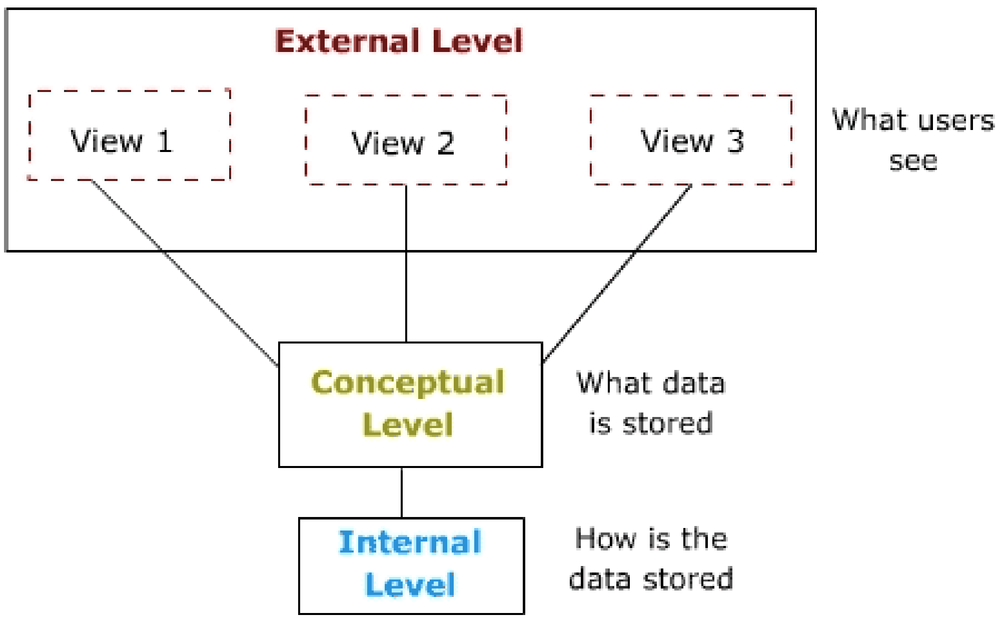
Database Management System and its types

A database management system (**DBMS**) is a computer software application that interacts with the user, other applications, and the database itself to capture and analyze data. It is a collection of interrelated data and a set of programs to access those data. The collection of data is usually referred to as the **database**.
The primary goal of a DBMS is to provide a way to store and retrieve database information that is both convenient and efficient.
Database systems are designed to manage large amount of information. Management of data involves both defining structures for storage of information and providing mechanisms for the manipulation of information. In addition, the database system also ensures the safety of the information stored, despite system crashes or attempts at unauthorized access. A general-purpose **DBMS** is designed to allow the definition, creation, querying, update, and administration of databases.



3-Level Database System Architecture

Types of Data Base Management System

1. Data Model *Classification*

I. Hierarchical data model

It is represented by an upside down tree where the origin of the database is known as the roots .the root acts as a parent to the nodes beneath it .the subsequent nodes that are the child of the root act as parents to the next level of nodes. The last node is called the leaf. It consisted of a one to many relationships.

Advantages

- Simplicity
- Data Independence
- Data Sharing
- Available Expertise

Disadvantages

- Lack of standards
- Lack of structural independence
- Application programming complexity

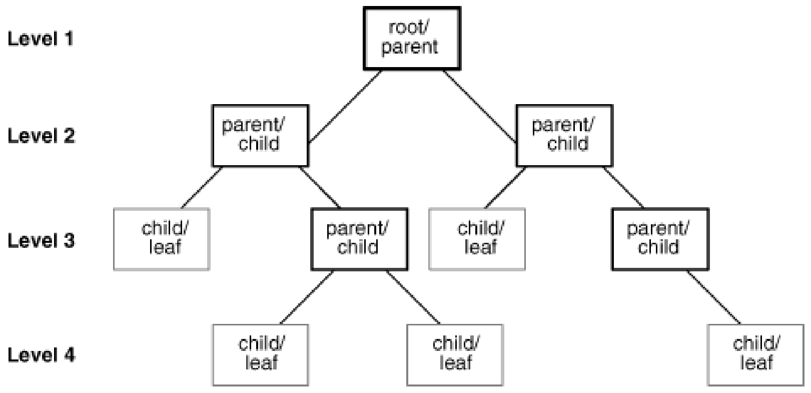


Fig 1.1 a diagram representing hierarchical data model

II. Network data model

A network data model is similar to hierarchical data model except that a record can have multiple parents. Therefore it allows many to many relationship.

Advantages

- Simplicity
- Data Independence
- Standardized as compared to hierarchical
- Many to many relationship possible

Disadvantages

- Complexity
- Lack of structural independence
- Not user friendly

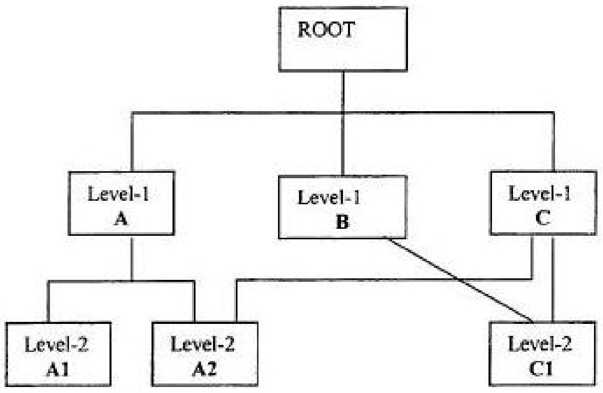


Fig 1.2 Network Database Model Diagram

III. Relational data model

A relational data model is a collection of tables to represent data. Where the tables are known as relations and they consist of multiple rows and columns known as tupples and fields respectively. Number of rows is known as the cardinality whereas number of columns is called degree.

They are the most widely used database management models. Examples are Oracle, My-SQL etc.

Advantages

- Simplicity
- Data Independence
- Standardized
- East to design ,implement and maintain
- Flexible
- Structurally independent

Disadvantages

- Hardware overheads(need more powerful hardware that slows the speed)
- Information Islands causing isolation and hampering reusability
- Bad design due to easy to design capability

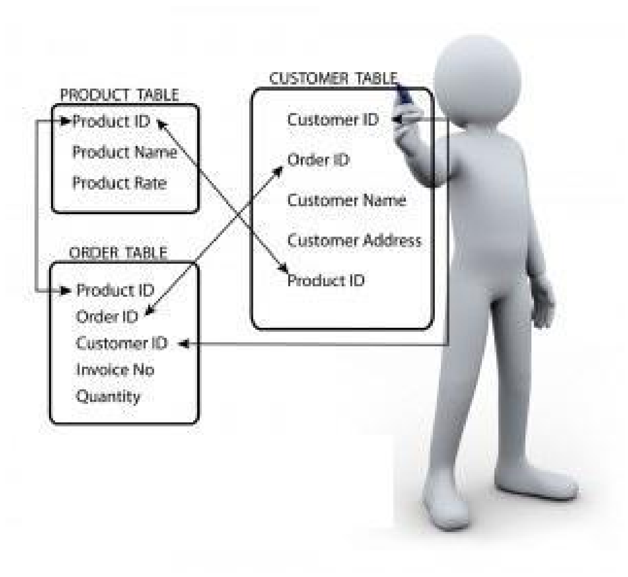


Fig 1.3 relational data model consisting of tables

IV. Entity-Relationship data model

It is a logical and semantic data model. It is a collection of entities of similar type that make up the entity set. Entities can be concrete or abstract and need not be always disjoint. Each entity has some associated descriptive properties called attributes.

Advantages

- Graphical representation for better understanding
- Easy conversion from E-R to other data model.

Disadvantages

- Popular for high level design
- No industry standard for notation

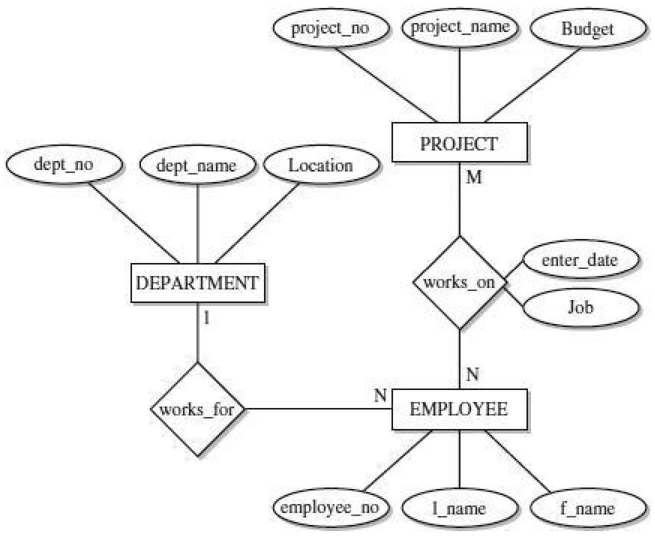


Fig 1.4 E-R model representation

V. Object oriented data model

It captures the semantics of objects that are supported by object oriented programming. Many concepts of oops like data encapsulation, inheritance, data abstraction are also incorporated in the object oriented data model. When OODBMS and RDBMS are combined together it is known as ORDBMS (object relational database management system.)

Advantages

- Can handle a variety of data types
- Improved productivity
- Concepts of oops with database technology

Disadvantages

- Difficult to maintain
- No precise definition

2. On basis of site location

Centralized database system

The management system and the data are controlled centrally from any one or central site since it is physically confined to a single location. Updating, backup, query were easy to accomplish.

Parallel database system

It consists of multiple processing units that improve processing. Data storage disks are present in parallel which increase input /output speed. Querying very large databases becomes easy.

Client/ Server database system

Consists of two logical components that are server and client. Where server (backend) contains the dbms software and client (front end) contains the application and tools of dbms that make requests.

Distributed Database system

Data in this system is spread across variety of different databases that are managed by different dbms running at geographically distinct locations. Has better efficiency and performance.

3. On basis of number of users

- **Single user DBMS**
- **Multiple user DBMS**

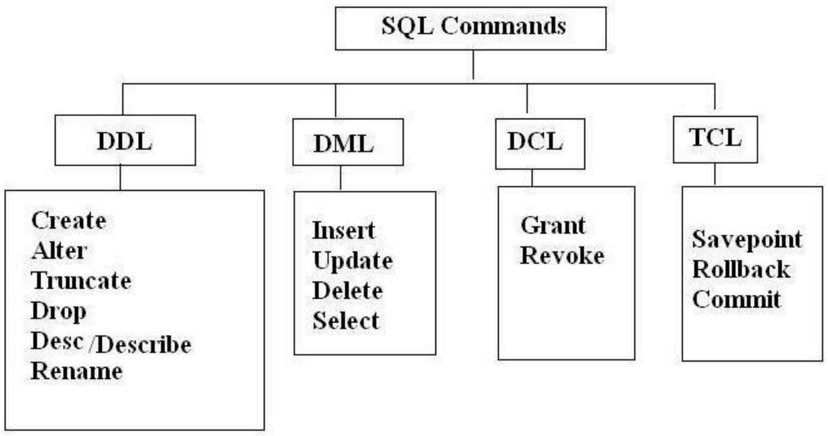
4. On basis of type and extent of use

- **Transactional or production DBMS**
- **Data Warehouse /Decision support**

Structured query language and its command types

SQL stands for structured query language. It is a standard command set that is used to communicate with rdbms. It is a fourth generation language. Although we refer to the SQL language as a “query language,” it can do much more than just query a database. It can define the structure of the data, modify data in the database, and specify security constraints. SQL has clearly established itself as the standard relational database language. It is a format free language as well as case insensitive.

Types of SQL commands



1. DDL(Data Definition Language)

It contains commands for defining relation schemas, deleting and modifying relations. The commands are:

CREATE
ALTER
DROP
RENAME
TRUNCATE
DESCRIBE

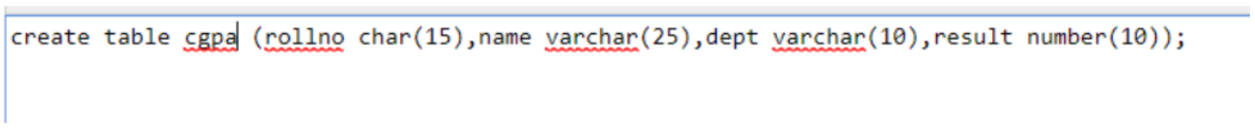
CREATE

I. Create command is also used to create a table. We can specify names and data types of various columns along.

SYNTAX:

```
CREATE TABLE table-name
{
 Column-name1 datatype1 (size),
 Column-name2 datatype2 (size),
};
```

EXAMPLE:



```
create table cgpa (rollno char(15),name varchar(25),dept varchar(10),result number(10));
```

RESULT:

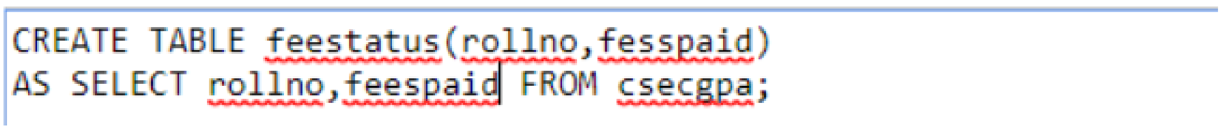
```
Table created.
```

0.02 seconds

II. **In order to create tables from an existing table use the following syntax:**

```
CREATE TABLE table-name (Column-name1, Column-name2);
AS SELECT Column-name1, Column-name2
FROM table-name_existing
```

EXAMPLE:



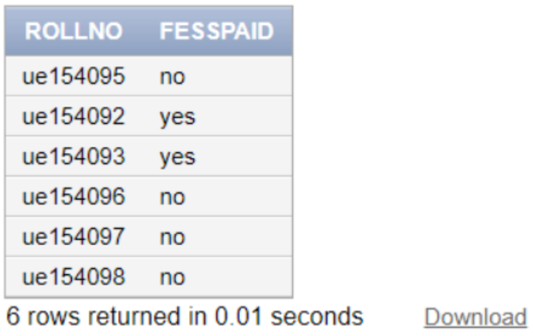
```
CREATE TABLE feestatus(rollno,fesspaid)
AS SELECT rollno,feespaid FROM csecgpa;
```

RESULT:

```
Table created.
```

0.02 seconds

SELECT * FROM feestatus;



ROLLNO	FESSPAID
ue154095	no
ue154092	yes
ue154093	yes
ue154096	no
ue154097	no
ue154098	no

6 rows returned in 0.01 seconds   Download

NOTE: In case the data types are same but the selected attribute is diff from the specified attribute then the specified attribute must be selected from the table .this is not true in case when data types are different.

ALTER

Alter command is used for alteration of table structures. There are various uses of alter command, such as,

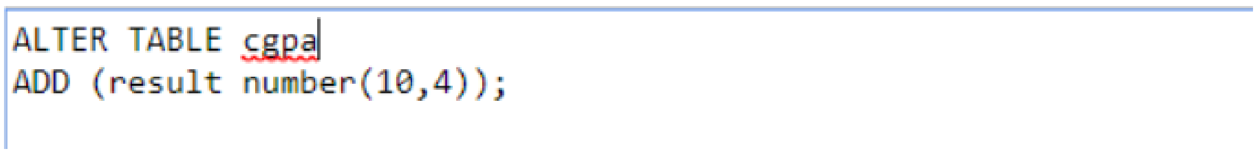
1. To Add Column to existing Table

Using alter command we can add a column to an existing table.

SYNTAX:

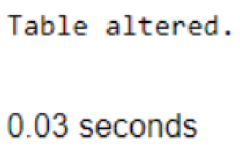
```
ALTER TABLE table-name
ADD (column-name data -type);
```

EXAMPLE:



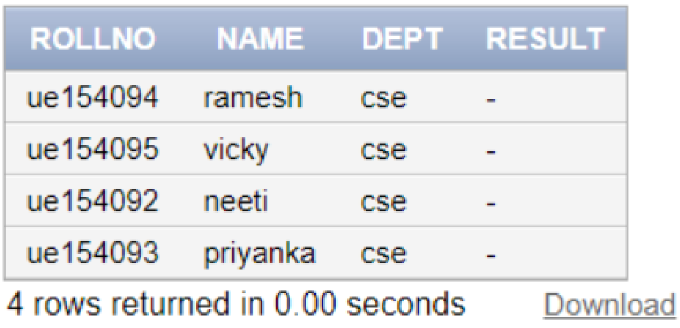
```
ALTER TABLE cgpa
ADD (result number(10,4));
```

RESULT:



```
Table altered.

0.03 seconds
```

SELECT * FROM cgpa;



ROLLNO	NAME	DEPT	RESULT
ue154094	ramesh	cse	-
ue154095	vicky	cse	-
ue154092	neeti	cse	-
ue154093	priyanka	cse	-

4 rows returned in 0.00 seconds   Download

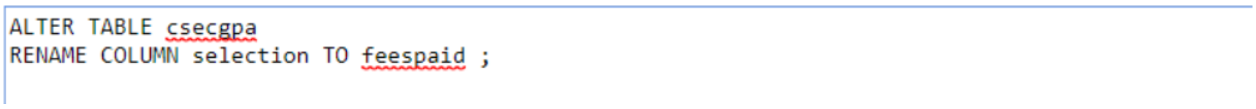
2. To Rename a column

Using alter command you can rename an existing column.

SYNTAX:

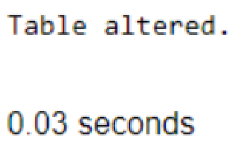
```
ALTER TABLE table-name
RENAME old-column-name to column-name;
```

EXAMPLE:



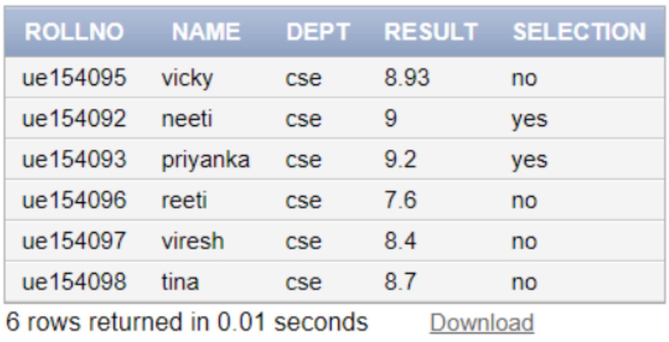
```
ALTER TABLE csecgpa
RENAME COLUMN selection TO feespaid ;
```

RESULT:



```
Table altered.

0.03 seconds
```

SELECT * FROM csecgpa;



ROLLNO	NAME	DEPT	RESULT	SELECTION
ue154095	vicky	cse	8.93	no
ue154092	neeti	cse	9	yes
ue154093	priyanka	cse	9.2	yes
ue154096	reeti	cse	7.6	no
ue154097	viresh	cse	8.4	no
ue154098	tina	cse	8.7	no

6 rows returned in 0.01 seconds   Download

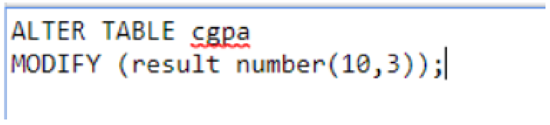
3. To Modify an existing Column

Alter command is used to modify data type of an existing column.

SYNTAX:

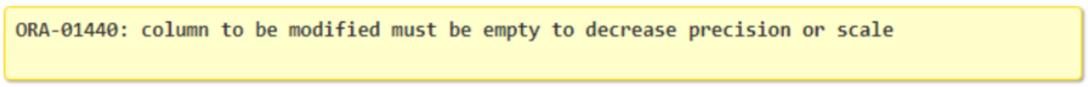
```
ALTER TABLE table-name
MODIFY (column-name data -type);
```

EXAMPLE:



```
ALTER TABLE cgpa
MODIFY (result number(10,3));
```

NOTE:



```
ORA-01440: column to be modified must be empty to decrease precision or scale
```

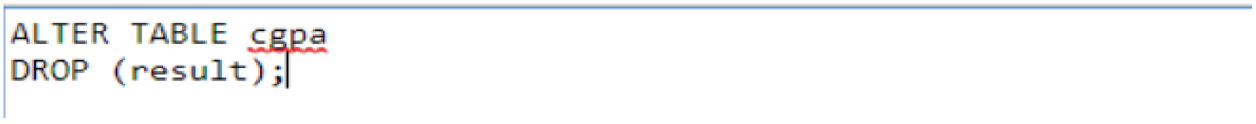
4. **To Drop a Column**

Alter command is also used to drop columns also. Following is the Syntax,

SYNTAX:

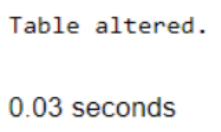
```
ALTER TABLE table-name
DROP (column-name);
```

EXAMPLE:



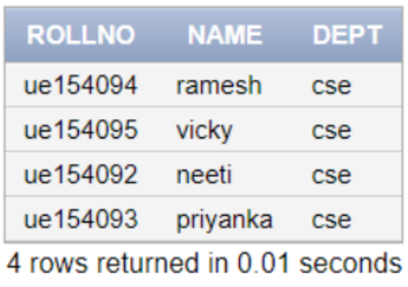
```
ALTER TABLE cgpa
DROP (result);
```

RESULT:



```
Table altered.

0.03 seconds
```

SELECT * FROM cgpa;



ROLLNO	NAME	DEPT
ue154094	ramesh	cse
ue154095	vicky	cse
ue154092	neeti	cse
ue154093	priyanka	cse

4 rows returned in 0.01 seconds

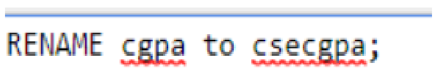
RENAME

Used to change the name of table

SYNTAX:

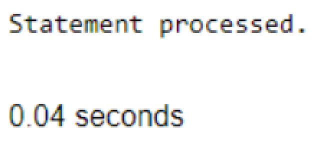
```
RENAME table_name TO new_table_name;
```

EXAMPLE:



```
RENAME cgpa to csecgpa;
```

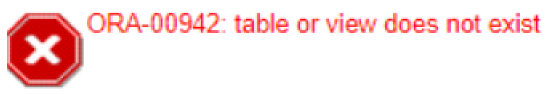
RESULT:



```
Statement processed.

0.04 seconds
```

OBSERVATION: A copy with new name is created and the original table no longer exists.

SELECT * FROM cgpa;



```
ORA-00942: table or view does not exist
```

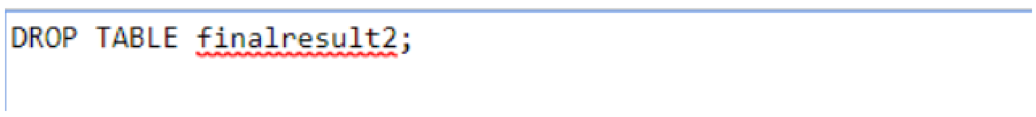
DROP

The DROP TABLE statement is used to drop an existing table in a database.

SYNTAX:

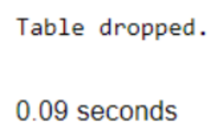
```
DROP TABLE table_name;
```

EXAMPLE:



```
DROP TABLE finalresult2;
```

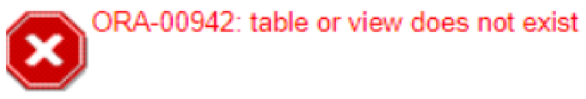
RESULT:



```
Table dropped.

0.09 seconds
```

OBSERVATION: The table no longer exists

SELECT * FROM finalresult;



```
ORA-00942: table or view does not exist
```

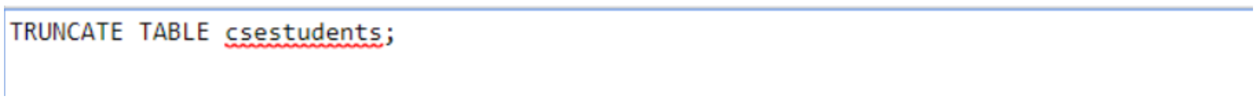
<u>TRUNCATE</u>

Truncate command removes all records from a table. But this command will not destroy the table's structure. When we apply truncate command on a table its Primary key is re-initialized.

SYNTAX:

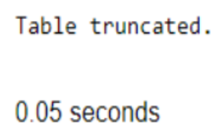
```
TRUNCATE TABLE table-name;
```

EXAMPLE:



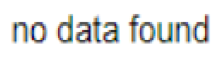
```
TRUNCATE TABLE csestudents;
```

RESULT:



```
Table truncated.

0.05 seconds
```

SELECT * FROM csestudents;



```
no data found
```

OBSERVATION:

Truncate command is different from **delete** command. Delete command will delete all the rows from a table whereas truncate command re-initializes a table (like a newly created table).

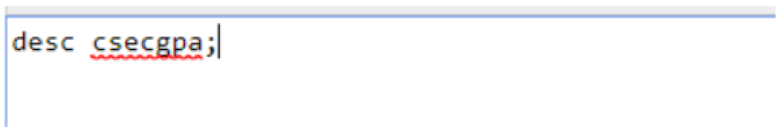
For example, If you have a table with 10 rows and an auto increment primary key, if you use delete command to delete all the rows, it will delete all the rows, but will not initialize the primary key, hence if you will insert any row after using delete command, the auto increment primary key will start from 11. But in case of truncate command, primary key is re-initialized

DESCRIBE/DESC

Provides the description of the specified table or view.

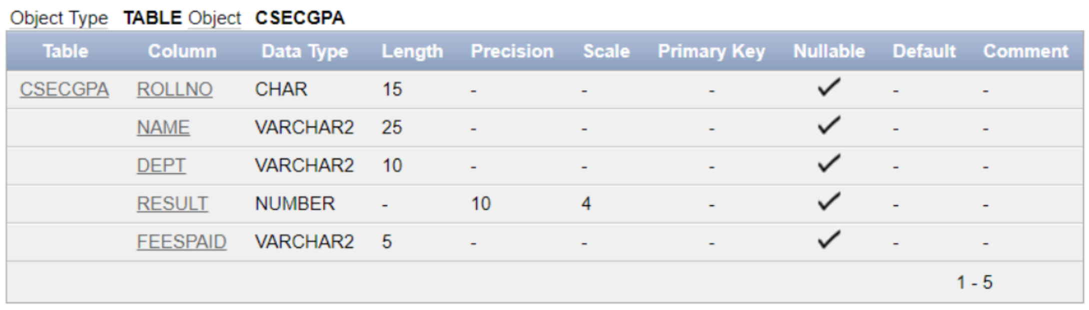
```
DESC table-name;
```

EXAMPLE:



```
desc csecgpa;
```

RESULT:



Object Type **TABLE** Object **CSECGPA**

Table	Column	Data Type	Length	Precision	Scale	Primary Key	Nullable	Default	Comment
CSECGPA	ROLLNO	CHAR	15	-	-	-	✓	-	-
	NAME	VARCHAR2	25	-	-	-	✓	-	-
	DEPT	VARCHAR2	10	-	-	-	✓	-	-
	RESULT	NUMBER	-	10	4	-	✓	-	-
	FEESPAID	VARCHAR2	5	-	-	-	✓	-	-

1 - 5

OBSERVATION: We see varchar2 instead of varchar as varchar is reserved to distinguish between null and empty string.

III. DML(Data Manipulation Language)

It provides commands that aim at manipulation of data like

INSERT
UPDATE
DELETE
SELECT

INSERT

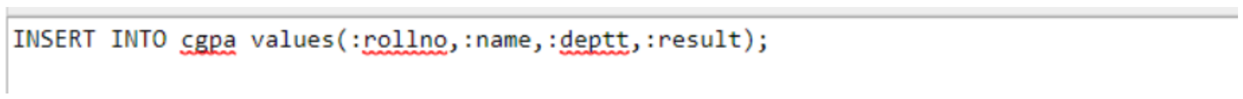
We can insert rows by two values. One is the reference method and the other where we directly insert the values.

1. **Insertion using reference method:**

SYNTAX:

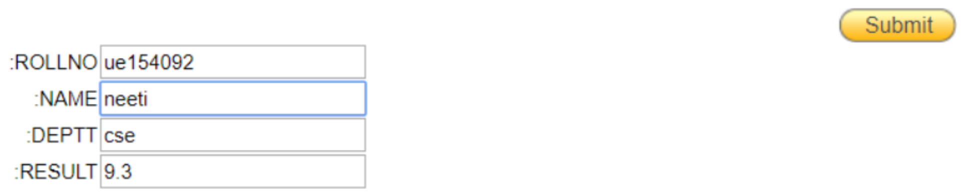
```
INSERT INTO table-name VALUES
{
 : Column-name1,: column-name2
};
```

EXAMPLE:



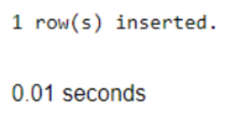
```
INSERT INTO cgpa values(:rollno,:name,:deptt,:result);
```

The following box appears on running the statement



:ROLLNO	ue154092
:NAME	neeti
:DEPTT	cse
:RESULT	9.3

RESULT:



```
1 row(s) inserted.

0.01 seconds
```

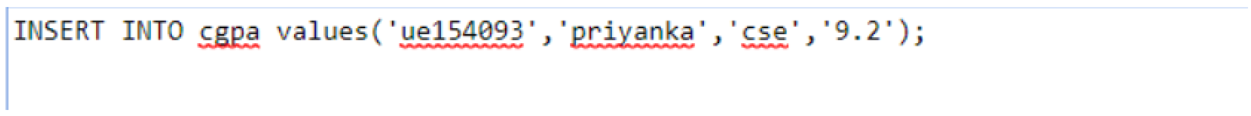
OBSERVATION: colon must be prefixed in reference method before the name.

2. **Insertion using reference method :**

SYNTAX:

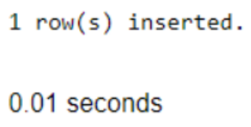
```
INSERT INTO table-name VALUES
{
 ‘Value 1’, ‘value 2’, ‘value 3’
};
```

EXAMPLE:



```
INSERT INTO cgpa values('ue154093','priyanka','cse','9.2');
```

RESULT:



```
1 row(s) inserted.

0.01 seconds
```

OBSERVATION: strings must be enclosed in single inverted commas.

SELECT

The SELECT statement is used to select data from a database.

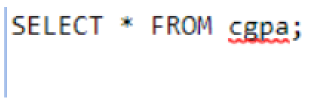
SYNTAX:

```
SELECT column1, column2
FROM table_name;
```

Here, column1, column2 are the field names of the table we want to select data from. If we want to select all the fields available in the table, we use the following syntax:

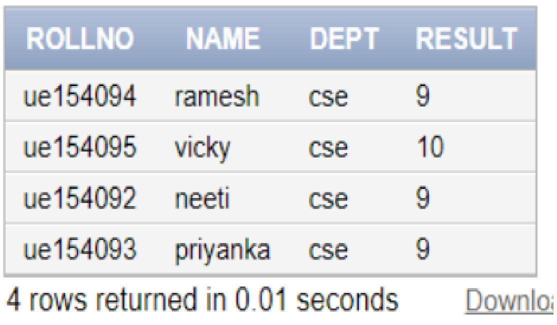
```
SELECT * FROM table_name;
```

EXAMPLE:



```
SELECT * FROM cgpa;
```

RESULT:



ROLLNO	NAME	DEPT	RESULT
ue154094	ramesh	cse	9
ue154095	vicky	cse	10
ue154092	neeti	cse	9
ue154093	priyanka	cse	9

4 rows returned in 0.01 seconds Downlo :

OBSERVATION: Precision missing since we did not specify the size of number as (n,p)

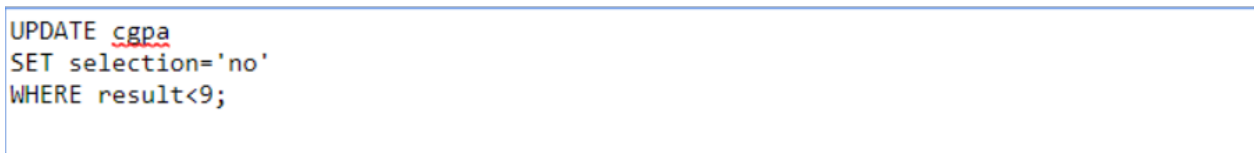
UPDATE

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

SYNTAX:

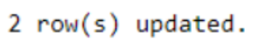
```
UPDATE table_name
SET column1 = value1, column2 = value2
WHERE condition;
```

EXAMPLE:



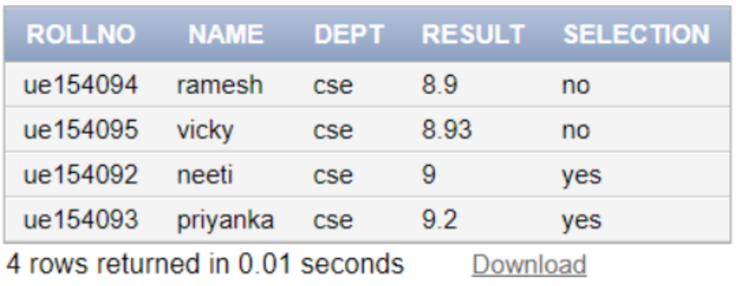
```
UPDATE cgpa
SET selection='no'
WHERE result<9;
```

RESULT:



```
2 row(s) updated.
```

SELECT * FROM cgpa;



ROLLNO	NAME	DEPT	RESULT	SELECTION
ue154094	ramesh	cse	8.9	no
ue154095	vicky	cse	8.93	no
ue154092	neeti	cse	9	yes
ue154093	priyanka	cse	9.2	yes

4 rows returned in 0.01 seconds Download

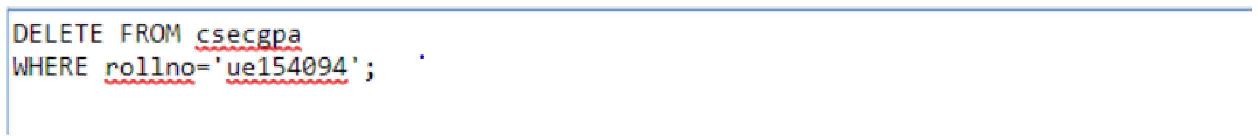
DELETE

The DELETE statement is used to delete existing records in a table or the entire table.

SYNTAX:

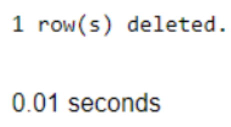
```
DELETE FROM table_name
WHERE condition;
```

EXAMPLE:



```
DELETE FROM csecgpa
WHERE rollno='ue154094';
```

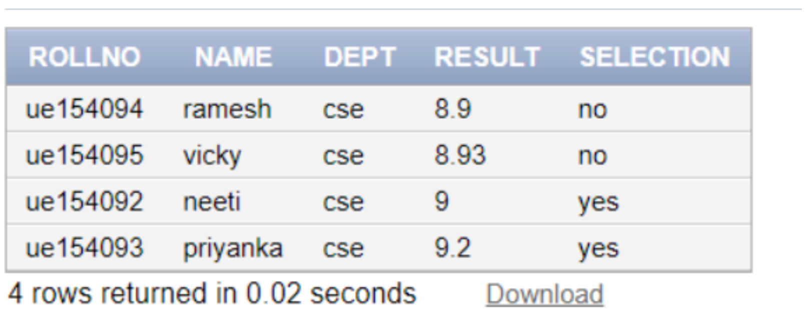
RESULT:



```
1 row(s) deleted.

0.01 seconds
```

SELECT * FROM csecgpa;

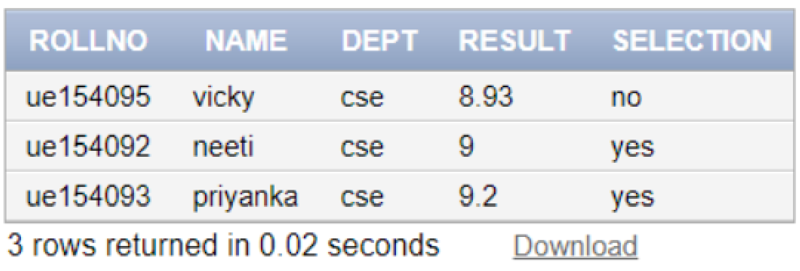
Earlier:



ROLLNO	NAME	DEPT	RESULT	SELECTION
ue154094	ramesh	cse	8.9	no
ue154095	vicky	cse	8.93	no
ue154092	neeti	cse	9	yes
ue154093	priyanka	cse	9.2	yes

4 rows returned in 0.02 seconds   Download

After deletion:



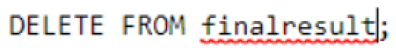
ROLLNO	NAME	DEPT	RESULT	SELECTION
ue154095	vicky	cse	8.93	no
ue154092	neeti	cse	9	yes
ue154093	priyanka	cse	9.2	yes

3 rows returned in 0.02 seconds   Download

SYNTAX:

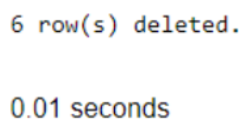
```
DELETE FROM table_name;
```

EXAMPLE:



```
DELETE FROM finalresult;
```

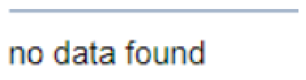
RESULT:



```
6 row(s) deleted.

0.01 seconds
```

OBSERVATION: The schema of the table still exists but primary key is not re-initialized. TRUNCATE works the same way as delete but the major difference is delete shows' **no. of rows deleted'** in results whereas truncate shows **'table truncated'** and no data found when select command used

SELECT * FROM finalresult;



```
no data found
```

IV. DCL*(*Data Control Language*)*

It provides commands to help the DBA (Database Administrator) to control the database.

GRANT
REVOKE

GRANT

GRANT is a command used to provide access or privileges on the database objects to the users.

SYNTAX:

```
GRANT privilege_name
ON object_name
TO user_name /ALL;
```

```
GRANT ALL                         //all data manipulation permission
ON object_name/table_name
TO user_name /ALL;
```

```
GRANT ALL
ON object_name/table_name
TO user_name /ALL
WITH GRANT OPTION;               //further grants on table to others
```

REVOKE

The REVOKE command removes user access rights or privileges to the database objects.

SYNTAX:

```
REVOKE privilege_name
ON object_name
FROM user_name;
```

```
REVOKE ALL
ON object_name
FROM user_name;
```

V. TCL(Transaction Control Language)

Transaction Control Language (TCL) commands are used to manage transactions in database. These are used to manage the changes made by DML statements.

COMMIT
ROLLBACK
SAVEPOINT

COMMIT

Commit command is used to permanently save any transaction into database.

SYNTAX:

```
COMMIT;
```

NOTE: DDL statements are auto-commit.

ROLLBACK

This command restores the database to last committed state. It is also use with savepoint command to jump to a savepoint in a transaction.

SYNTAX:

```
ROLLBACK TO savepoint-name;
```

NOTE: Rollback doesn't work on commit.

SAVEPOINT

This command is used to temporarily save a transaction so that you can rollback to that point whenever necessary.

SYNTAX:

```
SAVEPOINT savepoint-name;
```

Constraints and its types

Applying constraints means giving instructions to filter what is being stored in the table.

Types of constraints

1. I/O Constraints

Constrains that control data insertion and data retrieval speed are known as I/O (Input-Output) constraints.

It includes the following constraints.

- **Primary Key**
- **Foreign Key**

2. Business Constraints

The rules are applied to data prior the data being inserted into the table columns. For example

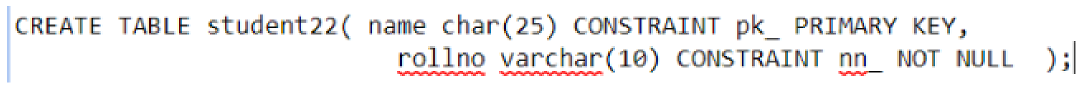
- **Unique,**
- **Not NULL**
- **Default constraints**
- **Check constraint**

Oracle allows the user to define constraints at

- **Column level**
  If data constraints are defined as an attribute of a column definition when creating or altering the table structure they are called column level constraints.
- **Table level**
  If data constraints are defined after defining all table column attributes when creating or altering the table structure they are called table level constraints.

NOTE: Naming of constraints is useful for easy debugging

Column level naming:

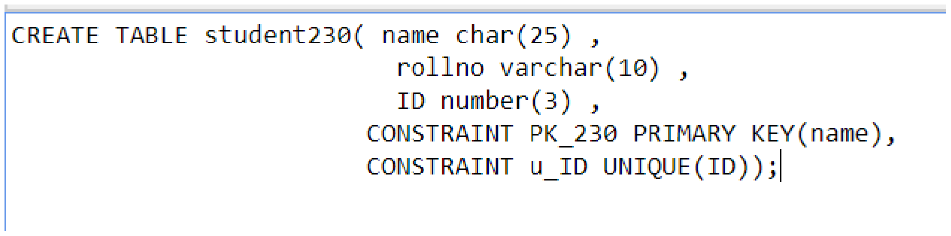


```
CREATE TABLE student22( name char(25) CONSTRAINT pk_ PRIMARY KEY,
                        rollno varchar(10) CONSTRAINT nn_ NOT NULL  );
```

Column_name datatype(size) CONSTRAINT constraint_name TYPE

Table level naming:

CONSTRAINT constraint_name TYPE(attribute)



```
CREATE TABLE student230( name char(25) ,
                         rollno varchar(10) ,
                         ID number(3) ,
                       CONSTRAINT PK_230 PRIMARY KEY(name),
                       CONSTRAINT u_ID UNIQUE(ID));
```

PRIMARY KEY

- Primary key is used to identify a record uniquely from the database table.
- A primary key means **UNIQUE + NOT NULL.**
- Value must be available for the column on which primary key has been defined.
- User cannot leave the field blank.
- It cannot contain duplicate values.
- Primary key can be defined either at table level or at column level.
- **Primary key** keyword is used to define primary key constraint

- We can define single primary key for a single table.
- When a single column is not sufficient for the primary key purpose than primary key can be defined on multiple columns. Such primary key is known as **composite primary key**

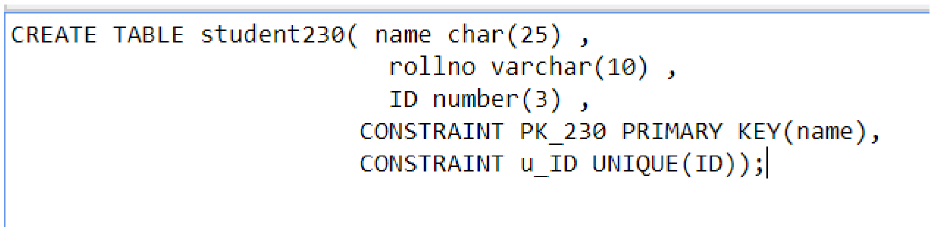
Primary key (at column level):

```
CREATE TABLE student (rollno (3) PRIMARY KEY,
                        Name varchar2 (15));
```

Primary key (at table level):

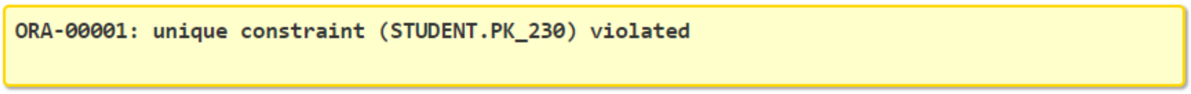
```
CREATE TABLE emp (firstname varchar2 (15),
                   lastname varchar2 (15),
                  salary number (7, 2),
                  PRIMARY KEY (firstname, lastname)
                 );
```

EXAMPLE:



```
CREATE TABLE student230( name char(25) ,
                          rollno varchar(10) ,
                          ID number(3) ,
                        CONSTRAINT PK_230 PRIMARY KEY(name),
                        CONSTRAINT u_ID UNIQUE(ID));
```

OBSERVATION: has to be unique and not null else it's a violation



```
ORA-00001: unique constraint (STUDENT.PK_230) violated
```

FOREIGN KEY

- A foreign key constraint is used to establish logical relationship between two or more tables.
- Foreign key is also known as **referential key**.
- It can be defined using the keyword **"references"**.

- The main table which is logically linked with other table is known as '**Parent table**' or "**master table"** while other table is referred to as '**Child table' or "detail table".**

The following point should be kept in the mind while defining foreign key.

1. **Data type** and **size** of the parent table and child table should be the same.
2. Record can be inserted in detail table only if relevant record is available into the master table that means while inserting record first we need to insert record into the parent table then and then we can insert record into the child table.
3. To delete a specific record, first we need to delete a relevant record from the detail table and after that record can be deleted from the parent table.

Foreign Key (column level)

// Parent table definition

```
 CREATE TABLE student (rollno number (3) primary           key,
name varchar2 (15));
```

//Child table definition

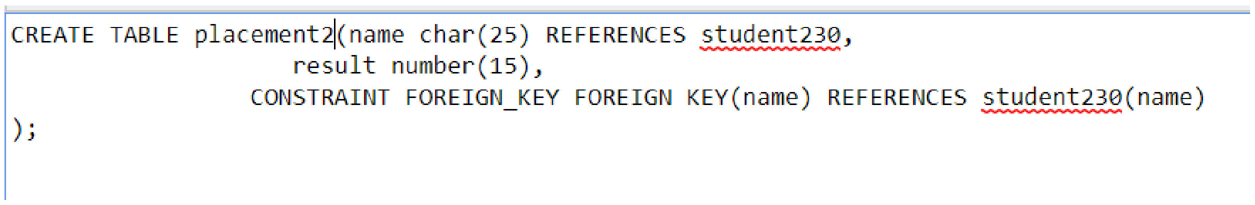
```
CREATE TABLE studdetail (rollno number (3) REFERENCES student
(rollno), address varchar2 (30), city varchar2 (15));
```

Foreign Key (Table level)

```
CREATE TABLE studdetail (rollno number (3),

 Address varchar2 (30), city varchar2 (15), FOREIGN KEY (rollno)
REFERENCES student (rollno));
```

EXAMPLE:

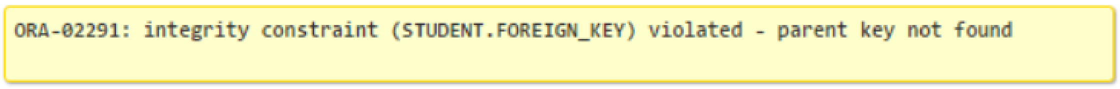
Table level:



```
CREATE TABLE placement2(name char(25) REFERENCES student230,
                    result number(15),
                 CONSTRAINT FOREIGN_KEY FOREIGN KEY(name) REFERENCES student230(name)
);
```

OBSERVATION:

If the domain of foreign key is not primary key it results in violation of differential integrity constraint and the following message appears.



```
ORA-02291: integrity constraint (STUDENT.FOREIGN_KEY) violated - parent key not found
```

UNIQUE

- A unique key constraint can be defined when we do not want the user to enter duplicate values.
- A unique constraint allows only unique value to be inserted.
- However it allows null value to be inserted.
- Similar to not null constraint unique key can also be define on multiple columns.
- **"Unique"** keyword is use to define unique constraint.

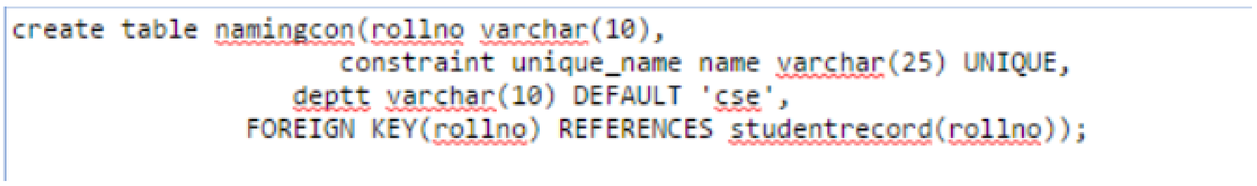
Unique Key (Column level)

```
CREATE TABLE student (rollno number (3),
                      Name varchar2 (3),
                      Mobile varchar2 (10) UNIQUE
                     );
```

Unique Key (Table level)

```
CREATE TABLE student (rollno varchar2 (3),
                     Name varchar2 (15),
                      City varchar2 (15),
                     UNIQUE (rollno, name)
                    );
```

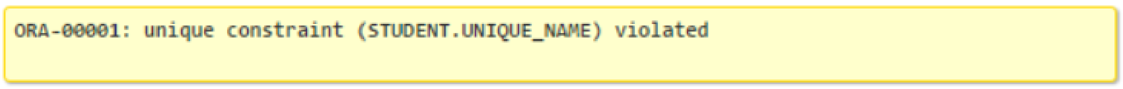
EXAMPLE:



```
create table namingcon(rollno varchar(10),
                    constraint unique_name name varchar(25) UNIQUE,
                 deptt varchar(10) DEFAULT 'cse',
              FOREIGN KEY(rollno) REFERENCES studentrecord(rollno));
```

OBSERVATION:

In case of violation



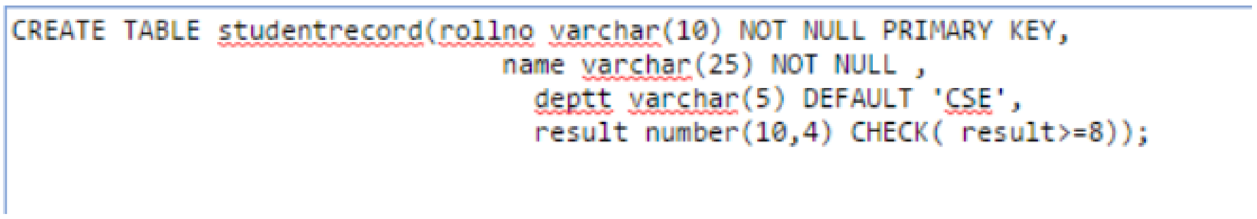
```
ORA-00001: unique constraint (STUDENT.UNIQUE_NAME) violated
```

NOT NULL

- We can define not null constraint when we do not want the user to leave the field blank.
- However it allows duplicate value.
- A table can contain not null constraint on multiple columns.
- **Not null constraint can only be defined at column level that means it cannot be defined at table level.**
- **'Not Null'** keyword is used to define not null constraint.

```
CREATE TABLE student (rollno number (3) Primary key,
                      Name varchar2 (15) not null,
                      Mob number (15) not null
                    );
```

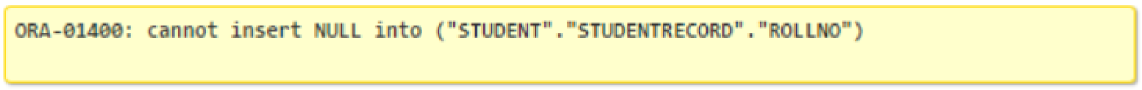
EXAMPLE:



```
CREATE TABLE studentrecord(rollno varchar(10) NOT NULL PRIMARY KEY,
                            name varchar(25) NOT NULL ,
                              deptt varchar(5) DEFAULT 'CSE',
                              result number(10,4) CHECK( result>=8));
```

OBSERVATION:

In case of violation



```
ORA-01400: cannot insert NULL into ("STUDENT"."STUDENTRECORD"."ROLLNO")
```

CHECK

- **"Check"** keyword is used to define business constraints.
- It can be defined as a logical expression that returns either *true* or *false*.
- Each time when a new record inserted, check constraint will be evaluate and if it returns true then record will be inserted and rejected otherwise.
- A check constraint expression must be a logical expression.
- It can also be defined either at column level or at table level.

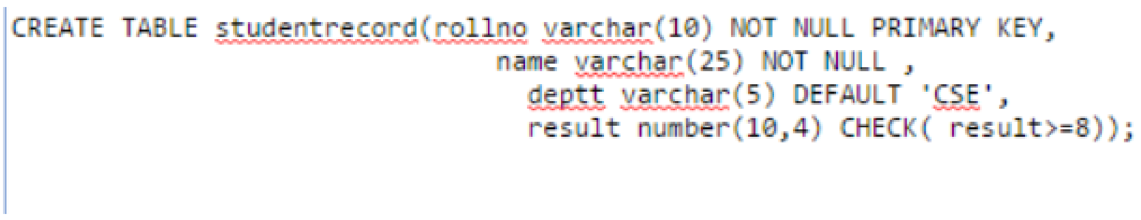
Check constraint (Column level)

```
CREATE TABLE emp ( emp_code varchar2 (4) check (emp_code LIKE 'C%'),
                 emp_name varchar2 (15) not null,
                 Sal number (7, 2) CHECK(Sal>1000)
                ) ;
```

Check constraint (Table level)

```
CREATE TABLE emp ( emp_code varchar2 (4),
                 emp_name varchar2 (15) not null,
                 Sal number (7, 2),
                CHECK (emp_code LIKE 'C %'),
                 CHECK (Sal>1000)
                ) ;
```

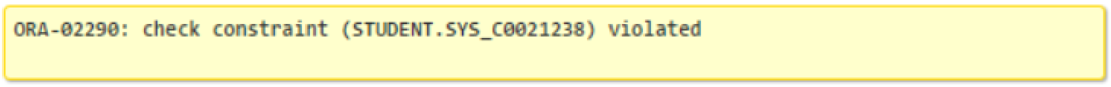
EXAMPLE:



```
CREATE TABLE studentrecord(rollno varchar(10) NOT NULL PRIMARY KEY,
                               name varchar(25) NOT NULL ,
                                 deptt varchar(5) DEFAULT 'CSE',
                                 result number(10,4) CHECK( result>=8));
```

OBSERVATION:

In case of violation



```
ORA-02290: check constraint (STUDENT.SYS_C0021238) violated
```

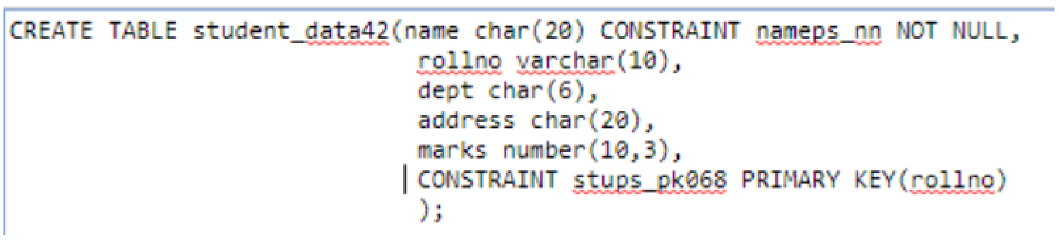
DEFAULT

The DEFAULT constraint is used to provide a default value for a column. The default value will be added to all new records IF no other value is specified.

NOTE: no naming of default constraint

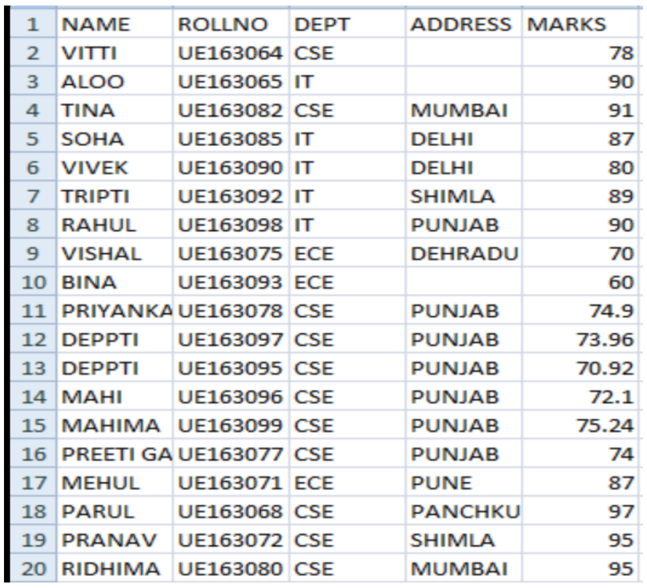
```
CREATE TABLE Persons (
    ID number(3) NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age number(50),
    City varchar(255) DEFAULT 'Sandnes'
);
```

String and numeric functions



```
CREATE TABLE student_data42(name char(20) CONSTRAINT nameps_nn NOT NULL,
                            rollno varchar(10),
                            dept char(6),
                            address char(20),
                            marks number(10,3),
                            CONSTRAINT stups_pk068 PRIMARY KEY(rollno)
                            );
```

Let the original database be as follow:



1	NAME	ROLLNO	DEPT	ADDRESS	MARKS
2	VITTI	UE163064	CSE		78
3	ALOO	UE163065	IT		90
4	TINA	UE163082	CSE	MUMBAI	91
5	SOHA	UE163085	IT	DELHI	87
6	VIVEK	UE163090	IT	DELHI	80
7	TRIPTI	UE163092	IT	SHIMLA	89
8	RAHUL	UE163098	IT	PUNJAB	90
9	VISHAL	UE163075	ECE	DEHRADU	70
10	BINA	UE163093	ECE		60
11	PRIYANKA	UE163078	CSE	PUNJAB	74.9
12	DEPPTI	UE163097	CSE	PUNJAB	73.96
13	DEPPTI	UE163095	CSE	PUNJAB	70.92
14	MAHI	UE163096	CSE	PUNJAB	72.1
15	MAHIMA	UE163099	CSE	PUNJAB	75.24
16	PREETI GA	UE163077	CSE	PUNJAB	74
17	MEHUL	UE163071	ECE	PUNE	87
18	PARUL	UE163068	CSE	PANCHKU	97
19	PRANAV	UE163072	CSE	SHIMLA	95
20	RIDHIMA	UE163080	CSE	MUMBAI	95

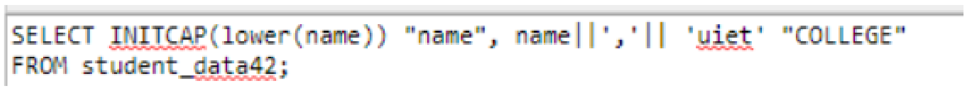
String Functions

Function	Description
CONCATENATE	Concatenates two or more strings together
INSTR	Returns the position of the first occurrence of a string in another string
INITCAP	Capitalizes the first alphabet
LENGTH	Returns the length of the specified string (in bytes)

LOWER/LCASE	Converts a string to lower-case
LPAD	Returns a string that is left-padded with a specified string to a certain length
LTRIM	Removes leading spaces from a string
RPAD	Returns a string that is right-padded with a specified string to a certain length
RTRIM	Removes trailing spaces from a string
SUBSTR	Extracts a substring from a string (starting at any position)
UPPER/UCASE	Converts a string to upper-case
TRANSLATE	This function replaces a sequence of characters in a string with another sequence of characters.

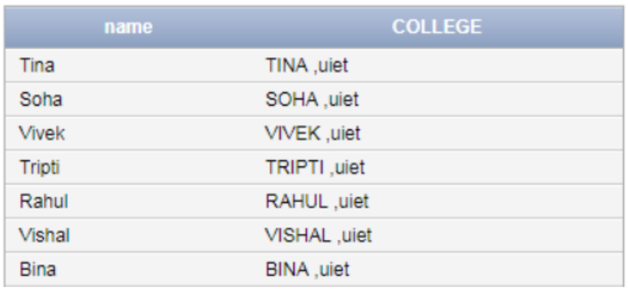
CONCATENATE ()

EXAMPLE:



```
SELECT INITCAP(lower(name)) "name", name||','|| 'uiet' "COLLEGE"
FROM student_data42;
```

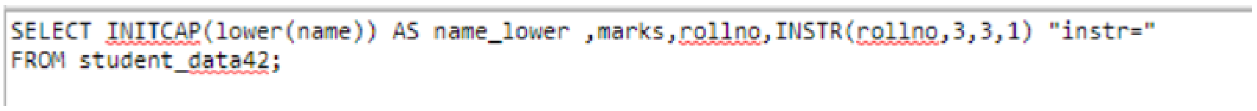
RESULT: Uiet is concatenated with name



name	COLLEGE
Tina	TINA ,uiet
Soha	SOHA ,uiet
Vivek	VIVEK ,uiet
Tripti	TRIPTI ,uiet
Rahul	RAHUL ,uiet
Vishal	VISHAL ,uiet
Bina	BINA ,uiet

INSTR () and INITCAP ()

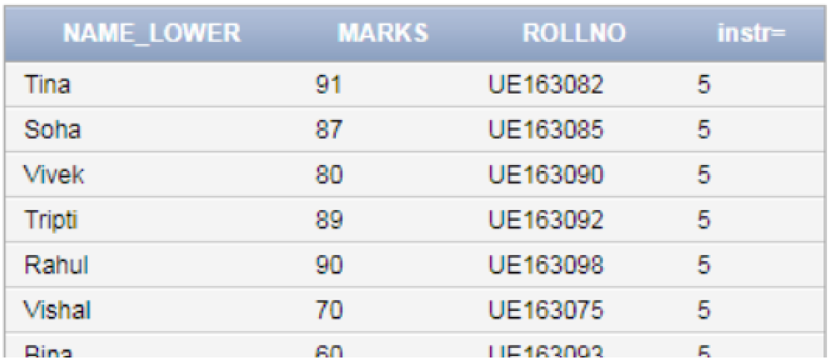
EXAMPLE:



```
SELECT INITCAP(lower(name)) AS name_lower ,marks,rollno,INSTR(rollno,3,3,1) "instr="
FROM student_data42;
```

RESULT: Looks for the first occurrence of 3 , starting from the third position in rollno .

INTITCAP () capitalizes the first alphabet.



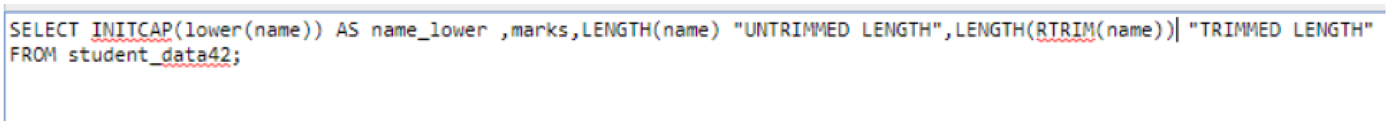
NAME_LOWER	MARKS	ROLLNO	instr=
Tina	91	UE163082	5
Soha	87	UE163085	5
Vivek	80	UE163090	5
Tripti	89	UE163092	5
Rahul	90	UE163098	5
Vishal	70	UE163075	5
Bina	60	UE163093	5

NOTE:

- INSTR() performs a case-insensitive search
- The first position in *string* is 1
- If *substring* is not found within *string*, the INSTR() function will return 0

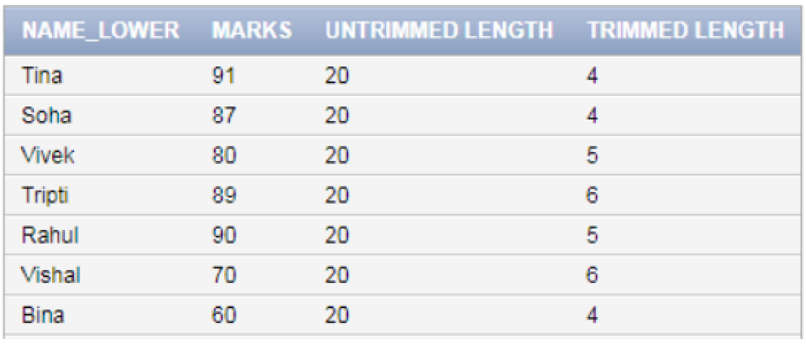
<u>LENGTH () and RTRIM ()</u>

EXAMPLE:



```
SELECT INITCAP(lower(name)) AS name_lower ,marks,LENGTH(name) "UNTRIMMED LENGTH",LENGTH(RTRIM(name)) "TRIMMED LENGTH"
FROM student_data42;
```

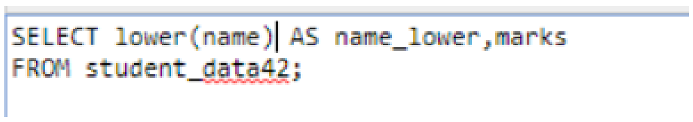
RESULT: LENGTH () function gives the untrimmed length i.e. with trailing spaces.



NAME_LOWER	MARKS	UNTRIMMED LENGTH	TRIMMED LENGTH
Tina	91	20	4
Soha	87	20	4
Vivek	80	20	5
Tripti	89	20	6
Rahul	90	20	5
Vishal	70	20	6
Bina	60	20	4

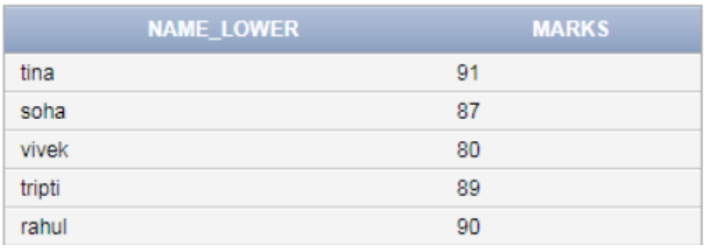
LOWER ()

EXAMPLE:



```
SELECT lower(name) AS name_lower,marks
FROM student_data42;
```

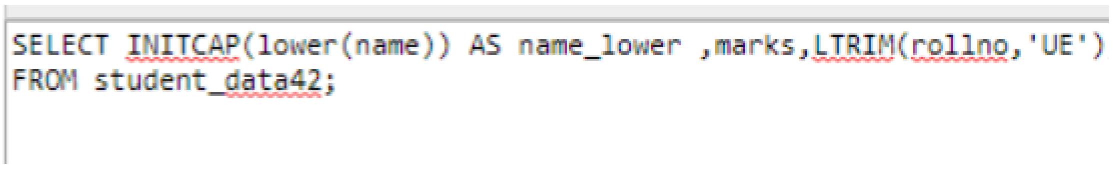
RESULT:



NAME_LOWER	MARKS
tina	91
soha	87
vivek	80
tripti	89
rahul	90

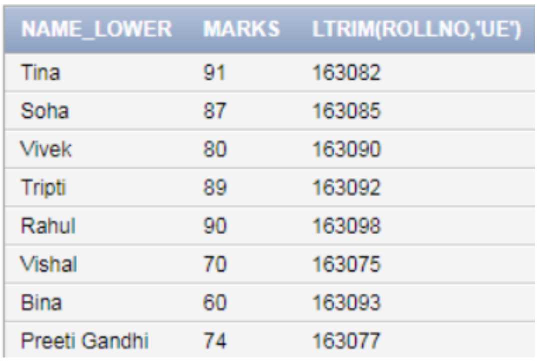
LTRIM ()

EXAMPLE:



```
SELECT INITCAP(lower(name)) AS name_lower ,marks,LTRIM(rollno,'UE')
FROM student_data42;
```

RESULT:



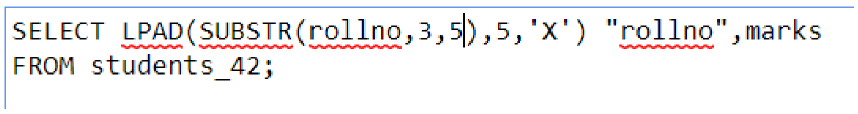
NAME_LOWER	MARKS	LTRIM(ROLLNO,'UE')
Tina	91	163082
Soha	87	163085
Vivek	80	163090
Tripti	89	163092
Rahul	90	163098
Vishal	70	163075
Bina	60	163093
Preeti Gandhi	74	163077

LPAD ()

SYNTAX:

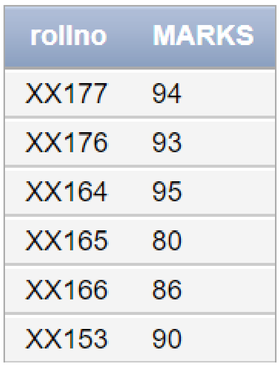
LPAD (*string to right pad*, *length*, *pad_string*)

EXAMPLE:



```
SELECT LPAD(SUBSTR(rollno,3,5),5,'X') "rollno",marks
FROM students_42;
```

RESULT:



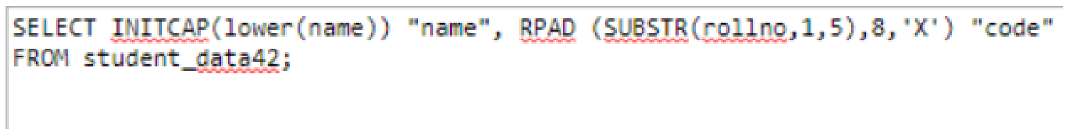
rollno	MARKS
XX177	94
XX176	93
XX164	95
XX165	80
XX166	86
XX153	90

RPAD ()

SYNTAX:

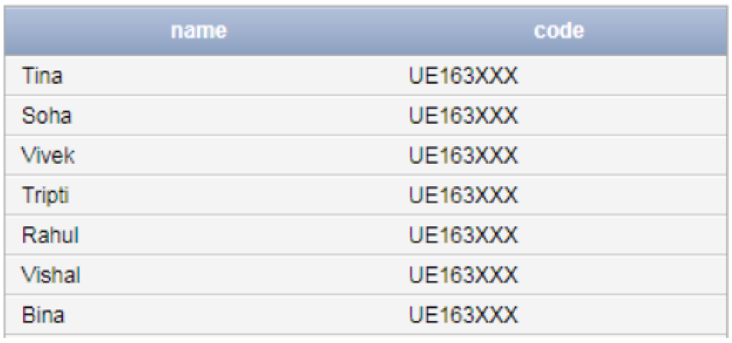
RPAD (*string to right pad*, *length*, *pad_string*)

EXAMPLE:



```
SELECT INITCAP(lower(name)) "name", RPAD (SUBSTR(rollno,1,5),8,'X') "code"
FROM student_data42;
```

RESULT:



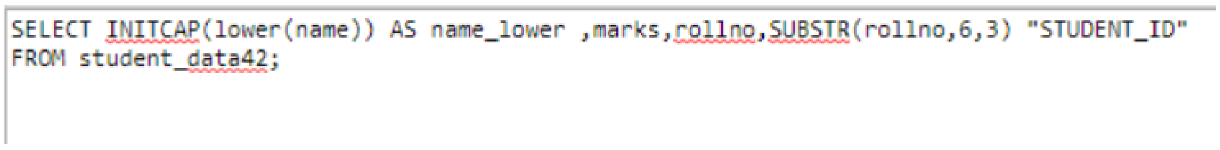
name	code
Tina	UE163XXX
Soha	UE163XXX
Vivek	UE163XXX
Tripti	UE163XXX
Rahul	UE163XXX
Vishal	UE163XXX
Bina	UE163XXX

SUBSTR ()

SYNTAX:

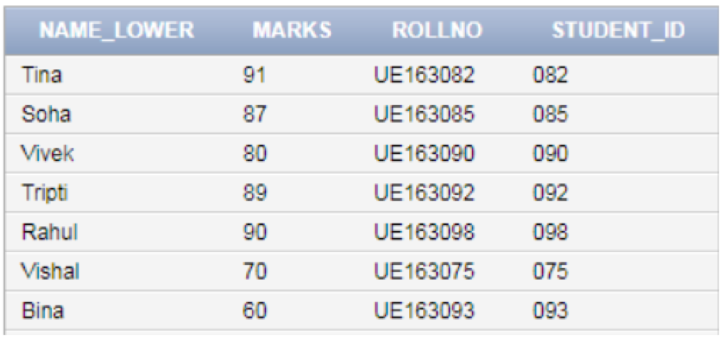
SUBSTR (*string*, *start*, *length*)

EXAMPLE:



```
SELECT INITCAP(lower(name)) AS name_lower ,marks,rollno,SUBSTR(rollno,6,3) "STUDENT_ID"
FROM student_data42;
```

RESULT: 6 is the starting position and 3 characters of the string are selected

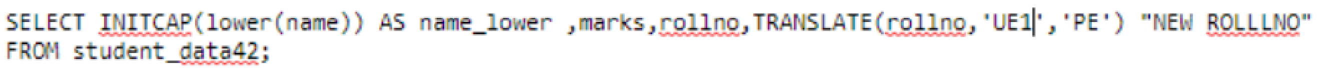


NAME_LOWER	MARKS	ROLLNO	STUDENT_ID
Tina	91	UE163082	082
Soha	87	UE163085	085
Vivek	80	UE163090	090
Tripti	89	UE163092	092
Rahul	90	UE163098	098
Vishal	70	UE163075	075
Bina	60	UE163093	093

TRANSLATE ()

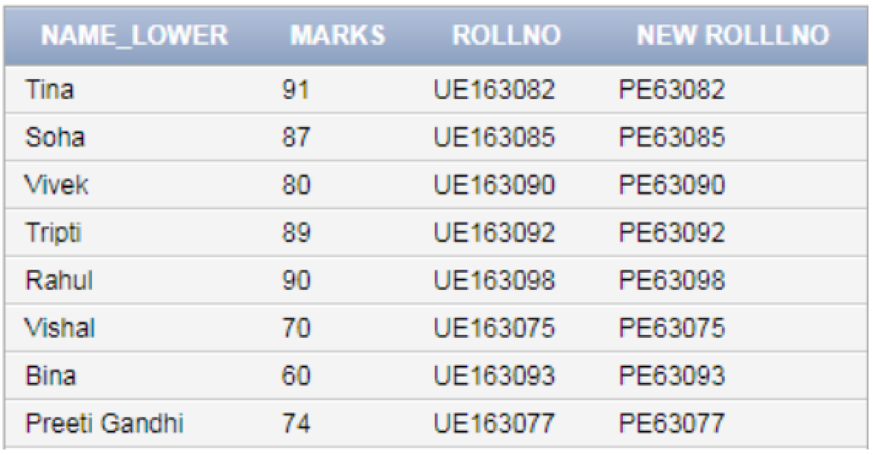
There is no one to one correspondence between replacing of characters

EXAMPLE:



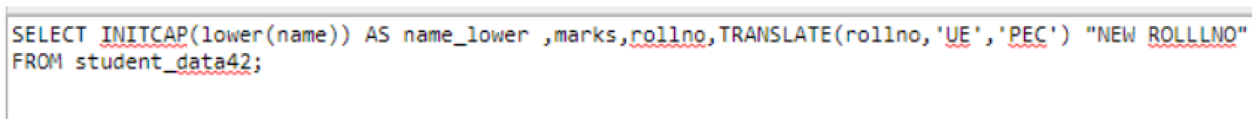
```
SELECT INITCAP(lower(name)) AS name_lower ,marks,rollno,TRANSLATE(rollno,'UE1','PE') "NEW ROLLLNO"
FROM student_data42;
```

RESULT:

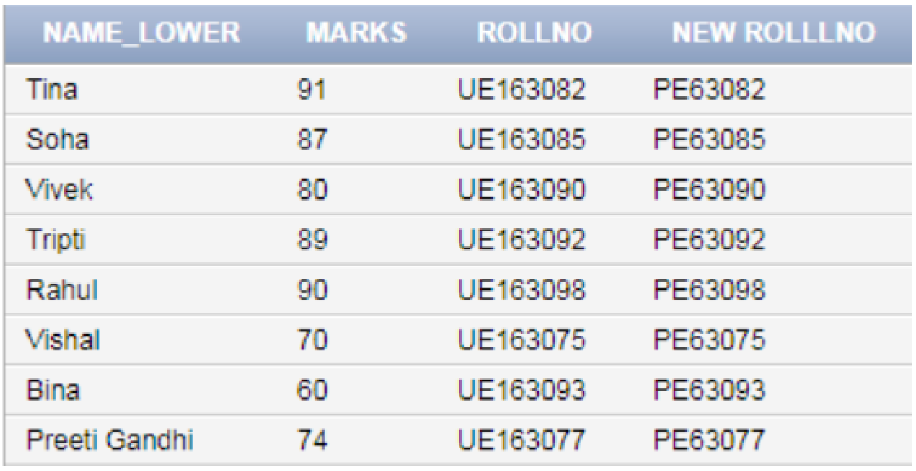


NAME_LOWER	MARKS	ROLLNO	NEW ROLLLNO
Tina	91	UE163082	PE63082
Soha	87	UE163085	PE63085
Vivek	80	UE163090	PE63090
Tripti	89	UE163092	PE63092
Rahul	90	UE163098	PE63098
Vishal	70	UE163075	PE63075
Bina	60	UE163093	PE63093
Preeti Gandhi	74	UE163077	PE63077

NOTE: A longer string cannot replace a shorter string



```
SELECT INITCAP(lower(name)) AS name_lower ,marks,rollno,TRANSLATE(rollno,'UE','PEC') "NEW ROLLLNO"
FROM student_data42;
```



NAME_LOWER	MARKS	ROLLNO	NEW ROLLLNO
Tina	91	UE163082	PE63082
Soha	87	UE163085	PE63085
Vivek	80	UE163090	PE63090
Tripti	89	UE163092	PE63092
Rahul	90	UE163098	PE63098
Vishal	70	UE163075	PE63075
Bina	60	UE163093	PE63093
Preeti Gandhi	74	UE163077	PE63077

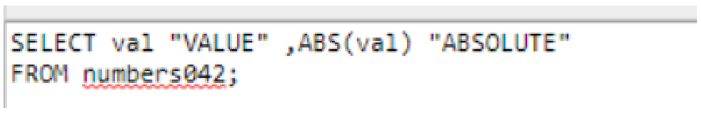
Numeric Functions

Function	**Description**
ABS	Returns the absolute value of a number
AVG	Returns the average value of an expression
CEIL	Returns the smallest integer value that is greater than or equal to a number
COUNT	Returns the number of records in a select query
EXP	Returns e raised to the power of number
FLOOR	Returns the largest integer value that is less than or equal to a number
GREATEST	Returns the greatest value in a list of expressions
LEAST	Returns the smallest value in a list of expressions

MAX	Returns the maximum value of an expression
MIN	Returns the minimum value of an expression
MOD	Returns the remainder of n divided by m
POWER	Returns m raised to the nth power
ROUND	Returns a number rounded to a certain number of decimal places
SQRT	Returns the square root of a number
SUM	Returns the summed value of an expression
TRUNCATE	Returns a number truncated to a certain number of decimal places

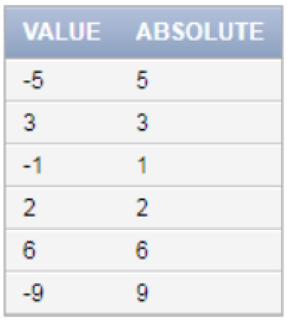
ABS ()

EXAMPLE:



```
SELECT val "VALUE" ,ABS(val) "ABSOLUTE"
FROM numbers042;
```

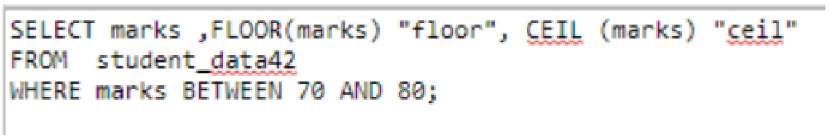
RESULT:



VALUE	ABSOLUTE
-5	5
3	3
-1	1
2	2
6	6
-9	9

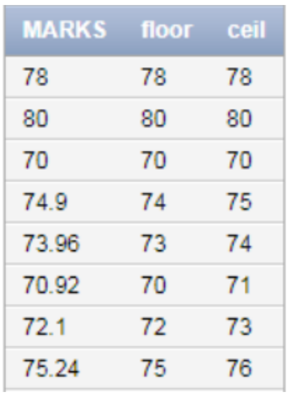
CEIL () and FLOOR ()

EXAMPLE:



```
SELECT marks ,FLOOR(marks) "floor", CEIL (marks) "ceil"
FROM  student_data42
WHERE marks BETWEEN 70 AND 80;
```

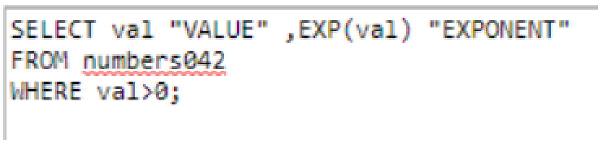
RESULT:



MARKS	floor	ceil
78	78	78
80	80	80
70	70	70
74.9	74	75
73.96	73	74
70.92	70	71
72.1	72	73
75.24	75	76

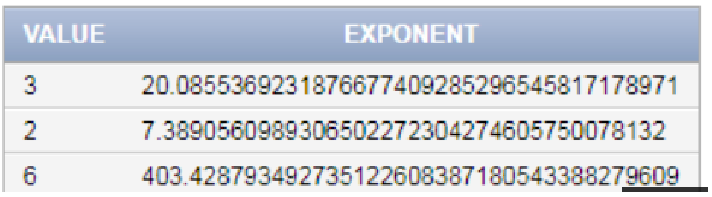
EXP ()

EXAMPLE:



```
SELECT val "VALUE" ,EXP(val) "EXPONENT"
FROM numbers042
WHERE val>0;
```

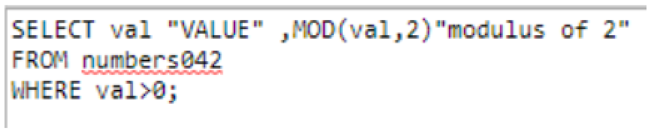
RESULT:



VALUE	EXPONENT
3	20.085536923187667740928529654581717897 1
2	7.3890560989306502272304274605750078132
6	403.42879349273512260838718054338827960 9

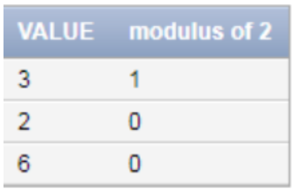
MOD ()

EXAMPLE:



```
SELECT val "VALUE" ,MOD(val,2)"modulus of 2"
FROM numbers042
WHERE val>0;
```

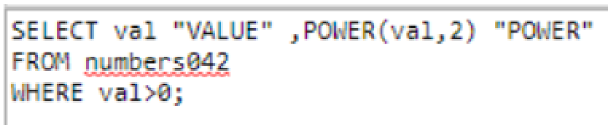
RESULT:



VALUE	modulus of 2
3	1
2	0
6	0

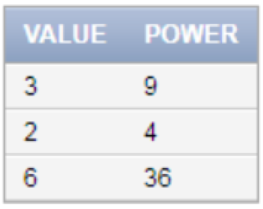
POWER ()

EXAMPLE:



```
SELECT val "VALUE" ,POWER(val,2) "POWER"
FROM numbers042
WHERE val>0;
```

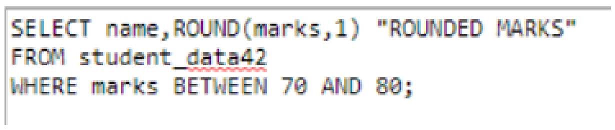
RESULT:



VALUE	POWER
3	9
2	4
6	36

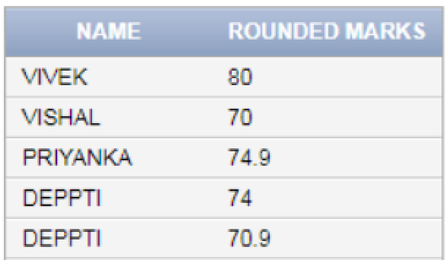
ROUND ()

EXAMPLE:



```
SELECT name,ROUND(marks,1) "ROUNDED MARKS"
FROM student_data42
WHERE marks BETWEEN 70 AND 80;
```

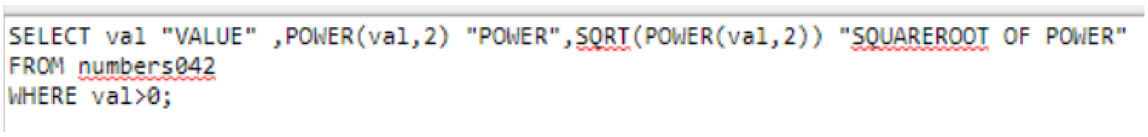
RESULT:



NAME	ROUNDED MARKS
VIVEK	80
VISHAL	70
PRIYANKA	74.9
DEPPTI	74
DEPPTI	70.9

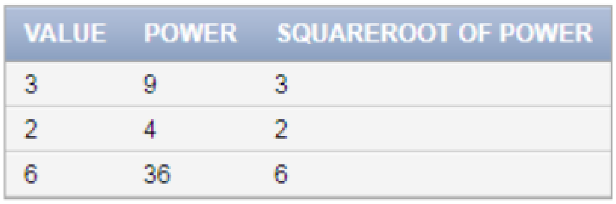
SQRT ()

EXAMPLE:



```
SELECT val "VALUE" ,POWER(val,2) "POWER",SQRT(POWER(val,2)) "SQUAREROOT OF POWER"
FROM numbers042
WHERE val>0;
```

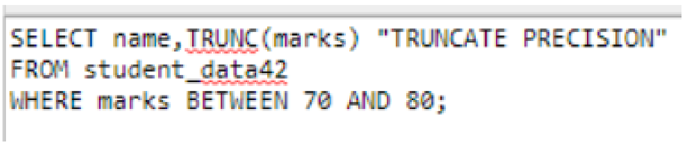
RESULT:



VALUE	POWER	SQUAREROOT OF POWER
3	9	3
2	4	2
6	36	6

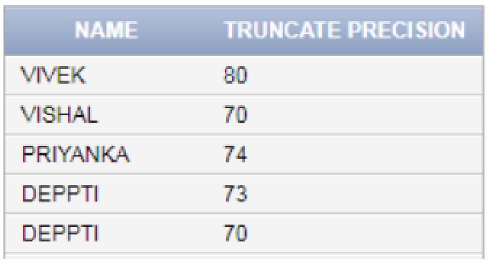
TRUNCATE ()

EXAMPLE:



```
SELECT name,TRUNC(marks) "TRUNCATE PRECISION"
FROM student_data42
WHERE marks BETWEEN 70 AND 80;
```

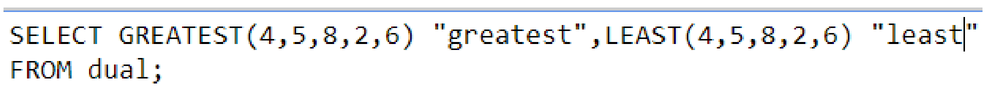
RESULT:



NAME	TRUNCATE PRECISION
VIVEK	80
VISHAL	70
PRIYANKA	74
DEPPTI	73
DEPPTI	70

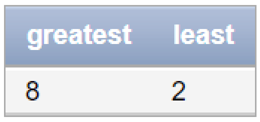
GREATEST () and LEAST()

EXAMPLE:



```
SELECT GREATEST(4,5,8,2,6) "greatest",LEAST(4,5,8,2,6) "least"
FROM dual;
```

RESULT:



greatest	least
8	2

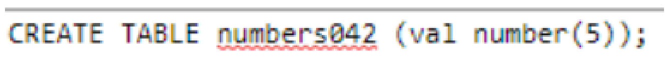
NOTE: greatest () and least () take multiple numbers and input and not columns like max () and min ()

DUAL: The DUAL is special one row, one column table present by default in all Oracle databases. My SQL allows DUAL to be specified as a table in queries that do not need data from any tables.

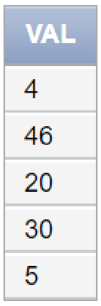
Aggregate functions

COUNT (), AVG () , SUM () ,MAX (),MIN () are aggregate functions which take set of values as input and return a single value as output.

Let the table be



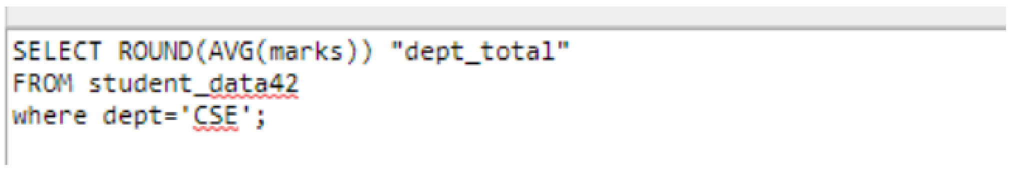
```
CREATE TABLE numbers042 (val number(5));
```



VAL
4
46
20
30
5

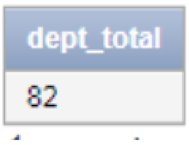
AVG ()

EXAMPLE:



```
SELECT ROUND(AVG(marks)) "dept_total"
FROM student_data42
where dept='CSE';
```

RESULT:

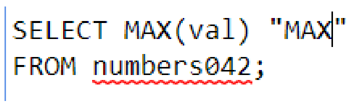


dept_total
82

The above query returns the avg marks in the cse department as dept_total after rounding it.

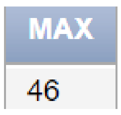
MAX ()

EXAMPLE:



```
SELECT MAX(val) "MAX"
FROM numbers042;
```

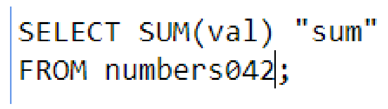
RESULT:



MAX
46

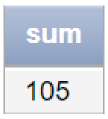
SUM ()

EXAMPLE:



```
SELECT SUM(val) "sum"
FROM numbers042;
```

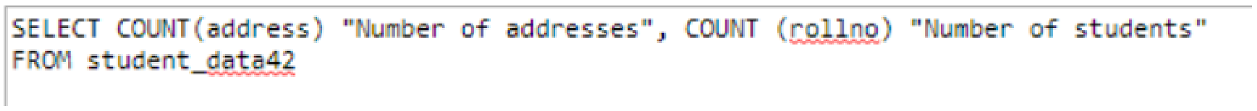
RESULT:



sum
105

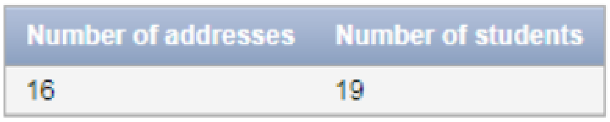
COUNT ()

EXAMPLE:



```
SELECT COUNT(address) "Number of addresses", COUNT (rollno) "Number of students"
FROM student_data42
```

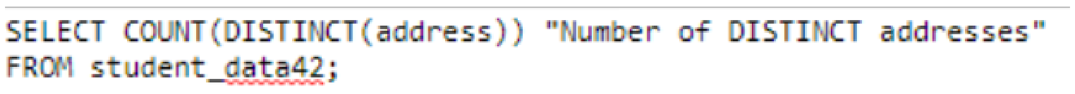
RESULT: Count does not include null spaces



Number of addresses	Number of students
16	19

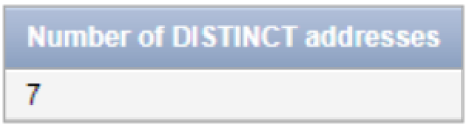
NOTE: Aggregate functions include duplicate inputs as well. Use DISTINCT keyword to avoid duplicacy.

EXAMPLE:



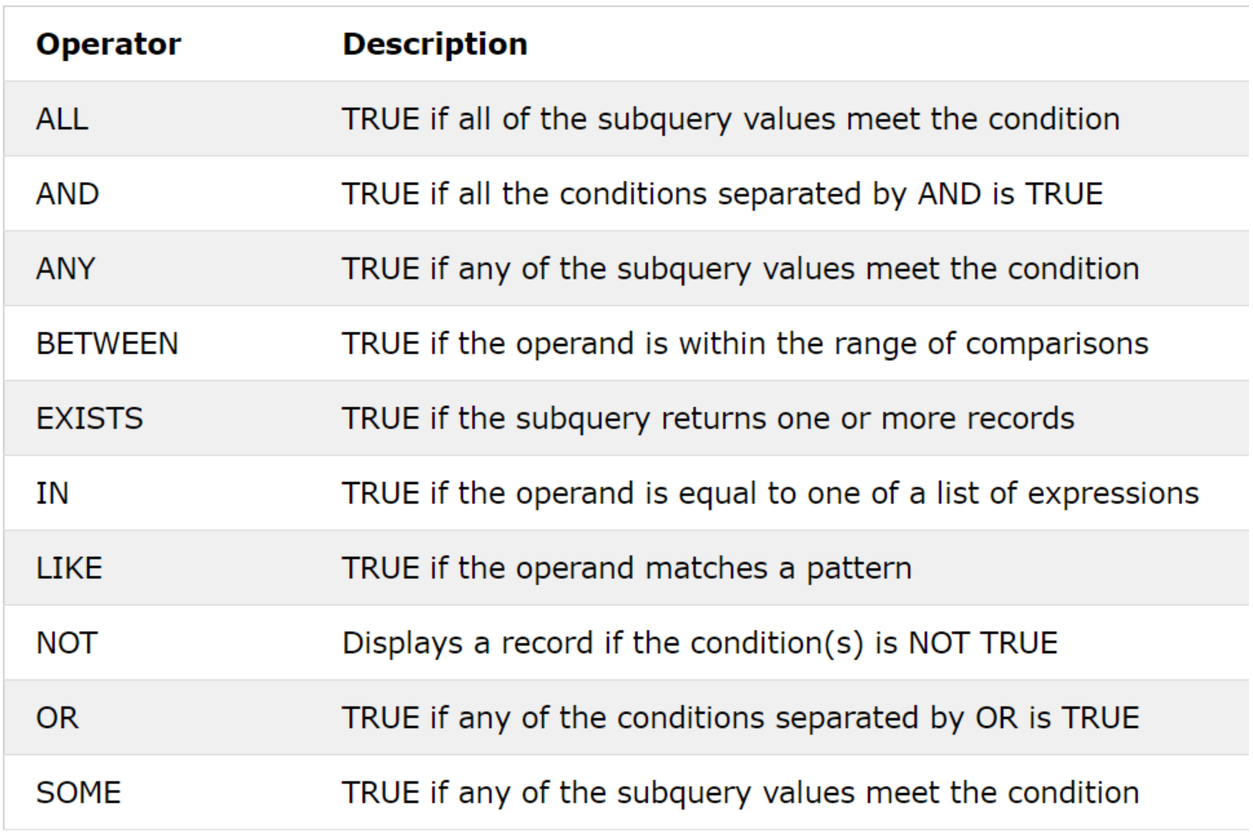
```
SELECT COUNT(DISTINCT(address)) "Number of DISTINCT addresses"
FROM student_data42;
```

RESULT:



Number of DISTINCT addresses
7

Logical operators



Operator	Description
ALL	TRUE if all of the subquery values meet the condition
AND	TRUE if all the conditions separated by AND is TRUE
ANY	TRUE if any of the subquery values meet the condition
BETWEEN	TRUE if the operand is within the range of comparisons
EXISTS	TRUE if the subquery returns one or more records
IN	TRUE if the operand is equal to one of a list of expressions
LIKE	TRUE if the operand matches a pattern
NOT	Displays a record if the condition(s) is NOT TRUE
OR	TRUE if any of the conditions separated by OR is TRUE
SOME	TRUE if any of the subquery values meet the condition

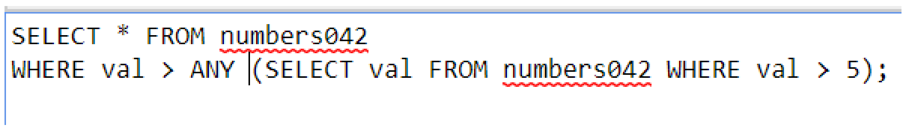
SYNTAX for ANY/ALL/SOME

Comparison operator is used with ANY/ALL/SOME

```
SELECT [column_name... | expression1 ]
FROM [table_name]
WHERE expression2 comparison_operator {ALL | ANY | SOME} ( subquery )
```

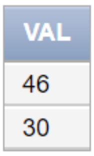
ANY ()

EXAMPLE:



```
SELECT * FROM numbers042
WHERE val > ANY (SELECT val FROM numbers042 WHERE val > 5);
```

RESULT:



VAL
46
30

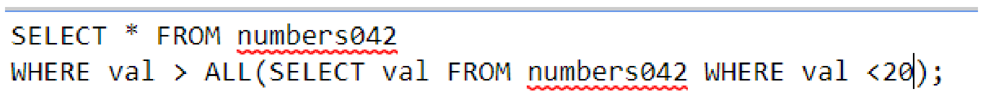
All values greater than at least one value is returned

Result of sub query: ANY (20,46,30)

Values greater than 20 returned

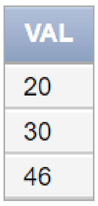
ALL ()

EXAMPLE:



```
SELECT * FROM numbers042
WHERE val > ALL(SELECT val FROM numbers042 WHERE val <20);
```

RESULT:



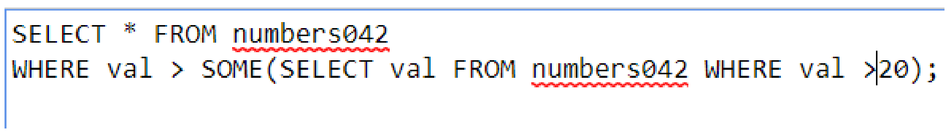
VAL
20
30
46

Values greater than all values of the result of sub query returned

Result of sub query: ALL (5,4)

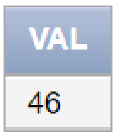
SOME ()

EXAMPLE:



```
SELECT * FROM numbers042
WHERE val > SOME(SELECT val FROM numbers042 WHERE val >20);
```

RESULT: using greater than (**>**) with SOME means greater than at least one value.



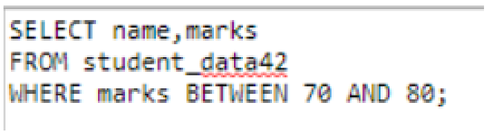
VAL
46

OBSERVATION:

SOME compare a value to each value in a list or results from a query and evaluate to true if the result of an inner query contains at least one row. SOME must match at least one row in the sub query and must be preceded by comparison operators.

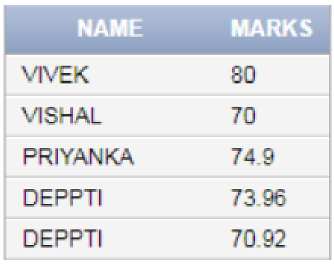
BETWEEN ()

EXAMPLE:



```
SELECT name,marks
FROM student_data42
WHERE marks BETWEEN 70 AND 80;
```

RESULT:



NAME	MARKS
VIVEK	80
VISHAL	70
PRIYANKA	74.9
DEPPTI	73.96
DEPPTI	70.92

OBSERVATION:

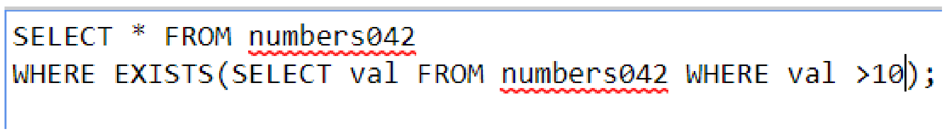
70 and 80 are also included

SYNTAX for EXISTS

```
SELECT [column_name... | expression1 ]
FROM [table_name]
WHERE [NOT] EXISTS (subquery)
```

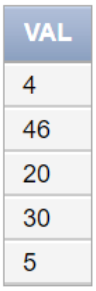
EXISTS ()

EXAMPLE:



```
SELECT * FROM numbers042
WHERE EXISTS(SELECT val FROM numbers042 WHERE val >10);
```

RESULT:



VAL
4
46
20
30
5

OBSERVATION: The EXISTS checks the existence of a result of a Sub query. The EXISTS sub query tests whether a sub query fetches at least one row. When no data is returned then this operator returns 'FALSE'.

Sub query is true so all values returned if sub query was false no data found

LIKE ()

The LIKE operator is used in a WHERE clause to search for a specified pattern in a column.

There are two wildcards used in conjunction with the LIKE operator:

- % - The percent sign represents zero, one, or multiple characters
- _ - The underscore represents a single character

OBSERVATION: UE% implies must begin with UE

%UE implies must end with UE

The `_` character looks for a presence of (any) one single character. If you search by `columnName LIKE '_abc'`, it will give you result with rows having `'aabc'`, `'xabc'`, `'1abc'`, `'#abc'` but NOT `'abc'`, `'abcc'`, `'xabcd'` and so on.

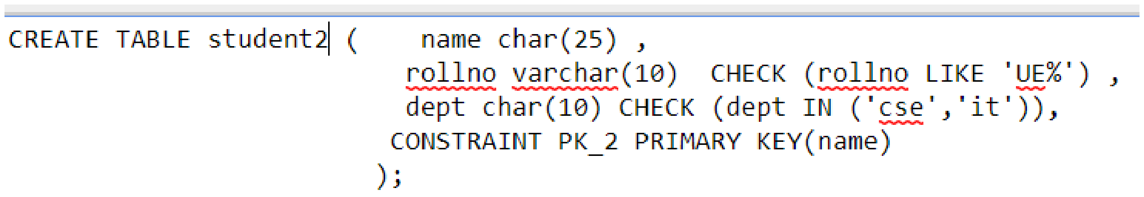
The `'%'` character is used for matching 0 or more number of characters. ThaT means, if you search by `columnName LIKE '%abc'`, it will give you result with having `'abc'`, `'aabc'`, `'xyzabc'` and so on, but no `'xyzabcd'`, `'xabcdd'` and any other string that does not end with `'abc'`.

SYNTAX FOR **IN ():**

```
SELECT column_name(s)
FROM table_name
WHERE column_name IN (‘value1’, ‘value2’, ...);
```

IN ()

EXAMPLE:



```
CREATE TABLE student2 (     name char(25) ,
                           rollno varchar(10)  CHECK (rollno LIKE 'UE%') ,
                           dept char(10) CHECK (dept IN ('cse','it')),
                          CONSTRAINT PK_2 PRIMARY KEY(name)
                         );
```

OBSERVATION: Here rollno must begin with UE and dept must be in CSE or IT else it will be a violation. The IN operator allows you to specify multiple values in a WHERE clause.

AND

The AND operator displays a record if all the conditions separated by AND is TRUE.

SYNTAX:

```
SELECT column1, column2, ...
FROM table_name
WHERE condition1 AND condition2 AND condition3 ...;
```

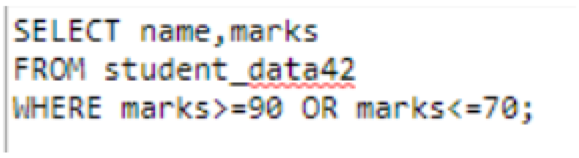
OR

The OR operator displays a record if any of the conditions separated by OR is TRUE

SYNTAX:

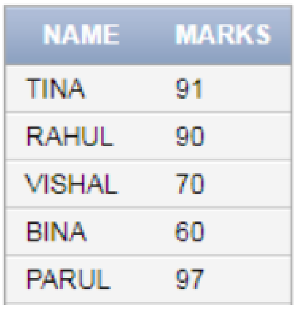
```
SELECT column1, column2, ...
FROM table_name
WHERE condition1 OR condition2 OR condition3 ...;
```

EXAMPLE:



```
SELECT name,marks
FROM student_data42
WHERE marks>=90 OR marks<=70;
```

RESULT:



NAME	MARKS
TINA	91
RAHUL	90
VISHAL	70
BINA	60
PARUL	97

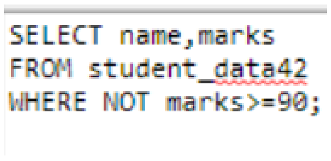
NOT

The NOT operator displays a record if the condition(s) is NOT TRUE.

SYNTAX:

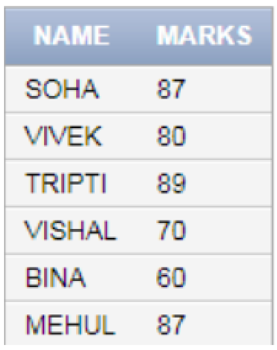
```
SELECT column1, column2, ...
FROM table_name
WHERE NOT condition;
```

EXAMPLE:



```
SELECT name,marks
FROM student_data42
WHERE NOT marks>=90;
```

RESULT:

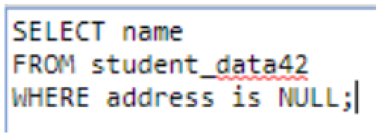


NAME	MARKS
SOHA	87
VIVEK	80
TRIPTI	89
VISHAL	70
BINA	60
MEHUL	87

Null functions

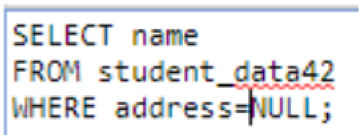
IS NULL

EXAMPLE:



```
SELECT name
FROM student_data42
WHERE address is NULL;
```

NOTE: This is incorrect

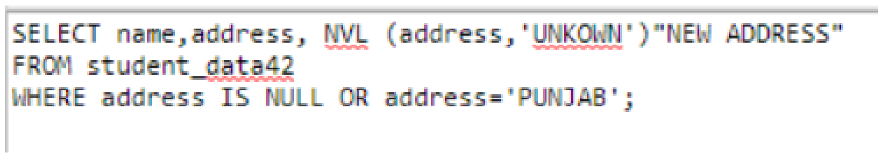


```
SELECT name
FROM student_data42
WHERE address=NULL;
```

NVL ()

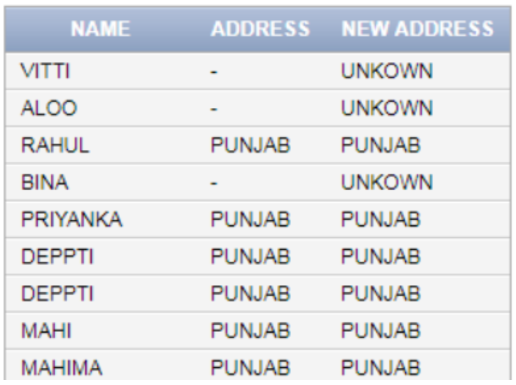
This function is used to replace NULL value with another value

EXAMPLE:



```
SELECT name,address, NVL (address,'UNKOWN')"NEW ADDRESS"
FROM student_data42
WHERE address IS NULL OR address='PUNJAB';
```

RESULT:



NAME	ADDRESS	NEW ADDRESS
VITTI	-	UNKOWN
ALOO	-	UNKOWN
RAHUL	PUNJAB	PUNJAB
BINA	-	UNKOWN
PRIYANKA	PUNJAB	PUNJAB
DEPPTI	PUNJAB	PUNJAB
DEPPTI	PUNJAB	PUNJAB
MAHI	PUNJAB	PUNJAB
MAHIMA	PUNJAB	PUNJAB

Order By, Group By and having statements

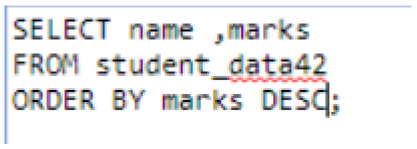
ORDER BY

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

SYNTAX:

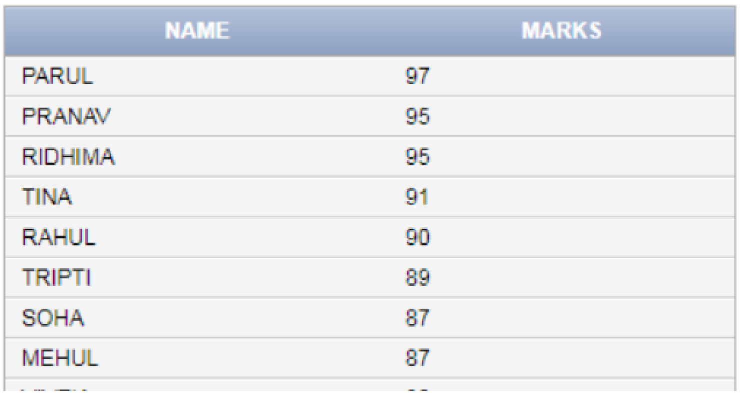
```
SELECT column1, column2, ...
FROM table_name
ORDER BY column1, column2, ... ASC|DESC;
```

EXAMPLE:



```
SELECT name ,marks
FROM student_data42
ORDER BY marks DESC;
```

RESULT:



NAME	MARKS
PARUL	97
PRANAV	95
RIDHIMA	95
TINA	91
RAHUL	90
TRIPTI	89
SOHA	87
MEHUL	87
[illegible]	[illegible]

GROUP BY

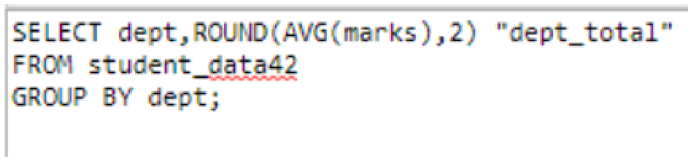
The GROUP BY statement is often used with aggregate functions (COUNT, MAX, MIN, SUM, AVG) to group the result-set by one or more columns.

SYNTAX:

```
SELECT column_name(s)
FROM table_name
```

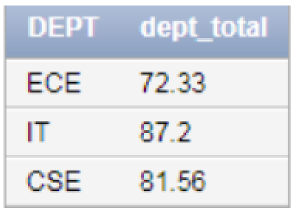
```
WHERE condition
GROUP BY column_name(s);
```

EXAMPLE:



```
SELECT dept,ROUND(AVG(marks),2) "dept_total"
FROM student_data42
GROUP BY dept;
```

RESULT:



DEPT	dept_total
ECE	72.33
IT	87.2
CSE	81.56

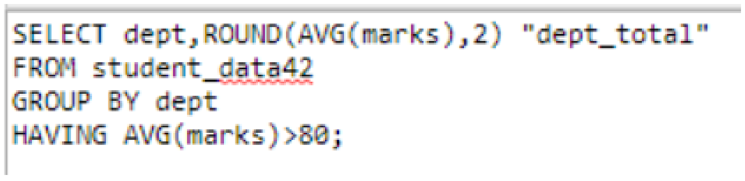
HAVING

The HAVING clause was added to SQL because the **WHERE keyword could not be used with aggregate functions.**

SYNTAX:

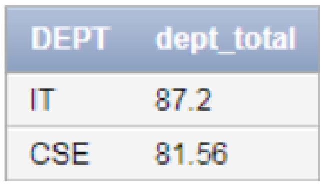
```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s)
HAVING condition;
```

EXAMPLE:



```
SELECT dept,ROUND(AVG(marks),2) "dept_total"
FROM student_data42
GROUP BY dept
HAVING AVG(marks)>80;
```

RESULT:



DEPT	dept_total
IT	87.2
CSE	81.56

Joins

An SQL **join** clause combines columns from one or more tables in a relational database

Let the two tables be:

```
CREATE TABLE students_42 (name varchar(20), rollno varchar(15),
                          department varchar(15),marks number (3));
```

NAME	ROLLNO	DEPARTMENT	MARKS
RAHUL	UE177	CSE	94
PREETI	UE176	CSE	93
NEETI	UE164	IT	95
NIA	UE165	IT	80
MEETHIA	UE166	IT	86
VIVEK	UE153	BIO	87
YUVI	UE154	BIO	89
MEHAK	UE155	BIO	84
PURU	UE174	CSE	90

```
CREATE TABLE student_ (rollno varchar(15), company varchar(15), join_date date);
```

ROLLNO	COMPANY	JOIN_DATE
UE154	ZS	04/12/2016
UE155	MAHINDRA	06/12/2016
UE156	MAHINDRA	12/12/2016
UE166	INFOSYS	12/12/2016
UE167	INFOSYS	11/04/2016
UE177	ZS	06/05/2016
UE174	DOCIN	06/06/2016
UE169	INFOSYS	11/05/2016

NOTE: Date is entered in MM/DD/YYYY format

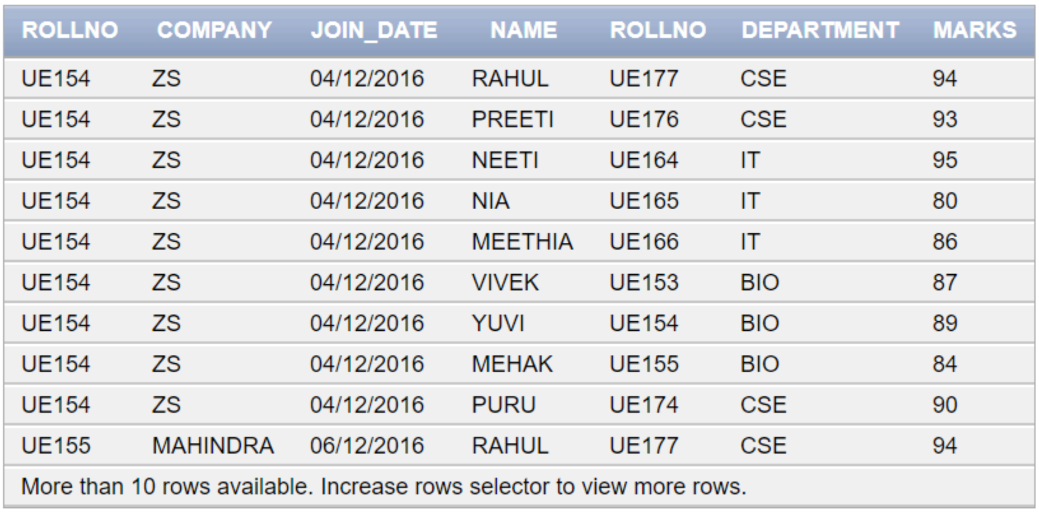
CROSS JOIN

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 as Cartesian Product.

EXAMPLE:

```
SELECT *
FROM student_  CROSS JOIN students_42  ;
```

RESULT:



ROLLNO	COMPANY	JOIN_DATE	NAME	ROLLNO	DEPARTMENT	MARKS
UE154	ZS	04/12/2016	RAHUL	UE177	CSE	94
UE154	ZS	04/12/2016	PREETI	UE176	CSE	93
UE154	ZS	04/12/2016	NEETI	UE164	IT	95
UE154	ZS	04/12/2016	NIA	UE165	IT	80
UE154	ZS	04/12/2016	MEETHIA	UE166	IT	86
UE154	ZS	04/12/2016	VIVEK	UE153	BIO	87
UE154	ZS	04/12/2016	YUVI	UE154	BIO	89
UE154	ZS	04/12/2016	MEHAK	UE155	BIO	84
UE154	ZS	04/12/2016	PURU	UE174	CSE	90
UE155	MAHINDRA	06/12/2016	RAHUL	UE177	CSE	94

More than 10 rows available. Increase rows selector to view more rows.

NOTE:

```
SELECT *
FROM student_  , students_42 ;
```

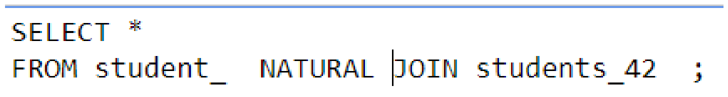
By default this statement does cross join

NATURAL JOIN

The SQL NATURAL JOIN is a type of JOIN in which columns with the same name of associated tables will appear once only.

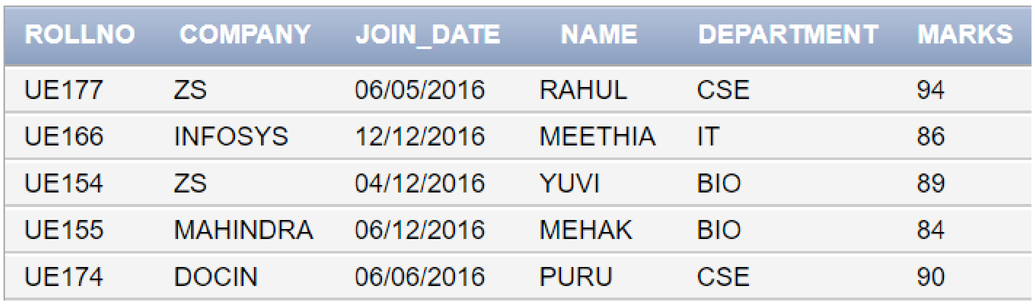
- The associated tables have one or more pairs of identically named columns.
- The columns must be the same data type.
- Don’t use where clause in a natural join.

EXAMPLE:



```
SELECT *
FROM student_  NATURAL JOIN students_42  ;
```

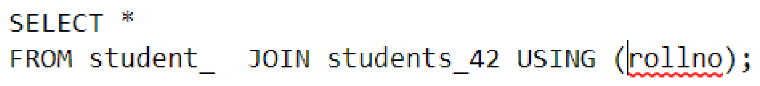
RESULT:



ROLLNO	COMPANY	JOIN_DATE	NAME	DEPARTMENT	MARKS
UE177	ZS	06/05/2016	RAHUL	CSE	94
UE166	INFOSYS	12/12/2016	MEETHIA	IT	86
UE154	ZS	04/12/2016	YUVI	BIO	89
UE155	MAHINDRA	06/12/2016	MEHAK	BIO	84
UE174	DOCIN	06/06/2016	PURU	CSE	90

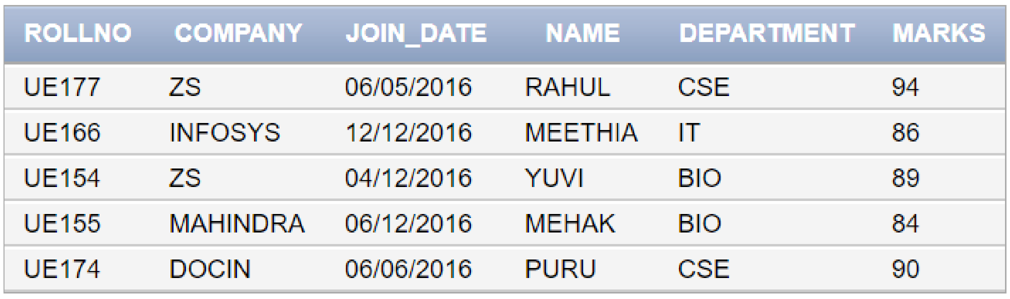
Implementing natural join with keyword using

EXAMPLE:



```
SELECT *
FROM student_  JOIN students_42 USING (rollno);
```

RESULT:

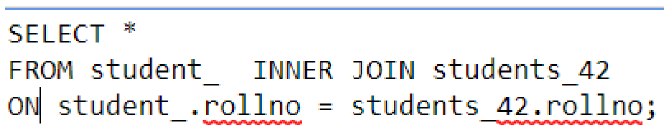


ROLLNO	COMPANY	JOIN_DATE	NAME	DEPARTMENT	MARKS
UE177	ZS	06/05/2016	RAHUL	CSE	94
UE166	INFOSYS	12/12/2016	MEETHIA	IT	86
UE154	ZS	04/12/2016	YUVI	BIO	89
UE155	MAHINDRA	06/12/2016	MEHAK	BIO	84
UE174	DOCIN	06/06/2016	PURU	CSE	90

INNER JOIN

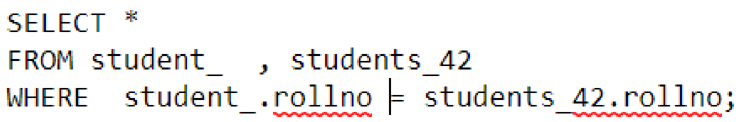
The INNER JOIN selects all rows from both participating tables as long as there is a match between the columns. An SQL INNER JOIN is same as JOIN clause, combining rows from two or more tables.

EXAMPLE:



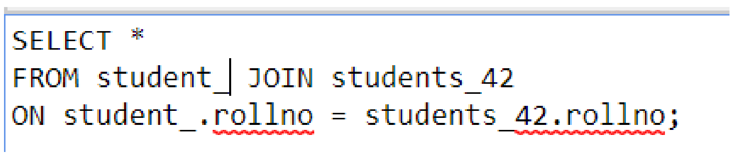
```
SELECT *
FROM student_  INNER JOIN students_42
ON student_.rollno = students_42.rollno;
```

OR



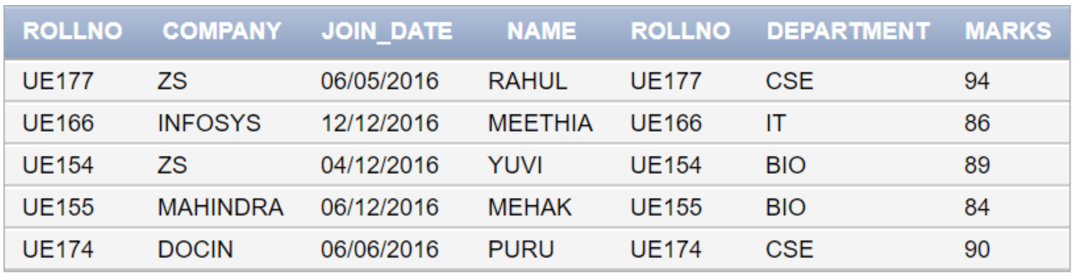
```
SELECT *
FROM student_  , students_42
WHERE  student_.rollno = students_42.rollno;
```

OR



```
SELECT *
FROM student_ JOIN students_42
ON student_.rollno = students_42.rollno;
```

RESULT:



ROLLNO	COMPANY	JOIN_DATE	NAME	ROLLNO	DEPARTMENT	MARKS
UE177	ZS	06/05/2016	RAHUL	UE177	CSE	94
UE166	INFOSYS	12/12/2016	MEETHIA	UE166	IT	86
UE154	ZS	04/12/2016	YUVI	UE154	BIO	89
UE155	MAHINDRA	06/12/2016	MEHAK	UE155	BIO	84
UE174	DOCIN	06/06/2016	PURU	UE174	CSE	90

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.

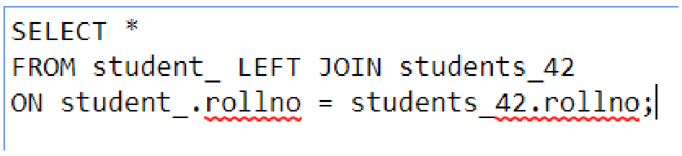
The subtypes of SQL OUTER JOIN

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

LEFT OUTER JOIN

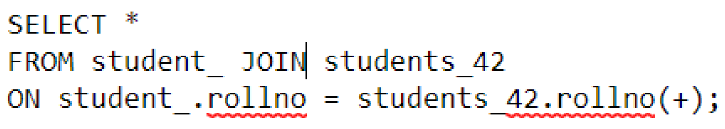
The SQL LEFT JOIN (specified with the keywords LEFT JOIN and ON) joins two tables and fetches all matching rows of two tables for which the SQL-expression is true, plus rows from the first table that do not match any row in the second table.

EXAMPLE:



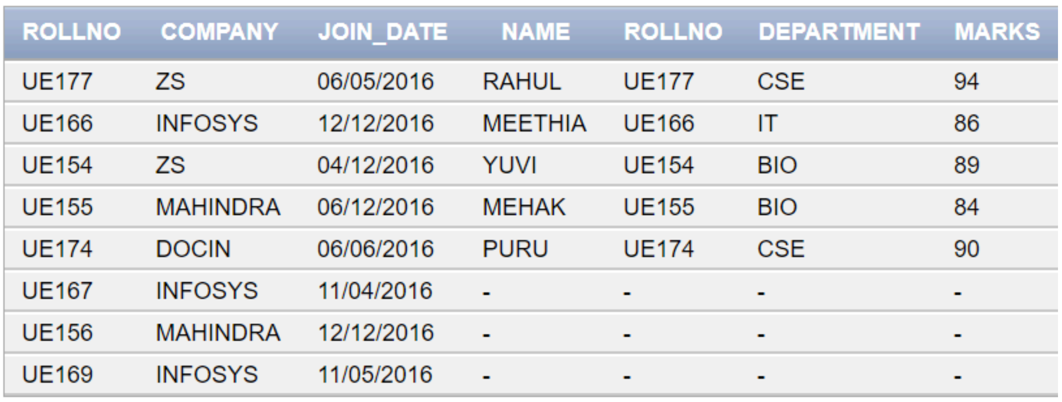
```
SELECT *
FROM student_ LEFT JOIN students_42
ON student_.rollno = students_42.rollno;
```

OR



```
SELECT *
FROM student_ JOIN students_42
ON student_.rollno = students_42.rollno(+);
```

RESULT:



ROLLNO	COMPANY	JOIN_DATE	NAME	ROLLNO	DEPARTMENT	MARKS
UE177	ZS	06/05/2016	RAHUL	UE177	CSE	94
UE166	INFOSYS	12/12/2016	MEETHIA	UE166	IT	86
UE154	ZS	04/12/2016	YUVI	UE154	BIO	89
UE155	MAHINDRA	06/12/2016	MEHAK	UE155	BIO	84
UE174	DOCIN	06/06/2016	PURU	UE174	CSE	90
UE167	INFOSYS	11/04/2016	-	-	-	-
UE156	MAHINDRA	12/12/2016	-	-	-	-
UE169	INFOSYS	11/05/2016	-	-	-	-

NOTE: The (+) after the students_42.rollno field indicates that, if a rollno value in the first table does not exist in the second table, all fields in the second table will be displayed as NULL in the result set.

RIGHT OUTER JOIN

The SQL RIGHT JOIN , joins two tables and fetches rows based on a condition, which is matching in both the tables, and the unmatched rows will also be available from the table written after the JOIN clause

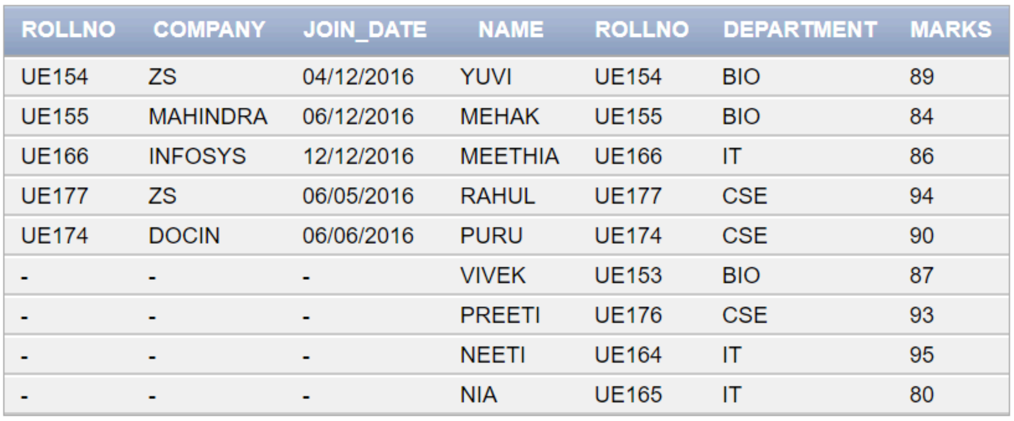
EXAMPLE:

```
SELECT *
FROM student_ RIGHT JOIN students_42
ON student_.rollno = students_42.rollno;
```

OR

```
SELECT *
FROM student_  JOIN students_42
ON student_.rollno(+) = students_42.rollno;
```

RESULT:



ROLLNO	COMPANY	JOIN_DATE	NAME	ROLLNO	DEPARTMENT	MARKS
UE154	ZS	04/12/2016	YUVI	UE154	BIO	89
UE155	MAHINDRA	06/12/2016	MEHAK	UE155	BIO	84
UE166	INFOSYS	12/12/2016	MEETHIA	UE166	IT	86
UE177	ZS	06/05/2016	RAHUL	UE177	CSE	94
UE174	DOCIN	06/06/2016	PURU	UE174	CSE	90
-	-	-	VIVEK	UE153	BIO	87
-	-	-	PREETI	UE176	CSE	93
-	-	-	NEETI	UE164	IT	95
-	-	-	NIA	UE165	IT	80

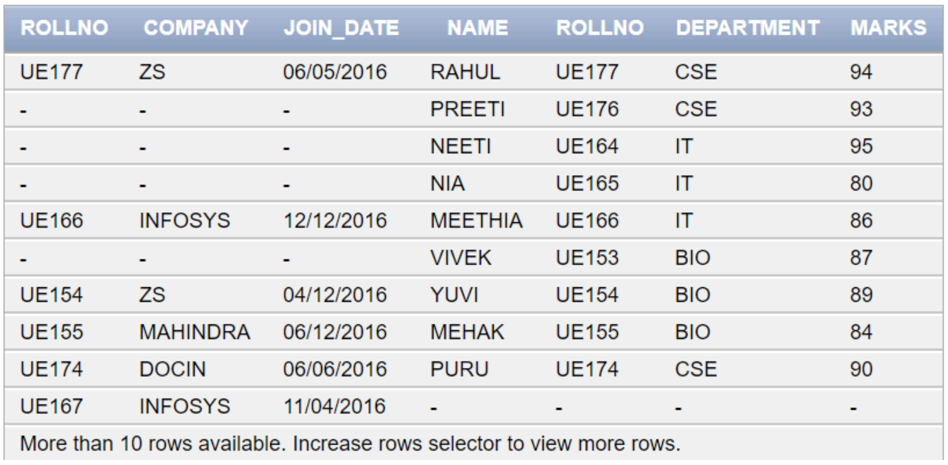
FULL OUTER JOIN

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.

EXAMPLE:

```
SELECT *
FROM student_ FULL JOIN students_42
ON student_.rollno = students_42.rollno;
```

RESULT:



ROLLNO	COMPANY	JOIN_DATE	NAME	ROLLNO	DEPARTMENT	MARKS
UE177	ZS	06/05/2016	RAHUL	UE177	CSE	94
-	-	-	PREETI	UE176	CSE	93
-	-	-	NEETI	UE164	IT	95
-	-	-	NIA	UE165	IT	80
UE166	INFOSYS	12/12/2016	MEETHIA	UE166	IT	86
-	-	-	VIVEK	UE153	BIO	87
UE154	ZS	04/12/2016	YUVI	UE154	BIO	89
UE155	MAHINDRA	06/12/2016	MEHAK	UE155	BIO	84
UE174	DOCIN	06/06/2016	PURU	UE174	CSE	90
UE167	INFOSYS	11/04/2016	-	-	-	-

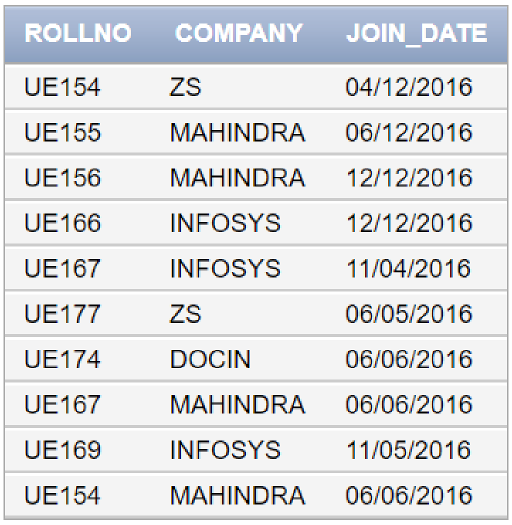
More than 10 rows available. Increase rows selector to view more rows.

NOTE: Using **WHERE** keyword instead of **ON** doesn’t work

You can **USING** keyword instead of ON and WHERE as shown in the case of NATURAL JOIN

Set operation on relation

Let the table be



ROLLNO	COMPANY	JOIN_DATE
UE154	ZS	04/12/2016
UE155	MAHINDRA	06/12/2016
UE156	MAHINDRA	12/12/2016
UE166	INFOSYS	12/12/2016
UE167	INFOSYS	11/04/2016
UE177	ZS	06/05/2016
UE174	DOCIN	06/06/2016
UE167	MAHINDRA	06/06/2016
UE169	INFOSYS	11/05/2016
UE154	MAHINDRA	06/06/2016

UNION

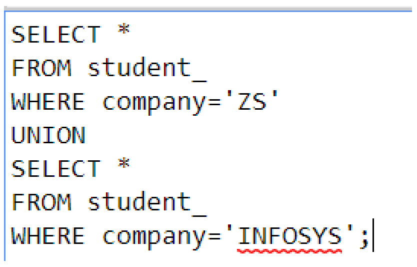
The SQL UNION operator combines the results of two or more queries and makes a result set which includes fetched rows from the participating queries in the UNION. The two tables must be compatible as follow:

- Each SELECT statement within UNION must have the same number of columns
- The columns must also have similar data types

SYNTAX:

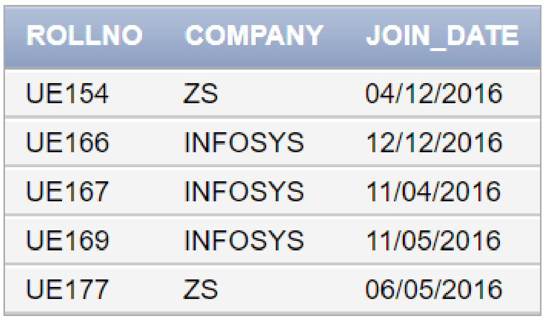
```
SELECT column_name(s) FROM table1
UNION
SELECT column_name(s) FROM table2;
```

EXAMPLE:



```
SELECT *
FROM student_
WHERE company='ZS'
UNION
SELECT *
FROM student_
WHERE company='INFOSYS';
```

RESULT:

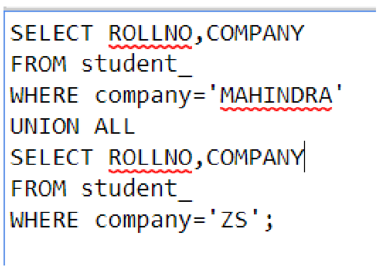


ROLLNO	COMPANY	JOIN_DATE
UE154	ZS	04/12/2016
UE166	INFOSYS	12/12/2016
UE167	INFOSYS	11/04/2016
UE169	INFOSYS	11/05/2016
UE177	ZS	06/05/2016

NOTE: The UNION operator **selects only distinct values** by default. To allow duplicate values, use **UNION ALL**:

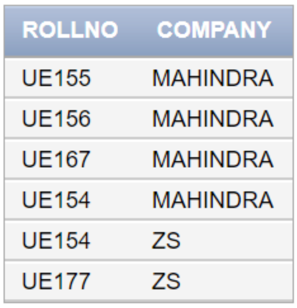
```
SELECT column_name(s) FROM table1
UNION ALL
SELECT column_name(s) FROM table2;
```

EXAMPLE:



```
SELECT ROLLNO,COMPANY
FROM student_
WHERE company='MAHINDRA'
UNION ALL
SELECT ROLLNO,COMPANY
FROM student_
WHERE company='ZS';
```

RESULT:



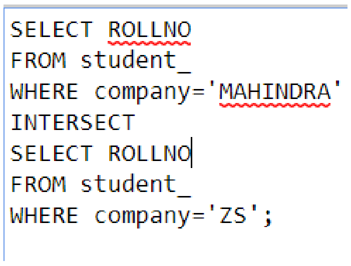
ROLLNO	COMPANY
UE155	MAHINDRA
UE156	MAHINDRA
UE167	MAHINDRA
UE154	MAHINDRA
UE154	ZS
UE177	ZS

INTERSECT

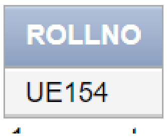
The resultant relation consists of tupples belonging to both relations.

The two tables must be compatible

EXAMPLE:



```
SELECT ROLLNO
FROM student_
WHERE company='MAHINDRA'
INTERSECT
SELECT ROLLNO
FROM student_
WHERE company='ZS';
```

RESULT: 

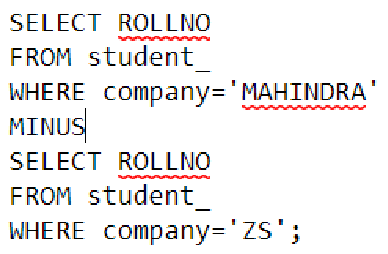
ROLLNO
UE154

MINUS

The relation contains tupples which are present in the first relation but not in the second relation.

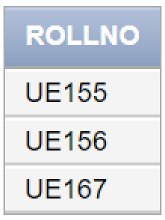
The two tables must be compatible

EXAMPLE:



```
SELECT ROLLNO
FROM student_
WHERE company='MAHINDRA'
MINUS
SELECT ROLLNO
FROM student_
WHERE company='ZS';
```

RESULT:



ROLLNO
UE155
UE156
UE167

NOTE: in order to retain duplicates use **ALL** keywords with set operators

Views

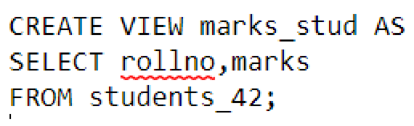
Views are the virtual relations created on response to a query and contain conceptually the result of the query. The virtual relation is not precompiled and stored, but instead is computed by executing the query. Any such relation that is not part of the logical model, but is made visible to a user as a virtual relation, is called a **view**.

Materialized Views Certain database systems allow view relations to be stored, but they make sure that, if the actual relations used in the view definition change, the view is kept up-to-date. Such views are called **materialized views**.

SYNTAX:

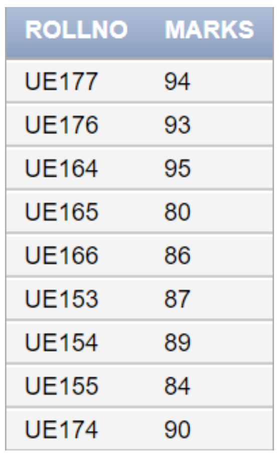
```
CREATE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

EXAMPLE:



```
CREATE VIEW marks_stud AS
SELECT rollno,marks
FROM students_42;
```

RESULT: SELECT * FROM marks_stud;

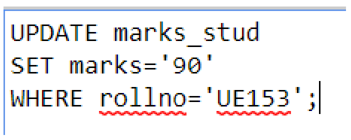


ROLLNO	MARKS
UE177	94
UE176	93
UE164	95
UE165	80
UE166	86
UE153	87
UE154	89
UE155	84
UE174	90

Updatable View

A change in view is visible in the master table and similarly change in master table is visible in view.

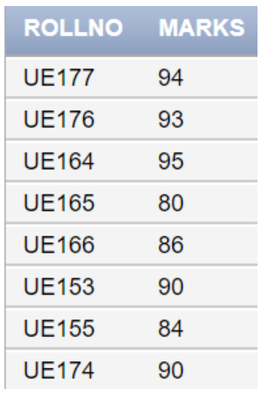
EXAMPLE:



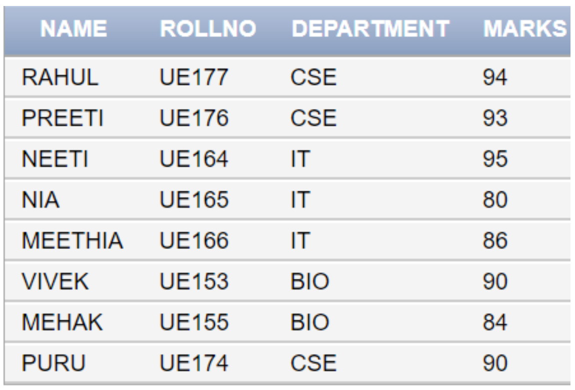
```
UPDATE marks_stud
SET marks='90'
WHERE rollno='UE153';
```

RESULT: earlier marks were 87

View: Master table:

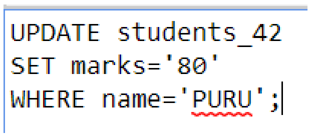


ROLLNO	MARKS
UE177	94
UE176	93
UE164	95
UE165	80
UE166	86
UE153	90
UE155	84
UE174	90



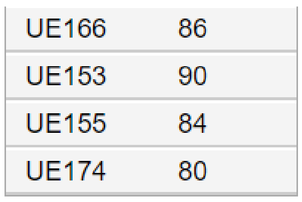
NAME	ROLLNO	DEPARTMENT	MARKS
RAHUL	UE177	CSE	94
PREETI	UE176	CSE	93
NEETI	UE164	IT	95
NIA	UE165	IT	80
MEETHIA	UE166	IT	86
VIVEK	UE153	BIO	90
MEHAK	UE155	BIO	84
PURU	UE174	CSE	90

updating master table :

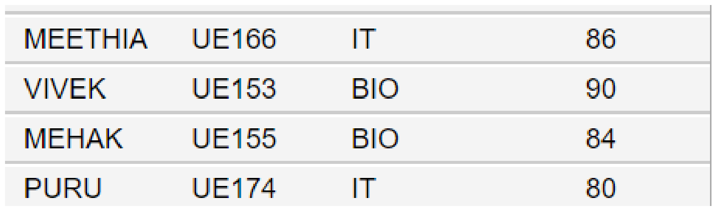


```
UPDATE students_42
SET marks='80'
WHERE name='PURU';
```

View: Master table:



UE166	86
UE153	90
UE155	84
UE174	80



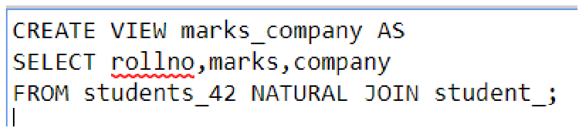
MEETHIA	UE166	IT	86
VIVEK	UE153	BIO	90
MEHAK	UE155	BIO	84
PURU	UE174	IT	80

NOTE: You can check if a view in a database in updatable by querying the is_updatable column from the views table in the information_schema database.

```
SELECT
    table_name, is_updatable
FROM
    information_schema.views
WHERE
    table_schema = 'classicmodels';
```

IMPORTANT: When view depends on more than one table then view is affected by changes in master table **but changing the view does not affect the master table**

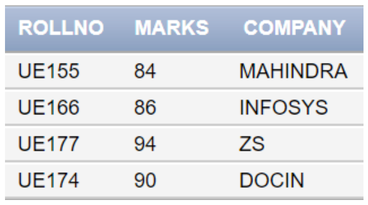
EXAMPLE:



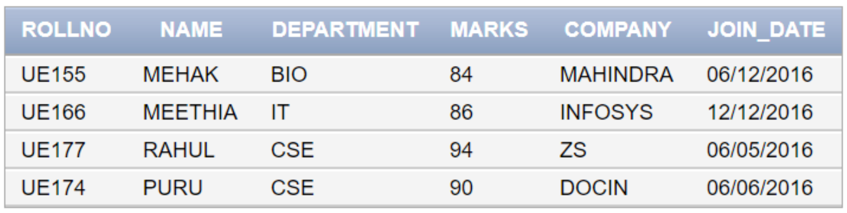
```
CREATE VIEW marks_company AS
SELECT rollno,marks,company
FROM students_42 NATURAL JOIN student_;
```

RESULT:

View: Master table:

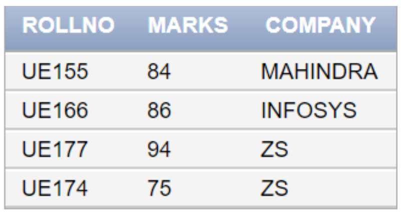


ROLLNO	MARKS	COMPANY
UE155	84	MAHINDRA
UE166	86	INFOSYS
UE177	94	ZS
UE174	90	DOCIN

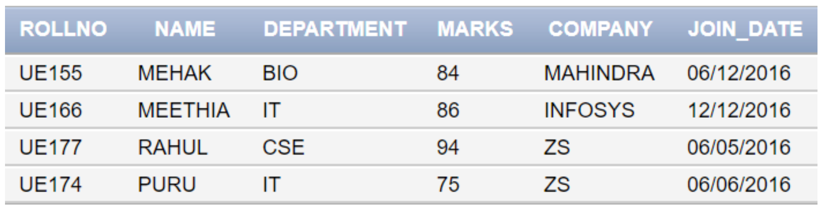


ROLLNO	NAME	DEPARTMENT	MARKS	COMPANY	JOIN_DATE
UE155	MEHAK	BIO	84	MAHINDRA	06/12/2016
UE166	MEETHIA	IT	86	INFOSYS	12/12/2016
UE177	RAHUL	CSE	94	ZS	06/05/2016
UE174	PURU	CSE	90	DOCIN	06/06/2016

After few changes in master table:



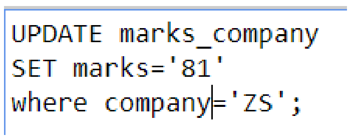
ROLLNO	MARKS	COMPANY
UE155	84	MAHINDRA
UE166	86	INFOSYS
UE177	94	ZS
UE174	75	ZS



ROLLNO	NAME	DEPARTMENT	MARKS	COMPANY	JOIN_DATE
UE155	MEHAK	BIO	84	MAHINDRA	06/12/2016
UE166	MEETHIA	IT	86	INFOSYS	12/12/2016
UE177	RAHUL	CSE	94	ZS	06/05/2016
UE174	PURU	IT	75	ZS	06/06/2016

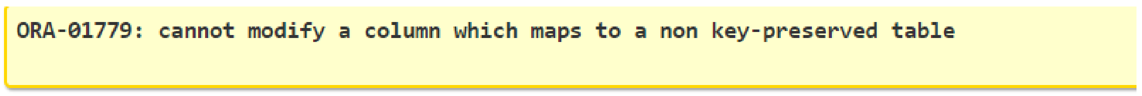
change in view:

EXAMPLE:



```
UPDATE marks_company
SET marks='81'
where company='ZS';
```

RESULT: master table could not be affected

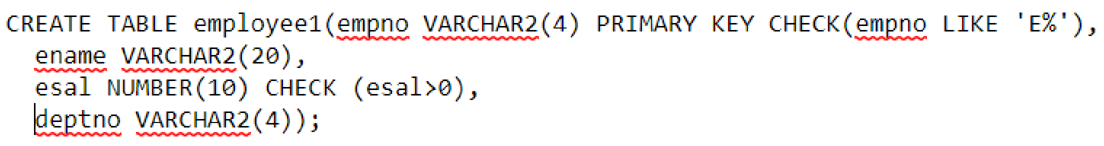


```
ORA-01779: cannot modify a column which maps to a non key-preserved table
```

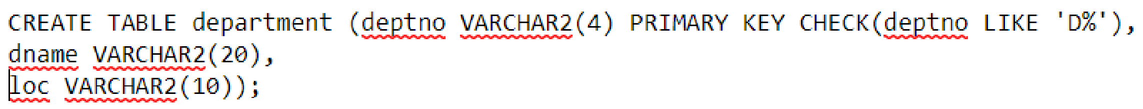
Nested Queries

A **Subquery** or **Inner query** or a **Nested query** is a **query** within another **SQL query** and embedded within the WHERE clause. A **subquery** is used to return data that will be used in the main **query** as a condition to further restrict the data to be retrieved. Subqueries are nested, when the subquery is executed first,and its results are inserted into Where clause of the main query.

Given two tables employee1 and department:



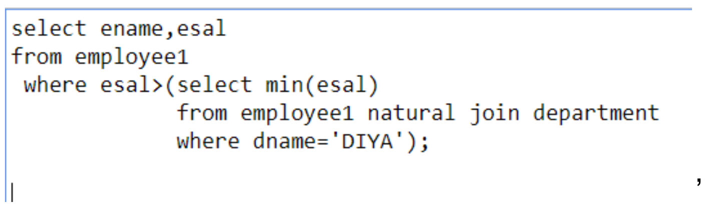
```
CREATE TABLE employee1(empno VARCHAR2(4) PRIMARY KEY CHECK(empno LIKE 'E%'),
  ename VARCHAR2(20),
  esal NUMBER(10) CHECK (esal>0),
  deptno VARCHAR2(4));
```



```
CREATE TABLE department (deptno VARCHAR2(4) PRIMARY KEY CHECK(deptno LIKE 'D%'),
dname VARCHAR2(20),
loc VARCHAR2(10));
```

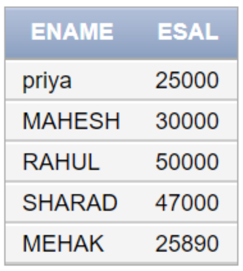
Q .1. List name and salary of employee whose salary is greater than minimum salary dname= 'DIYA

QUERY:



```
select ename,esal
from employee1
 where esal>(select min(esal)
             from employee1 natural join department
             where dname='DIYA');
```

RESULT:



ENAME	ESAL
priya	25000
MAHESH	30000
RAHUL	50000
SHARAD	47000
MEHAK	25890

Q.2. List name and salary of employee whose salary is greater than salary of dept with name=MINS.

QUERY:

```
select ename,esal
from employee1
 where esal>(select max(esal)
             from employee1 natural join department
             where dname='MINS');
```

RESULT:

ENAME	ESAL
MAHESH	30000
RAHUL	50000
SHARAD	47000
MEHAK	25890

Q.3. List details of the department where empno=E1.

QUERY:

```
select deptno,dname,loc
from department
 where deptno IN(select deptno
            from employee1
            where empno='E1');
```

RESULT:

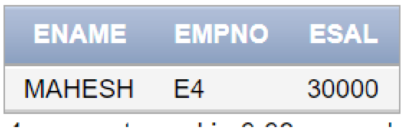
DEPTNO	DNAME	LOC
D03	MINS	MIZORAM

Q.4. List employees belonging to the dept with name INNI.

QUERY:

```
select ename,empno,esal
from employee1
where deptno IN(select deptno
             from department
             where dname='INNI');
```

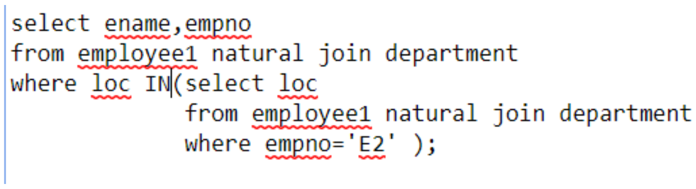
RESULT:



ENAME	EMPNO	ESAL
MAHESH	E4	30000

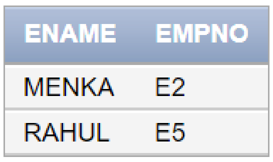
Q.5.List name of emp has the same department loc as emp no E2

QUERY:



```
select ename,empno
from employee1 natural join department
where loc IN(select loc
             from employee1 natural join department
             where empno='E2' );
```

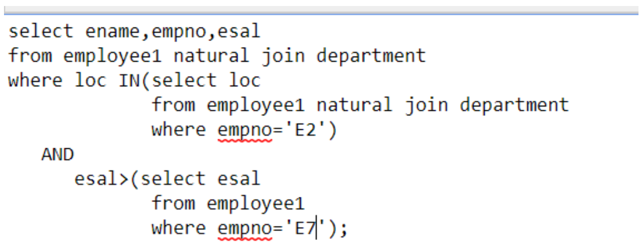
RESULT:



ENAME	EMPNO
MENKA	E2
RAHUL	E5

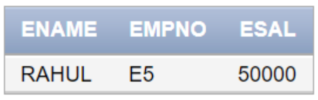
Q.6.List name of emp has the same department loc as emp no E2 and sal is greater than E7

QUERY:



```
select ename,empno,esal
from employee1 natural join department
where loc IN(select loc
             from employee1 natural join department
             where empno='E2')
   AND
      esal>(select esal
             from employee1
             where empno='E7');
```

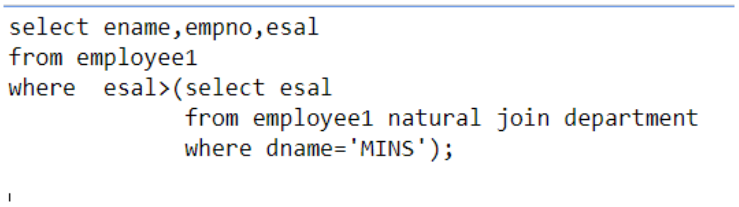
RESULT:



ENAME	EMPNO	ESAL
RAHUL	E5	50000

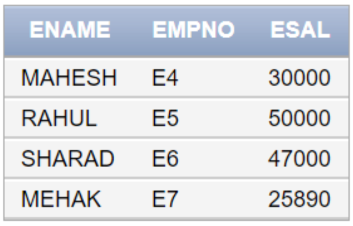
Q.7. List name and salary of employee who get salary greater than the department MINS.

QUERY:



```
select ename,empno,esal
from employee1
where  esal>(select esal
             from employee1 natural join department
             where dname='MINS');
```

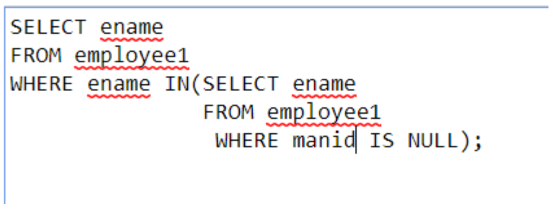
RESULT:



ENAME	EMPNO	ESAL
MAHESH	E4	30000
RAHUL	E5	50000
SHARAD	E6	47000
MEHAK	E7	25890

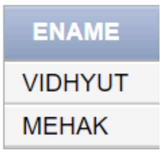
Q.8. List the employee who do not manage any employee.

QUERY:



```
SELECT ename
FROM employee1
WHERE ename IN(SELECT ename
               FROM employee1
                WHERE manid IS NULL);
```

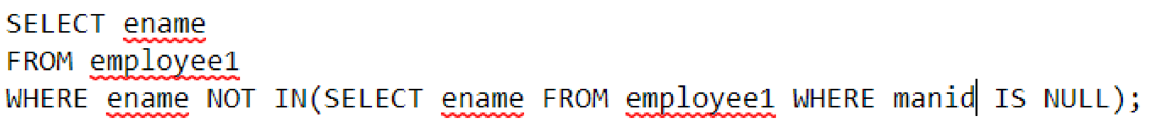
RESULT:



ENAME
VIDHYUT
MEHAK

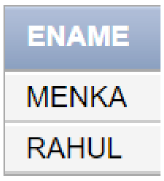
Q.9. List all emp who have at least 1 person reporting to them.

QUERY:



```
SELECT ename
FROM employee1
WHERE ename NOT IN(SELECT ename FROM employee1 WHERE manid IS NULL);
```

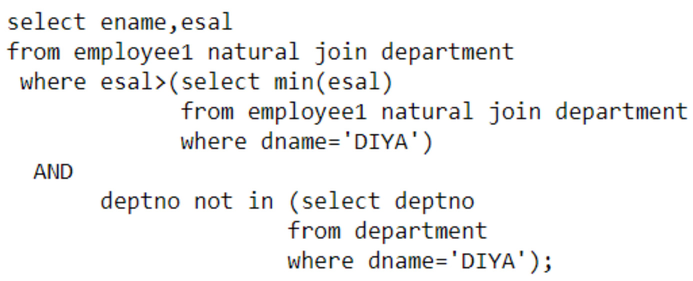
RESULT:



ENAME
MENKA
RAHUL

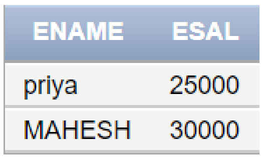
Q.10.List employee whose salary is greater than min salary of DIYA department and they should not be of same department.

QUERY:



```
select ename,esal
from employee1 natural join department
 where esal>(select min(esal)
             from employee1 natural join department
             where dname='DIYA')
  AND
       deptno not in (select deptno
                      from department
                      where dname='DIYA');
```

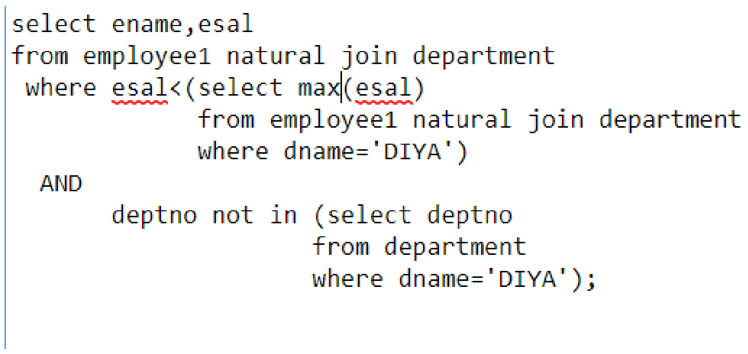
RESULT:



ENAME	ESAL
priya	25000
MAHESH	30000

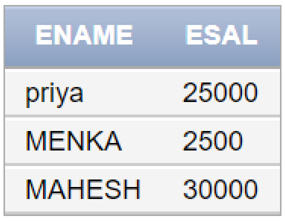
Q.11. List employee whose salary is less than max salary of DIYA department and they should not be of same department.

QUERY:



```
select ename,esal
from employee1 natural join department
 where esal<(select max(esal)
             from employee1 natural join department
             where dname='DIYA')
  AND
       deptno not in (select deptno
                      from department
                      where dname='DIYA');
```

RESULT:



ENAME	ESAL
priya	25000
MENKA	2500
MAHESH	30000

Correlated Queries

A **Correlated Subquery** is one that is **executed after the outer query is executed**. So correlated subqueries take an approach opposite to that of normal subqueries. The correlated subquery execution is as follows:

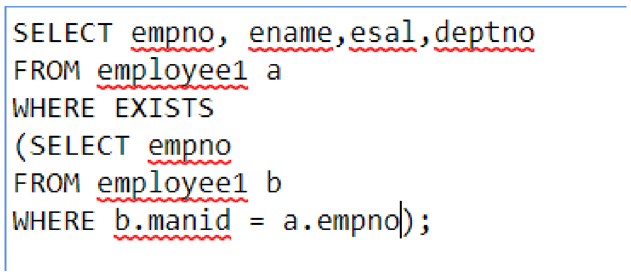
-The outer query receives a row.
- the subquery references the table in the outer query to fetch result
-The process is repeated for all rows.

Correlated Subqueries differ from the normal subqueries in that the nested SELECT statement referes back to the table in the first SELECT statement.

NOTE: Correlated Queries are also called as **Synchronized queries**

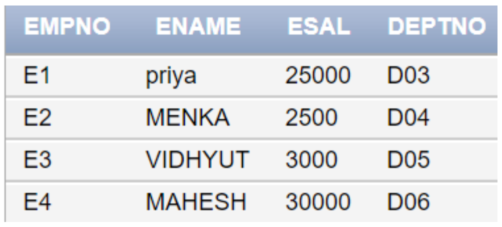
It is not recommended to use Correlated Subqueries as it slows down the performance

QUERY:



```
SELECT empno, ename,esal,deptno
FROM employee1 a
WHERE EXISTS
(SELECT empno
FROM employee1 b
WHERE b.manid = a.empno);
```

RESULT:



EMPNO	ENAME	ESAL	DEPTNO
E1	priya	25000	D03
E2	MENKA	2500	D04
E3	VIDHYUT	3000	D05
E4	MAHESH	30000	D06

Advanced SQL

Indexes

Indexes are used to retrieve data from the database very fast. 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, only create indexes on columns that will be frequently searched against.

The CREATE INDEX Command

The basic syntax of a **CREATE INDEX** is as follows:

```
CREATE INDEX index_name ON table_name;
```

Single-Column Indexes

A single-column index is created based on only one table column. The basic syntax is as follows.

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

Unique Indexes

unique index does not allow any duplicate values to be inserted into the table. The basic syntax is as follows.

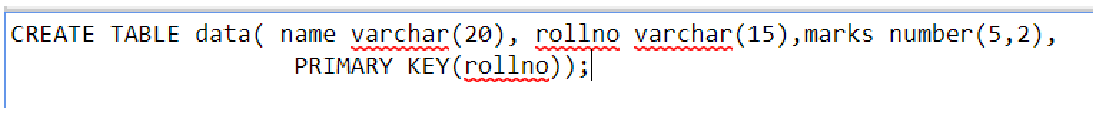
```
CREATE UNIQUE INDEX index_name
on table_name (column_name);
```

Composite Indexes

A composite index is an index on two or more columns of a table. Its basic syntax is as follows.

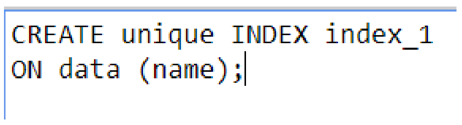
```
CREATE INDEX index_name
on table_name (column1, column2);
```

Let table be:



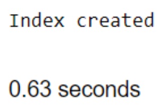
```
CREATE TABLE data( name varchar(20), rollno varchar(15),marks number(5,2),
                   PRIMARY KEY(rollno));
```

EXAMPLE:



```
CREATE unique INDEX index_1
ON data (name);
```

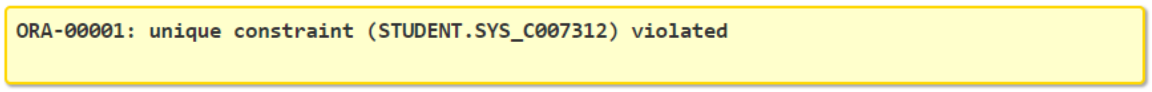
RESULT:



```
Index created.
```

0.63 seconds

OBSERVATION: on inserting same name in unique index following violation



```
ORA-00001: unique constraint (STUDENT.SYS_C007312) violated
```

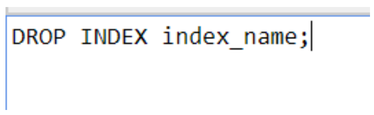
<u>The DROP INDEX Command</u>

An index can be dropped using SQL **DROP** command.

SYNTAX:

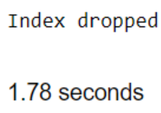
```
DROP INDEX index_name;
```

EXAMPLE:



```
DROP INDEX index_name;
```

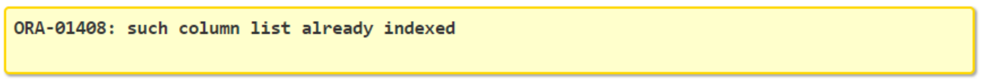
RESULT:



```
Index dropped.
```

1.78 seconds

OBSERVATION: When index is created on primary key it says



```
ORA-01408: such column list already indexed
```

Implicit indexes are indexes that are automatically created by the database server when an object is created. Indexes are automatically created for primary key constraints and unique constraints.

NOTE : indexes are never executed explicitly they are pointers.

Sequences

Sequence is a feature supported by some database systems to produce unique values on demand ,it automatically increments the column value by 1 each time a new record is entered into the table and has some extra features.

SYNTAX:

```
CREATE Sequence sequence-name
Start with initial-value
Increment by increment-value
maxvalue maximum-value
cycle|nocycle
cache number
ORDER|NOORDER;
```

INITIAL-VALUE

Specifies the starting value of the Sequence,

INCREMENT-VALUE

Is the value by which sequence will be incremented

MAXVALUE

Specifies the maximum value until which sequence will increment itself.

CYCLE

Specifies that if the maximum value exceeds the set limit, sequence will restart its cycle from the beginning.

NO CYCLE

Specifies that if sequence exceeds **maxvalue** an error will be thrown.

CACHE

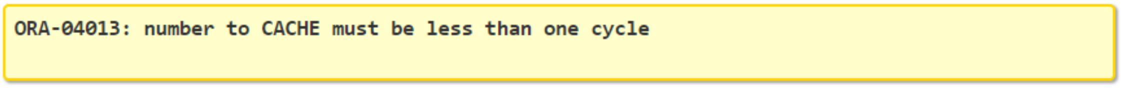
Specify how many values of the sequence the database preallocates and keeps in memory for faster access. This integer value can have 28 or fewer digits. The minimum value for this parameter is 2. For sequences that cycle, this value must be less than the number of values in the cycle.

OBSERVATION: You cannot cache more values than will fit in a given cycle of sequence numbers. Therefore, the maximum value allowed for CACHE must be less than the value determined by the following formula:

```
(CEIL (MAXVALUE - MINVALUE)) / ABS (INCREMENT)
```

If a system failure occurs, then all cached sequence values that have not been used in committed DML statements are lost. The potential number of lost values is equal to the value of the CACHE parameter.

Otherwise violation as



```
ORA-04013: number to CACHE must be less than one cycle
```

NOCACHE

Specify NOCACHE to indicate that values of the sequence are not preallocated. If you omit both CACHE and NOCACHE,

OBSERVATION: Then the database caches 20 sequence numbers by default.

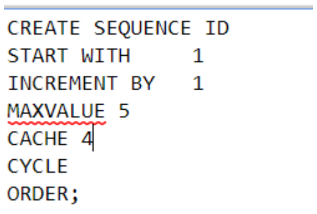
ORDER

Specify ORDER to guarantee that sequence numbers are generated in order of request. This clause is useful if you are using the sequence numbers as timestamps. Guaranteeing order is usually not important for sequences used to generate primary keys.

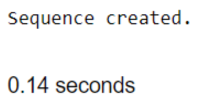
NOORDER

Specify NOORDER if you do not want to guarantee sequence numbers are generated in order of request. This is the default.

EXAMPLE:



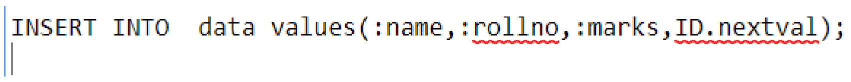
```
CREATE SEQUENCE ID
START WITH     1
INCREMENT BY   1
MAXVALUE 5
CACHE 4
CYCLE
ORDER;
```



```
Sequence created.

0.14 seconds
```

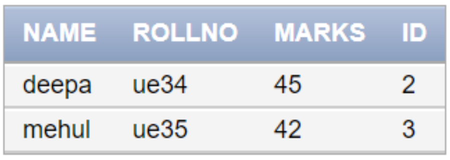
RESULT:

Now



```
INSERT INTO  data values(:name,:rollno,:marks,ID.nextval);
```

RESULT:



NAME	ROLLNO	MARKS	ID
deepa	ue34	45	2
mehul	ue35	42	3

Since next is used incremented ID used

<u>The DROP SEQUENCE Command</u>

```
DROP SEQUENCE sequence_name;
```

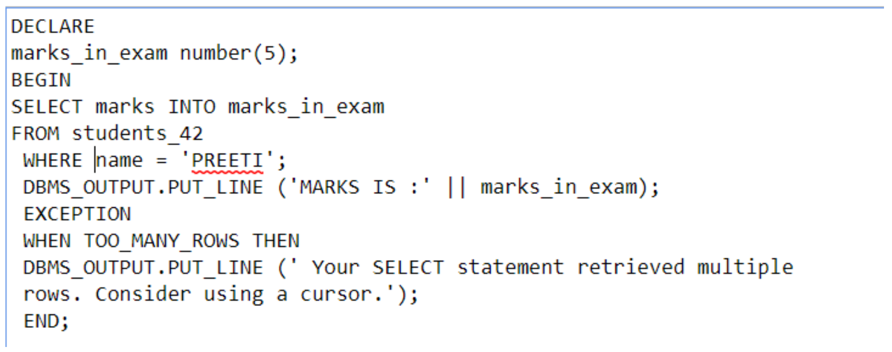
Introduction to PL/SQL

PL/SQL has procedural capabilities and enables condition checking , looping and branching . It also allows executing more than one statement at a time.
it has facility for programmed handling of errors.

A PL/SQL block is as follow:

```
[DECLARE]
   Declaration statements;
BEGIN
   Execution statements;
  [EXCEPTION]
      Exception handling statements;
END;
```

EXAMPLE:



```
DECLARE
marks_in_exam number(5);
BEGIN
SELECT marks INTO marks_in_exam
FROM students_42
 WHERE name = 'PREETI';
 DBMS_OUTPUT.PUT_LINE ('MARKS IS :' || marks_in_exam);
 EXCEPTION
 WHEN TOO_MANY_ROWS THEN
 DBMS_OUTPUT.PUT_LINE (' Your SELECT statement retrieved multiple
 rows. Consider using a cursor.');
 END;
```

RESULT:

```
 Your SELECT statement retrieved multiple
 rows. Consider using a cursor.

Statement processed.
```

Control Structures

The flow of control of statements can be classified into the following categories:

1. Conditional control

2. Iterative Control

3. Sequential control

Conditional Control: IF and CASE Statements

IF-THEN Statement

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

IF-THEN-ELSE Statement

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

IF-THEN-ELSIF Statement

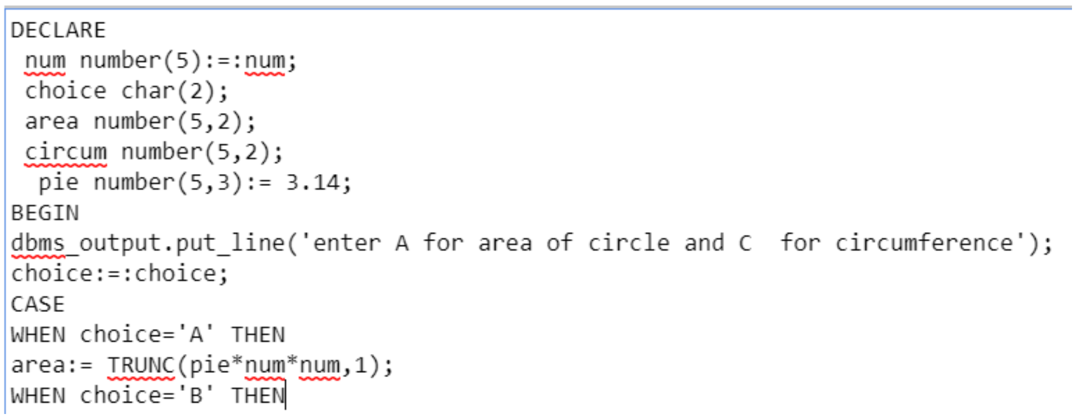
```
IF condition1 THEN
  sequence_of_statements1
ELSIF condition2 THEN
  sequence_of_statements2
ELSE
  sequence_of_statements3
END IF;
```

CASE Statement

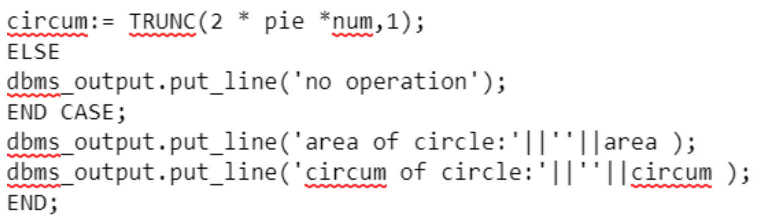
```
CASE selector
  WHEN expression1 THEN sequence_of_statements1;
  WHEN expression2 THEN sequence_of_statements2;
  ...
  WHEN expressionN THEN sequence_of_statementsN;
 [ELSE sequence_of_statementsN+1;]
END CASE ;
```

PROGRAMS

1. To find area and circumference of a circle when radius is input



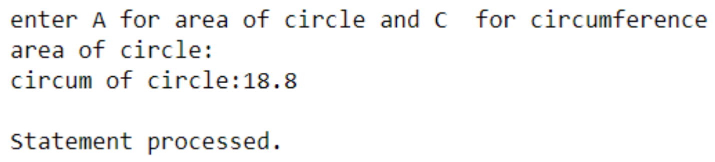
```
DECLARE
 num number(5):=:num;
 choice char(2);
 area number(5,2);
 circum number(5,2);
  pie number(5,3):= 3.14;
BEGIN
dbms_output.put_line('enter A for area of circle and C  for circumference');
choice:=:choice;
CASE
WHEN choice='A' THEN
area:= TRUNC(pie*num*num,1);
WHEN choice='B' THEN
```



```
circum:= TRUNC(2 * pie *num,1);
ELSE
dbms_output.put_line('no operation');
END CASE;
dbms_output.put_line('area of circle:'||''||area );
dbms_output.put_line('circum of circle:'||''||circum );
END;
```

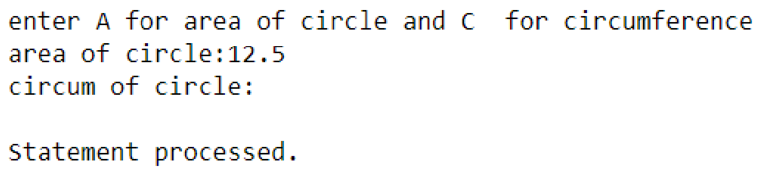
OUTPUT:

num =3 , choice= B



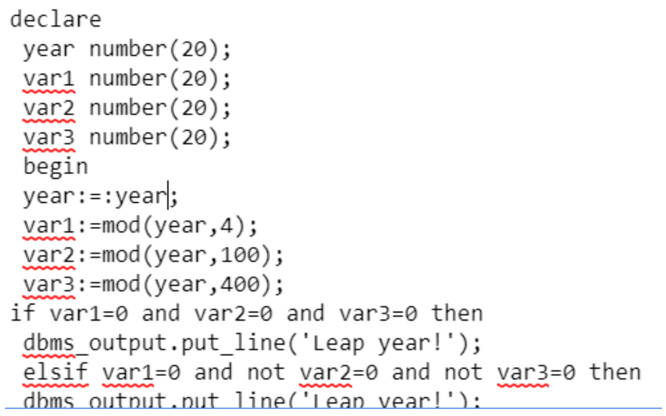
```
enter A for area of circle and C  for circumference
area of circle:
circum of circle:18.8

Statement processed.
```

num =2, choice= A

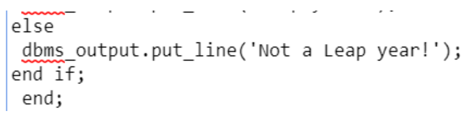


```
enter A for area of circle and C  for circumference
area of circle:12.5
circum of circle:

Statement processed.
```

2. Check whether leap year or not



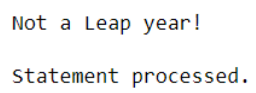
```
declare
 year number(20);
 var1 number(20);
 var2 number(20);
 var3 number(20);
 begin
 year:=:year;
 var1:=mod(year,4);
 var2:=mod(year,100);
 var3:=mod(year,400);
if var1=0 and var2=0 and var3=0 then
 dbms_output.put_line('Leap year!');
 elsif var1=0 and not var2=0 and not var3=0 then
 dbms_output.put_line('Leap year!');
```



```
else
 dbms_output.put_line('Not a Leap year!');
end if;
 end;
```

Input :1998

OUTPUT:



```
Not a Leap year!

Statement processed.
```

Iterative Control: LOOP and EXIT Statements

LOOP

```
LOOP
   sequence_of_statements
END LOOP;
```

EXIT

The EXIT statement forces a loop to complete unconditionally.

EXIT-WHEN

The EXIT-WHEN statement lets a loop complete conditionally.

WHILE-LOOP

```
WHILE condition LOOP
   sequence_of_statements
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 counter IN [REVERSE] lower_bound..higher_bound LOOP
   sequence_of_statements
```

```
END LOOP;
```

OBSERVATION: The variable in for loop need not be incremented

3. Check whether prime number or not

```
declare
num number(5);
temp number(5);
var number(5);
flag number(5):=0;
begin
num:=:num;
temp:=num/2;
for cntr in 2..temp
loop
var:=mod(num,cntr);
if var=0 then
flag:=1;

end if;
end loop;
if flag=1 then
dbms_output.put_line('Not a Prime Number');
else
dbms_output.put_line('Prime Number');
end if;
end;
```

Input:3

OUTPUT:

```
Prime Number

Statement processed.
```

Sequential control

GOTO statement changes the flow of control within a PL/SQL block

Cursor

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

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 attributes such as **%FOUND, %ISOPEN, %NOTFOUND**, and **%ROWCOUNT.** The following table provides the description of the most used attributes −

S.No	Attribute & Description
1	**%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.
2	**%NOTFOUND** The logical opposite of %FOUND. It returns TRUE if an INSERT, UPDATE, or DELETE statement affected no rows,

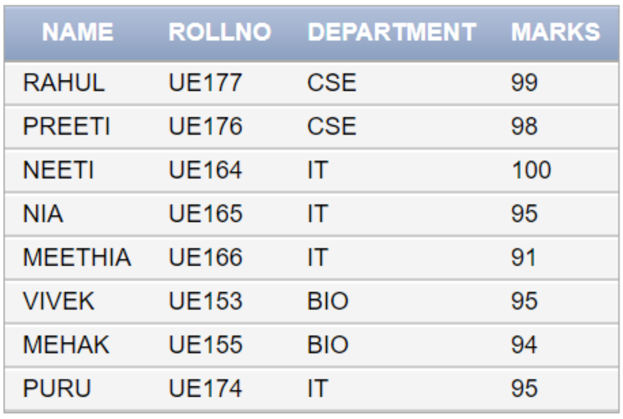
	or a SELECT INTO statement returned no rows. Otherwise, it returns FALSE.
3	**%ISOPEN** Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement.
4	**%ROWCOUNT** Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement.

Any SQL cursor attribute will be accessed as **sql%attribute_name**

PROGRAM

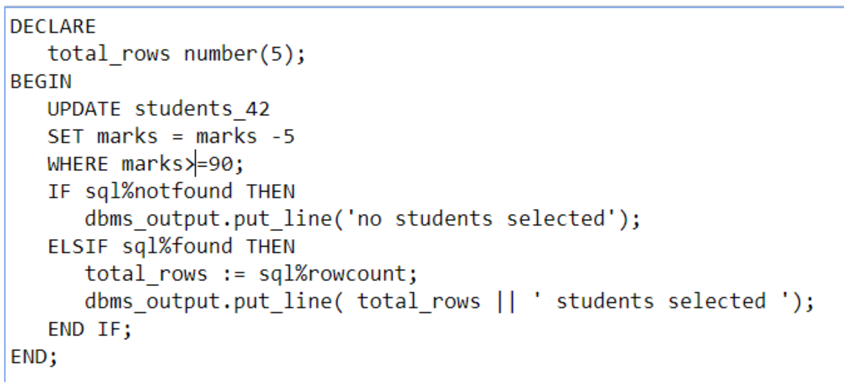
1. Counting number of records using implicit cursor

In table



NAME	ROLLNO	DEPARTMENT	MARKS
RAHUL	UE177	CSE	99
PREETI	UE176	CSE	98
NEETI	UE164	IT	100
NIA	UE165	IT	95
MEETHIA	UE166	IT	91
VIVEK	UE153	BIO	95
MEHAK	UE155	BIO	94
PURU	UE174	IT	95

EXAMPLE:



```
DECLARE
   total_rows number(5);
BEGIN
   UPDATE students_42
   SET marks = marks -5
   WHERE marks>=90;
   IF sql%notfound THEN
      dbms_output.put_line('no students selected');
   ELSIF sql%found THEN
      total_rows := sql%rowcount;
      dbms_output.put_line( total_rows || ' students selected ');
   END IF;
END;
```

RESULT:

```
8 students selected

1 row(s) updated.
```

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.

SYNTAX:

```
CURSOR cursor_name IS select_statement;
```

Working with an explicit cursor includes the following steps −

- Declaring the cursor for initializing the memory
- Opening the cursor for allocating the memory
- Fetching the cursor for retrieving the data
- Closing the cursor to release the 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 the memory for the cursor and makes it ready for fetching the rows returned by the SQL statement into it. For example, we will open the 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 the above-opened cursor as follows −

```
CLOSE c_customers;
```

PROGRAM

2. Displaying records using explicit cursor.

EXAMPLE:

```
CURSOR c1 IS
SELECT name,rollno,department
FROM students_42
WHERE marks>=90;
rec c1%rowtype;
BEGIN
OPEN c1;
LOOP
FETCH c1 into rec;
EXIT WHEN c1%notfound;
dbms_output.put_line(rec.name||'  '||rec.rollno||'  '|| rec.department);

END LOOP;
CLOSE c1;
END;
```

RESULT:

```
RAHUL  UE177  CSE
PREETI  UE176  CSE
NEETI  UE164  IT
NIA  UE165  IT
MEETHIA  UE166  IT
VIVEK  UE153  BIO
MEHAK  UE155  BIO
PURU  UE174  IT

Statement processed.
```

NOTE: rec c1%rowtype is used to make the variable rec compatible with c1 which has multiple columns instead of 1.So rec of rowtype c1 is being used .

No need to explicitly increment.

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

SYNTAX:

```
CREATE [OR REPLACE ] TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF }
{INSERT [OR] | UPDATE [OR] | DELETE}  [OF col_name]
ON table_name
[REFERENCING {OLD AS old, NEW AS new}]
[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 will be executed.

  NOTE: 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 will be updated.

- **[ON table_name]** –

  This specifies the name of the table associated with the trigger.

- **[REFERENCING {OLD AS old, NEW AS new}] –**

  This allows you to refer new and old values for various DML statements, such as INSERT, UPDATE, and DELETE.

- **[FOR EACH ROW] –**

  This specifies a **row-level trigger**, i.e., the trigger will 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.

  NOTE: This clause is valid only for row-level triggers.

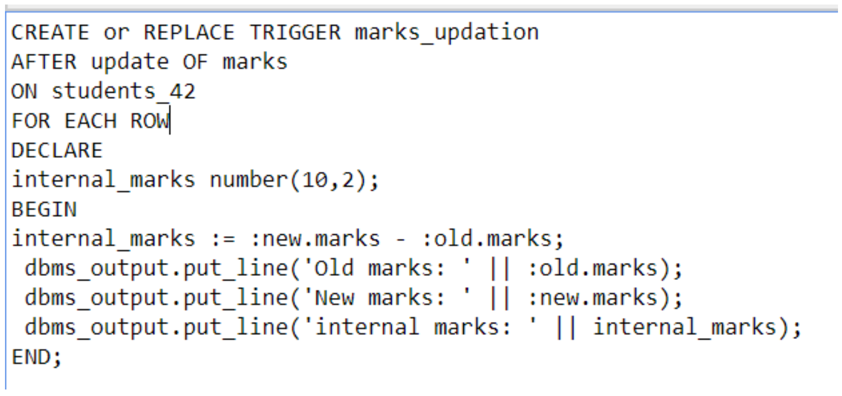
Types of PL/SQL Triggers

There are two types of triggers based on the which level it is triggered.
1) Row level trigger - An event is triggered for each row upated, inserted or deleted.
2) Statement level trigger - An event is triggered for each sql statement executed.

NOTE: A trigger can also be dropped using command

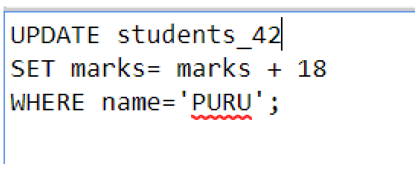
DROP TRIGGER trigger_name;

EXAMPLE OF ROW LEVEL TRIGGER:



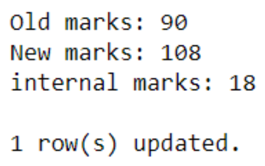
```
CREATE or REPLACE TRIGGER marks_updation
AFTER update OF marks
ON students_42
FOR EACH ROW
DECLARE
internal_marks number(10,2);
BEGIN
internal_marks := :new.marks - :old.marks;
 dbms_output.put_line('Old marks: ' || :old.marks);
 dbms_output.put_line('New marks: ' || :new.marks);
 dbms_output.put_line('internal marks: ' || internal_marks);
END;
```

on giving this command



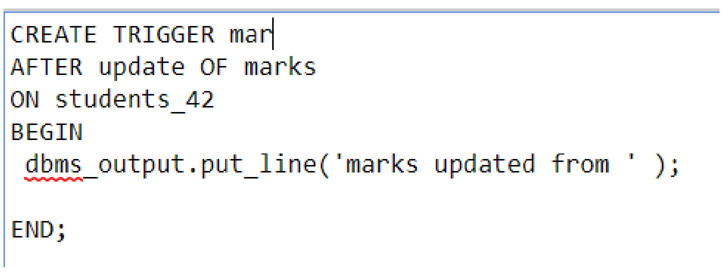
```
UPDATE students_42
SET marks= marks + 18
WHERE name='PURU';
```

RESULT:



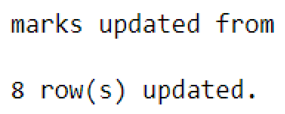
```
Old marks: 90
New marks: 108
internal marks: 18

1 row(s) updated.
```

EXAMPLE OF TABLE LEVEL TRIGGER:



```
CREATE TRIGGER mar
AFTER update OF marks
ON students_42
BEGIN
 dbms_output.put_line('marks updated from ' );

END;
```

RESULT:



```
marks updated from

8 row(s) updated.
```

NOTE: A table level trigger is a trigger that doesn't fire for each row to be changed. Accordingly, it lacks the for each row. Consequently, both, the :new and :old are not permitted in the trigger's PL/SQL block, otherwise, an **ORA-04082: NEW or OLD references not allowed in table level triggers** is thrown.

Subprograms in PL/SQL

Terminologies in PL/SQL Subprograms

Parameter:

The parameter is variable or placeholder of any valid PL/SQL datatype through which the PL/SQL subprogram exchange the values with the main code.

Types of parameter:

IN Parameter:

- This parameter is used for giving input to the subprograms.

OUT Parameter:

- This parameter is used for getting output from the subprograms.

IN OUT Parameter:

- This parameter is used for both giving input and for getting output from the subprograms.

Return

RETURN is the keyword that actually instructs the compiler to switch the control from the subprogram to the calling statement.

Procedures and functions are made of :

- **A declarative part**

Contains declarations of cursors , constants, variables, exceptions and sub programs. these objects are local to procedure or function. The object becomes invalid on exit.

- **An executable part**

It assigns values to the variables ,control execution and manipulate data

- **An optional error handling part**

Contains code that deals with exceptions that may be raised during the execution of the executable part.

Procedures

SYNTAX:

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

```
< exception in block>
END procedure_name;
```

Where,

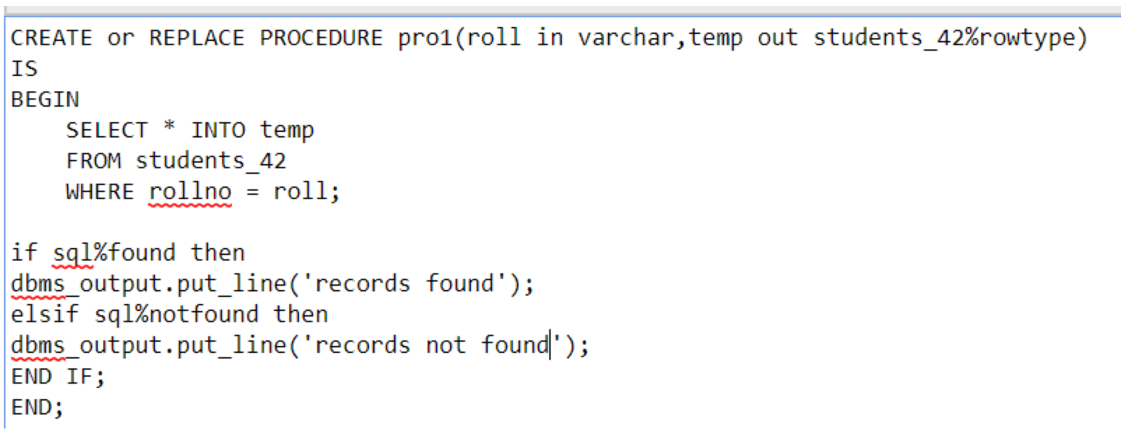
- ***procedure-name*** specifies the name of the procedure.
- [OR REPLACE] option allows the modification of an existing procedure.
- The optional parameter list contains name, mode and types of the parameters.

  IN represents the value that will be passed from outside and

  OUT represents the parameter that will be used to return a value outside of the procedure.
- *procedure-body* contains the executable part.

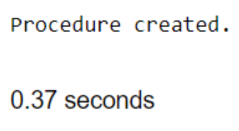
The AS keyword is used instead of the IS keyword for creating a standalone procedure

EXAMPLE:



```
CREATE or REPLACE PROCEDURE pro1(roll in varchar,temp out students_42%rowtype)
IS
BEGIN
    SELECT * INTO temp
    FROM students_42
    WHERE rollno = roll;

if sql%found then
dbms_output.put_line('records found');
elsif sql%notfound then
dbms_output.put_line('records not found');
END IF;
END;
```

RESULT:



```
Procedure created.

0.37 seconds
```

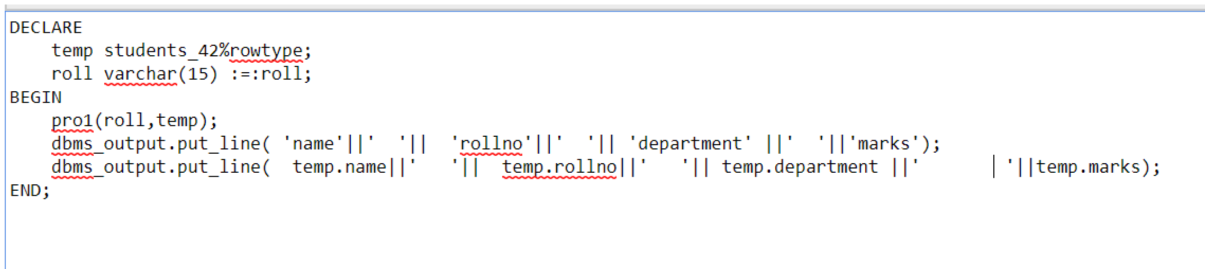
NOTE: %rowtype makes the datatype as that of the table

Call a procedure

In order to invoke a procedure that is stored in a DATABASE. Here is the syntax:

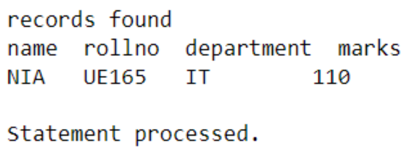
```
procedure_name([parameter[,...]])
```

EXAMPLE:



```
DECLARE
    temp students_42%rowtype;
    roll varchar(15) :=:roll;
BEGIN
    pro1(roll,temp);
    dbms_output.put_line( 'name'||'  '||  'rollno'||'  '|| 'department' ||'  '||'marks');
    dbms_output.put_line(  temp.name||'   '||  temp.rollno||'   '|| temp.department ||'      | '||temp.marks);
END;
```

RESULT:



```
records found
name  rollno  department  marks
NIA   UE165   IT          110

Statement processed.
```

OBSERVATION:

elsif part of procedure didn't work.

<u>Drop a procedure</u>

```
DROP PROCEDURE proc_name
```

Functions

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

SYNTAX:

```
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 the modification of an existing function.
- The optional parameter list contains name, mode and types of the parameters. IN represents the value that will be passed from outside and OUT represents the parameter that will be used to return a value outside of the procedure.
- The function must contain a **return** statement.
- The *RETURN* clause specifies the data type you are going to return from the function.

NOTE: don't mention size of type with parameter name.

Calling a Function

While creating a function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task. When a program calls a function, the program control is transferred to the called function.

Let entries in table be

NAME	ROLLNO	MARKS	ID
deepa	ue34	45	2
mehul	ue35	42	3

EXAMPLE:

```
CREATE FUNCTION totalmarks
RETURN number IS
   total number(10);
BEGIN
   SELECT SUM(marks) into total
   FROM data;

   RETURN total;
END;
```

CALLING A FUNCTION:

```
DECLARE
   c number(10);
BEGIN
   c := totalmarks();
   dbms_output.put_line('Sum of marks: ' || c);
END;
```

RESULT:

```
Sum of marks: 87

Statement processed.
```

PROGRAMS

1. Fibonnaci series

Main program

```
DECLARE
num number(2);
c number(2);
i number(1);
BEGIN
c:=:no_of_elements;
for i IN 0..(c-1)
LOOP
num:=fibo(i);
dbms_output.put_line(num);
END LOOP;
END;
```

Result

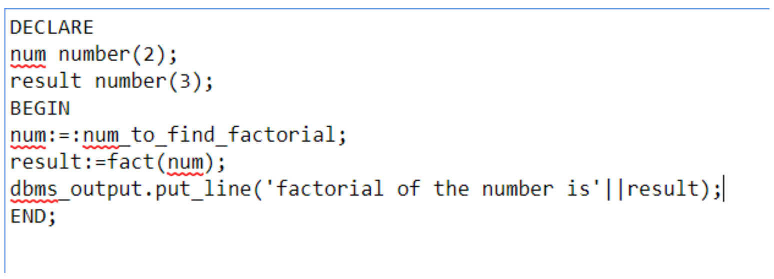
```
0
1
1
2
3

Statement processed.
```

Function

```
create function fibo(n IN number)
return number
AS
v number(2);
BEGIN
if n=0 THEN
v:=0;
return v;
elsif n=1 THEN
v:=1;
return v;
else
v:= fibo(n-1)+fibo(n-2);
return v;
end if;

END fibo;
```

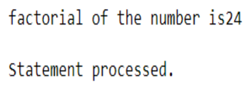
2. Factorial of a number

Main program



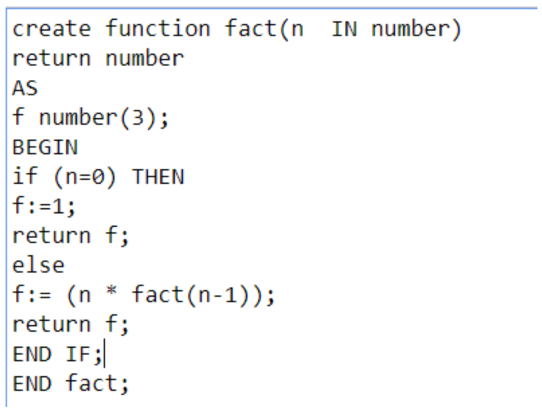
```
DECLARE
num number(2);
result number(3);
BEGIN
num:=:num_to_find_factorial;
result:=fact(num);
dbms_output.put_line('factorial of the number is'||result);
END;
```

Result



```
factorial of the number is24

Statement processed.
```

Function



```
create function fact(n  IN number)
return number
AS
f number(3);
BEGIN
if (n=0) THEN
f:=1;
return f;
else
f:= (n * fact(n-1));
return f;
END IF;
END fact;
```

Packages

A package is a group of functions and procedures.

A package will have two mandatory parts –

- **<u>Package specification</u>**

The package specification contains public declarations. The declared items are accessible from anywhere in the package and to any other subprograms in the same schema.

- **<u>Package body or definition</u>**

The package body contains the implementation of every cursor and subprogram declared in the package spec.

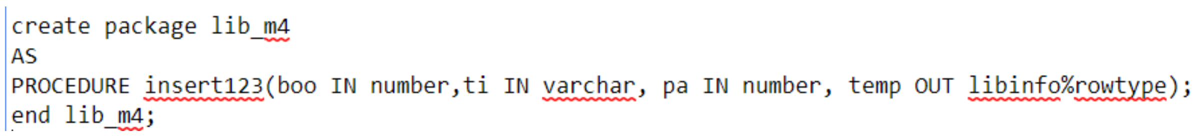
PL/SQL package body contains all the code that implements stored functions, procedures, and cursors listed in the package specificatio

SYNTAX OF CREATING PACKAGE :

```
CREATE [OR REPLACE] PACKAGE package_name
```

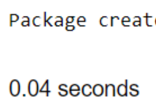
```
[ AUTHID { CURRENT_USER | DEFINER } ]
{ IS | AS }

   [definitions of public TYPES
   ,declarations of public variables, types, and objects
   ,declarations of exceptions
   ,pragmas
   ,declarations of cursors, procedures, and functions
   ,headers of procedures and functions]
END [package_name];
```

EXAMPLE:



```
create package lib_m4
AS
PROCEDURE insert123(boo IN number,ti IN varchar, pa IN number, temp OUT libinfo%rowtype);
end lib_m4;
```

RESULT:



```
Package created.

0.04 seconds
```

SYNTAX OF CREATING PACKAGE BODY:

```
CREATE [OR REPLACE] PACKAGE BODY package_name
   { IS | AS }

   [definitions of private TYPEs
   ,declarations of private variables, types, and objects
   ,full definitions of cursors
   ,full definitions of procedures and functions]
[BEGIN
   sequence_of_statements

[EXCEPTION
   exception_handlers ] ]

END [package_name];
```

EXAMPLE:

```
create package body lib_m4
AS
procedure insert123(boo IN number,ti IN varchar, pa IN number,temp OUT libinfo%rowtype)
AS
id number(2);
name varchar(30);
page number(30);
BEGIN
id:=boo;
name:=ti ;
page:=pa;
insert into libinfo values(id,name,page);
select *  into temp from libinfo
where bookid=id;
END insert123;
end lib_m4;
```

RESULT:

```
Package Body created.
```

0.14 seconds

Using the Package Elements

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

```
package_name.element_name;
```

```
declare
boo number(2):=:i;
ti varchar(30):=:t;
pa number(30):=:p;
temp libinfo%rowtype;
begin
lib_m4.insert123(boo,ti,pa,temp);
dbms_output.put_line('inserted'||'  '|| temp.title ||'  '||temp.pages||'  '||temp.bookid);
end;
```

RESULT:

```
inserted  hiya  241  11

Statement processed.
```

Miscellaneous

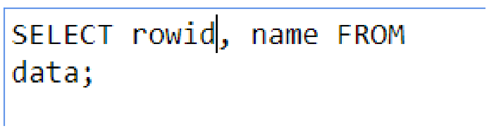
Pseudo columns –ROW ID and ROW NUM

Pseudo columns are actually associated with the table data but it has nothing to do with table data. ROWID & ROWNUM are pseudo columns which are not actual columns in the table but behave like actual columns.

<u>What is ROWID?</u>

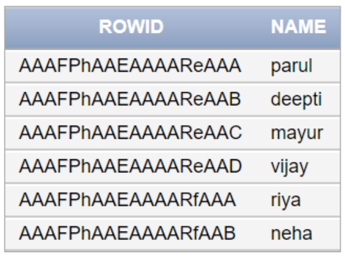
1. ROWID is nothing but the physical memory location on which that data/row is stored. ROWID basically returns address of row.
2. ROWID uniquely identifies row in database.
3. ROWID is 16 digit hexadecimal number whose data type is also ROWID Or UROWID

EXAMPLE:



```
SELECT rowid, name FROM
data;
```

RESULT:



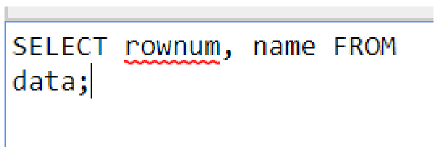
ROWID	NAME
AAAFPhAAEAAAAReAAA	parul
AAAFPhAAEAAAAReAAB	deepti
AAAFPhAAEAAAAReAAC	mayur
AAAFPhAAEAAAAReAAD	vijay
AAAFPhAAEAAAARfAAA	riya
AAAFPhAAEAAAARfAAB	neha

<u>What is ROWNUM?</u>

1. ROWNUM is magical column in Oracle which assigns the sequence number to the rows retrievals in the table.

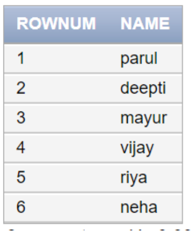
2. ROWNUM is logical number assigned temporarily to the physical location of the row.

EXAMPLE:



```
SELECT rownum, name FROM
data;
```

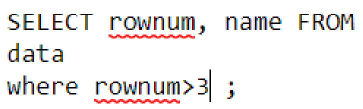
RESULT:



ROWNUM	NAME
1	parul
2	deepti
3	mayur
4	vijay
5	riya
6	neha

OBSERVATION:

rownum with greater than does not work but with less then works



```
SELECT rownum, name FROM
data
where rownum>3 ;
```

NOTE: rownum begins with 1 and not 0

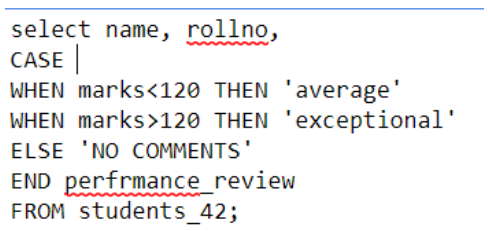
Searched Case

The SQL searched CASE expression gives you more flexibility when writing your comparison conditions.

SYNTAX:

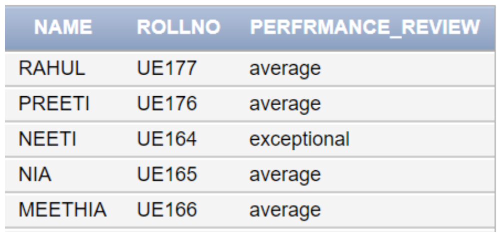
```
CASE
WHEN condition1 THEN result_expression1
[ WHEN condition2 THEN result_expression2 ]
[ ELSE result_expression ]
END label_name
```

EXAMPLE:



```
select name, rollno,
CASE
WHEN marks<120 THEN 'average'
WHEN marks>120 THEN 'exceptional'
ELSE 'NO COMMENTS'
END perfrmance_review
FROM students_42;
```

RESULT:



NAME	ROLLNO	PERFRMANCE_REVIEW
RAHUL	UE177	average
PREETI	UE176	average
NEETI	UE164	exceptional
NIA	UE165	average
MEETHIA	UE166	average

Introduction to dual tables

The DUAL table is a special one-row, one-column table present by default in Oracle and other database installations. In Oracle, the table has a single VARCHAR2(1) column called DUMMY that has a value of 'X'. It is suitable for use in selecting a pseudo column such as SYSDATE or USER.

ARITHMETIC OPERATIONS

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.

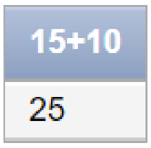
SYNATX:

```
SELECT <Expression>[arithmetic operator]<expression>...

 FROM [table_name]

 WHERE [expression];
```

EXAMPLE:

PLUS OPERATOR

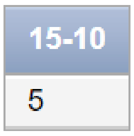
SELECT 15+10 FROM Dual;



15+10
25

MINUS OPERATOR

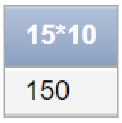
SELECT 15-10 FROM Dual;



15-10
5

MULTIPLICATION OPERATOR

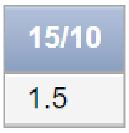
SELECT 15*10 FROM Dual;



15*10
150

DIVISION OPERATOR

SELECT 15/10 FROM Dual;



15/10
1.5

INSERTION FROM ANOTHER TABLE

The INSERT INTO SELECT statement copies data from one table and inserts it into another table.

INSERT INTO SELECT requires that data types in source and target tables match

The existing records in the target table are unaffected.

SYNTAX:

```
INSERT INTO table2 (column1, column2, column3, ...)

SELECT column1, column2, column3, ...

FROM table1

WHERE condition;
```

EXAMPLE:

INSERT INTO Empcopy

SELECT * FROM employee1

WHERE empno in ('E1','E8');

```
1 row(s) inserted.
```

ALIASING OF COLUMN OR TABLE

SQL aliases are used to give a table, or a column in a table, a temporary name.Aliases are often used to make column names more readable.An alias only exists for the duration of the query.

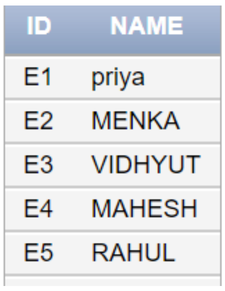
<u>COLUMN ALIAS</u>

SYNTAX:

```
SELECT column_name AS alias_name

FROM table_name;
```

EXAMPLE:

SELECT empno as ID, ename as NAME FROM employee1;



ID	NAME
E1	priya
E2	MENKA
E3	VIDHYUT
E4	MAHESH
E5	RAHUL

<u>TABLE ALIAS</u>

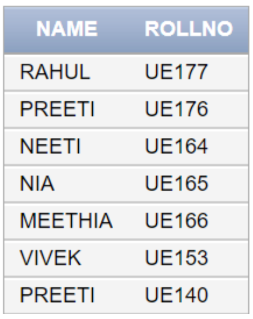
SYNTAX:

```
SELECT column_name(s)

FROM table_name AS alias_name;
```

EXAMPLE:

SELECT name,rollno

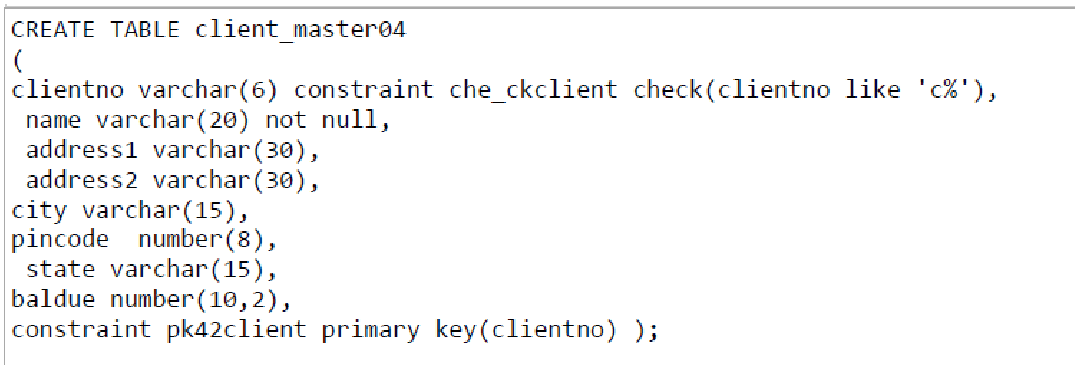
from students_42 T;



NAME	ROLLNO
RAHUL	UE177
PREETI	UE176
NEETI	UE164
NIA	UE165
MEETHIA	UE166
VIVEK	UE153
PREETI	UE140

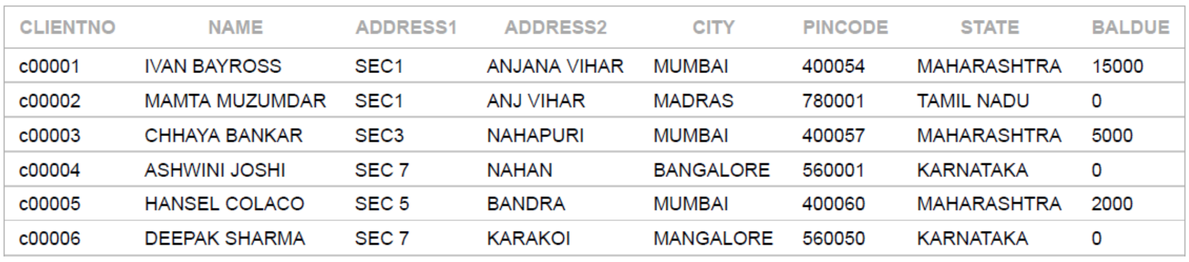
Hand’s on exercise

Table client_master04



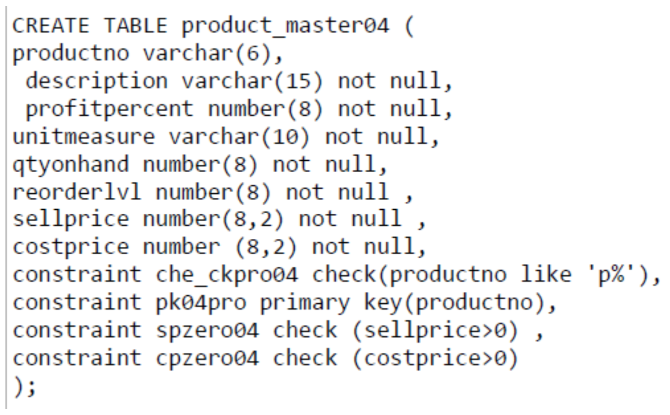
```
CREATE TABLE client_master04
(
clientno varchar(6) constraint che_ckclient check(clientno like 'c%'),
 name varchar(20) not null,
 address1 varchar(30),
 address2 varchar(30),
city varchar(15),
pincode  number(8),
 state varchar(15),
baldue number(10,2),
constraint pk42client primary key(clientno) );
```

Table contents



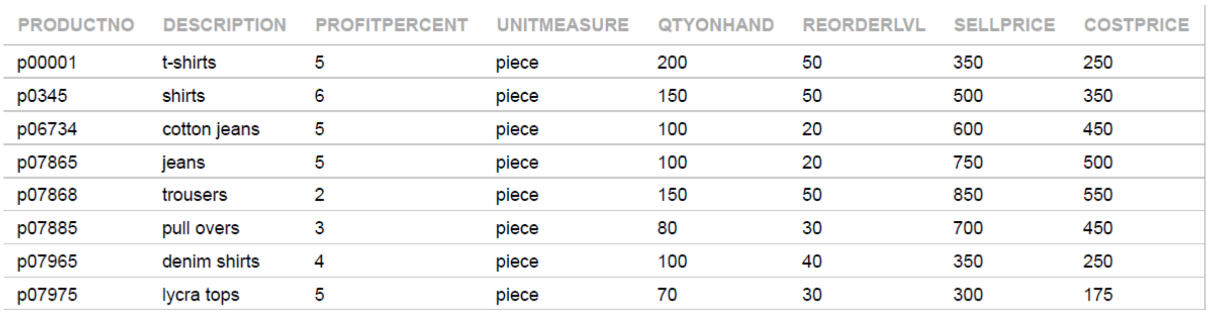
CLIENTNO	NAME	ADDRESS1	ADDRESS2	CITY	PINCODE	STATE	BALDUE
c00001	IVAN BAYROSS	SEC1	ANJANA VIHAR	MUMBAI	400054	MAHARASHTRA	15000
c00002	MAMTA MUZUMDAR	SEC1	ANJ VIHAR	MADRAS	780001	TAMIL NADU	0
c00003	CHHAYA BANKAR	SEC3	NAHAPURI	MUMBAI	400057	MAHARASHTRA	5000
c00004	ASHWINI JOSHI	SEC 7	NAHAN	BANGALORE	560001	KARNATAKA	0
c00005	HANSEL COLACO	SEC 5	BANDRA	MUMBAI	400060	MAHARASHTRA	2000
c00006	DEEPAK SHARMA	SEC 7	KARAKOI	MANGALORE	560050	KARNATAKA	0

Table products_master04



```
CREATE TABLE product_master04 (
productno varchar(6),
 description varchar(15) not null,
 profitpercent number(8) not null,
unitmeasure varchar(10) not null,
qtyonhand number(8) not null,
reorderlvl number(8) not null ,
sellprice number(8,2) not null ,
costprice number (8,2) not null,
constraint che_ckpro04 check(productno like 'p%'),
constraint pk04pro primary key(productno),
constraint spzero04 check (sellprice>0) ,
constraint cpzero04 check (costprice>0)
);
```

Table contents

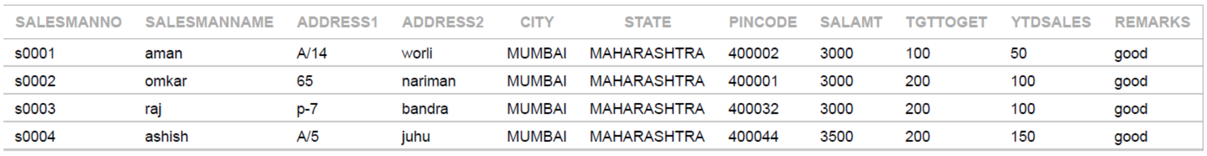


PRODUCTNO	DESCRIPTION	PROFITPERCENT	UNITMEASURE	QTYONHAND	REORDERLVL	SELLPRICE	COSTPRICE
p00001	t-shirts	5	piece	200	50	350	250
p0345	shirts	6	piece	150	50	500	350
p06734	cotton jeans	5	piece	100	20	600	450
p07865	jeans	5	piece	100	20	750	500
p07868	trousers	2	piece	150	50	850	550
p07885	pull overs	3	piece	80	30	700	450
p07965	denim shirts	4	piece	100	40	350	250
p07975	lycra tops	5	piece	70	30	300	175

Table sales_master04

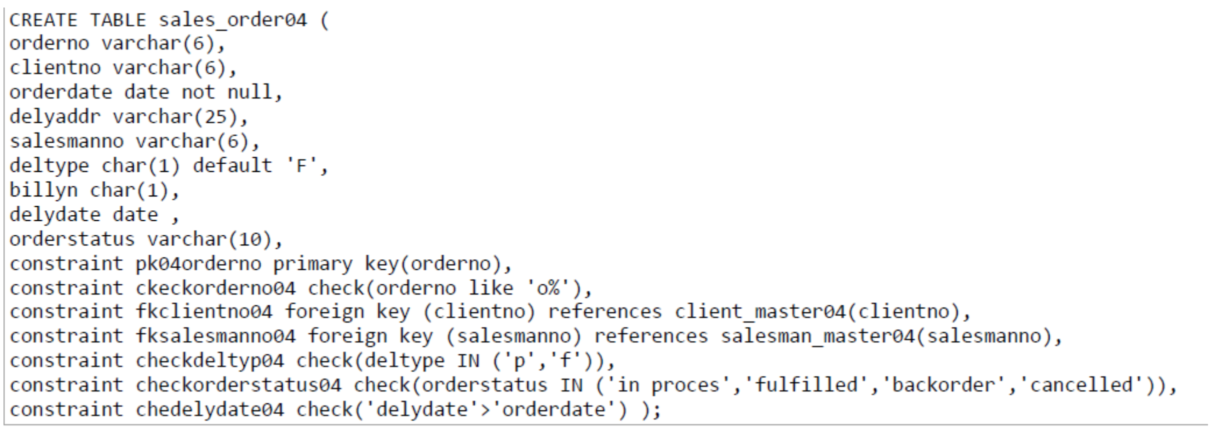
```
CREATE TABLE salesman_master04 (
salesmanno varchar(6),
salesmanname varchar(20) not null,
address1 varchar(30) not null,
address2 varchar(30),
city varchar(20),
state varchar(20),
pincode number(8) ,
salamt number(8,2) not null ,
tgttoget number(6,2) not null,
ytdsales number(6,2) not null ,
remarks varchar(60),
constraint checksm04 check(salesmanno like 's%'),
constraint pk04sm primary key(salesmanno),
constraint sal04zero check (salamt>0) ,
constraint tgt04zero check (tgttoget>0)
);
```

Table contents



SALESMANNO	SALESMANNAME	ADDRESS1	ADDRESS2	CITY	STATE	PINCODE	SALAMT	TGTTOGET	YTDSALES	REMARKS
s0001	aman	A/14	worli	MUMBAI	MAHARASHTRA	400002	3000	100	50	good
s0002	omkar	65	nariman	MUMBAI	MAHARASHTRA	400001	3000	200	100	good
s0003	raj	p-7	bandra	MUMBAI	MAHARASHTRA	400032	3000	200	100	good
s0004	ashish	A/5	juhu	MUMBAI	MAHARASHTRA	400044	3500	200	150	good

Table sales_order04

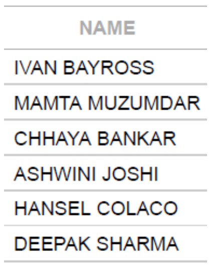


```
CREATE TABLE sales_order04 (
orderno varchar(6),
clientno varchar(6),
orderdate date not null,
delyaddr varchar(25),
salesmanno varchar(6),
deltype char(1) default 'F',
billyn char(1),
delydate date ,
orderstatus varchar(10),
constraint pk04orderno primary key(orderno),
constraint ckeckorderno04 check(orderno like 'o%'),
constraint fkclientno04 foreign key (clientno) references client_master04(clientno),
constraint fksalesmanno04 foreign key (salesmanno) references salesman_master04(salesmanno),
constraint checkdeltyp04 check(deltype IN ('p','f')),
constraint checkorderstatus04 check(orderstatus IN ('in proces','fulfilled','backorder','cancelled')),
constraint chedelydate04 check('delydate'>'orderdate') );
```

QUERIES

1. Find out names of all the clients.

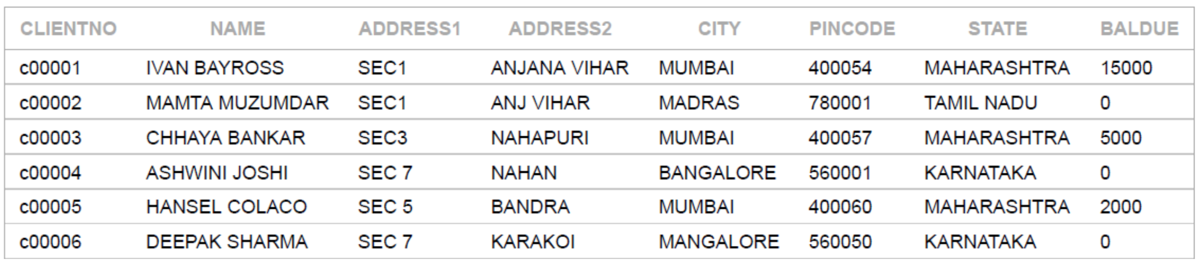
```
Select  name from client_master04;
```



NAME
IVAN BAYROSS
MAMTA MUZUMDAR
CHHAYA BANKAR
ASHWINI JOSHI
HANSEL COLACO
DEEPAK SHARMA

2. Retrieve the entire contents of the Client_Master table.

```
Select * from client_master04;
```



CLIENTNO	NAME	ADDRESS1	ADDRESS2	CITY	PINCODE	STATE	BALDUE
c00001	IVAN BAYROSS	SEC1	ANJANA VIHAR	MUMBAI	400054	MAHARASHTRA	15000
c00002	MAMTA MUZUMDAR	SEC1	ANJ VIHAR	MADRAS	780001	TAMIL NADU	0
c00003	CHHAYA BANKAR	SEC3	NAHAPURI	MUMBAI	400057	MAHARASHTRA	5000
c00004	ASHWINI JOSHI	SEC 7	NAHAN	BANGALORE	560001	KARNATAKA	0
c00005	HANSEL COLACO	SEC 5	BANDRA	MUMBAI	400060	MAHARASHTRA	2000
c00006	DEEPAK SHARMA	SEC 7	KARAKOI	MANGALORE	560050	KARNATAKA	0

3. Find the names of all the salesmen who have salary equal to Rs.3000

```
Select  salesmanname from sales_master04;
```

SALESMANNAME
aman
omkar
raj
ashish

4. Delete all salesmen from the Sales_Master04 whose salaries are equal to Rs.3000

```
delete from sales_master04 where salamt=3000;
```

4 row deleted

5. Change the city of client no ‘C0005’ to ‘Bangalore’.

```
 update client_master04
 set city='Bangalore'
where clientno='C0005';
```

1 row updated.

6. Add a column called ‘Telephone’ of ‘number’ type and size ‘10’ to the Client_Master04 table.

```
 alter table client_master04 add telephone number(10);
```

```
Table altered.
```