# INDEX

| LAB | CONTENT | PAGE NO | DATE | REMARKS |
|---|---|---|---|---|
| 1 | (a) Introduction to SQL:<br>DQL : Select<br>DDL : Create, Rename, Alter, Describe<br>DML : Insert, Update, Delete | | | |
| 2 | (a) DCL : Commit, Rollback, Savepoint<br>(b) Drop and rename a column<br>(c) Drop and Truncate | | | |
| 3 | (a) Data Constraints :<br>(i) I/O Constraints<br>(ii)Business Constraints<br>(b) Eliminating Duplicacy | | | |
| 4 | (a) Introduction to Dual table<br>(b) Arithmetic Operators<br>(c) Alicing of column<br>(d) Insertion from another table | | | |
| 5 | (a) Built-In Functions :<br>(i) Single row : Character function, Number function<br>(ii)Multiple row : Group function<br>(b) Comparison condition<br>(c) Logical condition | | | |

| | | | | |
|---|---|---|---|---|
| 6 | (a) Order by & Group by clause<br>(b) Subquery :<br>(i) Single row<br>(ii) Multi row<br>(iii)Correlated | | | |
| 7 | (a) Joining of Tables :<br>(i) Cross Join<br>(ii) Natural Join<br>(iii)Outer Join | | | |
| 8 | (a) Creation and Deletion of Views :<br>(i) Single Table<br>(ii) Joined Tables<br>(b) Sequence | | | |
| 9 | (a) Creation and Deletion of Index :<br>(i) Single Column<br>(ii) Multiple Columns<br>(b) Row id and Row num | | | |
| 10 | Introduction to PL/SQL | | | |
| 11 | Control Structures in PL/SQL | | | |
| 12 | Procedures and Functions in PL/SQL | | | |
| 13 | Cursors in PL/SQL | | | |
| 14 | Package and Exception handling in PL/SQL | | | |
| 15 | Triggers in PL/SQL | | | |

# DATABASE MANAGEMENT SYSTEM

## DATA AND DATABASE :

A **Database** is a collection of related data organised in a way that data can be easily accessed, managed and updated. Any piece of information can be a data, for example name of your school. Database is actualy a place where related piece of information is stored and various operations can be performed on it.

## DBMS :

Database management systems are computer software applications that interact with the user, other applications, and the database itself to capture and analyze data. A general-purpose DBMS is designed to allow the definition, creation, querying, update, and administration of databases. Well-known DBMSs include MySQL, PostgreSQL, Microsoft SQL Server, Oracle, Sybase and IBM DB2.

## INTRODUCTION TO SQL :

**Structure Query Language(SQL)** is a programming language used for storing and managing data in RDBMS. SQL was the first commercial language introduced for E.F Codd's **Relational** model. Today almost all RDBMS(MySql, Oracle, Infomix, Sybase, MS Access) uses **SQL** as the standard database language. SQL is used to perform all type of data operations in RDBMS.

## SQL COMMANDS :

1. **DDL : Data Definition Language**

   All DDL commands are auto-committed. That means it saves all the changes permanently in the database.

| COMMAND | DESCRIPTION |
|---|---|
| CREATE | TO CREATE NEW TABLES OR DATABASE |
| ALTER | FOR ALTERATION |
| TRUNCATE | DELETE DATA FROM TABLE |
| DROP | TO DROP A TABLE |
| RENAME | TO RENAME A TABLE |

## 2. DML : Data Manipulation Language

DML commands are not auto-committed. It means changes are not permanent to database, they can be rolled back.

| COMMAND | DESCRIPTION |
|---|---|
| INSERT | TO INSERT A NEW ROW |
| UPDATE | TO UPDATE EXISTING ROW |
| DELETE | TO DELETE A ROW |
| MERGE | MERGING TWO ROWS OR TWO TABLES |

## 3. DCL : Data Control Language

These commands are to keep a check on other commands and their affect on the database. These commands can annul changes made by other commands by rolling back to original state. It can also make changes permanent.

| COMMAND | DESCRIPTION |
|---|---|
| COMMIT | TO PERMANENTLY SAVE |
| ROLLBACK | TO UNDO CHANGE |
| ROLLBACK | TO SAVE TEMPORARILY |

## 4. DQL : Data Query Language

| COMMAND | DESCRIPTION |
|---|---|
| SELECT | RETRIVE RECORDS FROM ONE OR MORE TABLE |

# LAB 1

## The SQL SELECT Statement

The SELECT statement is used to select data from a database. The result is stored in a result table, called the result-set.

### Syntax

SELECT *column_name*,*column_name*
FROM *table_name*;

and

SELECT * FROM *table_name*;

### The SQL CREATE TABLE Statement

The CREATE TABLE statement is used to create a table in a database.

Tables are organized into rows and columns; and each table must have a name.

## SQL CREATE TABLE Statement

### Syntex

CREATE TABLE *table_name*
(
*column_name1 data_type*(*size*),
*column_name2 data_type*(*size*),
*column_name3 data_type*(*size*),
....
);

The column_name parameters specify the names of the columns of the table.

The data_type parameter specifies what type of data the column can hold (e.g. varchar, integer, decimal, date, etc.).

The size parameter specifies the maximum length of the column of the table.

## SQL ALTER TABLE Statement

The SQL ALTER TABLE statement is used to add, modify, or drop/delete columns in a table. The SQL ALTER TABLE statement is also used to rename a table.

## Syntax

(1) To add a column in a table, the SQL ALTER TABLE syntax is:

```
ALTER TABLE table_name
ADD column_name column-definition;
```

(2) To modify a column in an existing table, the SQL ALTER TABLE syntax is:

```
ALTER TABLE table_name
MODIFY column_name column_type;
```

(3) To drop a column in an existing table, the SQL ALTER TABLE syntax is:

```
ALTER TABLE table_name
DROP COLUMN column_name;
```

(4) To rename a column in an existing table, the SQL ALTER TABLE syntax is:

```
ALTER TABLE table_name
RENAME COLUMN old_name to new_name;
```

(5) To rename a table, the SQL ALTER TABLE syntax is:

```
ALTER TABLE table_name
RENAME TO new_table_name;
```

# SQL INSERT INTO Statement

## It is possible to write the INSERT INTO statement in two forms.

The first form does not specify the column names where the data will be inserted, only their values:

INSERT INTO *table_name*
VALUES (*value1*,*value2*,*value3*,...);

The second form specifies both the column names and the values to be inserted:

INSERT INTO *table_name* (*column1*,*column2*,*column3*,...)
VALUES (*value1*,*value2*,*value3*,...);

# SQL UPDATE Statement

The UPDATE statement is used to update existing records in a table.

## Syntax

```
UPDATE table_name
SET column1=value1, column2=value2, ...
WHERE some_column=some_value;
```

# SQL DELETE Statement

The DELETE statement is used to delete rows in a table.

## Syntax

DELETE FROM *table_name*
WHERE *some_column=some_value*;

# LAB 1

## CREATE TABLE TABLENAME(COLUMN1 DATATYPE(SIZE),COLUMN2 DATATYPE (SIZE) ,.....) IS USED TO CREATE TABLE

SQL> create table ajaysmarty(rno varchar(10),name varchar(15),contact number(10));

Table created.

## SELECT * FROM TABLENAME IS USED TO SELECT ALL DATA IN THE TABLE

SQL> select * from ajaysmarty;

| RNO | NAME | CONTACT |
|---|---|---|
| 2 | aman | 0 |

## INSERT INTO TABLENAME VALUES() IS USED TO INSERT A NEW ROW IN THE TABLE WITH PROVIDED VALUES.

SQL> insert into ajaysmarty values(1,'ajay',1234);

1 row created.

SQL> select * from ajaysmarty;

| RNO | NAME | CONTACT |
|---|---|---|
| 1 | ajay | 1234 |
| 2 | aman | 0 |

## INSERT INTO TABLENAME VALUES('&COLUMN1 ',&COLUMN2, .....) IS USED TO INSERT A ROW WITH SOME FIELD VALUE PROVIDED BY USER.

SQL> insert into ajaysmarty values('&rno','&name',&contact);

Enter value for rno: 3

Enter value for name: akshat

Enter value for contact: 1212

old 1: insert into ajaysmarty values('&rno','&name',&contact)

new 1: insert into ajaysmarty values('3','akshat',1212)

1 row created.

## A FORWARD SLASH (“/”) IS USED TO EXECUTE THE PREVIOUS COMMAND AGAIN

SQL> /

Enter value for rno: 4

Enter value for name: akshay

Enter value for contact: 1596

old 1: insert into ajaysmarty values('&rno','&name',&contact)

new 1: insert into ajaysmarty values('4','akshay',1596)

1 row created.

## RENAME TABLENAME TO TABLENAME1 IS USED TO RENAME THE TABLE FROM TABLENAME TO TABLENAME1

SQL> rename ajaysmarty to ajay133007;

Table renamed.

SQL> select * from ajay133007;

| RNO | NAME | CONTACT |
|---|---|---|
| 1 | ajay | 1234 |
| 3 | akshat | 1212 |
| 4 | akshay | 1596 |
| 5 | ankit | 1546 |
| 2 | aman | 0 |

## UPDATE TABLENAME SET COLUMN NAME=VALUE WHERE COLUMN NAME1=VALUE1 IS USED TO UPDATE SOME FIELD VALUE

SQL> update ajaysmarty set contact=1679 where rno='2';

1 row updated.

```
SQL> select * from ajaysmarty;

RNO        NAME         CONTACT
---------- ----------   ----------
1           ajay          1234
3          akshat         1212
4          akshay         1596
5          ankit          1546
2          aman           1679
```

## ALTER TABLE TABLENAME ADD(COLUMN DATATYPE(SIZE)) IS USED TO ADD A COLUMN IN THE TABLE

```
SQL> alter table ajaysmarty add(deptt varchar(5));

Table altered.


SQL> select * from ajaysmarty;

RNO        NAME     CONTACT    DEPTT
---------- ----------   ----------      -----
1          ajay         1234
3          akshat       1212
4          akshay       1596
5          ankit        1546
2          aman         1679
```

## ALTER TABLE TABLENAME MODIFY(COLUMN DATATYPE(SIZE)) IS USED TO CHANGE THE DATATYPE OR SIZE OF THE COLUMN

```
SQL> alter table ajaysmarty modify(name char(2));

alter table ajaysmarty modify(name char(2))
                              *

ERROR at line 1:

ORA-01441: cannot decrease column lenGth because some value is too biG
```

## DELETE FROM TABLENAME WHERE COLUMNNAME=VALUE IS USED TO DELETE A ROW FROM TABLE

SQL> delete from ajaysmarty where rno='4';

1 row deleted.

SQL> select * from ajaysmarty;

| RNO | NAME | CONTACT | DEPTT |
|---|---|---|---|
| 1 | ajay | 1234 | cse |
| 3 | akshat | 1212 | it |
| 5 | ankit | 1546 | mech |
| 2 | aman | 1679 | cse |

## SELECT COLUMNNAME FROM TABLENAME IS USED TO SELECT A PERTICULAR COLUMN FROM THE TABLE

SQL> select name from ajaysmarty;

NAME

---

ajay

akshat

ankit

aman

# LAB 2

## Transaction Control:

There are following commands used to control transactions:

- **COMMIT:** to save the changes.
- **ROLLBACK:** to rollback the changes.
- **SAVEPOINT:** creates points within groups of transactions in which to ROLLBACK

### The COMMIT Command :

The COMMIT command is the transactional command used to save changes invoked by a transaction to the database.

The COMMIT command saves all transactions to the database since the last COMMIT or ROLLBACK command.

**The syntax** for COMMIT command is as follows:

```
COMMIT;
```

### The ROLLBACK Command :

The ROLLBACK command is the transactional command used to undo transactions that have not already been saved to the database.

The ROLLBACK command can only be used to undo transactions since the last COMMIT or ROLLBACK command was issued.

**The syntax** for ROLLBACK command is as follows:

```
ROLLBACK;
```

### The SAVEPOINT Command:

A SAVEPOINT is a point in a transaction when you can roll the transaction back to a certain point without rolling back the entire transaction.

**The syntax** for SAVEPOINT command is as follows:

```
SAVEPOINT SAVEPOINT_NAME;
```

This command serves only in the creation of a SAVEPOINT among transactional statements. The ROLLBACK command is used to undo a group of transactions.

**The syntax** for rolling back to a SAVEPOINT is as follows:

```
ROLLBACK TO SAVEPOINT_NAME;
```

### The RELEASE SAVEPOINT Command :

The RELEASE SAVEPOINT command is used to remove a SAVEPOINT that you have created.

**The syntax** for RELEASE SAVEPOINT is as follows:

```
RELEASE SAVEPOINT SAVEPOINT_NAME;
```

Once a SAVEPOINT has been released, you can no longer use the ROLLBACK command to undo transactions performed since the SAVEPOINT.

## SQL TRUNCATE TABLE Command :

The SQL TRUNCATE TABLE command is used to delete complete data from an existing table.

### Syntax:

TRUNCATE TABLE table_name;

## SQL DROP TABLE Statement :

The DROP TABLE statement is used to delete a table.

### Syntax:

DROP TABLE table_name;

## Difference between TRUNCATE, DELETE and DROP commands

## DELETE

The DELETE command is used to remove rows from a table. A WHERE clause can be used to only remove some rows. If no WHERE condition is specified, all rows will be removed. After performing a DELETE operation you need to COMMIT or ROLLBACK the transaction to make the change permanent or to undo it.

## TRUNCATE

TRUNCATE removes **all rows** from a table. The operation cannot be rolled back and no triggers will be fired. As such, TRUCATE is faster and doesn't use as much undo space as a DELETE.

## DROP

The DROP command removes a table from the database. All the tables' rows, indexes and privileges will also be removed. No DML triggers will be fired. The operation cannot be rolled back.

# QUERIES

## COMMIT IS USED TO PERMANENTLY SAVE THE DATA.

SQL> commit;

Commit complete.

## SAVEPOINT SAVEPOINTNAME IS USED TO SAVE DATA TEMPORARILY

SQL> select * from ajaysmarty;

| RNO | NAME | CONTACT | DEPTT |
|---|---|---|---|
| 1 | ajay | 1234 | cse |
| 3 | akshat | 1212 | it |
| 4 | akshay | 1596 | |
| 5 | ankit | 1546 | |
| 2 | aman | 1679 | |

SQL> savepoint smarty1;

Savepoint created.

## NOW WE UPDATE THE TABLE

SQL> select * from ajaysmarty;

| RNO | NAME | CONTACT | DEPTT |
|---|---|---|---|
| 1 | ajay | 1234 | cse |
| 3 | akshat | 1212 | it |
| 4 | akshay | 1596 | bio |
| 5 | ankit | 1546 | mech |
| 2 | aman | 1679 | cse |

## ROLLBACK TO SAVEPOINT SAVEPOINTNAME IS USED TO UNDO THE CHANGES UPTO THE GIVEN SAVEPOINT, IF THE DATA IS NOT PERMANENTLY SAVED

SQL> rollback to savepoint smarty1;

Rollback complete.

SQL> select * from ajaysmarty;

| RNO | NAME | CONTACT | DEPTT |
|---|---|---|---|
| 1 | ajay | 1234 | cse |
| 3 | akshat | 1212 | it |
| 4 | akshay | 1596 | |
| 5 | ankit | 1546 | |
| 2 | aman | 1679 | |

## AFTER CREATING A SAVEPOINT WE CAN ROLLBACK ANY NUMBER OF TIMES UPTO IT UNTIL AN COMMIT COMMAND OR ANY AUTO COMMIT COMMAND IS EXECUTED.

SQL> savepoint smarty2;

Savepoint created.

SQL> rollback to smarty2;

Rollback complete.

SQL> rollback to smarty2;

Rollback complete.

## CREATION OF TABLE CLIENT MASTER

SQL> CREATE TABLE CLIENT_MASTER_07(CLIENTNO VARCHAR(6),NAME VARCHAR(20),ADDRESS1 VARCHAR(30), ADDRESS2 VARCHAR(30),CITY VARCHAR(15),PINCODE NUMBER(8),STATE VARCHAR(15),BALDUE NUMBER(10,2));

Table created.

## CREATION OF TABLE PRODUCT MASTER

SQL> CREATE TABLE PRODUCT_MASTER_07(PRODUCTNO VARCHAR(6),DESCRIPTION VARCHAR(15), PROFITPERCENT NUMBER(4,2),UNITMEASURE VARCHAR(10),QTYONHAND NUMBER(8),REORDERLVL NUMBER(8),SELLPRICE NUMBER (8,2),COSTPRICE NUMBER(8,2));

Table created.

## CREATION OF TABLE SALESMAN MASTER

```
SQL> CREATE TABLE SALESMAN_MASTER_07(SALESMANNO VARCHAR(6),SALESMANNAME VARCHAR(20),ADDRESS1
VARCHAR(30),ADDRESS2 VARCHAR(30),CITY VARCHAR(20),PINCODE NUMBER(8),STATE VARCHAR(20),SALAMT NUMB
ER(8,2),TGTTOGET NUMBER(6,2),YTDSALES NUMBER(6,2), REMARKS VARCHAR(60));

Table created.
```

## SHOWING THE CONTENT OF TABLE CLIENT MASTER

```
SQL> select clientno,name,city,pincode,state,baldue from client_master_07 ;
```

| CLIENT | NAME | CITY | PINCODE | STATE | BALDUE |
|---|---|---|---|---|---|
| c00001 | ivanbayross | mumbai | 400054 | maharashtra | 1500 |
| c00002 | mamta muzumdar | madras | 400057 | tamil nadu | 0 |
| c00003 | chhaya bankar | mumbai | 400057 | maharashtra | 5000 |
| c00004 | ashwini joshi | BanGalore | 560001 | Karnataka | 0 |
| c00005 | Hansel Colaco | Mumbai | 400060 | Maharashtra | 2000 |
| c00006 | Deepak Sharma | ManGalore | 560050 | Karnataka | 0 |

6 rows selected.

## SHOWING THE CONTENTS OF TABLE PRODUCT MASTER

```
SQL> SELECT * FROM PRODUCT_MASTER_07;
```

| PRODUCT | DESCRIPTION | PROFITPERCENT | UNITMEASUR | ATYONHAND | REORDERLVL | SELLPRICE | COSTPRICE |
|---|---|---|---|---|---|---|---|
| p00001 | T-shirts | 5 | piece | 200 | 50 | 350 | 250 |
| p0345 | Shirts | 6 | piece | 150 | 50 | 500 | 350 |
| p06734 | Cotton jeans | 5 | piece | 100 | 20 | 600 | 450 |
| p07865 | Jeans | 5 | piece | 100 | 20 | 750 | 500 |
| p07868 | Trousers | 2 | piece | 150 | 50 | 850 | 550 |
| p07885 | Pull overs | 2.5 | piece | 80 | 30 | 700 | 450 |
| p07965 | Denim shirts | 4 | piece | 100 | 40 | 350 | 250 |
| p07975 | Lycra Tops | 5 | piece | 70 | 30 | 300 | 175 |
| p07865 | Skirts | 5 | piece | 75 | 30 | 450 | 300 |

9 rows selected.

## SHOWING THE CONTENTS OF TABLE SALESMAN MASTER

```
SQL> select salesmanno,salesmanname,address1,address2,city,pincode,state from salesman_master_07 ;
```

| SALESMNO | SALESMANNAME | ADDRESS1 | ADDRESS2 | CITY | PINCODE | STATE |
|---|---|---|---|---|---|---|
| s00001 | Aman | A/14 | Worli | Mumbai | 400062 | Maharashtra |
| s00002 | Omkar | 65 | Nariman | Mumbai | 400001 | Maharashtra |
| s00003 | Raj | P-7 | Bandra | Mumbai | 400032 | Maharashtra |
| s00004 | Ashish | A/5 | Juhu | Mumbai | 400044 | Maharashtra |

```
SQL> select salesmanno,salesmanname,salamt,tGttoGet,ytdsales,remarks from salesman_master_07 ;
```

| SALESMNO | SALESMANNAME | SALAMT | TGTTOGET | YTDSALES | REMARKS |
|---|---|---|---|---|---|
| s00001 | Aman | 3000 | 100 | 50 | Good |
| s00002 | Omkar | 3000 | 200 | 100 | Good |
| s00003 | Raj | 3000 | 200 | 100 | Good |
| s00004 | Ashish | 3500 | 200 | 150 | Good |

## ALTER TABLE TABLENAME RENAME COLUMN COLUMN1 TO COLUMN2 IS USED TO RENAME THE COLUMN FROM COLUMN1 TO COLUMN2

```
SQL> alter table client_master_07 rename column name to smarty;

Table altered.
```

## DESC TABLENAME IS USED TO DESCRIBE ALL THE FIELDS PRESENT IN THE TABLE

```
SQL> desc client_master_07;
```

| Name | Null? | Type |
|---|---|---|
| CLIENTNO | | VARCHAR2(6) |
| SMARTY | | VARCHAR2(20) |
| ADDRESS1 | | VARCHAR2(30) |
| ADDRESS2 | | VARCHAR2(30) |
| CITY | | VARCHAR2(15) |

| | |
|---|---|
| PINCODE | NUMBER(8) |
| STATE | VARCHAR2(15) |
| BALDUE | NUMBER(10,2) |

SQL> alter table client_master_07 rename column smarty to NAME;

Table altered.

**ALTER TABLE TABLENAME DROP COLUMN COLUMNNAME IS USED TO DELETE THE COLUMN FROM TABLE**

SQL> ALTER TABLE SALESMAN_MASTER_07 DROP COLUMN SALESMANNO;

Table altered.

SQL> DESC SALESMAN_MASTER_07;

| Name | Null? | Type |
|---|---|---|
| SALESMANNAME | | VARCHAR2(20) |
| ADDRESS1 | | VARCHAR2(30) |
| ADDRESS2 | | VARCHAR2(30) |
| CITY | | VARCHAR2(20) |
| PINCODE | | NUMBER(8) |
| STATE | | VARCHAR2(20) |
| SALAMT | | NUMBER(8,2) |
| TGTTOGET | | NUMBER(6,2) |
| YTDSALES | | NUMBER(6,2) |
| REMARKS | | VARCHAR2(60) |

**AS THE ALTER COMMAND IS AUTO-COMMIT SO ROLLBACK COMMAND WILL NOT WORK AFTER IT.**

SQL> ROLLBACK;

Rollback complete.

```
SQL> DESC SALESMAN_MASTER_07;
 Name                      Null?    Type
 ------------------------- -------- ----------------------------
 SALESMANNAME                       VARCHAR2(20)
 ADDRESS1                           VARCHAR2(30)
 ADDRESS2                           VARCHAR2(30)
 CITY                               VARCHAR2(20)
 PINCODE                            NUMBER(8)
 STATE                              VARCHAR2(20)
 SALAMT                             NUMBER(8,2)
 TGTTOGET                           NUMBER(6,2)
 YTDSALES                           NUMBER(6,2)
 REMARKS                            VARCHAR2(60)
```

## AS ROLLBACK COMMAND DOESN'T WORK SO ADD THE COLUMN WITH ALTER TABLE COMMAND.

```
SQL> ALTER TABLE SALESMAN_MASTER_07 ADD(SALESMANNO VARCHAR(6));

Table altered.
```

## IF WE WANT TO INSERT DATA IN SOME PARTICULAR COLUMN THAN UPDATE COMMAND IS USED INSTEAD OF INSERT COMMAND.

```
SQL> UPDATE SALESMAN_MASTER_07 SET SALESMANNO='s00001' WHERE PINCODE=400002;

1 row updated.

SQL> UPDATE SALESMAN_MASTER_07 SET SALESMANNO='s00002' WHERE PINCODE=400001;

1 row updated.

SQL> UPDATE SALESMAN_MASTER_07 SET SALESMANNO='s00003' WHERE PINCODE=400032;
1 row updated.

SQL> UPDATE SALESMAN_MASTER_07 SET SALESMANNO='s00004' WHERE PINCODE=400044;

1 row updated.
```

**TRUNCATE TABLE TABLENAME IS USED TO DELETE ALL THE DATA FROM TABLENAME EXCEPT IT'S HEADER.**

SQL> TRUNCATE TABLE AJAY_07;

Table truncated.

**NO ROWS ARE SELECTED AS THE TABLE HAS BEEN TRUNCATED.**

SQL> SELECT * FROM AJAY_07;

no rows selected

**CREATE TABLE TABLENAME1 AS SELECT * FROM TABLENAME IS USED TO CREATE A COPY OF TABLE TABLENAME AS TABLENAME1**

SQL> CREATE TABLE SMARTYTABLE3 AS SELECT * FROM SALESMAN_MASTER_07;

Table created.

SQL> CREATE TABLE SMARTYTABLE1 AS SELECT * FROM CLIENT_MASTER_07;

Table created.

SQL> CREATE TABLE SMARTYTABLE2 AS SELECT * FROM PRODUCT_MASTER_07;

Table created.

# QUERIES

## SELECTING NAME FROM CLENT MASTER

```
SQL> SELECT NAME FROM SMARTYTABLE1;

NAME

--------------------

Ivan Bayross

Mamta Muzumdar

Chhaya Bankar

 Ashwini Joshi

Hansel Colaco

Deepak Sharma

6 rows selected.
```

## SHOWING THE CONTENTS OF CLIENT MASTER

```
SQL> SELECT * FROM SMARTYTABLE1;

CLIENT  NAME                 CITY          PINCODE   STATE            BALDUE

------  --------------------  ----------  ----------  ----------------  ----------

c00001  ivanbayross          mumbai       400054     maharashtra      1500

c00002  mamta muzumdar       madras       400057     tamil nadu       0

c00003  chhaya bankar        mumbai       400057     maharashtra      5000

c00004  ashwini joshi        BanGalore    560001     Karnataka        0

c00005  Hansel Colaco        Mumbai       400060     Maharashtra      2000

c00006  Deepak Sharma        ManGalore    560050     Karnataka        0

6 rows selected.
```

## SHOWING THE NAME,CITY AND STATE FROM TABLE CLIENT MASTER

```
SQL> SELECT NAME,CITY,STATE FROM SMARTYTABLE1;

NAME                 CITY          STATE
-------------------- ------------- ---------------
Ivan Bayross         Mumbai        Maharashtra
Mamta Muzumdar       Madras        Tamil Nadu
Chhaya Bankar        Mumbai        Maharashtra
Ashwini Joshi        BanGalore     Karnataka
Hansel Colaco        Mumbai        Maharashtra
Deepak Sharma        ManGalore     Karnataka

6 rows selected.
```

## SHOWING PRODUCTS AVAILABLE FROM TABLE PRODUCT MASTER

```
SQL> SELECT DESCRIPTION FROM SMARTYTABLE2;

DESCRIPTION
---------------
T-Shirts
Shirts
Cotton Jeans
Jeans
Trousers
Pull Overs
Denim Shirts
Lycra Tops
Skirts

9 rows selected.
```

## SELECT NAME FROM TABLE CLIENT MASTER WHO LIVED IN MUMBAI

SQL> SELECT NAME FROM SMARTYTABLE1 WHERE CITY='Mumbai';

NAME

---

Ivan Bayross

Chhaya Bankar

Hansel Colaco

## SHOWING NAMES OF SALESMAN WITH A SALARY OF RS 3000

SQL> SELECT SALESMANNAME FROM SMARTYTABLE3 WHERE SALAMT=3000;

SALESMANNAME

---

Aman

Omkar

Raj

## CHANGING THE CITY OF CLIENT WITH CLIENTNO C00005 TO BANGLORE

SQL> UPDATE SMARTYTABLE1 SET CITY='BanGlore' WHERE CLIENTNO='C00005';

1 row updated.

## CHANGING THE BALDUE OF CLIENT WITH CLIENTNO C00001 TO 1000

SQL> UPDATE SMARTYTABLE1 SET BALDUE=1000 WHERE CLIENTNO='C00001';

1 row updated.

## CHANGING THE COSTPRICE TO 950 WHERE DESCRIPTION IS TROUSERS

SQL> UPDATE SMARTYTABLE2 SET COSTPRICE=950.00 WHERE DESCRIPTION='Trousers';

1 row updated.

## CHANGING THE CITY OF ALL THE CLIENTS TO PUNE FROM MUMBAI

SQL> UPDATE SMARTYTABLE3 SET CITY='Pune' WHERE CITY='Mumbai';

4 rows updated.

## DELETING THE SALESMAN RECORD WITH SALARY 3500 RS

SQL> DELETE FROM SMARTYTABLE3 WHERE SALAMT=3500;

1 row deleted.

## DELETING THE ROW FROM PRODUCT MASTER WITH QTYONHAND TO BE 100

SQL> DELETE FROM SMARTYTABLE2 WHERE QTYONHAND=100;

3 rows deleted.

## DELETING FROM CLIENT MASTER WHERE STATE IS TAMILNADU

SQL> DELETE FROM SMARTYTABLE1 WHERE STATE='Tamil Nadu';

1 row deleted.

## ADD A COLUMN TO CLIENT MASTER WITH NAME TELEPHONE AND DATATYPE AND SIZE TO BE NUMBER AND 10 RESPECTIVELY

SQL> ALTER TABLE SMARTYTABLE1 ADD(Telephone number(10));

Table altered.

## MODIFY THE COLUMN SELLPRICE WITH DATATYPE NUMBER AND SIZE (10,2)

SQL> ALTER TABLE SMARTYTABLE2 MODIFY(SELLPRICE NUMBER(10,2));

Table altered.

## DROP THE TABLE CLIENT MASTER

SQL> DROP TABLE SMARTYTABLE1;

Table dropped.

## RENAME TABLE SALESMAN MASTER TO SMAN_MAST

SQL> RENAME SMARTYTABLE3 TO SMAN_MAST_07;

Table renamed.

# LAB 3

## SQL Constraints

SQL constraints are used to specify rules for the data in a table.

If there is any violation between the constraint and the data action, the action is aborted by the constraint.

Constraints can be specified when the table is created (inside the CREATE TABLE statement) or after the table is created (inside the ALTER TABLE statement).

### SQL CREATE TABLE + CONSTRAINT Syntax

CREATE TABLE *table_name*
(
*column_name1 data_type*(*size*) *constraint_name*,
*column_name2 data_type*(*size*) *constraint_name*,
*column_name3 data_type*(*size*) *constraint_name*,
....
);

In SQL, we have the following constraints:

- **NOT NULL** - Indicates that a column cannot store NULL value
- **UNIQUE** - Ensures that each row for a column must have a unique value
- **PRIMARY KEY** - A combination of a NOT NULL and UNIQUE. Ensures that a column (or combination of two or more columns) have an unique identity which helps to find a particular record in a table more easily and quickly
- **FOREIGN KEY** - Ensure the referential integrity of the data in one table to match values in another table
- **CHECK** - Ensures that the value in a column meets a specific condition
- **DEFAULT** - Specifies a default value when specified none for this column

## DISTINCT

In a table, a column may contain many duplicate values; and sometimes you only want to list the different (distinct) values.
The DISTINCT keyword can be used to return only distinct (different) values.

### SQL SELECT DISTINCT Syntax

SELECT DISTINCT *column_name,column_name*
FROM *table_name*;

# QUERIES

## CREATING A TABLE STUDENT_07

SQL> create table student_07(roll_no varchar(10),name varchar(30),dept varchar(10),addr varchar(30),phone number(10));

Table created.

## RETREIVING DATA FROM TABLE STUDENT_07

SQL> SELECT * FROM STUDENT_07;

| ROLL_NO | NAME | DEPT | ADDR | PHONE |
|---|---|---|---|---|
| UE133007 | AJAY | CSE | CHANDIGARH | 1234 |
| UE133008 | AKSHAT | MECH | UP | 1456 |
| UE133010 | AKSHAY KUCHHAL | DELHI | DELHI | 1598 |
| UE133011 | AKSHAY SHARMA | BIO | HIMACHAL | 1546 |
| UE133012 | AMANDEEP SINGH SAINI | IT | CHANDIGARH | 1546 |
| UE133015 | ANKIT KATHURIA | ECE | CHANDIGARH | 1587 |

6 rows selected.

## UPDATE TABLENAME SET COLUMN=*VALUE* WHERE COLUMN=*VALUE1* IS USED TO UPDATE AN ROW WHERE VALUE IN SET KEYWORD IS TO BE ADDED TO THE ROW HAVING VALUE1 OF THE COLUMN UNDER WHERE KEYWORD.

SQL> UPDATE STUDENT_07 SET DEPT='CSE' WHERE ROLL_NO='UE133010';

1 row updated.

```
SQL> SELECT * FROM STUDENT_07;

ROLL_NO     NAME                    DEPT       ADDR                PHONE
----------  ----------------------  ---------  ------------------  ----------
UE133007    AJAY                    CSE        CHANDIGARH          1234
UE133008    AKSHAT                  MECH       UP                  1456
UE133010    AKSHAY KUCHHAL          CSE        DELHI               1598
UE133011    AKSHAY SHARMA           BIO        HIMACHAL            1546
UE133012    AMANDEEP SINGH SAINI    IT         CHANDIGARH          1546
UE133015    ANKIT KATHURIA          ECE        CHANDIGARH          1587

6 rows selected.
```

**ALTER TABLE TABLENAME ADD CONSTRAINT CONSTRAINT_NAME CONSTRAINT ( COLUMN ) IS USED TO ADD A CONSTRAINT AFTER THE TABLE HAS BEEN CREATED.**

```
SQL> ALTER TABLE STUDENT_07 ADD CONSTRAINT SMARTY1 PRIMARY KEY(ROLL_NO);

Table altered.
```

**THE UNIQUE CONSTRAINT IS NOT ADDED TO THE COLUMN PHONE DUE TO THE PRESENCE OF DUPLICATE DATA IN THE COLUMN PHONE.**

```
SQL> ALTER TABLE STUDENT_07 ADD CONSTRAINT SMARTY2 UNIQUE(PHONE);

ALTER TABLE STUDENT_07 ADD CONSTRAINT SMARTY2 UNIQUE(PHONE)
                                      *

ERROR at line 1:

ORA-02299: cannot validate (STUDENT.SMARTY2) - duplicate keys found
```

**ALTER TABLE TABLENAME ADD CONSTRAINT CONSTRAINT_NAME CHECK(COLUMN IN(a,b,c,....) IS USED TO INSERT ONLY THOSE VALUE IN COLUMN WHICH ARE INCLUDED IN THE IN KEYWORD i.e., a,b,c...**

```
SQL> ALTER TABLE STUDENT_07 ADD CONSTRAINT SMARTY2 CHECK(DEPT IN ('CSE','IT','BI

O','MECH','ECE'));

Table altered.
```

**SELECT DISTINCT COLUMN_NAME FROM TABLENAME IS USED TO SELECT ALL THE DISTICT ROWS OF THE MENTIONED COLUMN IN THE TABLENAME.**

```
SQL> SELECT DISTINCT ROLL_NO,NAME,DEPT FROM STUDENT_07;

ROLL_NO      NAME                      DEPT
-----------  ------------------------  ---------
UE133007     AJAY                      CSE
UE133008     AKSHAT                    MECH
UE133010     AKSHAY KUCHHAL            CSE
UE133011     AKSHAY SHARMA             BIO
UE133012     AMANDEEP SINGH SAINI      IT
UE133015     ANKIT KATHURIA            ECE

6 rows selected.
```

# LAB 4

## SQL DUAL table

The DUAL is special one row, one column table present by default in all Oracle databases. The owner of DUAL is SYS (SYS owns the data dictionary, therefore DUAL is part of the data dictionary.) but DUAL can be accessed by every user. The table has a single VARCHAR2(1) column called DUMMY that has a value of 'X'. MySQL allows DUAL to be specified as a table in queries that do not need data from any tables. In SQL Server DUAL table does not exist, but you could create one.

## SQL Arithmetic Operators

Arithmetic operators can perform arithmetical operations on numeric operands involved. Arithmetic operators are addition(+), subtraction(-), multiplication(*) and division(/). The + and - operators can also be used in date arithmetic.

## Syntax

```
SELECT <Expression>[arithmetic operator]<expression>...
 FROM [table_name]
 WHERE [expression] ;
```

| Parameter | Description |
|---|---|
| Expression | Expression made up of a single constant, variable, scalar function, or column name and can also be the pieces of a SQL query that compare values against other values or perform arithmetic calculations. |
| arithmetic operator | Plus(+), minus(-), multiply(*), and divide(/). |
| table_name | Name of the table. |

## SQL Aliases

SQL aliases are used to give a database table, or a column in a table, a temporary name. Basically aliases are created to make column names more readable.

### SQL Alias Syntax for Columns

SELECT *column_name* AS *alias_name*
FROM *table_name;*

## SQL Alias Syntax for Tables

SELECT *column_name(s)*
FROM *table_name* AS *alias_name;*

# The SQL INSERT INTO SELECT Statement

The INSERT INTO SELECT statement selects data from one table and inserts it into an existing table. Any existing rows in the target table are unaffected.

## SQL INSERT INTO SELECT Syntax

We can copy all columns from one table to another, existing table:

INSERT INTO *table2*
SELECT * FROM *table1;*

Or we can copy only the columns we want to into another, existing table:

INSERT INTO *table2*
*(column_name(s))*
SELECT *column_name(s)*
FROM *table1;*

# QUERIES

## SELECT MARKS+50 FROM TABLENAME IS USED TO DISPLAY A COLUMN WITH MARKS 50 MORE THAN THE ORIGINAL MARKS. THIS COLUMN IS NOT PERMANENTLY ADDED TO THE TABLE.

```
SQL> SELECT MARKS+50 FROM STUDENT_07;

MARKS+50
----------
       950
       850
       750
       650
       550
       450

6 rows selected.
```

## ALICING A COLUMN NAME USING OPTIONAL AS KEYWORD.

```
SQL> SELECT MARKS+50 AS "Marks" FROM STUDENT_07;

  Marks
----------
 950
 850
 750
 650
 550
 450

6 rows selected.
```

## IF WE DON'T USE THE QUOTATION MARKS IN THE ALICING NAME THAN BY DEFAULT ALICE IS DISPLAYED IN CAPITAL LETTERS.

```
SQL> SELECT MARKS+50 AS Marks FROM STUDENT_07;

 MARKS
 ----------
 950
 850
 750
 650
 550
 450
6 rows selected.
```

## WE CAN SKIP THE AS KEYWORD AS IT IS OPTIONAL.

```
SQL> SELECT MARKS+50 Marks FROM STUDENT_07;

 MARKS
 ----------
 950
 850
 750
 650
 550
 450
 6 rows selected.
```

## CREATE TABLE TABLENAME1 AS SELECT * FROM TABLENAME IS USED TO CREATE THE COPY OF THE TABLE.

```
SQL> CREATE TABLE AJAY_07 AS SELECT * FROM STUDENT_07;

Table created.
```

SQL> SELECT * FROM AJAY_07;

| ROLL_NO | NAME | DEPT | ADDR | PHONE |
|---|---|---|---|---|
| UE133007 | AJAY | CSE | CHANDIGARH | 1234 |
| UE133008 | AKSHAT | MECH | UP | 1456 |
| UE133010 | AKSHAY KUCHHAL | CSE | DELHI | 1598 |
| UE133011 | AKSHAY SHARMA | BIO | HIMACHAL | 1546 |
| UE133012 | AMANDEEP SINGH SAINI | IT | CHANDIGARH | 1546 |
| UE133015 | ANKIT KATHURIA | ECE | CHANDIGARH | 1587 |

6 rows selected.

SQL> ALTER TABLE AJAY_07 DROP COLUMN ROLL_NO;

Table altered.

**INSERT INTO TABLENAME1 SELECT COLUMN FROM TABLENAME IS USED TO INSERT A COLUMN FROM TABLENAME TO TABLENAME1.**

SQL> INSERT INTO AJAY_07(ROLL_NO) SELECT ROLL_NO FROM STUDENT_07;

6 rows created.

SQL> SELECT * FROM STUDENT_07;

| ROLL_NO | NAME | DEPT | ADDR | PHONE | MARKS |
|---|---|---|---|---|---|
| UE133007 | AJAY | CSE | CHANDIGARH | 1234 | 900 |
| UE133008 | AKSHAT | MECH | UP | 1456 | 800 |
| UE133010 | AKSHAY KUCHHAL | CSE | DELHI | 1598 | 700 |
| UE133011 | AKSHAY SHARMA | BIO | HIMACHAL | 1546 | 600 |
| UE133012 | AMANDEEP SINGH SAINI | IT | CHANDIGARH | 1546 | 500 |
| UE133015 | ANKIT KATHURIA | ECE | CHANDIGARH | 1587 | 400 |

6 rows selected.

# LAB 5

## SQL character function

A character or string function is a function which takes one or more characters or numbers as parameters and returns a character value. Basic string functions offer a number of capabilities and return a string value as a result set.

| Functions | **Description** |
|---|---|
| lower() | The SQL LOWER() function is used to convert all characters of a string to lower case. |
| upper() | The SQL UPPER() function is used to convert all characters of a string to uppercase. |
| trim() | The SQL TRIM() removes leading and trailing characters(or both) from a character string. |
| translate() | The SQL TRANSLATE() function replaces a sequence of characters in a string with another sequence of characters. The function replaces a single character at a time. |

## Numeric Functions

SQL numeric functions are used primarily for numeric manipulation and/or mathematical calculations. The following table details the numeric functions:

| Name | Description |
|---|---|
| ABS() | Returns the absolute value of numeric expression. |
| CEIL() | Returns the smallest integer value that is not less than passed numeric expression |
| DEGREES() | Returns numeric expression converted from radians to degrees. |
| EXP() | Returns the base of the natural logarithm (e) raised to the power of passed numeric expression. |
| FLOOR() | Returns the largest integer value that is not greater than passed numeric expression. |
| GREATEST() | Returns the largest value of the input expressions. |
| INTERVAL() | Takes multiple expressions exp1, exp2 and exp3 so on.. and returns 0 if exp1 is less than exp2, returns 1 if exp1 is less than exp3 and so on. |
| LEAST() | Returns the minimum-valued input when given two or more. |
| LOG() | Returns the natural logarithm of the passed numeric expression. |
| LOG10() | Returns the base-10 logarithm of the passed numeric expression. |
| MOD() | Returns the remainder of one expression by diving by another expression. |

| | |
|---|---|
| PI() | Returns the value of pi |
| POW() | Returns the value of one expression raised to the power of another expression |
| RADIANS() | Returns the value of passed expression converted from degrees to radians. |
| ROUND() | Returns numeric expression rounded to an integer. Can be used to round an expression to a number of decimal points |

## SQL GROUP Functions

Group functions are built-in SQL functions that operate on groups of rows and return one value for the entire group. These functions are: COUNT, MAX, MIN, AVG, SUM, DISTINCT .

**SQL COUNT ():** This function returns the number of rows in the table that satisfies the condition specified in the WHERE condition. If the WHERE condition is not specified, then the query returns the total number of rows in the table.
**SQL DISTINCT():** This function is used to select the distinct rows.
**SQL MAX():** This function is used to get the maximum value from a column.
**SQL MIN():** This function is used to get the minimum value from a column.
**SQL AVG():** This function is used to get the average value of a numeric column.
**SQL SUM():** This function is used to get the sum of a numeric column.

## SQL Comparison Keywords

There are other comparison keywords available in sql which are used to enhance the search capabilities of a sql query. They are "IN", "BETWEEN...AND", "IS NULL", "LIKE".

| Comparision Operators | Description |
|---|---|
| LIKE | column value is similar to specified character(s). |
| IN | column value is equal to any one of a specified set of values. |
| BETWEEN...AND | column value is between two values, including the end values specified in the range. |
| IS NULL | column value does not exist. |

## SQL Logical Operators

There are three Logical Operators namely, AND, OR, and NOT. These operators compare two conditions at a time to determine whether a row can be selected for the output. When retrieving data using a SELECT statement, you can use logical operators in the WHERE clause, which allows you to combine more than one condition.

| Logical Operators | Description |
|---|---|
| OR | For the row to be selected at least one of the conditions must be true. |
| AND | For a row to be selected all the specified conditions must be true. |
| NOT | For a row to be selected the specified condition must be false. |

## TABLE USED FOR QUERIES :

SQL> SELECT * FROM STUDENT_07;

| ROLL_NO | NAME | DEPT | ADDR | PHONE |
|---|---|---|---|---|
| UE133007 | AJAY | CSE | CHANDIGARH | 1234 |
| UE133008 | AKSHAT | MECH | UP | 1456 |
| UE133010 | AKSHAY KUCHHAL | CSE | DELHI | 1598 |
| UE133011 | AKSHAY SHARMA | BIO | HIMACHAL | 1546 |
| UE133012 | AMANDEEP SINGH SAINI | IT | CHANDIGARH | 1546 |
| UE133015 | ANKIT KATHURIA | ECE | CHANDIGARH | 1587 |

6 rows selected.

# QUERIES

## USING MAX FUNCTION. IT IS USED TO FIND THE MAXIMUM VALUE IN THE MENTIONED COLUMN.

```
SQL> SELECT MAX(MARKS) FROM STUDENT_07;

MAX(MARKS)

----------

 900
```

## USING AVG FUNCTION. IT IS USED TO FIND THE AVERAGE OF ALL NUMBERS OF THE MENTIONED COLUMN.

```
SQL> SELECT AVG(MARKS) FROM STUDENT_07;

AVG(MARKS)

----------

 650
```

## USING SUM FUNCTION. IT IS USED TO FIND THE SUM OF THE VALUES OF THE MENTIONED COLUMN.

```
SQL> SELECT SUM(MARKS) FROM STUDENT_07;

SUM(MARKS)

----------

 3900
```

## USING MIN FUNCTION. IT IS USED TO FIND THE MINIMUM VALUE IN THE MENTIONED COLUMN.

```
SQL> SELECT MIN(MARKS) FROM STUDENT_07;

MIN(MARKS)

----------

 400
```

**USING COUNT FUNCTION.** THIS WILL COUNT THE NUMBER OF NOT NULL ROWS FROM THE MARKS COLUMN.

```
SQL> SELECT COUNT(MARKS) FROM STUDENT_07;
 COUNT(MARKS)
 ------------
 6
```

**USING COUNT(*) FUNCTION.** THIS WILL COUNT THE TOTAL NUMBER OF ROWS IN THE TABLE.

```
SQL> SELECT COUNT(*) FROM STUDENT_07;

COUNT(*)

----------
 6
```

**USING ABS FUNCTION.** THIS WILL CALCULATE THE ABSOLUTE VALUE.

```
SQL> SELECT ABS(-50) FROM DUAL;

 ABS(-50)

----------

  50
```

**USING POWER FUNCTION.** IT IS A MATHEMATICAL FIUNCTION TO CALCULATE THE POWER OF ANY NUMBER.

```
SQL> SELECT POWER(5,2) FROM DUAL;

POWER(5,2)

----------

 25
```

**USING ROUND FUNCTION.** IT IS USED TO ROUND OFF THE PROVIDED VALUE UPTO ANY PRECISION.

```
SQL> SELECT ROUND(5.256) FROM DUAL;

 ROUND(5.256)

------------

  5
```

**USING TRUNC FUNCTION.** IT IS USED TO TRUNCATE THE PROVIDED VALUE UPTO ANY PRECISION.

```
SQL> SELECT TRUNC(2.56,1) FROM DUAL;

TRUNC(2.56,1)
-------------
 2.5
```

**USING MOD FUNCTION.** THIS IS USED TO CALCULATE THE REMAINDER.

```
SQL> SELECT MOD(5,2) FROM DUAL;

 MOD(5,2)
----------
 1
```

**USING GREATEST FUNCTION.** IT IS USED TO FIND THE GREATEST OF THE NUMBERS PROVIDED.

```
SQL> SELECT GREATEST(5,2) FROM DUAL;

 GREATEST(5,2)

-------------

 5
```

**USING GREATEST FUNCTION.** HERE WE CAN NOT COMPARE THE INTEGER AND CHARACTER VALUES.

```
SQL> SELECT GREATEST(5,2,A) FROM DUAL;

SELECT GREATEST(5,2,A) FROM DUAL

                *

ERROR at line 1:

ORA-00904: "A": invalid identifier
```

**USING LOWER(COLUMN) FUNCTION.** USED TO CONVERT THE DATA IN PARTICULAR COLUMN TO LOWERCASE.

```
SQL> SELECT LOWER(NAME) FROM STUDENT_07;

LOWER(NAME)

------------------------------
ajay
```

akshat

akshay kuchhal

akshay sharma

amandeep sinGh saini

ankit kathuria

6 rows selected.

**USING UPPER(COLUMN) FUNCTION.** USED TO CONVERT THE DATA IN PARTICULAR COLUMN TO UPPERCASE.

SQL> SELECT UPPER(NAME) FROM STUDENT_07;

UPPER(NAME)

---

AJAY

AKSHAT

AKSHAY KUCHHAL

AKSHAY SHARMA

AMANDEEP SINGH SAINI

ANKIT KATHURIA

6 rows selected.

**USING INITCAP(NAME) FUNCTION.** USED TO CONVERT THE DATA IN PARTICULAR COLUMN TO INITCAP.

SQL> SELECT INITCAP(NAME) FROM STUDENT_07;

INITCAP(NAME)

---

Ajay

Akshat

Akshay Kuchhal

Akshay Sharma

Amandeep SinGh Saini

Ankit Kathuria

6 rows selected.

**USING SUBSTR(str,m,n) FUNCTION. THIS WILL PROVIDE YOU THE STRING FROM Mth CHARACTER TO THE Nth CHARACTER.**

```
SQL> SELECT SUBSTR('AJAY SHARMA',2,7) FROM DUAL;

SUBSTR(
-------
JAY SHA
```

**USING INSTR(str1,str2,m) FUNCTION. THIS WILL PROVIDE YOU THE POSITION OF STR2 IN STR1 AT IT'S Mth OCCURENCE.**

```
SQL> SELECT INSTR('ABC XYZABC','ABC',2) FROM DUAL;

INSTR('ABCXYZABC','ABC',2)
--------------------------
 8
```

**USING TRANSLATE(str,str1,str2) FUNCTION. THIS WILL REPLACE STR1 BY STR2 IN STR.**

```
SQL> SELECT TRANSLATE('ABC XYZABC','ABC','BDK') FROM DUAL;

TRANSLATE(

----------

BDK XYZBDK
```

**SELECT SYSDATE FROM DUAL IS USED TO SELECT THE CURRENT DATE FROM THE SYSTEM**

```
SQL> select sysdate from dual;

SYSDATE

---------

21-JAN-15
```

**SELECT * FROM TABLENAME WHERE COLUMN BETWEEN T1 AND T2 IS USED TO DISPLAY THOSE ROWS HAVING THEIR COLUMN VALUE IN BETWEEN T1 AND T2 OR EQUAL TO T1 AND T2.**

```
SQL> select * from student_07 where marks between 100 and 700;

ROLL_NO     NAME                    DEPT       ADDR                PHONE     MARKS
----------  ----------------------  ---------  ------------------  ----------  ----------
UE133010    AKSHAY KUCHHAL          CSE        DELHI               1598        700
```

| | | | | | |
|---|---|---|---|---|---|
| UE133011 | AKSHAY SHARMA | BIO | HIMACHAL | 1546 | 600 |
| UE133012 | AMANDEEP SINGH SAINI | IT | CHANDIGARH | 1546 | 500 |
| UE133015 | ANKIT KATHURIA | ECE | CHANDIGARH | 1587 | 400 |

4 rows selected.

## SELECT COLUMN_NAME FROM TABLENAME WHERE COLUMN LIKE '_J%' IS USED WHEN THE VALUE OF THE SPECIFIED COLUMN IS HAVING IT'S FIRST CHARACTER ANYTHING(DENOTED BY _ ), SECOND CHARACTER MUST BE 'J' AND ANY STRING AFTER THAT(DENOTED BY %).

```
SQL> SELECT NAME FROM STUDENT_07 WHERE NAME LIKE'_J%';

NAME

--------------------

AJAY
```

## USING THE NOT LIKE CONDITION

```
SQL> SELECT NAME FROM STUDENT_07 WHERE NAME NOT LIKE'_J%';

NAME

-----------------------------

AKSHAT

AKSHAY KUCHHAL

AKSHAY SHARMA

AMANDEEP SINGH SAINI

ANKIT KATHURIA
```

## TABLES USED FOR QUERIES :

## CREATING TABLE SALES_ORDER_7.

SQL> CREATE TABLE SALES_ORDER_7(ORDERNO VARCHAR(10),CLIENTNO VARCHAR(10),ORDERDATE DATE,SALESMAN NO VARCHAR(10),DELYTYPE CHAR(1),BILLYN CHAR(1),DELYDATE DATE,ORDERSTATUS VARCHAR(30));

Table created.

## DISPLAYING TABLE SALES_ORDER_7.

SQL> select * from SALES_ORDER_7;

| ORDERN | CLIENT | ORDERDATE | SALESM | D | B | DELYDATE | ORDERSTATU |
|---|---|---|---|---|---|---|---|
| o19001 | c00001 | 20-JUL-02 | s00001 | f | n | 12-JUN-04 | in process |
| o19002 | c00002 | 27-JUN-02 | s00002 | p | n | 25-JUN-04 | cancelled |
| o46865 | c00003 | 20-FEB-02 | s00003 | f | y | 18-FEB-04 | fulfilled |
| o19003 | c00001 | 07-APR-02 | s00001 | f | y | 03-APR-04 | fulfilled |
| o46866 | c00004 | 22-MAY-02 | s00002 | p | n | 20-MAY-04 | cancelled |
| o19008 | c00005 | 26-JUL-02 | s00004 | f | n | 24-MAY-04 | in process |

6 rows selected.

## CREATING TABLE SALES_ORDER_DETAILS_7

SQL> create table SALES_ORDER_DETAILS_7(orderno varchar(6) references SALES_ORDER_7(orderno),productno varchar(6) references PRODUCT_MASTER_7(productno),qtyordered number(8),qtydisp number(8),productrate numb er(10,2),primary key(orderno));

Table created.

## DISPLAYING CONTENTS OF SALES_ORDER_DETAILS_7

SQL> select * from SALES_ORDER_DETAILS_7;

| ORDERN | PRODUC | QTYORDERED | QTYDISP | PRODUCTRATE |
|---|---|---|---|---|
| o19001 | p00001 | 4 | 4 | 525 |
| o19001 | p07965 | 2 | 1 | 8400 |
| o19001 | p07885 | 2 | 1 | 5250 |
| o19002 | p00001 | 10 | 0 | 525 |
| o46865 | p07868 | 3 | 3 | 3150 |
| o46865 | p07885 | 3 | 1 | 5250 |
| o46865 | p00001 | 10 | 10 | 525 |
| o46865 | p0345 | 4 | 4 | 1050 |
| o19003 | p0345 | 2 | 2 | 1050 |
| o19003 | p06734 | 1 | 1 | 12000 |
| o46866 | p07965 | 1 | 0 | 8400 |
| o4686 | p07975 | 1 | 0 | 1050 |
| o19008 | p00001 | 10 | 5 | 525 |
| o19008 | p07975 | 5 | 3 | 1050 |

14 rows selected.

# QUERIES

## SELECTING NAME HAVING SECOND CHARACTER AS a.

SQL> SELECT NAME FROM CLIENT_MASTER_7 WHERE NAME like'_a%';

NAME

---

mamta muzumar

hansel colaco

## SELECTING CITY HAVING FIRST CHARACTER AS m.

SQL> SELECT CITY FROM CLIENT_MASTER_7 WHERE CITY like'm%';

CITY

---

mumbai

madras

mumbai

mumbai

manGalore

## SELECTING NAME WHERE CITY IS HAVING FIRST CHARACTER AS m.

SQL> SELECT NAME FROM CLIENT_MASTER_7 WHERE CITY like'm%';

NAME

---

ivan bayross

mamta muzumar

chaya bankar

hansel colaco

deepak sharma

## SELECTING NAME WHO LIVES IN MANGLORE OR BANGLORE .

```
SQL> SELECT NAME FROM CLIENT_MASTER_7 WHERE CITY in('manGlore','banGlore');

 NAME

--------------------

ashwini joshi

deepak sharma
```

## SELECTING NAME WHERE BALDUE IS GREATER THAN 10000.

```
SQL> SELECT NAME FROM CLIENT_MASTER_7 WHERE baldue>10000;

NAME

--------------------

ivan bayross
```

## SELECTING DATA WHERE CLIENTNO IS EITHER C00001 OR C00002.

```
SQL> select * from SALES_ORDER_7 where clientno in('c00001','c00002');

ORDERN    CLIENT     ORDERDATE    SALESM    D    B      DELYDATE      ORDERSTATU

----------    ----------    ---------------       ----------    ---    ----    --------------    ----------------

o19001      c00001     20-JUL-02        s00001    f      n     12-JUN-04     in process

o19002      c00002     27-JUN-02        s00002    p      n     25-JUN-04     cancelled

o19003       c00001    07-APR-02        s00001    f      y      03-APR-04    fulfilled
```

## SELECTING DESCRIPTION WHERE SELLPRICE IN BETWEEN 500 AND 750.

```
SQL> select description from PRODUCT_MASTER_7 where sellprice BETWEEN 500 AND 750;

DESCRIPTION

---------------

Shirts

Pull overs
```

## SELECTING DESCRIPTION WHERE SELLPRICE IS GREATER THAN 500.

```
SQL> select description from PRODUCT_MASTER_7 where sellprice>500;

DESCRIPTION
---------------
Trousers
Pull overs
```

## SELECTING SELLPRICE*15 FROM PRODUCT MASTER.

```
SQL> select sellprice*15 from PRODUCT_MASTER_7;

SELLPRICE*15
------------
5250
7500
12750
10500
4500
6750
6 rows selected.
```

## SELECTING NAME, CITY, STATE FROM CLIENT MASTER WHERE STATE IS NOT MAHARASHTRA.

```
SQL> select name,city,state from client_master_19  where state not in('maharashtra');

NAME                CITY           STATE
--------------------  ---------------  ---------------
mamta muzumdar  madras         tamil nadu
ashwini joshi          banGalore     karnataka
deepak sharma        manGalore    karnataka
```

## COUNT TOTAL NUMBER OF ORDERS.

```
SQL> select count(orderno) total_orders from sales_order_16;

TOTAL_ORDERS
------------
6
```

## FIND AVERAGE COSTPRICE FROM PRODUCT MASTER.

```
SQL> select AVG(COSTPRICE) from PRODUCT_MASTER_7;

AVG(COSTPRICE)

--------------

 345.833333
```

## FIND MAX SELLPRICE FROM PRODUCT MASTER.

```
SQL> select max(sellprice) max_price from PRODUCT_MASTER_7;

 MAX_PRICE

----------

  850
```

## FIND AVERAGE PRODUCT RATE FROM SALES ORDER DETAILS.

```
SQL> select avG(productrate) averaGe_price from SALES_ORDER_DETAILS_7;

 AVERAGE_PRICE

-------------

  3482.14286
```

## FIND MAXIMUM AND MINIMUM PRODUCT RATE FROM SALES ORDER DETAILS.

```
SQL> select max(productrate) max_price,min(productrate) min_price from SALES_ORDER_DETAILS_7;

 MAX_PRICE    MIN_PRICE

----------    ----------

 12000          525
```

## FIND TOTAL NUMBER OF PRODUCTS HAVING PRODUCT RATE LESS THAN OR EQUAL TO 500.

SQL> select count(orderno) count from SALES_ORDER_DETAILS_7 where productrate<=500;

COUNT

---

0

## FIND MINIMUM SELLPRICE FROM PRODUCT MASTER.

SQL> select min(sellprice) min_price from PRODUCT_MASTER_7;

MIN_PRICE

---

300

## SELECT DESCRIPTION WHERE SELLPRICE IS LESS THAN OR EQUAL TO 500.

SQL> select description from PRODUCT_MASTER_7 where sellprice<=500;

DESCRIPTION

---

T-shirts

Shirts

Lycra Tops

Skirts

## SELECT DESCRIPTION WHERE QTYONHAND IS LESS THAN REORDERLVL.

SQL> select description from PRODUCT_MASTER_7 where qtyonhand<reorderlvl;

no rows selected

# LAB 6

## SQL ORDER BY Keyword

The ORDER BY keyword is used to sort the result-set by one or more columns.

The ORDER BY keyword sorts the records in ascending order by default. To sort the records in a descending order, you can use the DESC keyword.

### Syntax

SELECT *column_name*, *column_name*
FROM *table_name*
ORDER BY *column_name* ASC|DESC, *column_name* ASC|DESC;

## GROUP BY Statement

The GROUP BY statement is used in conjunction with the aggregate functions to group the result-set by one or more columns.

### Syntax

SELECT column_name, aggregate_function(column_name)
FROM table_name
WHERE column_name operator value
GROUP BY column_name;

## SQL Subqueries

**A subquery is a SQL query nested inside a larger query.**

- A subquery may occur in :
  - - A SELECT clause
  - - A FROM clause
  - - A WHERE clause
- The subquery can be nested inside a SELECT, INSERT, UPDATE, or DELETE statement or inside another subquery.
- A subquery is usually added within the WHERE Clause of another SQL SELECT statement.
- You can use the comparison operators, such as >, <, or =. The comparison operator can also be a multiple-row operator, such as IN, ANY, or ALL.
- A subquery can be treated as an inner query, which is a SQL query placed as a part of another query called as outer query.
- The inner query executes first before its parent query so that the results of inner query can be passed to the outer query.

## Type of Subqueries

- Single row subquery : Returns zero or one row.

- Multiple row subquery : Returns one or more rows.

- Correlated subqueries : Reference one or more columns in the outer SQL statement. The subquery is known as a correlated subquery because the subquery is related to the outer SQL statement.

# LAB 7

## SQL JOINING

### SQL Cross Join

The SQL CROSS JOIN produces a result set which is the number of rows in the first table multiplied by the number of rows in the second table, if no WHERE clause is used along with CROSS JOIN. This kind of result is called a Cartesian Product. If, WHERE clause is used with CROSS JOIN, it functions like an INNER JOIN.
An alternative way of achieving the same result is to use column names separated by commas after SELECT and mentioning the table names involved, after a FROM clause.

#### Syntax

```
SELECT *
FROM table1
CROSS JOIN table2;
```

### SQL Natural Join

We have already learned that an EQUI JOIN performs a JOIN against equality or matching column(s) values of the associated tables and an equal sign (=) is used as comparison operator in the where clause to refer equality.
The SQL NATURAL JOIN is a type of EQUI JOIN and is structured in such a way that, columns with same name of associate tables will appear once only.

#### Natural Join : Guidelines

- The associated tables have one or more pairs of identically named columns.
- The columns must be the same data type.
- Don’t use ON clause in a natural join.

#### Syntax

```
SELECT *
FROM table1
NATURAL JOIN table2;
```

### SQL Outer Join

The SQL OUTER JOIN returns all rows from both the participating tables which satisfy the join condition along with rows which do not satisfy the join condition. The SQL OUTER JOIN operator (+) is used only on one side of the join condition only.

**The subtypes of SQL OUTER JOIN**

- LEFT OUTER JOIN or LEFT JOIN
- RIGHT OUTER JOIN or RIGHT JOIN
- FULL OUTER JOIN

## SQL Left Join

The SQL LEFT JOIN (specified with the keywords LEFT JOIN and ON) joins two tables and fetches all matching rows of two tables for which the sql-expression is true, plus rows from the fri st table that do not match any row in the second table.

### Left Join : Syntax

```
SELECT *
FROM table1
LEFT [ OUTER ] JOIN table2
ON table1.column_name=table2.column_name;
```

## SQL Right Join

The SQL RIGHT JOIN, joins two tables and fetches rows based on a condition, which are matching in both the tables ( before and after the JOIN clause mentioned in the syntax below) , and the unmatched rows will also be available from the table written after the JOIN clause ( mentioned in the syntax below ).

### Syntax

```
SELECT *
FROM table1
RIGHT [ OUTER ] JOIN table2
ON table1.column_name=table2.column_name;
```

## SQL Full Outer Join

In SQL the FULL OUTER JOIN combines the results of both left and right outer joins and returns all (matched or unmatched) rows from the tables on both sides of the join clause.

### Syntax

```
SELECT *
FROM table1
FULL OUTER JOIN table2
ON table1.column_name=table2.column_name;
```

# QUERIES

## USING THE ORDER BY CLAUSE WILL SORT THE TABLE ROWS ACCORDING TO THE COLUMN MENTIONED.

```
SQL> SELECT * FROM STUDENT_07;

ROLL_NO     NAME                    DEPT       ADDR                PHONE
----------  ----------------------  ---------  ------------------  ----------
UE133007    AJAY                    CSE        CHANDIGARH          1234
UE133008    AKSHAT                  MECH       UP                  1456
UE133010    AKSHAY KUCHHAL          CSE        DELHI               1598
UE133011    AKSHAY SHARMA           BIO        HIMACHAL            1546
UE133012    AMANDEEP SINGH SAINI    IT         CHANDIGARH          1546
UE133015    ANKIT KATHURIA          ECE        CHANDIGARH          1587

6 rows selected.

SQL> SELECT * FROM STUDENT_07 ORDER BY ROLL_NO;

ROLL_NO     NAME                    DEPT       ADDR                PHONE
----------  ----------------------  ---------  ------------------  ----------
UE133007    AJAY                    CSE        CHANDIGARH          1234
UE133008    AKSHAT                  MECH       UP                  1456
UE133010    AKSHAY KUCHHAL          CSE        DELHI               1598
UE133011    AKSHAY SHARMA           BIO        HIMACHAL            1546
UE133012    AMANDEEP SINGH SAINI    IT         CHANDIGARH          1546
UE133015    ANKIT KATHURIA          ECE        CHANDIGARH          1587

6 rows selected.
```

## WHEN TWO COLUMNS ARE INCLUDED IN THE ORDER BY CLAUSE THAN FIRTLY THE SORTING IS DONE ACCORDING TO THE FIRST MENTIONED COLUMN AND IF THEIR IS ANY DUPLICACY IN FIRST COLUMN THAN SORTING IS DONE ACCORDING TO THE SECOND MENTIONED COLUMN.

SQL> SELECT * FROM STUDENT_07 ORDER BY NAME,DEPT;

| ROLL_NO | NAME | DEPT | ADDR | PHONE |
|---|---|---|---|---|
| UE133007 | AJAY | CSE | CHANDIGARH | 1234 |
| UE133008 | AKSHAT | MECH | UP | 1456 |
| UE133010 | AKSHAY KUCHHAL | CSE | DELHI | 1598 |
| UE133011 | AKSHAY SHARMA | BIO | HIMACHAL | 1546 |
| UE133012 | AMANDEEP SINGH SAINI | IT | CHANDIGARH | 1546 |
| UE133015 | ANKIT KATHURIA | ECE | CHANDIGARH | 1587 |

6 rows selected.

## TWO TABLES ARE USED ARE EMP_7 & DEPT_7.

SQL> select * from emp_7;

| ENO | ENAME | JOB | MANAGE | JOINDATE | SALARY | DEPTNO |
|---|---|---|---|---|---|---|
| UEE133001 | AKSHAY | PROGRAMMER | M13001 | 10-JAN-13 | 100000 | 1 |
| UEE133002 | AKSHAT | DBA | M13002 | 10-JAN-13 | 90000 | 1 |
| UEE133003 | AJAY | PROJ LED | M13003 | 10-FEB-13 | 85000 | 2 |
| UEE133004 | ANKIT | HR | M13004 | 15-FEB-13 | 80000 | 3 |
| UEE133005 | AMAN | PL | M13005 | 14-MAR-13 | 75000 | 4 |

SQL> select * from dept_7;

| DEPTNO | DNAME | DLOC |
|---|---|---|
| 1 | CSE | MUMBAI |
| 2 | IT | DELHI |
| 3 | ECE | BANGALORE |
| 4 | COMP | GURGOAN |

## MAKING DEPTNO AS PRIMARY KEY ON TABLE DEPT_7.

SQL> alter table dept_7 modify(deptno number(10) primary key);

Table altered.

## MAKING DEPTNO AS FOREIGN KEY REFERENCING DEPTNO FROM DEPT_7 TABLE.

SQL> alter table emp_7 modify(deptno number(10) references dept_7(deptno));

Table altered.

## USE OF GROUP BY CLAUSE.

SQL> select DEPTNO,SUM(salary) TOTAL from emp_7 Group by DEPTNO;

| DEPTNO | TOTAL |
|---|---|
| 1 | 190000 |
| 2 | 85000 |
| 4 | 75000 |
| 3 | 80000 |

SQL> select JOB,AVG(salary) avG from emp_7 Group by job;

| JOB | AVG |
|---|---|
| DBA | 90000 |
| HR | 80000 |
| PROGRAMMER | 100000 |
| PROJ LED | 80000 |

## IF WE INCLUDE ANY EXTRA COLUMN IN SELECT COLUMN OTHER THAN AGGREGATE FUNCTION THAN IT MUST BE INCLUDED IN GROUP BY CLAUSE.

```
SQL> select deptno,job,AVG(salary) avG from emp_7 Group by deptno,job;

 DEPTNO  JOB                 AVG
 ----------  ------------------- ----------
 4           PROJ LED          75000
 3           HR                80000
 1           DBA               90000
 1           PROGRAMMER  100000
 2           PROJ LED          85000
```

```
SQL> select job, sum(salary),max(salary),min(salary),avG(salary) from emp_7 Group by job;

JOB                 SUM(SALARY)    MAX(SALARY)    MIN(SALARY)    AVG(SALARY)
------------------- -----------    -----------    -----------    -----------
DBA                 90000          90000          90000          90000
HR                  80000          80000          80000          80000
PROGRAMMER  100000         100000         100000         100000
PROJ LED            160000         85000          75000          80000
```

```
SQL> select sum(salary),max(salary),min(salary),avG(salary) from emp_7 where deptno=1  Group by job;

SUM(SALARY)     MAX(SALARY)     MIN(SALARY)     AVG(SALARY)
-----------     -----------     -----------     -----------
 90000           90000           90000           90000
 100000          100000          100000          100000
```

## IF WE WANT TO PUT CONDITION ON THE AGGREGATE FUNCTION THAN HAVING KEYWORD IS USED.

```
SQL> select job,sum(salary),max(salary),min(salary),avG(salary) from emp_7 where deptno=1  Group by job havinG avG(salary)>10000;

JOB                 SUM(SALARY)    MAX(SALARY)    MIN(SALARY)    AVG(SALARY)
------------------- -----------    -----------    -----------    -----------
PROGRAMMER          100000         100000         100000         100000
```

## USE OF ORDER BY CLAUSE.

```
SQL> select ename,deptno from emp_7 order by deptno;

ENAME                DEPTNO
-------------------- ----------
AKSHAY               1
AKSHAT               1
AJAY                 2
ANKIT                3
AMAN                 4

SQL> select ename,job from emp_7 order by job;

ENAME      JOB
---------- --------------------
AKSHAT     DBA
ANKIT      HR
AKSHAY     PROGRAMMER
AMAN       PROJ LED
AJAY       PROJ LED
```

## USE OF SUBQUERIES

## HERE NONE OF THE ROWS GET SELECTED AS DATA STORED IN TABLE IS ' AKSHAY ' AND WE ARE SEARCHING FOR ' akshay '.

SQL> select ename,deptno from emp_7 where deptno=(select deptno from emp_7 where ename='akshay');

no rows selected

## HERE WE USE CHARACTER MANUPULATION FUNCTION FOR SEARCHING THE ENAME.

SQL> select ename,deptno from emp_7 where deptno=(select deptno from emp_7 where lower(ename) = 'akshay');

ENAME DEPTNO

---------- ----------

AKSHAY 1

AKSHAT 1

SQL> select ename from emp_7 where job=(select job from emp_7 where eno='UEE133003');

ENAME

----------

AJAY

SQL> select dname,e.deptno from dept_7 d,emp_7 e where e.deptno=d.deptno and e.deptno=(select deptno from emp_7 where manaGe='M13003');

DNAME DEPTNO

---------- ----------

IT 2

SQL> select ename,job from emp_7 where job=(select job from emp_7 where eno='UEE133001') and salary> (select salary from emp_7 where eno='UEE133002');

ENAME JOB

---------- ----------

AKSHAY PROGRAMMER

SQL> select ename,salary from emp_7 where salary>(select min(salary) from emp_7 e,dept_7 d where e.deptno= d.deptno and d.dname='IT');

```
ENAME          SALARY
----------     ----------
AKSHAY         100000
AKSHAT         90000
```

SQL> select salary,ename from emp_7 where salary<(select AVG(salary) from emp_7 where deptno=(select deptno from dept_7 where dname='IT')) and joindate<'15-may-13';

```
    SALARY   ENAME
    ----------  ----------
     80000   ANKIT
     75000   AMAN
```

SQL> select ename,salary from emp_7 where salary>(select avG(salary) from emp_7 e,dept_7 d where e.deptno=d.deptno and d.dname='IT') and joindate<'15-may-13';

```
ENAME    SALARY
----------    ----------
AKSHAY    100000
AKSHAT    90000
```

SQL> select dname,ename from dept_7 d,emp_7 e where e.deptno=d.deptno and e.deptno=(select deptno from emp_7 where manaGe='M13003');

```
DNAME   ENAME
----------   ----------
IT             AJAY
```

SQL> select ename,salary,dname from dept_7 d,emp_7 e where e.deptno=d.deptno and salary=(select min(salary) from emp_7);

```
ENAME   SALARY   DNAME
----------   ----------   ----------
AMAN     75000     COMP
```

```
SQL> select ename,job from EMP_7 e where e.salary < (select max(salary) from EMP_7 where job='DBA') and e.job !
='DBA';

ENAME   JOB

----------  --------------------

AJAY        PROJ LED

ANKIT       HR

AMAN      PROJ LED


SQL> select ename from EMP_7 where salary >all (select avG(salary) from EMP_7 Group by deptno);

ENAME

----------

AKSHAY
```

# LAB 8

## SQL Views

A view is a virtual table.

### CREATE VIEW Statement

In SQL, a view is a virtual table based on the result-set of an SQL statement.
A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database.

You can add SQL functions, WHERE, and JOIN statements to a view and present the data as if the data were coming from one single table.

### SQL CREATE VIEW Syntax

```
CREATE VIEW view_name AS
SELECT column_name(s)
FROM table_name
WHERE condition;
```

### UPDATING A VIEW

**SQL CREATE OR REPLACE VIEW Syntax**

```
CREATE OR REPLACE VIEW view_name AS
SELECT column_name(s)
FROM table_name
WHERE condition;
```

### DROPPING A VIEW

**SQL DROP VIEW Syntax**

```
DROP VIEW view_name;
```

# SEQUENCE

A sequence is a set of integers 1, 2, 3, … that are generated in order on demand. Sequences are frequently used in databases because many applications require each row in a table to contain a unique value, and sequences provide an easy way to generate them.

**Syntex**

CREATE SEQUENCE seq_person

MINVALUE  value

START WITH  value

INCREMENT BY  value

CACHE / NOCACHE

CYCLE / NOCYCLE;

# LAB 9

## INDEX

An index can be created in a table to find data more quickly and efficiently.

The users cannot see the indexes, they are just used to speed up searches/queries.

**Note:** Updating a table with indexes takes more time than updating a table without (because the indexes also need an update). So you should only create indexes on columns (and tables) that will be frequently searched against.

### SQL CREATE INDEX Syntax

Creates an index on a table. Duplicate values are allowed:

```
CREATE INDEX index_name
ON table_name (column_name);
```

### SQL CREATE UNIQUE INDEX Syntax

Creates a unique index on a table. Duplicate values are not allowed:

```
CREATE UNIQUE INDEX index_name
ON table_name (column_name);
```

## DROPING A INDEX

**SYNTEX**

```
DROP INDEX TABLE_NAME.INDEX_NAME;
```

## ROW ID

For each row in the database, the `ROWID` pseudocolumn returns the address of the row. Oracle Database rowid values contain information necessary to locate a row:

- The data object number of the object
- The data block in the datafile in which the row resides
- The position of the row in the data block (first row is 0)
- The datafile in which the row resides (first file is 1). The file number is relative to the tablespace.

# ROW NUM

Rownum is a pseudo column. It numbers the records in a result set. The first record that meets the where criteria in a select statement is given rownum=1, and every subsequent record meeting that same criteria increases rownum.

After issuing a *select* statement, one of the last steps that oracle does is to assign an increasing (starting with 1, increased by 1) number to each row returned. The value of this row number can always be queried with rownum in a select statement.

# VIEWS, INDEXES & SEQUENCE

## TABLES USED

## (1) EMPLOYEE TABLE:

SQL> select * from emp_7;

| ENO | ENAME | JOB | MANAGE | JOINDATE | SALARY | DEPTNO |
|---|---|---|---|---|---|---|
| UEE133001 | AKSHAY | PROGRAMMER | M13001 | 10-JAN-13 | 100000 | 1 |
| UEE133002 | AKSHAT | DBA | M13002 | 10-JAN-13 | 90000 | 1 |
| UEE133003 | AJAY | PROJ LED | M13003 | 10-FEB-13 | 85000 | 2 |
| UEE133004 | ANKIT | HR | M13004 | 15-FEB-13 | 80000 | 3 |
| UEE133005 | AMAN | PROJ LED | M13005 | 14-MAR-13 | 75000 | 4 |

## (2) DEPARTMENT TABLE:

SQL> select * from dept_7;

| DEPTNO | DNAME | DLOC |
|---|---|---|
| 1 | CSE | MUMBAI |
| 2 | IT | DELHI |
| 3 | ECE | BANGLORE |
| 4 | COMP | GURGAON |

# VIEWS

## CREATE VIEW *VIEW_NAME* AS SELECT *COLUMN_NAMES* FROM *TABLENAME* IS USED TO CREATE A VIEW ON THE SINGLE TABLE.

```
SQL> create view view_emp_7 as select eno,ename,job,salary from emp_7;

View created.
```

## RETREIVING DATA FROM VIEW.

```
SQL> select * from view_emp_7;

ENO         ENAME     JOB                 SALARY
----------  ----------  ------------------  ----------
UEE133001   AKSHAY     PROGRAMMER   100000
UEE133002   AKSHAT     DBA                 90000
UEE133003   AJAY       PROJ LED            85000
UEE133004   ANKIT      HR                  80000
UEE133005   AMAN       PROJ LED            75000
```

## INSERTION IN VIEW:

## INSERT INTO VIEW_NAME VALUES(.....) IS USED TO INSERT VALUE INTO THE VIEW.

```
SQL> INSERT INTO VIEW_EMP_7 VALUES('UE133006','AISHWARYA','PROJ LED',85000);

1 row created.
```

## INSERTION INTO VIEW SUCCEEDED:

```
SQL> SELECT * FROM VIEW_EMP_7;

ENO         ENAME      JOB                 SALARY
----------  ----------  ------------------  ----------
UEE133001   AKSHAY     PROGRAMMER   100000
UEE133002   AKSHAT     DBA                 90000
UEE133003   AJAY       PROJ LED            85000
UEE133004   ANKIT      HR                  80000
UEE133005   AMAN       PROJ LED            75000
UE133006    AISHWARYA  PROJ LED            85000

6 rows selected.
```

## INSERTION INTO SINGLE TABLE SUCCEEDED USING INSERTION INTO VIEWS:

```
SQL> SELECT * FROM EMP_7;

ENO        ENAME      JOB                MANAGE     JOINDATE      SALARY     DEPTNO
---------- ---------- ------------------- ---------- --------- ---------- ----------
UEE133001  AKSHAY     PROGRAMMER  M13001     10-JAN-13     100000     1
UEE133002  AKSHAT     DBA         M13002     10-JAN-13     90000      1
UEE133003  AJAY       PROJ LED    M13003     10-FEB-13     85000      2
UEE133004  ANKIT      HR          M13004     15-FEB-13     80000      3
UEE133005  AMAN       PROJ LED    M13005     14-MAR-13     75000      4
UE133006 AISHWARYA PROJ LED                                85000

6 rows selected.
```

## INSERTING VALUES INTO EMPLOYEE TABLE:

```
SQL> INSERT INTO EMP_7 VALUES('UE133007','ABHISHEK','DBA','M13007','14-MAR-13',80000,2);

1 row created.
```

## INSERTION INTO TABLE SUCCEEDED:

```
SQL> SELECT * FROM EMP_7;

ENO        ENAME      JOB                MANAGE     JOINDATE      SALARY     DEPTNO
---------- ---------- ------------------- ---------- --------- ---------- ----------
UEE133001  AKSHAY     PROGRAMMER  M13001     10-JAN-13     100000     1
UEE133002  AKSHAT     DBA         M13002     10-JAN-13     90000      1
UEE133003  AJAY       PROJ LED    M13003     10-FEB-13     85000      2
UEE133004  ANKIT      HR          M13004     15-FEB-13     80000      3
UEE133005  AMAN       PROJ LED    M13005     14-MAR-13     75000      4
UE133006   AISHWARYA  PROJ LED                                85000
UE133007   ABHISHEK   DBA         M13007 14-MAR-13 80000      2

7 rows selected.
```

## INSERTION INTO VIEW SUCCEEDED BY INSERTING INTO TABLE:

```
SQL> SELECT * FROM VIEW_EMP_7;

ENO        ENAME      JOB                 SALARY
---------- ---------- ------------------- ----------
UEE133001  AKSHAY     PROGRAMMER  100000
UEE133002  AKSHAT     DBA          90000
UEE133003  AJAY       PROJ LED     85000
UEE133004  ANKIT      HR           80000
UEE133005  AMAN       PROJ LED     75000
UE133006   AISHWARYA  PROJ LED     85000
UE133007   ABHISHEK   DBA          80000

7 rows selected.
```

## CREATING VIEW ON TWO JOINED TABLES:

```
SQL> create view view_emp_dept as (select eno,ename,salary,dname,e.deptno from emp_7 e join dept_7
d on e.deptno=d.deptno);

View created.
```

## INSERTING VALUES IN ONE TABLE EMPLOYEES:

```
SQL> INSERT INTO EMP_7 VALUES('UE133008','ANSHIKA','HR','M13007','20-MAR-13',75000,1);

1 row created.
```

## INSERTION INTO EMPLOYEE TABLE SUCCEEDED:

```
SQL> SELECT * FROM EMP_7;

ENO        ENAME      JOB                 MANAGE     JOINDATE      SALARY     DEPTNO
---------- ---------- ------------------- ---------- ---------     ---------- ----------
UEE133001  AKSHAY     PROGRAMMER  M13001     10-JAN-13     100000     1
UEE133002  AKSHAT     DBA          M13002     10-JAN-13     90000      1
UEE133003  AJAY       PROJ LED     M13003     10-FEB-13     85000      2
UEE133004  ANKIT      HR           M13004     15-FEB-13     80000      3
UEE133005  AMAN       PROJ LED     M13005     14-MAR-13     75000      4
UE133006   AISHWARYA  PROJ LED                               85000
UE133007   ABHISHEK   DBA          M13007     14-MAR-13     80000      2
UE133008   ANSHIKA    HR           M13007     20-MAR-13     75000      1

8 rows selected.
```

## INSERTION INTO VIEW USING INSERTION INTO TABLE SUCCEEDED:

```
SQL> SELECT * FROM VIEW_EMP_DEPT;

ENO        ENAME      SALARY     DNAME      DEPTNO
---------- ---------- ---------- ---------- ----------
UEE133001  AKSHAY     100000     CSE        1
UEE133002  AKSHAT     90000      CSE        1
UEE133003  AJAY       85000      IT         2
UEE133004  ANKIT      80000      ECE        3
UEE133005  AMAN       75000      COMP       4
UE133007   ABHISHEK   80000      IT         2
UE133008   ANSHIKA    75000      CSE        1

7 rows selected.
```

## INSERTION INTO VIEW IS NOT SUCCEEDED IF THE VIEW IS CREATED ON JOINED TABLES.

```
SQL> INSERT INTO VIEW_EMP_DEPT VALUES('UE133009','ANKITA',56000,'ECE',3);
INSERT INTO VIEW_EMP_DEPT VALUES('UE133009','ANKITA',56000,'ECE',3)
*
ERROR at line 1:
ORA-01779: cannot modify a column which maps to a non key-preserved table
```

# SEQUENCE

## CREATING A SEQUENCE:

```
SQL> CREATE SEQUENCE SEQ_AJAY
  2  INCREMENT BY 2
  3  START WITH 100
  4  MAXVALUE 200
  5  NOCYCLE;

Sequence created.
```

## INSERTING ENO IN EMPLOYEE TABLE USING *SEQUENCE_NAME.NEXTVAL*:

```
SQL> INSERT INTO EMP_7(ENO,ENAME) VALUES(SEQ_AJAY.NEXTVAL,'&NAME');
Enter value for name: ADITI
old   1: INSERT INTO EMP_7(ENO,ENAME) VALUES(SEQ_AJAY.NEXTVAL,'&NAME')
new   1: INSERT INTO EMP_7(ENO,ENAME) VALUES(SEQ_AJAY.NEXTVAL,'ADITI')

1 row created.

SQL> /
Enter value for name: KAJOL
old   1: INSERT INTO EMP_7(ENO,ENAME) VALUES(SEQ_AJAY.NEXTVAL,'&NAME')
new   1: INSERT INTO EMP_7(ENO,ENAME) VALUES(SEQ_AJAY.NEXTVAL,'KAJOL')

1 row created.

SQL> /
Enter value for name: SMARTY
old   1: INSERT INTO EMP_7(ENO,ENAME) VALUES(SEQ_AJAY.NEXTVAL,'&NAME')
new   1: INSERT INTO EMP_7(ENO,ENAME) VALUES(SEQ_AJAY.NEXTVAL,'SMARTY')

1 row created.
```

## RETREIVING DATA FROM TABLE:

```
SQL> SELECT * FROM EMP_7;
```

| ENO | ENAME | JOB | MANAGE | JOINDATE | SALARY | DEPTNO |
|---|---|---|---|---|---|---|
| UEE133001 | AKSHAY | PROGRAMMER | M13001 | 10-JAN-13 | 100000 | 1 |
| UEE133002 | AKSHAT | DBA | M13002 | 10-JAN-13 | 90000 | 1 |
| UEE133003 | AJAY | PROJ LED | M13003 | 10-FEB-13 | 85000 | 2 |
| UEE133004 | ANKIT | HR | M13004 | 15-FEB-13 | 80000 | 3 |
| UEE133005 | AMAN | PROJ LED | M13005 | 14-MAR-13 | 75000 | 4 |
| UE133006 | AISHWARYA | PROJ LED | | | 85000 | |
| UE133007 | ABHISHEK | DBA | M13007 | 14-MAR-13 | 80000 | 2 |
| UE133008 | ANSHIKA | HR | M13007 | 20-MAR-13 | 75000 | 1 |
| 100 | ADITI | | | | | |
| 102 | KAJOL | | | | | |
| 104 | SMARTY | | | | | |

```
11 rows selected.
```

## ALTER THE SEQUENCE: BUT YOU CAN NOT ALTER THE START VALUE OF A SEQUENCE.

```
SQL> ALTER SEQUENCE SEQ_AJAY
  2  INCREMENT BY 40;

Sequence altered.
```

## INSERTING THE VALUE IN EMPLOYEE TABLE:

```
SQL> INSERT INTO EMP_7(ENO,ENAME) VALUES(SEQ_AJAY.NEXTVAL,'&NAME');
Enter value for name: BHARTI
old   1: INSERT INTO EMP_7(ENO,ENAME) VALUES(SEQ_AJAY.NEXTVAL,'&NAME')
new   1: INSERT INTO EMP_7(ENO,ENAME) VALUES(SEQ_AJAY.NEXTVAL,'BHARTI')

1 row created.

SQL> /
Enter value for name: SOMBVANSHI
old   1: INSERT INTO EMP_7(ENO,ENAME) VALUES(SEQ_AJAY.NEXTVAL,'&NAME')
new   1: INSERT INTO EMP_7(ENO,ENAME) VALUES(SEQ_AJAY.NEXTVAL,'SOMBVANSHI')

1 row created.
```

## RETREIVING DATA FROM EMPLOYEE TABLE:

SQL> SELECT * FROM EMP_7;

| ENO | ENAME | JOB | MANAGE | JOINDATE | SALARY | DEPTNO |
|---|---|---|---|---|---|---|
| UEE133001 | AKSHAY | PROGRAMMER | M13001 | 10-JAN-13 | 100000 | 1 |
| UEE133002 | AKSHAT | DBA | M13002 | 10-JAN-13 | 90000 | 1 |
| UEE133003 | AJAY | PROJ LED | M13003 | 10-FEB-13 | 85000 | 2 |
| UEE133004 | ANKIT | HR | M13004 | 15-FEB-13 | 80000 | 3 |
| UEE133005 | AMAN | PROJ LED | M13005 | 14-MAR-13 | 75000 | 4 |
| UE133006 | AISHWARYA | PROJ LED | | | 85000 | |
| UE133007 | ABHISHEK | DBA | M13007 | 14-MAR-13 | 80000 | 2 |
| UE133008 | ANSHIKA | HR | M13007 | 20-MAR-13 | 75000 | 1 |
| 100 | ADITI | | | | | |
| 102 | KAJOL | | | | | |
| 104 | SMARTY | | | | | |
| 144 | BHARTI | | | | | |
| 184 | SOMBVANSHI | | | | | |

13 rows selected.

# INDEX

## CREATE INDEX INDEX_NAME ON TABLE_NAME(COLUMN_NAME) IS USED TO CREATE INDEX ON A PARTICULAR COLUMN OF A TABLE.

```
SQL> CREATE INDEX INDEX_EMP ON EMP_7(ENO);

Index created.
```

## DROP INDEX INDEX_NAME IS USED TO DROP A CREATED INDEX.

```
SQL> DROP INDEX INDEX_EMP;

Index dropped.
```

## ADDING PRIMARY KEY CONSTRAINT TO THE ENO COLUMN OF EMPLOYEE TABLE.

```
SQL> ALTER TABLE EMP_7 MODIFY (ENO PRIMARY KEY);

Table altered.
```

## CREATING INDEX ON THE COLUMN HAVING PRIMARY KEY CONSTRAINT.

## AN ERROR OCCURED AS THE PRIMARY KEY AND UNIQUE KEY CONSTRAINT AUTOMATICALLY FORM AN INDEX ON THE COLUMN WHOSE NAME IS SAME AS THAT OF CONSTRAINT NAME.

```
SQL> CREATE INDEX INDEX_EMP ON EMP_7(ENO);
CREATE INDEX INDEX_EMP ON EMP_7(ENO)
                          *
ERROR at line 1:
ORA-01408: such column list already indexed
```

## CREATING COMPOSITE INDEX:

```
SQL> CREATE INDEX INDEX2_EMP ON EMP_7(ENO,ENAME);

Index created.
```

## INDEX IS CREATED IN THE DECSENDING VALUES OF ENO.

```
SQL> CREATE INDEX INDEX3_EMP ON EMP_7(ENO DESC);

Index created.
```

# ROW ID AND ROW NUM

## RETREIVING DATA FROM EMPLOYEE TABLE WITH ROWID AND ROWNUM:

SQL> SELECT ROWID,ROWNUM,ENO,ENAME,SALARY,DEPTNO FROM EMP_7;

| ROWID | ROWNUM | ENO | ENAME | SALARY | DEPTNO |
|---|---|---|---|---|---|
| AAAM3MAAEAAAAHUAAA | 1 | UEE133001 | AKSHAY | 100000 | 1 |
| AAAM3MAAEAAAAHUAAB | 2 | UEE133002 | AKSHAT | 90000 | 1 |
| AAAM3MAAEAAAAHUAAC | 3 | UEE133003 | AJAY | 85000 | 2 |
| AAAM3MAAEAAAAHUAAD | 4 | UEE133004 | ANKIT | 80000 | 3 |
| AAAM3MAAEAAAAHUAAE | 5 | UEE133005 | AMAN | 75000 | 4 |
| AAAM3MAAEAAAAHUAAF | 6 | UE133006 | AISHWARYA | 85000 | 2 |
| AAAM3MAAEAAAAHUAAG | 7 | UE133007 | ABHISHEK | 80000 | 2 |
| AAAM3MAAEAAAAHUAAH | 8 | UE133008 | ANSHIKA | 75000 | 1 |

8 rows selected.

## ROWID OR ROWNUM ARE NOT CHANGED WITH THE USE OF ORDER BY CLAUSE:

SQL> SELECT ROWID,ROWNUM,ENO,ENAME,SALARY,DEPTNO FROM EMP_7 ORDER BY ENAME;

| ROWID | ROWNUM | ENO | ENAME | SALARY | DEPTNO |
|---|---|---|---|---|---|
| AAAM3MAAEAAAAHUAAG | 7 | UE133007 | ABHISHEK | 80000 | 2 |
| AAAM3MAAEAAAAHUAAF | 6 | UE133006 | AISHWARYA | 85000 | 2 |
| AAAM3MAAEAAAAHUAAC | 3 | UEE133003 | AJAY | 85000 | 2 |
| AAAM3MAAEAAAAHUAAB | 2 | UEE133002 | AKSHAT | 90000 | 1 |
| AAAM3MAAEAAAAHUAAA | 1 | UEE133001 | AKSHAY | 100000 | 1 |
| AAAM3MAAEAAAAHUAAE | 5 | UEE133005 | AMAN | 75000 | 4 |
| AAAM3MAAEAAAAHUAAD | 4 | UEE133004 | ANKIT | 80000 | 3 |
| AAAM3MAAEAAAAHUAAH | 8 | UE133008 | ANSHIKA | 75000 | 1 |

8 rows selected.

## DELETING SOME ROWS FROM EMPLOYEE TABLE:

SQL> DELETE EMP_7 WHERE SALARY=80000;

2 rows deleted.

**AFTER THE DELETE OPERATION THE ROWID REMAINS THE SAME BUT ROWNUM IS AGAIN ALLOCATED TO THE REMAINING ROWS.**

```
SQL> SELECT ROWID,ROWNUM,ENO,ENAME,SALARY,DEPTNO FROM EMP_7;

ROWID              ROWNUM   ENO        ENAME      SALARY   DEPTNO
------------------ ---------- ---------- ---------- ---------- ----------
AAAM3MAAEAAAAHUAAA  1         UEE133001  AKSHAY     100000    1
AAAM3MAAEAAAAHUAAB  2         UEE133002  AKSHAT     90000     1
AAAM3MAAEAAAAHUAAC  3         UEE133003  AJAY       85000     2
AAAM3MAAEAAAAHUAAE  4         UEE133005  AMAN       75000     4
AAAM3MAAEAAAAHUAAF  5         UE133006   AISHWARYA  85000     2
AAAM3MAAEAAAAHUAAH  6         UE133008   ANSHIKA    75000     1

6 rows selected.
```

# LAB 10

## INTRODUCTION TO PL/SQL

PL/SQL stands for Procedural Language extension of SQL. PL/SQL is a combination of SQL along with the procedural features of programming languages.

### The PL/SQL Engine:

Oracle uses a PL/SQL engine to processes the PL/SQL statements. A PL/SQL language code can be stored in the client system (client-side) or in the database (server-side).

### A Simple PL/SQL Block:

Each PL/SQL program consists of SQL and PL/SQL statements which from a PL/SQL block.

### PL/SQL Block consists of three sections:

- The Declaration section (optional).
- The Execution section (mandatory).
- The Exception Handling (or Error) section (optional).

### Declaration Section:

The Declaration section of a PL/SQL Block starts with the reserved keyword DECLARE. This section is optional and is used to declare any placeholders like variables, constants, records and cursors, which are used to manipulate data in the execution section. Placeholders may be any of Variables, Constants and Records, which stores data temporarily. Cursors are also declared in this section.

### Execution Section:

The Execution section of a PL/SQL Block starts with the reserved keyword BEGIN and ends with END. This is a mandatory section and is the section where the program logic is written to perform any task. The programmatic constructs like loops, conditional statement and SQL statements form the part of execution section.

### Exception Section:

The Exception section of a PL/SQL Block starts with the reserved keyword EXCEPTION. This section is optional. Any errors in the program can be handled in this section, so that the PL/SQL Blocks terminates gracefully. If the PL/SQL Block contains exceptions that cannot be handled, the Block terminates abruptly with errors.

## How a Sample PL/SQL Block Looks

```
DECLARE
    Variable declaration

BEGIN
    Program Execution

EXCEPTION
    Exception handling

END ;
```

# LAB 11

## CONTROL STRUCTURES

There may be a situation when you need to execute a block of code several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

| Loop Type | Description |
|---|---|
| Basic LOOP | In this loop structure, sequence of statements is enclosed between the LOOP and END LOOP statements. At each iteration, the sequence of statements is executed and then control resumes at the top of the loop. |
| WHILE LOOP | Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body. |
| FOR LOOP | Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable. |
| Nested loops | You can use one or more loop inside any another basic loop, while or for loop. |

### IF-THEN Statement

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

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

### IF-THEN-ELSE Statement

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

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

## IF-THEN-ELSIF Statement

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

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

## CASE Statement

```
CASE selector
  WHEN expression1 THEN sequence_of_statements1;
  WHEN expression2 THEN sequence_of_statements2;
  ...
  WHEN expressionN THEN sequence_of_statementsN;
 [ELSE sequence_of_statementsN+1;]
END CASE [label_name];
```

## WHILE-LOOP

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

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

## SIMPLE-LOOP

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

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

## FOR-LOOP

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

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

# QUERIES

```
SQL> set serveroutput on;
```

## DISPLAYING TABLE ACCOUNT:

```
ACC_NO NAME      BALANCE
---------- ---------- ----------
 100        ajay        9500
 200        akshay      5000
 300        akshat      7500
 400        ankit       3000
 500        aman        6000
```

## DISPLAYING TABLE EMPLOYEE:

```
EMPID    EMP_NAME SALARY
---------- ----------          ----------
1          ankit               10000
2          ajay                9000
3          akshay              7000
4          aman                12000
5          akshat              15000
```

# QUESTIONS

## DISPLAY THE SALARY OF THE PERSON HAVING EMPLOYEE ID AS 2.

```
SQL> declare
  2 ajay_empid number(5):=2;
  3  ajay_sal number(10);
  4  beGin
  5 select salary into ajay_sal from employee_07
  6 where empid=ajay_empid;
  7 dbms_output.put_line('salary of emp id '||ajay_empid||'is '||ajay_sal);
  8  end;
  9  /
```

**salary of emp id 2 is 9000**

PL/SQL procedure successfully completed.

## USING THE ACCOUNT NUMBER PROVIDED BY THE USER DEBIT RS 500 FROM THAT ACCOUNT IF THE BALANCE LEFT IS MORE THAN RS 1000.

```
SQL> declare
  2 ajay_acc_no number(5);
  3 ajay_balance number(10);
  4 ajay_min_balance number(10):=1000;
  5 ajay_debit_amt number(10):=500;
  6  beGin
  7 ajay_acc_no:=&account_no;
  8 select balance into ajay_balance
  9  from account_07
 10 where acc_no=ajay_acc_no;
 11 ajay_balance:=ajay_balance-ajay_debit_amt;
 12 if ajay_balance>=ajay_min_balance
 13 then update account_07
 14 set balance=ajay_balance
```

```
 15 where acc_no=ajay_acc_no;
 16 else dbms_output.put_line('invalid transaction');
 17 end if;
 18 end;
 19 /
Enter value for account_no: 100
old 7: ajay_acc_no:=&account_no;
new 7: ajay_acc_no:=100;

PL/SQL procedure successfully completed.

SQL> select * from account_07;

ACC_NO   NAME     BALANCE
----------  ---------- ----------
100         ajay       9500
200         akshay     5000
300         akshat     7500
400         ankit      3000
500         aman       6000
```

## TAKE RADIUS AS INPUT FROM THE USER AND CALCULATE THE AREA AND CIRCUMFERENCE OF THE CIRLE AND STORE IT IN A TABLE.

```
SQL> create table circle_07(radius number(5),circum number(10,2),area number(10,2));

Table created.

SQL> declare
  2 ajay_radius number(5);
  3  ajay_cir number(10,2);
  4 ajay_area number(10,2);
  5  beGin
  6  for x in 1..5
  7  loop
 8  ajay_radius:=x;
```

```
 9 ajay_cir:=2*3.14*ajay_radius;
10 ajay_area:=3.14*ajay_radius*ajay_radius;
11 insert into circle_07 values(ajay_radius,ajay_cir,ajay_area);
12 end loop;
13 end;
14 /
PL/SQL procedure successfully completed.

SQL> select * from circle_07;
RADIUS CIRCUM AREA
---------- ---------- ----------
1          6.28       3.14
2          12.56      12.56
3          18.84      28.26
4          25.12      50.24
5          31.4       78.5
```

## DISPLAY THE FACTORIAL OF A NUMBER PROVIDED BY USER.

```
SQL> declare
 2  ajay_num number(10);
 3  ajay_x number(10):=1;
 4 ajay_result number(10):=1;
 5  beGin
 6  ajay_num:=&number;
 7 while ajay_x<=ajay_num
 8  loop
 9 ajay_result:=ajay_result*ajay_x;
10 ajay_x:=ajay_x+1;
11 end loop;
12 dbms_output.put_line('factorial is'||ajay_result);
13 end;
14 /
```

Enter value for number: 5

old 6: ajay_num:=&number;

new 6: ajay_num:=5;

**factorial is 120**

PL/SQL procedure successfully completed.

# LAB 12

## PROCEDURES

PL/SQL procedure is a named block that does a specific task. PL/SQL procedure allows you to encapsulate complex business logic and reuse it in both database layer and application layer.

### Creating a Procedure

A procedure is created with the CREATE OR REPLACE PROCEDURE statement. The simplified syntax for the CREATE OR REPLACE PROCEDURE statement is as follows:

```
CREATE [OR REPLACE] PROCEDURE procedure_name
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
{IS | AS}
BEGIN
  < procedure_body >
END procedure_name;
```

**Where**

- *procedure-name* specifies the name of the procedure.
- [OR REPLACE] option allows modifying an existing procedure.
- The optional parameter list contains name, mode and types of the parameters. IN represents that value will be passed from outside and OUT represents that this parameter will be used to return a value outside of the procedure.
- *procedure-body* contains the executable part.
- The AS keyword is used instead of the IS keyword for creating a standalone procedure.

# FUNCTIONS

A PL/SQL function is same as a procedure except that it returns a value.

## Creating a Function

A standalone function is created using the CREATE FUNCTION statement. The simplified syntax for the CREATE OR REPLACE PROCEDURE statement is as follows:

```
CREATE [OR REPLACE] FUNCTION function_name
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
RETURN return_datatype
{IS | AS}
BEGIN
  < function_body >
END [function_name];
```

Where,

- *function-name* specifies the name of the function.
- [OR REPLACE] option allows modifying an existing function.
- The optional parameter list contains name, mode and types of the parameters. IN represents that value will be passed from outside and OUT represents that this parameter will be used to return a value outside of the procedure.
- The function must contain a return statement.
- *RETURN* clause specifies that data type you are going to return from the function.
- *function-body* contains the executable part.
- The AS keyword is used instead of the IS keyword for creating a standalone function.

# QUERIES

## PROCEDURE FOR CALCULATING FACTORIAL

```
SQL> declare
  2  a number;
  3  b number;
  4  i number;
  5  procedure fact_16(a in out number,b in out number) is
  6  begin
  7  i:=a;
  8  while i>0
  9  loop
 10  b:=b*i;
 11  i:=i-1;
 12  end loop;
 13  dbms_output.put_line('Factorial is '||b);
 14  end;
 15  begin
 16  a:=&no1;
 17  b:=1;
 18  fact_16(a,b);
 19  end;
 20  /
Enter value for no1: 4
old  16: a:=&no1;
new  16: a:=4;
Factorial is 24


PL/SQL procedure successfully completed.
```

## FUNCTION FOR CALCULATING FACTORIAL

```
SQL> declare
  2  a number;
  3  b number;
  4  function fact_16(a in out number,b in out number) return number
  5  is
  6  begin
  7  while a>0
  8  loop
  9  b:=b*a;
 10  a:=a-1;
 11  end loop;
 12  return b;
 13  end;
 14  begin
 15  a:=&no1;
 16  b:=1;
 17  b:=fact_16(a,b);
 18  dbms_output.put_line('Factorial is: '||b);
 19  end;
 20  /
Enter value for no1: 4
old  15: a:=&no1;
new  15: a:=4;
Factorial is: 24


PL/SQL procedure successfully completed.
```

## PROCEDURE FOR WRITING FIBONACCI SERIES

```
SQL> declare
  2  a number;
  3  b number;
  4  c number;
  5  i number;
  6  procedure fibo_16(a in out number,b in out number,c in out number)
  7  is
  8  begin
  9  dbms_output.put_line(b);
 10  dbms_output.put_line(c);
 11  for i in 1..8
 12  loop
 13  a:=b;
 14  b:=c;
 15  c:=a+b;
 16  dbms_output.put_line(c);
 17  end loop;
 18  end;
 19  begin
 20  a:=1;
 21  b:=1;
 22  c:=1;
 23  fibo_16(a,b,c);
 24  end;
 25  /
1
1
2
3
5
```

8

13

21

34

55

PL/SQL procedure successfully completed.

## FUNCTION FOR WRITING FIBONACCI SERIES

```
SQL> declare
  2  a number;
  3  b number;
  4  c number;
  5  i number;
  6  function fibo_16(a in out number,b in out number,c in out number) return number
  7  is
  8  begin
  9  a:=b;
 10  b:=c;
 11  c:=a+b;
 12  return c;
 13  end;
 14  begin
 15  a:=1;
 16  b:=1;
 17  c:=1;
 18  dbms_output.put_line(b);
 19  dbms_output.put_line(c);
 20  for i in 1..8
 21  loop
 22  c:=fibo_16(a,b,c);
 23  dbms_output.put_line(c);
```

```
 24  end loop;
 25  end;
 26  /
1
1
2
3
5
8
13
21
34
55

PL/SQL procedure successfully completed.
```

## PROCEDURE FOR CHECKING WEATHER A NUMBER IS PRIME OR NOT

```
SQL> declare
  2  n number;
  3  i number;
  4  f number;
  5  procedure prime_16(n in out number,f in out number)
  6  is
  7  begin
  8  i:=0;
  9  for x in 2..(n-1)
 10  loop
 11  f:=mod(n,x);
 12  if f=0
 13  then
 14  i:=1;
 15  end if;
 16  end loop;
 17  if i>0
 18  then
 19  dbms_output.put_line('Not Prime');
 20  else
 21  dbms_output.put_line('Prime');
 22  end if;
 23  end;
 24  begin
 25  n:=&no1;
 26  f:=1;
 27  prime_16(n,f);
 28  end;
 29  /
Enter value for no1: 12
```

```
old  25: n:=&no1;

new  25: n:=12;

Not Prime


PL/SQL procedure successfully completed.


SQL> /

Enter value for no1: 13

old  25: n:=&no1;

new  25: n:=13;

Prime


PL/SQL procedure successfully completed.
```

## FUNCTION FOR CHECKING WEATHER A NUMBER IS PRIME OR NOT

```
SQL> declare
  2  n number;
  3  i number;
  4  f number;
  5  function prime_16(n in out number,f in out number) return number
  6  is
  7  begin
  8  i:=0;
  9  for x in 2..(n-1)
 10  loop
 11  f:=mod(n,x);
 12  if f=0
 13  then
 14  i:=1;
 15  end if;
 16  end loop;
 17  return i;
 18  end;
 19  begin
 20  n:=&no1;
 21  f:=1;
 22  i:=prime_16(n,f);
 23  if i>0
 24  then
 25  dbms_output.put_line('Not Prime');
 26  else
 27  dbms_output.put_line('Prime');
 28  end if;
 29  end;
 30  /
```

Enter value for no1: 12

old  20: n:=&no1;

new  20: n:=12;

Not Prime

PL/SQL procedure successfully completed.

SQL> /

Enter value for no1: 13

old  20: n:=&no1;

new  20: n:=13;

Prime

PL/SQL procedure successfully completed.

## PROCEDURE FOR REVERSING A GIVEN NUMBER

```
SQL> declare
  2  a number;
  3  r number;
  4  l number;
  5  procedure rev_16(a in out number,r in out number,l in out number)
  6  is
  7  begin
  8  l:=length(a);
  9  while l>0
 10  loop
 11  r:=r||substr(a,l,1);
 12  l:=l-1;
 13  end loop;
 14  dbms_output.put_line(r);
 15  end;
 16  begin
 17  a:=&no;
 18  l:=1;
 19  rev_16(a,r,l);
 20  end;
 21  /
Enter value for no: 12345
old  17: a:=&no;
new  17: a:=12345;
54321


PL/SQL procedure successfully completed.
```

## FUNCTION FOR REVERSING A GIVEN NUMBER

```
SQL> declare
  2  a number;
  3  r number;
  4  l number;
  5  function rev_16(a in out number,r in out number,l in out number) return number
  6  is
  7  begin
  8  l:=length(a);
  9  while l>0
 10  loop
 11  r:=r||substr(a,l,1);
 12  l:=l-1;
 13  end loop;
 14  return r;
 15  end;
 16  begin
 17  a:=&no;
 18  l:=1;
 19  r:=rev_16(a,r,l);
 20  dbms_output.put_line(r);
 21  end;
 22  /
Enter value for no: 12345
old  17: a:=&no;
new  17: a:=12345;
54321


PL/SQL procedure successfully completed.
```

## PROCEDURE FOR CHECKING A STRING TO BE A PALINDROME OR NOT

```
SQL> declare
  2  a number;
  3  r number;
  4  l number;
  5  procedure rev_16(a in out number,r in out number,l in out number)
  6  is
  7  begin
  8  l:=length(a);
  9  while l>0
 10  loop
 11  r:=r||substr(a,l,1);
 12  l:=l-1;
 13  end loop;
 14  if r=a
 15  then
 16  dbms_output.put_line('Palindrome');
 17  else
 18  dbms_output.put_line('Not Palindrome');
 19  end if;
 20  end;
 21  begin
 22  a:=&no;
 23  l:=1;
 24  rev_16(a,r,l);
 25  end;
 26  /
Enter value for no: 12345
old  22: a:=&no;
new  22: a:=12345;
Not Palindrome
```

PL/SQL procedure successfully completed.

SQL> /

Enter value for no: 12321

old  22: a:=&no;

new  22: a:=12321;

Palindrome

PL/SQL procedure successfully completed.

## FUNCTION FOR CHECKING A STRING TO BE A PALINDROME OR NOT

```
SQL> declare
  2  a number;
  3  r number;
  4  l number;
  5  function rev_16(a in out number,r in out number,l in out number) return number
  6  is
  7  begin
  8  l:=length(a);
  9  while l>0
 10  loop
 11  r:=r||substr(a,l,1);
 12  l:=l-1;
 13  end loop;
 14  return r;
 15  end;
 16  begin
 17  a:=&no;
 18  l:=1;
 19  r:=rev_16(a,r,l);
 20  if r=a
 21  then
 22  dbms_output.put_line('Palindrome');
 23  else
 24  dbms_output.put_line('Not Palindrome');
 25  end if;
 26  end;
 27  /
Enter value for no: 12345
old  17: a:=&no;
new  17: a:=12345;
```

Not Palindrome

PL/SQL procedure successfully completed.

SQL> /

Enter value for no: 12321

old  17: a:=&no;

new  17: a:=12321;

Palindrome

PL/SQL procedure successfully completed.

# LAB 13

## CURSORS

Oracle creates a memory area, known as context area, for processing an SQL statement, which contains all information needed for processing the statement, for example, number of rows processed, etc.

A cursor is a pointer to this context area. PL/SQL controls the context area through a cursor. A cursor holds the rows (one or more) returned by a SQL statement. The set of rows the cursor holds is referred to as the active set.

You can name a cursor so that it could be referred to in a program to fetch and process the rows returned by the SQL statement, one at a time. There are two types of cursors:

- Implicit cursors
- Explicit cursors

### Implicit Cursors

Implicit cursors are automatically created by Oracle whenever an SQL statement is executed, when there is no explicit cursor for the statement. Programmers cannot control the implicit cursors and the information in it.

Whenever a DML statement (INSERT, UPDATE and DELETE) is issued, an implicit cursor is associated with this statement. For INSERT operations, the cursor holds the data that needs to be inserted. For UPDATE and DELETE operations, the cursor identifies the rows that would be affected.

In PL/SQL, you can refer to the most recent implicit cursor as the SQL cursor, which always has the attributes like %FOUND, %ISOPEN, %NOTFOUND, and %ROWCOUNT. The SQL cursor has additional attributes, %BULK_ROWCOUNT and %BULK_EXCEPTIONS, designed for use with the FORALL statement. The following table provides the description of the most used attributes:

| Attribute | Description |
|---|---|
| %FOUND | Returns TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows or a SELECT INTO statement returned one or more rows. Otherwise, it returns FALSE. |
| %NOTFOUND | The logical opposite of %FOUND. It returns TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, it returns FALSE. |
| %ISOPEN | Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement. |
| %ROWCOUNT | Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement. |

## Explicit Cursors

Explicit cursors are programmer defined cursors for gaining more control over the context area. An explicit cursor should be defined in the declaration section of the PL/SQL Block. It is created on a SELECT Statement which returns more than one row.

The syntax for creating an explicit cursor is :

CURSOR cursor_name IS select_statement;

Working with an explicit cursor involves four steps:

- Declaring the cursor for initializing in the memory
- Opening the cursor for allocating memory
- Fetching the cursor for retrieving data
- Closing the cursor to release allocated memory

### Declaring the Cursor

Declaring the cursor defines the cursor with a name and the associated SELECT statement. For example:

```
CURSOR c_customers IS
  SELECT id, name, address FROM customers;
```

## Opening the Cursor

Opening the cursor allocates memory for the cursor and makes it ready for fetching the rows returned by the SQL statement into it. For example, we will open above-defined cursor as follows:

```
OPEN c_customers;
```

## Fetching the Cursor

Fetching the cursor involves accessing one row at a time. For example we will fetch rows from the above-opened cursor as follows:

```
FETCH c_customers INTO c_id, c_name, c_addr;
```

## Closing the Cursor

Closing the cursor means releasing the allocated memory. For example, we will close above-opened cursor as follows:

```
CLOSE c_customers;
```

# QUERIES

## TABLE USED

SQL> select * from emp0_16;

ENO ENAME DNO SALARY JOB

----- ---------- ----- ---------- ----------

e1 ankit d1 2000 manager

e2 aman d2 1500 tl

e3 ajay d1 500 pl

e4 abhinav d1 700 analyst

e5 amit d2 1800 programmer

e6 akshat d3 1000 manager

e7 akshay d3 2000 pl

7 rows selected.

# DELETING THE RECORD OF THE EMPLOYEE IF FOUND AND DISPLAY THE NUMBER OF RECORDS DELETED

```
SQL> declare
  2  empno varchar(10);
  3  begin
  4  empno:='&a';
  5  delete from emp0_16 where eno=empno;
  6  if sql%found
  7  then
  8  dbms_output.put_line('No. of data deleted= '||sql%rowcount);
  9  else
 10  dbms_output.put_line('No rows found');
 11  end if;
 12  end;
 13  /
Enter value for a: e9
old   4: empno:='&a';
new   4: empno:='e9';
No rows found

PL/SQL procedure successfully completed.

SQL> /
Enter value for a: e7
old   4: empno:='&a';
new   4: empno:='e7';
No. of data deleted= 1


PL/SQL procedure successfully completed.
```

# UPDATING THE RECORD IN THE TABLE AND DISPLAYING NUMBER OF RECORDS UPDATED

```
SQL> declare
  2  empno varchar(10);
  3  begin
  4  empno:='&a';
  5  update emp0_16 set salary=1700 where eno=empno;
  6  dbms_output.put_line('No. of data updated= '||sql%rowcount);
  7  end;
  8  /
Enter value for a: e6
old   4: empno:='&a';
new   4: empno:='e6';
No. of data updated= 1

PL/SQL procedure successfully completed.

SQL> select * from emp0_16;
ENO   ENAME          DNO       SALARY JOB
----- ---------------- ----- ---------- ----------
e1    ankit          d1         2000 manager
e2    aman           d2          1500 tl
e3    ajay           d1          500 pl
e4    abhinav        d1           700 analyst
e5    amit           d2         1800 programmer
e6    akshat         d3         1700 manager

6 rows selected.
```

## DISPLAYING THE EMPLOYEE NUMBER AND NUMBER OF ROWS SELECTED HAVING SALARY EQUAL TO 1700

```
SQL> declare
  2  empno varchar(10);
  3  begin
  4  select eno into empno from emp0_16 where salary=1700;
  5  dbms_output.put_line('Eno selected ='||empno);
  6  dbms_output.put_line('No of rows selected'||sql%rowcount);
  7  end;
  8  /
Eno selected =e6
No of rows selected1


PL/SQL procedure successfully completed.
```

## INSERTING A RECORD IN THE TABLE AND DISPLAY THE ROWCOUNT

```
SQL> declare
  2  eno16 varchar(10):='e7';
  3  ename16 varchar(10):='Akshay';
  4  dno16 varchar(10):='d3';
  5  salary number:=2000;
  6  job16 varchar(10):='pl';
  7  begin
  8  insert into emp0_16 values(eno16,ename16,dno16,salary,job16);
  9  dbms_output.put_line('No of rows selected'||sql%rowcount);
 10  end;
 11  /
No of rows selected1


PL/SQL procedure successfully completed.
```

```
SQL> select * from emp0_16;

ENO   ENAME          DNO      SALARY JOB
----- --------------- ----- ---------- ----------
e1    ankit          d1        2000 manager
e2    aman           d2         1500 tl
e3    ajay           d1         500 pl
e4    abhinav        d1          700 analyst
e5    amit           d2        1800 programmer
e6    akshat         d3         1700 manager
e7    Akshay         d3         2000 pl

7 rows selected.
```

## CHECKING WEATHER THE DEFAULT CURSOR IS OPEN OR NOT

```
SQL> declare
  2  begin
  3  update emp0_16 set salary=1600 where eno='e6';
  4  if sql%isopen
  5  then
  6  dbms_output.put_line('cursor open');
  7  else
  8  dbms_output.put_line('cursor not open');
  9  end if;
 10  end;
 11  /
cursor not open

PL/SQL procedure successfully completed.
```

# CREATING A CURSOR AND DISPLAYING THE RECORDS OF THE TABLE USING IT

```
SQL> declare
  2  cursor c1 is select eno,ename from emp0_16 where dno='d1';
  3  rec c1%rowtype;
  4  begin
  5  open c1;
  6  loop
  7  fetch c1 into rec;
  8  dbms_output.put_line(rec.eno||rec.ename);
  9  exit when c1%notfound;
 10  end loop;
 11  close c1;
 12  end;
 13  /
e1ankit
e3ajay
e4abhinav
e4abhinav


PL/SQL procedure successfully completed.
```

# DISPLAYING THE TOP FIVE HIGHEST PAID EMPLOYEES

```
SQL> declare
  2  cursor c2 is select eno,ename from emp0_16 order by salary desc;
  3  rec c2%rowtype;
  4  begin
  5  open c2;
  6  loop
  7  fetch c2 into rec;
  8  dbms_output.put_line(rec.eno||' '||rec.ename);
  9  exit when c2%rowcount>4 or c2%notfound;
 10  end loop;
 11  close c2;
 12  end;
 13  /
e1  ankit
e7  Akshay
e5  amit
e6  akshat
e2  aman

PL/SQL procedure successfully completed.
```

# INCREMENT THE SALARIES OF THE EMPLOYEES ACCORDING TO THEIR JOB

```
SQL>declare
  2 cursor c2 is select eno,salary,job from emp0_16;
  3 rec c2%rowtype;
  4 begin
  5 open c2;
  6 loop
  7 fetch c2 into rec;
  8 if rec.job='manager'
  9 then
 10 update emp0_16 set salary=rec.salary+0.2*rec.salary where eno=rec.eno;
 11 else if rec.job='tl'
 12 then
 13 update emp0_16 set salary=rec.salary+0.15*rec.salary where eno=rec.eno;
 14 else
 15 update emp0_16 set salary=rec.salary+0.10*rec.salary where eno=rec.eno;
 16 end if;
 17 end if;
 18 exit when c2%notfound;
 19 end loop;
 20 close c2;
 21 end;
 22 /
```

# LAB 14

## PACKAGES

PL/SQL packages are schema objects that groups logically related PL/SQL types, variables and subprograms.
A package will have two mandatory parts:

- Package specification
- Package body or definition

### Package Specification

The specification is the interface to the package. It just DECLARES the types, variables, constants, exceptions, cursors, and subprograms that can be referenced from outside the package. In other words, it contains all information about the content of the package, but excludes the code for the subprograms.

All objects placed in the specification are called public objects. Any subprogram not in the package specification but coded in the package body is called a private object.

The following code snippet shows a package specification having a single procedure. You can have many global variables defined and multiple procedures or functions inside a package.

```
CREATE PACKAGE cust_sal AS
   PROCEDURE find_sal(c_id customers.id%type);
END cust_sal;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
Package created.
```

### Package Body

The package body has the codes for various methods declared in the package specification and other private declarations, which are hidden from code outside the package.

The CREATE PACKAGE BODY Statement is used for creating the package body. The following code snippet shows the package body declaration for the *cust_sal* package created above.

```
CREATE OR REPLACE PACKAGE BODY cust_sal AS
  PROCEDURE find_sal(c_id customers.id%TYPE) IS
  c_sal customers.salary%TYPE;
  BEGIN
    SELECT salary INTO c_sal
    FROM customers
    WHERE id = c_id;
    dbms_output.put_line('Salary: '|| c_sal);
  END find_sal;
END cust_sal;
/
```

When the above code is executed at SQL prompt, it produces the following result:

Package body created.

## Using the Package Elements

The package elements (variables, procedures or functions) are accessed with the following syntax:

package_name.element_name;

# EXCEPTION HANDLING

An error condition during a program execution is called an exception in PL/SQL. PL/SQL supports programmers to catch such conditions using EXCEPTION block in the program and an appropriate action is taken against the error condition. There are two types of exceptions:

- System-defined exceptions
- User-defined exceptions

## Syntax for Exception Handling

The General Syntax for exception handling is as follows. Here you can list down as many as exceptions you want to handle. The default exception will be handled using *WHEN others THEN*:

```
DECLARE
  <declarations section>
BEGIN
  <executable command(s)>
EXCEPTION
  <exception handling goes here >
  WHEN exception1 THEN
     exception1-handling-statements
  WHEN exception2  THEN
    exception2-handling-statements
  WHEN exception3 THEN
    exception3-handling-statements
  ........
  WHEN others THEN
    exception3-handling-statements
END;
```

## Raising Exceptions

Exceptions are raised by the database server automatically whenever there is any internal database error, but exceptions can be raised explicitly by the programmer by using the command RAISE. Following is the simple syntax of raising an exception:

```
DECLARE
  exception_name EXCEPTION;
BEGIN
  IF condition THEN
```

```
    RAISE exception_name;
  END IF;
EXCEPTION
  WHEN exception_name THEN
  statement;
END;
```

You can use above syntax in raising Oracle standard exception or any user-defined exception. Next section will give you an example on raising user-defined exception, similar way you can raise Oracle standard exceptions as well.

## User-defined Exceptions

PL/SQL allows you to define your own exceptions according to the need of your program. A user-defined exception must be declared and then raised explicitly, using either a RAISE statement or the procedure DBMS_STANDARD.RAISE_APPLICATION_ERROR.

The syntax for declaring an exception is:

```
DECLARE
  my-exception EXCEPTION;
```

## Pre-defined Exceptions

PL/SQL provides many pre-defined exceptions, which are executed when any database rule is violated by a program. For example, the predefined exception NO_DATA_FOUND is raised when a SELECT INTO statement returns no rows. The following table lists few of the important pre-defined exceptions:

| Exception | Oracle Error | SQLCODE | Description |
|---|---|---|---|
| ACCESS_INTO_NULL | 06530 | -6530 | It is raised when a null object is automatically assigned a value. |
| CASE_NOT_FOUND | 06592 | -6592 | It is raised when none of the choices in the WHEN clauses of a CASE statement is selected, and there is no ELSE clause. |
| COLLECTION_IS_NULL | 06531 | -6531 | It is raised when a program attempts to apply collection methods other than EXISTS to an uninitialized nested table or varray, or the program attempts to assign values to the elements of an uninitialized nested table or varray. |
| DUP_VAL_ON_INDEX | 00001 | -1 | It is raised when duplicate values are attempted to |

| | | | |
|---|---|---|---|
| | | | be stored in a column with unique index. |
| INVALID_CURSOR | 01001 | -1001 | It is raised when attempts are made to make a cursor operation that is not allowed, such as closing an unopened cursor. |
| INVALID_NUMBER | 01722 | -1722 | It is raised when the conversion of a character string into a number fails because the string does not represent a valid number. |
| LOGIN_DENIED | 01017 | -1017 | It is raised when s program attempts to log on to the database with an invalid username or password. |
| NO_DATA_FOUND | 01403 | +100 | It is raised when a SELECT INTO statement returns no rows. |
| NOT_LOGGED_ON | 01012 | -1012 | It is raised when a database call is issued without being connected to the database. |
| PROGRAM_ERROR | 06501 | -6501 | It is raised when PL/SQL has an internal problem. |
| ROWTYPE_MISMATCH | 06504 | -6504 | It is raised when a cursor fetches value in a variable having incompatible data type. |
| SELF_IS_NULL | 30625 | -30625 | It is raised when a member method is invoked, but the instance of the object type was not initialized. |
| STORAGE_ERROR | 06500 | -6500 | It is raised when PL/SQL ran out of memory or memory was corrupted. |
| TOO_MANY_ROWS | 01422 | -1422 | It is raised when s SELECT INTO statement returns more than one row. |
| VALUE_ERROR | 06502 | -6502 | It is raised when an arithmetic, conversion, truncation, or size-constraint error occurs. |
| ZERO_DIVIDE | 01476 | 1476 | It is raised when an attempt is made to divide a number by zero. |

# QUERIES

## CREATING A TABLE CIRCLE :

```
SQL> create table circle0_16(radius number(5),area number(5,4),circum number(5,2));

Table created.
```

## CREATING A PACKAGE TO CALCULATE THE AREA AND CIRCUMFERENCE OF THE PROVIDED RADIUS

### PACKAGE SPECIFICATION :

```
SQL> create package circle_e_016
  2  as
  3  function area(radius1 in number,a in out number) return number;
  4  procedure circum(radius1 in number,a in out number);
  5  end;
  6  /

Package created.
```

### PACKAGE BODY :

```
SQL> create package body circle_e_016
  2  as
  3  function area(radius1 in number,a in out number) return number
  4  is
  5  begin
  6  a:=radius1*radius1*3.14;
  7  return a;
  8  end area;
  9  procedure circum(radius1 in number,a in out number)
 10  is
 11  c number(5,2);
 12  begin
 13  c:=radius1*2*3.14;
 14  insert into circle0_16 values(radius1,a,c);
 15  end circum;
 16  end circle_e_016;
```

```
 17  /

Package body created.

SQL> alter table circle0_16 modify area number(5,2);

Table altered.

SQL> declare
  2  r number(5);
  3  a number(5,2);
  4  begin
  5  r:=&radius;
  6  a:=circle_e_016.area(r,a);
  7  circle_e_016.circum(r,a);
  8  end;
  9  /
Enter value for radius: 2
old   5: r:=&radius;
new   5: r:=2;

PL/SQL procedure successfully completed.

SQL> /
Enter value for radius: 4
old   5: r:=&radius;
new   5: r:=4;

PL/SQL procedure successfully completed.

SQL> create sequence s16
  2  minvalue 1
  3  start with 1
  4  increment by 1;

Sequence created.
```

# CREATING A PACKAGE FOR EMPLOYEE MANAGEMENT

## PACKAGE SPECIFICATION :

```
SQL> create package emp_0
  2  as
  3  function hire(ename emp0_1_6.ename%type,sal emp0_1_6.sal%type,deptno
emp0_1_6.deptno%type) return number;
  4  procedure sal_inc(empid2 emp0_1_6.empid%type);
  5  end;
  6  /

Package created.
```

## PACKAGE BODY :

```
SQL> create package body emp_0
  2  as
  3  function hire(ename emp0_1_6.ename%type,sal emp0_1_6.sal%type,deptno emp0_1
_6.deptno%type) return number
  4  is
  5  begin
  6  insert into emp0_1_6 values(s16.nextval,ename,sal,deptno);
  7  return s16.currval;
  8  end hire;
  9  procedure sal_inc(empid2 emp0_1_6.empid%type)
 10  is
 11  begin
 12  update emp0_1_6 set sal=sal+sal where empid=empid2;
 13  end sal_inc;
 14  end emp_0;
 15  /

Package body created.
```

```
SQL> declare
  2  ename emp0_1_6.ename%type;
  3  sal emp0_1_6.sal%type;
  4  deptno emp0_1_6.deptno%type;
  5  empi emp0_1_6.empid%type;
  6  begin
  7  ename:='&name';
  8  sal:=&sal;
  9  deptno:=&deptno;
 10  empi:=emp_0.hire(ename,sal,deptno);
 11  emp_0.sal_inc(empi);
 12  end;
 13  /
Enter value for name: ajay
old   7: ename:='&name';
new   7: ename:='ajay';
Enter value for sal: 500
old   8: sal:=&sal;
new   8: sal:=500;
Enter value for deptno: 5
old   9: deptno:=&deptno;
new   9: deptno:=5;
```

## CHECKING THE OUTPUT IN TABLE

```
SQL> select * from emp0_1_6;

EMPID   ENAME  SAL     DEPTNO
---------- ----------  -------- ----------
1          ajay       500      5
```

# LAB 15

## TRIGGERS

Triggers are stored programs, which are automatically executed or fired when some events occur. Triggers are, in fact, written to be executed in response to any of the following events:

- A database manipulation (DML) statement (DELETE, INSERT, or UPDATE).
- A database definition (DDL) statement (CREATE, ALTER, or DROP).
- A database operation (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).

Triggers could be defined on the table, view, schema, or database with which the event is associated.

### Benefits of Triggers

Triggers can be written for the following purposes:

- Generating some derived column values automatically
- Enforcing referential integrity
- Event logging and storing information on table access
- Auditing
- Synchronous replication of tables
- Imposing security authorizations
- Preventing invalid transactions

### Creating Triggers

The syntax for creating a trigger is:

```
CREATE [OR REPLACE ] TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF }
{INSERT [OR] | UPDATE [OR] | DELETE}
[OF col_name]
ON table_name
[REFERENCING OLD AS o NEW AS n]
[FOR EACH ROW]
WHEN (condition)
DECLARE
  Declaration-statements
```

```
BEGIN
  Executable-statements
EXCEPTION
  Exception-handling-statements
END;
```

**Where**

- CREATE [OR REPLACE] TRIGGER trigger_name: Creates or replaces an existing trigger with the *trigger_name*.
- {BEFORE | AFTER | INSTEAD OF} : This specifies when the trigger would be executed. The INSTEAD OF clause is used for creating trigger on a view.
- {INSERT [OR] | UPDATE [OR] | DELETE}: This specifies the DML operation.
- [OF col_name]: This specifies the column name that would be updated.
- [ON table_name]: This specifies the name of the table associated with the trigger.
- [REFERENCING OLD AS o NEW AS n]: This allows you to refer new and old values for various DML statements, like INSERT, UPDATE, and DELETE.
- [FOR EACH ROW]: This specifies a row level trigger, i.e., the trigger would be executed for each row being affected. Otherwise the trigger will execute just once when the SQL statement is executed, which is called a table level trigger.
- WHEN (condition): This provides a condition for rows for which the trigger would fire. This clause is valid only for row level triggers.

# QUERIES

## CREATING A SEQUENCE

```
SQL> create sequence s016
  2  minvalue 1
  3  start with 1
  4  increment by 1;

Sequence created.
```

## CREATING A TABLE

```
SQL> create table auditor_016(eid number(5),operation varchar(15));
Table created.
```

## CREATING TRIGGER WHICH WILL GET TRIGGERED ON INSERT, UPDATE AND DELETE ON EMP_DBMS

```
SQL>  create trigger t_0816
  2  after insert or update or delete ON emp_dbms
  3  for each row
  4  declare
  5  begin
  6  if inserting
  7  then
  8  insert into auditor_016 values(s016.nextval,'INSERTION');
  9  else if updating
 10  then
 11  insert into auditor_016 values(s016.nextval,'UPDATING');
 12  else if deleting
 13  then
 14  insert into auditor_016 values(s016.nextval,'DELETING');
 15  end if;
 16  end if;
 17  end if;
 18  end;
 19  /
Trigger created.
```

## CREATING TRIGGER WHICH WILL GET TRIGGERED ON INSERT, UPDATE ON EMP0_1_6

```
SQL> create trigger t_0
  2  after insert or update ON emp0_1_6
  3  for each row
  4  declare
  5  begin
  6  if INSERTING
  7  then
  8  insert into auditor_016 values(s016.nextval,'Insert');
  9  else if UPDATING
 10  then
 11  insert into auditor_016 values(s016.nextval,'Update');
 12  end if;
 13  end if;
 14  end;
 15  /

Trigger created.
```