable. That means you can use an OUT formal parameter as if it were a local variable. You can change its value or reference the value in any way, as the following example shows:

```

PROCEDURE insert_employee(
    in_emp_id IN EMPLOYEE.emp_id%TYPE,
    in_firstname IN EMPLOYEE.emp_first_name%TYPE,
    in_lastname IN EMPLOYEE.emp_last_name%TYPE,
    out_return_flag OUT BOOLEAN )
IS
    v_count NUMBER;
BEGIN
    SELECT count(1)
    INTO v_count
    FROM EMPLOYEE
    WHERE emp_id = in_emp_id;
    in_firstname := 'Mr.' || in_firstname; -- causes compilation error
    IF v_count = 0
    THEN
        INSERT INTO EMPLOYEE( emp_id, emp_first_name, emp_last_name)
        VALUES ( in_emp_id, in_firstname, in_lastname);
        out_return_flag := TRUE;
        COMMIT;
    END IF;
EXCEPTION
WHEN OTHERS
THEN
    out_return_flag := FALSE;
    ROLLBACK;
    DBMS_OUTPUT.PUT_LINE( 'Error occurred while insert employee');
END insert_employee;

```

The actual parameter that corresponds to an OUT formal parameter must be a variable; it cannot be a constant or an expression. For example, the following procedure call is illegal:

```
insert_employee (23456, 'Ian', 'Kately', TRUE); causes compilation error
```

### Using the IN OUT Mode

An IN OUT parameter lets you pass initial values to the subprogram being called and return updated values to the caller. Inside the subprogram, an IN OUT parameter acts like an initialized variable. Therefore, it can be assigned a value and its value can be assigned to another variable. The actual parameter that corresponds to an IN OUT formal parameter must be a variable; it cannot be a constant or an expression.

```
PROCEDURE DPD_SEQ_NUM( pnm_seq_num IN OUT CL_SEQ_NUM.seq_num%TYPE, pch_tran_tp
                      IN          CHAR )
BEGIN
    IF pch_tran_tp = 'S' THEN
        /*
        ##          For Selection Max Sequence Number.
        */ BEGIN
            SELECT seq_num
            INTO      pnm_seq_num
            FROM      CL_SEQ_NUM;
        EXCEPTION
            WHEN others
            THEN
                /* Populate Error Log */
                dbms_output.put_line('Error during selection of seq number');

        END;
        ELSIF pch_tran_tp = 'U' THEN
            /*
            ##          For Updation of Max Sequence Number.
            */ BEGIN
                UPDATE cl_seq_num
                SET      seq_num = pnm_seq_num, upd_dt =
                        SYSDATE;
                COMMIT ;
            EXCEPTION
            WHEN others THEN
                /* Populate Error Log */
                dbms_output.put_line('Error during updation of seq number');
            END; END IF ;
        END DPD_SEQ_NUM ;
```

### Passing Large Data Structures with the NOCOPY Compiler Hint

Suppose a subprogram declares an IN parameter, an OUT parameter, and an IN OUT parameter. When you call the subprogram, the IN parameter is passed by reference. That is, a pointer to the IN actual parameter is passed to the corresponding formal parameter. So, both parameters reference the same memory location, which holds the value of the actual parameter.

By default, the OUT and IN OUT parameters are passed by value. That is, the value of the IN OUT actual parameter is copied into the corresponding formal parameter. Then, if the subprogram exits normally, the values assigned to the OUT and IN OUT formal parameters are copied into the corresponding actual parameters.

When the parameters hold large data structures such as collections, records, and instances of object types, all this copying slows down execution and uses up memory. To prevent that, you can specify the NOCOPY hint, which allows the PL/SQL compiler to pass OUT and IN OUT parameters by reference. PL/SQL 8i onward defines the compiler hint NOCOPY.

Here's the syntax of using NOCOPY:

```
parameter_name [OUT| IN OUT] NOCOPY datatype
```

where parameter\_name is the name of the parameter and datatype represents the data type of said parameter. You can use either the OUT or IN OUT parameter mode with NOCOPY. Here's an example:

```
DECLARE
  TYPE tab_emp_id IS TABLE OF EMPLOYEE.emp_id%TYPE INDEX BY BINARY_INTEGER;
  TYPE tab_firstname IS TABLE OF EMPLOYEE.emp_first_name%TYPE INDEX BY
  BINARY_INTEGER;
  TYPE tab_lastname IS TABLE OF EMPLOYEE.emp_last_name%TYPE INDEX BY
  BINARY_INTEGER;
  PROCEDURE getEmployees ( ptab_emp_id OUT NOCOPY tab_emp_id,
                           ptab_firstname OUT NOCOPY tab_firstname,
                           ptab_lastname OUT NOCOPY tab_lastname )
  IS
  .....
```

### Behavior of Various Parameter Modes

IN	IN OUT	OUT	IN OUT NOCOPY, OUT NOCOPY
Actual parameter is passed by reference.	The actual parameter is passed by value.	The actual parameter is passed by value.	The actual parameter is passed by reference.
A pointer (address) to the value is passed.	A copy of the value is passed out.	A copy of the value is passed in and out.	The address of the value is passed.
N/A	The OUT value is rolled back in case of an unhandled exception.	The OUT value is rolled back in case of an unhandled exception.	You can't predict the correctness of the OUT value always, as no rollback occurs in the case of an unhandled exception.

**TIP:** You can use NOCOPY only for parameters in OUT or IN OUT mode. NOCOPY is a compiler hint and not a directive, and hence it may be ignored sometimes. With NOCOPY, changes to formal parameter values affect the values of the actual parameters also, so if a subprogram exits with an unhandled exception, the unfinished changes are not rolled back. That is, the actual parameters may still return modified values.

## Performance Improvement of NOCOPY

NOCOPY improves performance when passing large data structures. It's significantly faster than passing by value. To illustrate the performance benefit of NOCOPY, consider the following small package consisting of two procedures and a function. Here, I declare a collection of type VARRAY containing 100,000 elements:

```
CREATE OR REPLACE PACKAGE NoCopyPkg is
  type arr is varray(100000) of hrc_tab%ROWTYPE;
  procedure p1(ip1 IN OUT arr); procedure p2(ip1
  IN OUT NOCOPY arr); FUNCTION get_time RETURN
  NUMBER;
END NoCopyPkg;
/
```

The corresponding package body is defined as follows:

```
CREATE OR REPLACE PACKAGE BODY NoCopyPkg is

  PROCEDURE p1(ip1 IN OUT arr) IS
  BEGIN NULL;
```

```

END;
PROCEDURE p2(ip1 IN OUT NOCOPY arr) IS
BEGIN NULL;
END;
FUNCTION get_time RETURN NUMBER IS
BEGIN
    RETURN (dbms_utility.get_time); EXCEPTION
WHEN OTHERS THEN
    RAISE_APPLICATION_ERROR(-20010, SQLERRM); END
get_time;
END NoCopyPkg;
/

```

This package defines a collection type called VARRAY, or variable array, having a maximum of 100,000 elements. It also defines two procedures that each pass an array of 100,000 elements in the IN OUT mode, one without NOCOPY and the other with NOCOPY specified. The function get\_time retrieves the current time in one-hundredth of a second.

Now, I can use the packaged procedures and functions by defining the following PL/SQL block:

```

declare
    arr1 NoCopyPkg.arr := NoCopyPkg.arr(null);
    cur_t1    number;    cur_t2
    number; cur_t3 number;
begin
    select * into arr1(1) from hrc_tab where hrc_code = 1;
    /* Create 99999 new elements in the variable array
       and populate each with the value in the 1st element */
    arr1.extend(99999, 1);
    cur_t1 := NoCopyPkg.get_time;
    NoCopyPkg.p1(arr1);
    cur_t2 := NoCopyPkg.get_time;
    NoCopyPkg.p2(arr1);
    cur_t3 := NoCopyPkg.get_time;
    dbms_output.put_line(' Without NOCOPY '||to_char((cur_t2-cur_t1)/100));
    dbms_output.put_line(' With NOCOPY '||to_char((cur_t3-cur_t2)/100));
end;
/

```

Here's the output of the preceding program:

```

Without NOCOPY .67
With NOCOPY 0

```

PL/SQL procedure successfully completed.

The output of the preceding program may vary depending on the hardware and software configuration, but you should note the difference in speed between using NOCOPY and not using it.

The following points are worth noting:

- ✱ The procedures p1 and p2 inside the package do nothing except pass the variable array of 100,000 elements. p1 uses call by value and p2 uses call by reference.
- ✱ The SELECT statement at the beginning of the PL/SQL block populates the very first element of the variable array arr1. Note that arr1 is initialized to null, meaning it creates one element that is null.
- ✱ The EXTEND method is used on the variable array to create 99,999 copies of the first element.
- ✱ The time taken to pass by value is significantly more than that taken to pass by reference.

## Restrictions on NOCOPY

As mentioned earlier, NOCOPY is only a compiler hint and not a directive. So in some cases it may be ignored and the parameter passed by value. Certain restrictions apply to NOCOPY. For NOCOPY to take effect

- ✦ The actual parameter can't be an element of an index-by table, but it can be an entire index-by table.
- ✦ The actual parameter can't be constrained by precision, scale, or NOT NULL. However, this restriction doesn't apply to constrained elements or attributes, or length-constrained character strings.
- ✦ The actual and formal parameters can't be explicitly defined records with either one or both records defined using %ROWTYPE or %TYPE, and constraints on corresponding fields in the records differ. This restriction also applies if the actual parameter is defined by means of an implicit record such as the index of a cursor FOR LOOP.
- ✦ Passing the actual parameter requires an implicit data type conversion.
- ✦ The subprogram containing the NOCOPY parameters can't be involved in an external or remote procedure call.

## Using Default Values for Subprogram Parameters

As the example below shows, you can initialize IN parameters to default values. That way, you can pass different numbers of actual parameters to a subprogram, accepting or overriding the default values as you please. Moreover, you can add new formal parameters without having to change every call to the subprogram.

```
PROCEDURE create_dept (  
  new_dname VARCHAR2 DEFAULT 'TEMP', new_loc  
  VARCHAR2 DEFAULT 'TEMP') IS BEGIN  
  INSERT INTO dept  
  VALUES (deptno_seq.NEXTVAL, new_dname, new_loc);  
... END;
```

If an actual parameter is not passed, the default value of its corresponding formal parameter is used. Consider the following calls to create\_dept:

```
create_dept; create_dept('MARKETING');  
create_dept('MARKETING', 'NEW YORK');
```

The first call passes no actual parameters, so both default values are used. The second call passes one actual parameter, so the default value for new\_loc is used. The third call passes two actual parameters, so neither default value is used.

## PL/SQL PACKAGES

### What Is a PL/SQL Package?

A **package** is a schema object that groups logically related PL/SQL types, items, and subprograms. Packages usually have two parts, a specification and a body, although sometimes the body is unnecessary. The **specification** (spec for short) is the interface to your applications; it declares the types, variables, constants, exceptions, cursors, and subprograms available for use. The **body** fully defines cursors and subprograms, and so implements the spec.

The spec holds **public declarations**, which are visible to your application. The body holds implementation details and **private declarations**, which are hidden from your application.

#### Example of a PL/SQL Package

In the example below, you package a record type, a cursor, and two employment procedures. Notice that the procedure `hire_employee` uses the database sequence `empno_seq` and the function `SYSDATE` to insert a new employee number and hire date, respectively.

```
CREATE OR REPLACE PACKAGE emp_actions AS
spec
  TYPE EmpRecTyp IS RECORD (emp_id INT, salary REAL); CURSOR
  desc_salary RETURN EmpRecTyp;

  PROCEDURE hire_employee (
    ename VARCHAR2, job
    VARCHAR2, mgr NUMBER,
    sal NUMBER, comm NUMBER,
    deptno NUMBER);

  PROCEDURE fire_employee (emp_id NUMBER); END
emp_actions;

CREATE OR REPLACE PACKAGE BODY emp_actions AS
body
  CURSOR desc_salary RETURN EmpRecTyp IS
    SELECT empno, sal FROM emp ORDER BY sal DESC;

  PROCEDURE hire_employee (
    ename VARCHAR2, job
    VARCHAR2, mgr NUMBER,
    sal NUMBER, comm NUMBER,
    deptno NUMBER) IS
  BEGIN
    INSERT INTO emp VALUES (empno_seq.NEXTVAL, ename, job, mgr, SYSDATE,
                             sal, comm, deptno);
  END hire_employee;
```

```
PROCEDURE fire_employee (emp_id NUMBER) IS BEGIN
    DELETE FROM emp WHERE empno = emp_id; END
    fire_employee;
END emp_actions;
```

## Advantages of PL/SQL Packages

Packages offer several advantages: modularity, easier application design, information hiding, added functionality, and better performance.

### Modularity

Packages let you encapsulate logically related types, items, and subprograms in a named PL/SQL module. Each package is easy to understand, and the interfaces between packages are simple, clear, and well defined. This aids application development.

### Easier Application Design

When designing an application, all you need initially is the interface information in the package specs. You can code and compile a spec without its body. Then, stored subprograms that reference the package can be compiled as well. You need not define the package bodies fully until you are ready to complete the application.

### Information Hiding

With packages, you can specify which types, items, and subprograms are public (visible and accessible) or private (hidden and inaccessible). For example, if a package contains four subprograms, three might be public and one private. The package hides the implementation of the private subprogram so that only the package (not your application) is affected if the implementation changes. This simplifies maintenance and enhancement. Also, by hiding implementation details from users, you protect the integrity of the package.

### Added Functionality

Packaged public variables and cursors persist for the duration of a session. So, they can be shared by all subprograms that execute in the environment. Also, they allow you to maintain data across transactions without having to store it in the database.

### Better Performance

When you call a packaged subprogram for the first time, the whole package is loaded into memory. So, later calls to related subprograms in the package require no disk I/O. Also, packages stop cascading dependencies and thereby avoid unnecessary recompiling. For example, if you change the implementation of a packaged function, Oracle need not recompile the calling subprograms because they do not depend on the package body.

## Understanding The Package Spec

The package spec contains public declarations. The scope of these declarations is local to your database schema and global to the package.

The spec lists the package resources available to applications. All the information your application needs to use the resources is in the spec. For example, the following declaration shows that the function named `fac` takes one argument of type `INTEGER` and returns a value of type `INTEGER`:

```
FUNCTION fac (n INTEGER) RETURN INTEGER; returns n!
```

That is all the information you need to call the function.

Only subprograms and cursors have an underlying implementation. So, if a spec declares only types, constants, variables, exceptions, and call specs, the package body is unnecessary.

## Referencing Package Contents

To reference the types, items, subprograms, and call specs declared within a package spec, use dot notation, as follows:

```
package_name.type_name  
package_name.item_name  
package_name.subprogram_name  
package_name.call_spec_name
```

## Understanding The Package Body

The package body implements the package spec. That is, the package body contains the implementation of every cursor and subprogram declared in the package spec. Keep in mind that subprograms defined in a package body are accessible outside the package only if their specs also appear in the package spec.

The package body can also contain private declarations, which define types and items necessary for the internal workings of the package. The scope of these declarations is local to the package body. Therefore, the declared types and items are inaccessible except from within the package body.

Remember, if a package spec declares only types, constants, variables, exceptions, and call specs, the package body is unnecessary. However, the body can still be used to initialize items declared in the package spec.

## Overloading Packaged Subprograms

PL/SQL allows two or more packaged subprograms to have the same name. This option is useful when you want a subprogram to accept similar sets of parameters that have different datatypes. For example, the following package defines two procedures named `journalize`:

```
CREATE PACKAGE journal_entries AS  
...  
...
```



```
PROCEDURE journalize (amount REAL, trans_date VARCHAR2); PROCEDURE
journalize (amount REAL, trans_date INT);
END journal_entries;

CREATE PACKAGE BODY journal_entries AS
...
PROCEDURE journalize (amount REAL, trans_date VARCHAR2) IS BEGIN
    INSERT INTO journal
    VALUES (amount, TO_DATE(trans_date, 'DDMONYYYY')); END journalize;

PROCEDURE journalize (amount REAL, trans_date INT) IS BEGIN
    INSERT INTO journal
    VALUES (amount, TO_DATE(trans_date, 'J')); END
journalize;
END journal_entries;
```

The first procedure accepts trans\_date as a character string, while the second procedure accepts it as a number (the Julian day). Each procedure handles the data appropriately.

## DATABASE TRIGGER

A trigger:

- Is a PL/SQL block or a PL/SQL procedure associated with a table, view, schema, or the database
- Executes implicitly whenever a particular event takes place

### Objectives

Describe different types of triggers  
Describe database triggers and their use  
Create database triggers  
Describe database trigger firing rules  
Remove database triggers

### Different Types of Triggers

Application trigger: Fires whenever an event occurs with a particular application

Database trigger: Fires whenever a data event (such as DML) or system event (such as logon or shutdown) occurs on a schema or database

Application triggers execute implicitly whenever a particular data manipulation language (DML) event occurs within an application. An example of an application that uses triggers extensively is one developed with Oracle Forms Developer.

Database triggers execute implicitly when a data event such as DML on a table (an INSERT, UPDATE, or DELETE triggering statement), an INSTEAD OF trigger on a view, or data definition language (DDL) statements such as CREATE and ALTER are issued, no matter which user is connected or which application is used. Database triggers also execute implicitly when some user actions or database system actions occur, for example, when a user logs on, or the DBA shut downs the database.

**Note:** Database triggers can be defined on tables and on views. If a DML operation is issued on a view, the INSTEAD OF trigger defines what actions take place. If these actions include DML operations on tables, then any triggers on the base tables are fired.

Database triggers can be system triggers on a database or a schema. With a database, triggers fire for each event for all users; with a schema, triggers fire for each event for that specific user.

### Guidelines for Designing Triggers

- Design triggers to:
  - Perform related actions
  - Centralize global operations
- Do not design triggers:
  - Where functionality is already built into the Oracle server
  - That duplicate other triggers

- Create stored procedures and invoke them in a trigger, if the PL/SQL code is very lengthy.
- The excessive use of triggers can result in complex interdependencies, which may be difficult to maintain in large applications.

## Creating DML Triggers

A triggering statement contains:

- Trigger timing
  - For table: BEFORE, AFTER
  - For view: INSTEAD OF
- Triggering event: INSERT, UPDATE, or DELETE
- Table name: On table, view
- Trigger type: Row or statement
- WHEN clause: Restricting condition
- Trigger body: PL/SQL block

Before coding the trigger body, decide on the values of the components of the trigger: the trigger timing, the triggering event, and the trigger type.

If multiple triggers are defined for a table, be aware that the order in which multiple triggers of the same type fire is arbitrary. To ensure that triggers of the same type are fired in a particular order, consolidate the triggers into one trigger that calls separate procedures in the desired order.

### **DML Trigger Components:**

Trigger timing: When should the trigger fire?

- BEFORE: Execute the trigger body before the triggering DML event on a table.
- AFTER: Execute the trigger body after the triggering DML event on a table.
- INSTEAD OF: Execute the trigger body instead of the triggering statement. This is used for views that are not otherwise modifiable.

### **BEFORE Triggers**

This type of trigger is frequently used in the following situations:

- To determine whether that triggering statement should be allowed to complete. (This situation enables you to eliminate unnecessary processing of the triggering statement and its eventual rollback in cases where an exception is raised in the triggering action.)
- To derive column values before completing a triggering INSERT or UPDATE statement.
- To initialize global variables or flags, and to validate complex business rules.

### **AFTER Triggers**

This type of trigger is frequently used in the following situations:

- To complete the triggering statement before executing the triggering action.
- To perform different actions on the same triggering statement if a BEFORE trigger is already present.

### INSTEAD OF Triggers

This type of trigger is used to provide a transparent way of modifying views that cannot be modified directly through SQL DML statements because the view is not inherently modifiable.

You can write INSERT, UPDATE, and DELETE statements against the view. The INSTEAD OF trigger works invisibly in the background performing the action coded in the trigger body directly on the underlying tables.

**Triggering user event:** Which DML statement causes the trigger to execute? You can use any of the following:

• INSERT

---

• UPDATE

• DELETE

The triggering event or statement can be an INSERT, UPDATE, or DELETE statement on a table.

- When the triggering event is an UPDATE statement, you can include a column list to identify which columns must be changed to fire the trigger. You cannot specify a column list for an INSERT or for a DELETE statement, because they always affect entire rows.

... UPDATE OF salary ...

- The triggering event can contain one, two, or all three of these DML operations.

... INSERT or UPDATE or DELETE

... INSERT or UPDATE OF job\_id ...

**Trigger type:** Should the trigger body execute for each row the statement affects or only once?

- Statement: The trigger body executes once for the triggering event. This is the default. A statement trigger fires once, even if no rows are affected at all.

- Row: The trigger body executes once for each row affected by the triggering event. A row trigger is not executed if the triggering event affects no rows.

Statement Triggers and Row Triggers

You can specify that the trigger will be executed once for every row affected by the triggering statement (such as a multiple row UPDATE) or once for the triggering statement, no matter how many rows it affects.

### Statement Trigger

A statement trigger is fired once on behalf of the triggering event, even if no rows are affected at all. Statement triggers are useful if the trigger action does not depend on the data from rows that are affected or on data provided by the triggering event itself: for example, a trigger that performs a complex security check on the current user.

### Row Trigger

A row trigger fires each time the table is affected by the triggering event. If the triggering event affects no rows, a row trigger is not executed.

Row triggers are useful if the trigger action depends on data of rows that are affected or on data provided by the triggering event itself.

**Trigger body:** What action should the trigger perform?

The trigger body is a PL/SQL block or a call to a procedure.

The trigger action defines what needs to be done when the triggering event is issued. The PL/SQL block can contain SQL and PL/SQL statements, and can define PL/SQL constructs such as variables, cursors, exceptions, and so on. You can also call a PL/SQL procedure or a Java procedure.

Additionally, row triggers use correlation names to access the old and new column values of the row being processed by the trigger.

Note: The size of a trigger cannot be more than 32 K.

### **Syntax for Creating DML Statement Triggers** *CREATE [OR*

*REPLACE] TRIGGER trigger\_name timing*

*event1 [OR event2 OR event3] ON*

*table\_name*

*trigger\_body*

Note: Trigger names must be unique with respect to other triggers in the same schema.

*CREATE OR REPLACE TRIGGER secure\_emp*

*BEFORE INSERT ON employees*

*BEGIN*

*IF (TO\_CHAR(SYSDATE,'DY') IN ('SAT','SUN')) OR*

*(TO\_CHAR(SYSDATE,'HH24:MI')*

*NOT BETWEEN '08:00' AND '18:00')*

*THEN RAISE\_APPLICATION\_ERROR (-20500,'You may insert into*

*EMPLOYEES table only*

*during business hours.');*

*END IF;*

*END;*

*/*

### **Example:**

You can create a BEFORE statement trigger in order to prevent the triggering operation from succeeding if a certain condition is violated.

For example, create a trigger to restrict inserts into the EMPLOYEES table to certain business hours, Monday through Friday.

If a user attempts to insert a row into the EMPLOYEES table on Saturday, the user sees the message, the trigger fails, and the triggering statement is rolled back. Remember that the RAISE\_APPLICATION\_ERROR is a server-side built-in procedure that returns an error to the user and causes the PL/SQL block to fail.

When a database trigger fails, the triggering statement is automatically rolled back by the Oracle server.

**Testing SECURE\_EMP**

```
INSERT INTO employees (employee_id, last_name, first_name,
email, hire_date,
job_id, salary, department_id)
VALUES (300, 'Smith', 'Rob', 'RSMITH', SYSDATE,
'IT_PROG', 4500, 60);
```

```
INSERT INTO employees (employee_id, last_name, first_name, email,
*
ERROR at line 1:
ORA-20500: You may insert into EMPLOYEES table only during business hours.
ORA-06512: at "PLSQL.SECURE_EMP", line 4
ORA-04088: error during execution of trigger 'PLSQL.SECURE_EMP'
```

**Example**

Insert a row into the EMPLOYEES table during non business hours. When the date and time are out of the business timings specified in the trigger, you get the error message as shown in the slide.

**Creating a DML Row Trigger**

```
CREATE [OR REPLACE] TRIGGER trigger_name timing
event1 [OR event2 OR event3] ON
table_name
[REFERENCING OLD AS old | NEW AS new] FOR EACH
ROW
[WHEN (condition)]
trigger_body Example:
CREATE OR REPLACE TRIGGER restrict_salary BEFORE INSERT
OR UPDATE OF salary ON employees FOR EACH ROW
BEGIN
IF NOT (:NEW.job_id IN ('AD_PRES', 'AD_VP')) AND
:NEW.salary > 15000
THEN
RAISE_APPLICATION_ERROR (-20202, 'Employee cannot earn
this amount');
END IF; END;
/
```

You can create a BEFORE row trigger in order to prevent the triggering operation from succeeding if a certain condition is violated.

Create a trigger to allow only certain employees to be able to earn a salary of more than 15,000. If a user attempts to do this, the trigger raises an error.

```
UPDATE employees
SET salary = 15500
WHERE last_name = 'Russell';
```

```
UPDATE EMPLOYEES
*
```

ERROR at line 1:

ORA-20202: Employee can not earn this amount

ORA-06512: at "PLSQL.RESTRICT\_SALARY", line 5

ORA-04088: error during execution of trigger 'PLSQL.RESTRICT\_SALARY'

## Using OLD and NEW Qualifiers

```
CREATE OR REPLACE TRIGGER audit_emp_values AFTER DELETE
OR INSERT OR UPDATE ON employees FOR EACH ROW
BEGIN
INSERT INTO audit_emp_table (user_name, timestamp, id,
old_last_name, new_last_name, old_title,
new_title, old_salary, new_salary)
VALUES (USER, SYSDATE, :OLD.employee_id,
:OLD.last_name, :NEW.last_name, :OLD.job_id,
:NEW.job_id, :OLD.salary, :NEW.salary); END;
/
```

Within a ROW trigger, reference the value of a column before and after the data change by prefixing it with the OLD and NEW qualifier

Data Operation	Old Value	New Value
INSERT	NULL	Inserted value
UPDATE	Value before update	Value after update
DELETE	Value before delete	NULL

The OLD and NEW qualifiers are available only in ROW triggers.

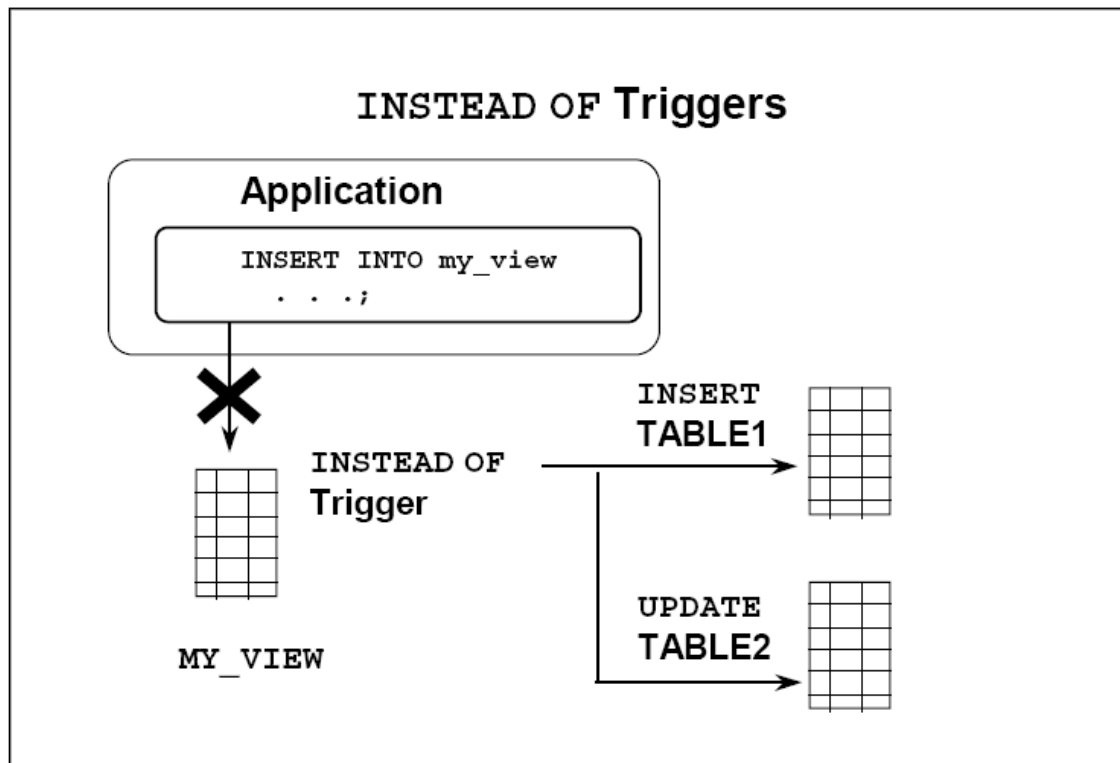
- Prefix these qualifiers with a colon (:) in every SQL and PL/SQL statement.
- There is no colon (:) prefix if the qualifiers are referenced in the WHEN restricting condition.

**Note:** Row triggers can decrease the performance if you do a lot of updates on larger tables.

To restrict the trigger action to those rows that satisfy a certain condition, provide a WHEN clause. Create a trigger on the EMPLOYEES table to calculate an employee's commission when a row is added to the EMPLOYEES table, or when an employee's salary is modified.

The NEW qualifier cannot be prefixed with a colon in the WHEN clause because the WHEN clause is outside the PL/SQL blocks.

## INSTEAD OF Triggers



Use INSTEAD OF triggers to modify data in which the DML statement has been issued against an inherently non-updatable view. These triggers are called INSTEAD OF triggers because, unlike other triggers, the Oracle server fires the trigger instead of executing the triggering statement. This trigger is used to perform an INSERT, UPDATE, or DELETE operation directly on the underlying tables.

You can write INSERT, UPDATE, or DELETE statements against a view, and the INSTEAD OF trigger works invisibly in the background to make the right actions take place.

### Why Use INSTEAD OF Triggers?

A view cannot be modified by normal DML statements if the view query contains set operators, group functions, clauses such as GROUP BY, CONNECT BY, START, the DISTINCT operator, or joins.

For example, if a view consists of more than one table, an insert to the view may entail an insertion into one table and an update to another. So, you write an INSTEAD OF trigger that fires when you write an insert against the view. Instead of the original insertion, the trigger body executes, which results in an insertion of data into one table and an update to another table.



Note: If a view is inherently updateable and has INSTEAD OF triggers, the triggers take precedence. INSTEAD OF triggers are row triggers.

The CHECK option for views is not enforced when insertions or updates to the view are performed by using INSTEAD OF triggers. The INSTEAD OF trigger body must enforce the check.

### Creating an INSTEAD OF Trigger

```
CREATE [OR REPLACE] TRIGGER trigger_name
INSTEAD OF
event1 [OR event2 OR event3] ON
view_name
[REFERENCING OLD AS old | NEW AS new] [FOR
EACH ROW]

trigger_body
```

### Syntax:

trigger_name	Is the name of the trigger.
INSTEAD OF	Indicates that the trigger belongs to a view
event	Identifies the data manipulation operation that causes the trigger to fire: INSERT UPDATE [OF column] DELETE
view_name	Indicates the view associated with trigger
REFERENCING	Specifies correlation names for the old and new values of the current row (The defaults are OLD and NEW)
FOR EACH ROW	Designates the trigger to be a row trigger; INSTEAD OF triggers can only be row triggers: if this is omitted, the trigger is still defined as a row trigger.
trigger body	Is the trigger body that defines the action performed by the trigger, beginning with either DECLARE or BEGIN, and ending with END or a call to a procedure

**Note:** INSTEAD OF triggers can be written only for views. BEFORE and AFTER options are not valid.

You can create an INSTEAD OF trigger in order to maintain the base tables on which a view is based.

Assume that an employee name will be inserted using the view EMP\_DETAILS that is created based on the EMPLOYEES and DEPARTMENTS tables. Create a trigger that results in the appropriate INSERT and UPDATE to the base tables. Because of the INSTEAD OF TRIGGER on the view EMP\_DETAILS, instead of inserting the new employee record into the EMPLOYEES table:

- A row is inserted into the NEW\_EMPS table.

- The `TOTAL_DEPT_SAL` column of the `NEW_DEPTS` table is updated. The salary value supplied for the new employee is added to the existing total salary of the department to which the new employee has been assigned.

## Differentiating Between Database Triggers and Stored Procedures

Triggers	Procedures
Defined with <code>CREATE TRIGGER</code>	Defined with <code>CREATE PROCEDURE</code>
Data dictionary contains source code in <code>USER_TRIGGERS</code>	Data dictionary contains source code in <code>USER_SOURCE</code>
Implicitly invoked	Explicitly invoked
<code>COMMIT</code> , <code>SAVEPOINT</code> , and <code>ROLLBACK</code> are not allowed	<code>COMMIT</code> , <code>SAVEPOINT</code> , and <code>ROLLBACK</code> are allowed

Triggers are fully compiled when the `CREATE TRIGGER` command is issued and the P code is stored in the data dictionary.

If errors occur during the compilation of a trigger, the trigger is still created.

### Managing Triggers:

**Disable or reenable a database trigger:**

```
ALTER TRIGGER trigger_name DISABLE | ENABLE
```

**Disable or reenable all triggers for a table:**

```
ALTER TABLE table_name DISABLE | ENABLE ALL TRIGGERS
```

**Recompile a trigger for a table:**

```
ALTER TRIGGER trigger_name COMPILE
```

### **Trigger Modes: Enabled or Disabled**

- When a trigger is first created, it is enabled automatically.
- The Oracle server checks integrity constraints for enabled triggers and guarantees that triggers cannot compromise them. In addition, the Oracle server provides read-consistent views for queries and constraints, manages the dependencies, and provides a two-phase commit process if a trigger updates remote tables in a distributed database.
- Disable a specific trigger by using the `ALTER TRIGGER` syntax, or disable *all* triggers on a table by using the `ALTER TABLE` syntax.

- Disable a trigger to improve performance or to avoid data integrity checks when loading massive amounts of data by using utilities such as SQL\*Loader. You may also want to disable the trigger when it references a database object that is currently unavailable, owing to a failed network connection, disk crash, offline data file, or offline tablespace.

### Compile a Trigger

- Use the ALTER TRIGGER command to explicitly recompile a trigger that is invalid.
- When you issue an ALTER TRIGGER statement with the COMPILE option, the trigger recompiles, regardless of whether it is valid or invalid.

## DROP TRIGGER Syntax

To remove a trigger from the database, use the DROP

TRIGGER syntax:

DROP TRIGGER *trigger\_name*;

TRIGGER secure\_emp; **Example:**

Note: All triggers on a table are dropped when the table is dropped.

When a trigger is no longer required, you can use a SQL statement in iSQL\*Plus to drop it.

## Trigger Test Cases

- Test each triggering data operation, as well as non triggering data operations.
- Test each case of the WHEN clause.
- Cause the trigger to fire directly from a basic data operation, as well as indirectly from a procedure.
- Test the effect of the trigger upon other triggers.
- Test the effect of other triggers upon the trigger.

## Testing Triggers

- Ensure that the trigger works properly by testing a number of cases separately.
- Take advantage of the DBMS\_OUTPUT procedures to debug triggers. You can also use the Procedure Builder debugging tool to debug triggers.

### Trigger Execution Model and Constraint Checking

1. Execute all BEFORE STATEMENT triggers.
2. Loop for each row affected:
  - a. Execute all BEFORE ROW triggers.
  - b. Execute all AFTER ROW triggers.
3. Execute the DML statement and perform integrity constraint checking.
4. Execute all AFTER STATEMENT triggers.

### Trigger Execution Model

A single DML statement can potentially fire up to four types of triggers: BEFORE and AFTER statement and row triggers. A triggering event or a statement within the trigger can cause one or more integrity constraints to be checked. Triggers can also cause other triggers to fire (cascading triggers). All actions and checks done as a result of a SQL statement must succeed. If an exception is raised within a trigger and the exception is not explicitly handled, all actions performed because of the original SQL statement are rolled back. This includes actions performed by firing triggers. This guarantees that integrity constraints can never be compromised by triggers.

When a trigger fires, the tables referenced in the trigger action may undergo changes by other users transactions. In all cases, a read-consistent image is guaranteed for modified values the trigger needs to read (query) or write (update).

### Trigger Execution Model and Constraint Checking: Example

```
CREATE OR REPLACE TRIGGER constr_emp_trig
AFTER UPDATE ON employees
FOR EACH ROW BEGIN
INSERT INTO departments
VALUES (999, 'dept999', 140, 2400);

END;
/

UPDATE employees SET department_id = 999
WHERE employee_id = 170;
-- Successful after trigger is fired
```

The example in the slide explains a situation in which the integrity constraint can be taken care of by using a trigger. Table EMPLOYEES has a foreign key constraint on the DEPARTMENT\_ID column of the DEPARTMENTS table.

In the first SQL statement, the DEPARTMENT\_ID of the employee with EMPLOYEE\_ID 170 is modified to 999.

Because such a department does not exist in the DEPARTMENTS table, the statement raises the exception - 2292 for the integrity constraint violation.

A trigger CONSTR\_EMP\_TRIG is created that inserts a new department 999 into the DEPARTMENTS table.

When the UPDATE statement that modifies the department of employee 170 to 999 is issued, the trigger fires. Then, the foreign key constraint is checked. Because the trigger inserted the department 999 into the DEPARTMENTS table, the foreign key constraint check is successful and there is no exception.

This process works with Oracle8i and later releases. The example described in the slide produces a run-time error in releases prior to Oracle8i.

### After Row and After Statement Triggers

```
CREATE OR REPLACE TRIGGER audit_emp_trig
AFTER UPDATE or INSERT or DELETE on EMPLOYEES FOR EACH
ROW
```

```

BEGIN
IF DELETING THEN var_pack.set_g_del(1); ELSIF
INSERTING THEN var_pack.set_g_ins(1); ELSIF UPDATING
('SALARY')
THEN var_pack.set_g_up_sal(1); ELSE
var_pack.set_g_upd(1); END IF;
END audit_emp_trig;
/
CREATE OR REPLACE TRIGGER audit_emp_tab AFTER
UPDATE or INSERT or DELETE on employees BEGIN
audit_emp;
END audit_emp_tab;

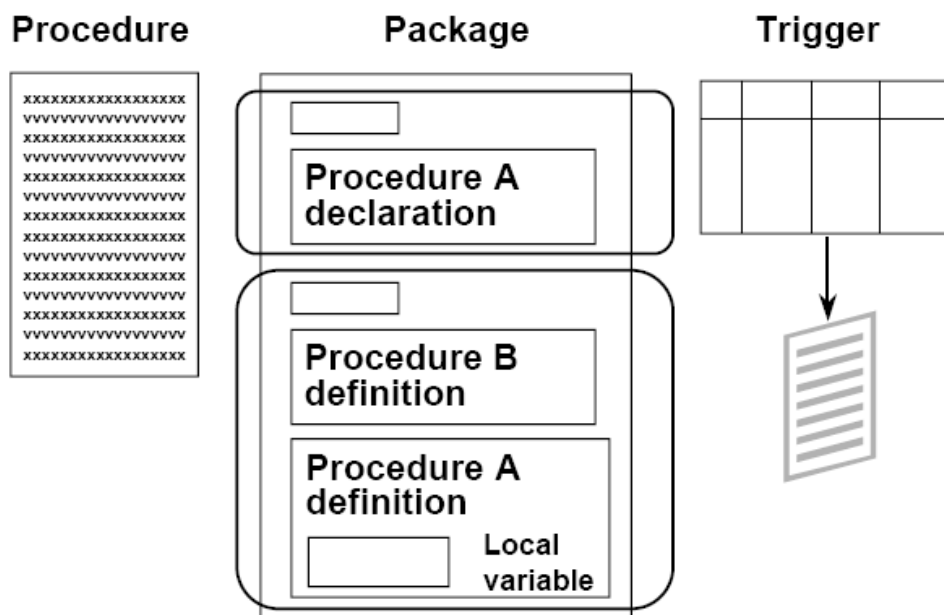
```

The trigger AUDIT\_EMP\_TRIG is a row trigger that fires after every row manipulated. This trigger invokes the package procedures depending on the type of DML performed. For example, if the DML updates salary of an employee, then the trigger invokes the procedure SET\_G\_UP\_SAL. This package procedure in turn invokes the function G\_UP\_SAL. This function increments the package variable GV\_UP\_SAL that keeps account of the number of rows being changed due to update of the salary.

The trigger AUDIT\_EMP\_TAB will fire after the statement has finished. This trigger invokes the procedure AUDIT\_EMP, which is on the following pages. The AUDIT\_EMP procedure updates the AUDIT\_TABLE table. An entry is made into the AUDIT\_TABLE table with the information such as the user who performed the DML, the table on which DML is performed, and the total number of such

data manipulations performed so far on the table (indicated by the value of the corresponding column in the AUDIT\_TABLE table). At the end, the AUDIT\_EMP procedure resets the package variables to 0.

## 12.10 Summary



Construct	Usage
Procedure	PL/SQL programming block that is stored in the database for repeated execution
Package	Group of related procedures, functions, variables, cursors, constants, and exceptions
Trigger	PL/SQL programming block that is executed implicitly by a data manipulation statement

## COLLECTIONS AND RECORDS

### Objectives

- Create user-defined PL/SQL records
- Create a record with the %ROWTYPE attribute
- Create an INDEX BY table
- Create an INDEX BY table of records
- Describe the difference between records, tables and tables of records

### Composite Data Types

Are of two types:

- PL/SQL RECORDs
- PL/SQL Collections
  - INDEX BY Table
- Nested Table
- VARRAY

Contain internal components

Are reusable

### RECORD and TABLE Data Types

Like scalar variables, composite variables have a data type. Composite data types (also known as collections) are RECORD, TABLE, NESTED TABLE, and VARRAY. You use the RECORD data type to treat related but dissimilar data as a logical unit. You use the TABLE data type to reference and manipulate collections of data as a whole object.

A record is a group of related data items stored as fields, each with its own name and data type. A table contains a column and a primary key to give you array-like access to rows. After they are defined, tables and records can be reused.

### PL/SQL Records

- Must contain one or more components of any scalar, RECORD, or INDEX BY table data type, called fields
  - Are similar in structure to records in a third generation language (3GL)
  - Are not the same as rows in a database table
  - Treat a collection of fields as a logical unit
  - Are convenient for fetching a row of data from a table for processing

## Creating a PL/SQL Record

### Syntax:

Where *field\_declaration* is: TYPE *type\_name* IS  
RECORD (*field\_declaration* [, *field\_declaration*]...);  
identifier *type\_name*;  
*field\_name* {*field\_type* | *variable%TYPE*  
| *table.column%TYPE* | *table%ROWTYPE*}  
[[NOT NULL] {:= | DEFAULT} *expr*]

## Defining and Declaring a PL/SQL Record

To create a record, you define a RECORD type and then declare records of that type. In the syntax:

*type\_name* is the name of the RECORD type. (This identifier is used to declare records.)

*field\_name* is the name of a field within the record.

*field\_type* is the data type of the field. (It represents any PL/SQL data type except REF CURSOR. You can use the %TYPE and %ROWTYPE attributes.)

*expr* is the *field\_type* or an initial value.

The NOT NULL constraint prevents assigning nulls to those fields. Be sure to initialize NOT NULL fields.

### Declare variables to store the name, job, and salary of a new employee. Example:

```
...  
TYPE emp_record_type IS RECORD (last_name  
VARCHAR2(25),  
job_id VARCHAR2(10), salary  
NUMBER(8,2));  
emp_record  
emp_record_type;  
...
```

Field declarations are like variable declarations. Each field has a unique name and a specific data type.

There are no predefined data types for PL/SQL records, as there are for scalar variables. Therefore, you must create the record type first and then declare an identifier using that type.

In the example on the slide, a EMP\_RECORD\_TYPE record type is defined to hold the values for the last\_name, job\_id, and salary. In the next step, a record EMP\_RECORD, of the type EMP\_RECORD\_TYPE is declared.

**Note:** You can add the NOT NULL constraint to any field declaration to prevent assigning nulls to that field. Remember, fields declared as NOT NULL must be initialized.

## PL/SQL Record Structure

Fields in a record are accessed by name. To reference or initialize an individual field, use dot notation and the following syntax:



record\_name.field\_name

For example, you reference the job\_id field in the emp\_record record as follows:

emp\_record.job\_id ...

You can then assign a value to the record field as follows:

emp\_record.job\_id := 'ST\_CLERK';

In a block or subprogram, user-defined records are instantiated when you enter the block or subprogram and cease to exist when you exit the block or subprogram.

## The %ROWTYPE Attribute

- Declare a variable according to a collection of columns in a database table or view.
- Prefix %ROWTYPE with the database table.
- Fields in the record take their names and data types from the columns of the table or view.

### Declaring Records with the %ROWTYPE Attribute

To declare a record based on a collection of columns in a database table or view, you use the %ROWTYPE attribute. The fields in the record take their names and data types from the columns of the table or view. The record can also store an entire row of data fetched from a cursor or cursor variable.

In the following example, a record is declared using %ROWTYPE as a data type specifier.

```
DECLARE
```

```
emp_record employees%ROWTYPE;
```

```
...
```

The emp\_record record will have a structure consisting of the following fields, each representing a column in the EMPLOYEES table.

### Syntax

```
DECLARE
```

```
identifier reference%ROWTYPE;
```

**where:** *identifier* is the name chosen for the record as a whole.

*reference* is the name of the table, view, cursor, or cursor variable on which the record is to be based. The table or view must exist for this reference to be valid.

To reference an individual field, you use dot notation and the following syntax:

record\_name.field\_name

For example, you reference the commission\_pct field in the emp\_record record as follows:

emp\_record.commission\_pct

You can then assign a value to the record field as follows:

emp\_record.commission\_pct:= .35;

### Assigning Values to Records

You can assign a list of common values to a record by using the SELECT or FETCH statement. Make sure that the column names appear in the same order as the fields in your record. You can also assign one record to another if they have the same data type. A user-defined record and a %ROWTYPE record *never* have the same data type.

### Advantages of Using %ROWTYPE

- The number and data types of the underlying database columns need not be known.
- The number and data types of the underlying database column may change at run time.
- The attribute is useful when retrieving a row with the SELECT \* statement.

### Examples:

Declare a variable to store the information about a department from the DEPARTMENTS table. Declare a variable to store the information about an employee from the EMPLOYEES table. dept\_record  
departments%ROWTYPE;

emp\_record employees%ROWTYPE;

The first declaration on the slide creates a record with the same field names and field data types as a row in the DEPARTMENTS table. The fields are DEPARTMENT\_ID, DEPARTMENT\_NAME,

MANAGER\_ID, and LOCATION\_ID. The second declaration creates a record with the same field names, field data types, and order as a row in the EMPLOYEES table. The fields are EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, PHONE\_NUMBER, HIRE\_DATE, JOB\_ID, SALARY, COMMISSION\_PCT, MANAGER\_ID, DEPARTMENT\_ID.

## INDEX BY Tables

- Are composed of two components:
  - Primary key of data type `BINARY_INTEGER`
  - Column of scalar or record data type
- Can increase in size dynamically because they are unconstrained

Objects of the `TABLE` type are called `INDEX BY` tables. They are modelled as (but not the same as) database tables. `INDEX BY` tables use a primary key to provide you with array-like access to rows.

A `INDEX BY` table:

- Is similar to an array
- Must contain two components:
  - A primary key of data type `BINARY_INTEGER` that indexes the `INDEX BY` table
  - A column of a scalar or record data type, which stores the `INDEX BY` table elements
- Can increase dynamically because it is unconstrained

## Creating a index by table

### Syntax:

```
TYPE type_name IS TABLE OF
{column_type | variable%TYPE
| table.column%TYPE} [NOT NULL]
| table.%ROWTYPE
[INDEX BY BINARY_INTEGER];
identifier type_name;
...
TYPE ename_table_type IS TABLE OF
employees.last_name%TYPE INDEX BY
BINARY_INTEGER; ename_table
ename_table_type;
```

There are two steps involved in creating a `INDEX BY` table.

1. Declare a `TABLE` data type.
2. Declare a variable of that data type.

In the syntax:

The `NOT NULL` constraint prevents nulls from being assigned to the PL/ SQL table of that type. Do not initialize the `INDEX BY` table.

INDEX-BY tables can have the following element types: BINARY\_INTEGER, BOOLEAN, LONG, LONG RAW, NATURAL, NATURALN, PLS\_INTEGER, POSITIVE, POSITIVEN, SIGNTYPE, and STRING. INDEX-BY tables are initially sparse. That enables you, for example, to store reference data in an INDEX-BY table using a numeric primary key as the index.

*type\_name* is the name of the TABLE type. (It is a type specifier used in

subsequent declarations of PL/SQL tables.)

*column\_type* is any scalar (scalar and composite) data type such as VARCHAR2, DATE, NUMBER or %TYPE. (You can use the %TYPE attribute to

provide the column datatype.)

*identifier* is the name of the identifier that represents an entire PL/SQL table.

## INDEX BY Table Structure

Like the size of a database table, the size of a INDEX BY table is unconstrained. That is, the number of rows in a INDEX BY table can increase dynamically, so that your INDEX BY table grows as new rows are added.

INDEX BY tables can have one column and a unique identifier to that one column, neither of which can be named. The column can belong to any scalar or record data type, but the primary key must belong to type BINARY\_INTEGER. You cannot initialize an INDEX BY table in its declaration. An INDEX BY table is not populated at the time of declaration. It contains no keys or no values. An

explicit executable statement is required to initialize (populate) the INDEX BY table.

```
DECLARE
TYPE ename_table_type IS TABLE OF
employees.last_name%TYPE INDEX BY
BINARY_INTEGER;
TYPE hiredate_table_type IS TABLE OF DATE INDEX BY
BINARY_INTEGER;
ename_table ename_table_type; hiredate_table
hiredate_table_type; BEGIN
ename_table(1) := 'CAMERON';
hiredate_table(8) := SYSDATE + 7; IF
ename_table.EXISTS(1) THEN INSERT INTO
...
... END;
```

## Referencing an INDEX BY Table Syntax:

INDEX\_BY\_table\_name(primary\_key\_value)

**where:** *primary\_key\_value* belongs to type BINARY\_INTEGER. Reference the third row in an INDEX BY table ENAME\_TABLE: ename\_table(3) ...

The magnitude range of a BINARY\_INTEGER is -2147483647 ... 2147483647, so the primary key value can be negative. Indexing does not need to start with 1.

**Note:** The *table*.EXISTS(*i*) statement returns TRUE if a row with index *i* is returned. Use the EXISTS statement to prevent an error that is raised in reference to a nonexisting table element.

## Using INDEX BY Table Methods

The following methods make INDEX BY tables easier to use:

- NEXT
- TRIM
- DELETE
- EXISTS
- COUNT
- FIRST and LAST
- PRIOR

A INDEX BY table method is a built-in procedure or function that operates on tables and is called using dot notation.

**Syntax:** *table\_name.method\_name*[ (*parameters*) ]

### Method Description

EXISTS(*n*) Returns TRUE if the *n*th element in a PL/SQL table exists

COUNT Returns the number of elements that a PL/SQL table currently contains

FIRST, LAST

Returns the first and last (smallest and largest) index numbers in a PL/SQL table. Returns NULL if the PL/SQL table is empty.

PRIOR(*n*) Returns the index number that precedes index *n* in a PL/SQL table NEXT(*n*)

Returns the index number that succeeds index *n* in a PL/SQL table TRIM removes one element from the end of a PL/SQL table.

TRIM(*n*) removes *n* elements from the end of a PL/SQL table. DELETE removes all elements from a PL/SQL table. DELETE(*n*) removes the *n*th element from a PL/SQL table.

DELETE(*m*, *n*) removes all elements in the range *m* ... *n* from a PL/SQL table.

### INDEX BY Table of Records

- Define a TABLE variable with a permitted PL/SQL data type.
- Declare a PL/SQL variable to hold department information.

#### Example:

DECLARE

TYPE dept\_table\_type IS TABLE OF

```
departments%ROWTYPE INDEX BY  
BINARY_INTEGER; dept_table  
dept_table_type;  
-- Each element of dept_table is a record
```

At a given point of time, a INDEX BY table can store only the details of any one of the columns of a database table. There is always a necessity to store all the columns retrieved by a query. The INDEX BY table of records offer a solution to this. Because only one table definition is needed to hold information about all of the fields of a database table, the table of records greatly increases the functionality of INDEX BY tables.

## Summary

In this chapter, you should have learned to:

- Define and reference PL/SQL variables of composite data types:
  - PL/SQL records
  - INDEX BY tables
  - INDEX BY table of records
- Define a PL/SQL record by using the %ROWTYPE attribute

A PL/SQL record is a collection of individual fields that represent a row in a table. By using records you can group the data into one structure and then manipulate this structure as one entity or logical unit. This helps reduce coding, and keeps the code easier to maintain and understand.

Like PL/SQL records, the table is another composite data type. INDEX BY tables are objects of a TABLE type and look similar to database tables but with a slight difference. INDEX BY tables use a primary key to give you array-like access to rows. The size of a INDEX BY table is unconstrained. INDEX BY tables can have one column and a primary key, neither of which can be named. The column can have any data type, but the primary key must be of the BINARY\_INTEGER type.

A INDEX BY table of records enhances the functionality of INDEX BY tables, because only one table definition is required to hold information about all the fields.

The following collection methods help generalize code, make collections easier to use, and make your applications easier to maintain: EXISTS, COUNT, LIMIT, FIRST and LAST, PRIOR and NEXT, TRIM , and DELETE

The %ROWTYPE is used to declare a compound variable whose type is the same as that of a row of a database table.

## BULK BINDING

### Objectives

- Bulk binding
- Tuning PL/SQL performance with bulk binding
- How do bulk binds improve performance
- Using the FOR ALL statement

### Tuning PL/SQL Performance with Bulk Binds

When SQL statements execute inside a loop using collection elements as bind variables, context switching between the PL/SQL and SQL engines can slow down execution. For example, the following UPDATE statement is sent to the SQL engine with each iteration of the FOR loop:

```
DECLARE
  TYPE NumList IS VARRAY(20) OF NUMBER;
  depts NumList := NumList(10, 30, 70, ...); -- department numbers
BEGIN
  ...
  FOR i IN depts.FIRST..depts.LAST LOOP
    ...
    UPDATE emp SET sal = sal * 1.10 WHERE deptno = depts(i); END
  LOOP;
END;
```

In such cases, if the SQL statement affects four or more database rows, the use of bulk binds can improve performance considerably. For example, the following

UPDATE statement is sent to the SQL engine just once, with the entire nested table:

```
FORALL i IN depts.FIRST..depts.LAST
  UPDATE emp SET sal = sal * 1.10 WHERE deptno = depts(i);
```

To maximize performance, rewrite your programs as follows:

If an INSERT, UPDATE, or DELETE statement executes inside a loop and references collection elements, move it into a FORALL statement.

If a SELECT INTO, FETCH INTO, or RETURNING INTO clause references a collection, incorporate the BULK COLLECT clause.

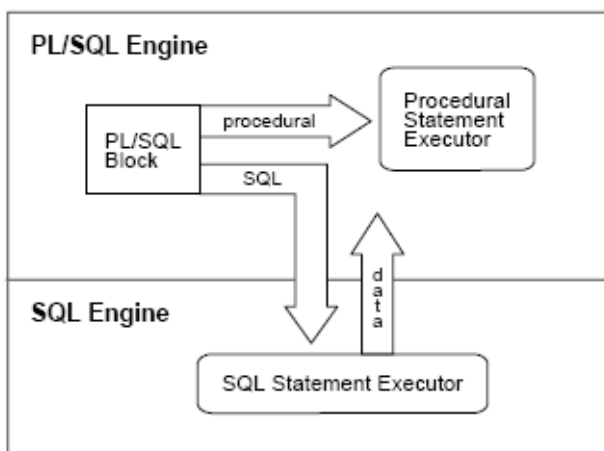
If possible, use host arrays to pass collections back and forth between your programs and the database server.

If the failure of a DML operation on a particular row is not a serious problem, include the keywords `SAVE EXCEPTIONS` in the `FORALL` statement and report or clean up the errors in a subsequent loop using the `%BULK_EXCEPTIONS` attribute.

These are not a trivial tasks. They require careful analysis of program control-flows and dependencies.

## Reducing Loop Overhead for Collections with Bulk Binds

As the figure below shows, the PL/SQL engine executes procedural statements but sends SQL statements to the SQL engine, which executes the SQL statements and, in some cases, returns data to the PL/SQL engine.



Too many context switches between the PL/SQL and SQL engines can harm performance. That can happen when a loop executes a separate SQL statement for each element of a collection, specifying the collection element as a bind variable. For example, the following `DELETE` statement is sent to the SQL engine with each iteration of the `FOR` loop:

```

DECLARE
  TYPE NumList IS VARRAY(20) OF NUMBER;
  depts NumList := NumList(10, 30, 70); -- department numbers
BEGIN
  ...
  FOR i IN depts.FIRST..depts.LAST LOOP DELETE FROM
  emp WHERE deptno = depts(i); END LOOP;
END;
  
```

In such cases, if the SQL statement affects four or more database rows, the use of bulk binds can improve performance considerably.



## How Do Bulk Binds Improve Performance?

The assigning of values to PL/SQL variables in SQL statements is called **binding**. PL/SQL binding operations fall into three categories:

- **in-bind** When a PL/SQL variable or host variable is stored in the database by an INSERT or UPDATE statement.
- **out-bind** When a database value is assigned to a PL/SQL variable or a host variable by the RETURNING clause of an INSERT, UPDATE, or DELETE statement.
- **define** When a database value is assigned to a PL/SQL variable or a host variable by a SELECT or FETCH statement.

A DML statement can transfer all the elements of a collection in a single operation, a process known as **bulk binding**. If the collection has 20 elements, bulk binding lets you perform the equivalent of 20 SELECT, INSERT, UPDATE, or DELETE statements using a single operation. This technique improves performance by minimizing the number of context switches between the PL/SQL and SQL engines. With bulk binds, entire collections, not just individual elements, are passed back and forth. To do bulk binds with INSERT, UPDATE, and DELETE statements, you enclose the SQL statement

within a PL/SQL FORALL statement. To do bulk binds with SELECT statements, you include the BULK COLLECT clause in the SELECT statement instead of using INTO.

### Example: Performing a Bulk Bind with DELETE

The following DELETE statement is sent to the SQL engine just once, even though it performs three DELETE operations:

```
DECLARE
TYPE NumList IS VARRAY(20) OF NUMBER;
depts NumList := NumList(10, 30, 70); -- department numbers
BEGIN
FORALL i IN depts.FIRST..depts.LAST
DELETE FROM emp WHERE deptno = depts(i); END;
```

### Example: Performing a Bulk Bind with INSERT

In the example below, 5000 part numbers and names are loaded into index-by tables. All table elements are inserted into a database table twice: first using a FOR loop, then using a FORALL statement. The FORALL version is much faster.

```
SQL> SET SERVEROUTPUT ON
SQL> CREATE TABLE parts (pnum NUMBER(4), pname CHAR(15)); Table
created.
SQL> GET test.sql
1 DECLARE
2 TYPE NumTab IS TABLE OF NUMBER(4) INDEX BY BINARY_INTEGER;
3 TYPE NameTab IS TABLE OF CHAR(15) INDEX BY BINARY_INTEGER;
4 pnums NumTab;
```

```

5 pnames NameTab;
6 t1 NUMBER(5);
7 t2 NUMBER(5);
8 t3 NUMBER(5);
9
10
11 BEGIN
12 FOR j IN 1..5000 LOOP -- load index-by tables
13 pnums(j) := j;
14 pnames(j) := 'Part No. ' || TO_CHAR(j);
15 END LOOP;
16 t1 := dbms_utility.get_time;
17 FOR i IN 1..5000 LOOP -- use FOR loop
18 INSERT INTO parts VALUES (pnums(i), pnames(i));
19 END LOOP;
20 t2 := dbms_utility.get_time;
21 FORALL i IN 1..5000 -- use FORALL statement
22 INSERT INTO parts VALUES (pnums(i), pnames(i));

23 get_time(t3);
24 dbms_output.put_line('Execution Time (secs)');
25 dbms_output.put_line('-----');
26 dbms_output.put_line('FOR loop: ' || TO_CHAR(t2 - t1));
27 dbms_output.put_line('FORALL: ' || TO_CHAR(t3 - t2));
28* END; SQL> /
Execution Time (secs)
----- FOR loop:
32
FORALL: 3
PL/SQL procedure successfully completed.

```

## Using the FORALL Statement

The keyword FORALL instructs the PL/SQL engine to bulk-bind input collections

before sending them to the SQL engine. Although the FORALL statement contains an iteration scheme, it is *not* a FOR loop. Its syntax follows:

```

FORALL index IN lower_bound..upper_bound
sql_statement;

```

The index can be referenced only within the FORALL statement and only as a collection subscript. The SQL statement must be an INSERT, UPDATE, or DELETE statement that references collection elements. And, the bounds must specify a valid range of consecutive index numbers. The SQL engine executes the SQL statement once for each index number in the range.

## Counting Rows Affected by FORALL Iterations with the %BULK\_ROWCOUNT Attribute

To process SQL data manipulation statements, the SQL engine opens an implicit cursor named SQL. This cursor's scalar attributes, %FOUND, %ISOPEN, %NOTFOUND, and %ROWCOUNT, return useful information about the most recently executed SQL data manipulation statement.

The SQL cursor has one composite attribute, %BULK\_ROWCOUNT, designed for use with the FORALL statement. This attribute has the semantics of an index-by table. Its *i*th element stores the number of rows processed by the *i*th execution of an INSERT, UPDATE or DELETE statement. If the *i*th execution affects no rows, %BULK\_ROWCOUNT(*i*) returns zero. An example follows:

```
DECLARE
  TYPE NumList IS TABLE OF NUMBER; depts
  NumList := NumList(10, 20, 50); BEGIN
  FORALL j IN depts.FIRST..depts.LAST
  UPDATE emp SET sal = sal * 1.10 WHERE deptno = depts(j);
  -- Did the 3rd UPDATE statement affect any rows? IF
  SQL%BULK_ROWCOUNT(3) = 0 THEN ... END;
```

The FORALL statement and %BULK\_ROWCOUNT attribute use the same subscripts. For example, if FORALL uses the range 5 .. 10, so does %BULK\_ROWCOUNT.

%BULK\_ROWCOUNT is usually equal to 1 for inserts, because a typical insert operation affects only a single row. But for the INSERT ... SELECT construct, %BULK\_ROWCOUNT might be greater than 1. For example, the FORALL statement below inserts an arbitrary number of rows for each iteration. After each iteration, %BULK\_ROWCOUNT returns the number of items inserted:

You can also use the scalar attributes %FOUND, %NOTFOUND, and %ROWCOUNT with bulk binds. For example, %ROWCOUNT returns the total number of rows processed by all executions of the SQL statement.

%FOUND and %NOTFOUND refer only to the last execution of the SQL statement.

However, you can use %BULK\_ROWCOUNT to infer their values for individual executions. For example, when %BULK\_ROWCOUNT(*i*) is zero, %FOUND and %NOTFOUND are FALSE and TRUE, respectively.

## Handling FORALL Exceptions with the %BULK\_EXCEPTIONS Attribute

PL/SQL provides a mechanism to handle exceptions raised during the execution of a FORALL statement. This mechanism enables a bulk-bind operation to save information about exceptions and continue processing.

To have a bulk bind complete despite errors, add the keywords SAVE EXCEPTIONS to your FORALL statement. The syntax follows:

```
FORALL index IN lower_bound..upper_bound SAVE EXCEPTIONS
```

*{insert\_stmt | update\_stmt | delete\_stmt}*

All exceptions raised during the execution are saved in the new cursor attribute `%BULK_EXCEPTIONS`, which stores a collection of records. Each record has two fields. The first field, `%BULK_EXCEPTIONS(i).ERROR_INDEX`, holds the "iteration" of the `FORALL` statement during which the exception was raised. The second field, `%BULK_EXCEPTIONS(i).ERROR_CODE`, holds the corresponding Oracle error code.

The values stored by `%BULK_EXCEPTIONS` always refer to the most recently executed `FORALL` statement. The number of exceptions is saved in the count attribute of `%BULK_EXCEPTIONS`, that is, `%BULK_EXCEPTIONS.COUNT`. Its subscripts range from 1 to `COUNT`.

If you omit the keywords `SAVE EXCEPTIONS`, execution of the `FORALL` statement stops when an exception is raised. In that case, `SQL%BULK_EXCEPTIONS.COUNT` returns 1, and `SQL%BULK_EXCEPTIONS` contains just one record. If no exception is raised during execution, `SQL%BULK_EXCEPTIONS.COUNT` returns 0.

The following example shows how useful the cursor attribute `%BULK_EXCEPTIONS` can be:

```
DECLARE
  TYPE NumList IS TABLE OF NUMBER;
  num_tab NumList := NumList(10,0,11,12,30,0,20,199,2,0,9,1);
  errors NUMBER;
  dml_errors EXCEPTION;
  PRAGMA exception_init(dml_errors, -24381); BEGIN
  FORALL i IN num_tab.FIRST..num_tab.LAST SAVE EXCEPTIONS DELETE
  FROM emp WHERE sal > 500000/num_tab(i); EXCEPTION
  WHEN dml_errors THEN
  errors := SQL%BULK_EXCEPTIONS.COUNT;
  dbms_output.put_line('Number of errors is ' || errors); FOR i IN
  1..errors LOOP
  dbms_output.put_line('Error ' || i || ' occurred during ' ||
  'iteration' || SQL%BULK_EXCEPTIONS(i).ERROR_INDEX);

  dbms_output.put_line('Oracle error is ' ||
  SQLERRM(-SQL%BULK_EXCEPTIONS(i).ERROR_CODE)); END
  LOOP;
  END;
```

In this example, PL/SQL raised the predefined exception `ZERO_DIVIDE` when `i` equalled 2, 6, 10. After the bulk-bind completed, `SQL%BULK_EXCEPTIONS.COUNT` returned 3, and the contents of `SQL%BULK_EXCEPTIONS` were (2,1476), (6,1476), and (10,1476). To get the Oracle error message (which includes the code), we negated the value of `SQL%BULK_EXCEPTIONS(i).ERROR_CODE` and passed the result to the error-reporting function `SQLERRM`, which expects a negative number.

Here is the output: Number of

errors is 3

Error 1 occurred during iteration 2

Oracle error is ORA-01476: divisor is equal to zero

Error 2 occurred during iteration 6

Oracle error is ORA-01476: divisor is equal to zero

Error 3 occurred during iteration 10

Oracle error is ORA-01476: divisor is equal to zero

## Retrieving Query Results into Collections with the BULK COLLECT Clause

The keywords BULK COLLECT tell the SQL engine to bulk-bind output collections before returning them to the PL/SQL engine. You can use these keywords in the SELECT INTO, FETCH INTO, and RETURNING INTO clauses.

Here is the syntax:

```
... BULK COLLECT INTO collection_name[, collection_name] ...
```

The SQL engine bulk-binds all collections referenced in the INTO list. The corresponding columns can store scalar or composite values including objects. In the following example, the SQL engine loads the entire empno and ename database columns into nested tables before returning the tables to the PL/SQL engine:

```
DECLARE  
TYPE NumTab IS TABLE OF emp.empno%TYPE; TYPE  
NameTab IS TABLE OF emp.ename%TYPE; enums NumTab; --  
no need to initialize  
names NameTab; BEGIN  
SELECT empno, ename BULK COLLECT INTO enums, names FROM emp;  
... END;
```

In the next example, the SQL engine loads all the values in an object column into a nested table before returning the table to the PL/SQL engine:

```
CREATE TYPE Coords AS OBJECT (x NUMBER, y NUMBER); CREATE  
TABLE grid (num NUMBER, loc Coords);  
INSERT INTO grid VALUES(10, Coords(1,2)); INSERT  
INTO grid VALUES(20, Coords(3,4)); DECLARE  
TYPE CoordsTab IS TABLE OF Coords;  
pairs CoordsTab; BEGIN  
SELECT loc BULK COLLECT INTO pairs FROM grid;  
-- now pairs contains (1,2) and (3,4) END;
```

The SQL engine initializes and extends collections for you. (However, it cannot extend varrays beyond their maximum size.) Then, starting at index 1, it inserts elements consecutively and overwrites any pre-existent elements.

The SQL engine bulk-binds entire database columns. So, if a table has 50,000 rows, the engine loads 50,000 column values into the target collection. However, you can use the pseudocolumn ROWNUM to limit the number of rows processed.

### **Examples of Bulk Fetching from a Cursor**

#### **Into One or More Collections**

You can bulk-fetch from a cursor into one or more collections:

```
DECLARE
  TYPE NameList IS TABLE OF emp.ename%TYPE; TYPE
  SalList IS TABLE OF emp.sal%TYPE;
  CURSOR c1 IS SELECT ename, sal FROM emp WHERE sal > 1000;
  names NameList; sals
  SalList; BEGIN
  OPEN c1;
  FETCH c1 BULK COLLECT INTO names, sals; END;
```

#### **Into a Collection of Records**

You can bulk-fetch from a cursor into a collection of records:

```
DECLARE
  TYPE DeptRecTab IS TABLE OF dept%ROWTYPE;
  dept_recs DeptRecTab; CURSOR
  c1 IS
  SELECT deptno, dname, loc FROM dept WHERE deptno > 10; BEGIN
  OPEN c1;
  FETCH c1 BULK COLLECT INTO dept_recs;

  END;
```

### **Limiting the Rows for a Bulk FETCH Operation with the LIMIT Clause**

The optional LIMIT clause, allowed only in bulk (not scalar) FETCH statements, lets you limit the number of rows fetched from the database. The syntax is

```
FETCH ... BULK COLLECT INTO ... [LIMIT rows];
```

where rows can be a literal, variable, or expression but must evaluate to a number.

Otherwise, PL/SQL raises the predefined exception VALUE\_ERROR. If the number is not positive, PL/SQL raises INVALID\_NUMBER. If necessary, PL/SQL rounds the number to the nearest integer.

**Retrieving DML Results into a Collection with the RETURNING INTO Clause**

You can use the BULK COLLECT clause in the RETURNING INTO clause of an INSERT, UPDATE, or DELETE statement, as the following example shows: *DECLARE*

```
TYPE NumList IS TABLE OF emp.empno%TYPE;
enums NumList; BEGIN
DELETE FROM emp WHERE deptno = 20
RETURNING empno BULK COLLECT INTO enums;
-- if there were five employees in department 20,
-- then enums contains five employee numbers
END;
```

## Restrictions on BULK COLLECT

The following restrictions apply to the BULK COLLECT clause:

You cannot bulk collect into an associative array that has a string type for the key.

You can use the BULK COLLECT clause only in server-side programs (not in client-side programs). Otherwise, you get the error *this feature is not supported in client-side programs*.

All targets in a BULK COLLECT INTO clause must be collections, as the following example shows:

```
DECLARE
TYPE NameList IS TABLE OF emp.ename%TYPE;
names NameList; salary
emp.sal%TYPE; BEGIN
SELECT ename, sal BULK COLLECT INTO names, salary -- illegal target
FROM emp WHERE ROWNUM < 50;
... END;
```

Composite targets (such as objects) cannot be used in the RETURNING INTO clause. Otherwise, you get the error *unsupported feature with RETURNING clause*.

When implicit datatype conversions are needed, multiple composite targets cannot be used in the BULK COLLECT INTO clause.

When an implicit datatype conversion is needed, a collection of a composite target (such as a collection of objects) cannot be used in the BULK COLLECT INTO clause.

## Using FORALL and BULK COLLECT Together

You can combine the BULK COLLECT clause with a FORALL statement, in which case, the SQL engine bulk-binds column values incrementally. In the following example, if collection `depts` has 3 elements, each of which causes 5 rows to be deleted, then collection `enums` has 15 elements when the statement completes:

```
FORALL j IN depts.FIRST..depts.LAST  
DELETE FROM emp WHERE empno = depts(j) RETURNING  
empno BULK COLLECT INTO enums;
```

The column values returned by each execution are added to the values returned previously. (With a FOR loop, the previous values are overwritten.)

You cannot use the SELECT ... BULK COLLECT statement in a FORALL statement.

Otherwise, you get the error *implementation restriction: cannot use FORALL and BULK COLLECT INTO together in SELECT statements*.

## Summary

In this chapter, we learned that Binding an entire collection at once is called bulk binding. Bulk binds improve performance by minimizing the number of context switches between the PL/SQL and SQL engines. With bulk binds, entire collections, not just individual elements, are passed back and forth.

The keyword FORALL instructs the PL/SQL engine to bulk bind input collections before sending them to the SQL engine. Although the FORALL statement contains an iteration scheme, it is not a FOR loop.

Tuning performance with bulk binds and reduce loop overhead for collections. Bulk collect and restrictions in using bulk collect.



## DBMS\_OUTPUT

**The DBMS\_OUTPUT package enables you to output messages from PL/SQL blocks. Available procedures include:**

- **PUT**
- **NEW\_LINE**
- **PUT\_LINE**
- **GET\_LINE**
- **GET\_LINES**
- **ENABLE/DISABLE**

The DBMS\_OUTPUT package outputs values and messages from any PL/SQL block.

Function or Procedure	Description
PUT	Appends text from the procedure to the current line of the line output buffer
NEW_LINE	Places an end_of_line marker in the output buffer
PUT_LINE	Combines the action of PUT and NEW_LINE
GET_LINE	Retrieves the current line from the output buffer into the procedure
GET_LINES	Retrieves an array of lines from the output buffer into the procedure
ENABLE/DISABLE	Enables or disables calls to the DBMS_OUTPUT procedures

### About the DBMS\_OUTPUT Package

Package DBMS\_OUTPUT enables you to display output from PL/SQL blocks and subprograms, which makes it easier to test and debug them. The procedure put\_line outputs information to a buffer in the SGA. You display the information by calling the procedure get\_line or by setting SERVEROUTPUT ON in SQL\*Plus.

For example, suppose you create the following stored procedure:

```
CREATE PROCEDURE calc_payroll (payroll OUT NUMBER) AS CURSOR  
c1 IS SELECT sal, comm FROM emp;  
BEGIN  
payroll := 0;
```

```
FOR c1rec IN c1 LOOP
c1rec.comm := NVL(c1rec.comm, 0);
payroll := payroll + c1rec.sal + c1rec.comm; END
LOOP;
/* Display debug info. */
dbms_output.put_line('Value of payroll: ' || TO_CHAR(payroll));
END;
```

When you issue the following commands, SQL\*Plus displays the value assigned by the procedure to parameter payroll:

```
SQL> SET SERVEROUTPUT ON SQL>
VARIABLE num NUMBER SQL> CALL
calc_payroll(:num); Value of payroll: 31225
```

## NATIVE DYNAMIC SQL

### Objectives

- What Is Dynamic SQL?
- The Need for Dynamic SQL
- Using the EXECUTE IMMEDIATE Statement
- Using the OPEN-FOR, FETCH, and CLOSE Statements
- Tips and Traps for Dynamic SQL

### What Is Dynamic SQL?

Most PL/SQL programs do a specific, predictable job. For example, a stored procedure might accept an employee number and salary increase, then update the sal column in the emp table. In this case, the full text of the UPDATE statement is known at compile time. Such statements do not change from execution to execution. So, they are called *static* SQL statements.

However, some programs must build and process a variety of SQL statements at run time. For example, a general-purpose report writer must build different SELECT statements for the various reports it generates. In this case, the full text of the statement is unknown until run time. Such statements can, and probably will, change from execution to execution. So, they are called *dynamic* SQL statements.

Dynamic SQL statements are stored in character strings built by your program at run time. Such strings must contain the text of a valid SQL statement or PL/SQL block. They can also contain placeholders for bind arguments. A *placeholder* is an undeclared identifier, so its name, to which you must prefix a colon, does not matter. For example, PL/SQL makes no distinction between the following strings:

```
'DELETE FROM emp WHERE sal > :my_sal AND comm < :my_comm'
```

```
'DELETE FROM emp WHERE sal > :s AND comm < :c'
```

To process most dynamic SQL statements, you use the EXECUTE IMMEDIATE statement. However, to process a multi-row query (SELECT statement), you must use the OPEN-FOR, FETCH, and CLOSE statements.

### The Need for Dynamic SQL

You need dynamic SQL in the following situations:

You want to execute a SQL data definition statement (such as CREATE), a data control statement (such as GRANT), or a session control statement (such as ALTER SESSION). In PL/SQL, such statements cannot be executed statically.

You want more flexibility. For example, you might want to defer your choice of schema objects until run time. Or, you might want your program to build different search conditions for the WHERE clause of a SELECT statement. A more complex program might choose from various SQL operations, clauses, etc.

You use package DBMS\_SQL to execute SQL statements dynamically, but you want better performance, something easier to use, or functionality that DBMS\_SQL lacks such as support for objects and collections.

## Using the EXECUTE IMMEDIATE Statement

The EXECUTE IMMEDIATE statement prepares (parses) and immediately executes a dynamic SQL statement or an anonymous PL/SQL block. The syntax is

```
EXECUTE IMMEDIATE dynamic_string
[INTO {define_variable[, define_variable]... | record}] [USING [IN
| OUT | IN OUT] bind_argument
|, [IN | OUT | IN OUT] bind_argument]...]
[{RETURNING | RETURN} INTO bind_argument[, bind_argument]...];
```

where *dynamic\_string* is a string expression that represents a SQL statement or PL/SQL block, *define\_variable* is a variable that stores a selected column value, and *record* is a user-defined or %ROWTYPE record that stores a selected row.

An input *bind\_argument* is an expression whose value is passed to the dynamic SQL statement or PL/SQL block. An output *bind\_argument* is a variable that stores a value returned by the dynamic SQL statement or PL/SQL block.

Except for multi-row queries, the dynamic string can contain any SQL statement (*without* the terminator) or any PL/SQL block (with the terminator). The string can also contain placeholders for bind arguments. However, you cannot use bind arguments to pass the names of schema objects to a dynamic SQL statement.

Used only for single-row queries, the INTO clause specifies the variables or record into which column values are retrieved. For each value retrieved by the query, there must be a corresponding, type-compatible variable or field in the INTO clause.

Used only for DML statements that have a RETURNING clause (without a BULK COLLECT clause), the RETURNING INTO clause specifies the variables into which column values are returned. For each value returned by the DML statement, there must be a corresponding, type-compatible variable in the RETURNING INTO clause.

You can place all bind arguments in the USING clause. The default parameter mode is IN. For DML statements that have a RETURNING clause, you can place OUT arguments in the RETURNING INTO clause without specifying the parameter mode, which, by definition, is OUT. If you use both the USING clause and the RETURNING INTO clause, the USING clause can contain only IN arguments.

At run time, bind arguments replace corresponding placeholders in the dynamic string. So, every placeholder must be associated with a bind argument in the USING clause and/or RETURNING INTO clause. You can use numeric, character, and string literals as bind arguments, but you cannot use Boolean literals (TRUE, FALSE, and NULL).

Dynamic SQL supports all the SQL datatypes. So, for example, define variables and bind arguments can be collections, LOBs, instances of an object type, and refs. As a rule, dynamic SQL does not support PL/SQL-specific types. So, for example, define variables and bind arguments cannot be Booleans or index-by tables. The only exception is that a PL/SQL record can appear in the INTO clause.

You can execute a dynamic SQL statement repeatedly using new values for the bind arguments. However, you incur some overhead because EXECUTE IMMEDIATE re-prepares the dynamic string before every execution.

### Some Examples of Dynamic SQL

The following PL/SQL block contains several examples of dynamic SQL:

```
DECLARE
```

```

sql_stmt VARCHAR2(200); plsql_block
VARCHAR2(500); emp_id NUMBER(4) :=
7566; salary NUMBER(7,2);

dept_id NUMBER(2) := 50;
dept_name VARCHAR2(14) := 'PERSONNEL';
location VARCHAR2(13) := 'DALLAS';
emp_rec emp%ROWTYPE; BEGIN
EXECUTE IMMEDIATE 'CREATE TABLE bonus (id NUMBER, amt NUMBER)';
sql_stmt := 'INSERT INTO dept VALUES (:1, :2, :3)';
EXECUTE IMMEDIATE sql_stmt USING dept_id, dept_name, location;
sql_stmt := 'SELECT * FROM emp WHERE empno = :id'; EXECUTE
IMMEDIATE sql_stmt INTO emp_rec USING emp_id; plsql_block := 'BEGIN
emp_pkg.raise_salary(:id, :amt); END;'; EXECUTE IMMEDIATE plsql_block
USING 7788, 500;
sql_stmt := 'UPDATE emp SET sal = 2000 WHERE empno = :1
RETURNING sal INTO :2';
EXECUTE IMMEDIATE sql_stmt USING emp_id RETURNING INTO salary; EXECUTE
IMMEDIATE 'DELETE FROM dept WHERE deptno = :num' USING dept_id;
EXECUTE IMMEDIATE 'ALTER SESSION SET SQL_TRACE TRUE'; END;

```

In the example below, a standalone procedure accepts the name of a database table (such as 'emp') and an optional WHERE-clause condition (such as 'sal > 2000').

If you omit the condition, the procedure deletes all rows from the table. Otherwise, the procedure deletes only those rows that meet the condition.

```

CREATE PROCEDURE delete_rows (
table_name IN VARCHAR2,
condition IN VARCHAR2 DEFAULT NULL) AS where_clause
VARCHAR2(100) := ' WHERE ' || condition; BEGIN
IF condition IS NULL THEN where_clause := NULL; END IF;
EXECUTE IMMEDIATE 'DELETE FROM ' || table_name || where_clause;
EXCEPTION
... END;

```

## Backward Compatibility of the USING Clause

When a dynamic INSERT, UPDATE, or DELETE statement has a RETURNING clause, output bind arguments can go in the RETURNING INTO clause or the USING clause. In new applications, use the RETURNING INTO clause. In old applications, you can continue to use the USING clause. For example, both of the following EXECUTE IMMEDIATE statements are allowed:

```
DECLARE
sql_stmt VARCHAR2(200); my_empno
NUMBER(4) := 7902; my_ename
VARCHAR2(10);

my_job VARCHAR2(9);
my_sal NUMBER(7,2) := 3250.00; BEGIN
sql_stmt := 'UPDATE emp SET sal = :1 WHERE empno = :2
RETURNING ename, job INTO :3, :4';
/* Bind returned values through USING clause. */ EXECUTE
IMMEDIATE sql_stmt
USING my_sal, my_empno, OUT my_ename, OUT my_job;
/* Bind returned values through RETURNING INTO clause. */ EXECUTE
IMMEDIATE sql_stmt
USING my_sal, my_empno RETURNING INTO my_ename, my_job;
... END;
```

## Specifying Parameter Modes

With the USING clause, you need not specify a parameter mode for input bind arguments because the mode defaults to IN. With the RETURNING INTO clause, you cannot specify a parameter mode for output bind arguments because, by definition, the mode is OUT. An example follows:

```
DECLARE
sql_stmt VARCHAR2(200); dept_id
NUMBER(2) := 30; old_loc
VARCHAR2(13); BEGIN
sql_stmt :=
'DELETE FROM dept WHERE deptno = :1 RETURNING loc INTO :2'; EXECUTE
IMMEDIATE sql_stmt USING dept_id RETURNING INTO old_loc;
... END;
```

To call the procedure from a dynamic PL/SQL block, you must specify the IN OUT mode for the bind argument associated with formal parameter deptno, as follows:

```

DECLARE
plsql_block VARCHAR2(500);
new_deptno NUMBER(2);
new_dname VARCHAR2(14) := 'ADVERTISING';
new_loc VARCHAR2(13) := 'NEW YORK'; BEGIN
plsql_block := 'BEGIN create_dept(:a, :b, :c); END;'; EXECUTE
IMMEDIATE plsql_block
USING IN OUT new_deptno, new_dname, new_loc; IF
new_deptno > 90 THEN ...
END;

```

## Using the OPEN-FOR Statements

You use three statements to process a dynamic multi-row query: OPEN-FOR, FETCH, and CLOSE. First, you OPEN a cursor variable FOR a multi-row query. Then, you FETCH rows from the result set one at a time. When all the rows are processed, you CLOSE the cursor variable.

### Opening the Cursor Variable

The OPEN-FOR statement associates a cursor variable with a multi-row query, executes the query, identifies the result set, positions the cursor on the first row in the result set, then zeroes the rows- processed count kept by %ROWCOUNT.

Unlike the static form of OPEN-FOR, the dynamic form has an optional USING clause. At run time, bind arguments in the USING clause replace corresponding placeholders in the dynamic SELECT statement. The syntax is

```

OPEN {cursor_variable | :host_cursor_variable} FOR dynamic_string
[USING bind_argument[, bind_argument]...];

```

where cursor\_variable is a weakly typed cursor variable (one without a return type), host\_cursor\_variable is a cursor variable declared in a PL/SQL host environment such as an OCI program, and dynamic\_string is a string expression that represents a multi-row query.

Any bind arguments in the query are evaluated only when the cursor variable is opened. So, to fetch from the cursor using different bind values, you must reopen the cursor variable with the bind arguments set to their new values.

### Examples of Dynamic SQL for Records, Objects, and Collections

As the following example shows, you can fetch rows from the result set of a dynamic multi-row query into a record:

```

DECLARE
TYPE EmpCurTyp IS REF CURSOR;
emp_cv EmpCurTyp; emp_rec
emp%ROWTYPE; sql_stmt
VARCHAR2(200);
my_job VARCHAR2(15) := 'CLERK'; BEGIN

```

```

sql_stmt := 'SELECT * FROM emp WHERE job = :j'; OPEN emp_cv
FOR sql_stmt USING my_job;
LOOP
FETCH emp_cv INTO emp_rec; EXIT WHEN
emp_cv%NOTFOUND;
-- process record END LOOP;
CLOSE emp_cv; END;

```

The next example illustrates the use of objects and collections. Suppose you define object type **Person** and **VARRAY** type **Hobbies**, as follows:

```

CREATE TYPE Person AS OBJECT (name VARCHAR2(25), age NUMBER);
CREATE TYPE Hobbies IS VARRAY(10) OF VARCHAR2(25);

```

Now, using dynamic SQL, you can write a package of procedures that uses these types, as follows:

```

CREATE PACKAGE teams AS
    PROCEDURE create_table (tab_name VARCHAR2);
    PROCEDURE insert_row (tab_name VARCHAR2, p Person, h Hobbies);
    PROCEDURE print_table (tab_name VARCHAR2);
END;

CREATE PACKAGE BODY teams AS
    PROCEDURE create_table (tab_name VARCHAR2) IS
BEGIN
    EXECUTE IMMEDIATE 'CREATE TABLE ' || tab_name || ' (pers Person, hobbs
Hobbies)';
END;

    PROCEDURE insert_row (
        tab_name VARCHAR2, p Person, h Hobbies) IS
BEGIN
    EXECUTE IMMEDIATE 'INSERT INTO ' || tab_name || ' VALUES (:1, :2)' USING p, h;
END;

    PROCEDURE print_table (tab_name VARCHAR2) IS TYPE RefCurTyp IS REF CURSOR;
        cv RefCurTyp;
        p Person;
        h Hobbies;
BEGIN
    OPEN cv FOR 'SELECT pers, hobbs FROM ' || tab_name;
    LOOP
        FETCH cv INTO p, h;
        EXIT WHEN cv%NOTFOUND;
        -- print attributes of 'p' and elements of 'h'
    END LOOP;
    CLOSE cv;
END

```



From an anonymous PL/SQL block, you might call the procedures in package teams, as follows:

```
DECLARE
team_name VARCHAR2(15);
...
BEGIN
...
team_name := 'Notables';
teams.create_table(team_name);
teams.insert_row(team_name, Person('John',
31), Hobbies('skiing', 'coin collecting',
'tennis')); teams.insert_row(team_name,
Person('Mary', 28), Hobbies('golf',
'quilting', 'rock climbing'));
teams.print_table(team_name);
END;
```

## Tips and Traps for Dynamic SQL

This section shows you how to take full advantage of dynamic SQL and how to avoid some common pitfalls.

### Improving Performance

In the example below, Oracle opens a different cursor for each distinct value of emp\_id. This can lead to resource contention and poor performance.

```
CREATE PROCEDURE fire_employee (emp_id NUMBER) AS BEGIN
EXECUTE IMMEDIATE
'DELETE FROM emp WHERE empno = ' || TO_CHAR(emp_id); END;
```

You can improve performance by using a bind variable, as shown below. This allows Oracle to reuse the same cursor for different values of emp\_id.

```
CREATE PROCEDURE fire_employee (emp_id NUMBER) AS BEGIN
EXECUTE IMMEDIATE
'DELETE FROM emp WHERE empno = :num' USING emp_id; END;
```

### Making Procedures Work on Arbitrarily Named Schema Objects

Suppose you need a procedure that accepts the name of any database table, then drops that table from your schema. Using dynamic SQL, you might write the following standalone procedure:

```
CREATE PROCEDURE drop_table (table_name IN VARCHAR2) AS BEGIN
EXECUTE IMMEDIATE 'DROP TABLE :tab' USING table_name; END;
```

However, at run time, this procedure fails with an *invalid table name* error. That is because you cannot use bind arguments to pass the names of schema objects to a dynamic SQL statement.

Instead, you must embed parameters in the dynamic string, then pass the names of schema objects to those parameters.

To debug the last example, you must revise the EXECUTE IMMEDIATE statement. Instead of using a placeholder and bind argument, you embed parameter table\_name in the dynamic string, as follows:

```
CREATE PROCEDURE drop_table (table_name IN VARCHAR2) AS BEGIN
EXECUTE IMMEDIATE 'DROP TABLE ' || table_name; END;
```

Now, you can pass the name of any database table to the dynamic SQL statement.

### Using Duplicate Placeholders

Placeholders in a dynamic SQL statement are associated with bind arguments in the USING clause by position, not by name. So, if the same placeholder appears two or more times in the SQL statement, each appearance must correspond to a bind argument in the USING clause. For example, given the dynamic string

sql\_stmt := 'INSERT INTO payroll VALUES (:x, :x, :y, :x)'; you might

code the corresponding USING clause as follows: *EXECUTE*

```
IMMEDIATE sql_stmt USING a, a, b, a;
```

However, only the unique placeholders in a dynamic PL/SQL block are associated with bind arguments in the USING clause by position. So, if the same placeholder appears two or more times in a PL/SQL block, all appearances correspond to one bind argument in the USING clause. In the example below, the first unique placeholder (x) is associated with the first bind argument (a). Likewise, the second unique placeholder (y) is associated with the second bind argument (b).

### Using Cursor Attributes

Every explicit cursor has four attributes: %FOUND, %ISOPEN, %NOTFOUND, and %ROWCOUNT. When appended to the cursor name, they return useful information about the execution of static and dynamic SQL statements.

To process SQL data manipulation statements, Oracle opens an implicit cursor named SQL. Its attributes return information about the most recently executed INSERT, UPDATE, DELETE, or single-row SELECT statement. For example, the following standalone function uses %ROWCOUNT to return the number of rows deleted from a database table:

```
CREATE FUNCTION rows_deleted (
table_name IN VARCHAR2,
condition IN VARCHAR2) RETURN INTEGER AS BEGIN
EXECUTE IMMEDIATE
```

```
'DELETE FROM ' || table_name || ' WHERE ' || condition; RETURN  
SQL%ROWCOUNT; -- return number of rows deleted END;
```

Likewise, when appended to a cursor variable name, the cursor attributes return information about the execution of a multi-row query

### Passing Nulls

Suppose you want to pass nulls to a dynamic SQL statement. For example, you might write the following EXECUTE IMMEDIATE statement:

```
EXECUTE IMMEDIATE 'UPDATE emp SET comm = :x' USING NULL;
```

However, this statement fails with a *bad expression* error because the literal NULL is not allowed in the USING clause. To work around this restriction, simply replace the keyword NULL with an uninitialized variable, as follows:

```
DECLARE  
a_null CHAR(1); -- set to NULL automatically at run time  
BEGIN  
EXECUTE IMMEDIATE 'UPDATE emp SET comm = :x' USING a_null; END;
```

### Avoiding Deadlocks

In a few situations, executing a SQL data definition statement results in a deadlock. For example, the procedure below causes a deadlock because it attempts to drop itself. To avoid deadlocks, never try to ALTER or DROP a subprogram or package while you are still using it.

```
CREATE PROCEDURE calc_bonus (emp_id NUMBER) AS BEGIN  
...  
EXECUTE IMMEDIATE 'DROP PROCEDURE calc_bonus';
```

## Summary

In this chapter, you learned how to use native dynamic SQL (dynamic SQL for short), a PL/SQL interface that makes your applications more flexible and versatile. You learned simple ways to write programs that can build and process SQL statements "on the fly" at run time.

Within PL/SQL, you can execute any kind of SQL statement (even data definition and data control statements) without resorting to cumbersome programmatic approaches. Dynamic SQL blends seamlessly into your programs, making them more efficient, readable, and concise.

## UTL\_FILES

### What Is the UTL\_FILE Package?

The UTL\_FILE package provides text file I/O from within PL/SQL. Client-side security implementation uses normal operating system file permission checking. Server-side security is implemented through restrictions on the directories that can be accessed. In the init.ora file, the initialization parameter UTL\_FILE\_DIR is set to the accessible directories desired.

**UTL\_FILE\_DIR = *directory\_name***

For example, the following initialization setting indicates that the directory /usr/ngreenbe/my\_app is accessible to the fopen function, assuming that the directory is accessible to the database server processes. This parameter setting is case-sensitive on casesensitive operating systems.

**UTL\_FILE\_DIR = /user/ngreenbe/my\_app**

The directory should be on the same machine as the database server. Using the following setting turns off database permissions and makes all directories that are accessible to the database server processes also accessible to the UTL\_FILE package.

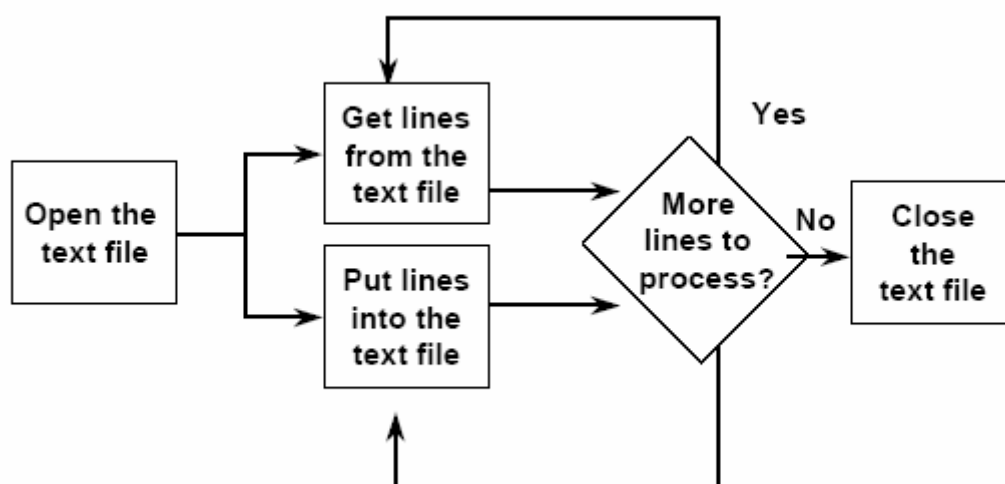
**UTL\_FILE\_DIR = \*** **Note:**

*UTL\_FILE\_DIR = \* is not recommended. Your init.ora file needs to include the parameter UTL\_FILE\_DIR. If it does not, edit the init.ora file and add the parameter. Then restart the database by using the SHUTDOWN and STARTUP commands.*

Using the procedures and functions in the package, you can open files, get text from files, put text into files, and close files. There are seven exceptions declared in the package to account for possible errors raised during execution.

### File Processing

Before using the UTL\_FILE package to read from or write to a text file, you must first check whether the text file is open by using the IS\_OPEN function. If the file is not open, you open the file with the FOPEN function. You then either read the file or write to the file until processing is done. At the end of file processing, use the FCLOSE procedure to close the file.



**Figure :** File Processing Using the UTL\_FILE Package

## The UTL\_FILE Package: Procedures and Functions

The UTL\_FILE package comes with several procedures and functions for handling text files within PL/SQL.

Function/ Procedure	Description
<b>FOPEN</b> function	Opens a file for input or output with the default line size.
<b>IS_OPEN</b> function	Determines if a file handle refers to an open file.
<b>FCLOSE</b> procedure	Closes a file.
<b>FCLOSE_ALL</b> procedure	Closes all open file handles.
<b>GET_LINE</b> procedure	Reads a line of text from an open file.
<b>PUT</b> procedure	Writes a line to a file. This does not append a line terminator.
<b>NEW_LINE</b> procedure	Writes one or more OS-specific line terminators to a file.
<b>PUT_LINE</b> procedure	Writes a line to a file. This appends an OS-specific line terminator.
<b>PUTF</b> procedure	A PUT procedure with formatting.

<b>FFLUSH</b> procedure	Physically writes all pending output to a file.
<b>FOPEN</b> function	Opens a file with the maximum line size specified.

### The UTL\_FILE Exceptions

The UTL\_FILE package declares seven exceptions that are raised to indicate an error condition in the operating system file processing.

Exception Name	Description
INVALID_PATH	File location or filename was invalid.
INVALID_MODE	The open_mode parameter in FOPEN was invalid.
INVALID_FILEHANDLE	File handle was invalid.
INVALID_OPERATION	File could not be opened or operated on as requested.
READ_ERROR	Operating system error occurred during the read operation.
WRITE_ERROR	Operating system error occurred during the write operation.
INTERNAL_ERROR	Unspecified PL/SQL error.

In addition to these package exceptions, procedures in UTL\_FILE can also raise predefined PL/SQL exceptions such as NO\_DATA\_FOUND or VALUE\_ERROR.

### Procedures and Functions: FOPEN

#### function

This function opens a file for input or output. The file location must be an accessible directory, as defined in the instance's initialization parameter UTL\_FILE\_DIR. The complete directory path must already exist; it is not created by FOPEN.

FOPEN returns a file handle, which must be used in all subsequent I/O operations on the file. You can have a maximum of 50 files open simultaneously.

#### Syntax

```
UTL_FILE.FOPEN (
```

```

location          IN VARCHAR2, filename
                  IN VARCHAR2, open_mode
IN VARCHAR2) RETURN
UTL_FILE.FILE_TYPE;

```

### Parameters

Parameters	Description
location	Operating system-specific string that specifies the directory in which to open the file.
filename	Name of the file, including extension (file type), without any directory path information. (Under the UNIX operating system, the filename cannot be terminated with a '/').
open_mode	String that specifies how the file is to be opened (either upper or lower case letters can be used).  The supported values, and the UTL_FILE procedures that can be used with them are:  'r' read text (GET_LINE)  'w' write text (PUT, PUT_LINE, NEW_LINE, PUTF, FFLUSH)  'a' append text (PUT, PUT_LINE, NEW_LINE, PUTF, FFLUSH)

*Note:* If you open a file that does not exist using the 'a' value for open\_mode, then the file is created in write ('w') mode.

### Returns

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

### Exceptions

```

INVALID_PATH
INVALID_MODE
INVALID_OPERATION

```

### IS\_OPEN function

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

### Syntax

```

UTL_FILE.IS_OPEN ( file IN
                  FILE_TYPE) RETURN BOOLEAN;

```

### Parameters

Parameter	Description
file	Active file handle returned by an FOPEN call.

**Returns**

TRUE or FALSE

**Exceptions**

None.

**FCLOSE procedure**

This procedure closes an open file identified by a file handle. If there is buffered data yet to be written when FCLOSE runs, then you may receive a WRITE\_ERROR exception when closing a file.

**Syntax**

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

**Parameters**

Parameter	Description
file	Active file handle returned by an FOPEN call.

**Exceptions** WRITE\_ERROR

INVALID\_FILEHANDLE

**FCLOSE\_ALL procedure**

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

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

**Syntax**

```
UTL_FILE.FCLOSE_ALL;
```

**Parameter** None.**Exceptions** WRITE\_ERROR



## GET\_LINE procedure

This procedure reads a line of 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.

If the line does not fit in the buffer, then a VALUE\_ERROR exception is raised. If no text was read due to "end of file," then the NO\_DATA\_FOUND exception is raised.

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

The maximum size of an input record is 1023 bytes, unless you specify a larger size in the overloaded version of FOPEN.

### Syntax

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

### Parameters

Parameters	Description
file	Active file handle returned by an FOPEN call.  The file must be open for reading (mode 'r'), otherwise an INVALID_OPERATION exception is raised.
buffer	Data buffer to receive the line read from the file.

### Exceptions

INVALID\_FILEHANDLE  
INVALID\_OPERATION  
READ\_ERROR NO\_DATA\_FOUND  
VALUE\_ERROR

## PUT procedure

PUT writes the text string stored in the buffer parameter to the open file identified by the file handle. The file must be open for write operations. No line terminator is appended by PUT; use NEW\_LINE to terminate the line or use PUT\_LINE to write a complete line with a line terminator.

The maximum size of an input record is 1023 bytes, unless you specify a larger size in the overloaded version of FOPEN.

### Syntax

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

### Parameters

Parameters	Description
file	Active file handle returned by an FOPEN call.
buffer	Buffer that contains the text to be written to the file.  You must have opened the file using mode 'w' or mode 'a'; otherwise, an INVALID_OPERATION exception is raised.

**Exceptions** INVALID\_FILEHANDLE

INVALID\_OPERATION

WRITE\_ERROR

### NEW\_LINE procedure

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

#### Syntax

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

#### Parameters

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

**Exceptions** INVALID\_FILEHANDLE

INVALID\_OPERATION

WRITE\_ERROR

### PUT\_LINE procedure

This procedure writes the text string stored in the buffer parameter to the open file identified by the file handle. The file must be open for write operations. PUT\_LINE terminates the line with the platform-specific line terminator character or characters.

The maximum size for an output record is 1023 bytes, unless you specify a larger value using the overloaded version of FOPEN.

#### Syntax

```
UTL_FILE.PUT_LINE (
```

```

file          IN FILE_TYPE, buffer
              IN VARCHAR2);

```

**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.

**Exceptions** INVALID\_FILEHANDLE

INVALID\_OPERATION

WRITE\_ERROR

**PUTF procedure**

This procedure is a formatted PUT procedure. It works like a limited printf(). The format string can contain any text, but the character sequences '%s' and '\n' have special 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.

**Syntax**

```

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

```

**Parameters**

Parameters	Description
file	Active file handle returned by an FOPEN call.
format	Format string that can contain text as well as the formatting characters '\n' and '%s'.

arg1..arg5	<p>From one to five operational argument strings.</p> <p>Argument strings are substituted, in order, for the '%s' formatters in the format string.</p> <p>If there are more formatters in the format parameter string than there are arguments, then an empty string is substituted for each '%s' for which there is no</p>
------------	---

argument.

**Exceptions** INVALID\_FILEHANDLE

INVALID\_OPERATION

WRITE\_ERROR

### FFLUSH procedure

FFLUSH physically writes all pending data to the file identified by the file handle. Normally, data being written to a file is buffered. The FFLUSH procedure forces any buffered data to be written to the file.

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 FILE_TYPE);
    invalid_maxlinesize      EXCEPTION;
```

#### Parameters

Parameters	Description
file	Active file handle returned by an FOPEN call.

**Exceptions** INVALID\_FILEHANDLE

INVALID\_OPERATION

WRITE\_ERROR

### FOPEN function

This function opens a file. You can have a maximum of 50 files open simultaneously.

#### Syntax

```
UTL_FILE.FOPEN (
    location          IN VARCHAR2, filename
                      IN VARCHAR2, open_mode
```

```
        IN VARCHAR2, max_linesize  
    IN BINARY_INTEGER) RETURN file_type;
```

**Parameters**

Parameter	Description
location	Directory location of file.
filename	File name (including extension).

open_mode	Open mode ('r', 'w', 'a').
max_linesize	Maximum number of characters per line, including the newline character, for this file. (minimum value 1, maximum value 32767).

**Returns**

Return	Description
file_type	Handle to open file.

**Exceptions**

INVALID\_PATH: File location or name was invalid. INVALID\_MODE: The open\_mode string was invalid. INVALID\_OPERATION: File could not be opened as requested.

INVALID\_MAXLINESIZE: Specified max\_linesize is too large or too small.

## SQL\*LOADER

### SQL\*Loader Basics

SQL\*Loader loads data from external files into tables of an Oracle database.

SQL\*Loader:

- Has a powerful data parsing engine which puts little limitation on the format of the data in the datafile.

- Can load data from multiple datafiles during the same load session. Can load data into multiple tables during the same load session.

- Is character set aware (you can specify the character set of the data).

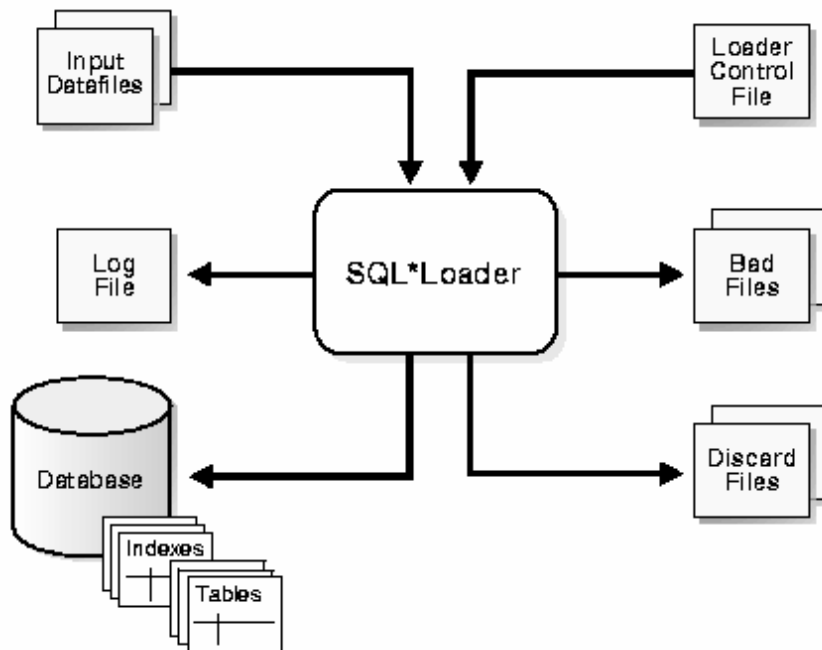
- Can selectively load data (you can load records based on the records' values). Can manipulate the data before loading it, using SQL functions.

- Can generate unique sequential key values in specified columns. Can use the operating system's file system to access the datafile(s). Can load data from disk, tape, or named pipe.

- Does sophisticated error reporting which greatly aids troubleshooting.

- Supports two loading "paths" -- Conventional and Direct. While conventional path loading is very flexibility, direct path loading provides superior loading performance.

- Can load arbitrarily complex object-relational data.



**Figure : SQL\*Loader Overview**

SQL\*Loader takes as input a control file, which controls the behavior of SQL\*Loader, and one or more datafiles. Output of the SQL\*Loader is an Oracle database (where the data is loaded), a log file, a bad file, and potentially a discard file.

### **SQL\*Loader Control File**

The control file is a text file written in a language that SQL\*Loader understands. The control file describes the task that the SQL\*Loader is to carry out. The control file tells SQL\*Loader where to find the data, how to parse and interpret the data, where to insert the data.

### **Input Data and Datafiles**

The other input to SQL\*Loader, other than the control file, is the data. SQL\*Loader reads data from one or more files (or operating system equivalents of files) specified in the control file. From SQL\*Loader's perspective, the data in the datafile is organized as records. A particular datafile can be in fixed record format, variable record format, or stream record format.

Important: If data is specified inside the control file (that is, INFILE \* was specified in the control file), then the data is interpreted in the stream record format with the default record terminator.

### **Fixed Record Format**

When all the records in a datafile are of the same byte length, the file is in fixed record format. Although this format is the least flexible, it does result in better performance than variable or stream format. Fixed format is also simple to specify, for example:

```
INFILE <datafile_name> "fix n"
```

specifies that SQL\*Loader should interpret the particular datafile as being in fixed record format where every record is n bytes long.

**Example : Loading Data in Fixed Record Format**

```
load data
infile 'example.dat'      "fix 11" into
table example
fields terminated by ',' optionally enclosed by '"' (col1
char(5),
col2 char(7))
```

example.dat:

```
001,      cd, 0002, fghi,
00003, lmn,
1, "pqrs",
0005, uvwx,
```

**Variable Record Format**

When you specify that a datafile is in variable record format, SQL\*Loader expects to find the length of each record in a character field at the beginning of each record in the datafile. This format provides some added flexibility over the fixed record format and a performance advantage over the stream record format. For example, you can specify a datafile which is to be interpreted as being in variable record format as follows:

```
INFILE "datafile_name" "var n"
```

where *n* specifies the number of bytes in the record length field. Note that if *n* is not specified, it defaults to five. If it is not specified, SQL\*Loader assumes a length of five. Also note that specifying *n* larger than  $2^{32} - 1$  will result in an error.

**Example : Loading Data in Variable Record Format**

```
load data
infile 'example.dat'      "var 3" into
table example
fields terminated by ',' optionally enclosed by '"' (col1
char(5),
col2 char(7))
```

example.dat:

```
009hello,cd,010world,im,
012my,name is,
```

**Discarded and Rejected Records**

Records read from the input file might not be inserted into the database. Figure below shows the stages at which records may be rejected or discarded.



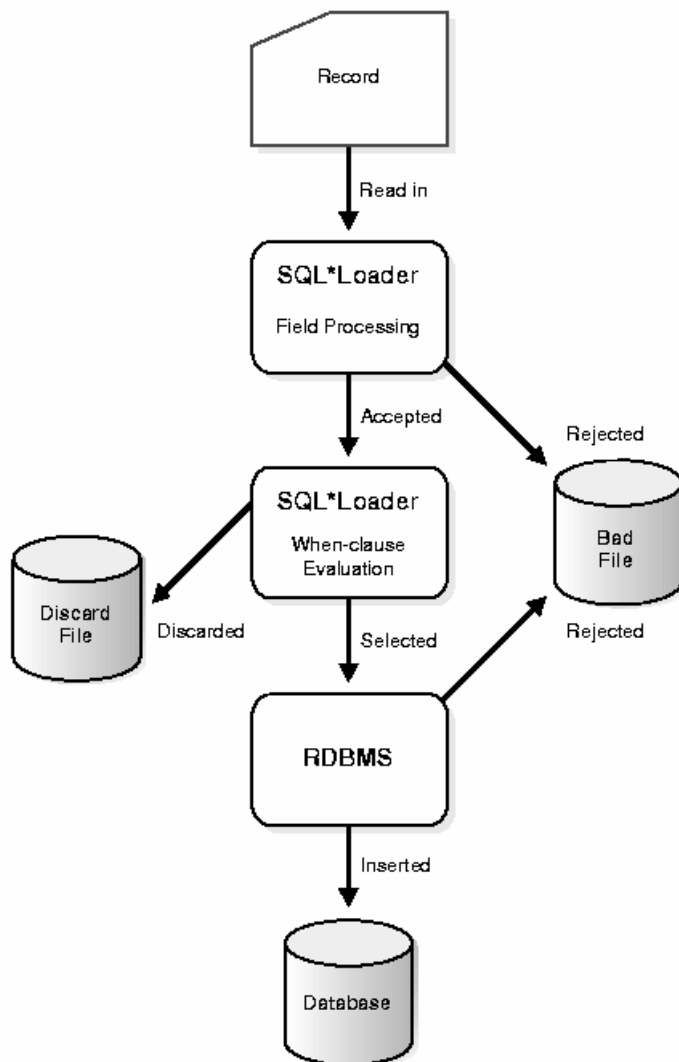
## The Bad File

The bad file contains records rejected, either by SQL\*Loader or by Oracle. Some of the possible reasons for rejection are discussed in the next sections.

### SQL\*Loader Rejects

Records are rejected by SQL\*Loader when the input format is invalid. For example, if the second enclosure delimiter is missing, or if a delimited field exceeds its maximum length, SQL\*Loader rejects the record. Rejected records are placed in the bad file.

**Figure : Record Filtering**



## Oracle Rejects

After a record is accepted for processing by SQL\*Loader, a row is sent to Oracle for insertion. If Oracle determines that the row is valid, then the row is inserted into the database. If not, the record is rejected, and SQL\*Loader puts it in the bad file. The row may be rejected, for example, because a key is not unique, because a required field is null, or because the field contains invalid data for the Oracle datatype.

The bad file is written in the same format as the datafile. So rejected data can be loaded with the existing control file after necessary corrections are made.

## SQL\*Loader Discards

As SQL\*Loader executes, it may create a file called the discard file. This file is created only when it is needed, and only if you have specified that a discard file should be enabled. The discard file contains records that were filtered out of the load because they did not match any record-selection criteria specified in the control file.

The discard file therefore contains records that were not inserted into any table in the database. You can specify the maximum number of such records that the discard file can accept. Data written to any database table is not written to the discard file.

The discard file is written in the same format as the datafile. The discard data can be loaded with the existing control file, after any necessary editing or correcting.

## Log File and Logging Information

When SQL\*Loader begins execution, it creates a log file. If it cannot create a log file, execution terminates. The log file contains a detailed summary of the load, including a description of any errors that occurred during the load.

## Conventional Path Load versus Direct Path Load

SQL\*Loader provides two methods to load data: Conventional Path, which uses a SQL INSERT statement with a bind array, and Direct Path, which loads data directly into a database. The tables to be loaded must already exist in the database, SQL\*Loader never creates tables, it loads existing tables. Tables may already contain data, or they may be empty.

The following privileges are required for a load:

You must have INSERT privileges on the table to be loaded.

You must have DELETE privilege on the table to be loaded, when using the REPLACE or TRUNCATE option to empty out the table's old data before loading the new data in its place.

### Conventional Path

During conventional path loads, the input records are parsed according to the field specifications, and each data field is copied to its corresponding bind array. When the bind array is full (or there is no more data left to read), an array insert is executed.

Note that SQL\*Loader stores LOB fields after a bind array insert is done. Thus, if there are any errors in processing the LOB field (for example, the LOBFILE could not be found), the LOB field is left empty.

There are no special requirements for tables being loaded via the conventional path.

**Direct Path**

A direct path load parses the input records according to the field specifications, converts the input field data to the column datatype and builds a column array. The column array is passed to a block formatter which creates data blocks in Oracle database block format. The newly formatted database blocks are written directly to the database bypassing most RDBMS processing. Direct path load is much faster than conventional path load, but entails several restrictions.

## USER MANAGEMENT

This chapter explains how you can control users' ability to execute system operations and to access schema objects by using privileges, roles, and security policies.

### Introduction to Privileges

A **privilege** is a right to execute a particular type of SQL statement or to access another user's object. Some examples of privileges include the right to:

- Connect to the database (create a session) Create a table
- Select rows from another user's table
- Execute another user's stored procedure

There are two distinct categories of privileges: System

privileges

Schema object privileges

#### **System Privileges:**

A system privilege is the right to perform a particular action, or to perform an action on any schema objects of a particular type. For example, the privileges to create tablespaces and to delete the rows of any table in a database are system privileges.

#### *Grant and Revoke System Privileges*

You can grant or revoke system privileges to users and roles. If you grant system privileges to roles, then you can use the roles to manage system privileges. For example, roles permit privileges to be made selectively available.

#### *Who Can Grant or Revoke System Privileges?*

Only users who have been granted a specific system privilege with the ADMIN OPTION or users with the system privileges GRANT ANY PRIVILEGE or GRANT ANY OBJECT PRIVILEGE can grant or revoke system privileges to other users.

#### **Schema Object Privileges:**

A **schema object privilege** is a privilege or right to perform a particular action on a specific schema object:

- Table View Sequence
- Procedure Function
- Package

Different object privileges are available for different types of schema objects. For example, the privilege to delete rows from the departments table is an object privilege.

A schema object and its synonym are equivalent with respect to privileges. That is, the object privileges granted for a table, view, sequence, procedure, function, or package apply whether referencing the base object by name or using a synonym.

If a synonym is dropped, all grants for the underlying schema object remain in effect, even if the privileges were granted by specifying the dropped synonym.

### ***Grant and Revoke Schema Object Privileges***

Schema object privileges can be granted to and revoked from users and roles. If you grant object privileges to roles, you can make the privileges selectively available.

### ***Who Can Grant Schema Object Privileges?***

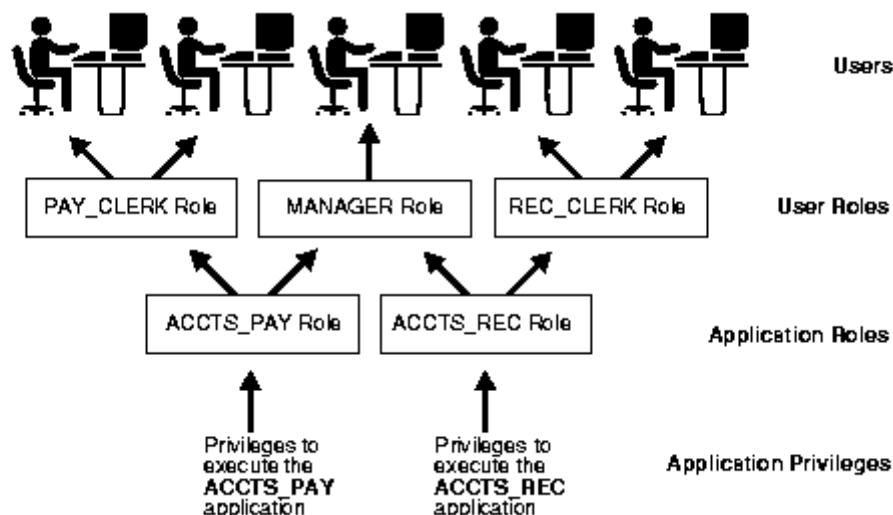
A user automatically has all object privileges for schema objects contained in his or her schema. A user can grant any object privilege on any schema object he or she owns to any other user or role. A user with the GRANT ANY OBJECT PRIVILEGE can grant or revoke any specified object privilege to another user with or without the GRANT OPTION of the GRANT statement. Otherwise, the grantee can use the privilege, but cannot grant it to other users.

## Introduction to Roles

Oracle provides for easy and controlled privilege management through roles. Roles are named groups of related privileges that you grant to users or other roles. Roles are designed to ease the administration of end-user system and schema object privileges. However, roles are not meant to be used for application developers, because the privileges to access schema objects within stored programmatic constructs need to be granted directly.

### **Common Uses for Roles**

In general, you create a role to serve one of two purposes: To manage the privileges for a database application To manage the privileges for a user group Diagram given below and the sections that follow describe the two uses of roles.



*Figure: Common Uses for Roles*

### ***Application Roles***

You grant an application role all privileges necessary to run a given database application. Then, you grant the secure application role to other roles or to specific users. An application can have several different roles, with each role assigned a different set of privileges that allow for more or less data access while using the application.

### ***User Roles***

You create a user role for a group of database users with common privilege requirements. You manage user privileges by granting secure application roles and privileges to the user role and then granting the user role to appropriate users.

### ***The Mechanisms of Roles***

Database roles have the following functionality:

A role can be granted system or schema object privileges.

A role can be granted to other roles. However, a role cannot be granted to itself and cannot be granted circularly. For example, role A cannot be granted to role B if role B has previously been granted to role A.

Any role can be granted to any database user.

Each role granted to a user is, at a given time, either enabled or disabled. A user's security domain includes the privileges of all roles currently enabled for the user and excludes the privileges of any roles currently disabled for the user. Oracle allows database applications and users to enable and disable roles to provide selective availability of privileges.

An indirectly granted role is a role granted to a role. It can be explicitly enabled or disabled for a user. However, by enabling a role that contains other roles, you implicitly enable all indirectly granted roles of the directly granted role.

### ***Grant and Revoke Roles***

You grant or revoke roles from users or other roles using the following options:

The Grant System Privileges/Roles dialog box and Revoke System Privileges/Roles dialog box of Oracle Enterprise Manager

The SQL statements GRANT and REVOKE

Privileges are granted to and revoked from roles using the same options. Roles can also be granted to and revoked from users using the operating system that executes Oracle, or through network services.

### ***Who Can Grant or Revoke Roles?***

Any user with the GRANT ANY ROLE system privilege can grant or revoke any role except a global role to or from other users or roles of the database.

Any user granted a role with the ADMIN OPTION can grant or revoke that role to or from other users or roles of the database. This option allows administrative powers for roles on a selective basis.

***Role Names***

Within a database, each role name must be unique, and no username and role name can be the same. Unlike schema objects, roles are not contained in any schema. Therefore, a user who creates a role can be dropped with no effect on the role.

***Predefined Roles***

The following roles are defined automatically for Oracle databases: CONNECT

RESOURCE DBA

EXP\_FULL\_DATABASE

IMP\_FULL\_DATABASE

These roles are provided for backward compatibility to earlier versions of Oracle and can be modified in the same manner as any other role in an Oracle database.

## IMPORT & EXPORT UTILITY

### Introduction

Export, Import are complementary utilities which allow you to write data in an ORACLE-binary format from the database into operating system files and to read data back from those operating system files. EXPORT, IMPORT are used for the following tasks:

- backup ORACLE data in operating system files restore tables
- that where dropped
- save space or reduce fragmentation in the database
- move data from one owner to another

Because of the special binary format, files which had been created by the EXPORT utility can only be read by IMPORT. Both tools are only used to maintain ORACLE database objects.

### What Is the Export Utility?

Export provides a simple way for you to transfer data objects between Oracle databases, even if they reside on platforms with different hardware and software configurations. Export extracts the object definitions and table data from an Oracle database and stores them in an Oracle binary-format Export dump file located typically on disk or tape.

When you run Export against an Oracle database, objects (such as tables) are extracted, followed by their related objects (such as indexes, comments, and grants) if any, and then written to the Export file.

### Export Method

You can invoke Export in one of the following ways: Enter the following command:

```
exp username/password PARFILE=filename
```

PARFILE is a file containing the export parameters you typically use. If you use different parameters for different databases, you can have multiple parameter files. This is the recommended method.

You can specify the username and password in the parameter file, although, for security reasons, this is not recommended. If you omit the username/password combination, Export prompts you for it.

Enter the following command, adding any needed parameters:

```
exp username/password
```



Enter only the command `exp username/password` to begin an interactive session and let Export prompt you for the information it needs. The interactive method provides less functionality than the parameter-driven method. It exists for backward compatibility.

For example, assume the parameters file `params.dat` contains the parameter `INDEXES=Y` and Export is invoked with the following line:

```
exp system/manager PARFILE=params.dat INDEXES=N
```

In this case, because `INDEXES=N` occurs *after* `PARFILE=params.dat`, `INDEXES=N` overrides the value of the `INDEXES` parameter in the `PARFILE`.

### Direct Path Export

Export provides two methods for exporting table data:

Conventional path Export

Direct path Export

Conventional path Export uses the SQL `SELECT` statement to extract data from tables. Data is read from disk into a buffer cache, and rows are transferred to the evaluating buffer. The data, after passing expression evaluation, is transferred to the Export client, which then writes the data into the export file.

Direct path Export extracts data much faster than a conventional path export. Direct path Export achieves this performance gain by reading data directly, bypassing the SQL command processing layer and saves on data copies whenever possible.

In a direct path Export, data is read from disk into the buffer cache and rows are transferred *directly* to the Export client. The evaluating buffer is bypassed. The data is already in the format that Export expects, thus avoiding unnecessary data conversion. The data is transferred to the Export client, which then writes the data into the export file.

To use direct path Export, specify the `DIRECT=Y` parameter on the command line or in the parameter file. The default is `DIRECT=N`, which extracts the table data using the conventional path.



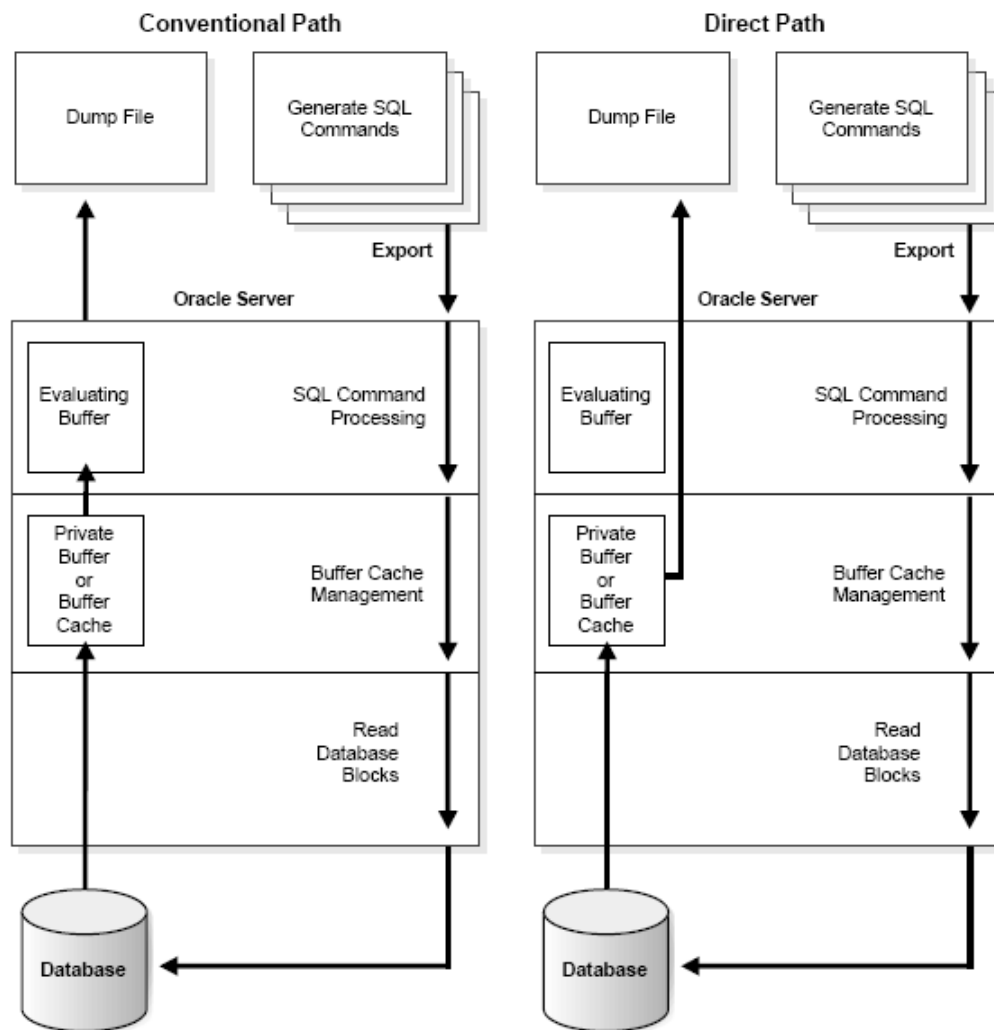


Figure: Database Reads on Conventional Path and Direct Path

### What Is the Import Utility?

The basic concept behind Import is very simple. Import inserts the data objects extracted from one Oracle database by the Export utility (and stored in an Export dump file) into another Oracle database. Export dump files can only be read by Import.

Import reads the object definitions and table data that the Export utility extracted from an Oracle database and stored in an Oracle binary-format Export dump file located typically on disk or tape.

### Invoking Import

You can invoke Import in three ways: Enter the

following command:

```
imp username/password PARFILE=filename
```

PARFILE is a file containing the Import parameters you typically use. If you use different parameters for different databases, you can have multiple parameter files. This is the recommended method.

Enter the following command

```
imp username/password <parameters>
```

Replace <parameters> with various parameters you intend to use. To begin an

interactive session, enter the following command:

```
imp username/password
```

Let Import prompt you for the information it needs. Note that the interactive method does not provide as much functionality as the parameter-driven method. It exists for backward compatibility.

You can use a combination of the first and second options. That is, you can list parameters both in the parameters file and on the command line. In fact, you can specify the same parameter in both places. The position of the PARFILE parameter and other parameters on the command line determines what parameters override others. For example, assume the parameters file params.dat contains the parameter INDEXES=Y and Import is invoked with the following line:

```
imp system/manager PARFILE=params.dat INDEXES=N
```

In this case, because INDEXES=N occurs *after* PARFILE=params.dat, INDEXES=N overrides the value of the INDEXES parameter in the PARFILE. You can specify the username and password in the parameter file, although, for security reasons, this is not recommended.