Table of contents

Chapter 1

After completing this chapter, you will be able to understand:

- ✓ **The concepts of RDBMS**
- ✓ **Design databases**
- ✓ **Relationship**
- ✓ **Degree of relationship**
- ✓ **What is SQL**

RDBMS Concepts

A database is a collection of interrelated data. The collection of data must be logically coherent with some inherent meaning.

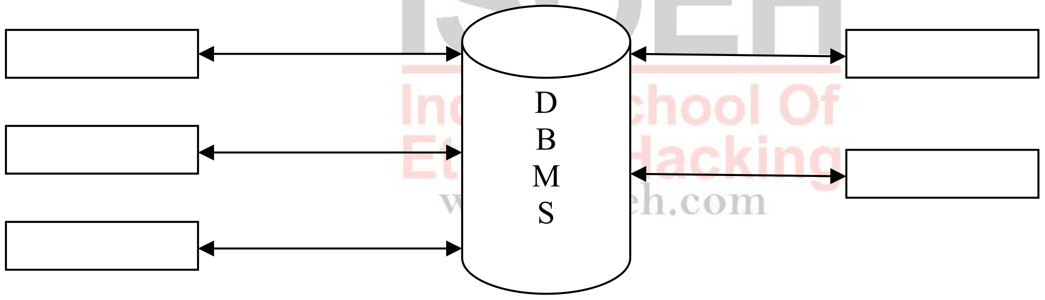
A DBMS is a software system that enables user to record and maintain database. It provides an environment where in data can be stored and retrieved from database easily and most efficiently.

As you might imagine, storing large amounts of data in a spreadsheet or word processing document can become overwhelming very quickly. These one dimensional databases have no efficient way of

1. filtering redundant data,
2. ensuring consistent data entry, or
3. Handling information retrieval.

(Explain: - Flat files in C, excel sheet. Ask them the problem on how to retrieve a name from C flat files. There can be multiple flat files to insert data, locking, access control)

The following picture shows a simplified picture of DBMS.



Application Programs **End users**

Relational Database Models:-

Oracle is a relational database management system, or RDBMS. Relational databases store data in tables. Tables are made up of columns that define the type of data that can be stored in them (character, number, etc.). A table has a minimum of one column. When data is placed in the table, it is stored in rows. This holds true for all relational database vendors.

In Oracle, tables are owned by a user, or schema. The schema is a collection of objects, like tables, that the database user owns. It is possible to have two tables in one database that have the same name as long as they are owned by different users.

Dr.E.F. Codd proposed the relational database model in 1970. It's the basic of RDBMS.

The Relational Model consists of the following.

- ✓ Collections of objects or relations
- ✓ Set of operators to act on relations
- ✓ Data integrity for accuracy and consistency.

A relational database is a collection of relations or two dimensional tables. For example you may store information about all the employees in your company.
In a RDBMS you create several tables to store different piece of information about your employees such as an employee table, a department table and a salary table.

Relating multiple tables

- Each row of data in a table is uniquely identified by a primary key(PK)
- You can logically relate data from multiple tables from foreign key.

Table Employee

Empid	**Fname**	**LName**	**Dept id**
174	**Rahul**	**Maity**	**80**
142	**Sanjana**	**Roy**	**50**
102	**Aniket**	**Dey**	**90**
104	**Moumita**	**Roy**	**60**
202	**Avijit**	**Sen**	**20**
206	**anurag**	**Paul**	**110**

Table Department

Deptid	**Dept name**	**Mgr id**	**Loc id**
10	**Administratiom**	**201**	**1700**
20	**Marketing**	**202**	**2300**
50	**shipping**	**208**	**1515**
60	**IT**	**210**	**2388**
80	**sales**	**220**	**1178**
90	**Accounting**	**222**	**1106**

Primary key **Foreign key** **Primary key**

Guideline for Primary keys and Foreign key

- ✓ You cannot insert duplicate values in a primary keys
- ✓ Primary keys are generally not changed
- ✓ Foreign keys are based on data values and are purely logical, not physical pointers
- ✓ A Foreign key value must match an existing primary key values or unique key values
- ✓ A Foreign key must reference either a primary key or a unique key column
- ✓ Primary keys are NOT NULL.

What is an entity and attribute?

Entity is real world object that have independent existence. An entity could be viewed as a set containing subsets.

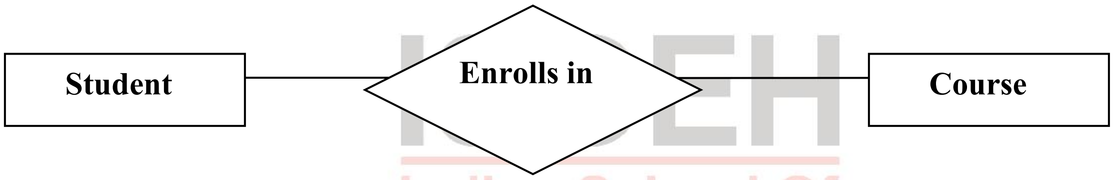
Attributes are data elements that describe an entity. A particular instance of an attribute is a value.

Example:-

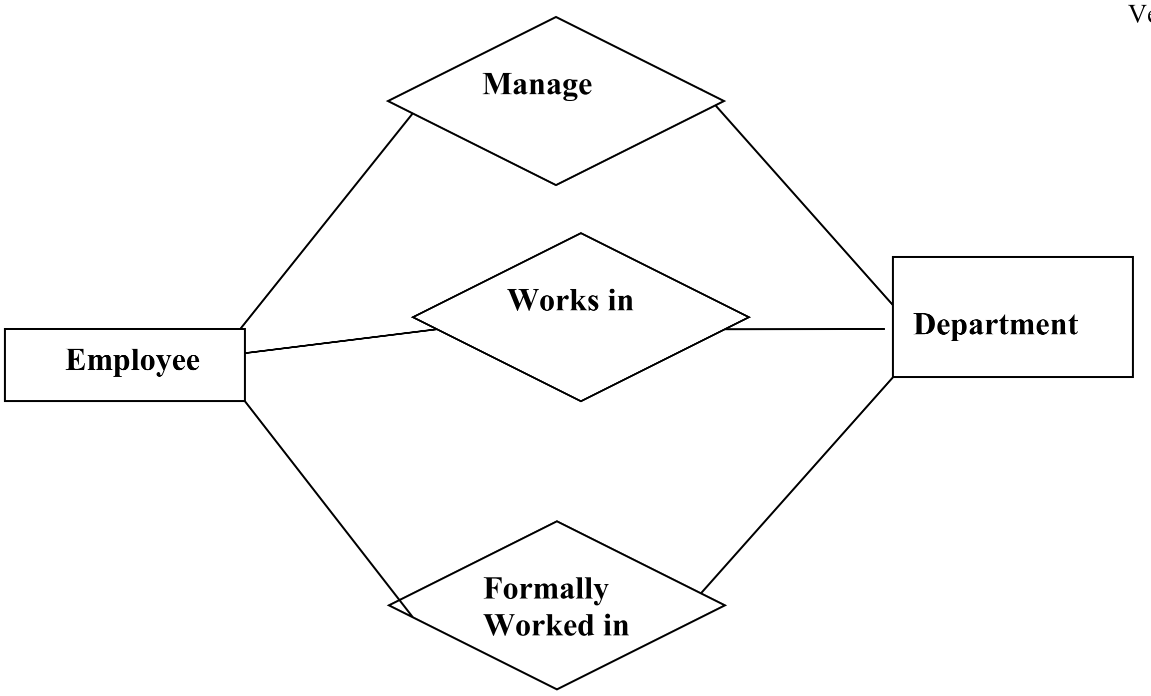
entity	**attribute**
Customer	**Name,address, status**
Book	**ISBN,Title,Author, price**

Relationship:-

This is an association between entities. It is represented by a diamond in the ER diagram.
For example there is a relationship between student and course.. This relationship represents the fact that a student enrolls in course. This relationship could be named as Enrolls in.



Several relationships may exist between the same entities. For example

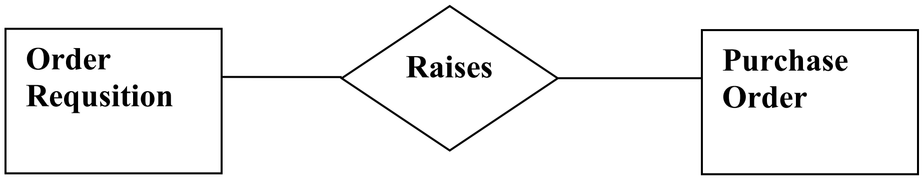


Degree of Relationship

- One to One(1:1)
- One to Many(1:N)
- Many to Many(M:N)

The degree of relationship indicates the link between two entities for a specified occurrence of each. The degree of a relationship is also called cardinality.

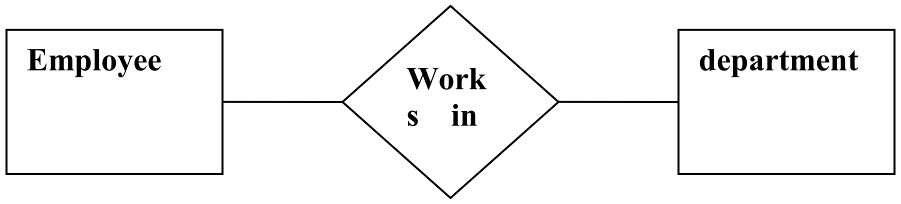
One to One(1:1)



One order requisition raises one purchase order. One purchase order is raised by one order requisition.

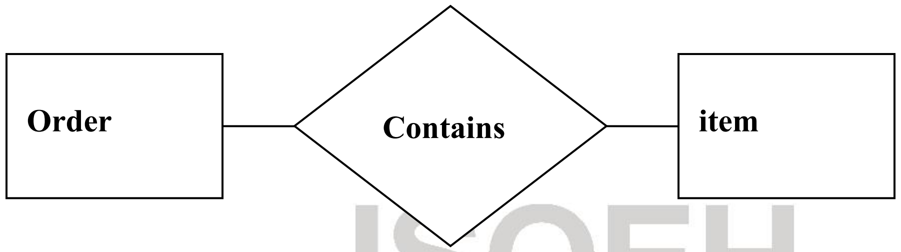
For one occurrence of first entity there can be at most one related occurrence of the second entity and vice versa.

One to Many(1:N)



One employee works in at most one department. One department can have many employees. For one occurrence of first entity there can exist many related occurrence of the second entity and for every occurrence of second entity there exist only one associated occurrence of the first.

Many to Many(M:N)



One order may contain many items. One item can be contained in many orders. For one occurrence of first entity there can exist many related occurrence of the second entity and for every occurrence of second entity there may exist many associated occurrence of the first.

Database Keys

Introduction

For the purposes of clarity we will refer to keys in terms of RDBMS tables but the same definition, principle and naming applies equally to Entity Modeling and Normalisation.

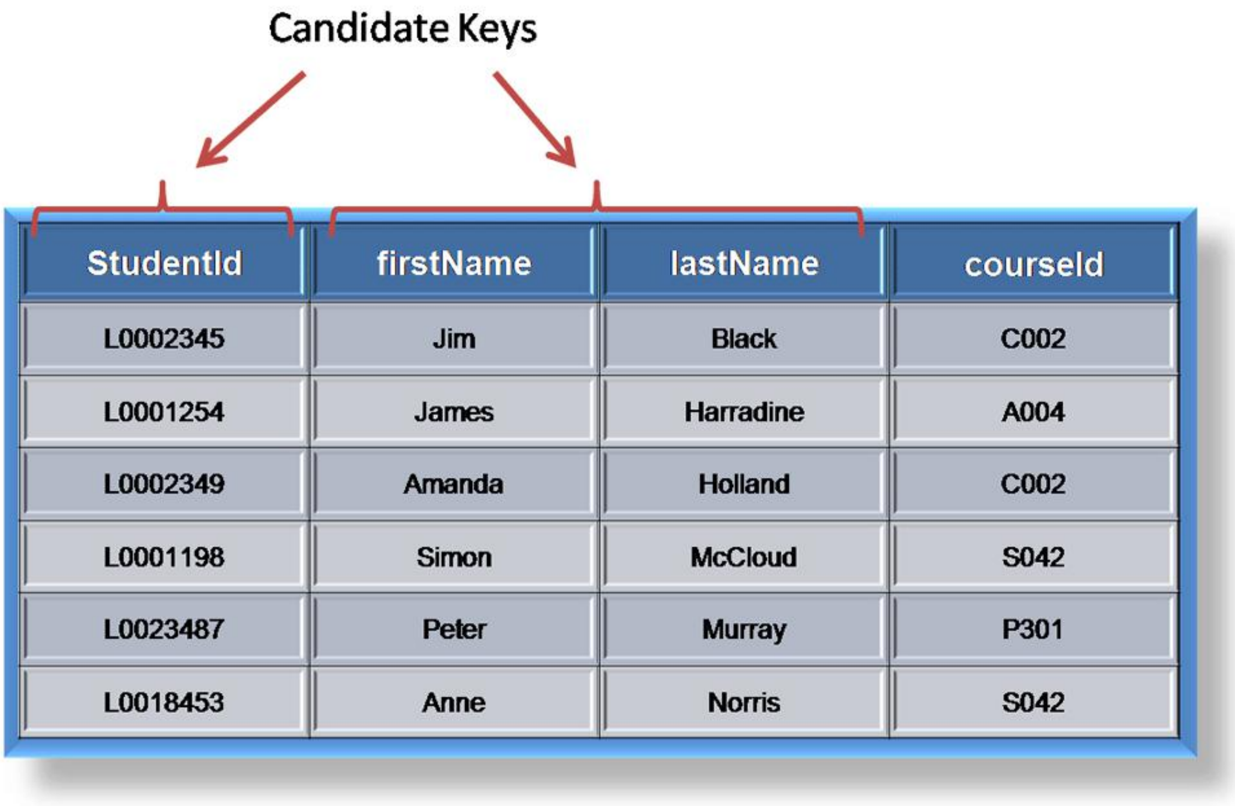
Keys are, as their name suggests, a key part of a relational database and a vital part of the structure of a table. They ensure each record within a table can be uniquely identified by one or a combination of fields within the table. They help enforce integrity and help identify the relationship between tables. There are three main types of keys, candidate keys, primary keys and foreign keys. There is also an alternative key or secondary key that can be used, as the name suggests, as a secondary or alternative key to the primary key.

Super Key

A Super key is any combination of fields within a table that uniquely identifies each record within that table.

Candidate Key

A candidate is a subset of a super key. A candidate key is a single field or the least combination of fields that uniquely identifies each record in the table. The least combination of fields distinguishes a candidate key from a super key. Every table must have at least one candidate key but at the same time can have several.


Candidate Keys

StudentId	firstName	lastName	courseId
L0002345	Jim	Black	C002
L0001254	James	Harradine	A004
L0002349	Amanda	Holland	C002
L0001198	Simon	McCloud	S042
L0023487	Peter	Murray	P301
L0018453	Anne	Norris	S042

As an example we might have a student_id that uniquely identifies the students in a student table. This would be a candidate key. But in the same table we might have the student's first name and last name that also, when combined, uniquely identify the student in a student table. These would both be candidate keys.

In order to be eligible for a candidate key it must pass certain criteria.

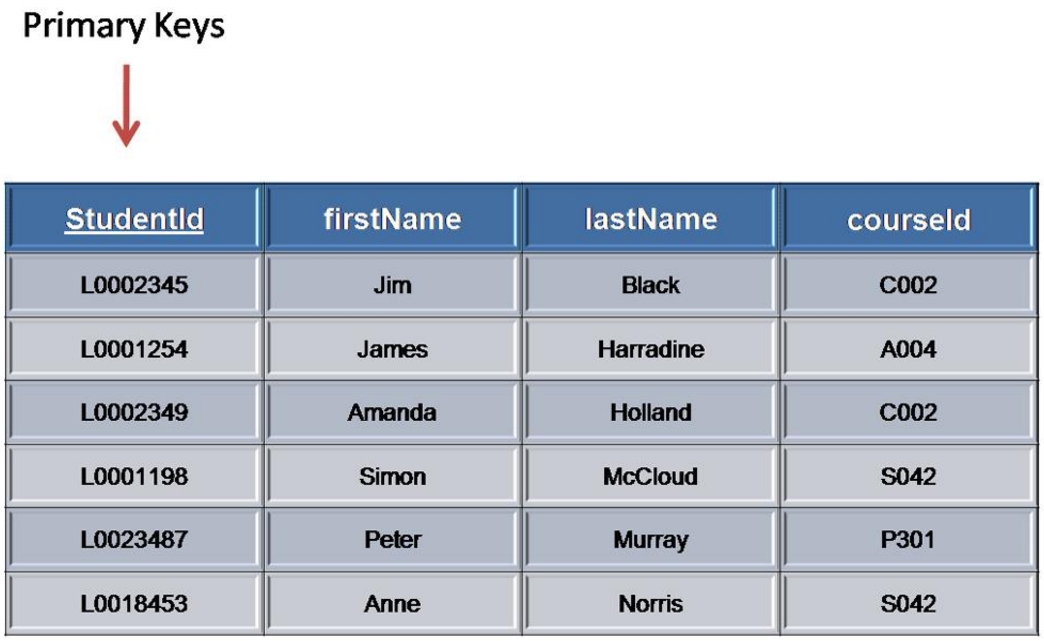
- It must contain unique values
- It must not contain null values
- It contains the minimum number of fields to ensure uniqueness
- It must uniquely identify each record in the table

Once your candidate keys have been identified you can now select one to be your primary key

Primary Key

A primary key is a candidate key that is most appropriate to be the main reference key for the table. As its name suggests, it is the primary key of reference for the table and is used throughout the database to help establish relationships with other tables. As with any candidate key the primary key must contain unique values, must never be null and uniquely identify each record in the table.

As an example, a student id might be a primary key in a student table, a department code in a table of all departments in an organisation. This module has the code DH3D 35 that is no doubt used in a database somewhere to identify RDBMS as a unit in a table of modules. In the table below we have selected the candidate key student_id to be our most appropriate primary key.


Primary Keys

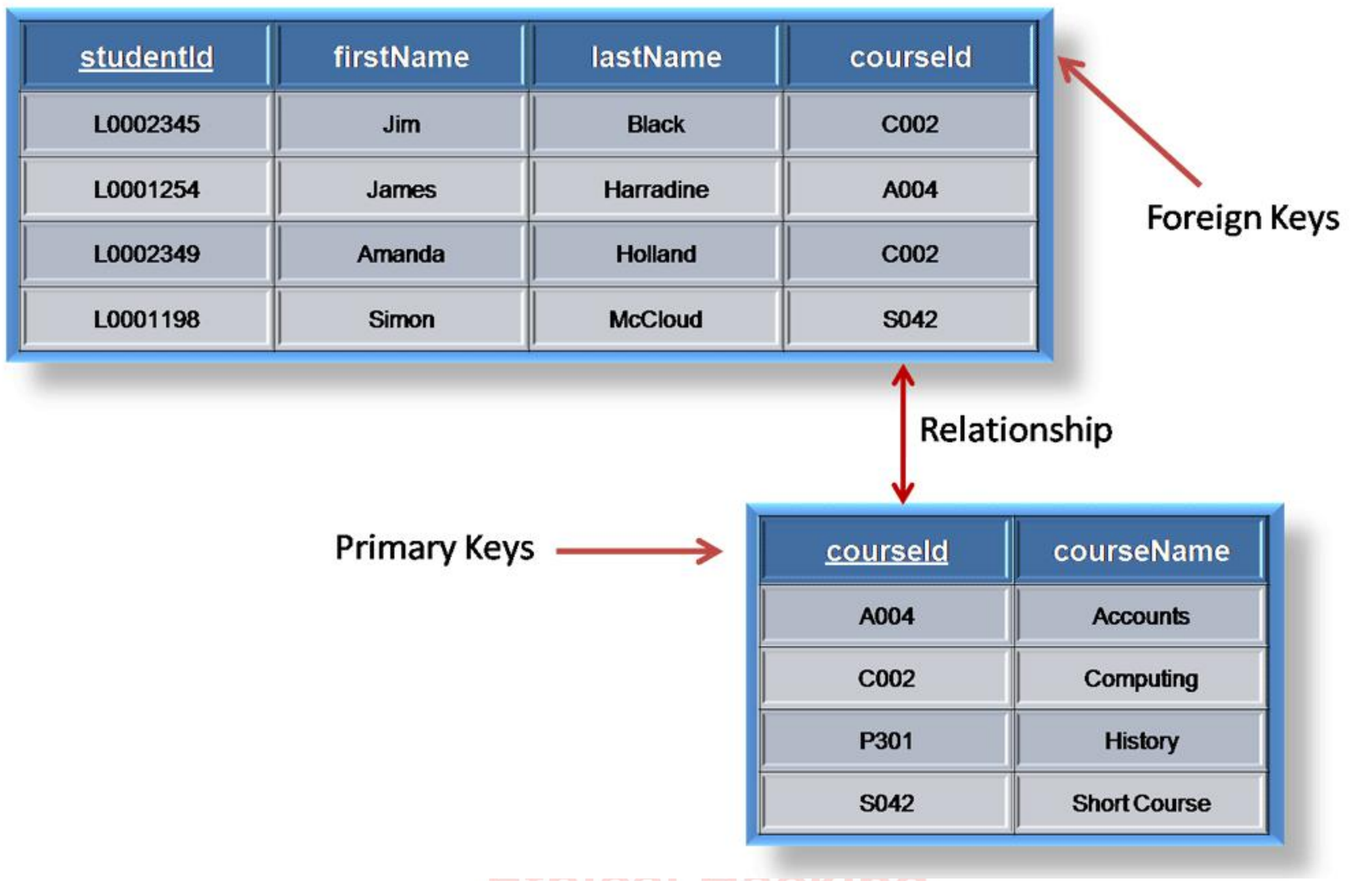
StudentId	firstName	lastName	courseId
L0002345	Jim	Black	C002
L0001254	James	Harradine	A004
L0002349	Amanda	Holland	C002
L0001198	Simon	McCloud	S042
L0023487	Peter	Murray	P301
L0018453	Anne	Norris	S042

Primary keys are mandatory for every table each record must have a value for its primary key. When choosing a primary key from the pool of candidate keys always choose a single simple key over a composite key.

Foreign Key

A foreign key is generally a primary key from one table that appears as a field in another where the first table has a relationship to the second. In other words, if we had a table A with a primary key X that linked to a table B where X was a field in B, then X would be a foreign key in B.

An example might be a student table that contains the course_id the student is attending. Another table lists the courses on offer with course_id being the primary key. The 2 tables are linked through course_id and as such course_id would be a foreign key in the student table.



studentId	firstName	lastName	courseId
L0002345	Jim	Black	C002
L0001254	James	Harradine	A004
L0002349	Amanda	Holland	C002
L0001198	Simon	McCloud	S042

courseId	courseName
A004	Accounts
C002	Computing
P301	History
S042	Short Course

Secondary Key or Alternative Key

A table may have one or more choices for the primary key. Collectively these are known as candidate keys as discuss earlier. One is selected as the primary key. Those not selected are known as secondary keys or alternative keys.

For example in the table showing candidate keys above we identified two candidate keys, studentId and firstName + lastName. The studentId would be the most appropriate for a

Primary key leaving the other candidate key as secondary or alternative key. It should be noted for the other key to be candidate keys, we are assuming you will never have a person with the same first and last name combination. As this is unlikely we might consider fistName+lastName to be a suspect candidate key as it would be restrictive of the data you might enter. It would seem a shame to not allow John Smith onto a course just because there was already another John Smith.

Simple Key

Any of the keys described before (ie primary, secondary or foreign) may comprise one or more fields, for example if firstName and lastName was our key this would be a key of two fields where as studentId is only one. A simple key consists of a single field to uniquely identify a record. In addition the field in itself cannot be broken down into other fields, for example, studentId, which uniquely identifies a particular student, is a single field and therefore is a simple key. No two students would have the same student number.

Compound Key

A compound key consists of more than one field to uniquely identify a record. A compound key is distinguished from a composite key because each field, which makes up the primary key, is also a simple key in its own right. An example might be a table that represents the modules a student is attending. This table has a studentId and a moduleCode as its primary key. Each of the fields that make up the primary key are simple keys because each represents a unique reference when identifying a student in one instance and a module in the other.

Composite Key

A composite key consists of more than one field to uniquely identify a record. This differs from a compound key in that one or more of the attributes, which make up the key, are not simple keys in their own right. Taking the example from compound key, imagine we identified a student by their firstName + lastName. In our table representing students on modules our primary key would now be firstName + lastName + moduleCode. Because firstName + lastName represent a unique reference to a student, they are not each simple keys, they have to be combined in order to uniquely identify the student. Therefore the key for this table is a composite key.

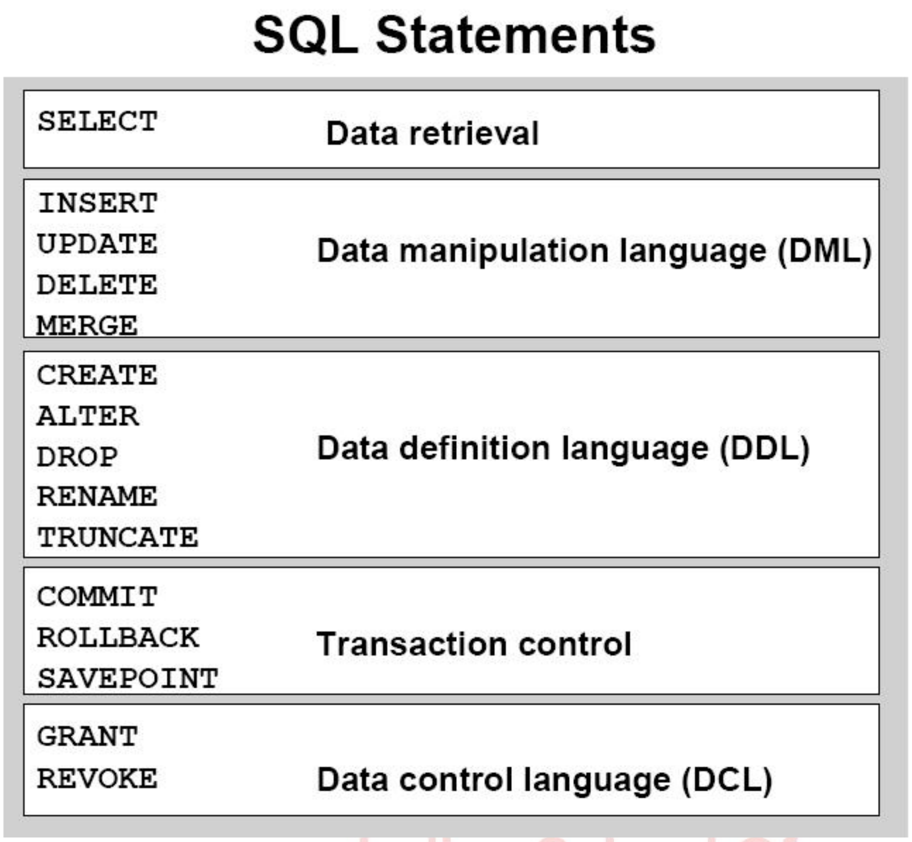
Introduction to SQL:-

To access database you execute a Structure Query Language(SQL) which is an ANSI standard language for accessing RDBMS. The language contains large set of operators for partitioning and combining relations.

SQL provides statements for variety of database tasks such as:-

- ✓ Querying data
- ✓ Inserting,updating and deleting rows in a table
- ✓ Creating,replacing,altering and droping objects
- ✓ Guaranteeing database consistency and integrity

Based on the most common database activities the SQL ststements are categorized as :



SQL Statements

SELECT	Data retrieval

INSERT
UPDATE
DELETE
MERGE
	Data manipulation language (DML)

CREATE ALTER DROP RENAME TRUNCATE	Data definition language (DDL)
COMMIT ROLLBACK SAVEPOINT	Transaction control
GRANT REVOKE	Data control language (DCL)

Oracle provides two types of interface for interacting with the database:-

SQL*Plus :- Allows users to interactively use SQL commands
iSQL*Plus :-Web based enterprise manager

Users:
scott / tiger
System / a1234
Connect / as SYSDBA

Chapter 2

After completing this chapter, you will be able to :

- ✓ **Write simple SELECT statements to retrieve data from the database**
- ✓ **Use column alias**
- ✓ **Use arithmetic expression**
- ✓ **Limit the rows retrieved by a query using WHERE clause**
- ✓ **Use Order By clause**
- ✓ **Perform multilevel sorting**

Basic Query statements

Executing simple select statements:-

We want to query EMP table so as to retrieve all rows. The command shown below:-
SQL> select * from emp;

```
EMPNO   ENAME        JOB        MGR  HIREDATE   SAL  COMM  DEPTNO
     7369 SMITH    CLERK       7902 17-DEC-80    800               20
     7499 ALLEN    SALESMAN    7698 20-FEB-81   1600    300        30
     7521 WARD     SALESMAN    7698 22-FEB-81   1250    500        30
     7566 JONES    MANAGER     7839 02-APR-81   2975               20
     7654 MARTIN   SALESMAN    7698 28-SEP-81   1250   1400        30
     7698 BLAKE    MANAGER     7839 01-MAY-81   2850               30
     7782 CLARK    MANAGER     7839 09-JUN-81   2450               10
     7788 SCOTT    ANALYST     7566 19-APR-87   3000               20
     7839 KING     PRESIDENT        17-NOV-81   5000               10
     7844 TURNER   SALESMAN    7698 08-SEP-81   1500      0        30
     7876 ADAMS    CLERK       7788 23-MAY-87   1100               20
     7900 JAMES    CLERK       7698 03-DEC-81    950               30
     7902 FORD     ANALYST     7566 03-DEC-81   3000               20
     7934 MILLER   CLERK       7782 23-JAN-82   1300               10
```

A select clause followed by select list that contains columns and expressions. An * indicates all columns are to be displayed.
A FROM clause specifies the table name that contains the columns listed in the SELECT clause.

Same way we can view the contents of DEPT and SALGRADE tables.

select * from dept;

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

select * from salgrade;

GRADE	LOSAL	HISAL
1	700	1200
2	1201	1400
3	1401	2000
4	2001	3000
5	3001	9999

To view the selective columns:-

select ename,job, sal from emp;

ENAME	JOB	SAL
SMITH	CLERK	800
ALLEN	SALESMAN	1600
WARD	SALESMAN	1250
JONES	MANAGER	2975
MARTIN	SALESMAN	1250
BLAKE	MANAGER	2850
CLARK	MANAGER	2450
SCOTT	ANALYST	3000
KING	PRESIDENT	5000
TURNER	SALESMAN	1500
ADAMS	CLERK	1100
JAMES	CLERK	950
FORD	ANALYST	3000
MILLER	CLERK	1300

14 rows selected.

To view structure of a table use DESCRIBE command as follows:-
describe emp;

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

You can display a single column as follows:-
select dname from dept;

DNAME
ACCOUNTING
RESEARCH
SALES
OPERATIONS

Column alias:-

By default the column heading is the same name as that of the column name. and is in uppercase.
But using column alias we can give alternative heading for a column
Name as follows:-

select deptno as "deptid", dname "Department name", loc Location
from dept;

deptid	Department name	Location
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

- The use of keyword as is optional between the column name and the alias
- Separate the column name and alias with a space.

Using Arithmetic expression: Let's query the EMP table and display employee name and salary along with the salary incremented by 500.

select ename,sal,sal+500
from emp;

ENAME	SAL	SAL+500
SMITH	800	1300
ALLEN	1600	2100
WARD	1250	1750
JONES	2975	3475
MARTIN	1250	1750
BLAKE	2850	3350
CLARK	2450	2950
SCOTT	3000	3500
KING	5000	5500
TURNER	1500	2000
ADAMS	1100	1600
JAMES	950	1450
FORD	3000	3500
MILLER	1300	1800

DISTINCT keyword

Let's query the job column from EMP table

select job from emp;

JOB
CLERK
SALESMAN
SALESMAN
MANAGER
SALESMAN
MANAGER
MANAGER
ANALYST

PRESIDENT
SALESMAN
CLERK
CLERK
ANALYST
CLERK

14 rows selected.

Here we find that output contains duplicate rows. To eliminate the duplicate rows we use DISTINCT keyword as follows.

```
SQL> select distinct job
  2  from emp;

JOB
---------
CLERK
SALESMAN
PRESIDENT
MANAGER
ANALYST
```

Remember DISTINCT keyword eliminates duplicate rows, not duplicate values.
DISTINCT keyword should be used only once in SELECT clause.
The following statement will cause an error.

```
select distinct deptno,distinct job
from emp;
```

<u>WHERE clause</u>

The WHERE clause allows you to filter the results or rows from an SQL statement - select, insert, update, or delete statement.

Let us query the emp table to retrieve the details of employees working in department no 20.

```
select * from emp
where deptno=20;
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	17-DEC-80	800		20

7566	JONES	MANAGER	7839	02-APR-81	2975		20
7788	SCOTT	ANALYST	7566	19-APR-87	3000		20
7876	ADAMS	CLERK	7788	23-MAY-87	1100		20
7902	FORD	ANALYST	7566	03-DEC-81	3000		20

Let us display employee name,job and salary of all employees whose job title is MANAGER.

select ename,job,sal
from emp
where job='MANAGER';

ENAME	JOB	SAL
JONES	MANAGER	2975
BLAKE	MANAGER	2850
CLARK	MANAGER	2450

Remember that date values format sensitive. The default date format is ‘DD-MM-YY’.
Let us display employee name,job and hiredate of all employees who ‘joined after 31st Dec 81’.

select ename,job,hiredate
from emp
where hiredate>'31-DEC-81';

ENAME	JOB	HIREDATE
SCOTT	ANALYST	19-APR-87
ADAMS	CLERK	23-MAY-87
MILLER	CLERK	23-JAN-82

<u>Comparison operators:-</u>

Operator	Description
Between…And	Between two values(inclusive)
IN	Comparision in a list of values
LIKE	Compare with a character pattern
IS NULL	Check if it is null

Let us display the details of employees whose salary is in between 2000 and 3000
select *
from emp
where sal between 2000 and 3000

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7566	JONES	MANAGER	7839	02-APR-81	2975		20
7698	BLAKE	MANAGER	7839	01-MAY-81	2850		30
7782	CLARK	MANAGER	7839	09-JUN-81	2450		10
7788	SCOTT	ANALYST	7566	19-APR-87	3000		20
7902	FORD	ANALYST	7566	03-DEC-81	3000		20

Let us display names and date of joining of those employees who have joined before 1st JAN 81 and after 31st DEC-82.

```
select ename,hiredate
from emp
where hiredate NOT BETWEEN '01-JAN-81' AND '31-DEC-82';
```

ENAME	HIREDATE
SMITH	17-DEC-80
SCOTT	19-APR-87
ADAMS	23-MAY-87

Let us display names,jobs,and salaries of all employees whose job title is CLERK or SALESMAN or ANALYST.

```
select ename,job,sal
from emp
where  job IN('CLERK','SALESMAN','ANALYST');
```

ENAME	JOB	SAL
SMITH	CLERK	800
ALLEN	SALESMAN	1600
WARD	SALESMAN	1250
MARTIN	SALESMAN	1250
SCOTT	ANALYST	3000
TURNER	SALESMAN	1500
ADAMS	CLERK	1100
JAMES	CLERK	950
FORD	ANALYST	3000
MILLER	CLERK	1300

10 rows selected.

Now let us display the names,jobs and dept no of those employees who do not belong to dept no 10 or 20.

```
select ename,job,deptno
from emp
where  deptno NOT IN(10,20);
```

ENAME	JOB	DEPTNO
ALLEN	SALESMAN	30
WARD	SALESMAN	30
MARTIN	SALESMAN	30
BLAKE	MANAGER	30
TURNER	SALESMAN	30
JAMES	CLERK	30

6 rows selected.

Let us display the details of those employees who do not have a manager.

```
select *
from emp
where  mgr is null;
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7839	KING	PRESIDENT		17-NOV-81	5000		10

Let us display the details of those employees who earn commission. Here we will use negation of IS NULL operator.

```
select *
from emp
where  comm IS NOT NULL;
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7499	ALLEN	SALESMAN	7698	20-FEB-81	1600	300	30
7521	WARD	SALESMAN	7698	22-FEB-81	1250	500	30
7654	MARTIN	SALESMAN	7698	28-SEP-81	1250	1400	30
7844	TURNER	SALESMAN	7698	08-SEP-81	1500	0	30

Let us display the names of employees whose names start with 'S'(not 's').

```
select  ename
from emp
```

where ename LIKE 'S%' ;

ENAME
SMITH
SCOTT

- **LIKE is a pattern matching comparison operator**
- **% and '_' are wild card characters used with LIKE operator to specify a pattern.**
- **'_' represent a single character.**

Consider the following query.

```
select  ename
from emp
where  ename LIKE  '_A%';
```

ENAME
WARD
MARTIN
JAMES

Let us display the name and hiredate of all the employees who did not join in the year 81.

```
select  ename,hiredate
from emp
where  hiredate NOT LIKE '%81' ;
```

ENAME	HIREDATE
SMITH	17-DEC-80
SCOTT	19-APR-87
ADAMS	23-MAY-87
MILLER	23-JAN-82

Logical Operator:-

Logical operator	Description
AND	Combines two or more conditions

OR	**Returns TRUE if either of the condition is true**
NOT	**Negates the result of a single condition**

Let us find the details of those employees whose salary are less than 2000 and whose job title is 'CLERK'.

select *
from emp
where sal<2000 AND job='CLERK' ;

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	17-DEC-80	800		20
7876	ADAMS	CLERK	7788	23-MAY-87	1100		20
7900	JAMES	CLERK	7698	03-DEC-81	950		30
7934	MILLER	CLERK	7782	23-JAN-82	1300		10

Consider the another query:-

select *
from emp
where sal<2000 OR job='CLERK' ;

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	17-DEC-80	800		20
7499	ALLEN	SALESMAN	7698	20-FEB-81	1600	300	30
7521	WARD	SALESMAN	7698	22-FEB-81	1250	500	30
7654	MARTIN	SALESMAN	7698	28-SEP-81	1250	1400	30
7844	TURNER	SALESMAN	7698	08-SEP-81	1500	0	30
7876	ADAMS	CLERK	7788	23-MAY-87	1100		20
7900	JAMES	CLERK	7698	03-DEC-81	950		30
7934	MILLER	CLERK	7782	23-JAN-82	1300		10

8 rows selected.

Consider one more query.
select *
from emp
where sal<3000 AND (job='CLERK' OR deptno=10);

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	17-DEC-80	800		20
7782	CLARK	MANAGER	7839	09-JUN-81	2450		10
7876	ADAMS	CLERK	7788	23-MAY-87	1100		20
7900	JAMES	CLERK	7698	03-DEC-81	950		30
7934	MILLER	CLERK	7782	23-JAN-82	1300		10

Sorting Rows retrieved by a query

Many times you may need to display the rows retrieved by a query in a particular order.
For example, let us query the emp table to retrieve the employee names and their salaries and sort the results according to the ascending order of the salary. Look at the use of ORDER BY clause in the following statement.

```
select ename,sal
from emp
ORDER BY sal;
```

ENAME	SAL
SMITH	800
JAMES	950
ADAMS	1100
WARD	1250
MARTIN	1250
MILLER	1300
TURNER	1500
ALLEN	1600
CLARK	2450
BLAKE	2850
JONES	2975
SCOTT	3000
FORD	3000
KING	5000

14 rows selected.

Here you find that rows are sorted in ascending order of salary. You may notice that nor sort order is specified i.e. default is taken as ascending order.
Now to sort it in ascending order

select ename,sal
from emp
ORDER BY sal desc;

ENAME	SAL
KING	5000
FORD	3000
SCOTT	3000
JONES	2975
BLAKE	2850
CLARK	2450
ALLEN	1600
TURNER	1500
MILLER	1300
WARD	1250
MARTIN	1250
ADAMS	1100
JAMES	950
SMITH	800

14 rows selected.

You can also specify column position instead of column name. Consider the following query.

select ename,sal,job
from emp
ORDER BY 2;

ENAME	SAL	JOB
SMITH	800	CLERK
JAMES	950	CLERK
ADAMS	1100	CLERK
WARD	1250	SALESMAN
MARTIN	1250	SALESMAN
MILLER	1300	CLERK
TURNER	1500	SALESMAN
ALLEN	1600	SALESMAN
CLARK	2450	MANAGER

BLAKE	**2850**	**MANAGER**
JONES	**2975**	**MANAGER**
SCOTT	**3000**	**ANALYST**
FORD	**3000**	**ANALYST**
KING	**5000**	**PRESIDENT**

14 rows selected.

You may even sort the result of a query by an expression.
Let us query the emp table to display the employee numbers,names and their annual salaries sorted in descending order.

```
select empno,ename,sal*12  ann_sal
from emp
ORDER BY ann_sal DESC
```

EMPNO	ENAME	ANN_SAL
7839	KING	60000
7902	FORD	36000
7788	SCOTT	36000
7566	JONES	35700
7698	BLAKE	34200
7782	CLARK	29400
7499	ALLEN	19200
7844	TURNER	18000
7934	MILLER	15600
7521	WARD	15000
7654	MARTIN	15000
7876	ADAMS	13200
7900	JAMES	11400
7369	SMITH	9600

14 rows selected.

You can sort the rows retrieved by a query that has a condition.
Consider the following query where we will display empno,names and hiredate who works in deptno 10 and we will sort the rows by hiredate.

```
select empno,ename,hiredate
from emp
where deptno=10
```

order by hiredate;

EMPNO	ENAME	HIREDATE
7782	CLARK	09-JUN-81
7839	KING	17-NOV-81
7934	MILLER	23-JAN-82

Sorting by multiple column

You can sort the rows retrieved by queries by multiple columns.
Let us query the emp table to display empname,job,deptno and salary. We will sort the results in ascending order and then on descending order of salary so that in each department salary appears from highest to the lowest.

select ename,job,deptno,sal
from emp
ORDER BY deptno, sal DESC;

ENAME	JOB	DEPTNO	SAL
KING	PRESIDENT	10	5000
CLARK	MANAGER	10	2450
MILLER	CLERK	10	1300
SCOTT	ANALYST	20	3000
FORD	ANALYST	20	3000
JONES	MANAGER	20	2975
ADAMS	CLERK	20	1100
SMITH	CLERK	20	800
BLAKE	MANAGER	30	2850
ALLEN	SALESMAN	30	1600
TURNER	SALESMAN	30	1500
MARTIN	SALESMAN	30	1250
WARD	SALESMAN	30	1250
JAMES	CLERK	30	950

14 rows selected.

Chapter 3

✓ **SQL Function**

Types of functions:

- ✓ **Single Row functions**
- ✓ **Group or Aggregate Functions**

  - Single-row functions operate on one row at a time and return one row of output for each input row.
  - Aggregate functions operate on multiple rows at the same time and return one row of the output.

Using Single-Row Functions

- Character functions
- Numeric functions
- Conversion functions
- Date functions
- General Functions

Character Functions

Character functions accept character input, which may come from a column in a table or, more generally, from any expression.

ASCII() and CHR()

You use ASCII(*x*) to get the ASCII value for the character *x*. You use CHR(*x*) to get the character with the ASCII value of *x*.
The following query gets the ASCII value of a, A, z, Z, 0, and 9 using ASCII():

```
SELECT ASCII('a'), ASCII('A'), ASCII('z'), ASCII('Z'), ASCII(0), ASCII(9)
FROM dual;
```

ASCII('A')	ASCII('A')	ASCII('Z')	ASCII('Z')	ASCII(0)	ASCII(9)
97	65	122	90	48	57

dual *table contains a single row through which you may perform queries that don't go against a particular table.*

The following query gets the characters with the ASCII values of 97, 65, 122, 90, 48, and 57 using CHR():

SELECT CHR(97), CHR(65), CHR(122), CHR(90), CHR(48), CHR(57)
FROM dual;

CHR	CHR	CHR	CHR	CHR	CHR
a	A	z	Z	0	9

CONCAT()

You use CONCAT(*x*, *y*) to append *y* to *x* and then return the new string. The following query appends last_name to first_name using CONCAT():

SELECT CONCAT(first_name, last_name)
FROM customers;

CONCAT(FIRST_NAME,LAST_NAME)
JohnBrown
CynthiaGreen
SteveWhite
GailBlack
DoreenBlue

INITCAP()

You use INITCAP(*x*) to convert the initial letter of each word in *x* to uppercase.
SELECT INITCAP(description)
FROM products;

INITCAP(DESCRIPTION)
Their Greatest Hits
A Description Of Modern Science
Introduction To Chemistry
A Star Explodes
Action Movie About A Future War
Series On Mysterious Activities
Aliens Return
Adventures Of Heroes
Alien From Another Planet Lands On Earth
The Best Classical Music
The Best Popular Music
Debut Album

12 rows selected.

INSTR()

You use INSTR(*x*, *find_string* [, *start*] [, *occurrence*]) to search for *find_string* in *x*. INSTR() returns the position at which *find_string* occurs. You can supply an optional *start* position to begin the search. You can also supply an optional *occurrence* that indicates which occurrence of *find_string* should be returned.

The following query gets the position where the string Science occurs in the name column for product #1:

```
SELECT name, INSTR(name, 'Science')
FROM products
WHERE product_id = 1;

NAME INSTR(NAME,'SCIENCE')
------------------------------ ---------------------
Modern Science                                   8
```

The next query displays the position where the second occurrence of the e character occurs, starting from the beginning of the product name:

```
SELECT name, INSTR(name, 'e', 1, 2)
FROM products
WHERE product_id = 1;
NAME                             INSTR(NAME,'E',1,2)
------------------------------ ------------------
Modern Science                              11
```

Notice the second e in Modern Science is the eleventh character.

LENGTH()

You use LENGTH(*x*) to get the number of characters in *x*.

select length('abcd')
from dual;

LENGTH('ABCD')
4

LOWER() and UPPER()

You use LOWER(*x*) to convert the letters in *x* to lowercase. Similarly, you use UPPER(*x*) to convert the letters in *x* to uppercase.

```
SELECT UPPER(first_name), LOWER(last_name)
FROM customers;
```

UPPER(FIRST_NAME)	LOWER(LAST_NAME)
JOHN	brown
CYNTHIA	green
STEVE	white
GAIL	black
DOREEN	blue

LPAD() and RPAD()

You use LPAD(*x*, *width* [, *pad_string*]) to pad *x* with spaces to the left to bring the total length of the string up to *width* characters. You can supply an optional *pad_string*, which specifies a string to be repeated to the left of *x* to fill up the padded space. The resulting padded string is then returned. Similarly, you use RPAD(*x*, *width* [, *pad_string*]) to pad *x* with strings to the right.

```
SELECT RPAD(name, 30, '.'), LPAD(price, 8, '*+')
FROM products
WHERE product_id < 4;

RPAD(NAME,30,'.') LPAD(PRICE)
----------------------------- --------
Modern Science................ *+*19.95
Chemistry..................... *+*+*+30
Supernova..................... *+*25.99
```

LTRIM(), RTRIM(), and TRIM()

You use LTRIM(*x* [, *trim_string*]) to trim characters from the left of *x*. You can supply an optional *trim_string,* which specifies the characters to trim; if no *trim_string* is supplied; spaces are trimmed by default. Similarly, you use RTRIM() to trim characters from the right of *x*.

You use TRIM() to trim characters from the left and right of *x*. The following query uses these three functions:

```
SELECT
LTRIM(' Hello Gail Seymour!'),
RTRIM('Hi Doreen Oakley!abcabc', 'abc'),
TRIM('0' FROM '000Hey Steve Button!00000')
FROM dual;

LTRIM('HELLOGAILSEY RTRIM('HIDOREENOA TRIM('0'FROM'000H
------------------- ----------------- -----------------
```

Hello Gail Seymour! Hi Doreen Oakley! Hey Steve Button!

REPLACE()

You use REPLACE(*x*, *search_string*, *replace_string*) to search *x* for *search_string* and replace it with *replace_string*.

```
SELECT REPLACE(name, 'Science', 'Physics')
FROM products
WHERE product_id = 1;
REPLACE(NAME,'SCIENCE','PHYSICS')
---------------------------------
Modern Physics
```

NOTE
REPLACE() *doesn't modify the actual row in the database; only the row returned by the function is modified.*

SUBSTR()

You use SUBSTR(*x*, *start* [, *length*]) to return a substring of *x* that begins at the position specified by *start*. You can also provide an optional *length* for the substring.

```
SELECT SUBSTR('computer', 2, 3)
FROM  dual;
```

SUBSTR('C
omp

- **Numeric functions**

ABS()

You use ABS(*x*) to get the absolute value of *x*.

```
SELECT ABS(10), ABS(-10)
FROM dual;
```

ABS(10)	ABS(-10)
10	10

CEIL()

You use CEIL(*x*) to get the smallest integer greater than or equal to *x*. The following query uses CEIL() to get the absolute values of 5.8 and –5.2:

```
SELECT CEIL(5.8), CEIL(-5.2)
FROM dual;
CEIL(5.8) CEIL(-5.2)
---------- ----------
6 -5
```

FLOOR()

You use FLOOR(*x*) to get the largest integer less than or equal to *x*. The following query uses FLOOR() to get the absolute value of 5.8 and –5.2:

```
SELECT FLOOR(5.8), FLOOR(-5.2)
FROM dual;
FLOOR(5.8) FLOOR(-5.2)
---------- -----------
5 -6
```

The floor for 5.8 is 5; because 5 is the largest integer less than 5.8. The floor for –5.2 is –6, because –5.2 is negative, and the largest integer less than this is –6.

MOD()

You use MOD(*x*, *y*) to get the remainder when *x* is divided by *y*. The following query uses MOD() to get the remainder when 8 is divided by 3 and 4:

```
SELECT MOD(8, 3), MOD(8, 4)
FROM dual;
MOD(8,3) MOD(8,4)
---------- ----------
2 0
```

POWER()

You use POWER(*x*, *y*) to get the result of *x* raised to the power *y*. The following query uses POWER() to get 2 raised to the power 1 and 3:

```
SELECT POWER(2, 1), POWER(2, 3)
FROM dual;
POWER(2,1) POWER(2,3)
---------- ----------
2 8
```

ROUND()

You use ROUND(*x*, [*y*]) to get the result of rounding *x* to an optional *y* decimal places. If *y* is omitted, *x* is rounded to zero decimal places. If *y* is negative, *x* is rounded to the left of the decimal point.

```
SELECT ROUND(5.75), ROUND(5.75, 1), ROUND(5.75, -1)
FROM dual;
ROUND(5.75) ROUND(5.75,1) ROUND(5.75,-1)
----------- ------------- --------------
6 5.8 10
```

5.75 rounded to zero decimal places is 6; 5.75 rounded to one decimal place (to the right of the decimal point) is 5.8; and 5.75 rounded to one decimal place to the left of the decimal point (as indicated using a negative sign) is 10.

SIGN()

You use SIGN(*x*) to get the sign of *x*. SIGN() returns –1 if x is negative, 1 if *x* is positive, or 0 if *x* is zero. The following query gets the sign of –5, 5, and 0:

```
SELECT SIGN(-5), SIGN(5), SIGN(0)
FROM dual;
SIGN(-5) SIGN(5) SIGN(0)
---------- ---------- ----------
-1 1 0
```

SQRT()

You use SQRT(*x*) to get the square root of *x*. The following query gets the square root of 25 and 5:

```
SELECT SQRT(25), SQRT(5)
FROM dual;
SQRT(25) SQRT(5)
---------- ----------
5                    2.23606798
```

TRUNC()

You use TRUNC(*x*, [*y*]) to get the result of truncating the number *x* to an optional *y* decimal places. If *y* is omitted, *x* is truncated to zero decimal places. If *y* is negative, *x* is truncated to the left of the decimal point. The following query truncates 5.75 to zero, 1, and –1 decimal places:

```
SELECT TRUNC(5.75), TRUNC(5.75, 1), TRUNC(5.75, -1)
FROM dual;
TRUNC(5.75) TRUNC(5.75,1) TRUNC(5.75,-1)
```

```
----------- ------------- --------------
5 5.7 0
```

In the above, 5.75 truncated to zero decimal places is 5; 5.75 truncated to one decimal place (to the right of the decimal point) is 5.7; and 5.75 truncated to one decimal place to the left of the decimal point (as indicated using a negative sign) is 0.

Conversion functions

TO_CHAR()

You use TO_CHAR(*x* [, *format*]) to convert *x* to a string.
The following query converts 12345.67 to a string:

```
SELECT TO_CHAR(12345.67)
FROM dual;
---------
12345.67
```

The next query uses TO_CHAR() to convert 12345678.90 to a string and specifies this number is to be converted using the format 99,999.99.

```
SELECT TO_CHAR(12345.67, '99,999.99')
FROM dual;
TO_CHAR(12
----------
12,345.67
```

TO_NUMBER()

You use TO_NUMBER(*x* [, *format*]) to convert *x* to a number.

```
SELECT TO_NUMBER('970.13')
FROM dual;
TO_NUMBER('970.13')
-------------------
970.13
```

TO_DATE()

You use TO_DATE(*x* [, *format*]) to convert the *x* string to a datetime. You can provide an optional *format* string to indicate the format of *x*. If you omit *format*, the date must be in the default database format (usually DD-MON-YYYY or DD-MON-YY).

The following query uses TO_DATE() to convert the strings 04-JUL-2010 and 04-JUL-10 to the date July 4, 2010;

SELECT TO_DATE('04-JUL-2010'), TO_DATE('04-JUL-10')
FROM dual;

TO_DATE('04-JUL-20	TO_DATE('04-JUL-10
04-JUL-10	04-JUL-10

The following query uses TO_DATE() to convert the string July 4, 2010 to a date, passing the format string MONTH DD, YYYY to TO_DATE():

SELECT TO_DATE('July 4, 2010', 'MONTH DD, YYYY')
FROM dual;

```
TO_DATE('
---------
04-JUL-10
```

SELECT TO_DATE('07/11/2010', 'DD,MM,YYYY')
FROM dual;

TO_DATE('07/11/201
07-NOV-10

Regular Expression Functions

These functions allow you to search for a pattern of characters in a string.
The regular expression contains a number of *metacharacters*.

In following example, ^, [5-8], and $ are the metacharacters; ^ matches the beginning position of a string; [5-8] matches characters between 5 and 8; $ matches the end position of a string.

REGEXP_LIKE()

You use REGEXP_LIKE(*x*, *pattern* [, *match_option*]) to search *x* for the regular expression defined in the *pattern* parameter

SELECT customer_id, first_name, last_name, dob
FROM customers
WHERE REGEXP_LIKE(TO_CHAR(dob, 'YYYY'), '^196[5-8]$');

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB
1	John	Brown	01-JAN-65
2	Cynthia	Green	05-FEB-68

The next query retrieves customers whose first name starts with J or j.

```
SELECT customer_id, first_name, last_name, dob
FROM customers
WHERE REGEXP_LIKE(first_name, '^j', 'i');
CUSTOMER_ID FIRST_NAME LAST_NAME DOB
----------- ---------- ---------- ---------
1 John Brown 01-JAN-65
```

Date functions

SYSDATE

It returns current date and time.

```
select  sysdate
from dual;

SYSDATE
19-MAY-10
```

By default the database uses the format DD-MON-YYYY to represent a date.

systimestamp

This function returns system date,including fractional seconds and timezone of the database server.

```
select systimestamp
from dual;
```

SYSTIMESTAMP
19-MAY-10 09.44.31.890000 PM +05:30

CURRENT_DATE

It is similar to SYSDATE, however it returns the current date in the session timezone where as SYSDATE returns the current date in the database server's timezone.

```
select CURRENT_DATE
from dual;
```

CURRENT_DATE
19-MAY-10

Note:- A time zone is an offset from the time in Greenwich, England. The time in Greenwich was once known as Greenwich Mean Time (GMT), but is now known as Coordinated Universal Time (UTC)

CURRENT_TIMESTAMP
It returns the current date and time in the session time zone. It is similar to systimestamp. However systimestamp returns the current date and time in the database server timezone where as CURRENT_TIMESTAMP returns the current date and time in the session time zone.

select CURRENT_TIMESTAMP
FROM DUAL;

CURRENT_TIMESTAMP
20-MAY-10 01.01.02.000000 AM +05:30

SESSIONTIMEZONE

This function returns the value of the timezone of the current session.
From the output of the following query it can be seen that the session is running from a system with timezone +05:30 i.e. it is 5hrs and 30 minutes ahead of UTC.

select SESSIONTIMEZONE
from dual;

SESSIONTIMEZONE
+05:30

ADD_MONTHS()

ADD_MONTHS(*x*, *y*) returns the result of adding *y* months to *x*. If *y* is negative, then *y* months are subtracted from *x*.

The following example adds 14 months to January 1, 2010:

SELECT ADD_MONTHS('01-JAN-2010', 14)
FROM dual;
ADD_MONTHS
('01-JAN01-MAR-11

MONTHS_BETWEEN(*x*, y)

Returns the number of months between *x* and *y*. If *x* appears before *y* on the calendar, the number returned is positive; otherwise the number is negative.

The following example displays the number of months between May 25, 2010, and January 15, 2010.

SELECT MONTHS_BETWEEN('25-MAY-2010', '15-JAN-2010')
FROM dual;
MONTHS_BETWEEN('25-MAY-2010','15-JAN-2010')
4.32258065

Notice that since the later date (May 25, 2008) appears first, the result returned is a positive number:

LAST_DAY()

LAST_DAY(*x*) returns the date of the last day of the month part of *x*. The following example displays the last date in January 2010:

SELECT LAST_DAY('01-JAN-2010')
FROM dual;

LAST_DAY('01-JAN-2
31-JAN-10

NEXT_DAY()

NEXT_DAY(*x*, *day*) returns the date of the next day following *x*; you specify *day* as a literal string (SATURDAY, for example).

The following example displays the date of the next Saturday after 15th May 2010.

SELECT NEXT_DAY('15-MAY-2010', 'SATURDAY')
FROM dual;
NEXT_DAY
('15-MAY-222-MAY-10

ROUND()

ROUND(*x* [, *unit*]) rounds *x*, by default, to the beginning of the nearest day. If you supply an optional *unit* string, *x* is rounded to that unit; for example, YYYY rounds *x* to the first day of the nearest year.

SELECT ROUND(TO_DATE('25-OCT-2008'), 'YYYY')
FROM dual;

```
ROUND(TO_
---------
01-JAN-09

SELECT ROUND(TO_DATE('25-MAY-2008'), 'MM')
FROM dual;
ROUND(TO_
---------
01-JUN-08
```

TRUNC()

TRUNC(*x* [, *unit*]) truncates *x*. By default, *x* is truncated to the beginning of the day. If you supply an optional *unit* string, *x* is truncated to that unit; for example, MM truncates *x* to the first day in the month.

The following example uses TRUNC() to truncate May 25, 2008, to the first day in the year, which is January 1, 2008:

```
SELECT TRUNC(TO_DATE('25-MAY-2008'), 'YYYY')
FROM dual;
TRUNC(TO_
---------
01-JAN-08
```

The next example truncates May 25, 2008, to the first day in the month, which is May 1, 2008:

```
SELECT TRUNC(TO_DATE('25-MAY-2008'), 'MM')
FROM dual;
TRUNC(TO_
---------
01-MAY-08
```

Arithmetic with Dates

We can perform calculations using arithmetic operators such as addition and subtraction. You can add or subtract number constants as well as dates.

```
SELECT  SYSDATE, SYSDATE+5, SYSDATE -5
FROM dual;
```

SYSDATE	SYSDATE+5	SYSDATE-5
21-MAY-10	26-MAY-10	16-MAY-10

You can add hours to date by dividing the number of hours by 24. Consider the following query.

SELECT SYSDATE, SYSDATE+30/24, SYSDATE - 30/24
FROM dual;

SYSDATE	SYSDATE+30/24	SYSDATE-30/24
21-MAY-10	23-MAY-10	20-MAY-10

General Functions

NVL()

You use NVL() to convert a null value to another value. NVL(*x*, *value*) returns *value* if *x* is null; otherwise *x* is returned.

Let us calculate the annual salary of all employees.

select ename,comm,sal, (sal*12)+comm   "annual Salary"
from emp;

NAME	COMM	SAL	annual Salary
SMITH		800	
ALLEN	300	1600	19500
WARD	500	1250	15500
JONES		2975	
MARTIN	1400	1250	16400
BLAKE		2850	
CLARK		2450	
SCOTT		3000	
KING		5000	
TURNER	0	1500	18000
ADAMS		1100	
JAMES		950	
FORD		3000	
MILLER		1300	

14 rows selected.

A expression evaluates to null whenever a value in it evaluates to null. That's why few of employees annual salary are missing in this calculation.
Now we can tackle the problem using NVL() by providing a value say 0.

**select ename,comm,sal, (sal*12)+NVL(comm,0) "annual Salary"
from emp;**

ENAME	COMM	SAL	annual Salary
SMITH		800	9600
ALLEN	300	1600	19500
WARD	500	1250	15500
JONES		2975	35700
MARTIN	1400	1250	16400
BLAKE		2850	34200
CLARK		2450	29400
SCOTT		3000	36000
KING		5000	60000
TURNER	0	1500	18000
ADAMS		1100	13200
JAMES		950	11400
FORD		3000	36000
MILLER		1300	15600

14 rows selected.

The following query retrieves the customer_id and phone columns from the customers table. Null values in the phone column are converted to the string Unknown Phone Number by NVL():

```
SELECT customer_id, NVL(phone, 'Unknown Phone Number')
FROM customers;
CUSTOMER_ID NVL(PHONE,'UNKNOWNPH
----------- --------------------
1 800-555-1211
2 800-555-1212
3 800-555-1213
4 800-555-1214
5 Unknown Phone Number
```

NVL2()

NVL2(*x*, *value1*, *value2*) returns *value1* if *x* is not null; otherwise *value2* is returned.

```
SELECT customer_id, NVL2(phone, 'Known', 'Unknown')
FROM customers;
```

```
CUSTOMER_ID NVL2(PH
----------- -------
1 Known
2 Known
3 Known
4 Known
5 Unknown
```

NULLIF()

This function compares exp1 and exp2 and returns null if they are equal. If they are not equal it returns expr1

SELECT NULLIF(500,300), NULLIF(500,500)
from dual;

NULLIF(500,300)	NULLIF(500,500)
500	

GREATEST()

This function returns greatest expression in the arguments.

SELECT greatest(10,7,-1)
from dual;

GREATEST(10,7,-1)
10

LEAST()

This function returns least expression in the arguments.

SELECT least(10,7,-1)
from dual;

LEAST(10,7,-1)
-1

USER()

This function returns the name of the session user.

SELECT user
from dual;

```
USER
--------------------
SCOTT
```

Group or Aggregate Functions

The functions you've seen so far operate on a single row at a time and return one row of output for each input row. The aggregate functions, operate on a group of rows and return one row of output.

AVG()

You use AVG(*x*) to get the average value of *x*. The following query gets the average price of the products; notice that the price column from the products table is passed to the AVG() function:

```
SELECT AVG(price)
FROM products;
AVG(PRICE)
----------
19.7308333
```

You can use the DISTINCT keyword to exclude identical values from a computation. For example, the following query uses the DISTINCT keyword to exclude identical values in the price column when computing the average using AVG():

```
SELECT AVG(DISTINCT price)
FROM products;
AVG(DISTINCTPRICE)
-----------------
20.2981818
```

Notice that the average in this example is slightly higher than the average returned by the first query in this section. This is because the value for product #12 (13.49) in the price column is the same as the value for product #7; it is considered a duplicate and excluded from the computation performed by AVG().

COUNT()

This function finds number of rows, where expr evaluates to something other than null. COUNT(*) counts all selected rows including duplicates and rows with nulls. Remember if you use ALL options, null will not be considered,rather this option will include duplicate values.

```
SELECT  count(PRODUCT_TYPE_ID), count (all PRODUCT_TYPE_ID), count(distinct
PRODUCT_TYPE_ID)
FROM products;
```

COUNT(PRODUCT_TYPE_ID)	COUNT(ALLPRODUCT_TYPE_ID)	COUNT(DISTINCTPRODUCT_TYPE_ID)
11	11	4

SELECT count(*)
FROM products;

COUNT(*)

<u>Using Groups of Rows with Aggregate Functions</u>

- Use the GROUP BY clause to group rows into blocks
- Use COUNT(ROWID) to get the number of rows in each block.

SELECT product_type_id, COUNT(ROWID)
FROM products
GROUP BY product_type_id
ORDER BY product_type_id;

PRODUCT_TYPE_ID	COUNT(ROWID)
1	2
2	4
3	2
4	3

SUM()

SUM(*x*) adds all the values in *x* and returns the total. The following query gets the sum of the price column from the products table using SUM():

SELECT SUM(price)
FROM products;

```
SUM(PRICE)
----------
236.77
```

MAX() and MIN()

You use MAX(*x*) and MIN(*x*) to get the maximum and minimum values for *x*.

SELECT MAX(price), MIN(price)

```
FROM products;
MAX(PRICE) MIN(PRICE)
---------- ----------
49.99             10.99
```

Chapter 4

After completing this chapter, you will be able to understand:

- ✓ **SET OPERATORS**
- ✓ **UNION**
- ✓ **UNION ALL**
- ✓ **INTERSECT**
- ✓ **MINUS**
- ✓ **DISPLAYING DATA FROM MULTIPLE TABLES**
- ✓ **EQUI JOINS**
- ✓ **NONEQUI JOINS**
- ✓ **OUTER JOINS**
- ✓ **SELF JOINS**
- ✓ **SUBQURIES**
- ✓ **CASE EXPRESSION**

SET OPERATORS

You can combine multiple queries using the set operators UNION, UNION ALL, INTERSECT and MINUS to get a single result set. All set operators have equal precedence. If a SQL statement contains multiple set operators, oracle evaluates them from left to right.

UNION	**Rows of first query plus rows of second query, less duplicate rows**
UNION ALL	**Rows of first query plus rows of second query, including duplicate rows**
INTERSECT	**Common rows from all the queries**
MINUS	**Rows unique to the first query**

UNION

The UNION OPERATOR is used to combine two or more queries to obtain a single results set that contains all the rows retrieved by the components queries.
UNION OPERATORS does not display duplicate rows.

Look the following programmes displaying all the job titles in department 20 and 30 using the UNION OPERATORS given below and consider the output:

```
SQL> select job from emp
  2  where deptno=20
  3  UNION
  4  select job from emp
  5  where deptno=30;


JOB
---------
ANALYST
CLERK
MANAGER
SALESMAN
```

Here you can see the job title between 20 to 30 of department. Note there is no duplicate rows been displayed.
Keep the following things in mind while using UNION operator:

- Don't use the ORDER BY clause in the component queries. However you can order the final result of the entire UNION OPERATIONS. That is you can use the ORDER BY clause in the last component queries.
- The number of the column and their data type of all the component queries should be the same.
- The column used for ordering should be referenced by the position number rather than name. The reason being that SQL does not require the column names retrieved by the component queries to be equal.

Here is another example using the ORDER BY clause:

```
SQL> select empno empid,ename from emp
  2  where deptno=10
  3  UNION
  4  select empno,ename from emp
  5  where deptno=30
  6  order by 1;


     EMPID ENAME
---------- ----------
      7499 ALLEN
      7521 WARD
      7654 MARTIN
      7698 BLAKE
      7782 CLARK
      7839 KING
      7844 TURNER
      7900 JAMES
      7934 MILLER

9 rows selected.
```

So by using the ORDER BY clause the rows are sorted by employee number. Here the first component statement we have used an alias for empno, which makes it different from the second component query.

UNION ALL

The UNION ALL OPERATOR is similar to UNION . However , UNION ALL operator does not suppress the display of duplicate rows.
Look the following programmes displaying all the job titles in department 20 and 30 using the UNION ALL OPERATORS :

```
SQL> select job from emp
  2  where deptno=20
  3  UNION ALL
  4  select job from emp
  5  where deptno=30;


JOB
---------
CLERK
MANAGER
ANALYST
CLERK
ANALYST
SALESMAN
SALESMAN
SALESMAN
MANAGER
SALESMAN
CLERK

11 rows selected.
```

Here you can see the job title between 20 to 30 of department. Note there the duplicate rows are also been displayed.

INTERSECT

The INTERSECT OPERATOR is used to combine two or more queries to obtain a single queries result set that contain only the common rows retrieved by the component queries. Look the following programmes displaying all the job titles in department 20 and 30 using the INTERSECT OPERATORS :

```
SQL> select job from emp
  2  where deptno=20
  3  INTERSECT
  4  select job from emp
  5  where deptno=30;


JOB
---------
CLERK
MANAGER
```

Here you can see the job titles common to both the department 20 and 30.

MINUS

The MINUS OPERATOR is used to combine two or more queries to obtain a single result set that contain the rows unique to the first query.
Let us find out the job titles unique to department 20

```
SQL> select job from emp
  2  where deptno=20
  3  MINUS
  4  select job from emp
  5  where deptno IN(10,30);

JOB
---------
ANALYST
```

Here you can see there is an only job title that is unique to department 20.

DISPLAYING DATA FROM MULTIPLE TABLES

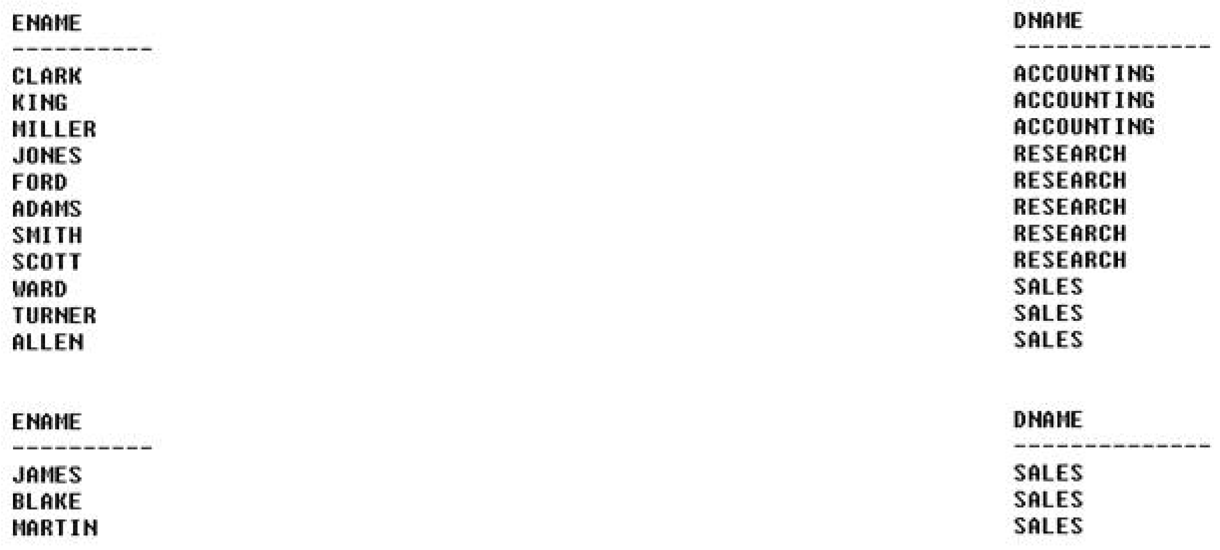
One of the most important features of SQL is the ability to define relationship between multiple tables and draw information from them in terms of these relationships, all using a single statement. In order to retrieve information from multiple tables we use joins.   We specify join condition in WHERE clause.

The following are the different method of joins:

- **EQUIJOIN**
- **NON-EQUIJOIN**
- **OUTER JOIN**
- **SELF-JOIN**

- **EQUI JOIN**

Let us suppose, we need to display employee names and department names .That is, we need to write a query, which will display the result that looks similar to the following:



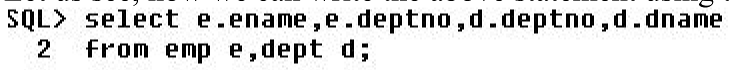
```
ENAME                                                  DNAME
----------                                             --------------
CLARK                                                  ACCOUNTING
KING                                                   ACCOUNTING
MILLER                                                 ACCOUNTING
JONES                                                  RESEARCH
FORD                                                   RESEARCH
ADAMS                                                  RESEARCH
SMITH                                                  RESEARCH
SCOTT                                                  RESEARCH
WARD                                                   SALES
TURNER                                                 SALES
ALLEN                                                  SALES

ENAME                                                  DNAME
----------                                             --------------
JAMES                                                  SALES
BLAKE                                                  SALES
MARTIN                                                 SALES
```

From the above it can be seen that employee names and their department names are displayed together. We know that ename column is present in the emp table and dname column is present in the dept table. Then, how do we retrieve data from multiple tables?
Consider the statement below:

```
SQL> SELECT  emp.ename ,   emp.deptno,  dept.deptno, dept.dname
  2    FROM  emp ,  dept;
```

Here, you notice that in the FROM clause we have specified two tables, namely emp and dept. Also, the column names are qualified by their corresponding table names. You must qualify the column name with the table name (or table alias) if it is common to two or more tables specified in the FROM clause. However, if the column is unique, qualifying it with table name or table alias is optional.

Let us see, how we can write the above statement using table alias:



```
SQL> select e.ename,e.deptno,d.deptno,d.dname
  2  from emp e,dept d;
```

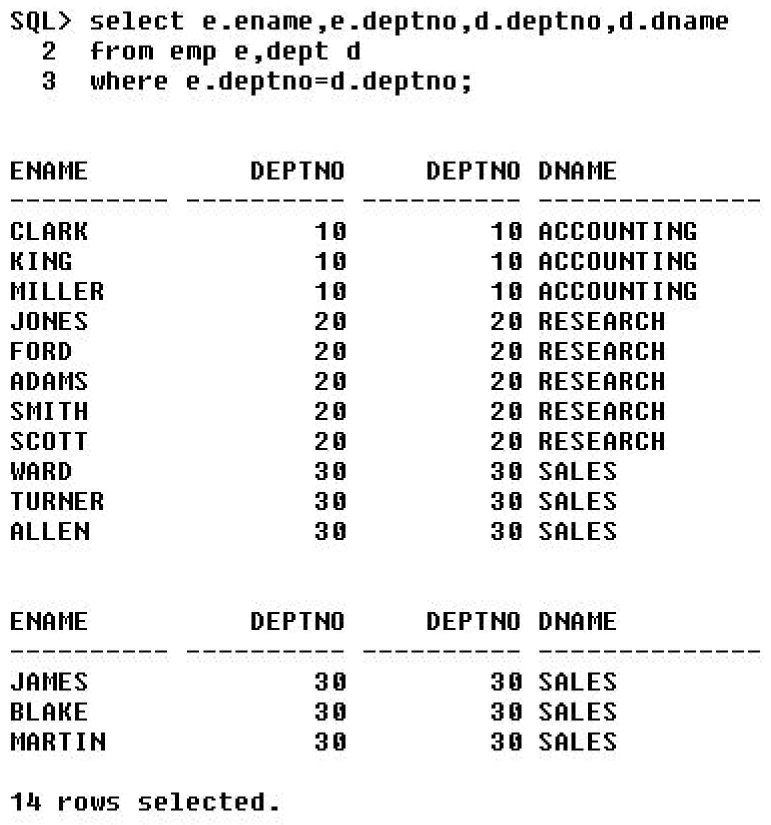
Here you can see, how table alias can be used; e for emp and d for dept. The alias can be up to 30 characters long. However,use short aliases as it makes easy to write SQL statements.
Now, as you have understood how to specify multiple tables in the FROM clause and specify columns thereof in the SELECT clause, let us see what will be the output of the above statement. Look at the following:

```
ENAME          DEPTNO     DEPTNO DNAME
---------- ---------- ---------- --------------
SMITH              20         10 ACCOUNTING
ALLEN              30         10 ACCOUNTING
WARD               30         10 ACCOUNTING
JONES              20         10 ACCOUNTING
MARTIN             30         10 ACCOUNTING
BLAKE              30         10 ACCOUNTING
CLARK              10         10 ACCOUNTING
SCOTT              20         10 ACCOUNTING
KING               10         10 ACCOUNTING
TURNER             30         10 ACCOUNTING
....
....
JAMES              30         10 ACCOUNTING
FORD               20         10 ACCOUNTING
MILLER             10         10 ACCOUNTING
SMITH              20         20 RESEARCH
ALLEN              30         20 RESEARCH
WARD               30         20 RESEARCH
JONES              20         20 RESEARCH
MARTIN             30         20 RESEARCH
```

56 rows selected.

Here you can see how 56 rows are been selected. If the two table specified in the FROM clause each row of the second table and the result is set displayed with the columns that you specify in the SELECT clause. In the above case, as we have 14 rows in the emp table and 4 rows in the dept table, we get 56 rows in the result set when we specify both the tables in the FROM clause. In fact, we can specify any number of tables in the FROM clause .Suppose we specify three tables in the FROM clause having m ,n and p rows respectively ,then the result set would contain m*n*p rows .we call this operation as Cartesian product.

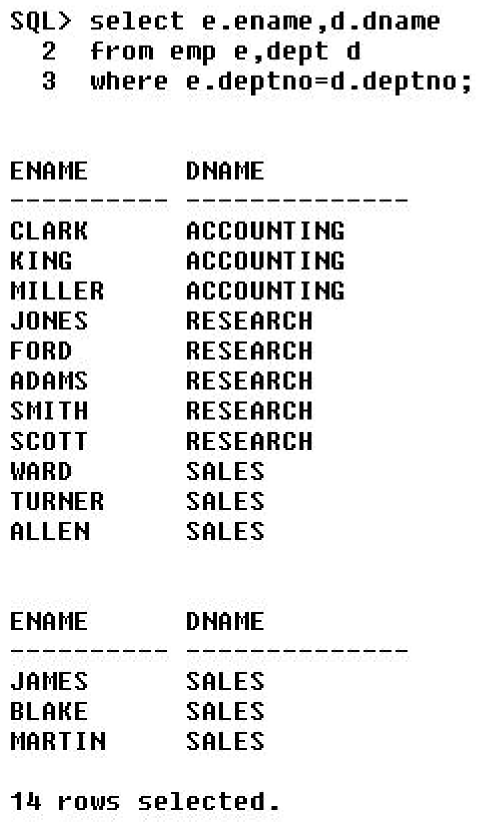
The Cartesian product, as can be seen in the above example does not give us any meaningful Information. In order to obtain any meaningful information, we need to specify one or more join conditions using the WHERE clause. If you observe the above-mentioned example, you will find that those rows having deptno of emp table equal to deptno of dept table, give us useful information. That is, we can find the employee names with their department names.



```
SQL> select e.ename,e.deptno,d.deptno,d.dname
  2  from emp e,dept d
  3  where e.deptno=d.deptno;

ENAME          DEPTNO     DEPTNO DNAME
---------- ---------- ---------- --------------
CLARK              10         10 ACCOUNTING
KING               10         10 ACCOUNTING
MILLER             10         10 ACCOUNTING
JONES              20         20 RESEARCH
FORD               20         20 RESEARCH
ADAMS              20         20 RESEARCH
SMITH              20         20 RESEARCH
SCOTT              20         20 RESEARCH
WARD               30         30 SALES
TURNER             30         30 SALES
ALLEN              30         30 SALES

ENAME          DEPTNO     DEPTNO DNAME
---------- ---------- ---------- --------------
JAMES              30         30 SALES
BLAKE              30         30 SALES
MARTIN             30         30 SALES

14 rows selected.
```

Here , you find the employee names and their departments are displayed . Notice the join condition specified in the WHERE clause. Here we have used the '=' operator in the join condition to compare the common columns of the relevant tables . This type of join is called equijoin.

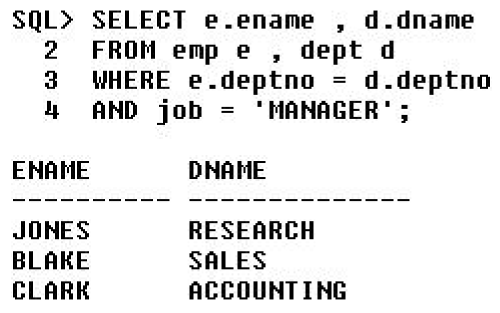
Now let us see the following query without using the e.deptno and d.deptno in the SELECT clause. See the following output :



```
SQL> select e.ename,d.dname
  2  from emp e,dept d
  3  where e.deptno=d.deptno;

ENAME      DNAME
---------- --------------
CLARK      ACCOUNTING
KING       ACCOUNTING
MILLER     ACCOUNTING
JONES      RESEARCH
FORD       RESEARCH
ADAMS      RESEARCH
SMITH      RESEARCH
SCOTT      RESEARCH
WARD       SALES
TURNER     SALES
ALLEN      SALES

ENAME      DNAME
---------- --------------
JAMES      SALES
BLAKE      SALES
MARTIN     SALES

14 rows selected.
```

Here you see the employee name and their department are displayed and we have not included the department numbers in the SELECT clause.

We can specify additional search conditions using AND operator in the WHERE clause. Let us display the name and department name of all employees whose job title is 'MANAGER' .It can be seen in the following example:



```
SQL> SELECT e.ename , d.dname
  2  FROM emp e , dept d
  3  WHERE e.deptno = d.deptno
  4  AND job = 'MANAGER';

ENAME      DNAME
---------- --------------
JONES      RESEARCH
BLAKE      SALES
CLARK      ACCOUNTING
```

Here you find that employee names and department names are displayed only for the employees with job title MANAGER.(note: here table aliases is also been specified for performing reasons). Remember, the columns that are specified in the WHERE clause to perform an equijoin need not have the same name . However, keep in mind that those columns should represent the values of the same attribute.

Very often you will find that these type of joins involve primary and foreign keys of the tables involved, although it is not a necessary condition for equijoin operation.
NOTE: Equijoins are also called simple joins or inner joins.

- **NON-EQUI JOIN**

  Look at the contents of the SALGRADE table shown below:

```
     GRADE      LOSAL      HISAL
---------- ---------- ----------
         1        700       1200
         2       1201       1400
         3       1401       2000
         4       2001       3000
         5       3001       9999
```

  Now suppose we want to display employee names, salaries and grade for all employees. We find grade in the SALGRADE table, and no column in the EMP table corresponds directly to a column in the SALGRADE table . Therefore, we cannot perform an equijoin here. Here we need to relate the EMP table and the SALGRADE table using non-equijoin. You can find the grade of an employee by comparing his salary with losal and hisal.

Look at the statement below and the following output:

```
SQL> select e.ename,e.sal,s.grade
  2  from emp e,salgrade s
  3  where e.sal BETWEEN s.losal AND s.hisal;


ENAME             SAL      GRADE
---------- ---------- ----------
SMITH             800          1
JAMES             950          1
ADAMS            1100          1
WARD             1250          2
MARTIN           1250          2
MILLER           1300          2
TURNER           1500          3
ALLEN            1600          3
CLARK            2450          4
BLAKE            2850          4
JONES            2975          4

ENAME             SAL      GRADE
---------- ---------- ----------
SCOTT            3000          4
FORD             3000          4
KING             5000          5

14 rows selected.
```

Here , you can see , a non-equijoin has been created to evaluated an employee's salary grade and the relationship is obtained using an operator other than equal(=).
In fact, we can retrieve information from any number of tables by creating the appropriate joins. Suppose, we need to display employee names, salaries, grades and department names . In order to do so, we need to retrieve information from three tables, namely EMP, DEPT and SALGRADE.

Look at the following:
```
SQL> select e.ename,e.sal,s.grade,d.dname
  2  from emp e,dept d,salgrade s
  3  where e.deptno=d.deptno
  4  AND e.sal BETWEEN s.losal AND s.hisal;


ENAME             SAL      GRADE DNAME
---------- ---------- ---------- --------------
SMITH             800          1 RESEARCH
JAMES             950          1 SALES
ADAMS            1100          1 RESEARCH
WARD             1250          2 SALES
MARTIN           1250          2 SALES
MILLER           1300          2 ACCOUNTING
TURNER           1500          3 SALES
ALLEN            1600          3 SALES
CLARK            2450          4 ACCOUNTING
BLAKE            2850          4 SALES
JONES            2975          4 RESEARCH

ENAME             SAL      GRADE DNAME
---------- ---------- ---------- --------------
SCOTT            3000          4 RESEARCH
FORD             3000          4 RESEARCH
KING             5000          5 ACCOUNTING

14 rows selected.
```

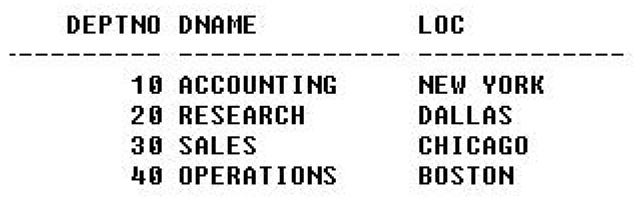
Here you can see that we have specified two join conditions in order to join three tables. The first condition creates an equijoin between EMP and DEPT tables and the second condition creates a non-equijoin between EMP and SALGRADE tables.

NOTE: Remember, in order to join n tables together, you need a minimum of (n-1) join conditions. Always include valid join condition(s) to avoid a Cartesian product.

- **OUTER JOINS**

  If a row does not satisfy a join condition , the row will not appear in the query result. For example , the department OPERATIONS does not appear in the equijoin condition of EMP and DEPT tables . This happens because no one works in that department. We can use outer join to return the row(s) with no direct match.

Look at the contents of the DEPT tables shown below:



```
    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
```

Here you see, the department number of OPERATIONS is 40 and in the table EMP there is no record with department number 40. That is why the row of DEPT table containing OPERATIONS does not meet the join condition and consequently is not displayed. Under what condition that row would have been displayed? As EMP table does not have any row with department number 40.In this context, we can say that the EMP table is deficient . The solution is, use the OUTER JOIN with the deficient table to display the rows that do not meet the join condition .The outer join operator is the plus sign with parentheses,(+).

select ename, d.deptno, dname
from emp e
RIGHT OUTER JOIN dept d
ON e.deptno=d.deptno
where d.deptno > 10; (e.deptno will not work, as it doesn't have 40 in the table)

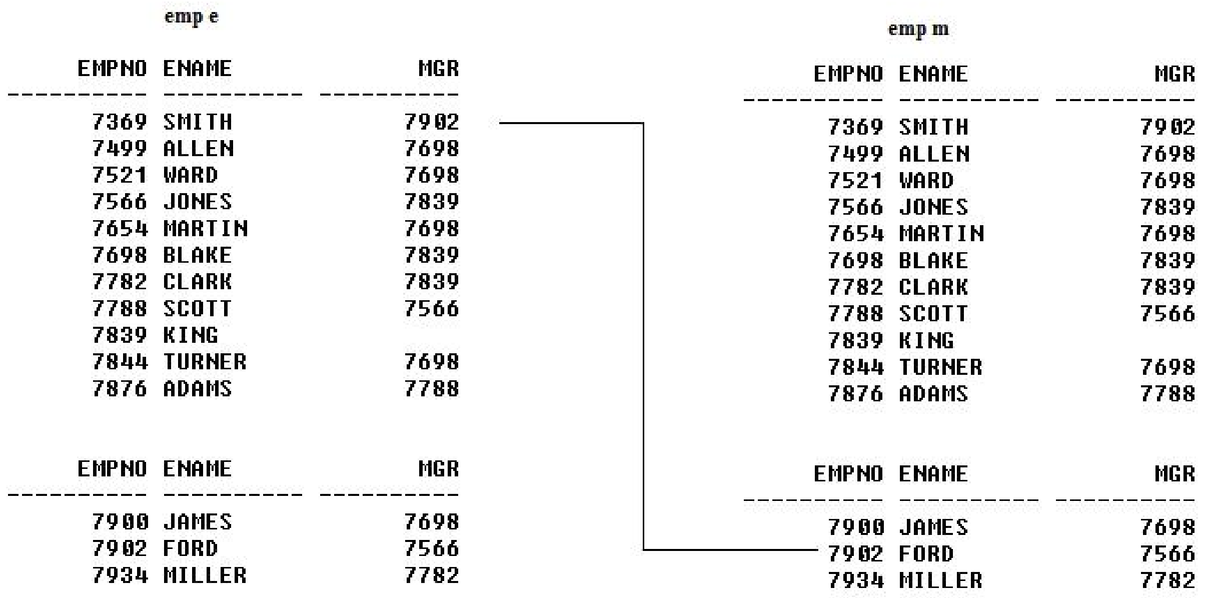
See the following example:

```
SQL> select e.ename,d.deptno,d.dname
  2  from emp e,dept d
  3  where e.deptno(+)=d.deptno;

ENAME          DEPTNO DNAME
---------- ---------- --------------
CLARK              10 ACCOUNTING
KING               10 ACCOUNTING
MILLER             10 ACCOUNTING
JONES              20 RESEARCH
FORD               20 RESEARCH
ADAMS              20 RESEARCH
SMITH              20 RESEARCH
SCOTT              20 RESEARCH
WARD               30 SALES
TURNER             30 SALES
ALLEN              30 SALES

ENAME          DEPTNO DNAME
---------- ---------- --------------
JAMES              30 SALES
BLAKE              30 SALES
MARTIN             30 SALES
                   40 OPERATIONS

15 rows selected.
```

Notice the use of the outer join operator (+). Here, the output displays numbers and names for all the departments. The OPERATION department, which does not have any employees , is also displayed.

- **SELF JOINS**

Sometimes, you need to join the table with itself and the operation is called a self join. Suppose we have to find the name of each employee manager. How do we processed to solve it. Go through the following two identical sets of employee date. Let us call the first set e and the second set m.



emp e

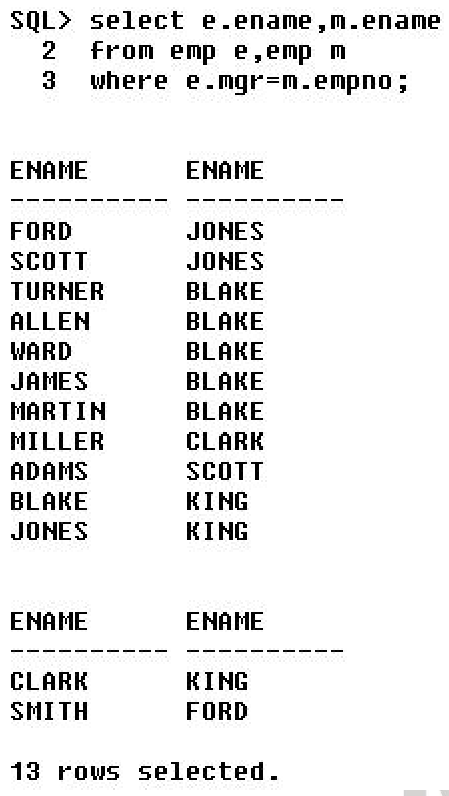
```
     EMPNO ENAME             MGR
---------- ---------- ----------
      7369 SMITH            7902
      7499 ALLEN            7698
      7521 WARD             7698
      7566 JONES            7839
      7654 MARTIN           7698
      7698 BLAKE            7839
      7782 CLARK            7839
      7788 SCOTT            7566
      7839 KING
      7844 TURNER           7698
      7876 ADAMS            7788


     EMPNO ENAME             MGR
---------- ---------- ----------
      7900 JAMES            7698
      7902 FORD             7566
      7934 MILLER           7782
```

emp m

```
     EMPNO ENAME             MGR
---------- ---------- ----------
      7369 SMITH            7902
      7499 ALLEN            7698
      7521 WARD             7698
      7566 JONES            7839
      7654 MARTIN           7698
      7698 BLAKE            7839
      7782 CLARK            7839
      7788 SCOTT            7566
      7839 KING
      7844 TURNER           7698
      7876 ADAMS            7788


     EMPNO ENAME             MGR
---------- ---------- ----------
      7900 JAMES            7698
      7902 FORD             7566
      7934 MILLER           7782
```

Who is the manager of SMITH? You look through emp e to find the mgr for SMITH, that is 7902. Again you look through the emp m to find the employee name with empno 7902. You find the FROD's employee number 7902. So FORD is SMITH's manager. In effect, to find out the manager of an employee. You look in the same table twice. First time you look in the emp table to find the smith in the ename column and mgr value of 7902. The second time you can look in the empno column to find the 7902 and the ename column to find the FORD.

Now let us write the query to create self join in the EMP table and display employee names and their managers names. See the following statement and its output:



```
SQL> select e.ename,m.ename
  2  from emp e,emp m
  3  where e.mgr=m.empno;

ENAME      ENAME
---------- ----------
FORD       JONES
SCOTT      JONES
TURNER     BLAKE
ALLEN      BLAKE
WARD       BLAKE
JAMES      BLAKE
MARTIN     BLAKE
MILLER     CLARK
ADAMS      SCOTT
BLAKE      KING
JONES      KING

ENAME      ENAME
---------- ----------
CLARK      KING
SMITH      FORD

13 rows selected.
```

Here you find the EMP table is joined to itself and the WHERE clause contain the join condition that means 'where an employee's manager number matches the employee number for the manager'. Notice the above output displays 13 rows only whereas the EMP has 14 row table. This happens because there is no manager for KING and consequently no corresponding match. However you can use the outer- join operator to display this missing row.

SUBQUERIES

A subquery is a SELECT statement that is nested within another SELECT statement and returns intermediate result .The result of the inner query is then used by the outer query or the main query. Using a subquery is equivalent to performing two sequential queries and using the result of the first as the search value in the second query.

Suppose, you want to write a query to find out who earns the highest salary.

To solve this problem, you need two queries: 1.one query to find out the maximum salary and 2.a second query to find out who earns that salary.

You can solve this problem by using subquery that is placing one query inside the other query. See the statement below and the following output:

```
SQL> select ename,sal
  2  from emp
  3  where sal=(SELECT max(sal)from emp);


ENAME             SAL
---------- ----------
KING             5000
```

Here , the inner query executes first and returns the maximum salary , which is 5000. This value is used in the WHERE clause of the outer query and the result is displayed.

Syntax:
SELECT select_list
FROM table
WHERE expr operator
(SELECT select _list
FROM table);

- In the syntax, operator includes a comparison operator such as >, = or IN .

Comparison operators fall into two classes:

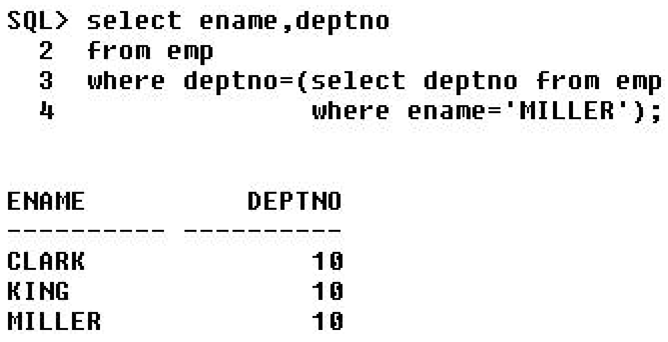
- Single-row comparison operators (> ,= ,>= ,< ,<> ,<=).
- Multiple-row comparison operators (IN, ANY, SOME, ALL). You can negate the meaning of any of these operators by prefixing it with NOT, for example, NOT IN, NOT ANY.
  You can place the subquery in the WHERE clause, HAVING clause or FROM clause.
  Subqueries can be nested up to 255 levels deep.

NOTE: The following things are to be kept in mind while using Subqueries:

- Place the subquery on the right side of the comparison operator.
- Enclose the subquery within parentheses.
- Do not specify an ORDER BY clause in a subquery because you can have only one ORDER BY clause for a SELECT statement, and if used it must be the last clause in the main SELECT statement.
- Use single-row comparison operators with single-row subqueries.
- Use multiple-row comparison operators with multiple-row subqueries.

- **<u>SINGLE-ROW SUBQUERY</u>**

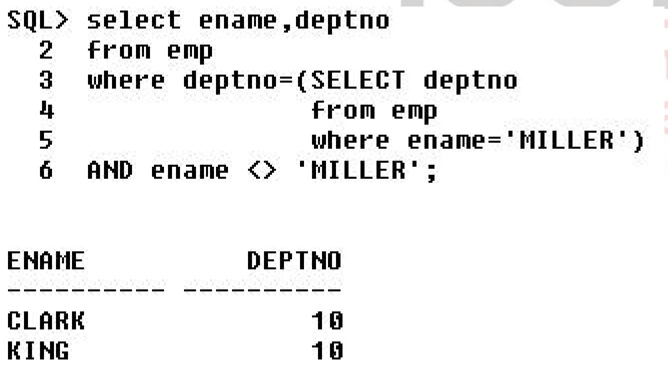
If the subquery returns only the row, then it is known as single-row subquery.
Let us see one example of single-row subquery. Let us display the employee belonging to the department of MILLER. See the following statement and its output:



```
SQL> select ename,deptno
  2  from emp
  3  where deptno=(select deptno from emp
  4                where ename='MILLER');


ENAME          DEPTNO
---------- ----------
CLARK              10
KING               10
MILLER             10
```

Here, the inner query returns the department number of MILLER and this value is used in the condition specified in the WHERE clause of the outer query. Finally, the result set is displayed. Notice that the result also has a row containing MILLER, which is obvious because MILLER works in his own department. If you do not want to display MILLER's information simply add another condition to restrict that row.

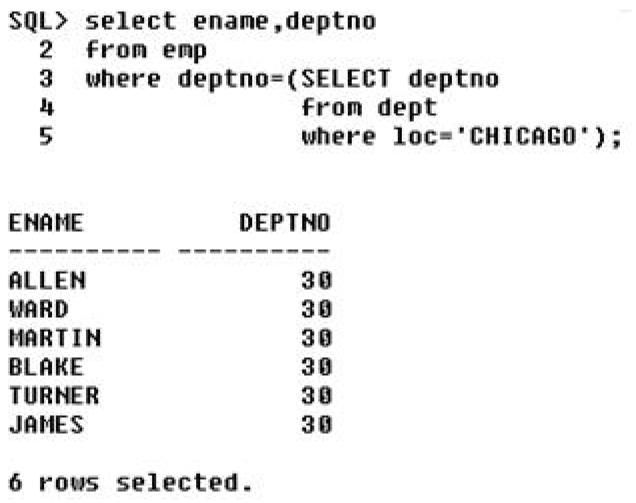
See the following example and consider the output:



```
SQL> select ename,deptno
  2  from emp
  3  where deptno=(SELECT deptno
  4                from emp
  5                where ename='MILLER')
  6  AND ename <> 'MILLER';


ENAME          DEPTNO
---------- ----------
CLARK              10
KING               10
```

Here you find no row containing MILLER.
The outer query and the inner query may refer todifferent tables . Suppose we need to display the names of all the employees whose depatment is located at CHICAGO.Here we writa a subquery that returns the department number corresponding to the location CHICAGO , and that department number is used in the main query to find out the employees.

See the following example:

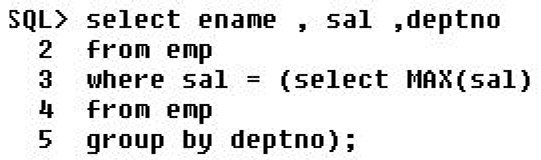


```
SQL> select ename,deptno
  2  from emp
  3  where deptno=(SELECT deptno
  4                from dept
  5                where loc='CHICAGO');

ENAME          DEPTNO
---------- ----------
ALLEN              30
WARD               30
MARTIN             30
BLAKE              30
TURNER             30
JAMES              30

6 rows selected.
```

Here the output displays the names of all the employees who are located in CHICAGO.

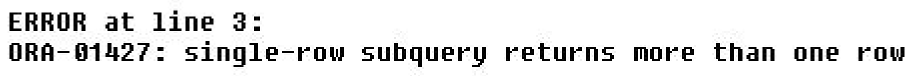
- Remember, whenever you see a single-row comparison operator, ensure that the subquery on its right returns only one row otherwise you will get a error.

  Look at the statement below



```
SQL> select ename , sal ,deptno
  2  from emp
  3  where sal = (select MAX(sal)
  4  from emp
  5  group by deptno);
```

Does the statement work? Observe the subquery. It contains a GROUP BYclause and therefore it returns the maximum salary for each department. As there are more than one department in the EMP table, this subquery returns multiple rows, and we are trying to compare these values using a single row comparison operator . So Oracle gives you the following error message when you execute this statement.



```
ERROR at line 3:
ORA-01427: single-row subquery returns more than one row
```

Hence, always consider the possible number of rows returned by the subquery and use the correct comparison operator. In order to deal with multiple rows returned by the subquery we use multiple-row comparison operators. This is discussed in the next section.

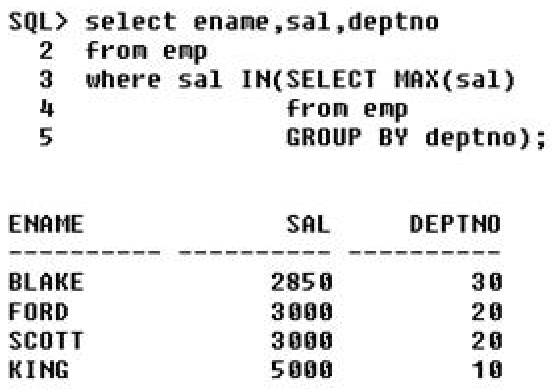
• MULTIPLE-ROW SUBQUERY

Subqueries that return more than one row are called multi-row subqueries. You use a multiple-row comparison operator with a multiple-row subquery. The multiple-row comparison operator expects one or more values.

- **IN OPERATOR**

Let us display the names and salaries of those employees whose salary matches with the maximum salary of any department. Here , we use a subquery that returns maximum salary for each department and we use the IN operator to compare this list of values in the outer query.

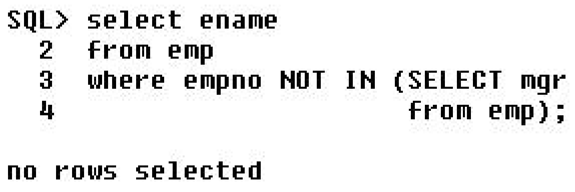
See the statement below and the following output:



```
SQL> select ename,sal,deptno
  2  from emp
  3  where sal IN(SELECT MAX(sal)
  4              from emp
  5              GROUP BY deptno);


ENAME             SAL     DEPTNO
---------- ---------- ----------
BLAKE            2850         30
FORD             3000         20
SCOTT            3000         20
KING             5000         10
```

Here the inner query is executed first and produces a result containing three rows: 5000, 3000 and 2850. The main query is then executed using the values to complete the search condition. The IN operator means equal to any member in the list.

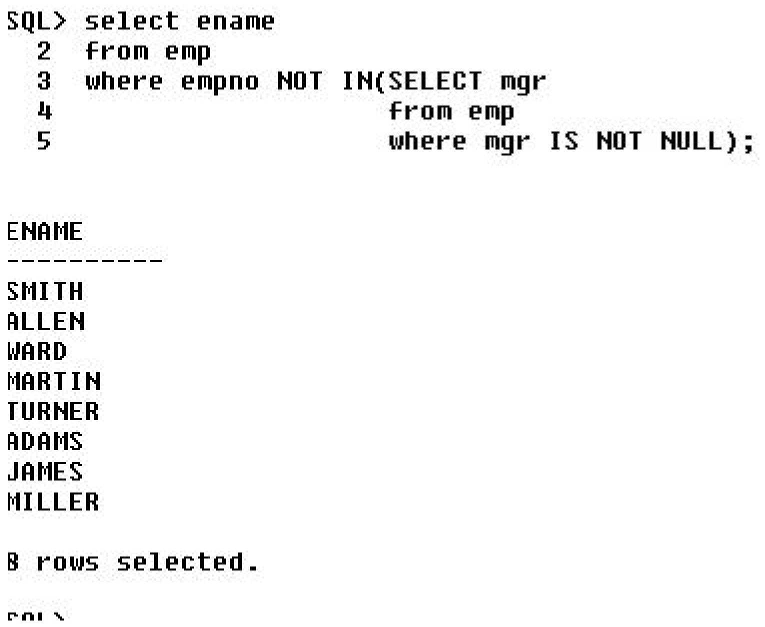
Now, suppose we want to display the names of those employees who do not have any subordinates. How do we find them? If an employee's empno is not present in the mgr column ,he is not a manager or in other words he does not have any subordinates. Let us see this programme given below and consider the output:



```
SQL> select ename
  2  from emp
  3  where empno NOT IN (SELECT mgr
  4                      from emp);

no rows selected
```

Here we find no rows are selected. but it doesn't means that all the employee have subordinates .As you know a comparison with null yields null and hence no rows will be selected. In the above example the list of values returned by the subquery contain a null value. However, we need not remember whether the list contains one or more null values because we can always eliminate them by adding a condition in the subquery.

See the following statement and its output:

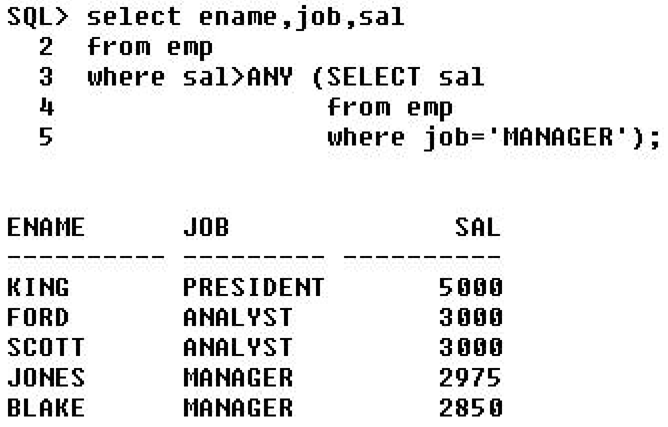


```
SQL> select ename
  2  from emp
  3  where empno NOT IN(SELECT mgr
  4                     from emp
  5                     where mgr IS NOT NULL);


ENAME
----------
SMITH
ALLEN
WARD
MARTIN
TURNER
ADAMS
JAMES
MILLER

8 rows selected.
```

Here you find the names of the employees who do not have subordinates.

ANY OPERATOR

The ANY operator compares a value to each value returned by the subquery. The SOME operator works in the same way the ANY operator does. Let us display the names, jobs and salaries of those employees whose salary is more than the salary of any employee with job title MANAGER. See the following example:



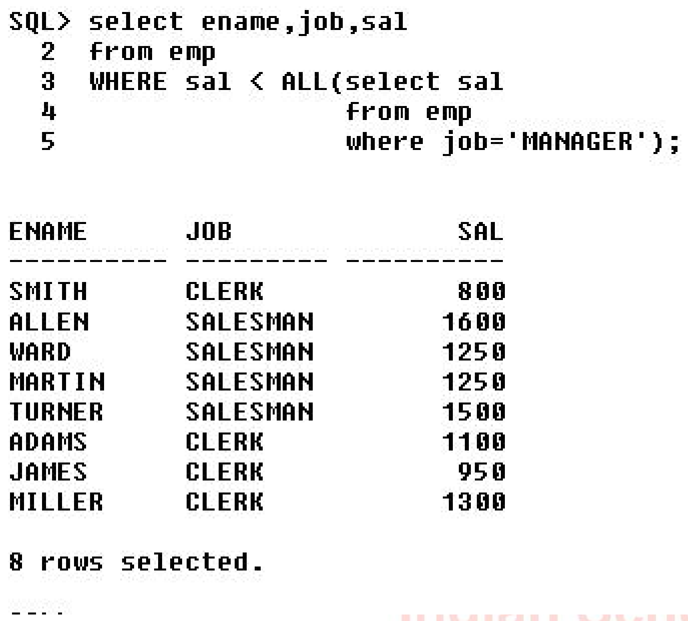
```
SQL> select ename,job,sal
  2  from emp
  3  where sal>ANY (SELECT sal
  4                 from emp
  5                 where job='MANAGER');


ENAME      JOB              SAL
---------- --------- ----------
KING       PRESIDENT       5000
FORD       ANALYST         3000
SCOTT      ANALYST         3000
JONES      MANAGER         2975
BLAKE      MANAGER         2850
```

Here you see that those employees whose salary is greater than that of the lowest paid employee with job title MANAGER are been displayed.

NOTE : >ANY means greater than the minimum value in the list and <ANY operators means less than the maximum value in the list.

ALL OPERATOR

The ALL operator compares the value to every given value by the subquery.
Let us the ALL operator to display the names, jobs and salaries of those employees whose salary is less than that of all the employees with job title MANAGER, given below and consider the output:

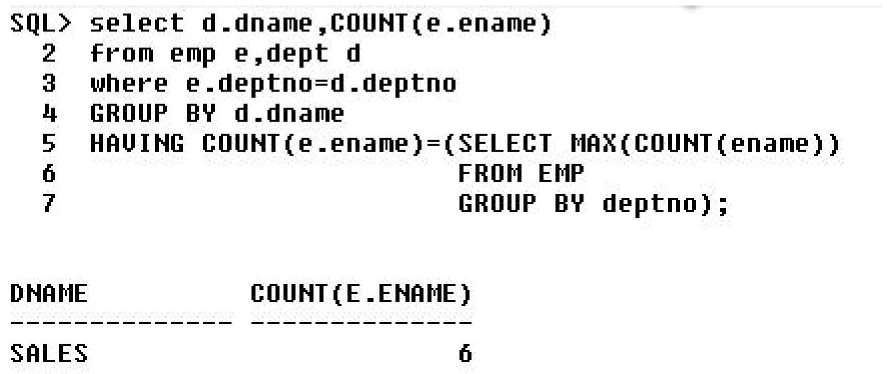


```
SQL> select ename,job,sal
  2  from emp
  3  WHERE sal < ALL(select sal
  4                  from emp
  5                  where job='MANAGER');


ENAME      JOB              SAL
---------- --------- ----------
SMITH      CLERK            800
ALLEN      SALESMAN        1600
WARD       SALESMAN        1250
MARTIN     SALESMAN        1250
TURNER     SALESMAN        1500
ADAMS      CLERK           1100
JAMES      CLERK            950
MILLER     CLERK           1300

8 rows selected.
```

Remember, < ALL means less than the minimum value in the value in the list and > ALL means greater than the maximum value in the list.

- **SUBQUERY IN THE HAVING CLAUSE**

You can use a subquery in the HAVING clause as well. Sometimes you need to restrict group result(s) by specifying a condition in the HAVING clause and the condition uses a value that is dynamically retrieved through a query. Suppose we need to write a query that displays the department name and the number of employees for the department having highest number of employees. Here we need to find number employees department wise and then restrict those group results where the employee count does not match with the highest value of the employee count.

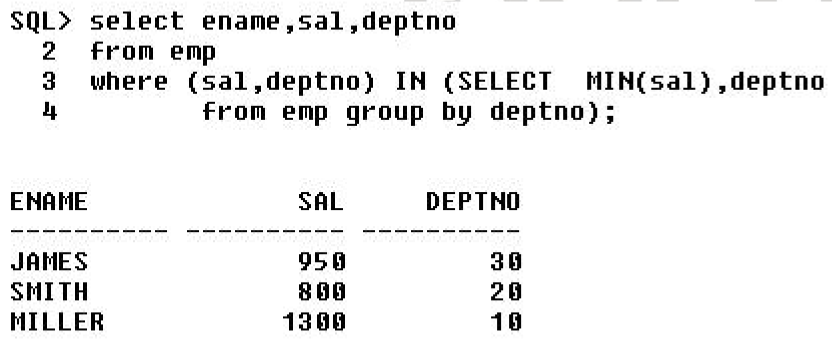


```
SQL> select d.dname,COUNT(e.ename)
     from emp e,dept d
     where e.deptno=d.deptno
     GROUP BY d.dname
     HAVING COUNT(e.ename)=(SELECT MAX(COUNT(ename))
                            FROM EMP
                            GROUP BY deptno);

DNAME          COUNT(E.ENAME)
-------------- --------------
SALES                       6
```

Here , you find that the SALES department has highest number of employees.

- **MULTIPLE COLUMN SUBQUERIES**

So far we have discussed single-row subqueries and multiple-row subqueries where only one column was compared in the WHERE clause or HAVING clause of the SELECT statement. You can compare two or more columns together using Multiple-row subquery.
Suppose you need to write a query to display the lowest earner of each department. Here you need to compare salary as well as department number both.
Look at the following:



```
SQL> select ename,sal,deptno
     from emp
     where (sal,deptno) IN (SELECT  MIN(sal),deptno
              from emp group by deptno);

ENAME             SAL     DEPTNO
---------- ---------- ----------
JAMES             950         30
SMITH             800         20
MILLER           1300         10
```

Here you find that the output displays department wise lowest earners. Notice the use of parentheses enclosing the columns on the left side of comparison operator.

EXITS OPERATOR

The EXITS operator is used to check for the existence of values. It takes a subquery as an argument and returns True if the subquery produces any output and it return false, if the subquery does not produce any output it return as non-null value(s).
Let us write a query that displays the list of all the employee who have at least one person reporting to them. See the following output:

```
SQL> select empno,ename,job,deptno
  2  from emp e1
  3  WHERE EXISTS (SELECT empno
  4                from emp e2
  5                WHERE e2.mgr=e1.empno);

     EMPNO ENAME      JOB           DEPTNO
---------- ---------- --------- ----------
      7566 JONES      MANAGER           20
      7698 BLAKE      MANAGER           30
      7782 CLARK      MANAGER           10
      7788 SCOTT      ANALYST           20
      7839 KING       PRESIDENT         10
      7902 FORD       ANALYST           20

6 rows selected.
```

CASE EXPRESSIONS

CASE expressions lets you use IF ....THEN....ELSE logic operator in the SQL statement. CASE expressions provide you more flexible and generalized way to write multi-conditional logic in SQL statement.

CASE expressions are of two types:

- **Simple CASE expression**
- **Searched CASE expression**

- **SIMPLE CASE EXPRESSION**

Consider the syntax:

SELECT col1|expr,...., CASE(simple_case_expression) [ELSE else_expr] END
FROM table;
The simple_case_expression is specified in the following manner:
 expr WHEN comparison_expr1 THEN return_expr1
WHEN comparison_expr 2 THEN return_expr2

In a simple CASE expression, Oracle searches for the first WHEN ... THEN pair for which expr is equal to comparison_expr1 and returns return_expr1. If none of the WHEN ... THEN pairs meet this condition, and an ELSE clauses exists, then Oracle returns else_expr. Otherwise, Oracle returns null.

Let us write a query statement using simple CASE expression that displays employee names, jobs and category. The category is based on the job and is determined according to the following table:

JOB TITLE	CATEGORY
PRESIDENT	M2
MANAGER	M1
ANALYST	E3
OTHERS	E2

```
SQL> select ename,job,
  2  CASE job WHEN 'PRESIDENT' THEN 'M2'
  3           WHEN 'MANAGER' THEN 'M1'
  4           WHEN 'ANALYST' THEN 'E3'
  5           ELSE 'E2'
  6  END "category"
  7  from emp;

ENAME      JOB       ca
---------- --------- --
SMITH      CLERK     E2
ALLEN      SALESMAN  E2
WARD       SALESMAN  E2
JONES      MANAGER   M1
MARTIN     SALESMAN  E2
BLAKE      MANAGER   M1
CLARK      MANAGER   M1
SCOTT      ANALYST   E3
KING       PRESIDENT M2
TURNER     SALESMAN  E2
ADAMS      CLERK     E2


ENAME      JOB       ca
---------- --------- --
JAMES      CLERK     E2
FORD       ANALYST   E3
MILLER     CLERK     E2

14 rows selected.
```

So the above expression display the employee name according to the given expression at the above categories mention.

- **SEARCHED CASE EXPRESSIONS**

Consider the syntax:

```
SELECT col1|expr,...., CASE( searched_case_expression )
FROM  table;
```

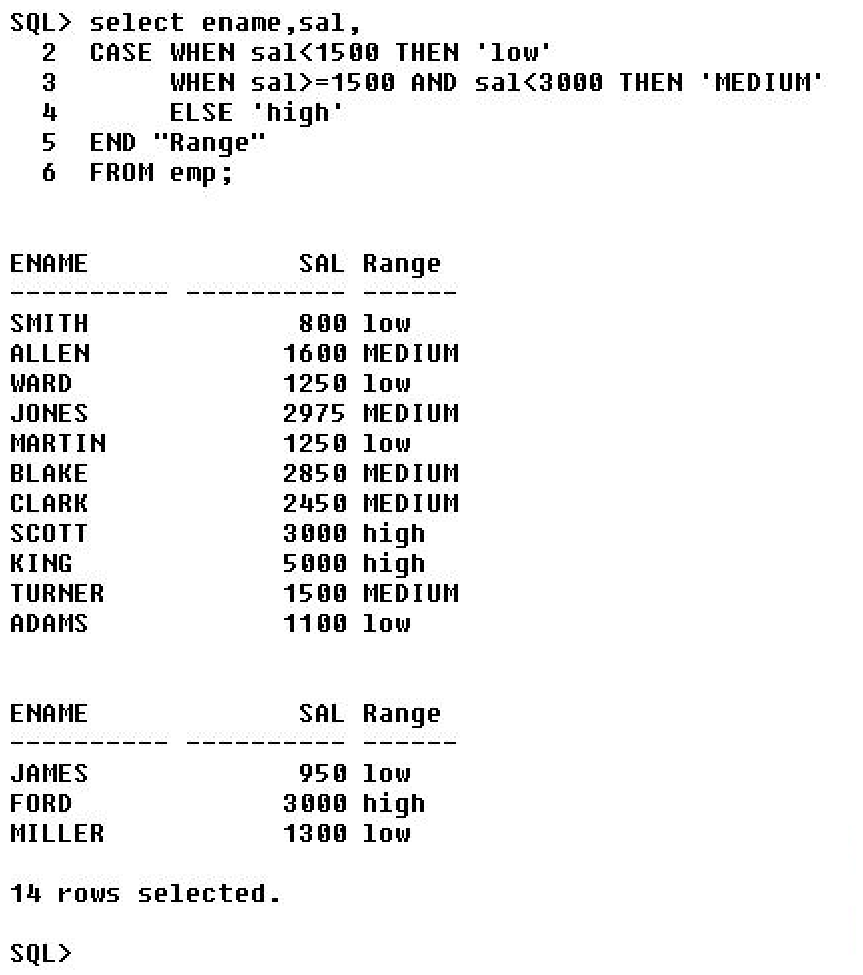
The searched_case_expression is specified in the following manner

```
expr WHEN condition1 THEN return_expr1
        WHEN condition2 THEN return_expr2
```

In a searched case expression, Oracle searches from left to right until it finds an occurrence of condition that is true, and then returns the corresponding return_expr. If no condition found is found to be true and an ELSE clause is present, Oracle returns else_expr. Otherwise Oracle returns null.

Let us, write a query statement using searched CASE expression, which displays employee names, salaries, and their ranges. The range is low if the salary is less than 1500, medium if salary is greater than or equal to 1500 but less than 3000 and high if salary is greater than or equal to 3000.

Look at the statement below:



```
SQL> select ename,sal,
     CASE WHEN sal<1500 THEN 'low'
          WHEN sal>=1500 AND sal<3000 THEN 'MEDIUM'
          ELSE 'high'
     END "Range"
     FROM emp;

ENAME             SAL Range
---------- ---------- ------
SMITH             800 low
ALLEN            1600 MEDIUM
WARD             1250 low
JONES            2975 MEDIUM
MARTIN           1250 low
BLAKE            2850 MEDIUM
CLARK            2450 MEDIUM
SCOTT            3000 high
KING             5000 high
TURNER           1500 MEDIUM
ADAMS            1100 low

ENAME             SAL Range
---------- ---------- ------
JAMES             950 low
FORD             3000 high
MILLER           1300 low

14 rows selected.

SQL>
```

Here, you find the output displays the employees with their ranges.

Chapter 5

After completing this chapter, you will be able to understand:

- ✓ **Substitution variables**
- ✓ **Create variables using define command**
- ✓ **Create variables using accept command**
- ✓ **Verifying and clearing variables using define and undefined commands**
- ✓ **Setting environment variables through set command**
- ✓ **Login.sql file**
- ✓ **Defining headers, footers and column headings**
- ✓ **Control-break reports**

SQL* Plus is an environment or interface provided by Oracle SQL* Plus enables you to execute SQL commands and PL/SQL blocks. In this chapter you will learn various SQL*Plus features that enables you to perform many additional tasks apart from using SQL and PL/SQL.

SUBSTITUTION VARIABLES

SINGLE AMPERSAND (&) SUBSTITUTION VARIABLES

You can use the single-ampersand (&) substitution variables to run SQL statements interactively. Suppose you want to display the names and salaries of the employees for a particular department. You want to provide the department number for the WHERE clause at the run time. You can do so by using a variable prefixing it with &.
Look at the statement below:

```
SQL> select ename,sal,deptno
  2  from emp
  3  where deptno=&dept_number;
```

Here we have used a substitution variable (&dept_number) in the WHERE clause instead of a specific value. Now, when you run this query, a prompt appears as shown below asking for a value for the variable.

```
Enter value for dept_number:
```

As soon as you provide a value for the variable it gets substituted by the value you provide and the query runs to display the output. Let us provide 10 here for the variable.

```
Enter value for dept_number: 10
```

And you get the following output:

```
old   3: where deptno=&dept_number
new   3: where deptno=10


ENAME             SAL     DEPTNO
---------- ---------- ----------
CLARK            2450         10
KING             5000         10
MILLER           1300         10
```

Here the first line labelled OLD displays the line containing the substitution variable and the second line labelled NEW displays that line after the substitution is done. Then the query result is displayed.

Note: You can suppress the display of old and new lines by issuing the value of the environment variable VERIFY to OFF. You can do this by issuing the following SQL*Plus command.

```
SQL> set verify off
```

If you want to provide date or character value in the WHERE clause at runtime using substitution variable, enclose the variable within single quotation marks. This lets you avoid the quotation marks at runtime.

Let us write a query to retrieve the employee name and salary of all employees for the job title entered at the prompt by the user at runtime.

Look at the statement below:

```
SQL> select ename, sal
  2  from emp
  3  where job='&job_title';
```

When we execute this statement it prompts to enter a value for job_title and the WHERE clause expects the value to be of character type. Let us enter MANAGER for it.

```
Enter value for job_title: MANAGER
old   3: where job='&job_title'
new   3: where job='MANAGER'


ENAME             SAL
---------- ----------
JONES            2975
BLAKE            2850
CLARK            2450
```

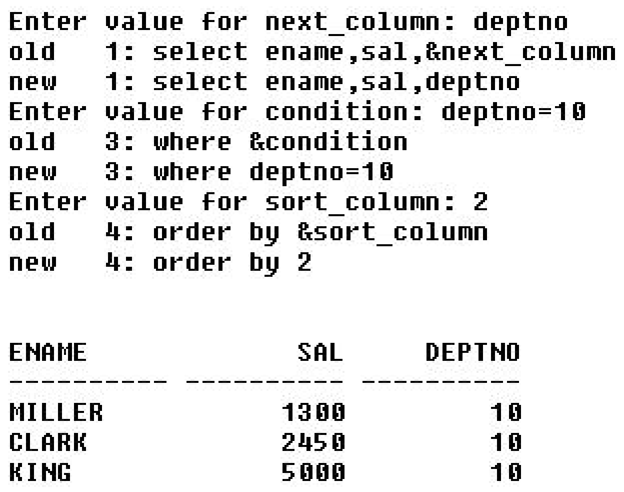
Here you can see, we have not enclosed the value for job_title within single quotes because the subtitution variable itself is enclosed within single quotes. However, if we had not enclosed the variable within quotes, we would have to enclose the value within quotes at the runtime. The same rule applies to date values as well.

The use of substitution variable is not limited for WHERE clause only. You can use substitution variables to specify column names, expressions, table name, ORDER BY clause, WHERE condition or even an entire SELECT statement (if it is subquery). Moreover, you can use multiple substitution variables in SQL statement.

Let us write a statement that displays employee name, salary and one more column that is specified at runtime. We will also specify a condition and sort column at the runtime. Look at the following statement below.

```
SQL> select ename,sal,&next_column
  2  from emp
  3  where &condition
  4  order by &sort_column;
```

When this statement is executed it will prompt for a value once for each substitution variable.
Let us execute this statement and provide values for the variables.
Look at the following:



```
Enter value for next_column: deptno
old   1: select ename,sal,&next_column
new   1: select ename,sal,deptno
Enter value for condition: deptno=10
old   3: where &condition
new   3: where deptno=10
Enter value for sort_column: 2
old   4: order by &sort_column
new   4: order by 2

ENAME             SAL     DEPTNO
---------- ---------- ----------
MILLER           1300         10
CLARK            2450         10
KING             5000         10
```

Here, the output displays the information according to the values we have specified for the substitution variables.
With the single ampersand, the user is prompted every time the command is executed, if the variable does not exist. However, if the variable exists, its value is substituted without any user interaction. If you do not enter a value for the substitution variable, you will get error when you execute the statement.

DOUBLE AMPERSAND (&&) SUBSTITUTION VARIABLES

You can use the double ampersand (&&) substitution variable if you want to reuse the value of the variable without prompting for it each time. You get the prompt for the value for only once. SQL*Plus stores the value supplied by you. It uses it again whenever you reference the variable name.
Look at the following:

```
SQL> select ename,sal,&&column_name
  2  from emp
  3  where sal>1500
  4  order by &column_name;
Enter value for column_name: deptno
old   1: select ename,sal,&&column_name
new   1: select ename,sal,deptno
old   4: order by &column_name
new   4: order by deptno

ENAME             SAL     DEPTNO
---------- ---------- ----------
CLARK            2450         10
KING             5000         10
FORD             3000         20
SCOTT            3000         20
JONES            2975         20
BLAKE            2850         30
ALLEN            1600         30

7 rows selected.
```

Here, you see the prompt has appeared once asking a value for column_name which is prefixed with double ampersand. The value of this variable is stored and is used when it is referenced again.
If you want to underfine the above variable enter the following SQL*Plus command:

```
SQL> undefine column_name
```

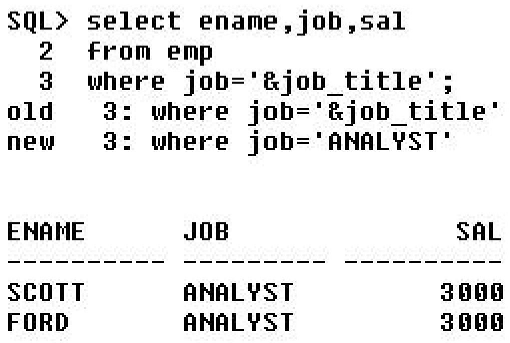
This command undefined the specified variable and if you execute the above SQL statement, it will prompt again for the variable.

CREATE VARIABLES USING DEFINE COMMAND

You can use the DEFINE command to create a user variable of CHAR datatype. You can then reference the variable thus created by prefixing it with single-ampersand. The syntax to create a variable using the DEFINE command is shown below:
DEFINE variable = value
Let us create a variable job_title to store a job_title, say ANALYST.
Look at the following:

```
SQL> define job_title=ANALYST
```

Here the variable job_title is created and is holding the value ANALYST.
Let us use this variable in the WHERE clause of a SELECT statement. Look at the following statement:



```
SQL> select ename,job,sal
  2  from emp
  3  where job='&job_title';
old   3: where job='&job_title'
new   3: where job='ANALYST'


ENAME      JOB              SAL
---------- --------- ----------
SCOTT      ANALYST         3000
FORD       ANALYST         3000
```

Here you find, the substitution variable does not prompt for a value because the variable has already been created and holding a value.

CREATE VARIABLES USING ACCEPT COMMAND

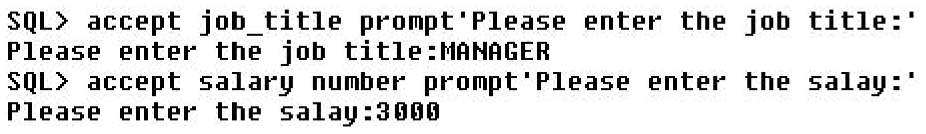
You can use the ACCEPT command to create a user variable of CHAR, NUMBER or DATE datatype. It is more flexible than the DEFINE command. Using ACCEPT command you can create a customized prompt while accepting user input and you may also hide the user input for security reasons.

You can then reference the variable by prefixing it with single-ampersand. The syntax to create a variable using the ACCEPT command is given below:

ACCEPT variable [data type] [PROMPT text] [HIDE]
In the syntax, variable is the name of the variable that stores the value. Datatype is either CHAR, NUMBER or DATE. The default is CHAR. PROMPT displays the prompt text. You can use the HIDE option to conceal what the user types.

Let us use a ACCEPT command to create variables that store job_title and salary.

Look at the following:



```
SQL> accept job_title prompt'Please enter the job title:'
Please enter the job title:MANAGER
SQL> accept salary number prompt'Please enter the salay:'
Please enter the salay:3000
```

Here we can create two variable namely, job_title and salary with the values MANAGER and 3000 respectively. Now, let us reference these variables in a SELECT statement. See the following example:

```
SQL> select ename,job,sal
  2  from emp
  3  where job='&job_title' and sal<&salary;
old   3: where job='&job_title' and sal<&salary
new   3: where job='MANAGER' and sal<      3000


ENAME      JOB              SAL
---------- --------- ----------
JONES      MANAGER         2975
BLAKE      MANAGER         2850
CLARK      MANAGER         2450
```

Here you see, we have referenced the variable in a SELECT statement by prefixing them with single ampersand and the output displays the rows accordingly.

VERIFYING AND CLEARING VARIABLES USING DEFINE AND UNDEFINE COMMAND

A variable remains defined until you either use the UNDEFINE command to clear it or exit SQL*Plus. You can use the DEFINE command the verify the variables that you have created. The syntax is:

DEFINE (variable_name)
If you omit variable_name, SQL*Plus displays all the variables that you have defined including the system variables.

Let us verify the variable job_title. Look at the following:

```
SQL> define job_title
DEFINE JOB_TITLE       = "MANAGER" (CHAR)
```

Here we find JOB_TITLE is defined and holding a value MANAGER, which is of CHAR data type.

You have already seen how to use the UNDEFINE command to clear the variables created using double-ampersand (&&). You use this command in the same way to clear the variables created using DEFINE or ACCEPT command.

Let us undefined the variable job_title. Look at the following:

```
SQL> undefine job_title
```

Now, the variable job_title is undefined. Let us verify it. Look at the following:

```
SQL> define job_title
SP2-0135: symbol job_title is UNDEFINED
```

So here you find the variable job_title does no longer exist.

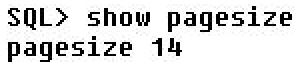
SETTING ENVIRONMNET VARIABLES THROUGH SET COMMAND

You can customize the SQL*Plus environment by setting the environment variables through the SET command. You can verify the current value of an environment variable by using the SHOW command.

Syntax for setting a value for an environment variable:
SET (environment_variable value)

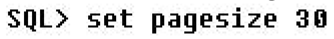
Syntax for verifying the value of an environment variable:
SHOW (environment_variable | ALL)

Let us verify the current value of PAGESIZE using the SHOW command.
Look at the following:



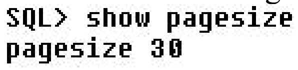
```
SQL> show pagesize
pagesize 14
```

Here you find, PAGESIZE is set to 14. Let us change it to 30 using SET command.
Look at the following:



```
SQL> set pagesize 30
```

Now, the PAGESIZE has been set to 30. Let us verify it again.
Look at the following:



```
SQL> show pagesize
pagesize 30
```

Here you confirm the PAGESIZE is 30.

[NOTE: The SHOW ALL command displays all the variables with their current values.]

The following table gives a reference of most commonly used environment variables with their default value

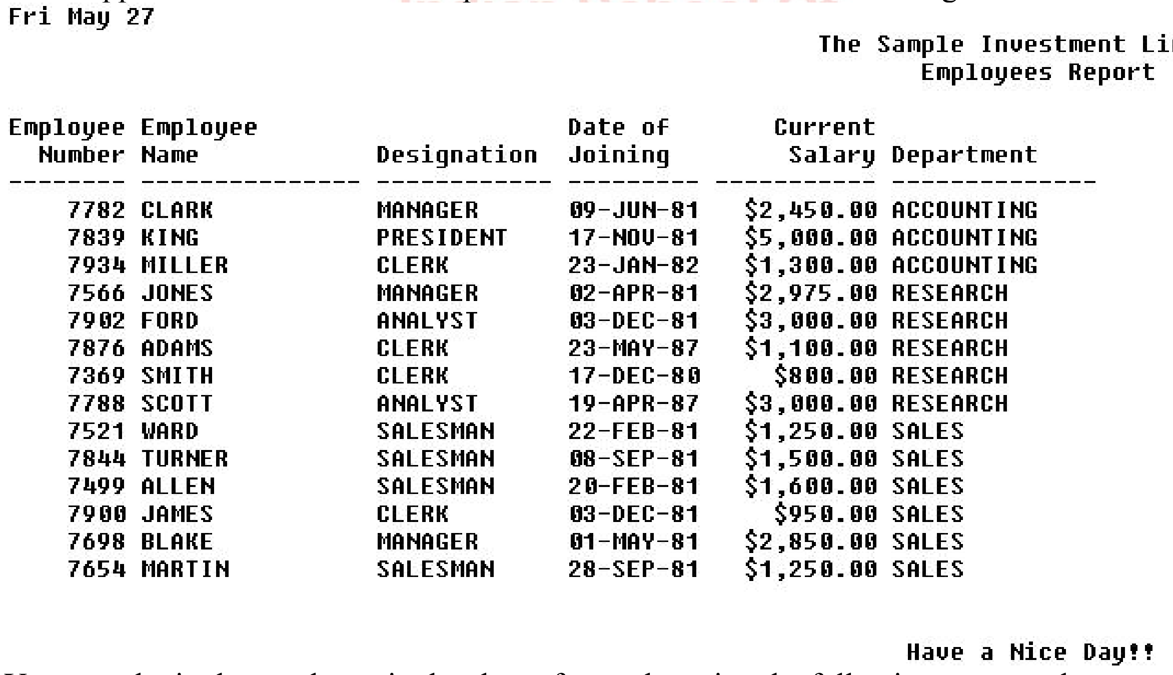
SET variable and values	Description
CLOSEP {_|text}	Sets text to be printed between columns. Default is single space.
FEED[BACK] {6|n|OFF|ON}	Displays the number of records returned by a query when the query selects at least n records.
HEA[DING] {OFF|ON}	Determine whether column headings are displayed the output.
LIN[ESIZE]{80|n}	Sets the number of characters displayed per line to n.
PAGES[IZE] {14|N}	Specifies the number of line per page.
PAU[SE] {OFF|ON|text}	Control scrolling of the output. (You must press ENTER after seeing each pause.)
LONG {80|n}	Sets the maximum width for displaying LONG values.
ECHO {ON|OFF}	Display the listing of commands in command files.

NOTE: The value 'n' represents a numeric value. The underline values indicate the default.

DEFINING HEADERS, FOOTERS AND COLUMN HEADINGS

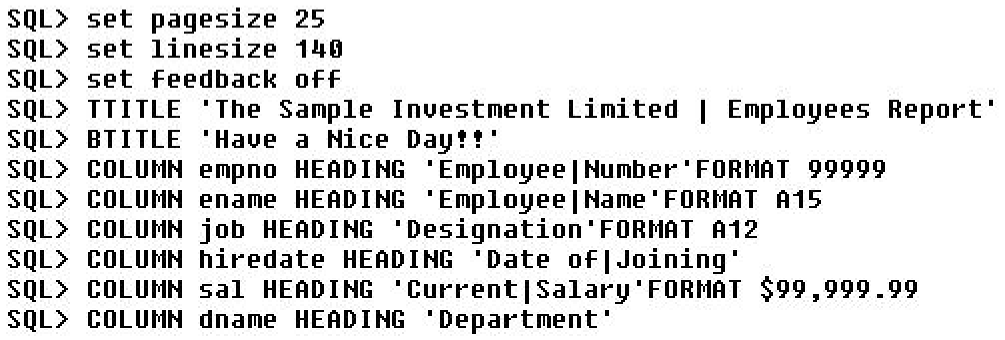
1. Headers and Footers are defined by using the TTITLE and BTITLE statements.
2. An SQL statement must be executed to see the effects of SQL*Plus format commands.
3. No semicolon is required after an SQL*Plus command for defining headers, footers and column headings.
4. Date and Page numbers automatically appear at the top of the page.
5. Both the Titles remain enabled until the user either resets them or disables them.
6. The Bottom and top titles are defined using the command.
    a. TITLE | BTITLE [options] <title-text>
    b. options are: LEFT, RIGHT and CENTER.
7. Enabling and disabling of titles is done by using the TTITLE | BTITLE ON|OFF command.
8. Column headings are defined using the command:
   COLUMN <col-name> HEADING <heading-text> FORMAT <format-text>
9. Single quotes are required if the heading contains more than one word.
10. The pipe symbol'|' is used in titles and column headings to split the heading into more than one line.
11. Column heading can be cleared by using COLUMN <col-name> CLEAR command.

Now suppose we have to create a report that looks similar to the following:



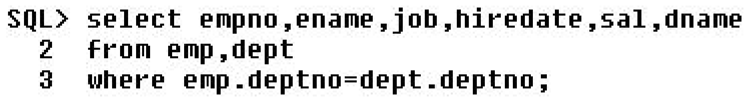
```
Fri May 27
                                                       The Sample Investment Limited
                                                              Employees Report

Employee Employee                        Date of       Current
  Number Name            Designation     Joining        Salary Department
-------- --------------- ------------ --------- ----------- --------------
    7782 CLARK           MANAGER      09-JUN-81   $2,450.00 ACCOUNTING
    7839 KING            PRESIDENT    17-NOV-81   $5,000.00 ACCOUNTING
    7934 MILLER          CLERK        23-JAN-82   $1,300.00 ACCOUNTING
    7566 JONES           MANAGER      02-APR-81   $2,975.00 RESEARCH
    7902 FORD            ANALYST      03-DEC-81   $3,000.00 RESEARCH
    7876 ADAMS           CLERK        23-MAY-87   $1,100.00 RESEARCH
    7369 SMITH           CLERK        17-DEC-80     $800.00 RESEARCH
    7788 SCOTT           ANALYST      19-APR-87   $3,000.00 RESEARCH
    7521 WARD            SALESMAN     22-FEB-81   $1,250.00 SALES
    7844 TURNER          SALESMAN     08-SEP-81   $1,500.00 SALES
    7499 ALLEN           SALESMAN     20-FEB-81   $1,600.00 SALES
    7900 JAMES           CLERK        03-DEC-81     $950.00 SALES
    7698 BLAKE           MANAGER      01-MAY-81   $2,850.00 SALES
    7654 MARTIN          SALESMAN     28-SEP-81   $1,250.00 SALES


                                                           Have a Nice Day!!
```

You can obtain the employee in the above format by using the following commands:



```
SQL> set pagesize 25
SQL> set linesize 140
SQL> set feedback off
SQL> TTITLE 'The Sample Investment Limited | Employees Report'
SQL> BTITLE 'Have a Nice Day!!'
SQL> COLUMN empno HEADING 'Employee|Number'FORMAT 99999
SQL> COLUMN ename HEADING 'Employee|Name'FORMAT A15
SQL> COLUMN job HEADING 'Designation'FORMAT A12
SQL> COLUMN hiredate HEADING 'Date of|Joining'
SQL> COLUMN sal HEADING 'Current|Salary'FORMAT $99,999.99
SQL> COLUMN dname HEADING 'Department'
```

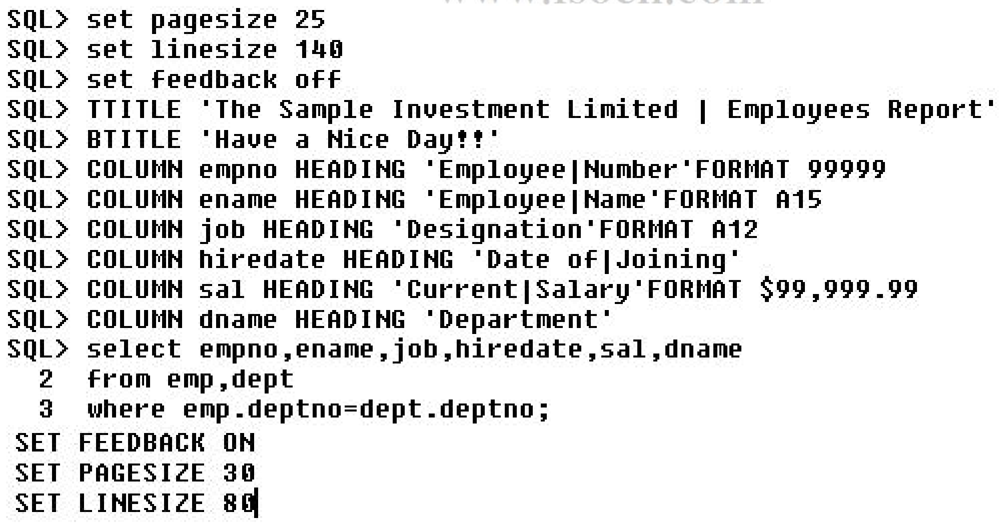
After executing these commands we execute the SELECT statement and the output will be displayed according to the above format specified.
Look at the following:



```
SQL> select empno,ename,job,hiredate,sal,dname
  2  from emp,dept
  3  where emp.deptno=dept.deptno;
```

Try to execute this statement after you have issued the format commands and match the output with the one given in the previous page.

Usually we create a script file to display such reports. Specify the necessary SET commands and format commands followed by the SELECT statement in the script file. You may also specify the set commands after the SELECT statement to restore them back.
For example, in order to obtain the above discussed report, we create a script file with following contents using the EDIT command (EDIT <file_name>.sql):



```
SQL> set pagesize 25
SQL> set linesize 140
SQL> set feedback off
SQL> TTITLE 'The Sample Investment Limited | Employees Report'
SQL> BTITLE 'Have a Nice Day!!'
SQL> COLUMN empno HEADING 'Employee|Number'FORMAT 99999
SQL> COLUMN ename HEADING 'Employee|Name'FORMAT A15
SQL> COLUMN job HEADING 'Designation'FORMAT A12
SQL> COLUMN hiredate HEADING 'Date of|Joining'
SQL> COLUMN sal HEADING 'Current|Salary'FORMAT $99,999.99
SQL> COLUMN dname HEADING 'Department'
SQL> select empno,ename,job,hiredate,sal,dname
  2  from emp,dept
  3  where emp.deptno=dept.deptno;
 SET FEEDBACK ON
 SET PAGESIZE 30
 SET LINESIZE 80
```

Save the script file

Enter START file_name or @file_name to run the script.

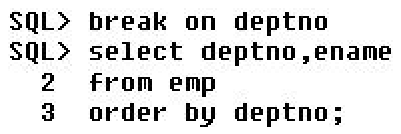
To clear the setting of the column headings, top title and bottom title, issue the following commands:
COLUMN empno CLEAR
COLUMN ename CLEAR
COLUMN job CLEAR
TTITLE OFF
BTITLE OFF
Formatting the columns
The data from the columns can be re-formatted according to the requirements. This is done by specifying the FORMAT clause after the column heading definition.
COLUMN <col-name> [HEADING <heading-text>] FORMAT <picture>
For example
COLUMN ename HEADING 'Employee | Name ' FORMAT A20
COLUMN sal HEADING 'Salary' FORMAT $99,999.00

Here 'A' stands for the formatting picture of character datatype, the number following it is the maximum length of characters that can be displayed. Similarly, the numbers are formatted using '$','9' and '0'. The number of '9's represents the maximum size and the decimal point indicates the position of the decimal point in the output.

CONTROL-BREAK REPORTS

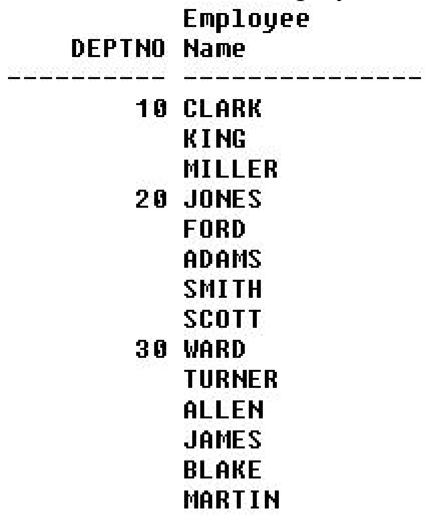
- You can use the BREAK command to suppress duplicates and section out rows.
- ORDER BY clause is required to control the break.
- You can also use Breaks to introduce line spacing, page breaks, etc.
- Clear the breaks, using CLEAR BREAKS command.
- Use the COMPUTE command, together with BREAK, to compute and display summaries along with the details.
- CLEAR COMPUTES will clear the settings.

Syntax
BREAK ON <col-name> [SKIP <n> | PAGE] [ON <col-name>......]
COMPUTE <function(s)> OF <column> ON <col-name> | ROW | PAGE | REPORT
Let use display a report with break on department number.
Look at the following:



```
SQL> break on deptno
SQL> select deptno,ename
  2  from emp
  3  order by deptno;
```

Now this statement displays the output in the following format:

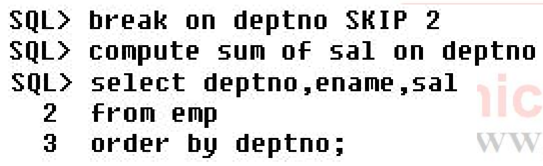


```
           Employee
    DEPTNO Name
---------- ---------------
        10 CLARK
           KING
           MILLER
        20 JONES
           FORD
           ADAMS
           SMITH
           SCOTT
        30 WARD
           TURNER
           ALLEN
           JAMES
           BLAKE
           MARTIN
```

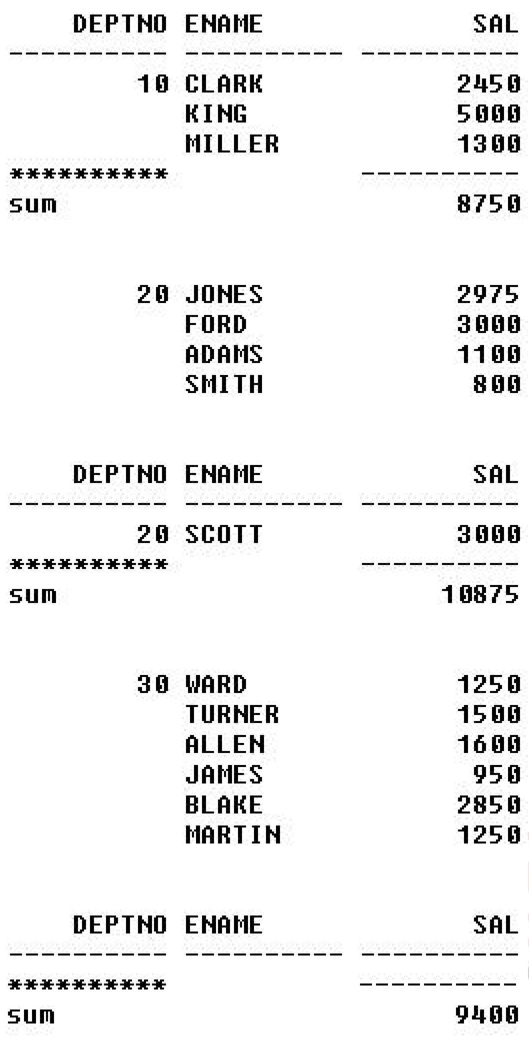
Here you see the duplicate values of deptno are suppressed.

Now let us add to the above report to display the sum of salaries along with the list of employees of each department and insert two blank lines at each break.

Look at the following:



```
SQL> break on deptno SKIP 2
SQL> compute sum of sal on deptno
SQL> select deptno,ename,sal
  2  from emp
  3  order by deptno;
```



```
    DEPTNO ENAME             SAL
---------- ---------- ----------
        10 CLARK            2450
           KING             5000
           MILLER           1300
**********            ----------
sum                         8750

        20 JONES            2975
           FORD             3000
           ADAMS            1100
           SMITH             800

    DEPTNO ENAME             SAL
---------- ---------- ----------
        20 SCOTT            3000
**********            ----------
sum                        10875

        30 WARD             1250
           TURNER           1500
           ALLEN            1600
           JAMES             950
           BLAKE            2850
           MARTIN           1250

    DEPTNO ENAME             SAL
---------- ---------- ----------
**********            ----------
sum                         9400
```

Here you find that total salary is calculated for each department.

Chapter 6

After completing this chapter you will be able to understand:

- ✓ **What is Data definition Language (DDL)**
- ✓ **How to create tables & include constraints to a table**
- ✓ **How to alter the table structure**
- ✓ **How to drop & truncate tables**
- ✓ **How to rename tables**

DATA DEFINITION LANGUAGE

Data definition language (DDL) is one of the subset of SQL. DDL statement enable you to create, alter or drop schema object like Table, View etc.

CREATING TABLES

Tables are logical container of data in a database.Tables are created using the DDL statement Create Table.

Keep the following in mind while creating tables:

The name of the table by the given user must be unique.

- The column names in the table must be unique.
- The column names in the table must be unique. However, column names can be duplicated across tables.
- Table names are not case sensitive. You can create table with a name in Uppercase, Lowercase and mixed case characters. However all table names are stored in uppercase in the Data dictionary.
- If you want create a table with the name in the lower case character,then you must enclose the table within double quotes while creating the table.

NAMING CONVENTION

The following are the general convention followed while creating table:

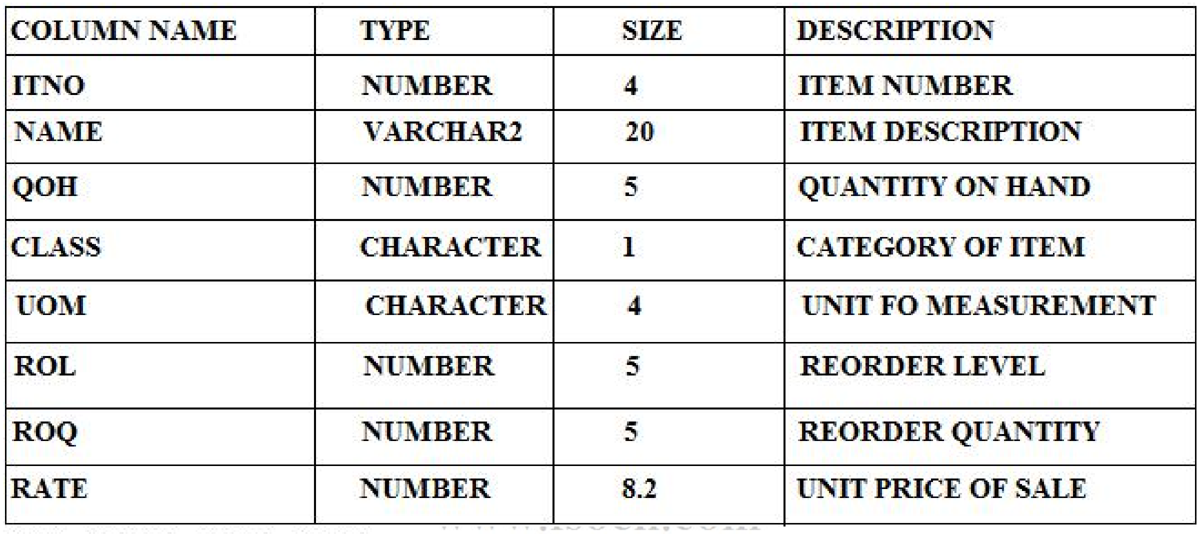
- The table name must begin with a letter A-Z or a-z.
- It may contain letters,numerals and special characters.
- It may be up to 30 characters long.
- The table name must not be a SQL reserved word.
- The name must not be same as the name of any other object in your schema.
- The number of column in a table can be up to a maximum of 1000.

  Consider the syntax:

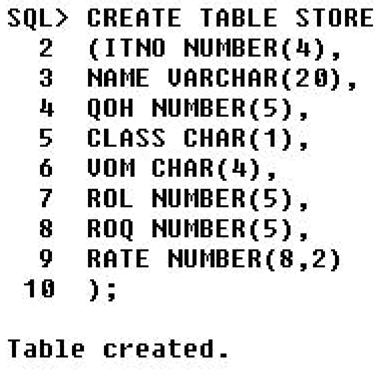
CREATE TABLE<table-name>
(<column-name><datatype (size) Default <expert> Constraint <constraint_name><Column_constraint>)

Where
<table-name> (specify the name of the table to be created)
<column-name> (specify the name of the a column in the table)
< datatype (size)> (specify the data type of the column)
Default <expert> (specify the default value that is taken when the value for the column is omitted
Constraint <constraint-name> (specify the name of the constraint)
<column_constraint> (specify the integrity constraint for a column)

Let us take an example that shows that how to create a table STORE,which has the following column:



COLUMN NAME	TYPE	SIZE	DESCRIPTION
ITNO	NUMBER	4	ITEM NUMBER
NAME	VARCHAR2	20	ITEM DESCRIPTION
QOH	NUMBER	5	QUANTITY ON HAND
CLASS	CHARACTER	1	CATEGORY OF ITEM
UOM	CHARACTER	4	UNIT FO MEASUREMENT
ROL	NUMBER	5	REORDER LEVEL
ROQ	NUMBER	5	REORDER QUANTITY
RATE	NUMBER	8.2	UNIT PRICE OF SALE



```
SQL> CREATE TABLE STORE
  2  (ITNO NUMBER(4),
  3  NAME VARCHAR(20),
  4  QOH NUMBER(5),
  5  CLASS CHAR(1),
  6  UOM CHAR(4),
  7  ROL NUMBER(5),
  8  ROQ NUMBER(5),
  9  RATE NUMBER(8,2)
 10  );

Table created.
```

Whenever you create a table, you can specify a default value for the column in case the value is not provided for that column. This is done using the DEFAULT clause with CREATE TABLE statement. Let us see how to use it.

Default clause:

This clause lets you specify a value to be assigned to the column if the user omits the value for the column

Keep the following in mind while using DEFAULT clause:

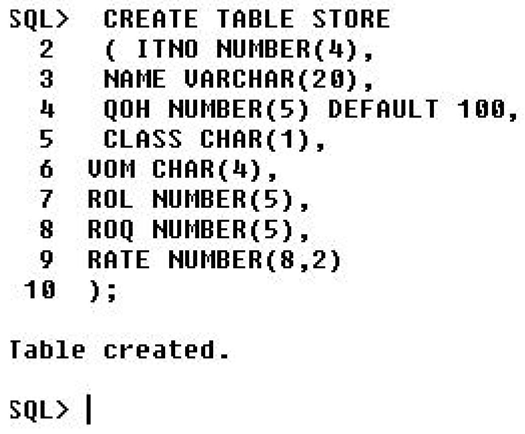
1. The data type of the expression must match the data type of the column.

2. The column must be long enough to hold the value of the expression.

3. The Default expression can also be an SQL function.

4. However, a default expression cannot contain a reference to PL/SQL functions, to the other columns in the table, to pseudo columns like LEVEL, ROWNUM and PRIOR.

EXAMPLE:
The following examples need to have a default value of 100 as the QOH in the STORE table, in case the user does not enter any value for it.



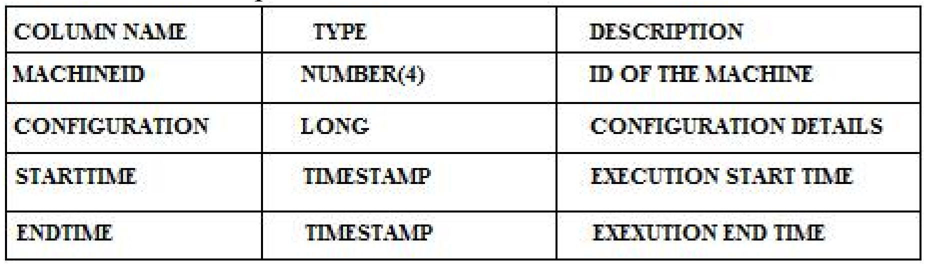
```
SQL>  CREATE TABLE STORE
  2   ( ITNO NUMBER(4),
  3   NAME VARCHAR(20),
  4   QOH NUMBER(5) DEFAULT 100,
  5   CLASS CHAR(1),
  6  UOM CHAR(4),
  7  ROL NUMBER(5),
  8  ROQ NUMBER(5),
  9  RATE NUMBER(8,2)
 10  );

Table created.

SQL> |
```

NOTE: Assigning a default value of 0 to numeric column helps in arithmetic computation because the column having NULL values are ignored with relational operators, and produce incorrect result with arithmetic operators.

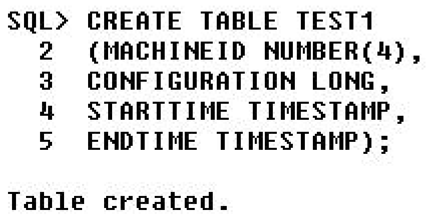
Creating table with TIMESTAMP data type:

The table can have columns with the new data type TIMESTAMP. Let us now see how to declare a column with Timestamp data type. Suppose you would like to record the time of execution of a statement in different machines having different architecture (OS, RAM capacity, Processor type etc) in a table. Then you can use the TIMESTAMP datatype for the columns STARTTIME and ENDTIME.

Let us take an example that creates table name TEST1 that has the following column:



COLUMN NAME	TYPE	DESCRIPTION
MACHINEID	NUMBER(4)	ID OF THE MACHINE
CONFIGURATION	LONG	CONFIGURATION DETAILS
STARTTIME	TIMESTAMP	EXECUTION START TIME
ENDTIME	TIMESTAMP	EXEXUTION END TIME



```
SQL> CREATE TABLE TEST1
  2  (MACHINEID NUMBER(4),
  3  CONFIGURATION LONG,
  4  STARTTIME TIMESTAMP,
  5  ENDTIME TIMESTAMP);

Table created.
```

Here the above programme shows how you can use the timestamp data type for the column starttime and endtime to record the accurate time with fractional seconds.

CONSTRAINTS

Constraints are the rule that restricts the values for one or more columns in a table.These rules gets active only during an insert,update or delete statement is issued against the table.If any of the constraint are violated during insert,update or delete statement it doesn't perform successfully. The constraints are declared during the creation of a table with the 'Create Table' command.

COLUMN CONSTRAINTS

The column constraints are the part of column definition.It can usually impose rules only on the column in which it is define.
The following are the list that shows the column constraint that can be applied to a column in a table:

- NULL/NOTNULL:Prevent a column from accepting NULL values.
- UNIQUE:Ensures uniqueness of the value in a column.
- PRIMARY KEY:Same as unique,but only one column per table is allowed.
- CHECK:Control the value odd column being inserted.
- REFERENCE:Assigns a foreign key constraint to maintain "Referential Integrity".

TABLE CONSTRIANTS

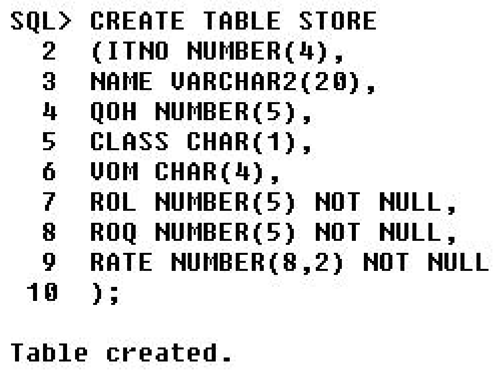
The table constraints are the part of the table definition.It can impose rule on any column in the table.This can impose any type of integrity constraint like(UNIQUE,PRIMARY KEY,CHECK AD REFERNCE) expect a NOT NULL constraint.

INCLUDING CONSTRAINT IN A TABLE:

All the detail of constraints are stored in the data dictionary.Each constraint is assigned a unique name which is easier to give user define name,so that it can easily reference when you need it or otherwise it will automatically generated the oracle in the following format as follows.
SYS_Cn
Where n is unique number.

NULL/NOT NULL

Any attempt to provide NULL values to a column declared with NOT NULL constraint will create an error and the statement will be rolled back.If you do not specify NULL OR NOTNULL to a column,the default constraint is taken as NULL.This means that the column can allow NULL values.
Let us take an example that will modify the column RATE,ROL AND ROQ of an item in the table STORE so as not to allow NULLs:



```
SQL> CREATE TABLE STORE
  2  (ITNO NUMBER(4),
  3  NAME VARCHAR2(20),
  4  QOH NUMBER(5),
  5  CLASS CHAR(1),
  6  UOM CHAR(4),
  7  ROL NUMBER(5) NOT NULL,
  8  ROQ NUMBER(5) NOT NULL,
  9  RATE NUMBER(8,2) NOT NULL
 10  );

Table created.
```

The above programme display the data type how to use the null/notnull constraint while creating the table.

(Note:The null/not null can only be specified in the column constraint and not in the table constraint).

UNIQUE

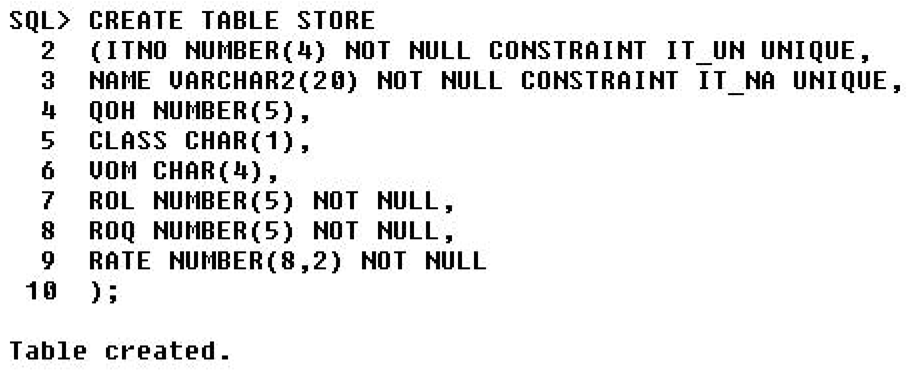
This constraint ensures that the values entered into a column are unique.User can make an individual or combination of the column to be unique.
Keep the following in mind while declaring a column as unique:

- A single column with UNIQUE constraint can contain NULL values.

- You cannot specify both the UNIQUE and PRIMARY KEY constraint on one column or on combination of column.

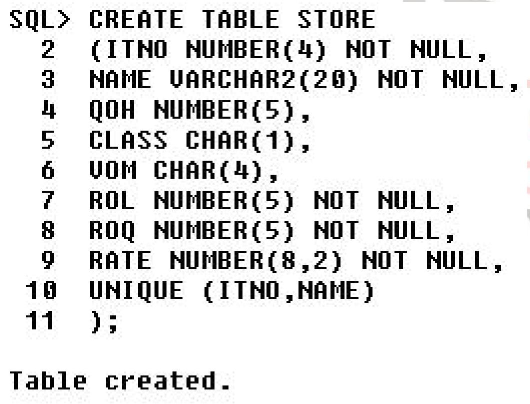
- A column with UNIQUE constraint cannot have LONG or LONGRAW data type.

Let us see an example that will made the STORE table to declare the ITNO and NMAE column as unique as follows:



```
SQL> CREATE TABLE STORE
  2  (ITNO NUMBER(4) NOT NULL CONSTRAINT IT_UN UNIQUE,
  3  NAME VARCHAR2(20) NOT NULL CONSTRAINT IT_NA UNIQUE,
  4  QOH NUMBER(5),
  5  CLASS CHAR(1),
  6  UOM CHAR(4),
  7  ROL NUMBER(5) NOT NULL,
  8  ROQ NUMBER(5) NOT NULL,
  9  RATE NUMBER(8,2) NOT NULL
 10  );

Table created.
```

In the above example if the description of an item can be duplicated then we cannot declare the NAME column as unique.So the combination of ITNO and NAME need to be declared as unique.Let us take another example that will show how the table data type will be as follows:



```
SQL> CREATE TABLE STORE
  2  (ITNO NUMBER(4) NOT NULL,
  3  NAME VARCHAR2(20) NOT NULL,
  4  QOH NUMBER(5),
  5  CLASS CHAR(1),
  6  UOM CHAR(4),
  7  ROL NUMBER(5) NOT NULL,
  8  ROQ NUMBER(5) NOT NULL,
  9  RATE NUMBER(8,2) NOT NULL,
 10  UNIQUE (ITNO,NAME)
 11  );

Table created.
```

PRIMARY KEY

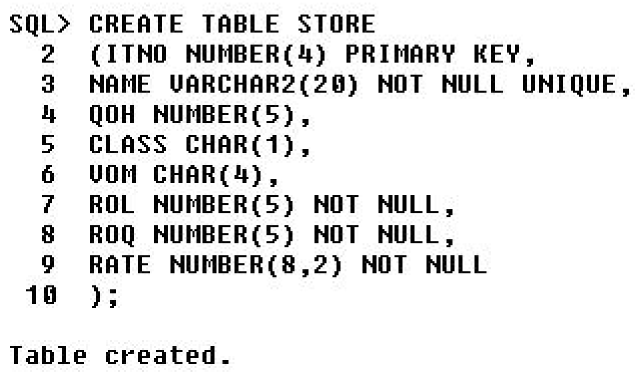
Primary key is a column in a table which must contain a unique value,which can be used to idenfiy each and every row of a table uniquely.Therefore we declare the primary key to a table with NOT NULL and UNIQUE constraint.

Consider the following point while declaring the primary key constraint:

- A table can contain only one primary key.
- The column that is declared as primary key that cannot have the following data types:LONG,LONG RAW,VARRAY,NESTED TABLE,OBJECT,LOB,BFILE or REF.

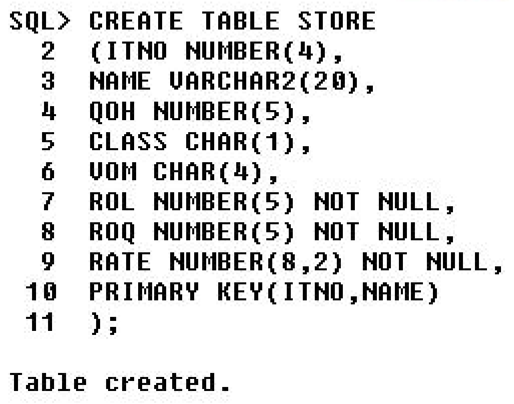
- No two rows can have the same value for a PRIMARY Key column.
- No column that is a PRIMARY Key can have NULL value.
- You cannot designation the same column as both PRIMARY and a UNIQUE key.
- In a table,the primary key table constraint can also apply to multiple columns,forcing a unique combination of value.

Let us take an example that shows the STORE table,the column can also be declared with a PRIMARY KEY as follows:



```
SQL> CREATE TABLE STORE
  2  (ITNO NUMBER(4) PRIMARY KEY,
  3  NAME VARCHAR2(20) NOT NULL UNIQUE,
  4  QOH NUMBER(5),
  5  CLASS CHAR(1),
  6  VOM CHAR(4),
  7  ROL NUMBER(5) NOT NULL,
  8  ROQ NUMBER(5) NOT NULL,
  9  RATE NUMBER(8,2) NOT NULL
 10  );

Table created.
```

Let us take another data type of declaring the combination of ITNO and NAME as the PRIMARY key as it can be used to define a composite primary key table as follows:



```
SQL> CREATE TABLE STORE
  2  (ITNO NUMBER(4),
  3  NAME VARCHAR2(20),
  4  QOH NUMBER(5),
  5  CLASS CHAR(1),
  6  VOM CHAR(4),
  7  ROL NUMBER(5) NOT NULL,
  8  ROQ NUMBER(5) NOT NULL,
  9  RATE NUMBER(8,2) NOT NULL,
 10  PRIMARY KEY(ITNO,NAME)
 11  );

Table created.
```

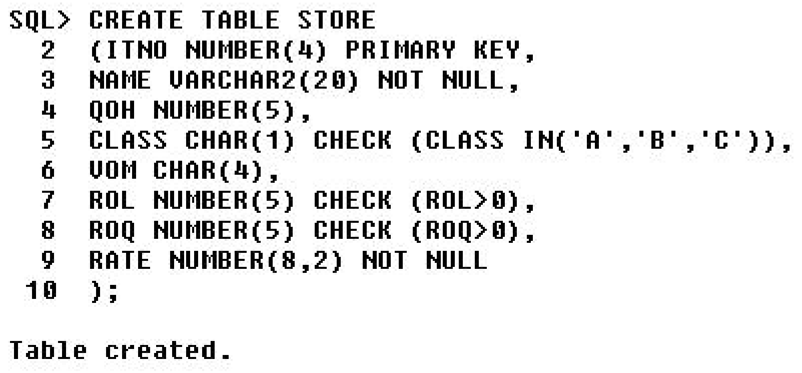
CHECK

SQL provide the CHECK constraint,which allows the user to define a condition,that a value entered into the table has to satisfy the condition before it can be accepted.A CHECK clause let you specify a condition that each row has to satisfy.
Consider the following point while declaring the CHECK statement:

- The condition of a CHECK clause can refer to any column with in the table but cannot refer to column in other table.

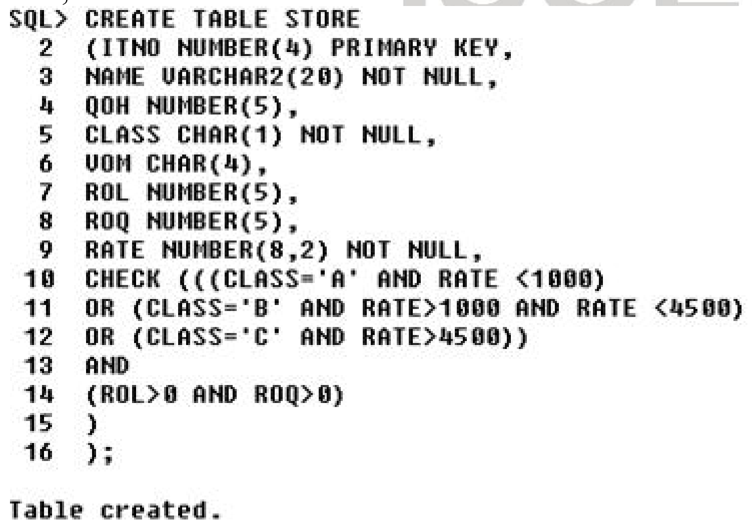
- The condition cannot contain sub queries, functions like SYSDATE, UID, USER or USERENV pseudo columns like CURRVAL, NEXTVAL, LEVEL or ROWNUM.

Let us take an example that will follow the changes to be made in the STORE table,the condition are :-if the class of an item is to be only 'A','B','C' so that no other value for this column is accepted,beside keeping a check that the ROQ and ROL cannot be zero,as follows:



```
SQL> CREATE TABLE STORE
  2  (ITNO NUMBER(4) PRIMARY KEY,
  3  NAME VARCHAR2(20) NOT NULL,
  4  QOH NUMBER(5),
  5  CLASS CHAR(1) CHECK (CLASS IN('A','B','C')),
  6  VOM CHAR(4),
  7  ROL NUMBER(5) CHECK (ROL>0),
  8  ROQ NUMBER(5) CHECK (ROQ>0),
  9  RATE NUMBER(8,2) NOT NULL
 10  );

Table created.
```

Let us see another example of table STORE that will include the rate of any item under class 'A':-which will be less than 1000.00,class 'B':-which will be more than 1000.00,butless than 4500.00 and class 'C':-which will be greater than 4500.00.Beside ROL and ROQ not being zero,as follows:



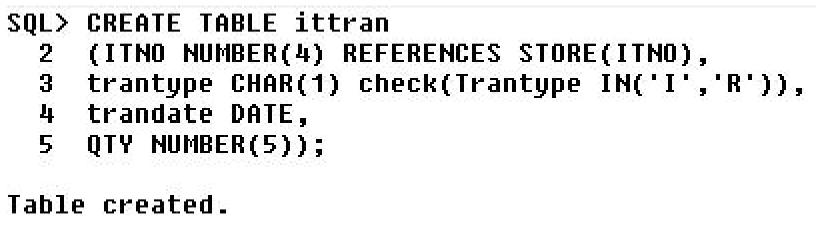
```
SQL> CREATE TABLE STORE
  2  (ITNO NUMBER(4) PRIMARY KEY,
  3  NAME VARCHAR2(20) NOT NULL,
  4  QOH NUMBER(5),
  5  CLASS CHAR(1) NOT NULL,
  6  VOM CHAR(4),
  7  ROL NUMBER(5),
  8  ROQ NUMBER(5),
  9  RATE NUMBER(8,2) NOT NULL,
 10  CHECK (((CLASS='A' AND RATE <1000)
 11  OR (CLASS='B' AND RATE>1000 AND RATE <4500)
 12  OR (CLASS='C' AND RATE>4500))
 13  AND
 14  (ROL>0 AND ROQ>0)
 15  )
 16  );

Table created.
```

REFERENTIAL INTEGRITY

Referential integrity constraint establishes a relation between a foreign key and a specified primary key.The primary key in this context is called the reference key.The table containing the foreign key is called the child object and the table containing reference key is called the parent object.The foreign key and the reference key can be in the same table also.
Here is some point that should be considered while designating the referential integrity as follows:

- The reference key column in the parent table must already be defined with PRIMARY KEY or UNIQUE constraint.
- A foreign key column cannot be of data type LONG or LONG RAW.
- The data type of a foreign key should match the data type of the reference key.

Let us take an example that shows an item transaction table with referential integrity as a column constraint as follows:

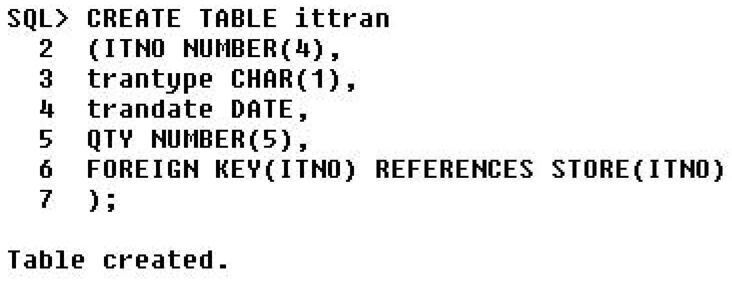


```
SQL> CREATE TABLE ittran
  2  (ITNO NUMBER(4) REFERENCES STORE(ITNO),
  3  trantype CHAR(1) check(Trantype IN('I','R')),
  4  trandate DATE,
  5  QTY NUMBER(5));

Table created.
```

(Note:The table with which the referential integrity is being specified(i.e. STORE)must already exist).

FOREIGN KEY

The foreign key clause should be used to define a composite foreign key,if the user want to designate a column or combination of columns as a foreign key.
Consider the following example that the STORE transaction table with referential integrity defined as a table level constraint as follows:



```
SQL> CREATE TABLE ittran
  2  (ITNO NUMBER(4),
  3  trantype CHAR(1),
  4  trandate DATE,
  5  QTY NUMBER(5),
  6  FOREIGN KEY(ITNO) REFERENCES STORE(ITNO)
  7  );

Table created.
```

ON DELETE CASCADE CLAUSE

Using this clause whenever a parent row is deleted then, the entire corresponding child rows are deleted from the detail Table. This option is always used with REFERENCES/FOREIGN KEY. Following example will create a table ITTRAN with ON DELETE CASCADE option as follows:

CREATE table ittran (ITNO NUMBER (4) REFERENCES STORE(ITNO)ON DELETE CASCADE,
        trantype CHAR (1) check (Trantype IN('I' , 'R')),

```
        trandate DATE,
QTY NUMBER(5));
```

ALTERING TABLE STRUCTURE

ALTERING TABLE command can change any column datatype provided if all rows of that column contain null using the modify clause.
Point to be reminded during altering the table structure is as follows:

- You can use it to reduce the size as long as the change does not require data the modified.
- You can modify the DATE column to TIMESTAMP.
- You can add a NOTNULL constraint to a column using MODIFY clause only if the column does not contain the NULL values.
- You can add new column using the ADD clause in the ALTER TABLE STATEMENT.
- You can add other integrity constraint like (CHECK, PRIMARY KEY, FORGEIN KEY, REFERENCE etc) using the ADD CONSTRIANT clause in the ALTER TABLE statement.
- You can drop a column using the DROP clause in the ALTER table.
- You can drop the integrity constraint from the table DROP CONSTRAINT clause.

The ADD CLAUSE:

Consider the following syntax:

```
ALTER TABLE <TABLE-NAME>
[ADD (<COL-ELEMENT>|CONSTRAINT <CONSTRAINT>......]
[MODIFY <COL-ELEMENT>|<CONSTRAINT>....]
[DROP COLUMN <COLUMNNAME>|[CONSTRIANT] <CONSTRAINT>......];
```

Let us take an example with ALTER table command to add new columns or constraint to the table as follows:

```
ALTER TABLE EMP1
ADD grade CHAR(1)
/
Table altered.
DESC EMP1
```

Name	Null?	Type
EMPNO		NUMBER(4)
ENAME		VARCHAR2(10)

JOB		VARCHAR2(9)
MGR		NUMBER(4)
HIREDATE		DATE
SAL		NUMBER(7,2)
COMM		NUMBER(7,2)
DEPTNO		NUMBER(2)
GRADE		CHAR(1)

The MODIFY CLAUSE:

The modify clause can be used to change the following are:

- Data type of an existing column.
- Column size.
- Default clause.
- Constraint NOTNULL /NULL.

The modify clause modifies the constraint, data type,column size of an existing column with the following restrictions:

- The type of a column can be change if every row for the column is NULL.
- A NOT NULL column may be added only to a table with no rows.
- An existing column can be modified to NOTNULL only if it has a NON NULL value in every row.

Let us take an example that modify the sal column of the emp1 table to NOTNULL and increase its size to 10 as follows:

```
SQL>ALTER TABLE emp1
2 MODIFY sal NUMBER (10, 2) NOT NULL;
Table altered.
```

The DROP CLAUSE:

The DROP clause removes a column or constraints from a table.
For dropping columns, DROP COLUMN clause of the ALTER TABLE command is used. It lets you to free space by dropping columns you no longer need in your table. You can also mark the columns to be dropped at a future time when there is less demand on your system resources.
The following example will drop a comm. column from the emp1 table:

```
SQL>ALTER TABLE emp1 DROP COLUMN comm;
Table altered.
```

SET UNUSED CLAUSE:

Mark the columns unused with SET UNUSED clause of the ATER TABLE command. Marking a column as unused does not remove the target columns from each row in the table. This means that unlike dropping a column it does not restore the disk space used by these columns. Unused columns are treated as if they were dropped, even though their column data remains in the table's rows. When the demand on the system resources in a database server is less then these unused columns can be dropped. This will improve the performance of your database when you have to drop many columns in a table with large number of rows.
Restriction that should be remembered during the use of these clause as follows:

- Even through the column data remain in the table,once the column are marked unused, you cannot roll back the result of this clause.This mean that you cannot mark the column as used again in order to access that column.
- If you mark a column of data type LONG to be unused, you cannot add another LONG column to the table until you actually drop the LONG column.

Let us take an example with the unused clause command as follows:

```
SQL>ALTER TABLE emp1 SET UNUSED COLUMN JOB;
Table altered.
```

DROPPING UNUSED COLUMN:

You can view all table with column marked UNUSED in the data dictionary views USER_UNUSED_COL_TABS.
Restrictions that apply to drop column operation are as follows:

- Column cannot be dropped from an object type table.
- Column cannot be dropped from nested tables.
- All the column cannot be dropped from a table.
- Partitioning key column cannot be dropped.
- Column from table that are owned by SYS cannot be dropped.

Let us take an example :

```
SQL>ALTER TABLE emp1 DROP UNUSED COLUMN;
Table altered.
```

DROPPING CONSTRAINTS

The constraint can be dropped using the DROP CONSTRIANT clause with the ALTER table command.
Let us take an example using the dropping constraint command as follows:
SQL>Alter table emp1
Drop constraint emp_sal
Table altered

THE ENABLE/DISABLE CLAUSE:

Constraints are enable or disable using the ENABLE/DISABLE clause of the ALTER table command. Constraint can be enable or disable without dropping them or recreating them.
Consider the following syantax:
ALTER TABLE<table-name>
[DISABLE] <Constraints>...]
[ENABLE]<Constraints>...]

Let us take an example that using this clause as follows:
SQL>ALTER TABLE EMP1
2 DISABLE CONSTRAINT EMP1_C;
Let us take another example using the keyword CASCADE to disable the dependent constraint as follows:
SQL>ALTER TABLE EMP1
2 DISABLE CONSTRAINT   EMP1_C   CASCADE;
(Note: Data dictionary for constraint isUSER_CONSTRAINTSand USER_CONS_COLUMNS. Using these data dictionary user can find out which constraint is attached to which column of a table).

Let us take an example using the user_constrsint for displaying the column and constraint of table emp1 as follows:
SQL>SELECT COLUMN NAME,
CONSTRAINT_NAME FROM USER_CONS_COLUMN WHERE
TABLE_NAME = 'EMP1';

RENAMING TABLES:

The RENAME statement is used to rename a table, view or sequence.
Point to remembered that Oracle automatically does the following tasks when any of these object as follows:

- It transfers the integrity constraint, indexes and grant on the old objects to new objects.
- It invalidates all objects that depend on the renamed object.

Let us see an example using the rename command as follows:
SQL>RENAME dept1 TO DEPARTMENT;
Table renamed.

DROPPING TABLES:

The drop table command removes the definition of a table and all its data from the database. Once the table is dropped you cannot roll back the drop table and restore the table again. To drop a table, the table should be in your own schema or you should have DROP ANY TABLE system privilege.
Consider the following syntax:
DROP TABLE <table _name> CASCADE CONSTRAINT
Let us see an example that will delete the table DEPT1 as follows:

```
SQL>DROP TABLE dept1;
Table dropped.
```

CASCADE CONSTRAINTS:

The CASCADE CONSTRAINT clause with DROP TABLE command is used to drop all referential integrity constraint that refers to primary and unique key in the dropped table. If this option is omitted and such referential integrity constraint exists. Oracle returns an error and does not drop the table.

Let us take an example that will delete table dept1 which has a referential integrity with the emp1 table as follows:

```
SQL>DROP TABLE DEPT1 CASCADE CONSTRAINTS;
Table dropped.
```

TRUNCATING TABLE:

The TURNCATE statement is used to remove all rows from all rows from the table. When you truncate a table the table structure is not removed unlike DROP table command. The truncate also remove all data from all indexes also.
(note: the truncate table cannot be roll back).
Consider the following syntax:
TRUNCATE TABLE < table name >
Let us see an example using the following command as follows:

```
SQL>TRUNCATE TABLE dept1;
Table truncated.
```

Chapter 7

After completing this chapter, you will be able to understand:

- ✓ **DML – Data manipulation language**
- ✓ **Inserting values into a table**
- ✓ **Updating column(s) of a table**
- ✓ **Deleting row(s) from a table**
- ✓ **Using MERGE statement**
- ✓ **Transaction control language**
- ✓ **COMMIT and ROLLBACK**

DML – DATA MANIPULATION LANGUAGE

Data manipulation language statements enable querying and manipulation of data in the existing oracle scheme objects such as tables and views. SELECT, INSERT, UPDATE, DELETE AND MERGE are DML statements that query and manipulate data.

INSERTING VALUES INTO A TABLE

INSERT statement is used to add rows to a table.
Keep in mind the following while using the INSERT statement:

- Character data should be enclosed within single quotes.

- Values for date columns and timestamp columns can be provided within single quote or you can use the conversion functions. When you provide the values for date (in 'DD-MON-YY' format) as a string (using single quotes). Oracle internally converts the string to date.

- Syntax:

  INSERT INTO <table-name> VALUES (<list-of-values>);

Where
<table-name> - is the name of the table.
<list-of-values> - specifies the data to be added to the new row of the table.
Example:-

```
SQL> insert into emp1
  2  values (7311,'TIMOTHY','CLERK',7001,'18-APR-95',3000,100,20);
```

Note: Only one row is added to the table at a time using this syntax.

INSERTING INTO SELECTED COLUMNS

Values can be inserted for all the columns or for the selected columns using the insert statement. If you want to provide values only for some of the columns in the table, then you should specify the column(s) for which you want to insert values along with the table name. You should provide values for all the columns if you have not specified the column list.

Example:-
To insert only the employee number, name and salary, the following command is used:

```
SQL> insert into emp1 (empno,ename,sal)
  2  values (8234,'SAMUEL',6000);
```

INSERTING NEW ROWS WITH NULL AND DEFAULT VALUES

In the above example, if you are omitting the values for the other columns in the table, and if the columns can accept NULL values or have a DEFAULT clause then NULL values or the

DEFAULT value is automatically inserted in those columns. Oracle performs this implicitly. You can also insert null values into the columns by using the NULL keyword.
Example:-

```
SQL> insert into dept1 values(50,'MARKETING',null);
```

Similarly, you can also insert DEFAULT values into the columns by using the DEFAULT keyword. For example, if the dept table had been created with a default value of 'NEW YORK' for the location column, you could issue the insert statement as follows:
Example:-

```
SQL> insert into dept1 values (60,'HRD',default);
```

INSERTING DATE AND TIMESTAMP VALUES

The default format for Date is 'DD-MON-YY'. It is normally used to insert a date value. If you would like to insert a date value in another century and a specific time is also required then you can use the TO_DATE conversation function.

Example:-

The following example, insert a record with the HireDate column to be February 3, 1997.

```
SQL> insert into emp1
  2  values (9898,'WHITE','SALESMAN',7369,TO_DATE('FEB 3,97','MON DD,YY'),2300,NULL,30);
```

The following example, insert a record for the TIMESTAMP data type.

```
SQL> insert into test1
  2  values(2343,'RAM 64MB,HDD 20GB,PIII,550MHZ,WIN2000 ADV.SERVER',
  3  '03-NOV-01 04.12.50.565679','03-NOV-01 04.12.51.222323');

1 row created.
SQL> select * from test1;
```

MACHINEID	CONFIGURATION	STARTTIME	ENDTIME
2343	RAM 64MB,HDD 20GB,PIII,550MHZ,WIN2000 ADV.SERVER	03-NOV-01 04.12.50.565679 AM	03-NOV-01 04.12.51.222323 AM

NOTE: Before inserting in "test1" you have to create "test1" table.

INSERTING ROWS USING SUBSTITUTION VARIABLES

SQL*Plus substitution variables allow the user to add values interactively in an Insert Statement. The '&' symbol is used as the substitution operator. When a substitution operator is used, SQL *Plus prompts for the values of the variable, accepts it and then substitutes it in place of the variable.

Example:-
The example below, insert a record into the dept table. It prompts the user for the deptno, dname and loc values using the substitution variables.

```
SQL> insert into dept1
  2  values(&deptid,'&deptname','&location');
Enter value for deptid: 70
Enter value for deptname: EDUCATION
Enter value for location: BOSTON
old   2: values(&deptid,'&deptname','&location')
new   2: values(70,'EDUCATION','BOSTON')
```

Any number of rows can be inserted one after the other by using the '/' (slash) which executes the last executes statement.
For date and character values, the '&' and the variable name are enclosed within single quotes. If you do not provide the single quotes in the Insert statement you should provide them while entering the data in the prompt.

INSERTING ROWS USING SUB-QUERIES

The INSERT command can also be used to derive values from one table and place them into another, by using it with a query. To achieve this, simply replace the VALUES clause with an appropriate query.
In order to use query with INSERT, the target table must fulfil the following conditions:

- The target table must have already been created.
- The target table must have the same number of columns with corresponding data type returned by the query.

Example:-
Insert records of MANAGER's into TEST table with an increment amount of Rs 500/-

```
SQL> insert into test
  2  select empno, 500, sysdate from emp1
  3  where job='MANAGER';
```

Here, the empno of the managers will be selected from the EMP1 table. For each row selected, it also selects the constant values '500' and today's date, which are then placed in the TEST table.

INSERTING ROWS INTO MULTIPLE TABLES

Till now we have learnt how to insert rows in a single table. It is also possible to insert rows into multiple tables using a single Insert statement. This is called as multi-table insert. In multi-table Insert, you insert computed rows derived from the rows returned from the evaluation of a sub query into one or more tables.
Syntax:-

```
INSERT ALL
[WHEN <condition> THEN]
INTO <table_name> VALUES (list-of-values)
[ELSE INTO <table_name> VALUES (list-of-values)]
<Subquery>
```

Example:-

The following example will insert rows from the dept table into dept1 and dept2 table. It is assumed that all the tables already exist.

```
SQL> insert all
  2  into dept1(deptno, dname, loc)
  3  values(deptno, dname, loc)
  4  into dept2(deptno, dname, loc)
  5  values(deptno, dname, loc)
  6  select * from dept;
```

There are some important points to be kept in mind while using multi-table insert statement:

- Each expression in the values_clause must refer to columns returned by select list of the sub query.
- If you omit the values_clause, the select list of the sub query determines the values to be inserted.
- It is necessary that the select list must have the same number of columns as the column list of the corresponding INTO<table name> (column-list) clause.
- If you do not provide the column-list then the sub query must provide values for all columns in the target table.

Let us take another example where you would like to insert rows with deptno less than 20 in the first table and deptno greater than 20 in the second table. This can be done using the WHEN clause in multitable INSERT statement.

Example:-

```
SQL> insert all
  2  when deptno<=20 then into dept1(deptno, dname, loc)
  3  values(deptno, dname, loc)
  4  when deptno>20 then into dept2(deptno, dname, loc)
  5  values(deptno, dname, loc)
  6  select * from dept;

4 rows created.

SQL> select * from dept1;
```

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS

```
SQL> select * from dept2;
```

DEPTNO	DNAME	LOC
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

UPDATING COLUMN(S) OF A TABLE

UPDATE statement is used to modify the existing values in a table. Values of a single columns or a group of columns can be updated. Updating can be carried out for all the rows in a table or selected rows. If you wish to UPDATE values in a table, you must own the table or you must have UPDATE privilege on that table.

NOTE: If you attempt to update a record with a value that does not conform to the integrity constraint defined, then you will get an error. For example, if you try to update the deptno in the emp table to a number that does not exists in the Dept table and there is a referential integrity constraint defined then you will receive Parent Key Violation error ORA-02291.
Syntax:

```
UPDATE <table-name> SET < col-name> = <value> [,col-name=value, .....]
[WHERE <condition>]
```

<table-name> - specifies the table name.
<col-name> - specifies the name of the columns to be modified.
<Value> - specifies the value or sub query for the column.
<Condition> - specifies the rows that need to be updated. It can comprise of column name(s), expression(s), constant(s), sub queries and comparison operators.
Example:-
Update the commission of all employees to 500:

```
SQL> update emp1 set comm = 500;
```

Expressions can also be used with UPDATE. Increase the salary of all employees by 10%:

```
SQL> update emp1 set sal = sal+(sal *.1);
```

USING DEFAULT KEYWORD WITH UPDATE STATEMENT

DEFAULT keyword can also be used to update a row. Update commission of all MANAGER'S to the DEFAULT value:

```
SQL> update emp1 set COMM=DEFAULT
  2  where job='MANAGER';
```

USING WHERE CLAUSE WITH UPDATE STATEMENT

WHERE clause allows the updating of selected rows.
Example:-
Change the department of KING to 40:

```
SQL> update emp1 set deptno=40
  2  where ename='KING';
```

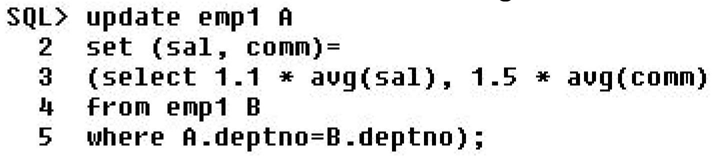
If WHERE clause is omitted here, all the rows in the table will be update.

USING WITH MULTIPLE COLUMN SUBQUERY

Multiple columns sub queries can be implemented in the SET clause of an UPDATE statement.
Syntax: UPDATE <table name>

SET (columns, column ...) = (SELECT column, column, FROM <table name>
WHERE <condition>)
[WHERE < condition>]

Example:-
Update each employee’s salary to 1.1 times the average salary of their department and each employee’s commission to 1.5 times the average commission of their department.



```
SQL> update emp1 A
     set (sal, comm)=
     (select 1.1 * avg(sal), 1.5 * avg(comm)
     from emp1 B
     where A.deptno=B.deptno);
```

RETRIEVING THE ROWS AFFECTED BY UPDATE STATEMENT

You can specify the Returning clause with the UPDATE statement to retrieve the rows affected by the statement. It will retrieve the column expressions of the affected row and store them in host variables or PL/SQL variables.

Syntax:
UPDATE <table-name> SET <col-name> = <values> [, Col-name=value ...]
[WHERE <condition>] [RETURNING <expr1>, <expr2> ... INTO <dataitem1>, <dataitem2>...

INTO clause specifies that the values of the changed rows are to be stored in the variable(s) given in the <data item> list.

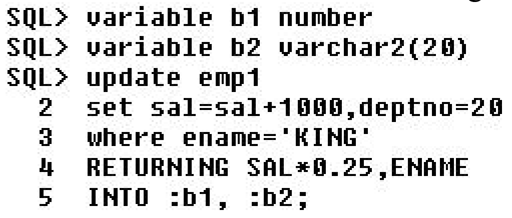
<Data item> each data item is a host (SQL *Plus) variable or PL/SQL variable that store the retrieved <expr> value.

NOTE: PL/SQL stands for Procedural Language/SQL. It will be dealt in detail in the later chapter.

Example:-
The following example returns values from the updated row and stores the result in host variables and defined b1 and b2.
Before UPDATE the table has the following values:

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7839	KING	PRESIDENT		17-NOV-81	5000		10

The UPDATE statement will be as following:



```
SQL> variable b1 number
SQL> variable b2 varchar2(20)
SQL> update emp1
     set sal=sal+1000,deptno=20
     where ename='KING'
     RETURNING SAL*0.25,ENAME
     INTO :b1, :b2;
```

If you print the value you will get:

```
SQL> print :b1 :b2


        B1
----------
    1762.5



B2
--------------------------------
KING
```

NOTE: VARIABLE is a SQL *Plus command for declaring a bind variable. The variable declared may also be directly referenced in PL/SQL. The syntax for using this is:

VARIABLE <var_name> NUMBER | CHAR (n) VARCHAR2 (n)
To display the value of a blind variable created with VARIABLE, PRINT command is used.

DELETING ROW(S) FROM A TABLE

The DELETE command removes one or more rows from a table. WHERE clause allows a set of rows to be deleted from a table by specifying the condition(s). If you omit the WHERE clause all the rows in the table will be deleted. If you want to delete rows from a table, you must either own the table or must have DELETE privilege on that table.

NOTE: If you are trying to delete a row that contains a Primary Key and that is used as a foreign key in another table, you will experience an error. For example, if you try to delete the deptno 10 in the dept table, and if there are child records in the emp table, which refer to this row, then you will get an error, ORA-02292: Integrity Constraint Violated – Child record Found.
Syntax: DELETE FROM <table name>
[WHERE <condition>]

Example:-

- Delete all records from emp1

```
SQL> delete from emp1;
```

- Delete all records of clerks

```
SQL> delete from emp1 where job='CLEARK';
```

TRANSACTION CONTROL LANGUAGE

A transaction is a sequence of SQL statement that changes a set of data from one state to another. A transaction is treated as a single unit by Oracle. A transaction either succeeds or fails as a

single unit. Never a part of the transaction is succeeded or failed. Transactions are integral part of any RDBMS and they prevent any inconsistency in the data base.
For example, consider a scenario of a bank transaction where money has to be transferred from one account to another (say from from_acct to to_acct). Two update statements have to be issued as follows:

```
UPDATE accounts SET balance = balance – tranamt
WHERE acctno = <from_acct>;
UPDATE accounts SET balance = balance – tranamt
WHERE acctno = <from_acct>;
```

Suppose the first UPDATE statement succeeds but the second statement fails due to an error. It is possible that before the second UPDATE could succeed the database server or the network had gone down. This will result in an inconsistent database where in 'from_acct' has been debited, but to_acct has not been credited. We can avoid such kind of scenarios by combining both the statements as a transaction, so that either both succeed or both fail. This ensures data consistency.

Transaction Control (TCL) statement control transactions and manage changes made by the DML statements. Following are the TCL statements for controlling the transactions.

TRANSACTION CONTROL STATEMENTS

The following are TCL statements:

- COMMIT
- ROLLBACK
- SAVEPOINT

The COMMIT command is used to make changes to data (inserts, updates, deletes) permanent.
The ROLLBACK command is used to discard parts or all the work the use has done in the current transaction SAVEPOINT statements are used to mark a save point with in the current transaction.

WHEN DOES A TRANSACTION BEGINS AND END?

A transaction begins with the first SQL statement issued after the previous transaction or the first SQL statement issued after connecting to the database.
A transaction ends when either one of the following happens:

- COMMIT or ROLLBACK statement is issued
- A DDL statement is issued
- Exits SQL*Plus
- Machine failure or system crashing

COMMIT AND ROLLBACK

The uncommitted changes made by the user are not visible to other sessions, giving them only preview of the data changes made to the database; while they continue to have access to these tables, they get the old information.

The COMMIT command is used to make the changes permanent to data base. The user can continue with any number of inserts, updates and/or deletes, and still can undo the work, issuing the ROLLBACK command. This is important in case an error is detected during these operations.

The transaction processing in Oracle can be controlled explicitly by using TCL commands, however during certain circumstances, Oracle manages transactions implicitly.

AUTO COMMIT

The action that will force a commit to occur, even without issuing the COMMIT command are:

- Normal exit from SQL*Plus
- Any DDL statement is issued
- Connecting to some other user and disconnecting from ORACLE

AUTO ROLLBACK

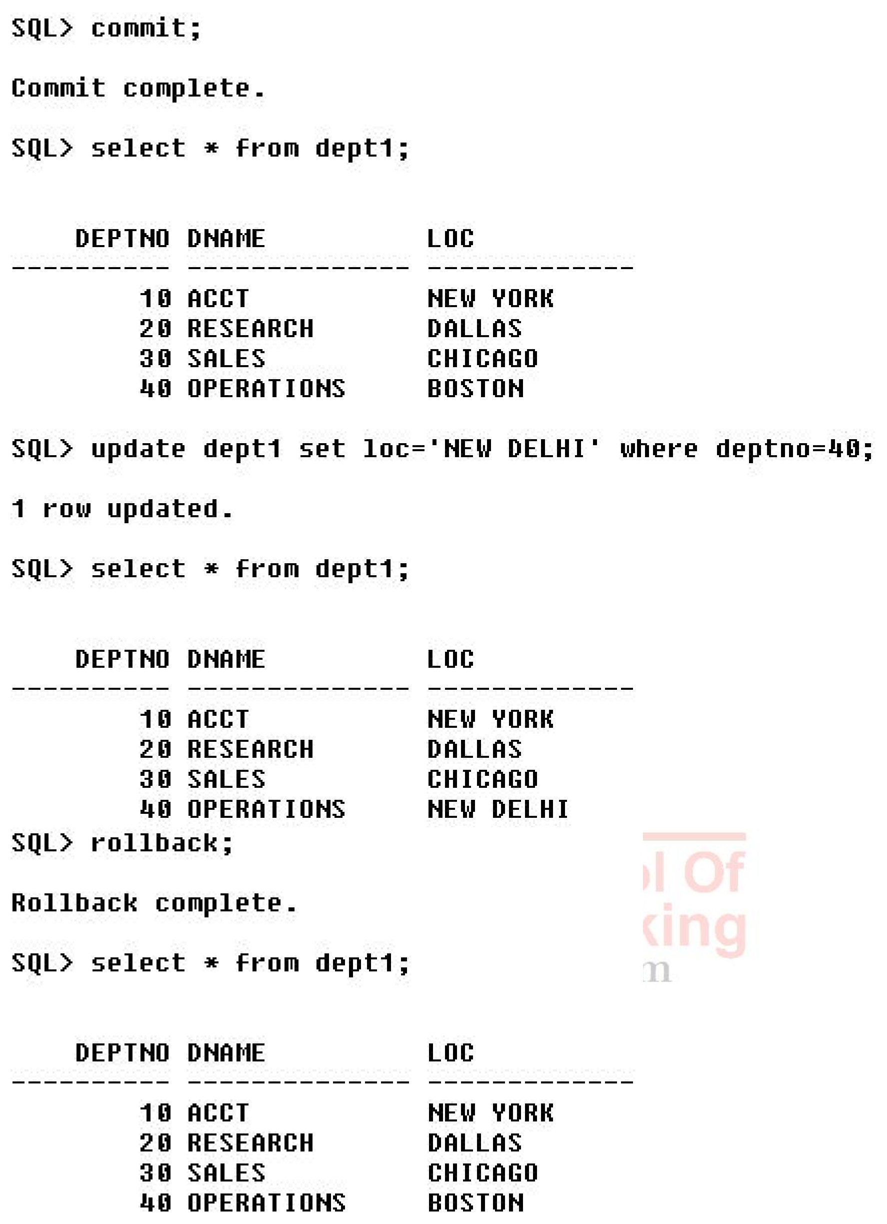
After completion of any DML statement, if the user experiences serious difficulty such as hardware failure and no explicit commit is made, ORACLE will automatically ROLLBACK all un-committed work. If the machine or data base goes down, it does this as cleanup work the next time the data base is brought up.

The actions that will force a ROLLBACK to occur, even without issuing the ROLLBACK command is:

- Abnormal termination of SQL*Plus
- System failure/network breakdown

NOTE: SQL*Plus has the facility to automatically commit all the work, without explicitly issuing COMMIT command.
SET AUTOCOMMIT ON - Enable auto commit feature
SET AUTOCOMMIT OFF – Is the default and disables the automatic committing
Examples:-
The following sequence of commands will illustrate the commit and rollback operation

```
SQL> select * from dept1;


    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON

SQL> insert into dept1 values (50,'CURRI.DEVP.','NEW DELHI');

1 row created.

SQL> select * from dept1;


    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
        50 CURRI.DEVP.    NEW DELHI
SQL> rollback;

Rollback complete.

SQL> select * from dept1;


    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
SQL> update dept1 set dname='ACCT' where deptno=10;

1 row updated.
SQL> select * from dept1;


    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCT           NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
```



```
SQL> commit;

Commit complete.

SQL> select * from dept1;


    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCT           NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON

SQL> update dept1 set loc='NEW DELHI' where deptno=40;

1 row updated.

SQL> select * from dept1;


    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCT           NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     NEW DELHI
SQL> rollback;

Rollback complete.

SQL> select * from dept1;


    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCT           NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
```

- SAVEPOINTS

Savepoint identifies a point in a transaction to which one can later rollback with the ROLLBACK statement. It is helpful when a transaction contains a large number of SQL statements and the user wants to commit only once when all are done. If required, one can rollback to a particular truncation.

Example:-
```
SQL> insert into dept1 values (60,'PURCHASE','NEW DELHI');

1 row created.

SQL> savepoint s1;

Savepoint created.

SQL> update emp1 set deptno=60 where ename='SMITH';

1 row updated.

SQL> rollback to s1;

Rollback complete.
```

The ROLLBACK statement will undo the updating only. All transaction before the save point ‘s1’ will remain unchanged, but not yet committed.
Keep in mind the following while using the SAVEPOINT:

- Save point names must be unique within a given transaction.
- If you create another save point with the same name as earlier one, the earlier save point is erased.

Chapter 8

After completing this chapter, you will be able to understand:

- ✓ **Views and their advantages**
- ✓ **How to create views**
- ✓ **How to create join views**
- ✓ **How to do DML's through views**
- ✓ **How to drop a view**

INTRODUCTION TO VIEWS

A View is a logical representation of a table or combination of tables. Views are database object whose data is derived from another table. The table on which the view is based is called as base table(s).

The following are some of the characteristics of a view as follows:

- A VIEW contains no data of its own. The data is derived from the base table(s). The base table can be actual table or can also be another view.
- The change made in the view actually affects the base table of the view.
- SELECT, INSERT, UPDATE and DELETE operations can be performed on a view as the user do in the standard tables.
- Any insertion, updation or deletion of rows in the base table will automatically reflect in the VIEW.

ADVANTAGES OF A VIEW:

- Restricted Access: A view restricts access to the database. Querying from a view can display only a restricted portion of the database. Critical data in the base table is safeguarded as access to such data can be controlled using views by not selecting those columns during the creation of a view.
- Reduced Complexity: View allows user to make simple queries to retrieve the result from complicated queries.
- Data Independence: View provides data independence for ad hoc user and application programme. One view can be used to transparently retrieved data from several tables.
- Data Presentation: Views allow the same data to be seen by different user in different ways.

CREATING VIEWS

Views can be categorized as simple and complex Views. A simple view is derived from one base table. But a complex view can be derived from more than one base table using joins. A simple view does not contain function or groups. You can perform DML operation through a simple view. A complex view can contain expressions, functions, groups or join. You cannot perform any DML operation through any complex view.

Consider the following syntax:
```
CREATE [OR REPLACE] [NO FORCE|FORCE] VIEW <view name>
[(Column1, Column2 ...)]
AS
```

<SELECT statement>
[WITH CHECK OPTION CONSTRAINT <constraint name>]
[WITH READ ONLY]
[OR REPLACE]: Specify this option to recreate the views if it already exists
[column1, column2.....]: are the names to be given to column in the view and must correspond to the column list in the select statement.
[FORCE]: specify this option if the view has to be created regardless of whether the views base tables exits or not.
[NO FORCE]: specify this option if the view has to be created only if the base table exists. This is the default option.
As <select statement> specify a subquery that identifies column and rows of the table that the view is based on.
[WITH CHECK OPTION]: specify this option to guarantee the inserts and update performed trough the view will result in rows that the view subquery can select.
[WITH READ ONLY]: specify this option to disallow any DML operation on a view.

CREATING SIMPLE VIEW

Let us take an example that will create view for the employee belonging to department 20 as follows:
CREATE VIEW dept20 AS
SELECT * FROM EMP1
WHERE DEPTNO=20;
Now the view can be queried just like the table as follows:
Select * from dept20;

CREATING COMPLEX VIEW

Let us take an example that creates view empview, which will contain the empno, ename, sal, deptno and dname as follows:
SQL>CREATE OR REPLACE VIEW empview as
2 SELECT empno, emp.deptno, ename, sal,
3 dname FROM emp, dept
4 WHERE emp.deptno = dept.deptno;

By default the column in the view will have the same name as the column name selected in the subquery. You can change the name of the columns in the View. To do so, you need to provide the names of the columns after the VIEW name, enclosed within parenthesis. Alternatively, you can provide column aliases in the SELECT statement.
The above VIEW definition can be written as follows:

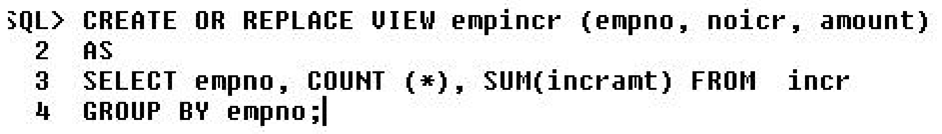
```
SQL> CREATE OR REPLACE VIEW empview (eno, dno, ename, sal, dname) as
  2  SELECT empno, emp.deptno,ename,sal, dname FROM emp, dept
  3  WHERE emp.empno=dept.deptno;
```
View created.

SQL>SELECT * FROM EMPVIEW WHERE dno =20;
You can see the entire info of deptno 20.
(Note: dno in the select statement refers to the column name in the view and not in the base table).

USING GROUP BY CLAUSE

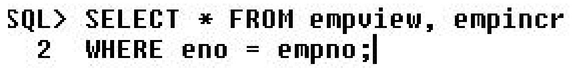
Create a VIEW for the INCR table containing the number of increments and total increment amount

Let us take an example using this group by clause as follows:



```
SQL> CREATE OR REPLACE VIEW empincr (empno, noicr, amount)
  2  AS
  3  SELECT empno, COUNT (*), SUM(incramt) FROM  incr
  4  GROUP BY empno;
```

You can also issue a join query on the view another VIEW and/or table

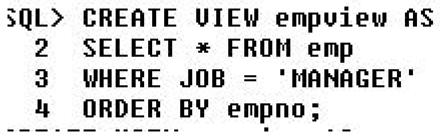


```
SQL> SELECT * FROM empview, empincr
  2  WHERE eno = empno;
```

The above query will display the list of employee containing employee number, name, salary, department number of increments and total increment amount.
Guidelines for creating a View
Keep the following guidelines while creating a View

1. The subquery that creates a View can have groups, functions or nested subqueries.
2. The subquery can have an ORDER BY clause. If you give an ORDER BY clause when you query the view, the new one overrides the ORDER BY clause in the View definition.

   Example



```
SQL> CREATE VIEW empview AS
  2  SELECT * FROM emp
  3  WHERE JOB = 'MANAGER'
  4  ORDER BY empno;
```

   View created.

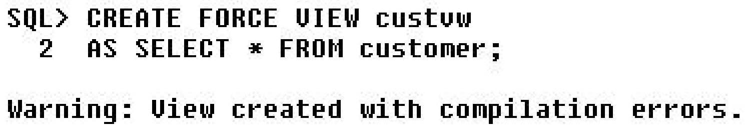
Data Dictionary for Views

Once the View is created, you can query the data dictionary table USER_VIEWS to see its definition.

Example:
SQL> SELECT text FROM user_views WHERE view_name = ‘dept20’;

USING FORCE OPTION

A view can be created even if the defining query of the view cannot be executed. A View can be created and entered into the data dictionary even if it refer to a non-existent base table(s) or an invalid column of the base table or the View does not have the required privileges. These Views are created with errors.
Example:



```
SQL> CREATE FORCE VIEW custvw
  2  AS SELECT * FROM customer;

Warning: View created with compilation errors.
```

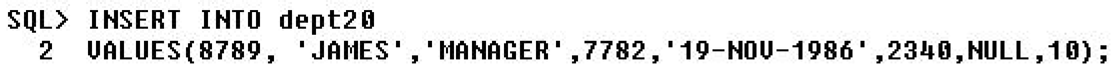
Note: 1.The status of the view created with error will be invalid.
2. Oracle dynamically compiles the invalid view if you attempt to use it.

DMLs THROUGH VIEWs

INSERT, UPDATE and DELETE statement can also be used with view. Using these commands with views is an indirect way of manipulating base table(s).

For example, you can insert a row in to the emp table using the dept20 View as follows

Example:



```
SQL> INSERT INTO dept20
  2  VALUES(8789, 'JAMES','MANAGER',7782,'19-NOV-1986',2340,NULL,10);
```

1 row created.
Now you can see that the row is inserted in the base (emp) table.

Restriction on DML operation for view uses the following criteria as follows:

- If the query in the view definition contain.
- SET or DISTINCT operator.
- GROUP BY clause.
- GROUP FUNCTION such as SUM, COUNT, MAX, MIN etc.
- If a NOT NULL column that does not have default clause is omitted from the view when it is created, then a row cannot be inserted into the base table using the view.

Let us take another example where INSERT, UPDATE OR DELETE cannot be done through these views as follows:

```
SQL>CREATE VIEW grp_vw
2 AS SELECT DEPTNO, COUNT (*) TOTALEMP1 FROM EMP1
3 GROUP BY DEPTNO;
```

Now try to insert row in the table using this view as follows. Oracle will not allow you to insert row in to table as follows:
SQL>INSERT INTO grp_vw
2 VALUES (40,9);
It gives an error like: ERROR at line 1:
ORA-01732: data manipulation operation not legal on this view

WITH READ ONLY CLAUSE

WITH READ ONLY clause with the View definition does not allow any DML operation on that View. The View is read only means you can only query the View.
Example:
Create a View of all managers in the emp table

```
SQL> CREATE OR REPLACE VIEW vw1
  2  AS SELECT * FROM emp
  3  WHERE JOB = 'MANAGER'
  4  WITH READ ONLY
```

Now you try to insert a record in to the table using this View as follows

```
SQL> INSERT INTO vw1
  2  VALUES (9845,'JEFE','MANAGER',7788,'02-NOV-1982',2111,NULL,20);
INSERT INTO vw1
*
ERROR at line 1:
ORA-01733: virtual column not allowed here
```

WITH CHECK OPTION CLAUSE

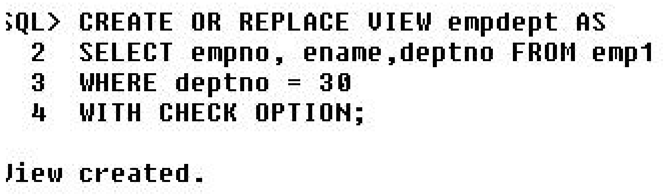
The with check option clause specifies that INSERT and UPDATE performed through the view are not allowed to created row which the view cannot select and therefore allow integrity constraint.
Now consider the following view:

```
SQL>  CREATE VIEW empdept AS
  2  SELECT empno, ename,deptno FROM emp1
  3   WHERE deptno = 30;

View created.
```

Rows can be inserted into the EMP table using this VIEW
Suppose, we insert the following:

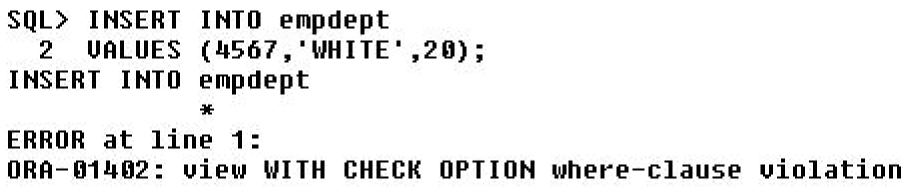
```
SQL> INSERT INTO empdept VALUES (1234, 'JACQUE', 20);

1 row created.
```

This row will be inserted without any error, but cannot be viewed by the EMPDEPT VIEW as this VIEW can only retrieve rows belonging to department 30.
Let us see the following example with the command 'WITH CHECK OPTION 'clause, which will restrict the insertion of a row that cannot be selected by a view as follows:



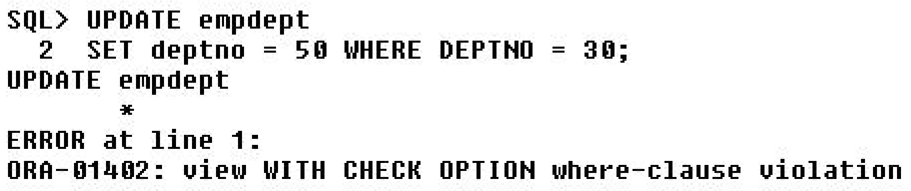
```
;QL> CREATE OR REPLACE VIEW empdept AS
  2  SELECT empno, ename,deptno FROM emp1
  3  WHERE deptno = 30
  4  WITH CHECK OPTION;

Jiew created.
```

Note: when the WITH CHECK option clause is used any DML command that manipulate rows which cannot be selected by a view are rejected.
Consider the following example from the above statement as follows:



```
SQL> INSERT INTO empdept
  2  VALUES (4567,'WHITE',20);
INSERT INTO empdept
            *
ERROR at line 1:
ORA-01402: view WITH CHECK OPTION where-clause violation
```

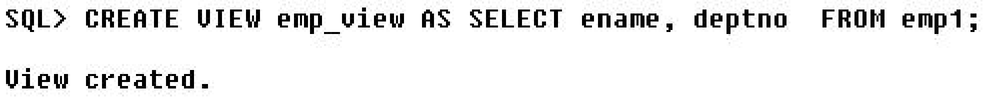
Now try to update the same view:



```
SQL> UPDATE empdept
  2  SET deptno = 50 WHERE DEPTNO = 30;
UPDATE empdept
       *
ERROR at line 1:
ORA-01402: view WITH CHECK OPTION where-clause violation
```

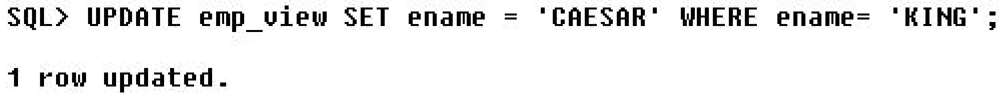
MODIFYING A JOIN VIEW

The Oracle server allows you with some restriction to modify view that involves joins. Consider the following simple view as follows:



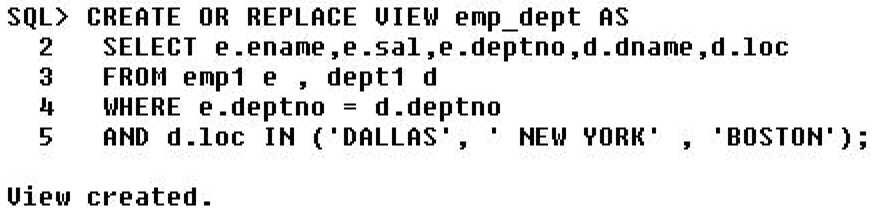
```
SQL> CREATE VIEW emp_view AS SELECT ename, deptno  FROM emp1;

View created.
```

This view does not involve a join operation. If you issue the SQL statement, the base (EMP) table that underlies the view changes, and employee 7839's name changes from KING to CAESAR in the emp table.



```
SQL> UPDATE emp_view SET ename = 'CAESAR' WHERE ename= 'KING';

1 row updated.
```

However, if you create a View with a join condition such as the one shown below, then Oracle allows you to modify the View, subject to some restrictions, which are described here:

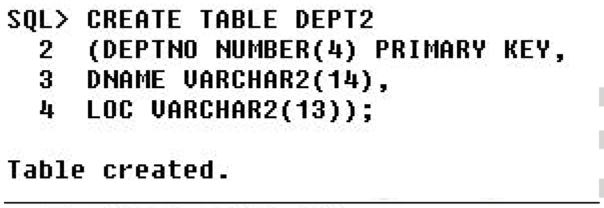


```
SQL> CREATE OR REPLACE VIEW emp_dept AS
  2   SELECT e.ename,e.sal,e.deptno,d.dname,d.loc
  3   FROM emp1 e , dept1 d
  4   WHERE e.deptno = d.deptno
  5   AND d.loc IN ('DALLAS', ' NEW YORK' , 'BOSTON');

View created.
```

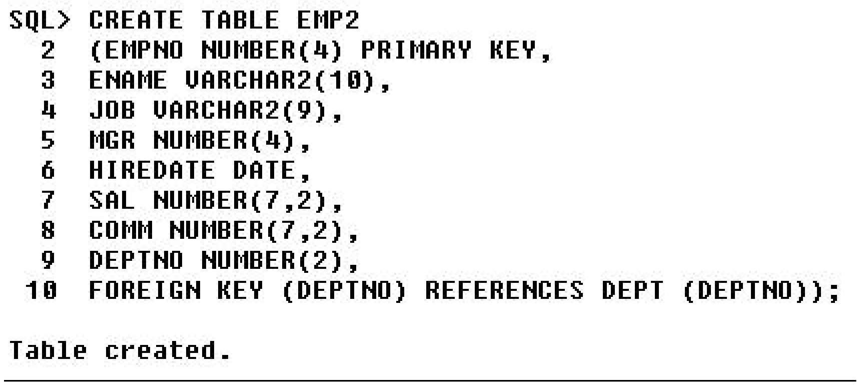
The modifiable join view is a view that contains more than one table in the top-level FROM clause of the SELECT statement and that does not contain any of the following as follows:

- DISTINCT operator
- Aggregate function: AVG, COUNT, GLB, MAX, MIN, STDDEV, SUM or VARIANCE.
- Set operators: UNION, UNIION ALL, INTERSECT, MINUS.
- GROUP BY or HAVING clauses.
- START WITH or CONNECT BY clause.
- ROWNUM pseudo column.

  The example in this section uses the familiar EMP and DEPT tables. However, the example works only if you explicitly define the primary and foreign keys in these tables, or define unique indexes. Here are the appropriately constrained table definitions for EMP and DEPT.



```
SQL> CREATE TABLE DEPT2
  2  (DEPTNO NUMBER(4) PRIMARY KEY,
  3  DNAME VARCHAR2(14),
  4  LOC VARCHAR2(13));

Table created.
```



```
SQL> CREATE TABLE EMP2
  2  (EMPNO NUMBER(4) PRIMARY KEY,
  3  ENAME VARCHAR2(10),
  4  JOB VARCHAR2(9),
  5  MGR NUMBER(4),
  6  HIREDATE DATE,
  7  SAL NUMBER(7,2),
  8  COMM NUMBER(7,2),
  9  DEPTNO NUMBER(2),
 10  FOREIGN KEY (DEPTNO) REFERENCES DEPT (DEPTNO));

Table created.
```

  You could also omit the primary and foreign key constraints listed above, and create a UNIQUE INDEX on DEPT (DEPTNO) to make the above example work.

- **KEY-PRESERVED TABLES**:-

A table is key preserved if every key of the table (emp) can also be a key (empno) of the result of the join. So a key preserved table has its keys preserved through a join.
In a key-preserve table, a user can perform all DML operation directly through the VIEW. So a key preserve table has its key-preserved through a join.
Let us take an example considering the view emp_dept, where view defined in the 'Modifying a join view section'.

In this view table EMP is a key preserved table because EMPNO is a key of the EMP table and also the key of the result of the join DEPT table is a non-key preserved table because although column deptno on DEPT table is a primary key, it is not the key of the join.
If you SELECT all rows from EMP_DEPT view defined above, the results are:

SELECT * FROM EMP_DEPT;

EMPNO	ENAME	SAL	DEPTNO	DNAME	LOC
7369	SMITH	800	20	RESEARCH	DALLAS
7566	JONES	2975	20	RESEARCH	DALLAS
7782	CLARK	2450	10	ACCOUNTING	NEW YORK
7788	SCOTT	3000	20	RESEARCH	DALLAS
7839	KING	5000	10	ACCOUNTING	NEW YORK
7876	ADAMS	1100	20	RESEARCH	DALLAS
7902	FORD	3000	20	RESEARCH	DALLAS
7934	MILLER	1300	10	ACCOUNTING	NEW YORK

8 rows selected.

Note: The key preserving property of a table does not depend on the actual data in the table. It is rather, a property of its schema and not of the data in the table. For example, if in the EMP table there was at most one employee in each department, then DEPT. DEPTNO would be unique in the result of a join of EMP and DEPT would still not be a key-preserved table.

- **Rules for DML Statements on Join Views**

  Any UPDATE, INSERT, OR DELETE statement on a join View can modify only one underlying base table.

- **UPDATING ROW IN JOIN VIEW**

  The following example shows an UPADATE statement that successfully modifies the EMP_DEPT View

```
SQL>  UPDATE emp_dept
  2   SET sal = sal * 1.10
  3   WHERE deptno = 10;
```

3 rows updated.

Consider the following query:

```
SQL> UPDATE emp_dept
  2  set loc = 'BOSTON'
  3  WHERE ename = 'SMITH';
set loc = 'BOSTON'
    *
ERROR at line 2:
ORA-01779: cannot modify a column which maps to a non key-preserved table
```

This statement fails because it attempts to modify the underlying DEPT table, and the DEPT table is not a key preserved table.

- **DELETING ROWS FROM JOIN VIEWS:**

  You can delete from a join View provided there is one and only one key-preserved table in the join.
  The following DELETE statement works on the EMP_DEPT View:

```
SQL> DELETE FROM emp_dept
  2  WHERE ename = 'SMITH';
```

  1 row deleted.

  1. If a view is defined using the WITH CHECK OPTION clause and the key-preserved table is repeated, then rows cannot be deleted from such a view.

  2. No deletion can be performed on this View because the View involves a self-join of the table that is **key-preserved.**

- **INSERTING ROWS IN JOIN VIEWS**

  The following INSERT statement on the EMP_DEPT View succeeds:

```
SQL> INSERT INTO emp_dept (ename,empno,deptno)
  2  VALUES ('JOHN',9010,40);
```

  This statement will work as one key-preserved base table is being modified (EMP) and 40 is a valid DEPTNO in the DEPT table.
  An INSERT statement such as the following:

```
SQL>  INSERT INTO emp_dept (ename,empno,deptno)
  2  VALUES ('JOHN',9010,77);
 INSERT INTO emp_dept (ename,empno,deptno)
*
ERROR at line 1:
ORA-02291: integrity constraint (SYSTEM.SYS_C005259) violated - parent key not
found
```

  Would fail because the FOREIGN KEY integrity constraint on the EMP table is violated.

- **DROPPING A VIEW**

DROP VIEW command removes the view definition from the database. Row and column are not affected since they are stored on the tables from which the view was derived. Views or other application based on a dropped view become invalid. Consider the following syntax as follows:

DROP VIEW <viewname>

Example: SQL>DROP VIEW V1;

Chapter 9

After completing this chapter, you will be able to understand:

- ✓ **Security**
- ✓ **Privileges**
- ✓ **Roles**
- ✓ **GRANT Command**
- ✓ **Data Dictionary for privileges**
- ✓ **REVOKE command**

SECURITY

In any origination, security of data is considered to be outmost important. In a multi-user environment the database access and uses should be restricted.
So oracle database security can be categorized into two: system security and data security**.**

SYSTEM SECURITY

System security restricts the users the access and use of database at the system level.For example, whether the user can change the password, the disk space allocated to the user, system operation allowed by the user.

DATA SECURITY

Data security restricts the users the access and use of database objects like table, views etc and the action that those user can have on the object. The action might be SELECT, INSERT, DELETE, UPDATE etc on the database object.

PRIVILEGES

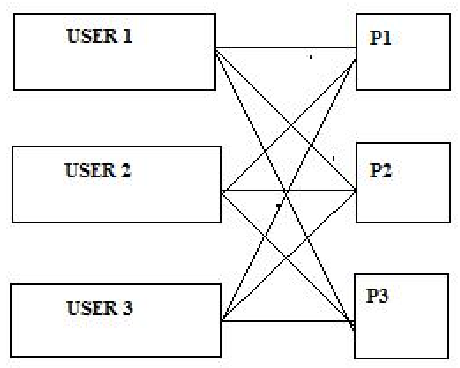
Privileges are the right to execute particular SQL statements Privileges can again be of two types

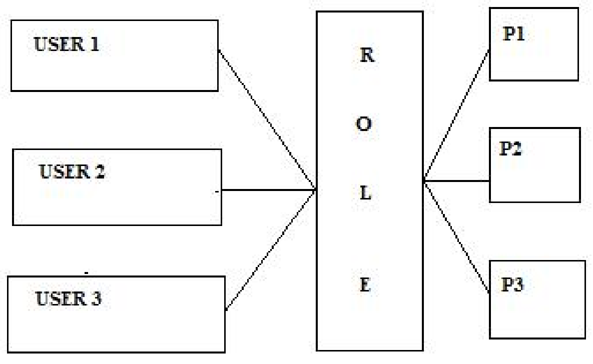
1. System Privileges
2. Object Privileges

A System privilege allows the user to gain access to the database while the object privilege allows the user to manipulate the content of the database objects.
The Database Administrator (DBA) in Oracle has high-level system privileges. A DBA can create new users, remove existing users, remove tables in any schema, backup the table and can grant system or object privileges to other users in the system. The user will be able to perform operations on the database only if he has the system privilege. The user who has been granted with the object privilege can manipulate the database objects using SELECT, INSERT etc according to the privileges granted to them.

ROLES

Roles are named collection of related privileges that can be granted to or revoked from the users. Roles are non-schema object that are not contained in a schema. Roles can be created and manipulated with SQL.
Using roles makes granting and revoking multiple privileges to many users simple. A user can have access to several roles and several users can be assigned the same role.





GRANT COMMAND

The grant command is used to grant object privilege for a particular object to specific user, roles and all users in the system. It also gives access to those database objects in his/her schema to another user.

Consider the following syntax:

```
GRANT <OBJECT_PRIV> [(columns)]|ALL
ON <object>
TO <USER|ROLE|PUBLIC>
[WITH GRANT OPTION];
```

<object_priv>: specify the object privilege to be granted such as ALTER, DELETE, INSERT, REFERENCE, SELECT or UPDATE for table.
<ALL>indicates all the object privilege.
ON<object> specifies the object on which the privileges are granted.
PUBLIC: it grants object privilege to all user.
[WITH GRANT OPTION]: allows the grantee to grant the object privilege to other user and roles.

The following table list the various object privileges that could be granted on table or rows as follows:

PRIVILEGE	OBJECT
SELECT	DATA IN A TABLE OR VIEW
INSERT	ROWS IN A TABLE OR VIEW
UPDATE	ROWS OR SPECIFIC COLUMN IN A TABLE OR VIEW
DELETE	ROWS FROM TABLE OR VIEW
ALTER	COLUMN DEFINITION IN A TABLE
INDEX	INDEX TO A TABLE
REFERENCE	REFER TO A TABLE NAMED WITHIN A TABLE OR COLUMN
ALL	REFER TO ALL THE PRIVILEGE SHOW ABOVE

Let us take an example that grants all permission on the EMP1 table to all user as follows:

```
GRANT ALL
ON emp
TO PUBLIC;
Grant succeeded.
```

Let us take another example that will grant only the select privilege to all user on a view empview as follows:

Step1: Create a view for the above table:

```
CREATE VIEW empview AS
SELECT empno, ename, sal FROM EMP;
```

Step2: Grant select privilege on the view empview to all user:

```
GRANT SELECT ON empview TO PUBLIC;
```

WITH GRANT OPTION

If the privilege is granted WITH GRANT CLAUSE, then it can be passed on to other user and roles by the grantee. Object privileges granted with grant option are revoked when the grantor's privilege is revoked.

Let us take an example that grant select permission on the dept table to user 'USER1' and allow the user to give further grants as follows:

```
GRANT SELECT
ON dept1 TO USER1
WITH GRANT OPTION;
```

If a grant on the emp table of scott is given to 'USER1'. Then USER1 can refer to the table by prefixing the owner's name to the table name.

Let us take an example that will refer USER1 to emp of scott, for which a grant is provided, using the following command as follows:

```
SELECT * FROM SCOTT.EMP1;
```

GRANTING PRIVILEGE ON SPECIFIC COLUMN

Granting object privilege can be granted on the entire object or on specific column to user and role.
Note: Before granting a column-specifc INSERT privilege, determine if the table contain any column on which NOT NULL constraint are define. Granting selective insert capability without including the NOT NULL column prevents the user from inserting any row into the table.
Important guidelines as follows:

- To grant privilege on a object, the object must be in your own schema.
- You must have been granted the object privilege WITH GRANT option.
- An object owner can grant any object privilege on the object to any other user.
- The owner of the object automatically acquires all object privilege on that object.

DATA DICTIONARY FOR PRIVILEGE

The following data dictionary table can be used to confirm the privilege granted for that user and object.
USER_TAB_PRIVS_RECD:-This table describes the object privilege granted to a user.
USER_TAB_PRIVS_MADE:-This table describe the object privilege granted on the user's object.
USER_COL_PRIVS_RECD:-This table describe the object privilege granted on the column of the user's object.
USER_COL_PRIVS_MADE:-This table describe the object privilege granted on the column of the user's object.

REVOKE COMMAND

Through revoke command the user can remove the object privilege granted to the other user or roles.
Consider the following syntax:

```
REVOKE <object_priv>|ALL
ON <object>
[CASCADE CONSTRAINTS];
```

Cascade Constraint

This option is required to remove any referential integrity constraint made to the object by the mean of the REFERENCE privilege.

Let us take an example that considers the scott user to revoke update privilege from all user on the emp table as follows:
REVOKE UPDATE on EMP1 FROM PUBLIC;
Revoke succeeded.

Examples (preetha):
The following syntax is used to create a role.
CREATE ROLE <role_name>;
The GRANT statement can be used to GRANT privileges such as SELECT, INSERT, UPDATE, DELETE etc. on a table or a view to a role. Following is the syntax.
GRANT <object_privileges>
ON <table_name|view_name>
TO <role_name>;

The GRANT statement can be used to GRANT a role to user(s), public or role(s). Following is the syntax.
GRANT <role_name>
TO <user_name/s|PUBLIC|role_name/s>;

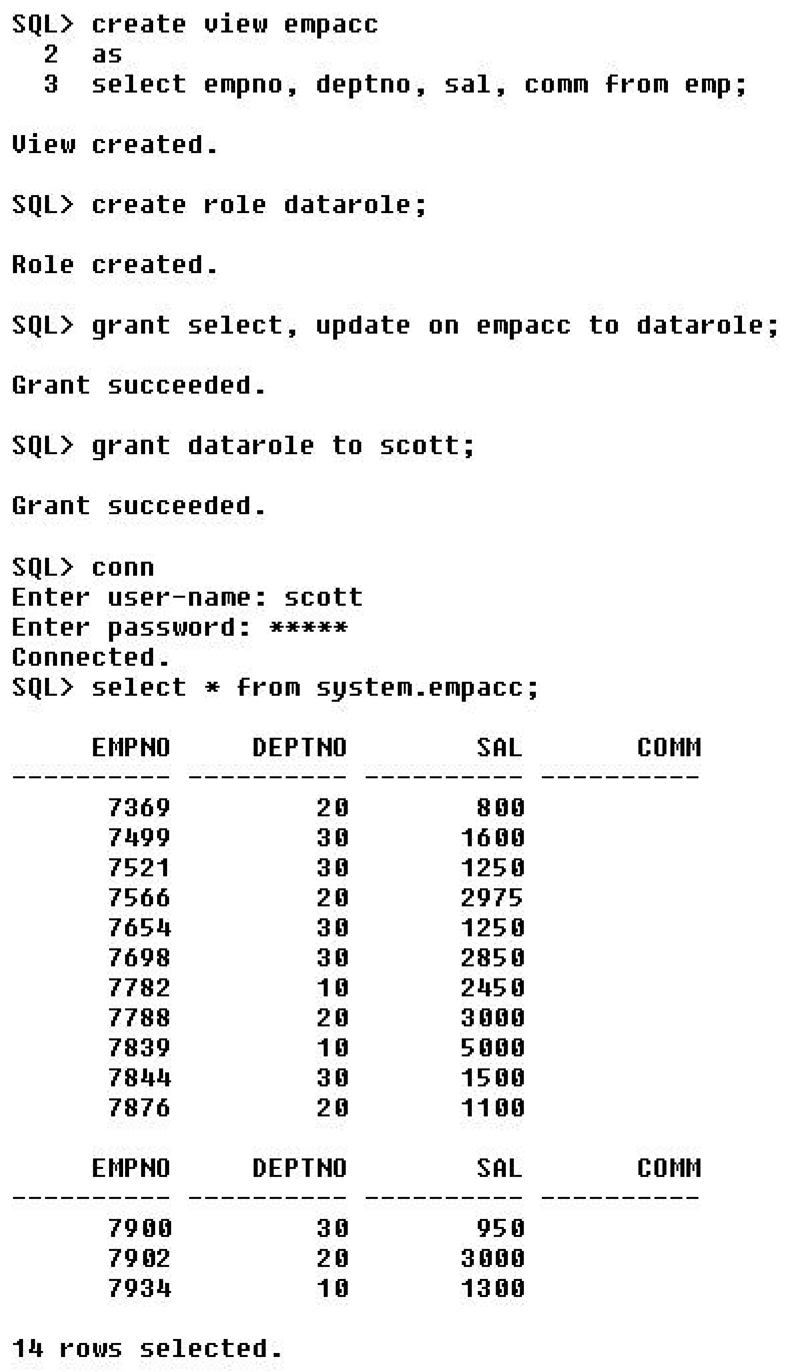
The REVOKE statement can be used to revoke a role from user(s), public or role(s). Following is the syntax.
REVOKE <object_privileges>
ON <table_name|view_name>
FROM <user_name/s|PUBLIC|role_name/s>;

The roles can be dropped using the following syntax.
DROP ROLE <role_name>
Example :

- We create a role named **datarole** under system and grant the privilege of only **select** and **update** to the user scott on a view **empacc**.
  We create a view first and then the role.

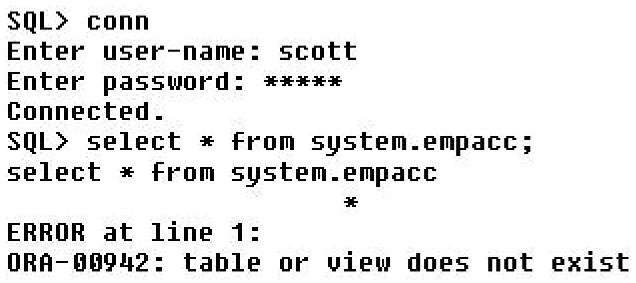
CREATE VIEW empacc
as
select empno, deptno, sal, comm from emp;
CREATE ROLE datarole;
GRANT select, update ON empacc TO datarole;
GRANT datarole TO SCOTT;

So, now user Scott cannot perform any action other than select or update from the view empacc. AN illustration is as follows.

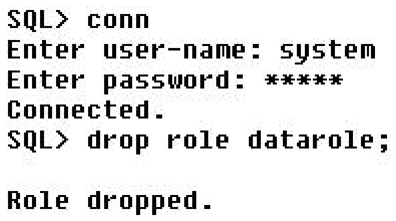


```
SQL> create view empacc
  2  as
  3  select empno, deptno, sal, comm from emp;

View created.

SQL> create role datarole;

Role created.

SQL> grant select, update on empacc to datarole;

Grant succeeded.

SQL> grant datarole to scott;

Grant succeeded.

SQL> conn
Enter user-name: scott
Enter password: *****
Connected.
SQL> select * from system.empacc;

     EMPNO     DEPTNO        SAL       COMM
---------- ---------- ---------- ----------
      7369         20        800
      7499         30       1600
      7521         30       1250
      7566         20       2975
      7654         30       1250
      7698         30       2850
      7782         10       2450
      7788         20       3000
      7839         10       5000
      7844         30       1500
      7876         20       1100

     EMPNO     DEPTNO        SAL       COMM
---------- ---------- ---------- ----------
      7900         30        950
      7902         20       3000
      7934         10       1300

14 rows selected.
```

```
SQL> update system.empacc
  2  set comm = 200
  3  where deptno=20;

5 rows updated.

SQL> select * from system.empacc;

     EMPNO     DEPTNO        SAL       COMM
---------- ---------- ---------- ----------
      7369         20        800        200
      7499         30       1600        300
      7521         30       1250        500
      7566         20       2975        200
      7654         30       1250       1400
      7698         30       2850
      7782         10       2450
      7788         20       3000        200
      7839         10       5000
      7844         30       1500          0
      7876         20       1100        200

     EMPNO     DEPTNO        SAL       COMM
---------- ---------- ---------- ----------
      7900         30        950
      7902         20       3000        200
      7934         10       1300

14 rows selected.

SQL> insert into system.empacc
  2  values(7380, 30,1300,100);
insert into system.empacc
                   *
ERROR at line 1:
ORA-01031: insufficient privileges

SQL> conn
Enter user-name: system
Enter password: *****
Connected.
SQL> revoke datarole from scott;

Revoke succeeded.
```



```
SQL> conn
Enter user-name: scott
Enter password: *****
Connected.
SQL> select * from system.empacc;
select * from system.empacc
                     *
ERROR at line 1:
ORA-00942: table or view does not exist
```



```
SQL> conn
Enter user-name: system
Enter password: *****
Connected.
SQL> drop role datarole;

Role dropped.
```

Chapter 10

After completing this chapter you will be able to understand:

- ✓ **INDEXES**
- ✓ **SEQUENCE**
- ✓ **SYNONYMS**

INDEXES

Indexes provide fast access path to column that are indexed. They are used to reference records in all the SQL statements, not just the query. Indexes are stored separately from the actual data. Indexes are referred to whenever the indexed columns are referenced in the WHERE clause. With every data manipulation the appropriate index is automatically updated. Indexes are most useful on larger tables and on column that are likely to appear in the WHERE clause.

To create an index that will guarantee the uniqueness of the primary key, the column must have the PRIMARY KEY constraint. Alternatively, the clause UNIQUE can also be used along with the CREATE INDEX command, but this statement will fail if any duplicate already exist.

Importance of index is as follows:

- Indexes are used to speed up the processes and enhance the performance in retrieving information from a table.
- Indexes can also be used to ensure that no duplicate values are entered into a column.
- Indexes do not have to be activated or deactivated.
- Oracle does not limit the number of indexes on a table.
- When using multiple columns, up to 16 columns or a maximum of 255 character of column space can be indexed.

Consider the following syntax as follows:

CREATE [UNIQUE] INDEX <index-name> ON <table-name> (<column-names>)
(Note: Oracle itself create unique index at the time of table creation, if the table definition contains a PRIMARY KEY/UNIQUE constraint.)

Let us see an example that will create a unique index for the EMP table on employee number as follows:

CREATE UNIQUE INDEX EMP1IDX ON EMP1 (ENAME);

Index created.

Let us take another example that will create an index on deptno and dname column of the dept table as follows:

CREATE INDEX DEPTNO ON DEPT (DEPTNO, DNAME);

Index created.

DROPPING AN INDEX

Drop index command is used to remove an index definition from the data dictionary.
Consider the following syntax as follows:
DROP INDEX <indexname>
(Note: Use the data dictionary USER_INDEXES to display the list of indexes in the User's schema).
Let us see an example using the user_index as follows:
SELECT* FROM USER_INDEXES;

CLUSTERING

- Clusters are used for performance improvement.
- Clustering is a method of sorting table that are related and often accessed together.
- A cluster is a group of rows from separate table stored in the same disk block.
- Clustered table must have a common column called the cluster column.
- To cluster table the user must own the tables.

Apart from using indexes, which improve performance, when used properly the use of cluster provides additional performance improvement. Clustering also result in significantly smaller storage requirement. Each cluster column is stored only once, regardless of whether it occurs once or many a time in the table.

Because clusters store related rows of different tables together in the same data blocks, disk I/O is reduced and access time improves for joins of clustered tables
Steps for creating cluster are as follows:

- Creating cluster.
- Creating cluster Table.
- Creating cluster index.

Creating cluster:-

To create a cluster in one's own schema one must have the CREATE CLUSTER system privilege and to create a cluster in another one's schema, one must have CREATE ANY CLUSTER privilege

Example:-
Step 1: Create the cluster item:

```
CREATE CLUSTER STORE (ITNO NUMBER (4));
Cluster created.
```

Creating cluster Table:-

To create a table in a cluster, the user must have either CREATE TABLE or CREATE ANY TABLE system privilege
Example:-
Step 2: Create the STORE1 table:

```
CREATE TABLE STORE1
(ITCODE NUMBER (4) NOT NULL PRIMARY KEY,
DESC1 VARCHAR (2) NOT NULL,
QOH NUMBER (5) DEFAULT 100,
CLASS CHAR (1),
UOM CHAR (4),
ROL NUMBER (5),
ROQ NUMBER (5),
RATE NUMBER (8, 2) NOT NULL)
CLUSTER STORE (ITNO);
Table created.
```

Step 3: create the STORE2 table

```
CREATE TABLE STORE2
(ITCODE NUMBER (4) REFERENCES STORE1 (ITCODE),
PDSESC VARCHAR (2) NOT NULL,
CLUSTER STORE (ITNO);
```

Creating cluster indexes as follows:

Cluster index determine the physical order of the data in the table and are usually created on the primary key. There can only be one cluster index per table. To create the cluster index one must have CREATE INDEX system privilege.

- In a clustered index, the actual data is stored in the indexed order.
- Each distinct cluster key value is stored only once in each data block.
- They save disk space and improve performance for many operations.

To create clustered index on the cluster store the following command is used:

CREATE INDEX STORE1_INDEX ON CLUSTER STORE1;
Note: the cluster key establishes the relationship of the table in the cluster.

DROPPING A CLUSTER

DROP CLUSTER command is used to remove a Cluster from data dictionary.
SYNTAX: DROP CLUSTER < cluster name >
To drop a cluster the cluster must be present in your schema, or must have the DROP ANY CLUSTER system privilege. Clustered tables can be dropped individually without affecting the table cluster or cluster index. DROP TABLE command is used to drop clustered table.
To drop a cluster which contains no tables DROP CLUSTER command is used.
Example: delete the cluster item

DROP CLUTER item;
To drop a cluster which contains one or more clustered table, INCLUDING TABLES option is included in the DROP CLUSTER command which will drop the tables as well.
If one or more tables in a cluster contains primary or unique keys that are referenced by foreign key constraints of tables outside the cluster, the cluster can't be dropped unless the dependent FOREIGN KEY constraints are also dropped. To drop such clusters CASCADE CONATRAINTS option can be used with DROP CLUSTER command.

SEQUENCES

The sequence generator is used to automatically generate sequence number for rows in the tables. It can be used to produce unique primary keys.
Following are the point if sequence as follows:

- A sequence is a database object that is used to generate unique integers.
- These integers are normally used as primary key values for a table.
- A sequence is created through the CREATE SEQUENCE command.
- The pseudocolumn CURRVAL is used to refer to a sequence number that is the current sequence number.

Consider the following syntax:
CREATE SEQUENCE <seq-name>
[INCREMENT by <n>]
[START WITH <m>]
[MAXVALUE N| NOMAXVALUE]
[MINVALUE N| NOMINVALUE]
Where
<seq-name> is the name of the sequence
<n> is the increment specified by the user. The default increment is 1.
START WITH is the number with which the sequence will begin

MINVALUE clause provides lower bound of the sequence. Default is 1 for ascending sequences and 10^28 for descending sequences
MAXVALUE clause provides upper bound. Default is 10^28 for ascending sequences and 1 for descending sequences.
Let us take an example that a sequence number using the sequence command as follows:
CREATE SEQUENCE emp1number INCREMENT BY 1 START WITH 8890;
Sequence created.
(Note: the above example will create a sequence 'emp1number that can be accessed by INSERT and UPDATE).

GENERATING SEQUENCE NUMBERS WITH NEXTVAL

When you reference to the pseudo column nextval a new sequence number is generated.
Following are the point that should be remembered while generating sequence as follows:

- Always prefix NEXTVAL with the sequence name.
- The sequence is incremented, independent of the transaction committing or rolling back.
- Sequence numbers are generated independent of the table, so the same sequence can be used for one for multiple tables.

Let us take an example using the NEXTVAL command as follows:

SELECT EMP1NUMBER.NEXTVAL FROM DUAL;

NEXTVAL
8890
(Note: the above example will insert 8891 in the column emp1no as the sequence start with 8890).

USING SEQUENCE NUMBERS WITH CURRVAL

The pseudo column CURRVAL is used to refer to the current sequence number. When you reference NEXTVAL for a given sequence, the current sequence number is placed in CURRVAL.
Let us take an example using the above example as follows:
INSERT INTO EMP1 (EMPNO, ENAME, SAL) VALUES
(EMP1NUMBER.CURRVAL,'AJAY', 7000);
1 row created.
This will insert 8892 in the column empno, as the CURRVAL is 8892.
Following are the point that should be remembered while using the NEXTVAL and CURRVAL as follows:
NEXTVAL AND CURRVAL can be used within:

- SELECT statements.
- VALUES list of INSERT statements.

- SET clause of UPDATE statements.

NEXTVAL AND CURVVAL cannot be used:

- With the keyword DISTINCT.
- With the ORDER BY GROUP BY or HAVING clause of a select statement.
- With the set operator UNION, INSERTSECT, MINUS.
- Within a subquery.

(Note: the use of the data dictionary ALL_SEQUENCE & USER_SEQUENCE to see all the sequence owned by the user).

NEXTVAL and CURRVAL cannot be used:-

- Within the keyword DISTINCT
- With the ORDERBY, GROUP BY or HAVING clause of a SELECT statement
- With set operators: UNION, INTERSECT, MINUS

ALTERING A SEQUENCE

Altering a sequence command is used to modify an existing sequence. It can be used to change the increment, minimum and maximum values.
Consider the following syntax:
ALTER SEQUNECE <seq-name>
[INCREMENT BY <n>]
[MAXVALUE N| NOMAXVALUE]
[MINVALUE N| NOMINVALUE]
Let us take an example using the above statement as follows:
ALTER SEQUENCE EMP1NUMBER INCREMENT BY 2;
Sequence altered.

REMOVING SEQUENCE

A sequence definition is removed using DROP SEQUENCE command. This statement removes the sequence definition from the data dictionary. One must have privilege to drop a sequence.
Consider the following syntax:
DROP SEQUENCE<sequence name>;

SYNONYMS

Synonyms are defines as follows:

- Synonyms are alternate name or aliases you can give for table, view, sequence, procedure, stored function, package or another synonym.
- Synonyms provide data independence and location transparency.
- Synonyms can be public or private.
- The following SQL statement can refer to synonyms:
- SELECT
- INSERT
- UPDATE
- DELETE
- GRANT
- REVOKE

Public synonyms - Public synonyms are those that can be access by all the users. To create public synonyms you should have create public synonyms system privilege.

Private synonyms - Private synonyms are accessible only within own schema. The name of the private synonyms must be unique to that schema. To create private synonyms in your own schema, you should have created synonyms system privilege.
Consider the following syntax:
CREATE [public] synonym <synonyname>
For <objectname@dblink>
Let us take an example that will create a synonym employee for the table emp in the schema Scott as follows:
CREATE SYNONYM EMPLOYEES
FOR SCOTT.EMP;

DROPPING SYNONYMS

The DROP SYNONYM statement is used to remove a synonym from the database or to change the definition by dropping and recreating it. To drop a private synonym you must have either DROP ANY SYNONYM system privilege.
Consider the following syntax:
DROP SYNONYM <synonyname>

Chapter 11

After completing this chapter, you will be able to understand:

- ✓ **INTRODUCTION TO PL/SQL**
- ✓ **PL/SQL ARCHITERCTURE**
- ✓ **FUNDAMENTAL OF PL/SQL**
- ✓ **PL/SQL DATA TYPE**
- ✓ **DATA TYPE CONVERSION**
- ✓ **SQL IN PL/SQL**
- ✓ **WRITING PL/SQL CODE**

INTRODUCTION TO PL/SQL

PL/SQL is oracle procedural language extension to SQL .It combines the power of procedure language statement and data accessing and manipulation of SQL. It is a powerful transaction control statement.

Advantages of PL/SQL:

- Support for SQL
- Block structure
- Higher productivity
- Better performance
- Integration
- Control structures
- Modularity
- Portability

Support for SQL

PL/SQL allows the use of all the SQL data manipulation, cursor and transaction control commands as well as all the SQL function, operators and pseudocolumn.Thus you can manipulate oracle data flexibly and safely.

Block structure

PL/SQL is a block structure language. That is the basic unit procedure, function and anonymous blocks) that make up the PL/SQL programme are logical book, which can contain any number of nested sub-block that are logically required in the unit.

Higher productivity

PL/SQL adds functionally to non-procedural tools such as forms and report. With PL/SQL in these tools can use familiar procedural constructs to build application. Thus PL/SQL increases productivity by using better tools.

Better performance

Without PL/SQL, oracle server processes SQL statement one at a time. Each SQL statement results in another call to oracle server and higher performance overhead. However with PL/SQL an entire block can be sent to oracle at a time. This can drastically reduce communication between application and oracle server.

Integration

PL/SQL bridges the gap between convenient access to data base technology and the need for procedure programming capabilities by using variable and datatype compatible with those of SQL and with the column in the database itself.

Control structure

Control structures are most important PL/SQL extension to SQL. It controls the procedure flow of the programme deciding if and when SQL and other action are to be executed. This avoids the need to next SQL statement within external programming language.

Modularity

Modularity breaks an application down into manageable, well defined logic modules PL/SQL meets this need with programme unit construct.

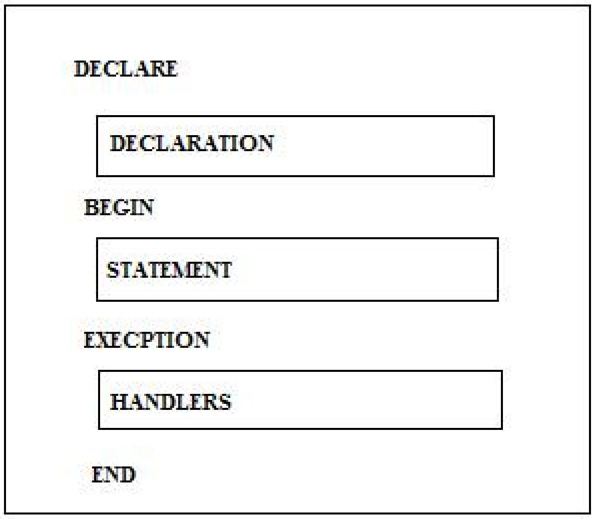
Portability

PL/SQL applications are portable to any operating system and platform in which oracle runs. This gives the advantage of writing potable PL/SQL libraries, which can be reused in different environment.

PL/SQL BLOCK STRUCTURE

PL/SQL is a block structure language. A block allows you to group logically related declarations and statement.
A PL/SQL has three blocks as follows:

- Declaration part
- Executable part
- Exception handling part



Declaration part

This part contains all variables, constraint cursors and user define exceptions that will be referenced within the executable part. This part is optional.

Executable part

This part contains SQL statement to manipulate data in the database and PL/SQL statement manipulates data in the block. This part is mandatory.

Exception handling part

This part specifies the action to perform when error and abnormal conditional arise during execution.

Type of blocks

Every unit of PL/SQL comprise of one or more block PL/SQL block can be of two types are as follows:

Anonymous block: They are unnamed block. They are declared at the point in an application where they are to be declared them either as procedure or as function. You can store programme at the server.

Subprogram: Subprogram are named PL/SQL block that can take parameter and can be invoked. You can declare them either as procedure or as function.

PL/SQL ARCHITECTURE

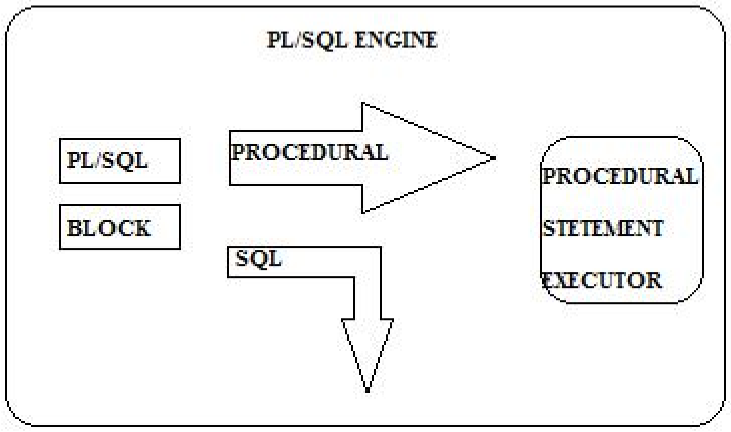
The PL/SQL runtime system is a technology and not an independent product. This technology is actually like an engine that executes PL/SQL block and subprogram. This engine can be installed in an oracle server.

Therefore PL/SQL can reside in two environments as follows:

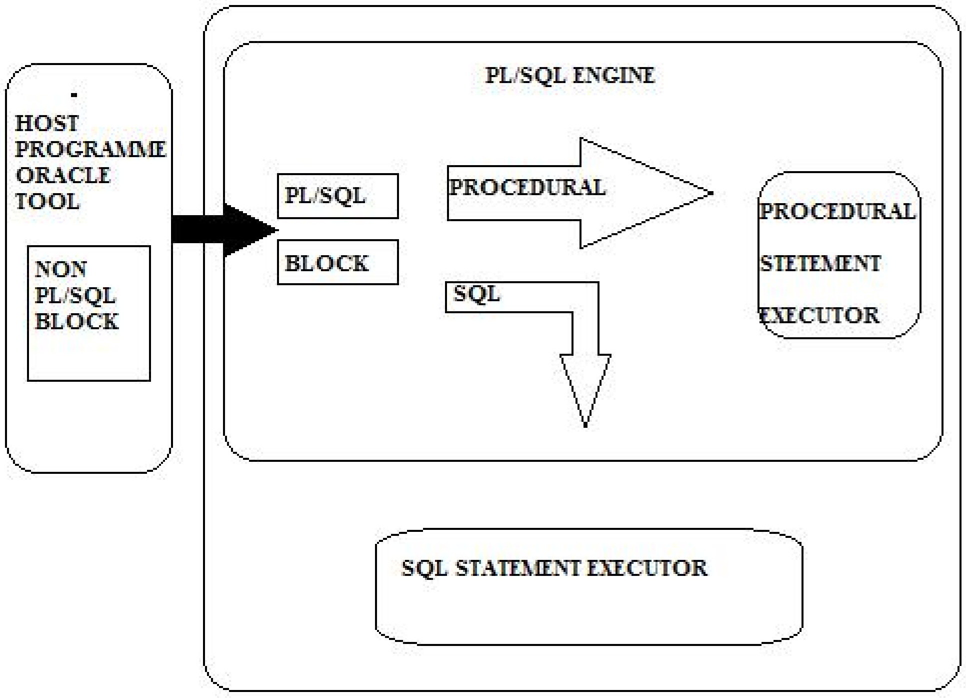
- The oracle server
- The oracle tools

These two environments are independent of each other. In either environment the PL/SQL engine accepts any valid PL/SQL block as input.

The PL/SQL engine executes the procedural part of the statement and sends the SQL statement to the SQL statement executor in the oracle server, thus reducing I/O and improving performance. Application development tools that lack a local PL/SQL engine must rely on oracle to process PL/SQL block and subprogram. The server passes the block and subprogram to its local PL/SQL.



The PL/SQL Engine



The PL/SQL Engine at server end

FUNDAMENTALS OF PL/SQL

Like every other programming language PL/SQL has a character set, reserved word, punctuation , datatype, rigid syntax and fixed rules of usage and statement formation. These elements of PL/SQL are used to represent real-world object and operation as follows:

- Character set
- Reserved word
- Lexical unit
- Delimiters
- Identifiers
- Literals

Character set

PL/SQL programme are written as lines of text a specific set of character.The pl/sql character set includes the following:

- The upper and lower case letter A-Z, a-z.
- The number 0...9
- Tabs, space and carriage return
- The symbols(),+-*!@#$%^&{}"?/[] etc

Reserved word

Reserved words have special syntactic meaning to PL/SQL and so it cannot be redefined. Reserve words are written in upper case to promote readability, but it can be written in lower or mixed case.

For example: BEGIN and END.

Lexical Units

Lexical units (that is identifiers, operator names, literals) may be separated by one or more spaces or other delimiters which would not be confused as part of the lexical unit.

Delimiters

A delimiter is a simple or compound symbol that has a special meaning to PL/SQL. Delimiter can be used to represent arithmetic operation such as additional and subtraction.
Simple Symbol: consist of one character. Such as: + (addition), - (subtraction), = (relational operator), @ (remote access indicator).

Compound Symbol: consist of two characters. Such as: ** (exponentiation), | | (concatenation).

Identifiers

Identifiers can be used to name PL/SQL programme objects and unit, which include constants, variables, exceptions, cursor, subprograms and packages. Identifier must start alphabetically, and may contain up to 30 characters.

Literals

Character and date literals must be enclosed in single quotes.
Numeric literals are of two kind integers and real number as follows:

- An integer literal is an optionally signed whole number without a decimal point

  e.g. 8.76, -675.
- A real literal is an optionally signed whole or fractional number with a decimal point e.g. 8.76, 4.67.

Operator Precedence:

The arithmetic, logical and concatenation operators used in PL/SQL are the same as those in SQL. In addition, there is an exponential operator (**).
The operations within an expression are done in a particular order depending on their priority.

The table shown below shows the default order of operations from first to last.

OPERATOR	OPERATION
** , NOT	Exponential, logical negation
+ , -	Identity, negation
*, /	Multiplication, division
+, -, ||	Addition, subtraction, concatenation
=.!=,< >,<=,>=,IS NULL	Comparison
LIKE, BETWEEN, IN	Comparison
AND	Conjunction
OR	Inclusion

Note : Operators with highest precedence are applied first.

PL/SQL DATA TYPES

PL/SQL supports a variety of data type which may be used for declaring variables and constants. Variables may optionally be assigned a value when declared and are allowed to change their value through further assignment later in the block.

Variables and Constants

Declaring variables: Declaring variable allocates storage space for a value, specify is datatype and name the storage location so that the value can be referenced.
Keep the following in the mind while declaring variable as follows:

- Variable are declared in the declared section of the PL/SQL block.
- Declaration involves the name of the variable followed by its datatype.
- All statement must end with a semicolon.

Consider the following syntax:
<VAR-NAME> <TYPE> [:= <VALUE>];
You can declare the variable as shown below:

```
Mname       CHAR (20);
Age         NUMBER (3);
Deptno      NUMBER (4) := 30;
Pin_code    CONSTANT NUMBER (6) :=110049;
```

Declaring constants

Constants are declared by specifying the keyword CONSTANT before the data type.
Example: Pin_code   CONSTANT NUMBER(6) :=110049;
Using Default
The reserved word Default can be used instead of the assignment operator to initialize variable and constant. Look at the following declaration as follows:
Deptno NUMBER (4) := 30;
Pin_code constant NUMBER (6):=110049;
These declarations can be rewritten as follows:
Deptno NUMBER (4) DEFAULT 30;
Pin _number Constant NUMBER (6) default 110049;

Using % TYPE

To avoid type and size conflict between a variable and the column of a table, the attribute % TYPE is used. Advantage of this method of defining a variable is that, whenever the type and/or size of a column in the table are changed, it is automatically reflected in the variable declaration.
temp_name EMP.ENAME%TYPE;
Here, the variable temp_name will be of the same size and type as that as that of the ename column of emp table.

Using %ROWTYPE

The %ROWTYPE attribute provides a record type that represents a row in a table or view. The record can store an entire row of data selected from the table or fetched by a cursor. In case, variables for the entire row of a table need to be declared, then instead of declaring them individually, the attribute %ROWTYPE is used.

```
  emp_row_var    EMP%ROWTYPE;
```

Here, the variable emp_row_var will be a composite variable, consisting of the column names of the table as its members. To refer to a specific variable, say sal, the following syntax will be used:

```
        emp_row_var.sal: = 5000;
```

Note: a %ROWTYPE declaration cannot include an initialization clause.

Assignments and Expressions

Assignment operator is the most basic operator. The PL/SQL assignment statement allows a value to be assigned or reassigned to a variable in the declare section of the block.
Consider the following syntax:
Variable: = Expression;
An expression is a sequence of variables or literals, separated by operators. Assignments are legal in the executable and exception handling section of a block. The variable to receive the new value must be explicitly named on the left of the assignment operator. Let us take an example with the above example as follows:

```
DECLARE
          str1 VARCHAR (10);
          str2 VARCHAR (15);
          num1 NUMBER;
BEGIN
          str1:='GOOD BYE';
          str2:=str1;
          num1:=34.9;
END;
```

PL/SQL procedure successfully completed.
Variables used before the assignment operator must refer to an actual storage location. Since the expressions are written into it. In the example discussed above the PL/SQL engine will allocate storage for variables, and the values 'GOODBYE' and 34.9 can be put into this storage. The expressions can be content of storage location or a literal. The above example illustrates both the cases 'GOODBYE' is a literal and str1 is a variable.
Note: there can be only one assignment operator in a statement.

REFERENCING NON-PL/SQL VARIABLES

Variables declared in the host environment like SQL *PLUS bind variable may be referenced in PL/SQL statement. Non PL/SQL variable are referenced by prefixing the variable by a colon inside a PL/SQL block. The usage is given below:

```
: var1:= 'ABC';
: var2:=2;
```

Data type conversion

PL/SQL attempts to convert data types dynamically if they are mixed within a statement If a number value is assigned to a char variable, then the number is dynamically translated into character representation so that it can be stored in the char variable.
Comm VARCHAR (12):= 1200;
(Note: =VLAUE_ERROR exception occurs if the values are incompatible).
Comm_per NUMBER (12, 2):= 'comm is 25%'; --------value error

Comment in PL/SQL

Comment in PL/SQL can have as follows:

- A double hyphen (--) preceding the line. The entire line will be treated as a comment. This is used for specifying single line comments.

- A slash followed by asterisk (*/ */).This form of specifying comments can be used to span across multiple lines.

Conditional and iterative control

The conditional controls available with PL/SQL are as follows:

- IF-THEN ELSE statement.

- CASE statement.

The types of loops available with PL/SQL are as follows:

- LOOP-END LOOP

- FOR-LOOP

- WHILE-LOOP

IF –THEN-ELSE statement

The IF clause can be used for the conditional processing of statement. If the condition specified after the if clause evaluates to true, the statement following the THEN clause are executed until one of the following is encountered: ELSEIF, ELSE, ENDIF.
Consider the following syntax:

```
IF<condition1> THEN
<statement1>
[ELSEIF <condition1>THEN
<statement2>]
ELSE
<statement3>
```

END IF;

ELSE clause

The ELSE clause is optional. It should always be accomplished with IF clause.
Consider the following syntax:

```
IF tran_type = 'C' THEN
UPDATE account SET bal = bal + credit WHERE.......
ELSE
UPDATE account SET bal = bal+ credit WHERE........
ELSE IF;
```

ELSIF clause

The ELSIF clause is an optional as ELSE clause. IF first condition evaluates to FALSE the ELSIF test another condition. If statement can have any number of ELSIF command.
Consider the following syntax:

```
IF sales>50000 THEN
Bonus: =1500;
ELSIF sales>3500 THEN
Bonus: =500;
ELSE
Bonus: =100;
ENDIF;
```

CASE STATEMENT

CASE statement also selects one sequence of statement to execute like an IF. Using case statement makes the programme more readable and efficient then lengthy IF THEN statement. The CASE statement begins with the keyword CASE.
Consider the following syntax:

```
CASE<expert>
WHEN<expr1> THEN sequence_of_stmt1;
WHEN<expr2>THEN sequence_of_stmt2;
[ELSE sequence_of_stmt]
END case
```

Consider the following example:-

```
CASE grade
WHEN 'A' THEN remarks:='Excellent';
WHEN 'B' THEN remarks:='Good';
WHEN 'C' THEN remarks:='Fair';
WHEN 'D' THEN remarks:='Poor';
ELSE remarks:='No such grade';
END CASE
```

LOOP (LOOP-ENDLOOP)

The LOOP...END LOOP construct can be used for the repeated processing of statements.
Consider the following syntax:

```
LOOP
<statements>
END LOOP;
```

EXIT statement

The exit statement allows you to control to be passed to the next statement beyond END LOOP, thus ending the loop immediately.
Consider the following syntax:

```
LOOP
ctr := ctr+1;
IF ctr=10 THEN
EXIT;
END IF;
END LOOP;
```

EXIT WHEN Statement

Exit when statement allows a loop to complete conditionally. When the exit statement encountered the condition in the WHEN clause is evaluated .If the condition evaluates to true, the loop complete and the control passed to the next statement after the loop.
Consider the following syntax:

```
LOOP
Ctr := ctr+1;
EXIT WHEN ctr = 10;
END LOOP;
```

FOR LOOP

The FOR loops iterate over a specified range of integer. The range is a part of an iteration scheme, which is enclosed by the keywords FOR and LOOP.
Consider the following syntax:

```
FOR<var> IN <lower>...<upper>LOOP
<Statements>
END LOOP;
```

The index variable <var> has the following properties as follows:

- It is of data type number and need not be declared.
- Its scope is only within the for loop.
- Within the for loop, index variable can be referenced but not modified.

<lower> and <upper> are integer expressions which determine the range of values for the control variable

You can use the FOR.. END FOR construct as shown below:

```
For ctr in 1..20
Loop
   Insert into temp values (ctr);
End Loop;
```

For reverse loop you can use REVERSE keyword:

```
For ctr in reverse 1..20
Loop
Insert into temp values (ctr);
End Loop;
```

WHILE LOOP

The while loop tests the condition provided and if it evaluates to true, then the statement within the loop and end loop are executed. The loop continues as long as the condition is true.

Consider the following syntax:

```
WHILE<condition> LOOP
<Statements>
END LOOP;
```

Example;-

```
WHILE i<=100
Loop
total:= total +i;
End loop
```

NESTED loop and LABELS

Loops can be nested to multiple levels. You can nest FOR, WHILE and simple LOOP within one another. The termination of a nested loop does not terminate the enclosing loop unless an exception was raised.

Using labels:

The labels are a place before the statement, either on the same line or on a separate line. You can label loops by placing the table before the loop statement with in the labels delimiters.

Let us take an example that will use the nested loop and the labels as shown below as follows:

```
BEGIN
<<outer_loop>>
LOOP
ctr:=ctr+1;
<<Inner_loop>>
LOOP
total:=total+ctr;
EXIT outer_loop when total_done='YES';
EXIT when Inner_done='YES';
END loop Inner_loop;
END loop Outer_loop;
END;
```

SQL IN PL/SQL

Whenever any sort of data manipulate has to be done using database or extract information from the database, SQL command are used. SQL commands allowed in PL/SQL program are DML and transaction control statement. DML commands can process multiple rows.
Following the some point that should be kept in mind while using PL/SQL block:

- A PL/SQL block is not a transaction unit, commits savepoints and rollbacks are independent of blocks. These commands can be issued within a block.
- PL/SQL block does not support data definition language (DDL) statement such as CREATE TABLE, ALTER or DROP TABLE etc directly.
- PL/SQL does not support data control language such as GRANT or REVOKE**.**

WRITING PL/SQL CODE

PL/SQL code is written using any text editor. The PL/SQL programme is compiled and executed using the command @<filename>.
dbms_output.put_line
The procedure dbms_output.put_line will show the output on the screen. It accepts only one argument. Hence the different variables are concatenated with double pipe (| |) symbol. To enable the server output, the SET SERVEROUTPUT ON command must be given at the SQL prompt to the execution of the dbms_output.put_line function.
Consider the following programme as follows:

1) The following example shows a PL/SQL code to update salary of employee number '7788' to 3000 if salary is less than 3000.

```
DECLARE
        X NUMBER (7, 2);
BEGIN
        SELECT sal INTO X FROM emp1 WHERE EMPNO=7788;
        IF X < 3000 THEN
                UPDATE EMP1 SET SAL=3000
                WHERE EMPNO=7788;
        END IF;
        COMMIT;
END;
```

2) The following PL/SQL code insert all the details of employee number 7 698 to a new table temp that has the same structure as emp1 table.

```
DELCARE
newrec emp1%ROWTYPE;
BEGIN
SELECT * INTO newrec from emp1 WHERE empno = 7698;
INSERT INTO TEMP;
VALUES (newrec.empno, newrec.ename, newrec.job, newrec.mgr, newrec.hiredate, newrec.sal,
newrec.comm, newrec.deptno);
COMMIT;
END;
```

3) The following program PL/SQL code print first 50 whole number. Also insert the list in temp1 table.

```
DECLARE
        Ctr NUMBER(2):=0;
BEGIN
        WHILE ctr<=50 LOOP
                Dbms_output.put_line (ctr);
                INSERT into temp1 values (ctr);
                Ctr:=ctr+1;
        END LOOP;
        COMMIT;
END;
```

4) The following PL/SQL code updates the commission of the employee number 7369 to 300,if it is NULL else raise the commission by 25%.

```
DECLARE
Var_empno NUMBER (4):=73 69;
Var_comm emp1.comm%TYPE;
BEGIN
        SELECT COMM INTO VAR_COMM FROM EMP1
        WHERE EMPNO=VAR_EMPNO;
        IF VAR_COMM IS NULL THEN
                UPDATE EMP1 SET COMM=300
                WHERE EMPNO=VAR_EMPNO;
        ELSE
                VAR_COMM:=VAR_EMPNO+VAR_COMM*.25;
                UPDATE EMP1 SET COMM=VAR_COMM
                WHERE EMPNO=VAR_EMPNO;
        END IF;
COMMIT;
END;
```

Composite datatype

Composite datatype are also known as collection. Composite datatype in oracle are as follows:

- RECORD
- TABLE

PL/SQL Records

A record is a group of related data item stored in the field, each with its own name and datatype. These data can be grouped as a singled unit, using PL/SQL record data type.
Following are some point that should be kept in mind while using PL/SQL:

- Each record defined can have as many fields as necessary.
- Records can be assigned initial values and can be dined as NOT NULL.
- Field without initial values are initialized to NULL.
- The default keyword can also be used-define records in the declarative part of any block, subprogram or package.
- You can declare and referenced nested records. A record can be the component of another recorded.

Consider the following syntax:
TYPE<type-name> IS RECORD
(<field –name1> {field -type| variable%TYPE |table.column%TYPE |table%ROWTYPE}
<Field-name2> {field-type|variable%TYPE|table.column%TYPE||table%ROWTYPE}
....
);
WHERE

<type-name>	is the record name
<field-type>	is any datatype including record datatype.

Consider the following declarative section:-
DECLARE
Stud_id NUMBER (5);
V_firstname VARCHAR2 (20);
V_lastname VARCHAR2 (20);
Now we will declare a record type using the above variables:-
Type stud_rec is RECORD (
Stud_id NUMBER (5);
V_firstname varchar2 (20);
V_lastname varchar2 (20));
/* declare a variable of this type*/
V_studrec stud_rec;
To reference individual fields in a record you can use dot notation as used with records defined with %ROWTYPE

record_name.field_name
For example to refer V-lastname we can write:

```
BEGIN
dbms.output.put.line (V_studrec.V_lastname);
END;
```

Let us see an example that will display the total salary which includes commission of the empno 7369.It should also display employee name, his department details and his old and new salary of the employee.

```
DECLARE
TYPE deptrec is record
(dno dept.deptno%TYPE,
vname dept.dname%TYPE,
vloc dept.loc%TYPE,
name emp1.ename%TYPE,
vsal emp1.sal%TYPE,
vcom emp1.comm%TYPE,
newsal emp1.sal%TYPE);
dept_det Deptrec;

BEGIN
select ename,sal,comm,dept.deptno,dname,loc into dept_det.name,
dept_det.vsal,dept_det.vcom,dept_det.dno,dept_det.vdname,dept_det.vloc
from emp,dept
where emp.deptno=dept.deptno
and empno=7369;
dept_det.newsal:=dept_det.vsal +NVL(dept_det.vcom,0);
dbms.output.put.line (dept_det.dno||dept_det.vdname||dept_det.vloc||
dept_det.name||dept_det.vsal||dept_det.vcom|| dept_det.newsal1);
END;
```

Record Assignment

In order to assign one record to another, both records must be of the same type. Even if the field of the record of different types match exactly, they cannot be assigned to each other.
Let us take an example that using the above statement as follows:

```
DECLARE
TYPE rec1 IS RECORD
dept_num NUMBER(2),
dept_name VARCHAR214));
TYPE rec2 IS RECORD(
dept_num NUMBER(2),
dept_name VARCHAR2(14));
r1 rec1;
r2 rec2;
BEGIN
```

```
r1:=r2;--illegal; different datatypes
END;
```

However you can assign a %ROWTYPE record to a user_define record if their field matches in number and order and corresponding fields have compatible datatype.

Let us take another example using the above statement as follows:

```
DECLARE
TYPE deptrecord IS RECORD
dept_num dept.deotno%type,
dept_name dept.dname%type,
dept_loc dept.loc%type);
deptrec1 deptrecord;
deptrec2 dept%rowtype;
BEGIN
select * INTO deptrec2 from dept WHERE deptno=20;
deptrec1:=deptrec2;
dbms_output.put_line (deptrec1.dept_num|| ' '||
deptrec1.dept_name|| ' '||deptrec1.dept_loc);
END;
```

Here you are declaring a variable of PL/SQL record data type and a variable of %ROWTYPE. Both of them have their field match in number, order and also have compatible data type.

PL/SQL tables

PL/SQL tables are modelled as database tables, but are actually not. Primary keys can be associated with them to have array like access to row. The size can be dynamically increased by adding more rows when required.
The tables are declared in two steps as follows:

- **Define a table type:**

Consider the following syntax:

```
TYPE<type-name> IS TABLE OF
<column-type| variable%type | table.column%type> [NOTNULL]
INDEX BY BINARY_INTEGER;
```

Where
<type-name> is specifier used in subsequent declaration of PL/SQL table.
<column_type> is any scalar (not composite) datatype such as CHAR, DATE or NUMBER
%TYPE attribute can be used to specify a column datatype.

- **Declare a variable of table type in declare section.**

  Example: Defining a table type EMPNAMETYPE:

```
TYPE EMPNAMETYPE IS TABLE OF EMP.ENAME%TYPE NOT NULL INDEX BY
BINARY_INTEGER;
```

Declaring a variable of table type in declares section:

EMPLIST EMPNAMETYPE;
To reference rows in PL/SQL table, you specify a primary key value using the array.
The syntax is:-
PL/SQL_table_name (primary_key_value)
Where primary_key_value belongs to BINARY_INTEGER
Example:
ENAMELIST (subscript)
Where subscript belongs to BINARY_INTEGER.
(Note: The PL/SQL table cannot be initialized in their declaration).

Using the FORALL statement

The keyword FORALL statement instructs the PL/SQL engine to bulk-bind input before sending them to the SQL engine. The FORALL statement also contain iteration method like FOR, LOOP etc but it is not a FOR loop.
Consider the following syntax:
FORALL index IN lower_bound...upper_bound
Sql_statement;
Where the index can be referenced only within the FORALL statement and only as a collection subscript.
Sql_statement can be an INSERT, UPDATE or DELETE statement that references the collection elements.
The lower_bound and upper_bound must specify a valid range of consecutive index numbers.
Let us take an example that will populate the PL/SQL table with employee names and salaries. Then we will insert the empno of the entire employee loaded in the PL/SQL table into another table 'Temp' using FORALL statement. We will also display the content of the table as follow:

```
DECLARE
TYPE EMPNAMETYPE IS TABLE OF EMP.ENAME%TYPE NOTNULL
TYPE EMPSALTYPE IS TABLE OF EMP.SAL%TYPE
INDEX BY BINARY_INTEGER;
NAMELIST EMPNAMETYPE;
SALARYLIST EMPSALTYPE;
SUBSCRIPT BINARY_INTEGER:=1;
CTR NUMBER (2):=1;
BEGIN
FOR EMPREC IN (SELECT ENAME, SAL FROM EMP) LOOP
ENAMELIST (SUBSCRIPT):=EMPREC.ENAME;
SALARYLIST (SUBSCRIPT):=EMPREC.SAL;
SUBSCRIPT: =SUBSCRIPT+1;
END LOOP;
FORALL I IN 1..NAMELIST.COUNT
INSERT INTO EMP
SELECT EMPNO FROM EMP WHERE ENAME=NAMELIST (I);
WHILE ctr<SUBSRIPT
LOOP
DBMS_OUTPUT.PUT_LINE(NAMELIST(ctr));
DBMS_OUTPUT.PUT_LINE(SALARYLIST(ctr));
ctr:=ctr+1;
END LOOP;
END;
```

In the above example the SQL engine executes the INSERT statement once for each index number in the specific range, that is once for every row in the emp table.

Chapter 12

After completing this chapter, you will be able to understand:

- ✓ **CURSOR MANAGEMENT IN PL/SQL**
- ✓ **BULK COLLECT**
- ✓ **EXCEPTION HANDLING IN PL/SQL**
- ✓ **PREDEFINED EXCEPTION AND USER DEFINED EXCEPTION**
- ✓ **NON PREDEFINED EXCEPTION**
- ✓ **PROPAGATION OF EXCEPTION**

For every SQL statement execution certain area in memory is allocated. PL/SQL allows you to name this area. This private SQL area is called context area or cursor. A cursor is a PL/SQL construct that allow you to name these work areas and access their stored information. A cursor acts as a handle or pointer to the context area.

A PL/SQL program controls the context area using the cursor. Cursor represents a structure in memory and is different from cursor variable.

When you declare a cursor, you get a pointer variable, which does not point any thing. When the cursor is opened, memory is allocated and the cursor structure is created. The cursor variable now points the cursor. When the cursor is closed the memory allocated for the cursor is released.

Cursors allow the programmer to retrieve data from a table and perform actions on that data one row at a time. There are two types of cursors implicit cursors and explicit cursors.

Implicit cursors

For SQL queries returning single row PL/SQL declares implicit cursors. Implicit cursors are simple SELECT statements and are written in the BEGIN block (executable section) of the PL/SQL. Implicit cursors are easy to code, and they retrieve exactly one row. PL/SQL implicitly declares cursors for all DML statements. The most commonly raised exceptions here are

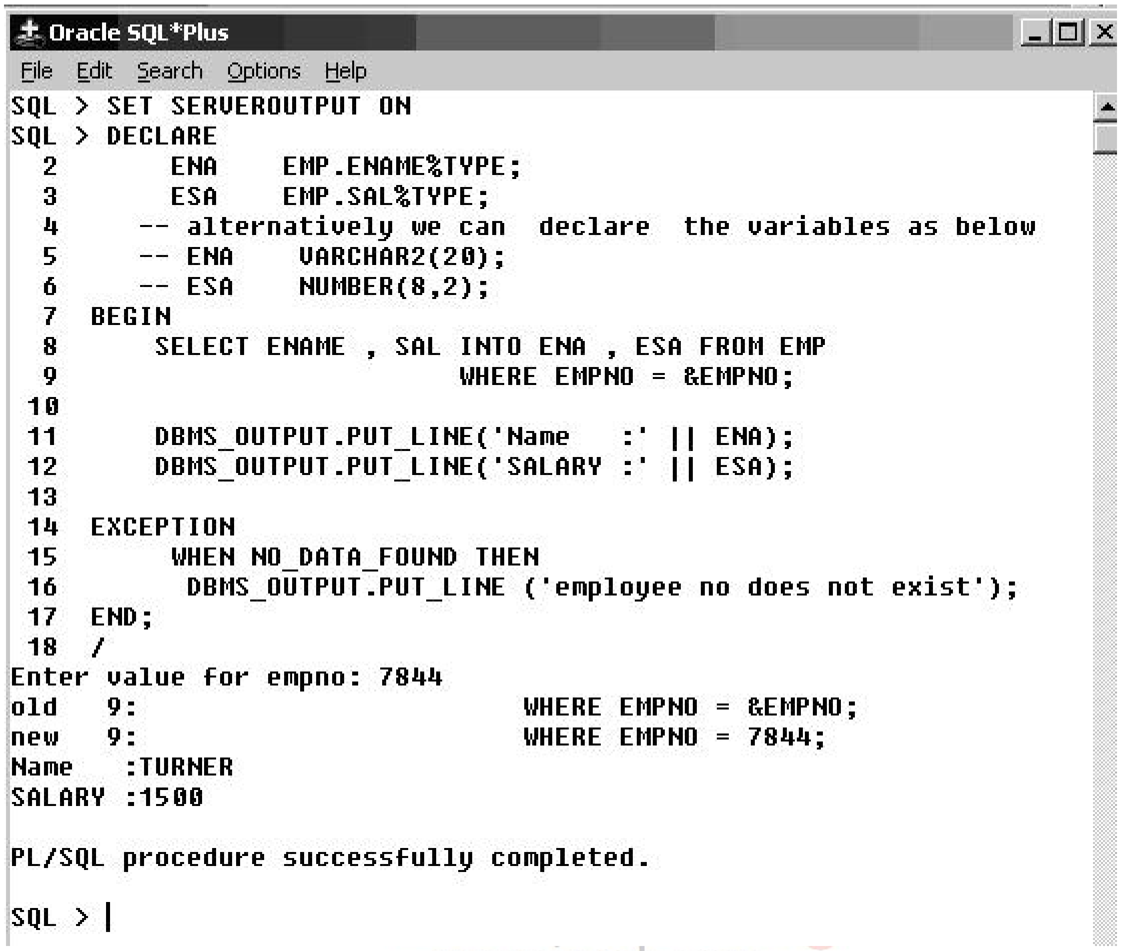
NO_DATA_FOUND or TOO_MANY_ROWS.

Syntax:

SELECT ename, sal INTO ena, esa FROM EMP WHERE EMPNO = 7844;

Note: Ename and sal are columns of the table EMP and ena and esa are the variables used to store ename and sal fetched by the query.

used to store ename and sal fetched by the query.



```
SQL > SET SERVEROUTPUT ON
SQL > DECLARE
          ENA     EMP.ENAME%TYPE;
          ESA     EMP.SAL%TYPE;
        -- alternatively we can  declare  the variables as below
        -- ENA     VARCHAR2(20);
        -- ESA     NUMBER(8,2);
     BEGIN
         SELECT ENAME , SAL INTO ENA , ESA FROM EMP
                            WHERE EMPNO = &EMPNO;

         DBMS_OUTPUT.PUT_LINE('Name   :' || ENA);
         DBMS_OUTPUT.PUT_LINE('SALARY :' || ESA);

     EXCEPTION
          WHEN NO_DATA_FOUND THEN
           DBMS_OUTPUT.PUT_LINE ('employee no does not exist');
     END;
     /
Enter value for empno: 7844
old   9:                        WHERE EMPNO = &EMPNO;
new   9:                        WHERE EMPNO = 7844;
Name    :TURNER
SALARY :1500

PL/SQL procedure successfully completed.

SQL > |
```

Explicit Cursors

Explicit cursors are used in queries that return multiple rows. The set of rows fetched by a query is called active set. The size of the active set meets the search criteria in the select statement. Explicit cursor is declared in the DECLARE section of PL/SQL program.

Explicit Cursor management is done through DECLARE, OPEN, FETCH and CLOSE statements.

Declaring a cursor

Syntax:

CURSOR <cursor-name> IS <select statement>

Sample Code:

```
DECLARE
CURSOR emp_cur IS SELECT ename FROM EMP;
BEGIN
----
---
END;
```

Opening Cursor

Syntax: OPEN <cursor-name>;

Example: OPEN emp_cur;

When a cursor is opened the active set is determined, the rows satisfying the where clause in the select statement are added to the active set. A pointer is established and cursor now points to the first row in the active set.

Retrieving individual row

Fetching from the cursor: To get the next row from the cursor we need to use fetch statement.

Syntax: FETCH <cursor-name> INTO <variables>;

Example: FETCH emp_cur INTO ena;

FETCH statement retrieves one row at a time.

Closing the cursor

After retrieving all the rows from active set the cursor should be closed. Resources allocated for the cursor are now freed. Once the cursor is closed the execution of fetch statement will lead to errors.

CLOSE <cursor-name>**;**

Explicit Cursor Attributes

Every cursor defined by the user has four attributes for obtaining the status information about the cursor

The attributes are:

1. %NOTFOUND: It is a Boolean attribute, which evaluates to true, if the last fetch failed. i.e. when there are no rows left in the cursor to fetch.
2. %FOUND: Boolean variable, which evaluates to true if the last fetch, succeeded.
3. %ROWCOUNT: It's a numeric attribute, which returns number of rows fetched by the cursor so far.
4. %ISOPEN: A Boolean variable, which evaluates to true if the cursor, is opened otherwise to false.

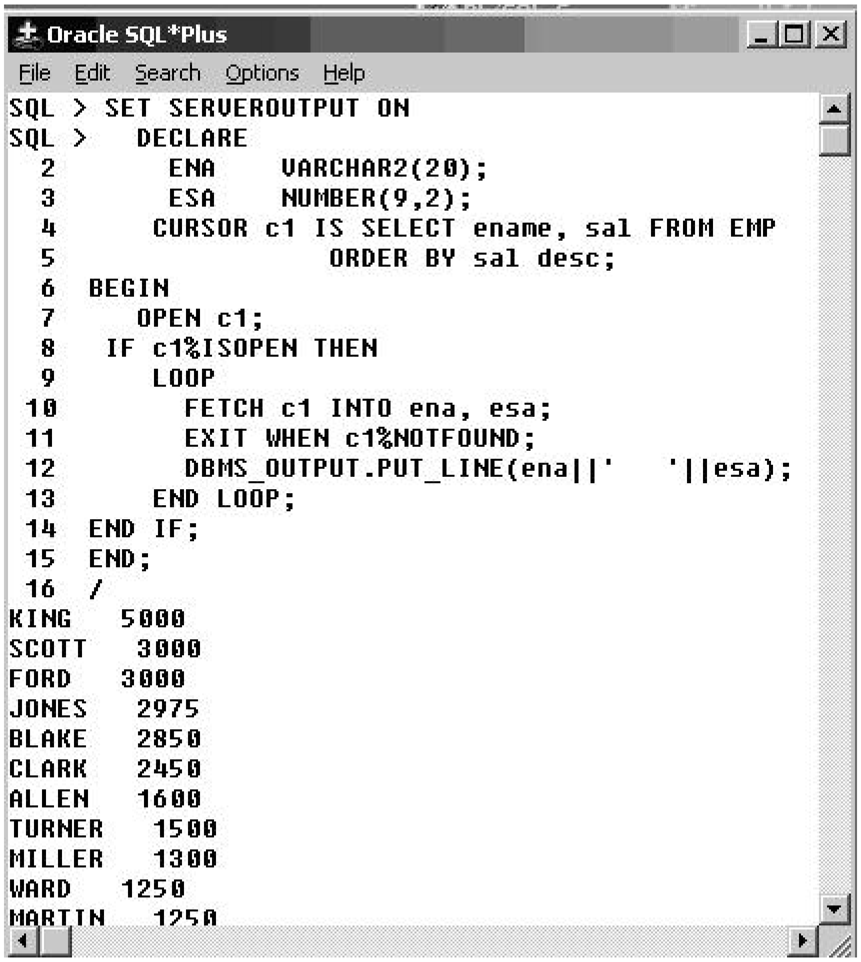
Consider the following cursor which displays the details of those employees having salary more than 2000

```
/*using cursor display the details of those employees having salary more than 2000*/

--set serveroutput on;

declare

vempno  emp.empno%type;

vename          emp.ename%type;

vsal    emp.sal%type;

vdeptno emp.deptno%type;

CURSOR c1 is

select empno,ename,sal,deptno from emp where sal>2000;

begin

        OPEN c1;

        LOOP

                Fetch c1 into vempno,vename,vsal,vdeptno;

                if c1% FOUND  then

                        dbms_output.put_line('EMP no '|| vempno || ' Empname ' || vename  || ' Sal '
                        || vsal || ' dept no ' || vdeptno);

                else
                exit;

            end if;

        end loop;

        CLOSE c1;

end;
```

Using WHILE:

While LOOP can be used as shown in the following example for accessing the cursor values.

Example:



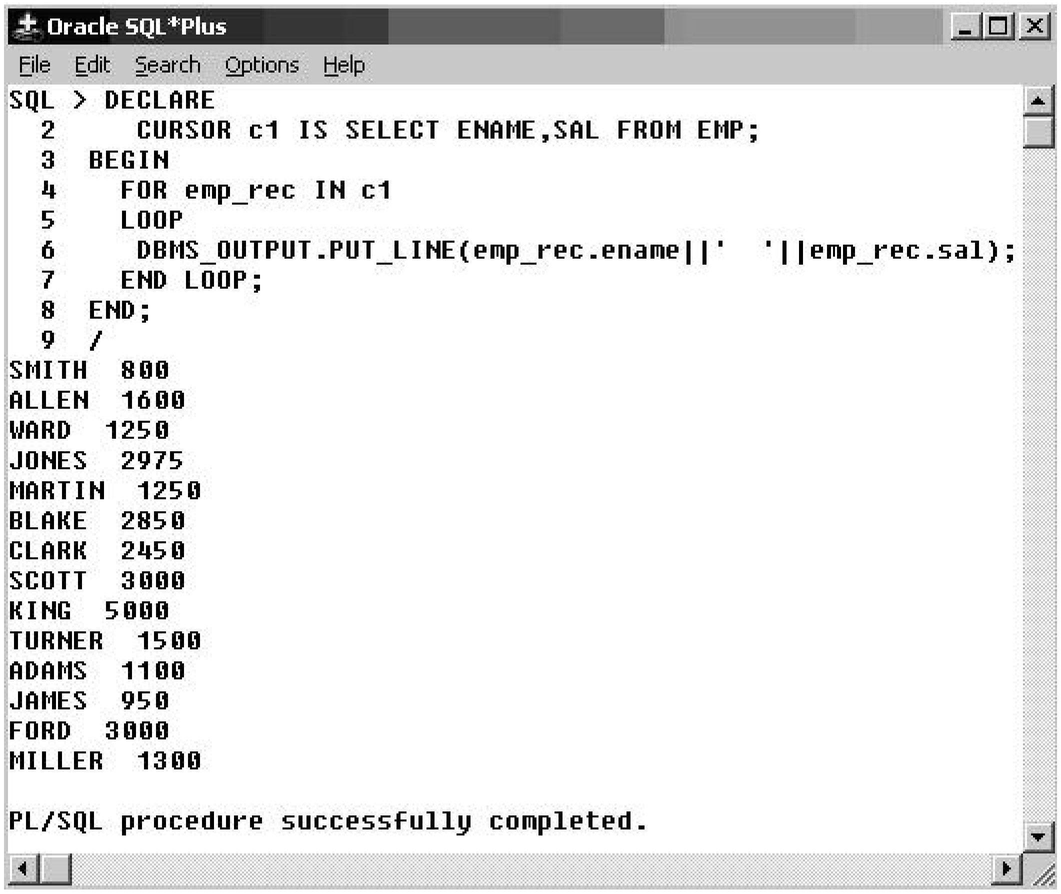
```
SQL > SET SERVEROUTPUT ON
SQL >   DECLARE
          ENA    VARCHAR2(20);
          ESA    NUMBER(9,2);
         CURSOR c1 IS SELECT ename, sal FROM EMP
                    ORDER BY sal desc;
     BEGIN
        OPEN c1;
      IF c1%ISOPEN THEN
         LOOP
           FETCH c1 INTO ena, esa;
           EXIT WHEN c1%NOTFOUND;
           DBMS_OUTPUT.PUT_LINE(ena||'    '||esa);
         END LOOP;
     END IF;
     END;
     /
KING   5000
SCOTT   3000
FORD   3000
JONES   2975
BLAKE   2850
CLARK   2450
ALLEN   1600
TURNER   1500
MILLER   1300
WARD   1250
MARTIN   1250
```

Using Cursor For Loop:

The cursor for Loop can be used to process multiple records. There are two benefits with cursor for Loop

1. A cursor FOR loop implicitly declares a %ROWTYPE variable, also uses it as LOOP index.
2. Cursor FOR Loop itself opens a cursor read records then closes the cursor automatically. Hence OPEN, FETCH and CLOSE statements are not necessary in it.

Example: consider the following example which displays emp name and their salary



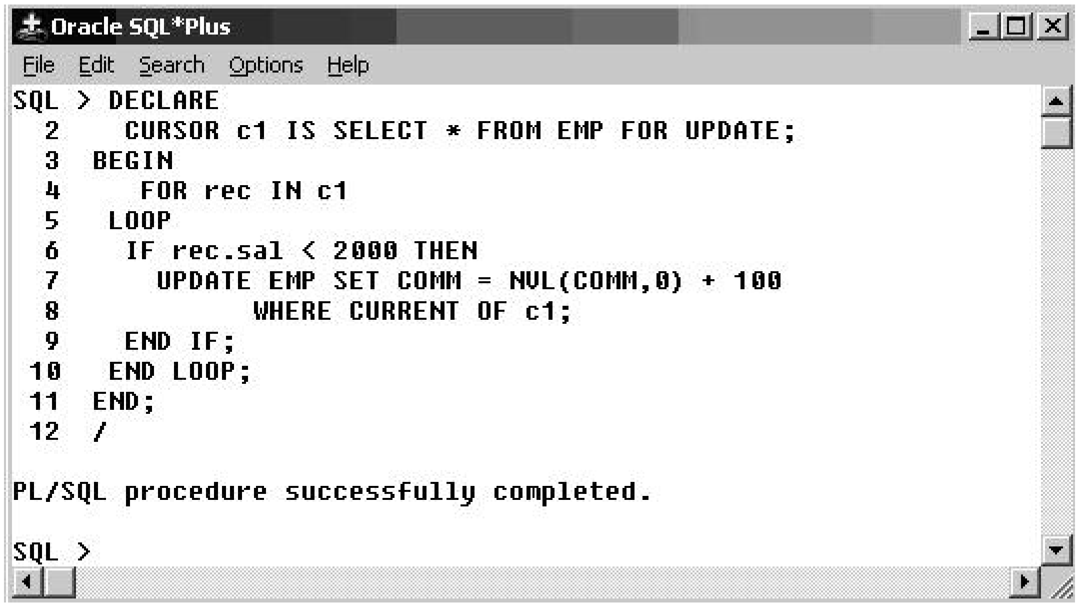
```
SQL > DECLARE
  2     CURSOR c1 IS SELECT ENAME,SAL FROM EMP;
  3   BEGIN
  4     FOR emp_rec IN c1
  5     LOOP
  6      DBMS_OUTPUT.PUT_LINE(emp_rec.ename||'  '||emp_rec.sal);
  7     END LOOP;
  8   END;
  9   /
SMITH  800
ALLEN  1600
WARD  1250
JONES  2975
MARTIN  1250
BLAKE  2850
CLARK  2450
SCOTT  3000
KING  5000
TURNER  1500
ADAMS  1100
JAMES  950
FORD  3000
MILLER  1300

PL/SQL procedure successfully completed.
```

emp_rec is automatically created variable of %ROWTYPE. We have not used OPEN, FETCH, and CLOSE in the above example as for cursor loop does it automatically.

<u>Deletion or Updation Using Cursor:</u>

In order to Update or Delete rows, the cursor must be defined with the FOR UPDATE clause. The Update or Delete statement must be declared with WHERE CURRENT OF

Following example updates comm of all employees with salary less than 2000 by adding 100 to existing comm.



```
SQL > DECLARE
  2    CURSOR c1 IS SELECT * FROM EMP FOR UPDATE;
  3  BEGIN
  4     FOR rec IN c1
  5   LOOP
  6    IF rec.sal < 2000 THEN
  7      UPDATE EMP SET COMM = NVL(COMM,0) + 100
  8            WHERE CURRENT OF c1;
  9    END IF;
 10   END LOOP;
 11  END;
 12  /

PL/SQL procedure successfully completed.

SQL >
```

IMPLICIT CURSOR ATTRIBUTES

- SQL%NOTFOUND: is true if statement was not successful.
- SQL%FOUND: is true if the DML statement was successful.
- SQL%ROWCOUNT: return the number of row affected by an INSERT, UPDATE, DELETE OR single row select.
- SQL%ISOPEN: Is always false because oracle automatically closes an implicit cursor after executing its SQL statement.
- OPEN, FETCH and CLOSE: cannot be used to manipulation the implicit cursor SQL.

SQL%NOTFOUND

This statement evaluates the true if an INSERT, UPDATE or DELETE affected no rows or a SELECT INTO return no rows. Otherwise SQL%NOTFOUND evaluates to false.
Follow the example:
………
……….

UPDATE emp set sal = sal+ (sal*0.5) where empno=7345;
IF SQL%NOTFOUND THEN
INSERT INTO temp values (empno, ename, sal, deptno);

END IF;

SQL%FOUND

This statement is the logical opposite of SQL%NOTFOUND. Until an SQL statement manipulation statement is executed, SQL%FOUND evaluates to NULL. This statement evaluates the TRUE if an INSERT, UPDATE or DELETE affects one or more rows or a SELECT INTO returns one or more rows, otherwise SQL%FOUND evaluates to FALSE
In following example, SQL%FOUND is used to insert a row if a deletion is successful.

```
........
........
DELETE FROM emp where empno= '7119';
If SQL%FOUND THEN
INSERT INTO emp values
(&empno, ename, &job, &mgr, &hiredate, &sal, &comm, &deptno);
END IF;
```

SQL%ROWCOUNT

This above statement return the number of the row effected by an INSERT,UPDATE and DELETE or returned by a SELECT INTO.SQL%ROWCOUNT return a zero if an INSERT,UPDATE or DELETE affected no rows or a SELECT INTO returned no row.
Follow the example:

```
.........
.........
DELETE FROM EMP WHERE
IF SQL%ROWCOUNT > 10 THEN
......
END IF;
.........
........
```

- SQL%ISOPEN

ORACLE closes the SQL cursor automatically after executing its associated SQL statement. As a result SQL%ISOPEN always evaluates to FALSE.

BULK COLLECT

The bulk collect clause let you bulk-bind entire column of Oracle data. That way you can fetch all rows from the result set at once. This clause can be used in SELECT INTO, FETCH INTO and RETURING INTO CLAUSE.
Consider the following syntax:
.....BULK COLLECT INTO<collection1> [,<collection2>,.........]
The SQL engine bulk binds all collections referenced in the INTO list. The corresponding columns can store scalar or composite values including objects.

Let us take an example where the SQL engine load the entire empno and ename database column into nested tables before returning the tables to the PL/SQL engine and then print the result as follows:

```
DECLARE
TYPE  numtab IS TABLE OF emp.empno%TYPE;
TYPE  nametab TABLE OF emp.ename%TYPE;
Enums numbtab;
Names nametab;
BEGIN
SELECT empno,ename BULK COLLECT INTO Enums, Names FROM emp;
FOR I in 1.. Enums.count
LOOP
dbms_output.put_line(Enums(i)|| ' ' || Names(i));
END LOOP;
END;
```

(Note: bulk collect clause is also used for bulk fetch from cursor. We can use this clause with fetch into clause).
Let us see another example that will show you the bulk fetch from a cursor into one or more collection as follows:

```
DECLARE
TYPE  namelist IS TABLE OF emp.empname%TYPE;
TYPE  sallist  IS TABLE OF emp.sal%TYPE;
Cursor c1 IS SELECT ename,sal from emp where sal>1000;
Name Namelist;
Sals Sallist;
BEGIN
OPEN C1;
FETCH BULK COLLECT INTO Name, Sals;
FOR I in 1.. Name.count
LOOP
dbms_output.put_line (Name (i) ||''|| Sals (i));
END LOOP;
END;
```

<u>Retrieving DML results into a collection with the RETURNING INTO clause.</u>

You can use BULK COLLECT clause in the RETURNING INTO clause of an INSERT, UPDATE or DELETE statement as the following example shows:
In the following example 'empno' field of delete records of deleted records of 'emp' table is stored in 'del_emp' table which has only one column 'del_empno'.

```
DECLARE
TYPE Numlist IS TABLE OF emp.empno%TYPE;
Enums Numlist;
BEGIN
DELETE FROM emp WHERE deptno=20
RETURING empno BULK COLLECT INTO Enums;
```

```
FOR i in 1..Enums.COUNT
LOOP
INSERT INTO del_emp VALUES (Enums (i));
Dbms_output.put_line (Enums (i));
END LOOP;
END;
```

Restriction on bulk collection as follow:

- You can use the bulk collect clause only in server-side programme like function, packages, not in client-side programme.
- All targets in a BULK COLLECT INTO clause must be collections.
- You cannot bulk-fetch from a cursor into a collection of records.

EXCEPTION HANDLING IN PL/SQL

An exception is an indefinite in PL/SQL raised during the execution of a block that terminates its main body of actions. A block always terminates when PL/SQL raises an exception handler to perform final action. When an error occurs an exception is raised i.e. the normal execution is handler then the exception handling part of the PL/SQL block.

Exceptions are of three types as follow:

- Predefine oracle Server: Oracle has several predefine exception that correspond to the most common oracle error. Predefine exception are declared in standard packages same as predefine type.
- Non-predefine oracle server: There are some Oracle errors for which are not declared by Oracle. Such error can associated a named exception in a declarative section.
- User Define: A condition that the developer determines is abnormal can be treated as an exception. As that condition is not an oracle error so the programme has to declare the exception and raise explicitly.

Advantages of Exception as follows:

- Without exception handling, every time a command is issued a check must be made for exception error. With exception error are handle conveniently without the need to code multiple check.
- Exception improves readability by isolating error-handling routine.
- Exception improve reliability .one need not worry about checking for an error at every point it might occur. If a PL/SQL block contain an exception handler and if an exception is ever raised in that block the exception handler will handle the exception.

- Helps successful execution of a block called from another PL/SQL block.

Predefine Exception

Predefine exception are internally defined by runtime system. These exceptions are raised implicitly. They are raised whenever PL/SQL program violates an ORACLE rule or exceeds a system-dependent limit. Every Oracle error has a number, but exceptions must be handled by name. Thus, PL/SQL predefines some common ORACLE errors as exception.

Exception Handler

If any type of exception is raised, control is passed to the exception section of the block in which the exception occurred. If the exception is not handled there or if there is no exception section at all then the block terminates with an exception, which may effect the enclosing environment. The same exception cannot be declared more than once in the same PL/SQL block.
Consider the following syntax:
WHEN<exception-identifier>THEN <actions>
Where 'action' may be one or more PL/SQL or SQL statement each terminated by a semi colon. The actions of one exception handler are determined by either the end of the block (END) or by the start of another exception handler (WHEN).
The main exceptions that are likely to occur as a result of a SELECT statement are NO_DATA_FOUND (no rows returned) and TOO_MANY_ROWS (more than one row returned).
(Note: There is no need to give any command to raise internal exception. Internal exceptions are raised automatically).

Let us take an example that will display the name of that manager who has joined the organisation in the year 1999.If there is no output or if there is more than one output then instead of giving error the programme will give proper message as follow:

```
DECLARE
        Vename emp.ename%TYPE;
        Vjob emp.job%TYPE;
BEGIN
        SELECT ename, job ITNO vename, vjob, FROM emp
        WHERE hiredate between '01-JAN-97' AND '31-DEC-97';
        Dbms_output.put_line (vename||''||vjob);
EXCEPTION
        WHEN NO_DATA_FOUND THEN dbms_output.put_line ('No employee hired in 97')
        WHEN TOO_MANY_ROWS THEN dbms_output.put_line ('More than one manager
        has joined in 97');
END;
```

(Note: Several exception handlers mat be defined for the block each with their own set of action. However when an exception occurs only one handler will be processed before leaving the block).
Some of the predefined exceptions are:

EXCEPTION NAME	ORACLE ERROR
CURSOR_ALREADY_OPEN	ORA-06511
DUP_VAL_ON_INDEX	ORA-00001
INVALID_CURSOR	ORA-01001
INVALID_NUMBER	ORA-01722
LOGIN_DEFINE	ORA-01017
NO_DATA_FOUND	ORA-01403
NOT_LOGGED_ON	ORA-01012
PROGRAM_ERROR	ORA-06501
STORAGE_ERROR	ORA-65000
TIMEOUT_ON_RESOURCE	ORA-00051
TOO_MANY_ROW	ORA-01422
TRANSACTION_BACKED_OUT	ORA-00061
VALUE_ERROR	ORA-06502
ZERO_DIVIDE	ORA-01476

The 'WHEN OTHERS' exception Handler

Although the exception section in the above example would trap the two exception specific other type of exception are ignored. Rather than during a separate handler for every exception type 'WHEN OTHERS' exception handler is defined, this handles all errors in the block.
Consider the same example mentioned above. Apart from the two error mentioned in the exception handler if any other error occurs then that error can be trapped by 'WHEN OTHERS' exception handler.

```
.........................
.........................
EXCEPTION
WHEN NO_DATA_FOUND THEN dbms_output.put_line ('No employee hired in 97');
WHEN TOO_MANY_ROWS THEN dbms_output.put_line ('More than one manager has joined in 97');
WHEN OTHER THEN dbms_output.put_line (' Error during block execution');
........................
........................
```

(Note: when an exception has occurred one wants to evaluate the associated error. This is more relevant when 'WHEN OTHERS' exception is handled as it help to know what error has exactly occurred).
PL/SQL has two functions for this purpose as follows:
SQLCODE: It returns the error number associated with the exception that has occurred. If used outside an exception handler the function will return zero. The error number returned may be assigned to a number variable with default precision.
SQLERRM: It returns character data. It return the complete error message associated with the exception including the error number.

Consider the following programme from the above statement as follows:

```
DECLARE
Error_message VARCHAR2 (100);
Error_code NUMBER;
BEGIN
...........
..........
EXCEPTION
WHEN OTHER THEN
Error_message:=SUBSTR (SQLERRM,1, 100);
Dbms_output.put_line (error_message||''||error_code);
END;
```

DECLARING AN EXCEPTION

Exception can be declared only in the declarative part of a PL/SQL block sub program or package.
Consider the following syntax:
Identifier EXCEPTION;
How exceptions are raised?
Exceptions are raised in two ways as follow:

- An error arises that is internally associated with an exception. This exception is raised automatically as a result.

- An exception is raised explicitly by using the raise statement within a block.

Consider the following syntax:
RAISE exception–identifier;
Consider the following statement from the following exception being deliberately raised as follows:

```
DECLARE
E_invalid_product EXCEPTION;
BEGIN
UPDATE PRODUCT set descript= '&product_description'
WHERE prodid = &product_number;
IF SQL%NOTFOUND THEN
RAISE E_invalid_product;
END IF;
COMMIT;
EXCEPTION
WHEN E_invalid_product THEN
Dbms_output.put_line ('invalid product number');
End;
```

NON-PREDEFINED EXCEPTION

Non-predefine exceptions are those oracle errors for which no name is defined.
Consider the following syntax:
PRAGMA EXCEPTION INIT(exception_name, oracle_error_number);
Let see a programme that which will trap for oracle server number-2292 as follows:

```
DECLARE
e_emps_remaining EXCEPTION;
PRAGMA EXCEPTION_INIT (e_emps_reamining,-2292);
v_deptno dept.deptno%TYPE := &p_deptno;
BEGIN
DELETE FROM dept WHERE deptno = v_deptno;
COMMIT;
EXCEPTION
WHERE e_emps_remaining THEN
dbms_output.put_line ('cannot remove dept '|| TO_CHAR (v_deptno)||' .employee exits.');
END;
```

(Note: in the above example we have declared one exception name e_name_remaining and then associated that name with error number -2292 using EXCEPTION_INIT pragma).

PROPAGATION OF EXCEPTION

In PL/SQL block can be nested

```
DECLARE
     ............
BEGIN
        .........
DECLARE
.......
BEGIN
   ........
   ........
END;
..........
END;
```

Consider the point from the above statement as follows:

- Exception cannot be declared twice in the same block. However the same exception can be declared in two different blocks.
- Exception declared in a block are considered local to that block and global to all its sub-block because a block can reference only the local or global exception, enclosing block cannot reference exception declared in a sub-block.

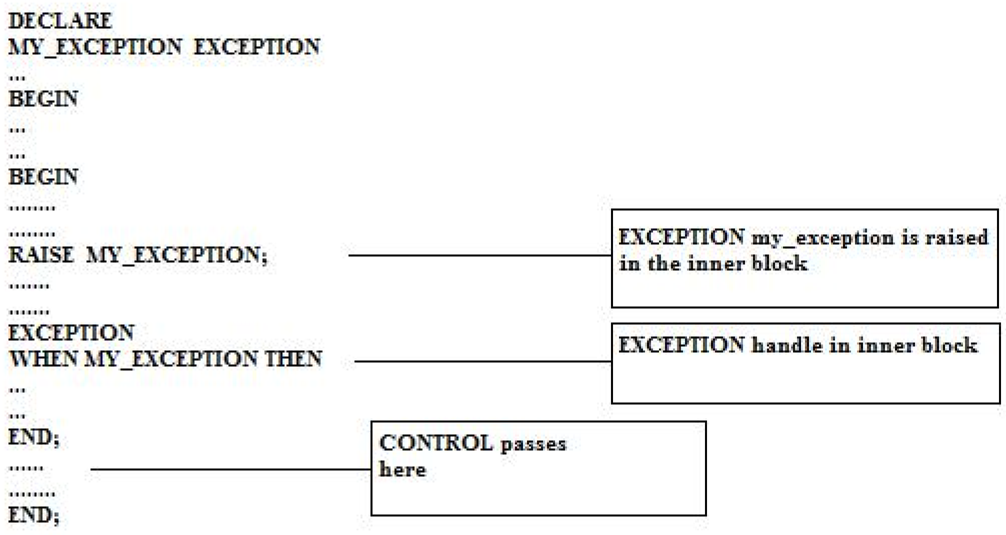
- If a global exception in a sub-block is re-declared, the local declaration prevails. So the sub-block cannot reference the global exception unless it was declared in a labelled block.

Consider the following syntax:

block_lablel.exception_name

When an exception is raised in the executable section of a block, the exception is handled in the following manner:

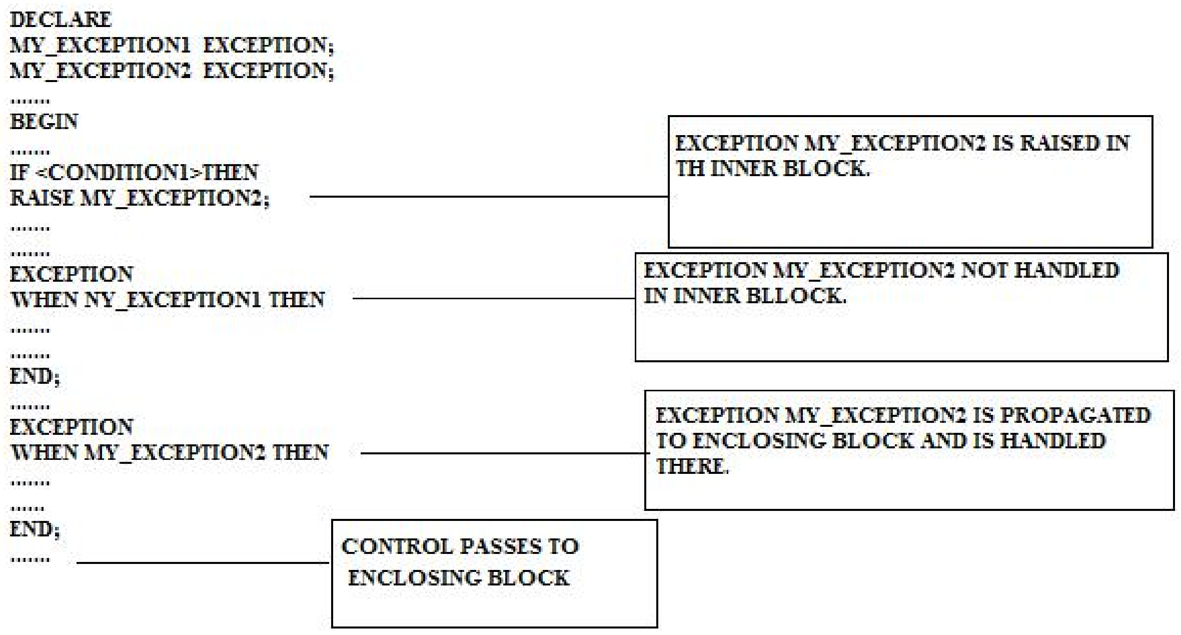
- The current block has the exception handler then it executes it and completes the block successfully. The control then passes to the enclosing block.



```
DECLARE
MY_EXCEPTION EXCEPTION
...
BEGIN
...
...
BEGIN
........
........
RAISE MY_EXCEPTION;
.......
.......
EXCEPTION
WHEN MY_EXCEPTION THEN
...
...
END;
......
........
END;
```

EXCEPTION my_exception is raised in the inner block
EXCEPTION handle in inner block
CONTROL passes here

- If there is no exception handler in the current block for a raised exception the execution propagates according to the following rules:
- If there is an enclosing block for the current block, the exception is passed on to that block. The enclosing block then becomes the current block. If a handler for the raised exception is not found the process is repeated.



```
DECLARE
MY_EXCEPTION1  EXCEPTION;
MY_EXCEPTION2  EXCEPTION;
.......
BEGIN
.......
IF <CONDITION1>THEN
RAISE MY_EXCEPTION2;
.......
.......
EXCEPTION
WHEN NY_EXCEPTION1 THEN
.......
.......
END;
.......
EXCEPTION
WHEN MY_EXCEPTION2 THEN
.......
......
END;
.......
```

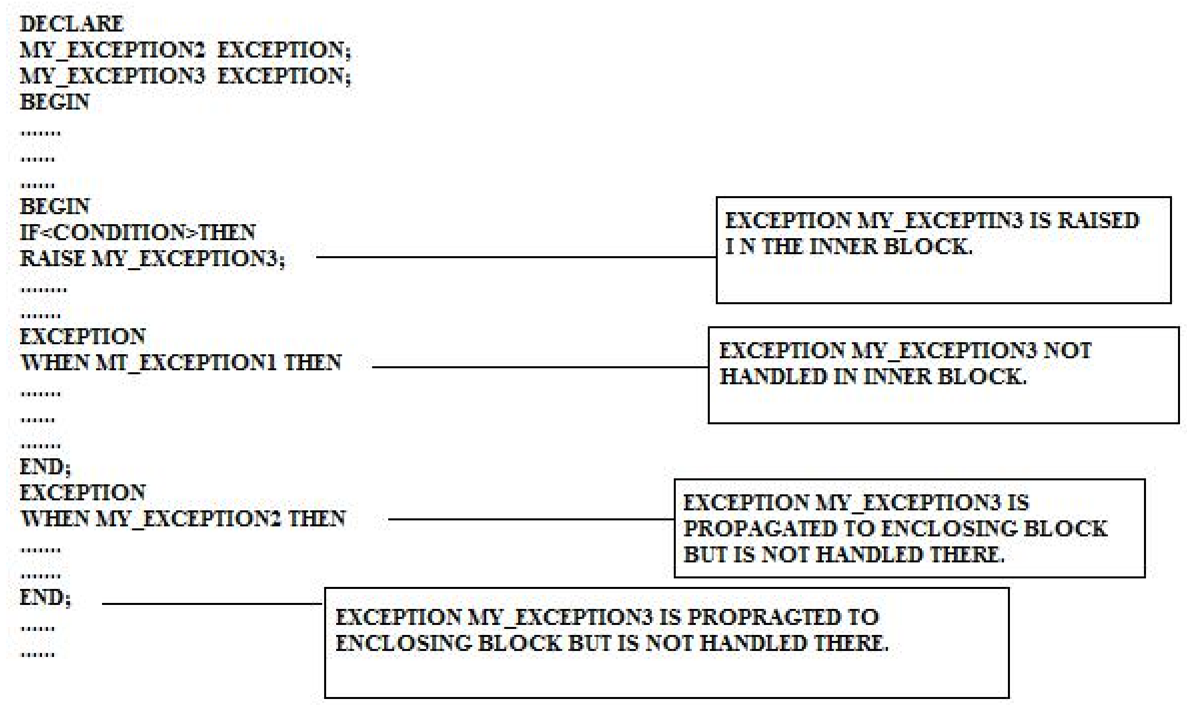
EXCEPTION MY_EXCEPTION2 IS RAISED IN TH INNER BLOCK.

EXCEPTION MY_EXCEPTION2 NOT HANDLED IN INNER BLLOCK.

EXCEPTION MY_EXCEPTION2 IS PROPAGATED TO ENCLOSING BLOCK AND IS HANDLED THERE.

CONTROL PASSES TO ENCLOSING BLOCK

- If there is no enclosing block for the current block, the exception is passed back to the environment that invoked the PL/SQL program.



```
DECLARE
MY_EXCEPTION2  EXCEPTION;
MY_EXCEPTION3  EXCEPTION;
BEGIN
.......
......
......
BEGIN
IF<CONDITION>THEN
RAISE MY_EXCEPTION3;
........
.......
EXCEPTION
WHEN MT_EXCEPTION1 THEN
.......
......
.......
END;
EXCEPTION
WHEN MY_EXCEPTION2 THEN
.......
.......
END;
......
......
```

EXCEPTION MY_EXCEPTIN3 IS RAISED I N THE INNER BLOCK.

EXCEPTION MY_EXCEPTION3 NOT HANDLED IN INNER BLOCK.

EXCEPTION MY_EXCEPTION3 IS PROPAGATED TO ENCLOSING BLOCK BUT IS NOT HANDLED THERE.

EXCEPTION MY_EXCEPTION3 IS PROPRAGTED TO ENCLOSING BLOCK BUT IS NOT HANDLED THERE.

- Only one exception at a time can be active in the exception handling part of a block.

Consider the following exception is raised in the scope rules for exception handling as follows:

```
DECLARE
MY_EXCEPTION EXCEPTION;
BEGIN
---------------------------------SUB BLOCK-----------------------------------------------------
DECLARE
MY_EXCEPTION EXCEPTION;
BEGIN
RAISE MY_EXCEPTION;

EXCEPTION
WHEN MY_EXCEPTION THEN
ROLLBACK;
END;
```

(Note: an exception declared in a sub block override an exception declared with the same name in an enclosing block and that enclosing block cannot reference exceptions declared in a sub-block).

Using RAISE_APPLICATION_ERROR

RAISE_APPLICATION_ERROR is used to create your own error messages, which can be more descriptive that named exception.

Consider the following syntax:

RAISE_APPLICTION_ERROR (error_number, error_message);

Let us see an example that will display an error_number which is a parameter between -20000 and 20999, error_message that is the text associated with the error:

```
BEGIN
UPDATE product SET descript= '&product_description'
WHERE prodid=&product_number;
IF SQL%NOTFOUND THEN
RAISE_APPLICATION_ERROR (-20001, 'INVALID PRODUCT');
END IF;
COMMIT;
END;
```

Chapter 13

After completing this chapter, you will be able to understand:

- ✓ **SUB PROGRAMME**
- ✓ **ADVANTAGES OF SUB PROGRAMME**
- ✓ **PROCEDURE**
- ✓ **FUNCTION**
- ✓ **ACTUAL VERSUS FORMAL PARAMETERS**
- ✓ **ARGUMENT MODES**
- ✓ **PACAKAGE**
- ✓ **COMPONENTS OF A PACKAGE**
- ✓ **CREATING THE PACKAGE**
- ✓ **REFERNCING PACKAGE CONTENTS**
- ✓ **ONE TIME PROCEDURE IN PACKAGE**
- ✓ **FUNCTION OVERLOADING**
- ✓ **DROPPING A PROCEDURE**
- ✓ **DROPPING A FUNCTION**
- ✓ **DROPPING A PACKAGE**
- ✓ **METHODS FOR PASSING PARAMETERS**
- ✓ **CURSOR EXPRESSION**
- ✓ **USING STORED FUNCTION IN SQL STATEMENT**
- ✓ **AUTONOMOUS TRANSACTION**

SUBPROGRAMME

Sub programme are named PL/SQL block that can accept parameter and be invoked whenever requires PL/SQL has two type of sub programme called procedures and functions.

The PL/SQL program can be stored in the database as stored programme and can be invoked whenever required. This avoids reparsing when multiple users invoke it. This also provides security and integrity control by allowing access on the sub program and not on the database objects directly.

Advantages of sub programme as follows:

- EXTENSIBILITY
- MODULARITY
- REUSABILITY AND MAINTAINABILITY
- ABSTRACTION

EXTENSIBILITY

Extensibility let you tailor the PL/SQL language to suit out need.

MODULARITY

Sub programme allow you to break a program down into manageable, well define logic modules. This supports top-down design and the stepwise refinement approach to problem solving.

RESUABILITY AND MAINTAINABILITY

Sub programs once validated can be used with confidence in any number of applications. Furthermore, only the sub program is affected if its definition changes. This simplifies maintenance and enhancement.

ABSTRACTION

Sub programme aid abstraction the mental separations from particular. To use sub programs it is important to know what they do, not how they know. Therefore application may be designed from top down without working about implementation details.

PROCEDURE

Procedure is a sub programme that performs a specific action. It can be called from any PL/SQL program. It can also be invoked from the SQL prompt.

A procedure has two parts: the specification and body.
A procedure specification begins with the keyword PROCEDURE followed by the procedure name and an optional list of argument, enclosed with parenthesis. Procedure that takes no parameters is written without parenthesis.
A procedure body begins with the keyword IS or AS and end with an END followed by optional procedure name. The procedure body has three parts:
(1) Declarative part
(2) Executable part and
(3)Exception handling part.
The declarative part contains local declaration. All the declarations are placed between the IS and BEGIN keyword.
The executable part contains statements, which are placed between the keywords BEGIN and EXCEPTION (or END)
The Exception handling part contains exception handlers, which are placed between the keywords EXCEPTION and END. This part is optional
Consider the following syntax for writing a procedure:

```
CREATE [OR REPLACE] PRODCEDURE <procedure_name>
(<arg1>[mode] <datatype>,.......)
IS| AS
[local declaration]
BEGIN
PL/SQL executable statement
[EXCEPTION exception handlers]
END [PROCEDURE_NAME];
```

(Note: where mode indicates the type of the argument(such as IN, OUT or INOUT) .
Consider the following example which creates a procedure by the name INCR and increases the salary of an employee as follows:

```
CREATE OR REPLACE PROCEDURE incr (e_id number, amt number) IS
        vsalary       NUMBER;
        salary_missing   EXCEPTION;
BEGIN
        SELECT sal INTO vsalary FROM EMP
                WHERE empno=e_id;
        IF vsalary IS NULL THEN
                RAISE salary_missing;
        ELSE
                UPDATE emp set sal= vsalary + amt
                Where empno= e_id;
        ENDIF;
EXCEPTION
        WHEN salary_missing THEN
        Dbms_output.put_line (e_id || ‘has salary as NULL’);
        WHEN NO_DATA_FOUND THEN
        Dbms_output.put_line (e_id || ‘No such number’);
END incr;
```

CALLING A PROCEDURE

A procedure is called as a PL/SQL statement. A procedure can be called from any PL/SQL program by giving their names followed by the parameter.
Consider the following syntax:
incr (empno, amount);
procedures can also be invoked from SQL prompt using EXECUTE command:
SQL> EXECUTE incr (empno, amount);

FUNCTIONS

A Function is a sub program that returns a value. A function must have a RETURN clause. Functions and procedures have a similar structure, except that the function have a RETURN clause and the function only take IN parameter, not OUT or INOUT.
Consider the following syntax:

```
CREATE [OR REPLACE] FUNCTION <FUNCTION_NAME>
[<arg1>[mode]<data type>.........]
RETURN datatype IS
[local Declaration]
BEGIN
PL/SQL executable statement
[EXCEPTION exception handler]
END [FUNCTION_NAME];
```

Like a procedure, a function has two parts: the specification and the body. The function specification begins with the keyword FUNCTION and ends with RETURNS clause, which specifies the data type of the result value.
The function body is exactly same as procedure body**.**

RETURN STATEMENT

RETURN statement immediately return control to the caller environment. Execution then resumes with the statement following the sub programme call.
A sub programme can contain several return statements. Executing any of them terminates the sub programme immediately. The RETURN statement used in function must contain an expression, which is evaluated when the return statement is executed. The resulting value is assigned to the function identifier.
Below is a program that creates function to calculate the net salary based on various conditions. The employee number is passed as parameter. The function returns the value of the increment.

```
CREATE OR REPLACE FUNCTION review (empid NUMBER)
        RETURN NUMBER IS
                incr emp.sal%TYPE;
                Net emp.sal%TYPE;
                vempno emp.Empno%TYPE;
                vsal emp.sal%TYPE;
                vcomm emp.comm%TYPE;
BEGIN
                Select empno, sal, nvl (comm.,0) into vempno, vsal, vcomm,, from emp where
                empno= empid;
                net := vsal+vcomm;
                If vsal <=3000 then
                        incr:= 0.20 * net;
                elsif  vsal > 3000 and vsal <=6000 then
                        incr := 0.30*net;
                Else
                        incr:=0.40 * net;
                End if;
        return (incr);
end review;
```

Function review can be called from any PL/SQL block

```
Declare
incr_sal number(7,2);
Begin
incr_sal:=review(7698);
Dbms_output.put_line(incr_sal);
End;
```

Rules for calling function from SQL Expressions as follows:

- A user define function must be a stored function.
- A user define function must be a single row function and not a group function.
- Data type must be valid SQL data type, char, date, number. It cannot be PL/SQL types.
- INSERT, UPDATE or DELETE commands are not allowed.
- Calls to sub programme that break the above restriction are not allowed.

Actual versus formal parameter

- The variables or expressions referenced in the parameter list of sub programme body are actual parameters.

  The following procedure call has two actual parameters as follows:
  Incr (empid, amt);
- The variable in a sub programme specification and referenced in the sub programme body are formal parameter.

  The following procedure declaration which has two formal parameters as follows:
  CREATE PROCEDURE increment (empno,incr)IS
  ....

```
............
BEGIN
.............
.............
UPDATE emp SET sal= sal+incr
WHERE
END increment;
```

Argument Mode

Generally the argument modes are used to define the behaviour of formal parameter.
Argument group can be divided into three modes as follows:
1) IN: The IN parameter lets the user pass value to the called sub programme. Inside the sub programme, the IN parameter acts like a constant therefore it cannot be modified.
2) OUT: The OUT mode parameter lets the user return values to the calling block. Inside the subprogram, the OUT parameter acts like an uninitialized variable. We cannot pass the value to the procedure through OUT parameter.
3)IN OUT: The IN OUT parameter lets the user pass initial values to the called subprogram and returns updated values to the calling block.

- **IN MODE**: Default mode of an argument is IN. The actual parameter that corresponds to an IN formal parameter can be constant, literal, initialized variable or expression.

See the example given below:

```
CREATE PROCEDURE ISSUE_QTY(ISS_ITEM ITEMMAST.ITNO%TYPE, ISS_OTY
NUMBER)
IS
BEGIN
UPDATE ITEMMAST SET QOH=QOH-ISS_QTY
WHERE ITNO=ISS_ITEM;
COMMIT;
END;
```

- **OUT MODE:** If an argument is being used for returning a value the out mode is specified. The actual parameter that corresponds to an out formal parameter must be a variable; it cannot be a constant or expression.

See the following example given below:

```
CREATE PROCEDURE GET_QOH (VAR_INTO ITEMMAST.ITNO%TYPE, QOH_BAL
OUT ITEMMAST.QOH%TYPE)
IS
BEGIN
SELECT QOH INTO QOH_BAL FROM ITEMMAST WHERE ITNO=VAR_ITNO;
END;
```

The following assignment causes a compilation error

```
CREATE PROCEDURE GET_QOH (VAR_INTO ITEMMAST.ITNO%TYPE, QOH_BAL
OUT ITEMMAST.QOH%TYPE)
IS
BEGIN
SELECT QOH INTO QOH_BAL FROM ITEMMAST WHERE ITNO=VAR_ITNO;
QOH_BAL:= QOH_BAL+10;  --error
END;
```

- **INOUT MODE:** If the argument is used to get as well as return value to the calling environment the IN OUT mode can be used inside the sub programme IN OUT parameter acts like an initialized variable INOUT parameter can be assigned a value and its value can be assigned to another variable.

Consider the following example as follows:

```
CREATE PROCEDURE NEW_QOH(VAR_ ITNO ITEMMAST.ITNO%TYPE, BAL_QOH
IN OUT ITEMMAST.QOH%TYPE)
IS
BEGIN
SELECT SUM (DECODE (TRANTYPE, 'R' ,QTY, -QTY))+ BAL_QOH
INTO BAL_QOH FROM ITTRAN
WHERE ITNO= VAR_ITNO;
END;
```

PACKAGE

A package is a database object that groups logically related PL/SQL objects. Package encapsulates related procedures, functions, associated cursors and variables together as a logical unit in the database. Packages are made of two components. The specification and the body as follows:

- The specification is the interface and has declaration and the body.
- The body of a package contain different procedure and function.

Components of a package

Package specification:

It declares the types, variables, constants, exceptions, cursors and sub programs. The specification holds declaration, which is visible to the application. The scope of this declaration is local to your database schema and global to the package**.**

The package Body

The body fully defines cursors and sub programs, and so implements the specification. The body holds implementation details and private declarations, which are hidden from the application.

(Note: unlike sub programme packages cannot be called or pass parameter or nested. Still the format of package is similar to that of a subprogram).
In the package we have two types of member functions, procedures and variables**.**

1) Public: public member of packages can be referenced from any oracle server environment. These members are declared within the package specification and may be define within the package body.

2) Private: private member can only be referenced by other construct, which are part of the same package.

Advantages of packages
Packages offer several advantages, which includes:

- MODULARITY
- EASIER APPLICATION DESIGN
- INFORMATION HIDING
- ADDED FUNCTIONALITY
- BETTER PERFORMANCE

Creating packages

A packages is created interactively with SQL*plus using the CREATE PACKAGE and CREATE PACKAGE BODY commands.
Consider the following syntax:

```
CREATE [OR REPLACE] PACKAGE <PKG-NAME> AS
/*Declaration of global variables and cursors(if any);
Procedure and functions;
*/
END [<PKG-NAME>];
CREATE [OR REPLACE] PACKAGE BODY <PKG-NAME> AS
/* private type and object declaration, subprogram bodies; */
[BEGIN
----ACTION STATEMENT;]
END [PKG-NAME];
```

Consider the following programme that will create with a procedure and a function. You can have any number of functions and procedures in a package as follows:

```
CREATE OR REPLACE PACKAGE EMPLPACK AS
```

```
PROCEDURE EMPPROC (EMPCODE IN EMP.EMPNO%TYPE);
FUNCTION INCREMENT (e_id NUMBER, amt NUMBER) RETURN NUMBER;
END EMPLPACK;
CREATE OR REPLACE PACKAGE BODY EMPLPACK
AS
PROCEDURE EMPPROC (EMPCODE IN EMP.EMPNO%TYPE)
IS
TEMPNAME EMP.ENAME%TYPE;
TESAL EMP.SAL%TYPE;
BEGIN
SELECT ENAME, SAL INTO TEMPNAME, TESAL FROM EMP WHERE
EMPNO=EMPCODE;
Dbms_output.put_line (EMPCODE||TEMPNAME||TESAL);
EXCEPTION
WHEN NO_DATA_FOUND THEN
Dbms_output.put_line (EMPCODE|| 'NOT FOUND');
END EMPPROC;
FUNCTION INCREMENT (e_id NUMBER, amt NUMBER)
RETURN NUMBER IS
vsalary     NUMBER;
salary_missing  EXCEPTION;
Temp NUMBER;
BEGIN
SELECT SAL INTO vsalary FROM emp
WHERE empno=e_id;
IF vsalary IS NULL THEN
RAISE salary_missing;
ELSE
UPDATE emp set sal =sal + amt
Where empno = e_id;
Temp:=vsalary +amt;
RETURN (temp);
END IF;
EXCEPTION
WHEN salary_missing THEN
Dbms_output.put_line (e_id|| 'has salary as NULL');
/* if the employee number is not found then exception is raised*/
WHEN NO_DATA_FOUND THEN
dbms_ouppput.put_line (e_id|| 'No such number');
END increment;
END EMPLPACK;
```

Function Overloading

This feature allows you to define different sub programs with the same name. The sub programs are distinguished on the basis of both name and the parameter of the sub programme.

Consider the following example:

```
CREATE OR REPLACE PACKAGE over_pack
IS
PROCEDURE add_dept
(V_deptno IN dept.deptno%TYPE,
V_name IN dept.dname%TYPE DEFAULT ‘UNKNOWN’,
V_loc IN dept.loc%TYPE DEFAULT ‘UNKNOWN’);

PRCEDURE add_dept
(V_name IN dept.dname%TYPE DEFAULT ‘UNKNOWN’,
V_LOC  IN dept.loc%TYPE DEFAULT ‘unknown’);
CREATE OR REPLACE PACKAGE BODY over_pack
IS
PROCEDURE add_dept
(V_deptno IN dept.deptno%TYPE,
V_name IN dept.dname%TYPE DEFAULT ‘UNKNOWN’,
V_loc IN Dept.loc%TYPE DEFAULT ‘UNKNOWN’)
IS
BEGIN
INSERT INTO dept VALUES(V_deptno, V_name, V_loc);
END add_dept;

PROCEDURE add_dept
(V_name IN dept.dname%TYPE DEFAULT ‘UNKNOWN’,
V_loc IN Dept.loc%TYPE DEFAULT ‘UNKNOWN’);
IS
BEGIN
INSERT INTO dept VALUES(dept_deptno.NEXTVAL, V_name, V_loc);
END add_dept;
END over_pack;
```

Dropping a Procedure

To drop a procedure, drop procedure command is used. In order to drop a procedure one must either own the procedure or have DROP ANY PROCEDURE system privilege.

Consider the following example:

```
DROP procedure proc1;
```

Dropping a Function

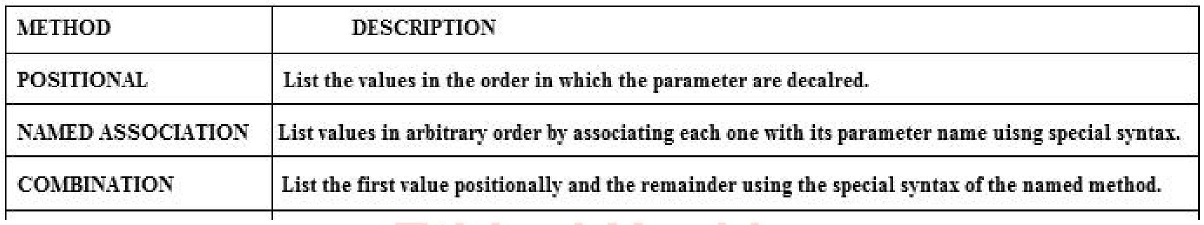
To drop a function, drop function command is used. In order to drop a function one must either own the function or have DROP ANY PROCEDURE system privileges.
Consider the following syntax:
Drop function function1;

Dropping a package

To drop a package, drop package command is used. To drop a package, one must either own the package or have DROP ANY PROCEDURE system privileges.
Consider the example:
Drop package emp1pack;

Methods for passing Parameter

A procedure containing multiple parameters, you can use a number of methods to specify the values of the parameter.



METHOD	DESCRIPTION
POSITIONAL	List the values in the order in which the parameter are decalred.
NAMED ASSOCIATION	List values in arbitrary order by associating each one with its parameter name uisng special syntax.
COMBINATION	List the first value positionally and the remainder using the special syntax of the named method.

See the following example as follows:

```
CREATE OR REPLACE PROCEDUR add_dept
(V_name IN dept.dname%TYPE DEFAULT 'unknown',
V_loc IN deptp.loc%TYPE DEFAULT 'unknown')
IS
BEGIN
INSERT INTO dept
VALUES (dept_deptno.NEXTVAL, V_name, V_loc);
END ADD_dept;
```

Now this procedure can be called by passing its parameters in different ways

```
BEGIN
add_dept;
add_dept('TRAINING','NEW YORK');
add_dept(V_loc=> 'DALLAS',V_name=> 'EDUCATION');
add_dept(V_loc=> 'BOSTON');
END;
```

Chapter 14

After completing this chapter, you will be able to understand:

- ✓ **DATABASE TRIGGERS**
- ✓ **PARTS OF A TRIGGER**
- ✓ **STATEMENT TRIGGER AND ROW TRIGGERS**
- ✓ **INSTEAD OF TRIGGERS**
- ✓ **DROPPING TRIGGERS**

DATABASE TRIGGER

A database trigger is a stored PL/SQL program unit associated with specific database table. Unlike the stored procedures, which have to be explicitly invoked, these triggers implicitly fire whenever a particular event takes place in the associated table. The event may be an INSERT, UPDATE, or DELETE on the table. Oracle also allows you to create trigger for views.
A trigger can include SQL and PL/SQL statement to execute as a unit and can invoke other stored procedures. However procedures and triggers differ in the way in which they are invoked. While a procedure is explicitly executed by a user or an application, a trigger is implicitly fired (executed) by oracle.

In oracle there is no limitation to the number of trigger that can be associated with given table.
To create a trigger the user must have CREATE TRIGGER privilege or own the table, have ALTER TABLE privileges for that table or have ALTER ANY TABLE privileges.
Following are the uses of the database trigger:

- Audit data information
- Log events transparently
- Enforce complex business rules
- Derive column values automatically
- Implement complex security authorizations
- Maintain replicate tables.

PARTS OF A TRIGGER

A database trigger has three parts as follows:

- Trigger timing
- Trigger event
- Trigger constraint
- Trigger action

  When the event occurs the database trigger fires and the PL/SQL block performs the action. Database triggers fire with the privileges of the owner, not the current user. Therefore, the owner must have appropriate access to all objects referenced by the trigger action.

- **TRIGGER TIMING**

Trigger timing determines when the trigger fires in relation to the triggering event. The triggering timing can be BEFORE, AFTER or INSTEAD OF.

- **TRIGGERING EVENT**

A triggering event or a statement is the SQL statement that causes a trigger to be fired. A triggering event can be an INSERT, UPDATE or DELETE statement for a specific table (or view)

- **TRIGGERING RESTRICTION**

A trigger restriction specifics a Boolean expression that must be true for the trigger action to be executed to fire. A trigger restriction is an option available for trigger that fire for each row. Its function is to conditionally control the execution of a trigger. A trigger restriction is specified using a WHEN clause.

- **TRIGGER ACTION**

A trigger action is the procedure (PL/SQL block) that contains the SQL statement and PL/SQL code to be executed when a triggering statement is issued and the trigger restriction evaluates to TRUE.

STATEMENT TRIGGERs AND ROW TIGGERs

Statement level triggers fire once for the triggering event. On the other hand a row trigger fires once for each row affected by the triggering event. For example if a statement updates 100 rows in the EMP table, then a statement trigger on that table would fire only once whereas the row trigger fire 100 times. A statement trigger always fires even if no rows are affected at all by the triggering event. However if the trigger event affect no row a row trigger is not executed at all.
Statement triggers are useful if the trigger action does not depend on data of rows that are affected or data provided by the triggering event itself.

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

CREATE [OR REPLACE] TRIGGER <trigger-name>

BEFORE |AFTER|INSTEAD OF → **Trigger Timing**

DELETE |[OR] INSERT |[OR] UPDATE [OF <COLUMN> [column>......]]

ON <table> **Triggering event**

[REFERNCEING [OLD [AS] <OLD>] [NEW [AS] <new>]]

[FOR EACH ROW [WHEN <condition>]]

<u>Trigger Restriction</u>

BEGIN
....
***/ PL/SQL BLOCK */**

....
END;

<u>Triggering Action</u>

REFERENCING: Specifies correlation names. The user could use the correlation names in the PL/SQL block and when clause of a row triggers to refer specifically to old and new values of the current row. The default correlation names are OLD and NEW.
WHEN: Specifies the trigger restriction. This condition has to be satisfied to fire the trigger. This condition can be specified for the row trigger.

- **<u>TYPES OF TRIGGER</u>**

There are 14 types of trigger .A trigger type is defined by the type of triggering event and by the level at which the trigger is executed. Here is the list of triggering actions:

Name	**Statement level**	**Row level**
BEFORE option	**Oracle fires the trigger only once, before executing the triggering statement**	**Oracle fires the trigger before modifying each row affected by the triggering statement**
AFTER option	**Oracle fires the trigger only once, after executing the triggering statement**	**Oracle fires the trigger after modifying each row affected by the triggering statement**
INSTEAD OF option	**Oracle fires the trigger only once to do something else instead of performing the action that executed the trigger**	**Oracle fires the trigger for each row, to do something else instead of performing the action that executed the trigger.**

When combining the different type of triggering actions, there are 14 possible valid trigger type available to us as mentioned below as follows:

- BEFORE INSERT statement
- BEFORE INSERT row
- AFTER INSERT statement
- AFTER INSERT row
- BEFORE UPDATE statement
- BEFORE UPDATE row
- AFTER UPDATE statement
- AFTER UPDATE row
- BEFORE DELETE statement
- BEFORE DELETE row
- AFTER DELETE statement
- AFTER DELETE row
- INSTEAD OF statement
- INSTEAD OF row

FIRING SEQUENCE OF TRIGGERS

The following list show the firing sequence of different trigger associated with table when a particular triggering event occurs.

BEFORE statement trigger	→	(Fire once)
BEFORE ROW TIRGGER	→	(FIRE ONCE FOR EACH ROW)
AFTER ROW TRIGGER	→	(FIRE ONCE FOR EACH ROW)
AFTER statement trigger.	→	(Fire once)

CREATING STATEMENT TRIGGERS

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

Let us take an example that will create trigger that restricts modification in the EMP table to certain business hours as follows:

```
CREATE OR REPLACE TRIGGER EMP1_UPDATE_CHK
        BEFORE UPDATE ON EMP1
BEGIN
        IF (TO_CHAR (sysdate,'HH24') BETWEEN '14' AND '16') THEN
                RAISE_APPLICATION_ERROR (-20100,'You cannot modify EMP1 table
                during 14:00 to 16:00 Hrs');
        END IF;
END;
Trigger created.
```

When ever you update the emp1 table during the period 14:00 to 16:00, the condition for the IF END IF block becomes true and the built in procedure RAISE_APPLICTION_ERROR is executed and consequently the specified message is displayed and the trigger fails.
Now if, you issue the following statement at 15.00 hrs

```
Update emp1
Set sal = sal+ 500
Where job=’ANALYST’;
```

You will get the following error.

```
Update emp1
*
Error al line 1:
ORA -20100: 'You cannot modify EMP1 table during 14:00 to 16:00 Hrs
```

- **CONDITIONAL PREDICATES**

The combination of several triggering that extend the event into one by using the conditional predicates, INSERTING, UPDATING, DELETING within the trigger body.
Let us take an example that create another trigger that extend the functionally of the above trigger. The trigger should restrict any DML done on emp1 table during 14:00 to 16:00 hrs. Now if the user attempts to perform any DML on the emp1 table during this period, appropriate message should appear.
Look at the following script:

```
CREATE OR REPLACE TRIGGER restrict_emp1_dml
BEFORE INSERT OR UPDATE OR DELETE ON EMP1
BEGIN
IF (TO_CHAR (sysdate,'HH24') BETWEEN '14' AND '16') THEN
IF INSERTING THEN
RAISE_APPLICATION_ERROR (-20100,
'YOU CANNOT INSERT INTO EMP1 TABLE DURING 14:00 TO 16:00 HRS');
END IF;
IF UPDATING THEN
RAISE_APPLICATION_ERROR (-20100,
'YOU CANNOT UPDATE EMP1 TABLE DURING 14:00 TO 16:00 HRS');
END IF;
```

```
IF DELETING THEN
RAISE_APPLICATION_ERROR (-20100,
'YOU CANNOT DELETE FROM EMP1 TABLE DURING 14:00 TO 16:00 HRS');
END IF;
END IF;
END;
Trigger created.
```

The above trigger fires whenever any user perform an insert, update or delete operation on the emp table and the triggering event is checked by the corresponding conditional procedure.(note: you can use the conditional predicates in row triggers as well).

CREATING ROW TRIGGER

Let us create a trigger for the emp1 table that makes the entry in ename column in uppercase irrespective of the case in which the user enters the value:

```
CREATE OR REPLACE TRIGGER upper_trig
BEFORE INSERT OR UPDATE OF ename ON EMP1
FOR EACH ROW
BEGIN
: new.ename:= UPPER (:old.ename);
END;
Trigger created
```

Here the above example used the keyword new, which refers to the new value of the column and the keyword old, which to the old value of the column. When referencing the new and old keyword in the PL/SQL block, prefix them by colon (:).

Note: use of new/old keywords are allowed row level triggers only.

- **RESTRICTING A ROW TRIGGER**

Let us take an example that create a row trigger for emp1 table fires only, when you insert a new row with job as SALESMAN or when you update the salary of an existing SALESMAN. The trigger should ensure that is if a new row is inserted, commission amount should be zero, and if the row is updated then commission should be equal to the old commission plus 10% of the increment as follows:

```
CREATE OR REPLACE TRIGGER EMP1_RESTRICT_FOR _SALES
BEFORE INSERT OR UPDATE OF SAL ON EMP1
FOE EACH ROW
WHEN (NEW.JOB='SALESMAN')
BEGIN
IF INSERTING THEN
:NEW.comm :=0;
ELSE
:NEW.comm := old.comm+(:new.sal - :old.sal)* 0.1;
```

```
END IF;
END;
```

Here you see when clause is used to specify the restriction for the rows. Remember the new or old qualifier do not need to be prefixed with a colon in the when clause
(Note: Except INSTEAD OF trigger, all trigger can be created for table. INSTEAD OF triggers can be used only for view).

- **INSTEAD OF TRIGGERS**

Instead of trigger tells Oracle what do instead of performing the action that caused the trigger to fire. You can write INSERT, UPDATE OR DELETE statement against a view, and the INSTEAD OF trigger works invisible in the background to make the right action take place.
Consider the following syntax:

```
CREATE [OR REPLACE] TRIGGER   trigger_name
INSTEAD OF event1 [OR event2 OR event3]
ON view_name
[REFERNCING OLD as OLD/NEW AS new]
[FOR EACH ROW]
BEGIN
..........
/*PL/SQL block*/
..........
END;
```

Let us take an example that view emp_dept is created by using the following statement as follows:

```
CREATE VIEW emp1_dept AS
SELECT empno, ename, emp.deptno, job, sal, dname, loc
FROM emp, dept
WHERE emp.deptno= dept.deptno;
```

Let us take another example that will delete all rows from the emp1 table for particular department number and at the same time delete the corresponding department information from the DEPT table simultaneously when a delete statement is executed for the view specifying a department number. This is possible only through INSTEAD OF triggers.

```
CREATE TRIGGER EMP1_DEPT_DELETE
INSTEAD OF DELETE ON EMP1_DEPT
FOR EACH ROW
BEGIN
DELETE FROM EMP1
WHERE DEPTNO=: old.deptno;
DELETE FROM DEPT
WHERE deptno =:old.deptno;
END;
```

Now if a delete statement is executed on view emp1_dept, for a particular department number, the corresponding employee's records will be deleted from the emp1 table and at the same time the corresponding department record will also be deleted.

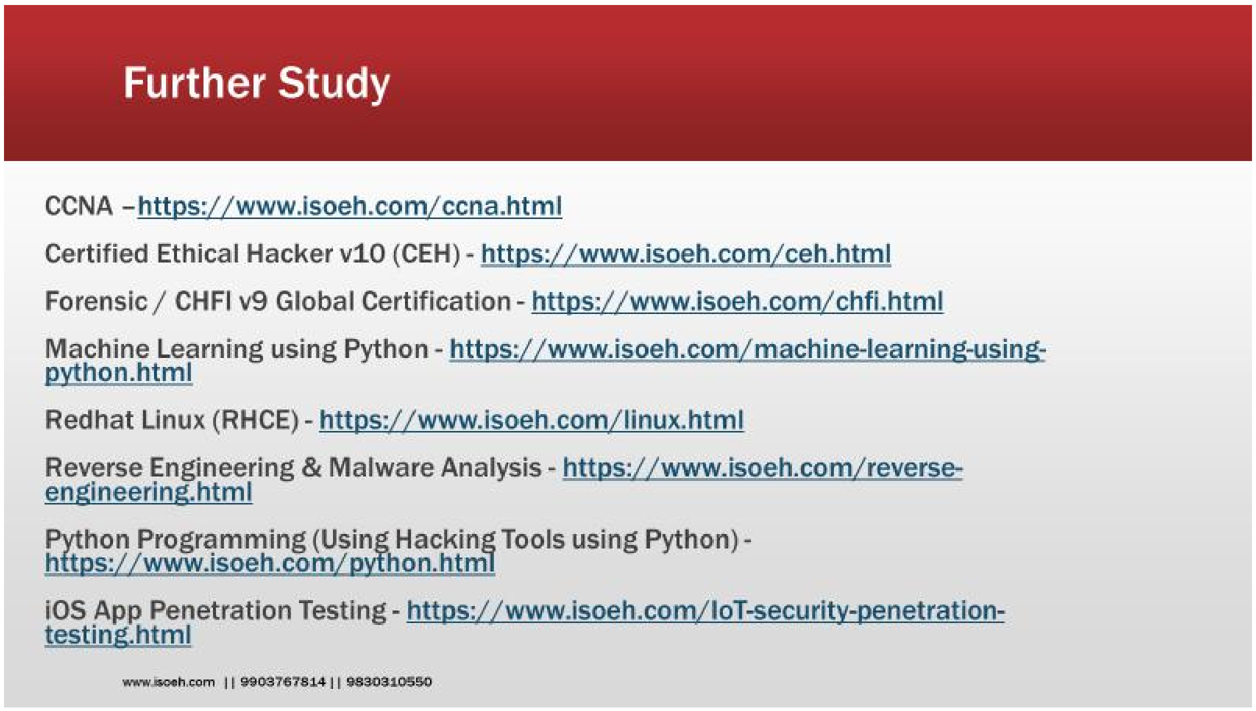
- **<u>DROPPING A TRIGGER</u>**

Triggers may be dropped using drop trigger command. In order to drop a trigger, you must either own the trigger or have the DROP ANY TRIGGER system privilege.
Syntax is:-
DROP TRIGGER trigger_name;

Evaluations will be based on:
1. Attendance in class (10 points)
2. Assignments (20 points)
3. Viva #1 to be conducted mid-course (10 points)
4. Viva #2 to be conducted end-course (20 points)
5. Practical Exam or Project (40 points)

Our Free Resources:

1. YouTube Channel (Tutorials) - https://www.youtube.com/channel/UCJB73X1BSGTEei9UAH0psgQ
2. Blogs - https://www.isoeh.com/exclusive-blog.html
3. Tools - https://www.isoeh.com/tools.html



Further Study

CCNA –https://www.isoeh.com/ccna.html

Certified Ethical Hacker v10 (CEH) - https://www.isoeh.com/ceh.html

Forensic / CHFI v9 Global Certification - https://www.isoeh.com/chfi.html

Machine Learning using Python - https://www.isoeh.com/machine-learning-using-python.html

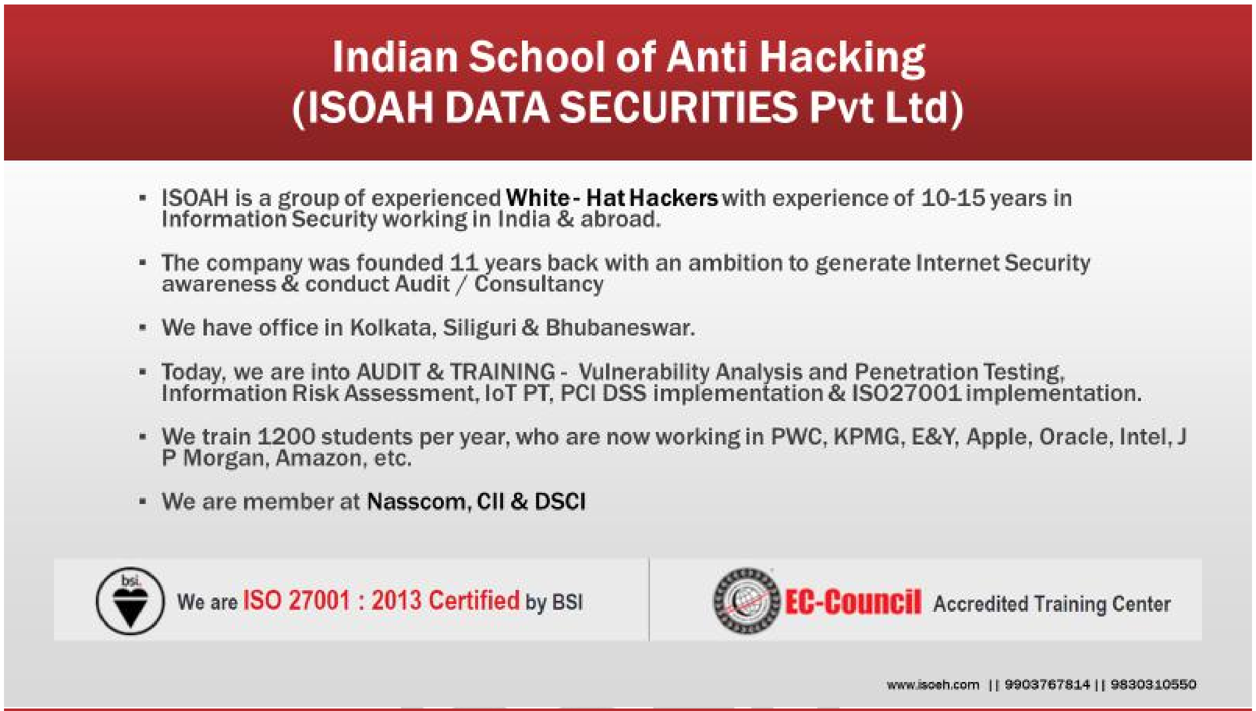
Redhat Linux (RHCE) - https://www.isoeh.com/linux.html

Reverse Engineering & Malware Analysis - https://www.isoeh.com/reverse-engineering.html

Python Programming (Using Hacking Tools using Python) - https://www.isoeh.com/python.html

iOS App Penetration Testing - https://www.isoeh.com/IoT-security-penetration-testing.html

www.isoeh.com || 9903767814 || 9830310550



Indian School of Anti Hacking (ISOAH DATA SECURITIES Pvt Ltd)

- ISOAH is a group of experienced **White- Hat Hackers** with experience of 10-15 years in Information Security working in India & abroad.
- The company was founded 11 years back with an ambition to generate Internet Security awareness & conduct Audit / Consultancy
- We have office in Kolkata, Siliguri & Bhubaneswar.
- Today, we are into AUDIT & TRAINING - Vulnerability Analysis and Penetration Testing, Information Risk Assessment, IoT PT, PCI DSS implementation & ISO27001 implementation.
- We train 1200 students per year, who are now working in PWC, KPMG, E&Y, Apple, Oracle, Intel, J P Morgan, Amazon, etc.
- We are member at **Nasscom, CII & DSCI**

We are ISO 27001 : 2013 Certified by BSI

EC-Council Accredited Training Center

www.isoeh.com || 9903767814 || 9830310550

ABOUT ISOAH

Our TRAINER / AUDITOR' PROFILES

Our auditors are OSCP, OSCE, ECSA, CISA certified;

and not just CEH, which is entry level certificate.

We have total **7** CEH, **1** OSCE, **2** OSCP, **1** ECSA, **2** ISO 27001 Lead Auditor, **2** CISA, **2** CCNA, **1** CCNP, **1** RHCE, **3** Forensic Expert (CHFI), **1** Data Protection & Privacy / GDPR Auditor, **1** ISO 22301 Business Continuity & organization resilience Auditor

Our **Auditors** are our **Trainers**

www.isoeh.com || 9903767814 || 9830310550