DBMS LAB RECORD

WEEK - 1

Aim:

File system Vs. DBMS [Reading student / employee details from a file with comma separated fields in each record]

Description:

File system:
A file system stores and organises data and can be thought of as a type of index for all the data contained in a storage device. These devices can include hard drives, optical drives and flash drives.

Syntax:

fopen:
FILE *fopen(const char *file_name, const char *mode_of_operation);

fclose:

int fclose(FILE *stream)

Program:

```
#include<stdio.h>
#include<stdlib.h>
typedef struct employee
{
  int emp_id;
  char emp_name[30];
  float emp_salary;
}emp;
emp e;
FILE *fp;
void Insert()
{
  fp = fopen("emp.txt","a");
  printf("Enter employee id,salary and name: ");
  scanf("%d %f %s",&e.emp_id,&e.emp_salary,e.emp_name);
  fprintf(fp,"%d %f %s\n",e.emp_id,e.emp_salary,e.emp_name);
  fclose(fp);
}
void Display()
{
 fp=fopen("emp.txt","r");
printf("\n========================================================\n");
 printf("   EMPLOYEE DETAILS");
 printf("\n========================================================");
 printf("\nEMPLOYEE ID\t\tSALARY\t\t\tNAME");
 printf("\n--------------------------------------------------------------\n");
 while(fscanf(fp, "%d %f %s ",&e.emp_id,&e.emp_salary,e.emp_name) != EOF)
 {
    printf("%d\t\t\t%f\t\t%s\n",e.emp_id,e.emp_salary,e.emp_name);
 }
 printf("=========================================================\n");
 fclose(fp);
}
void show_spec_fields()
{
  int num;
  while(1)
  {
      printf("Enter your choice:\n1)EMPLOYEE IDS\n2)EMPLOYEE
NAMES\n3)EMPLOYEE SALARY\n4)EXIT\n");
      scanf("%d",&num);
     fp=fopen("emp.txt","r");
     if(num==1)
     {
      printf("\n=========================");
      printf("\nEMPLOYEE ID");
      printf("\n--------------------------\n");
      while(fscanf(fp, "%d %f %s",&e.emp_id,&e.emp_salary,e.emp_name) != EOF)
       {
      printf("%d\n",e.emp_id);
       }
       printf("==========================\n");
       fclose(fp);
       }
     else if(num == 2)
     {
      fp=fopen("emp.txt","r");
      printf("\n=========================");
      printf("\nEMPLOYEE NAMES");
      printf("\n--------------------------\n");
      while(fscanf(fp, "%d %f %s ",&e.emp_id,&e.emp_salary,e.emp_name) != EOF)
       {
       printf("%s\n",e.emp_name);
       }
      printf("==========================\n");
      fclose(fp);
       }
      else if(num == 3)
     {
      fp=fopen("emp.txt","r");
      printf("\n========================");
      printf("\nEMPLOYEE SALARY");
      printf("\n-----------------------\n");
      while(fscanf(fp, "%d %f %s",&e.emp_id,&e.emp_salary,e.emp_name) != EOF)
       {
       printf("%f\n",e.emp_salary);
       }
      printf("==========================\n");
      fclose(fp);
      }
      else
       {
         exit(0);
       }
  }
}
void main()
{
  int n;
  while(1)
  {
  printf("Enter your choice: \n");
  printf("1)Insert\n2)Display\n3)Show specific fields\n4)Exit\n");
  scanf("%d",&n);
  if(n == 1)
  {
     Insert();
  }
  else if(n ==2)
  {
     Display();
  }
  else if(n==3)
  {
  show_spec_fields();
  }
  else
  {
     exit(0);
     fclose(fp);
  }
  }
}
```

Output:

```
Enter your choice:
1)Insert
```

```
2)Display
3)Show specific fields
4)Exit
1
Enter employee id,salary and name: 111 10000 HARSHA
Enter your choice:
1)Insert
2)Display
3)Show specific fields
4)Exit
1
Enter employee id,salary and name: 222 30000 ASHIK
Enter your choice:
1)Insert
2)Display
3)Show specific fields
4)Exit
2

=======================================================
  EMPLOYEE DETAILS
=======================================================
EMPLOYEE ID            SALARY                  NAME
-------------------------------------------------------
1                 10000.000000        HARSHA
2                 30000.000000        ASHIK
=======================================================
Enter your choice:
1)Insert
2)Display
3)Show specific fields
4)Exit
3
Enter your choice:
1)EMPLOYEE IDS
2)EMPLOYEE NAMES
3)EMPLOYEE SALARY
4)EXIT
1 2   2 3

=======================
EMPLOYEE NAMES
----------------------
HARSHA
ASHIK
=========================
Enter your choice:
1)EMPLOYEE IDS
2)EMPLOYEE NAMES
```

```
3)EMPLOYEE SALARY
4)EXIT

======================
EMPLOYEE SALARY
----------------------
10000.000000
30000.000000
========================
Enter your choice:
1)EMPLOYEE IDS
2)EMPLOYEE NAMES
3)EMPLOYEE SALARY
4)EXIT
4
```

Inference:

Data Sharing is too complex by using

Files Data is less secure when it is

stored in a file

Data can be easily manipulated using filesystem

For every search operation performed on the file system, a different application program has to be written.

WEEK - 2

Creating, Altering, Dropping tables with Constraints, Insert Table.

Experiment – 1:

Aim:

Create location, department, job_grade, and employee tables with the following columns.
Location: (location_id:number, city:string)
Department: (department_id:number, department_name:string, head:number, department_location:number)
Job_grade: (job_grade:string, lower_bound:number, upper_bound:number)
Employee: (employee_id:number, first_name:string, last_name:string, join_date:date, manager_id:number, salary:number)

Description:

Tables are used to store data in the database. Tables are uniquely named within a database and schema. Each table contains one or more columns. And each column has an associated data type that defines the kind of data it can store e.g., numbers, strings, or temporal data.

To create a new table, you use the CREATE TABLE statement as follows:

SYNTAX:

Create:

```
CREATE TABLE table_name (

        column1 datatype,
        column2 datatype,
        column3 datatype,
  ....
);
```

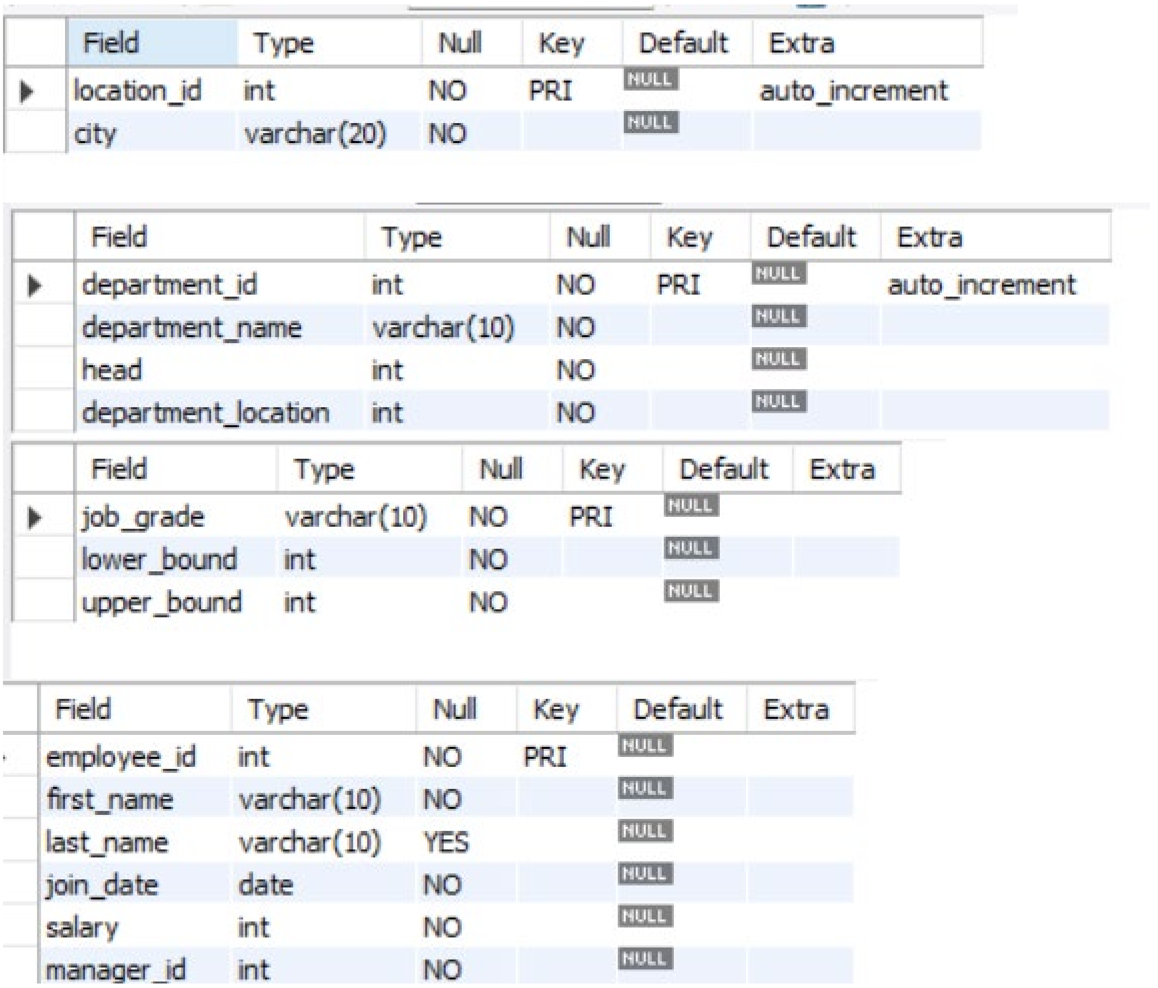
Query Command:

Program :-

```
CREATE DATABASE 20331A05F3;
USE 20331A05F3;

CREATE table location(location_id int primary key auto_increment, city varchar(20) not
null);
DESC location;
```

```
CREATE table department (department_id int primary key auto_increment,
department_name varchar(10) not null, head int not null, department_location int not null);
describe department;
CREATE table job_grade(job_grade varchar(10) primary key , lower_bound int not null ,
upper_bound int not null);
DESC job_grade;
CREATE table employee(employee_id int primary key, first_name varchar(10) not null,
last_name varchar(10), join_date date not null, salary int not null, manager_id int not null );
DESC employee;
```

Output:



Field	Type	Null	Key	Default	Extra
location_id	int	NO	PRI	NULL	auto_increment
city	varchar(20)	NO		NULL	

Field	Type	Null	Key	Default	Extra
department_id	int	NO	PRI	NULL	auto_increment
department_name	varchar(10)	NO		NULL	
head	int	NO		NULL	
department_location	int	NO		NULL	

Field	Type	Null	Key	Default	Extra
job_grade	varchar(10)	NO	PRI	NULL	
lower_bound	int	NO		NULL	
upper_bound	int	NO		NULL	

Field	Type	Null	Key	Default	Extra
employee_id	int	NO	PRI	NULL	
first_name	varchar(10)	NO		NULL	
last_name	varchar(10)	YES		NULL	
join_date	date	NO		NULL	
salary	int	NO		NULL	
manager_id	int	NO		NULL	

Inference:

We will learn how to create Tables .

Experiment 2:

Aim:

Alter employee table to add job_grade column which is of string data type.

Description:

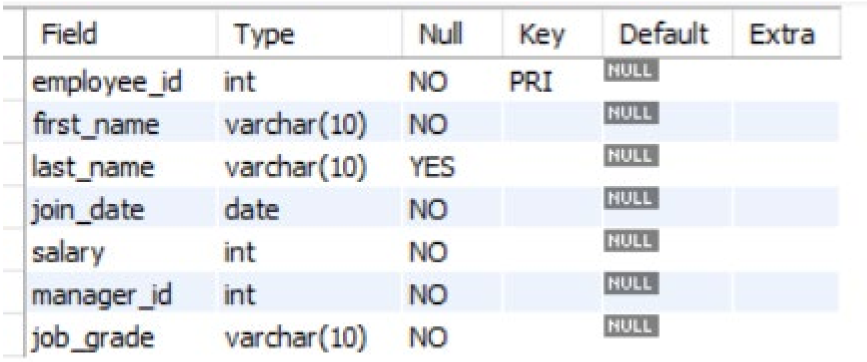
The SQL ALTER TABLE command is used to add, delete or modify columns in an existing table. You should also use the ALTER TABLE command to add and drop various constraints on an existing table.
The basic syntax of an ALTER TABLE command to add a New Column in an existing table is as follows.

Syntax:
Alter:
ALTER TABLE *table_name*
ADD *column_name datatype*;

Query Command:

ALTER table employee add column job_grade varchar(10) not null;
DESC employee;

Output:



Field	Type	Null	Key	Default	Extra
employee_id	int	NO	PRI	NULL	
first_name	varchar(10)	NO		NULL	
last_name	varchar(10)	YES		NULL	
join_date	date	NO		NULL	
salary	int	NO		NULL	
manager_id	int	NO		NULL	
job_grade	varchar(10)	NO		NULL	

Inference:

We will learn how to alter Tables .

Experiment 3:

Aim:

Alter employee table to make job_grade a foreign key to job_grade table, manager_id a foreign key to employee table, department_id a foreign key to department table.
Draw an ER diagram depicting the 4 tables from Experiment 1 with all the added constraints
from Experiment 2 to Experiment 4.

Description:

The ALTER TABLE command adds, deletes, or modifies columns in a table.
The ALTER TABLE command also adds and deletes various constraints in a table.
The basic syntax of an ALTER TABLE command to add a New Column in an existing table is as follows.

Syntax:

Alter:
ALTER TABLE *table_name*
ADD *column_name datatype*;

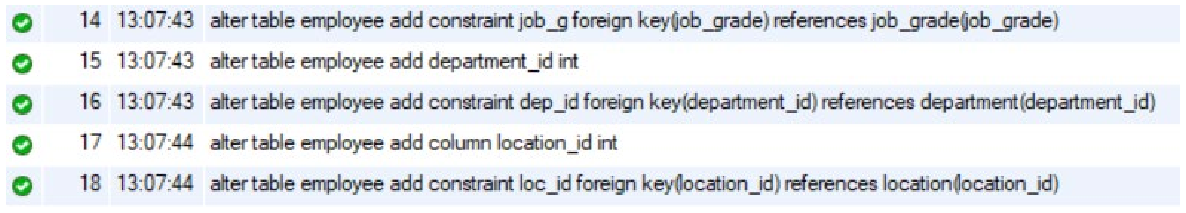
FORIGEN KEY:

```
CREATE TABLE Orders (
        OrderID int NOT NULL,
        OrderNumber int NOT NULL,
        PersonID int,
   PRIMARY KEY (OrderID),
   FOREIGN KEY (PersonID) REFERENCES Persons(PersonID)
);
```

Query Command:

```
ALTER table employee add constraint job_g foreign key(job_grade) references job_grade(job_grade);
ALTER table employee add department_id int ;
ALTER table employee add constraint dep_id foreign key(department_id) references department(department_id);
ALTER table employee add column location_id int;
ALTER table employee add constraint loc_id foreign key(location_id) references location(location_id);
```

Output:



```
✓  14  13:07:43  alter table employee add constraint job_g foreign key(job_grade) references job_grade(job_grade)
✓  15  13:07:43  alter table employee add department_id int
✓  16  13:07:43  alter table employee add constraint dep_id foreign key(department_id) references department(department_id)
✓  17  13:07:44  alter table employee add column location_id int
✓  18  13:07:44  alter table employee add constraint loc_id foreign key(location_id) references location(location_id)
```

Inference:

We will learn how to alter Tables .

Experiment 4:

Aim:

Create a dummy table called my_employee with the same definition as employee table and then drop the table.

Description:

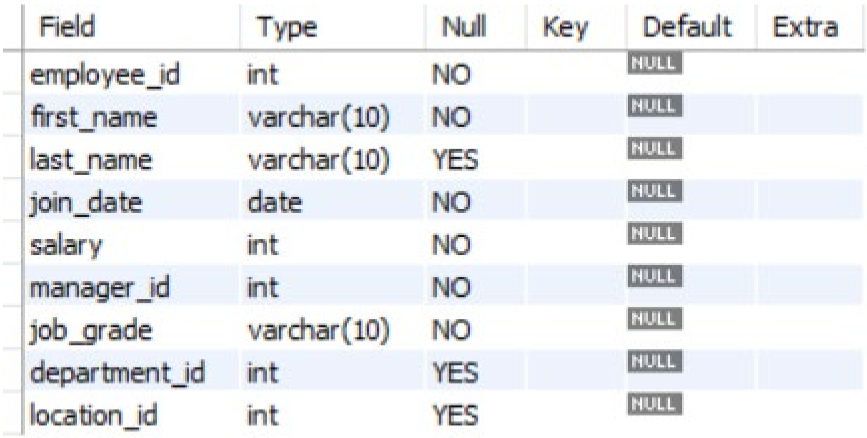
A DROP statement in SQL removes a component from a relational database management system (RDBMS).

Syntax:
Drop:
DROP TABLE *table_name*;

Query Command:

CREATE table my_employee as (select * from employee);
DESC my_employee;
DESC table my_employee;

Output:



Field	Type	Null	Key	Default	Extra
employee_id	int	NO		NULL	
first_name	varchar(10)	NO		NULL	
last_name	varchar(10)	YES		NULL	
join_date	date	NO		NULL	
salary	int	NO		NULL	
manager_id	int	NO		NULL	
job_grade	varchar(10)	NO		NULL	
department_id	int	YES		NULL	
location_id	int	YES		NULL	

Inference:

We will learn how to drop Tables .

Experiment 5:

Aim:

Insert data into location, department, job_grade & employee tables.

Description:

The INSERT INTO statement is used to insert new records in a table.
It is possible to write the INSERT INTO statement in **two** ways:

1. Specify both the column names and the values to be inserted:
INSERT INTO *table_name* (*column1*, *column2*, *column3*, ….
VALUES (*value1*, *value2*, *value3*, ...);

2. If you are adding values for all the columns of the table, you do not need to specify the column names in the SQL query. However, make sure the order of the values is in the same order as the columns in the table. Here, the INSERT INTO **syntax** would be as follows:

INSERT INTO *table_name*

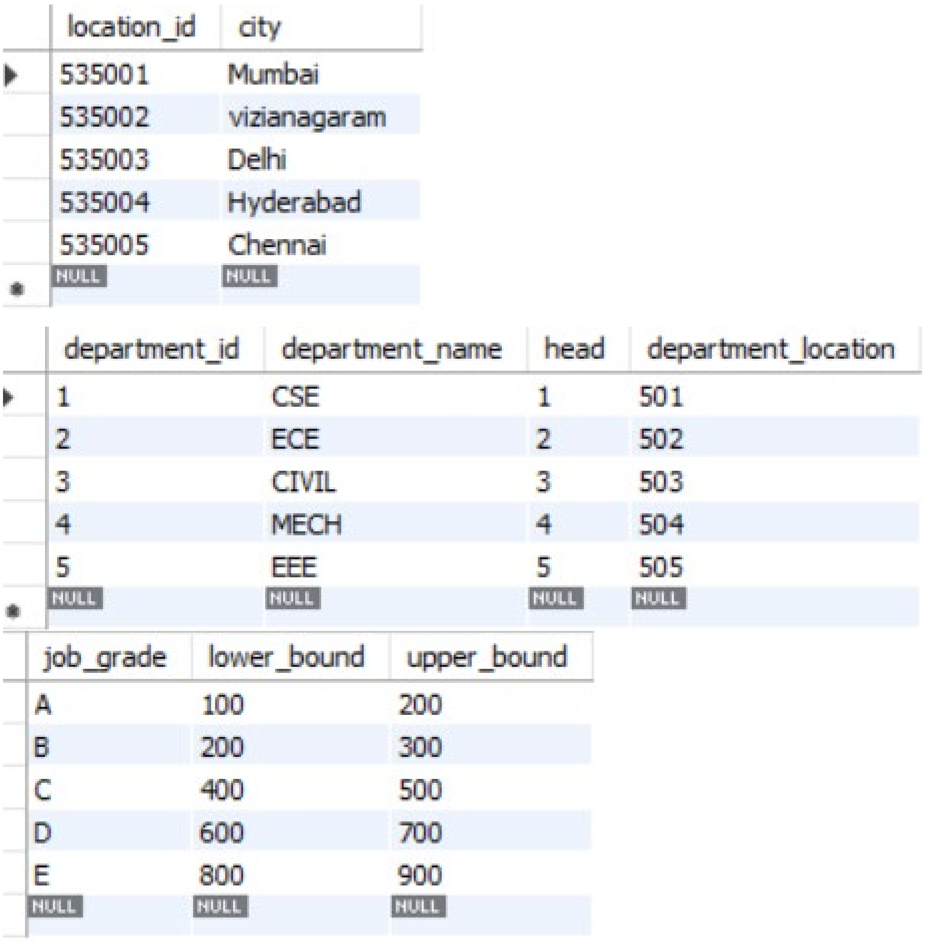
VALUES (*value1*, *value2*, *value3*, ...);

Query Command:

```
INSERT into location values(535002, "vizianagaram");
INSERT into location values(535001, "Mumbai");
INSERT into location values(535003, "Delhi");
INSERT into location values(535004, "Hyderabad");
INSERT into location values(535005, "Chennai");
SELECT  * from location;

INSERT into department values (001, "CSE", 1, 501);
INSERT into department values (002, "ECE", 2, 502);
INSERT into department values (003, "CIVIL", 3, 503);
INSERT into department values (004, "MECH", 4, 504);
INSERT into department values (005, "EEE", 5, 505);
SELECT  * from department;

INSERT into job_grade values ("A", 100, 200);
INSERT into job_grade values ("B", 200 , 300);
INSERT into job_grade values ("C", 400, 500);
INSERT into job_grade values ("D", 600, 700);
INSERT into job_grade values ("E", 800, 900);
SELECT * from job_grade;
```

```
INSERT into employee values (101,"Ashik","Ahamad","2003-05-25",80000,2201,"A",
001,535001);
INSERT into employee values (102,"Chandhan","lohit","2000-06-23",40000,2202,"B",
002,535002);
INSERT into employee values (103,"Satwik","Naidu","1999-5-18",60000,2203,"C",
003,535003);
INSERT into employee values (104,"Velamala","Karthik","2004-01-10",20000,2204,"D",
004,535004);
INSERT into employee values (105,"Suresh","Naidu","2002-03-30",50000,2205,"E",
005,535005);
INSERT into employee values (106,"Ashok","reddy","2002-03-30",50000,2205,"A",
001,535001);
INSERT into employee values (107,"Harsha","Vardhan","2003-05-30",80000,2202,"B",
001,535002);
INSERT into employee values (108,"Ashik","Ahamad","2007-06-18",30000,2203,"B",
002,535002);
INSERT into employee values (109,"lokesh","vegi","2000-06-18",60000,2204,"A",
003,535003);
SELECT  * from employee;
TRUNCATE employee;
```

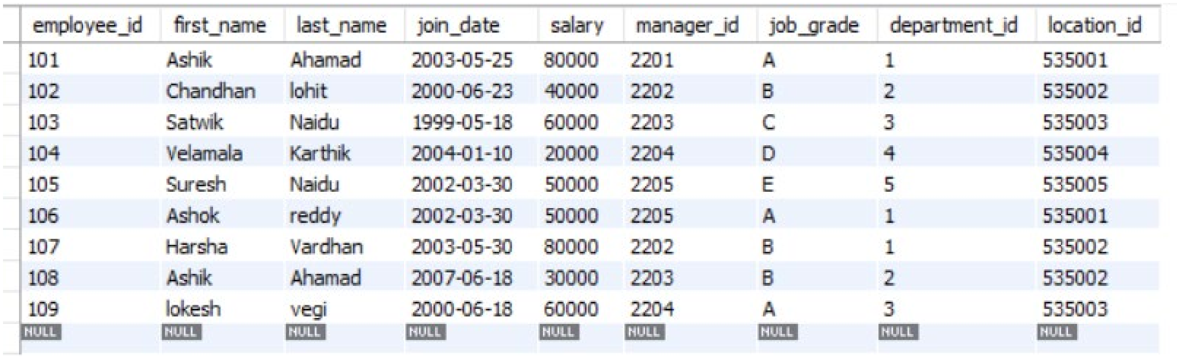
Output:



```
  location_id  city
▶ 535001       Mumbai
  535002       vizianagaram
  535003       Delhi
  535004       Hyderabad
  535005       Chennai
* NULL         NULL
```

```
  department_id  department_name  head  department_location
▶ 1              CSE              1     501
  2              ECE              2     502
  3              CIVIL            3     503
  4              MECH             4     504
  5              EEE              5     505
* NULL           NULL             NULL  NULL
```

```
job_grade  lower_bound  upper_bound
A          100          200
B          200          300
C          400          500
D          600          700
E          800          900
NULL       NULL         NULL
```



employee_id	first_name	last_name	join_date	salary	manager_id	job_grade	department_id	location_id
101	Ashik	Ahamad	2003-05-25	80000	2201	A	1	535001
102	Chandhan	lohit	2000-06-23	40000	2202	B	2	535002
103	Satwik	Naidu	1999-05-18	60000	2203	C	3	535003
104	Velamala	Karthik	2004-01-10	20000	2204	D	4	535004
105	Suresh	Naidu	2002-03-30	50000	2205	E	5	535005
106	Ashok	reddy	2002-03-30	50000	2205	A	1	535001
107	Harsha	Vardhan	2003-05-30	80000	2202	B	1	535002
108	Ashik	Ahamad	2007-06-18	30000	2203	B	2	535002
109	lokesh	vegi	2000-06-18	60000	2204	A	3	535003
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Inference:

We will learn how to insert records into Tables .

WEEK - 3

Inserting, Simple Select, Char, Number, Date functions

Experiment 6:

Aim:

Give a list of all employees (names as first_name, last_name) who belong to one department_id.

Description:

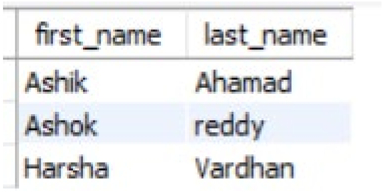
SELECT Syntax:

SELECT *column1*, *column2, ...*
FROM *table_name*;
Here, column1, column2, ... are the field names of the table you want to select data from. If you want to select all the fields available in the table,
use the following **syntax**:
SELECT * FROM *table_name*;

Query Command:

```
SELECT first_name, last_name from employee where department_id = 1;
```

Output:



```
first_name   last_name
Ashik        Ahamad
Ashok        reddy
Harsha       Vardhan
```

Inference:

We will learn how to write simple select statement, select statement with conditions and date, character, number functions.

Experiment 7:

Aim:

Select employee last_names from employee table who belong to a certain department_id and have a salary greater than 5000.

Description:

Syntax:

The basic syntax of the SELECT statement with the WHERE clause is as

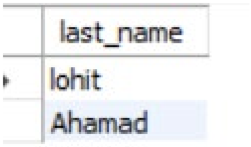
shown below.

```
SELECT column1, column2, columnN
FROM table_name
WHERE [condition]
```

Query Command:

```
SELECT last_name from employee where department_id = 2 and salary > 5000;
```

Output:



```
last_name
lohit
Ahamad
```

Inference:

We will learn how to write simple select statement, select statement with conditions and date, character, number functions.

Experiment 8:

Aim:

Select employee last_name with first letter in capital, all smalls and all capitals from employee table for all employees.

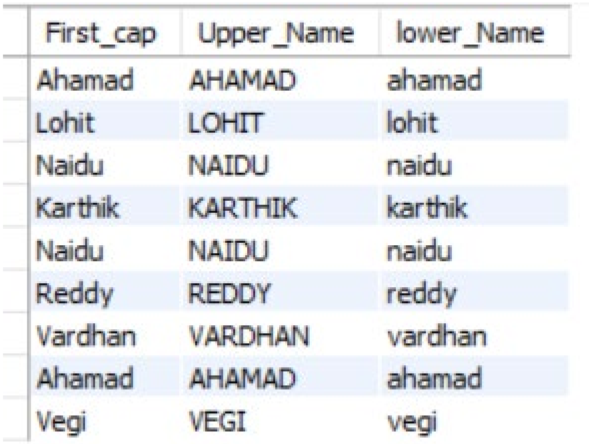
Description:

SQL provides a rich set of character functions that allow you to get information about strings and modify the contents of those strings in multiple ways. Character functions are of the following two types:
1. Case-Manipulative Functions (LOWER, UPPER and INITCAP)
2. Character-Manipulative Functions (CONCAT, LENGTH, SUBSTR, INSTR, LPAD, RPAD, TRIM and REPLACE)

Query Command:

```
SELECT concat(upper(substring(last_name,1,1)),lower(substring(last_name,2)))
First_cap, upper(last_name) Upper_Name, lower(last_name) lower_Name from employee;
```

Output:



First_cap	Upper_Name	lower_Name
Ahamad	AHAMAD	ahamad
Lohit	LOHIT	lohit
Naidu	NAIDU	naidu
Karthik	KARTHIK	karthik
Naidu	NAIDU	naidu
Reddy	REDDY	reddy
Vardhan	VARDHAN	vardhan
Ahamad	AHAMAD	ahamad
Vegi	VEGI	vegi

Inference:

We will learn how to write simple select statement, select statement with conditions and date, character, number functions.

Experiment 9:

Aim:

Select the salary and additional HRA (7.5% of the salary) for each employee in employee table
rounded to a whole number.

Description:

Numeric Functions are used to perform operations on numbers and return numbers.

ROUND():
It returns a number rounded to a certain number of decimal places.

Syntax:

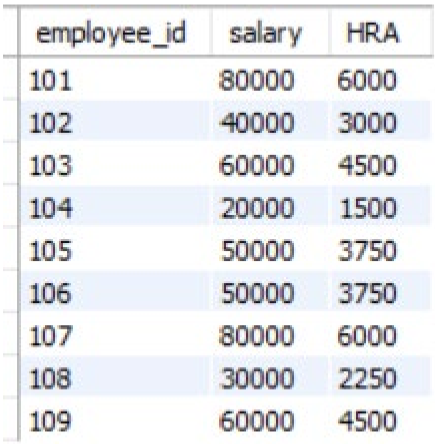
SELECT ROUND(5.553);

Output:6

Query Command:

```
SELECT employee_id, salary, round(salary*0.075) HRA from employee;
```

Output:



employee_id	salary	HRA
101	80000	6000
102	40000	3000
103	60000	4500
104	20000	1500
105	50000	3750
106	50000	3750
107	80000	6000
108	30000	2250
109	60000	4500

Inference:

We will learn how to write simple select statement, select statement with conditions and date, character, number functions.

Experiment 10:

Aim:

Select employee last_name, join_date, and the number of days he/she has been working in the firm as of today.

Description:

The date function DATEDIFF accepts a date part, start date and end date as date datetime, or valid date string and returns the difference between the dates in units based on the date part specified.

Syntax: DATEDIFF (date part, start date, end date)

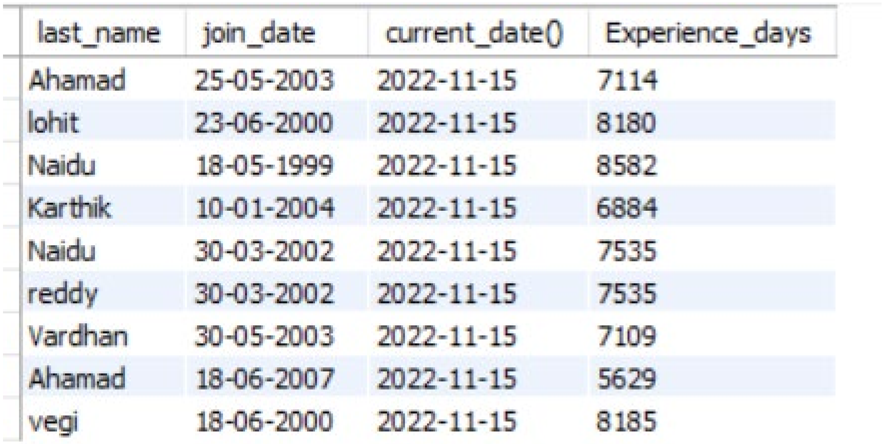
Date Parts: can use the name or listed abbreviations:

- year, yy, yyyy
- quarter, qq, q
- month, mm, m

Query Command:

```
SELECT last_name, date_format(join_date, "%d-%m-%Y") as join_date, current_date(),
datediff(current_date(),join_date) Experience_days from employee;
```

Output:



last_name	join_date	current_date()	Experience_days
Ahamad	25-05-2003	2022-11-15	7114
lohit	23-06-2000	2022-11-15	8180
Naidu	18-05-1999	2022-11-15	8582
Karthik	10-01-2004	2022-11-15	6884
Naidu	30-03-2002	2022-11-15	7535
reddy	30-03-2002	2022-11-15	7535
Vardhan	30-05-2003	2022-11-15	7109
Ahamad	18-06-2007	2022-11-15	5629
vegi	18-06-2000	2022-11-15	8185

Inference:

We will learn how to write simple select statement, select statement with conditions and date, character, number functions.

WEEK - 4

Detailed SELECT with subqueries, EQUI-JOINS, correlated subqueries.

Experiment 11:

Aim:

Select employee last_name of all employees whose salary is greater than the salary of employee with id = 2.

Description:

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

- A subquery may occur in:
  - A SELECT clause
  - A FROM clause
  - A WHERE clause

A single-row subquery is used when the outer query's results are based on a single, unknown value. Although this query type is formally called "single-row," the name implies that the query returns multiple columns-but only one row of results. However, a single-row subquery can return only one row of results consisting of only one column to the outer query.

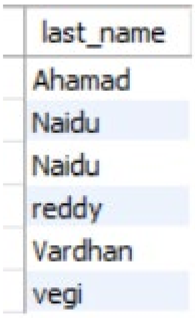
Syntax: Select column_names from table_name where (inner_query);

```
SELECT first_name, salary, department_id FROM employees WHERE salary = (SELECT MIN (salary)FROM employees);
```

Query Command:

```
select last_name from employee where salary > (select salary from employee where employee_id  = 102);
```

Output:



last_name
Ahamad
Naidu
Naidu
reddy
Vardhan
vegi

Inference:

We will learn how to write simple select statement, select statement with conditions and date, character, number functions and gain knowledge on sub-queries and understand about joins

Experiment 12:

Aim:

Select all employees whose salary is greater than the salaries of both employees with ids 2 &3.

Description:

Multiple Row Sub Query

Multiple-row subqueries are nested queries that can return more than one row of results to the parent query. Multiple-row subqueries are used most commonly in WHERE and HAVING clauses. Since it returns multiple rows, it must be handled by set comparison operators (IN, ALL, ANY).While IN operator holds the same meaning as discussed in the earlier chapter, ANY operator compares a specified value to each value returned by the subquery while ALL compares a value to every value returned by a subquery. The below query will show the error because single-row subquery returns multiple rows.

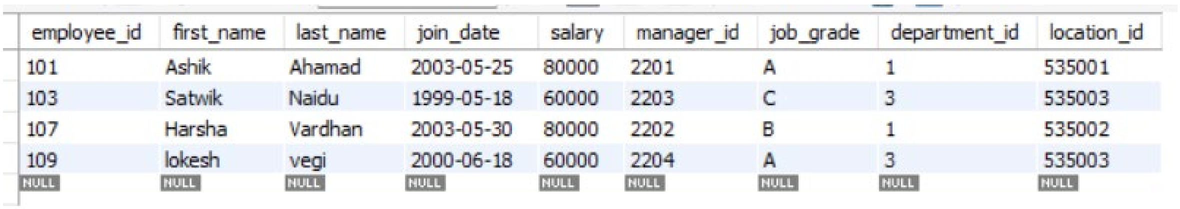
Syntax: Select column_names from table_name where column_names (operators) inner_query;

SELECT first_name, department_id FROM employees WHERE department_id = (SELECT department_id FROM employees WHERE LOCATION_ID = 100)

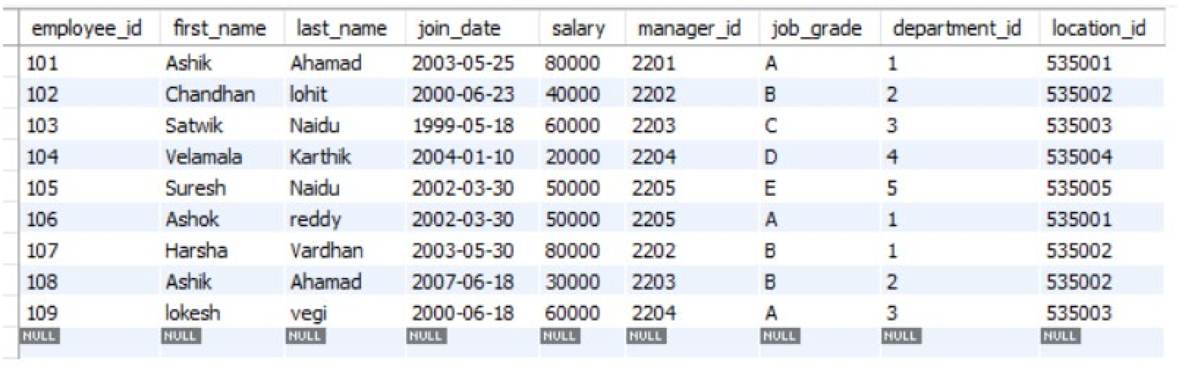
Query Command:

SELECT * from employee where salary > all (select salary from employee where employee_id in (102,105));
SELECT * from employee;

Output:



employee_id	first_name	last_name	join_date	salary	manager_id	job_grade	department_id	location_id
101	Ashik	Ahamad	2003-05-25	80000	2201	A	1	535001
103	Satwik	Naidu	1999-05-18	60000	2203	C	3	535003
107	Harsha	Vardhan	2003-05-30	80000	2202	B	1	535002
109	lokesh	vegi	2000-06-18	60000	2204	A	3	535003
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL



employee_id	first_name	last_name	join_date	salary	manager_id	job_grade	department_id	location_id
101	Ashik	Ahamad	2003-05-25	80000	2201	A	1	535001
102	Chandhan	lohit	2000-06-23	40000	2202	B	2	535002
103	Satwik	Naidu	1999-05-18	60000	2203	C	3	535003
104	Velamala	Karthik	2004-01-10	20000	2204	D	4	535004
105	Suresh	Naidu	2002-03-30	50000	2205	E	5	535005
106	Ashok	reddy	2002-03-30	50000	2205	A	1	535001
107	Harsha	Vardhan	2003-05-30	80000	2202	B	1	535002
108	Ashik	Ahamad	2007-06-18	30000	2203	B	2	535002
109	lokesh	vegi	2000-06-18	60000	2204	A	3	535003
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Inference:

We will learn how to write simple select statement, select statement with conditions and date, character, number functions and gain knowledge on sub-queries and understand about joins

Experiment 13:

Aim:

Select employee lastname and the corresponding department_name for all employees in employees table.

Description:

The process is called joining when we combine two or more tables based on some common columns and a join condition. An equijoin is an operation that combines multiple tables based on equality or matching column values in the associated tables.
We can use the equal sign (=) comparison operator to refer to equality in the WHERE clause. This joining operation returns the same result when we use the JOIN keyword with the ON clause and then specifying the column names and their associated tables.

Syntax:

1. SELECT column_name (s)
2. FROM table_name1, table_name2, ..., table_nameN
3. WHERE table_name1.column_name = table_name2.column_name;

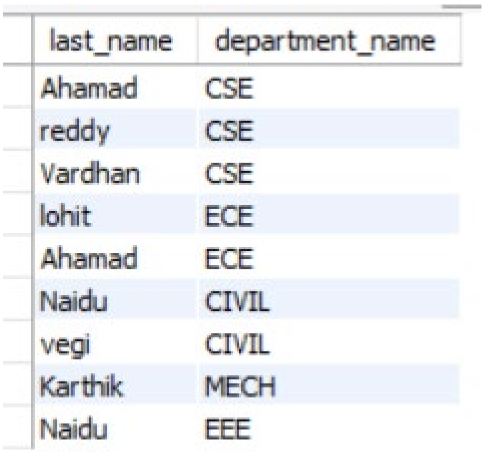
OR

1. SELECT (column_list | *)
2. FROM table_name1
3. JOIN table_name2
4. ON table_name1.column_name = table_name2.column_name;

Query Command:

SELECT emp.last_name, dep.department_name from employee emp inner join department dep on emp.department_id = dep.department_id;

Output:



last_name	department_name
Ahamad	CSE
reddy	CSE
Vardhan	CSE
lohit	ECE
Ahamad	ECE
Naidu	CIVIL
vegi	CIVIL
Karthik	MECH
Naidu	EEE

Inference:

We will learn how to write simple select statement, select statement with conditions and date, character, number functions and gain knowledge on sub-queries and understand about joins

Experiment 14:

Aim:

Select all employees whose salary is lesser than all employees in the same job grade.

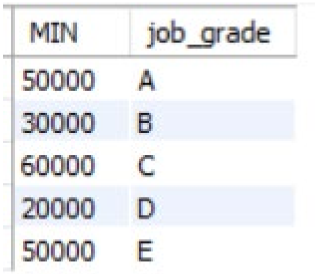
Description:

Use the correlated sub-query where the inner query will be made based on the data coming from the upper query. In this case, the salary and grade of each employee row is used to execute the inner query to compare if there are any employees in the same grade who have lesser salary using exists clause.

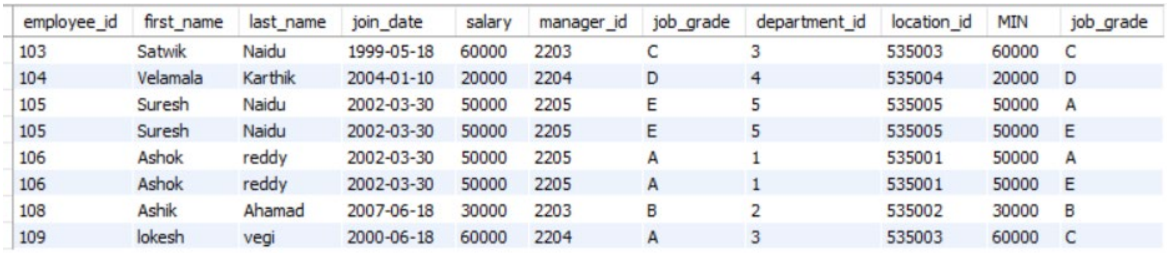
Query Command:

```
CREATE VIEW MIN_TABLE AS SELECT MIN(Salary) AS MIN,job_grade FROM
Employee GROUP BY(job_grade);
SELECT * FROM MIN_TABLE;
SELECT * FROM Employee INNER JOIN MIN_TABLE ON Employee.salary =
MIN_TABLE.MIN;
```

Output:



MIN	job_grade
50000	A
30000	B
60000	C
20000	D
50000	E



employee_id	first_name	last_name	join_date	salary	manager_id	job_grade	department_id	location_id	MIN	job_grade
103	Satwik	Naidu	1999-05-18	60000	2203	C	3	535003	60000	C
104	Velamala	Karthik	2004-01-10	20000	2204	D	4	535004	20000	D
105	Suresh	Naidu	2002-03-30	50000	2205	E	5	535005	50000	A
105	Suresh	Naidu	2002-03-30	50000	2205	E	5	535005	50000	E
106	Ashok	reddy	2002-03-30	50000	2205	A	1	535001	50000	A
106	Ashok	reddy	2002-03-30	50000	2205	A	1	535001	50000	E
108	Ashik	Ahamad	2007-06-18	30000	2203	B	2	535002	30000	B
109	lokesh	vegi	2000-06-18	60000	2204	A	3	535003	60000	C

Inference:

We will learn how to write simple select statement, select statement with conditions and date, character, number functions andl gain knowledge on sub-queries and understand about joins

WEEK - 5

GROUPING, SET, UPDATE, DELETE, VIEWS

Experiment 15:

Aim:

Select the average salary of all employees in department with department_id = 2.

Description:

The avg() function has the following syntax: SELECT AVG(column_name) FROM table_name; The avg() function can be used with the SELECT query for retrieving data from a table.

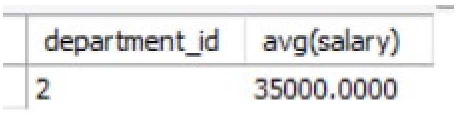
Syntax:

AVG(*expression*)

Query Command:

SELECT emp.department_id, avg(salary) from employee emp inner join department dep on dep.department_id = emp. department_id where dep.department_id =2;

Output:



```
department_id   avg(salary)
2               35000.0000
```

Inference:

We will get an exposure to clauses, views and aggregate functions.

Experiment 16:

Aim:

Select AVG salary of each department which has employees in employee table.

Description:

Avg: The avg() function has the following syntax: SELECT AVG(column_name) FROM table_name; The avg() function can be used with the SELECT query for retrieving data from a table.

Syntax:

AVG(*expression*)

Group By: The GROUP BY statement groups rows that have the same values into summary rows, like "find the number of customers in each country".

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.

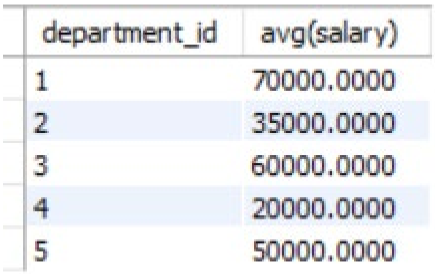
Syntax:

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

Query Command:

```
SELECT department_id, avg(salary) from employee group by department_id;
```

Output:



department_id	avg(salary)
1	70000.0000
2	35000.0000
3	60000.0000
4	20000.0000
5	50000.0000

Inference:

We will get an exposure to clauses, views and aggregate functions.

Experiment 17:

Aim:

Select minimum salary of all departments where the minimum salary is less than 1000.

Description:

The MIN() function returns the smallest value of the selected column.

The MAX() function returns the largest value of the selected column.

MIN() Syntax

SELECT MIN(*column_name*)

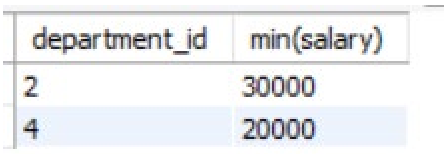
FROM *table_name*

WHERE *condition*;

Query Command:

SELECT department_id, min(salary) from employee group by department_id having min(salary)<50000;

Output:



department_id	min(salary)
2	30000
4	20000

Inference:

We will get an exposure to clauses, views and aggregate functions.

Experiment 18:

Aim:

Give a list of all employees who earn a salary greater than 10000 or work in job grade MANAGER.

Description:

The Union is a binary set operator in DBMS. It is used to combine the result set of two select queries. Thus, It combines two result sets into one. In other words, the result set obtained after union operation is the collection of the result set of both the tables.

But two necessary conditions need to be fulfilled when we use the union command. These are:

1. Both SELECT statements should have an equal number of fields in the same order.
2. The data types of these fields should either be the same or compatible with each other.

The syntax for the union operation is as follows:

SELECT (coloumn_names) from table1 [WHERE condition] UNION SELECT (coloumn_names) from table2 [WHERE condition];

The MySQL query for the union operation can be as follows:

SELECT color_name FROM colors_a UNION SELECT color_name FROM colors_b;

Intersect is a binary set operator in DBMS. The intersection operation between two selections returns only the common data sets or rows between them. It should be noted that the intersection operation always returns the distinct rows. The duplicate rows will not be returned by the intersect operator.

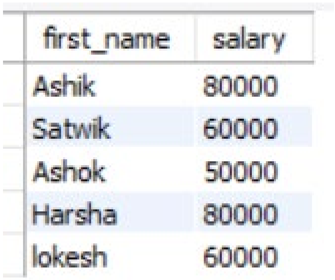
The syntax for the intersection operation is as follows:

SELECT (coloumn_names) from table1[WHERE condition] INTERSECT SELECT (coloumn_names) from table2 [WHERE condition];

Query Command:

```
SELECT first_name, salary from employee where salary>50000 or job_grade = "A";
```

Output:



first_name	salary
Ashik	80000
Satwik	60000
Ashok	50000
Harsha	80000
lokesh	60000

Inference:

We will get an exposure to clauses, views and aggregate functions.

Experiment 19:

Aim:

Create a view that shows all employees of department_id = 10 and select from the view.

Description:

In SQL, a view is a virtual table based on the result-set of an SQL statement.

A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database.

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

A view is created with the CREATE VIEW statement.

CREATE VIEW Syntax

REATE VIEW view_name AS

ELECT column1, column2, ...

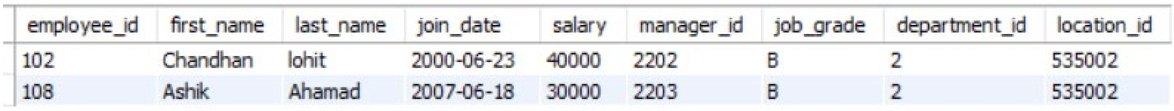
ROM table_name

VHERE condition;

Query Command:

CREATE view dep_view as (select e.* from employee e inner join department d on e.department_id = d.department_id where e.department_id =2);
SELECT * from dep_view;

Output:



employee_id	first_name	last_name	join_date	salary	manager_id	job_grade	department_id	location_id
102	Chandhan	lohit	2000-06-23	40000	2202	B	2	535002
108	Ashik	Ahamad	2007-06-18	30000	2203	B	2	535002

Inference:

We will get an exposure to clauses, views and aggregate functions.

WEEK - 7

Iterative PL/SQL Blocks and functions.

Experiment 20:

Aim:

Write a PL/SQL block which declares a variable and reads the last name of employee with id =5 and outputs that to standard output.

Description:

A PL/SQL block consists of three sections: declaration, executable, and exception-handling sections. In a block, the executable section is mandatory while the declaration and exception-handling sections are optional. A PL/SQL block has a name. Functions or Procedures is an example of a named block.

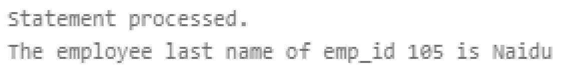
Syntax:

```
DECLARE
  <declarations section>
BEGIN
  <executable command(s)>
EXCEPTION
  <exception handling>
END;
```

Query Command:

```
DECLARE
L_NAME EMPLOYEE.LAST_NAME%TYPE;
BEGIN
SELECT LAST_NAME INTO L_NAME FROM EMPLOYEE WHERE EMPLOYEE_ID
= 105;
dbms_output.put_line('The employee last name of emp_id ' || 105 || ' is ' || L_NAME);
END;
```

Output:



```
Statement processed.
The employee last name of emp_id 105 is Naidu
```

Inference:

We will gain an exposure of Iterative Blocks in PL/SQL

Experiment 21:

Aim:

Write a PL/SQL block which declares a variable with a value and prints in all capitals if the value starts with 'S', in all smalls if it starts with 'R', and in initial capitals if otherwise.

Description:

It is always legal in PL/SQL programming to nest the IF-ELSE statements, which means you can use one IF or ELSE IF statement inside another IF or ELSE IF statement(s).

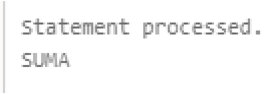
Syntax:

```
IF( boolean_expression 1)THEN
  -- executes when the boolean expression 1 is true
  IF(boolean_expression 2) THEN
    -- executes when the boolean expression 2 is true
    sequence-of-statements;
  END IF;
ELSE
  -- executes when the boolean expression 1 is not true
  else-statements;
END IF;
```

Query Command:

```
 DECLARE
name varchar2(10) := 'Suma';
BEGIN
case
when name LIKE 'S%' THEN dbms_output.put_line(UPPER(NAME));
WHEN name LIKE 'R%' then dbms_output.put_line(lower(name));
else dbms_output.put_line(initcap(name));
end case;
end;
```

Output:



```
Statement processed.
SUMA
```

Inference:

We will gain an exposure of Iterative Blocks in PL/SQL

Experiment 22:

Aim:

Write a PL/SQL block which declares two variables with values and prints first value if it is not null and prints second value if first is null.

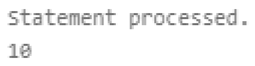
Description:

By using PL/SQL commands we should create a block which declares two variables with values in declare section and prints first value if it is not null or else it prints the second value. For that we use NULLIF function in IF THEN ELSE condition in the execution section.

Query Command:

```
DECLARE
a number := 10;
b number := 20;
BEGIN
IF (a is not null) then dbms_output.put_line(a);
else dbms_output.put_line(b);
END IF;
END;
```

Output:



```
Statement processed.
10
```

Inference:

We will gain an exposure of Iterative Blocks in PL/SQL

Experiment 23:

Aim:

Write a PL/SQL block which declares a variable and reads the last name of employees and outputs that to standard output.

Description:

A WHILE And FOR LOOP statement in PL/SQL programming language repeatedly executes a target statement as long as a given condition is true.

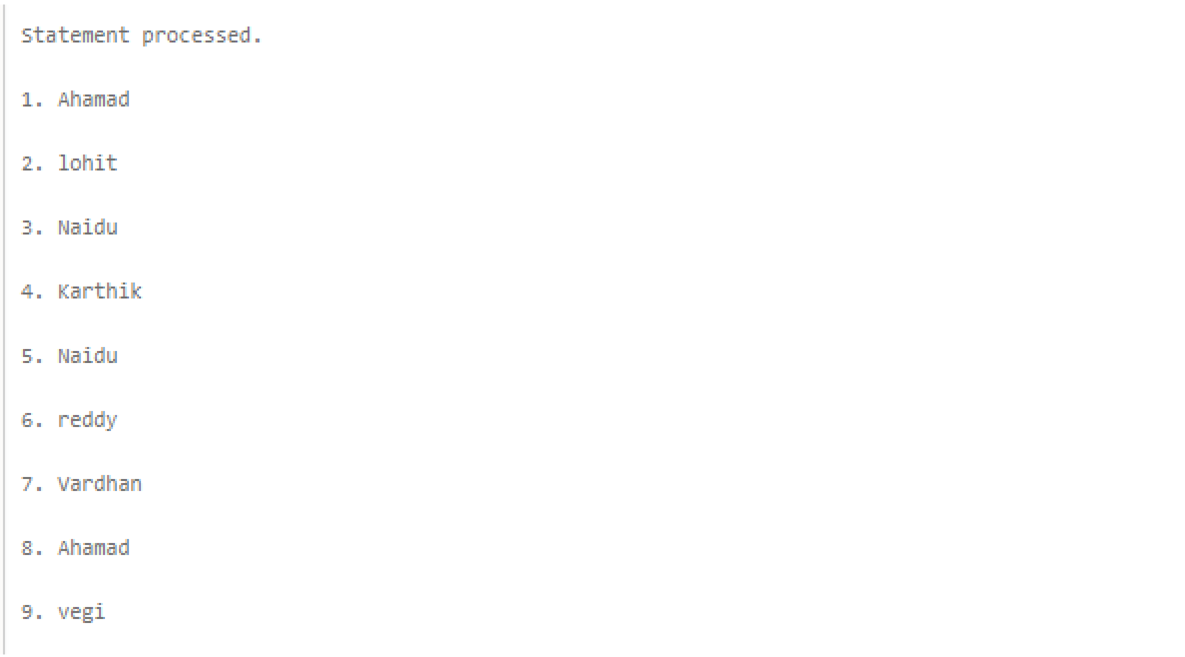
Syntax's:

```
WHILE condition LOOP
  sequence_of_statements
END LOOP;

FOR counter IN initial_value .. final_value LOOP
  sequence_of_statements;

END LOOP;.
```

Query Command:

```
DECLARE
i int := 0;
L_NAME EMPLOYEE.LAST_NAME%TYPE;
CURSOR EMP IS SELECT LAST_NAME FROM EMPLOYEE;
BEGIN
OPEN EMP;
LOOP
   i := i+1;
   FETCH EMP INTO L_NAME;
   EXIT WHEN EMP%NOTFOUND;
   dbms_output.put_line(CHR(10) || i|| '. ' || L_NAME);
END LOOP;
CLOSE EMP;
END;
```

Output:



```
Statement processed.

1. Ahamad

2. lohit

3. Naidu

4. Karthik

5. Naidu

6. reddy

7. Vardhan

8. Ahamad

9. vegi
```

Inference:

We will gain an exposure of Iterative Blocks in PL/SQL

WEEK - 8

Students will gain an exposure to give a transaction support in PL/SQL

Experiment 24:

Aim:

Write a PL/SQL block which updates 2 tables. If the second update fails the first update also has to be reversed.

Description:

COMMIT statement permanently save the state, when all the statements are executed successfully without any error.
Syntax of COMMIT statement are:

COMMIT;

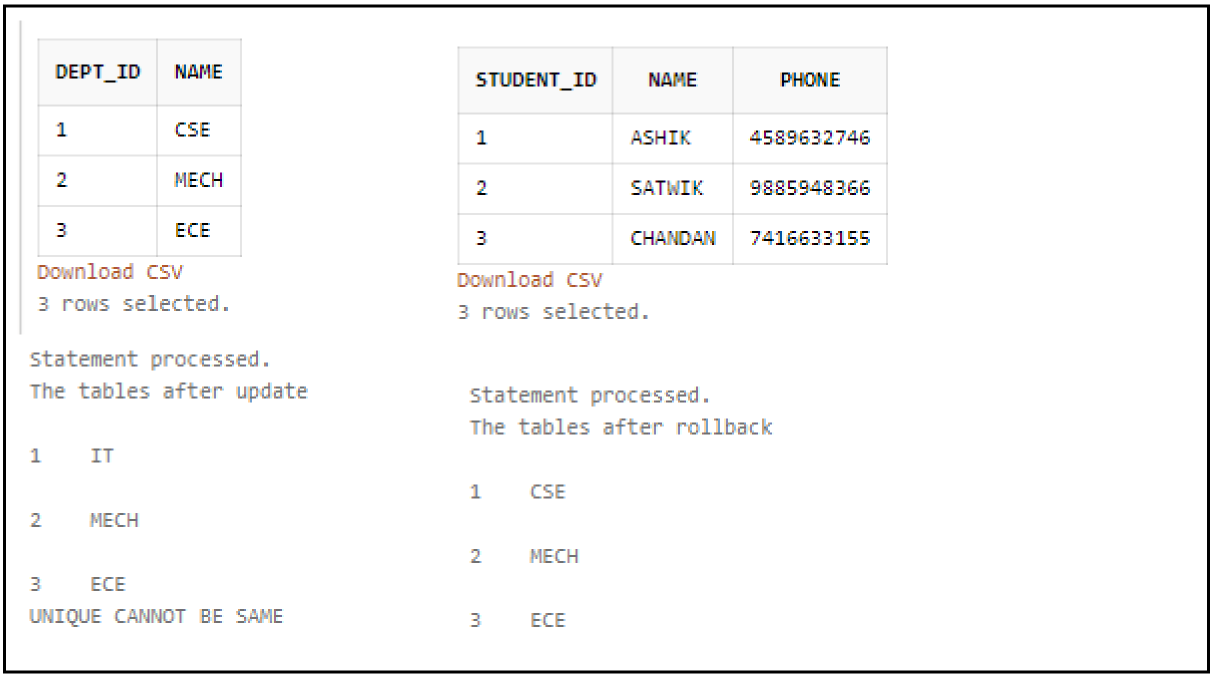
In ROLLBACK statement if any operations fail during the completion of a transaction, it cannot permanently save the change and we can undo them using this statement.
Syntax of ROLLBACK statement are:

ROLLBACK;

Query Command:

```
CREATE TABLE DEPT(DEPT_ID NUMBER PRIMARY KEY, NAME
VARCHAR2(5));
CREATE TABLE STUDENT(STUDENT_ID NUMBER PRIMARY KEY, NAME
VARCHAR2(10), PHONE INT UNIQUE);
INSERT ALL
INTO DEPT (DEPT_ID, NAME) VALUES (1, 'CSE')
INTO DEPT (DEPT_ID, NAME) VALUES (2, 'MECH')
INTO DEPT (DEPT_ID, NAME) VALUES (3, 'ECE')
SELECT * FROM DUAL;
INSERT ALL
INTO STUDENT (STUDENT_ID, NAME, PHONE) VALUES (1, 'ASHIK', 4589632746)
INTO STUDENT (STUDENT_ID, NAME, PHONE) VALUES (2, 'SATWIK',
9885948366)
INTO STUDENT (STUDENT_ID, NAME, PHONE) VALUES (3, 'CHANDAN',
7416633155)
SELECT 1  FROM DUAL;

SELECT * FROM DEPT;
SELECT * FROM STUDENT;
DECLARE
   CURSOR DEPT1 IS SELECT * FROM DEPT;
BEGIN
   COMMIT;
   UPDATE DEPT SET NAME = 'IT' WHERE DEPT_ID = 1;
   dbms_output.put_line('The tables after update');
   FOR i IN DEPT1
   LOOP
      dbms_output.put_line(CHR(10) || i.DEPT_ID || '    ' || i.NAME);
   END LOOP;
   UPDATE STUDENT SET PHONE = 9885948366 WHERE STUDENT_ID = 1;

   EXCEPTION
   WHEN DUP_VAL_ON_INDEX THEN
      dbms_output.put_line('UNIQUE CANNOT BE SAME');
      ROLLBACK;

END;
/
BEGIN
   dbms_output.put_line('The tables after rollback ');
   FOR i IN (SELECT * FROM DEPT)
   LOOP
      dbms_output.put_line(CHR(10) || i.DEPT_ID || '    ' || i.NAME);
   END LOOP;
END;
/
```

Output:



DEPT_ID	NAME
1	CSE
2	MECH
3	ECE

```
Download CSV
3 rows selected.

Statement processed.
The tables after update

1    IT

2    MECH

3    ECE
UNIQUE CANNOT BE SAME
```

STUDENT_ID	NAME	PHONE
1	ASHIK	4589632746
2	SATWIK	9885948366
3	CHANDAN	7416633155

```
Download CSV
3 rows selected.

Statement processed.
The tables after rollback

1    CSE

2    MECH

3    ECE
```

Inference:

We will gain an exposure to give a transaction support in PL/SQL

Experiment 25:

Aim:

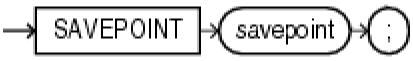
Write a PL/SQL block which updates 3 tables. If the third update fails the second update has to be reversed while first should not get effected.

Description:

Savepoint names must be distinct within a given transaction. If you create a second savepoint with the same identifier as an earlier savepoint, then the earlier savepoint is erased. After a savepoint has been created, you can either continue processing, commit your work, roll back the entire transaction, or roll back to the savepoint.

Syntax

***savepoint*::=**

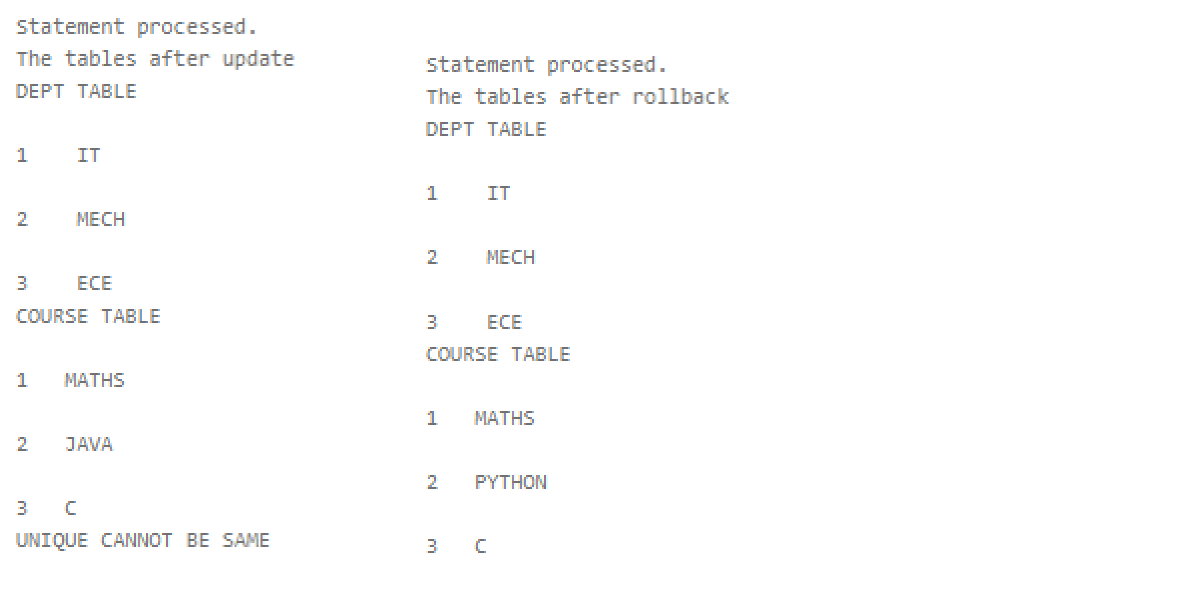


Query Command:

```
CREATE TABLE COURSE(COURSE_ID NUMBER PRIMARY KEY, NAME
VARCHAR2(10));
INSERT ALL
INTO COURSE (COURSE_ID, NAME) VALUES (1, 'MATHS')
INTO COURSE (COURSE_ID, NAME) VALUES (2, 'PYTHON')
INTO COURSE (COURSE_ID, NAME) VALUES (3, 'C')
SELECT * FROM DUAL;

DECLARE
  CURSOR DEPT1 IS SELECT * FROM DEPT;
BEGIN
   UPDATE DEPT SET NAME = 'IT' WHERE DEPT_ID = 1;
   savepoint s1;
   dbms_output.put_line('The tables after update');
   dbms_output.put_line('DEPT TABLE');
   FOR i IN DEPT1
   LOOP
      dbms_output.put_line(CHR(10) || i.DEPT_ID || '    ' || i.NAME);
   END LOOP;

   dbms_output.put_line('COURSE TABLE');
   UPDATE COURSE SET NAME = 'JAVA' WHERE COURSE_ID = 2;
   FOR j IN (SELECT * FROM COURSE)
   LOOP
```

```
        dbms_output.put_line(CHR(10) || j.COURSE_ID||'   '||j.NAME);
    end loop;
    UPDATE STUDENT SET PHONE = 9885948366 WHERE STUDENT_ID = 1;

    EXCEPTION
    WHEN DUP_VAL_ON_INDEX THEN
        dbms_output.put_line('UNIQUE CANNOT BE SAME');
        ROLLBACK to s1;

END;
/
BEGIN
    dbms_output.put_line('The tables after rollback ');
    dbms_output.put_line('DEPT TABLE');
    FOR i IN (SELECT * FROM DEPT)
    LOOP
        dbms_output.put_line(CHR(10) || i.DEPT_ID || '    ' || i.NAME);
    END LOOP;
    dbms_output.put_line('COURSE TABLE');
    FOR j IN (SELECT * FROM COURSE)
    LOOP
        dbms_output.put_line(CHR(10) || j.course_ID||'   '||j.NAME);
    end loop;
END;
/
```

Output:



```
Statement processed.
The tables after update
DEPT TABLE

1    IT

2    MECH

3    ECE
COURSE TABLE

1   MATHS

2   JAVA

3   C
UNIQUE CANNOT BE SAME
```

```
Statement processed.
The tables after rollback
DEPT TABLE

1    IT

2    MECH

3    ECE
COURSE TABLE

1   MATHS

2   PYTHON

3   C
```

Inference:

We will gain an exposure to give a transaction support in PL/SQL

WEEK - 9

Experiment 26:

Aim:

Write a PL/SQL block with a simple Exception Block.

Description:

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

- System-defined exceptions
- User-defined exceptions

Syntax for Exception Handling

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

Query Command:

```
DECLARE
   a number := 10;
   b number := 0;
BEGIN
   dbms_output.put_line(a/b);
   EXCEPTION
      WHEN ZERO_DIVIDE THEN
         dbms_output.put_line('CANNOT DIVIDE BY ZERO');
END;
/
```

Output:

```
Statement processed.
CANNOT DIVIDE BY ZERO
```

Inference:

We will gain an exposure of how to use Exceptions in oracle 9.

Experiment 27:

Aim:

Write a PL/SQL block which declares a variable and reads the last name of employee with id = 5. If there is no employee with id = 5 an error should be thrown. Otherwise the name has to be printed

Description:

In PL/SQL built in exceptions or you make user define exception. Examples of built-in type (internally) defined exceptions division by zero, out of memory. Some common built-in exceptions have predefined names such as ZERO_DIVIDE and STORAGE_ERROR.

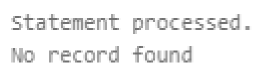
Syntax:

```
DECLARE
   declaration statement(s);
BEGIN
   statement(s);
EXCEPTION
   WHEN built-in_exception_name_1 THEN
      User defined statement (action) will be taken;
   WHEN built-in_exception_name_2 THEN
      User defined statement (action) will be taken;

END;
```

Query Command:

```
DECLARE
   L_NAME EMPLOYEE.LAST_NAME%TYPE;
BEGIN
   SELECT LAST_NAME INTO L_NAME FROM EMPLOYEE WHERE
EMPLOYEE_ID = 5;
   dbms_output.put_line(L_NAME);
   exception
      WHEN NO_DATA_FOUND THEN
         dbms_output.put_line('No record found');
END;
/
```

Output:



```
Statement processed.
No record found
```

Inference:

We will gain an exposure of how to use Exceptions in oracle 9.

Experiment 28:

Aim:

Write a PL/SQL block which declares a variable and reads the last name of employee with id = 5. If the name has 8 characters, raise an exception called too many characters and handle it by cutting the last_name to first eight characters.

Description:

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

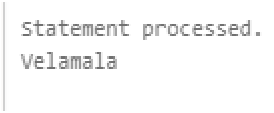
The syntax for declaring an exception is −

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

Query Command:

```
DECLARE
   L_NAME EMPLOYEE.FIRST_NAME%TYPE;
   TOO_MANY_CHARACTERS EXCEPTION;
BEGIN
   SELECT FIRST_NAME INTO L_NAME FROM EMPLOYEE WHERE
EMPLOYEE_ID = 104;
   IF LENGTH(L_NAME) >= 8 THEN
      RAISE TOO_MANY_CHARACTERS;
   ELSE
      dbms_output.put_line(L_NAME);
   END IF;
   exception
      WHEN TOO_MANY_CHARACTERS THEN
         dbms_output.put_line(SUBSTR(L_NAME,1,8));
END;
```

Output:



```
Statement processed.
Velamala
```

Inference:

We will gain an exposure of how to use Exceptions in oracle 9.

Experiment 29:

Aim:

Think of a problem that generates an exception is caught but an application error has to be raised because the exception in fatal.

Description:

Raising Exceptions:-

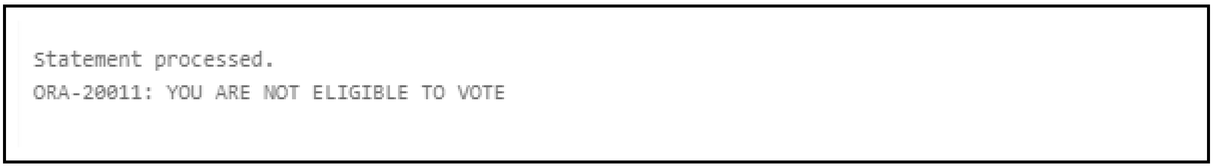
Exceptions are raised by the database server automatically whenever there is any internal database error, but exceptions can be raised explicitly by the programmer by using the command RAISE. Following is the simple syntax for raising an exception −

```
DECLARE
  exception_name EXCEPTION;
BEGIN
  IF condition THEN
    RAISE exception_name;
  END IF;
EXCEPTION
  WHEN exception_name THEN
  statement;
END;
```

Query Command:

```
DECLARE
   AGE NUMBER := 10;
BEGIN
   IF AGE < 18 THEN
      RAISE_APPLICATION_ERROR(-20011, 'YOU ARE NOT ELIGIBLE TO VOTE');
   END IF;
   dbms_output.put_line('you are eligible to vote');
   EXCEPTION
      WHEN OTHERS THEN
      dbms_output.put_line(SQLERRM);
END;
```

Output:



```
Statement processed.
ORA-20011: YOU ARE NOT ELIGIBLE TO VOTE
```

Inference:

We will gain an exposure of how to use Exceptions in oracle 9.

WEEK - 10

Experiment 30:

Aim:

Create a stored procedure which takes deparment_id as parameter, inserts all employees of that department in a table called dept_employee with the same structure.

Description:

Stored Procedure Parameters: Input, Output, Optional

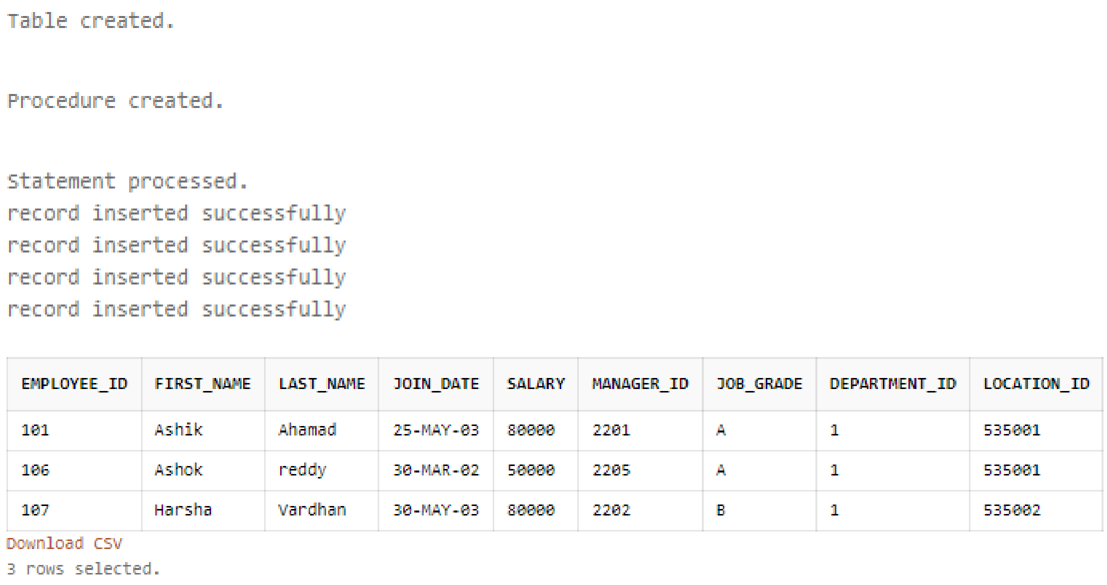
1. A stored procedure can have zero or more INPUT and OUTPUT parameters.
2. A stored procedure can have a maximum of 2100 parameters specified.
3. Each parameter is assigned a name, a data type, and direction like Input, Output, or Return.

Syntax:
CREATE PROCEDURE *procedure_name (parameters [mode] datatype ...)*
AS
sql_statement
GO;

Query Command:

```
CREATE TABLE DEPT_EMPLOYEE AS SELECT * FROM EMPLOYEE WHERE 1 =
0;
CREATE OR REPLACE PROCEDURE INSERTION (DEPT_ID IN
DEPARTMENT.DEPARTMENT_ID%TYPE)
--insert into dept_employee (select * from employee);
IS
BEGIN
   FOR i IN (SELECT * FROM EMPLOYEE INNER JOIN DEPARTMENT USING
(DEPARTMENT_ID) WHERE DEPARTMENT_ID = DEPT_ID)
   LOOP
      dbms_output.put_line('record inserted successfully');
      INSERT INTO DEPT_EMPLOYEE VALUES (i.employee_id, i.first_name,
i.last_name, i.join_date, i.salary, i.manager_id,
      i.job_grade, i.department_id, i.location_id);
   END LOOP;
END INSERTION;
/
BEGIN
   INSERTION(1);
   dbms_output.put_line('record inserted successfully');
end;
/
SELECT * FROM DEPT_EMPLOYEE;
```

Output:



```
Table created.


Procedure created.


Statement processed.
record inserted successfully
record inserted successfully
record inserted successfully
record inserted successfully
```

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	JOIN_DATE	SALARY	MANAGER_ID	JOB_GRADE	DEPARTMENT_ID	LOCATION_ID
101	Ashik	Ahamad	25-MAY-03	80000	2201	A	1	535001
106	Ashok	reddy	30-MAR-02	50000	2205	A	1	535001
107	Harsha	Vardhan	30-MAY-03	80000	2202	B	1	535002

```
Download CSV
3 rows selected.
```

Inference:

We will gain an exposure to create procedures.

Experiment 31:

Aim:

Create a function which takes deparment_id as parameter and returns the name of the department

Description:

A stored function is a special kind stored program that returns a single value. Typically, you use stored functions to encapsulate common formulas or business rules that are reusable among SQL statements or stored programs.

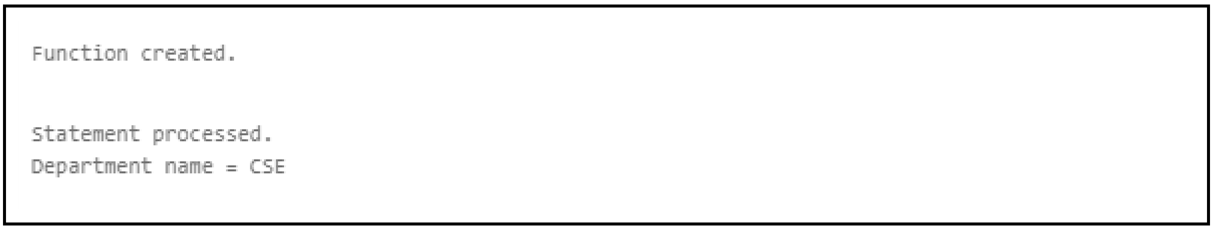
Syntax:

```
CREATE FUNCTION function_name(
   param1,
   param2,…
)
RETURNS datatype
[NOT] DETERMINISTIC
BEGIN
 -- statements
END;
```

Query Command:

```
CREATE OR REPLACE FUNCTION DEP_NAME (DEP_ID IN NUMBER)
RETURN DEPARTMENT.DEPARTMENT_NAME%TYPE
IS
   NAME DEPARTMENT.DEPARTMENT_NAME%TYPE;
BEGIN
   SELECT DEPARTMENT_NAME INTO NAME FROM DEPARTMENT WHERE
DEPARTMENT_ID = DEP_ID;
   RETURN NAME;
END DEP_NAME;
/
DECLARE
   NAME DEPARTMENT.DEPARTMENT_NAME%TYPE;
BEGIN
   NAME := DEP_NAME(1);
   dbms_output.put_line('Department name = ' || NAME);
END;
/
```

Output:



```
Function created.


Statement processed.
Department name = CSE
```

Inference:

We will gain an exposure to create functions.

Experiment 32:

Aim:

Write a package which implements a set of functions that will compute HRA, DA based on rules given

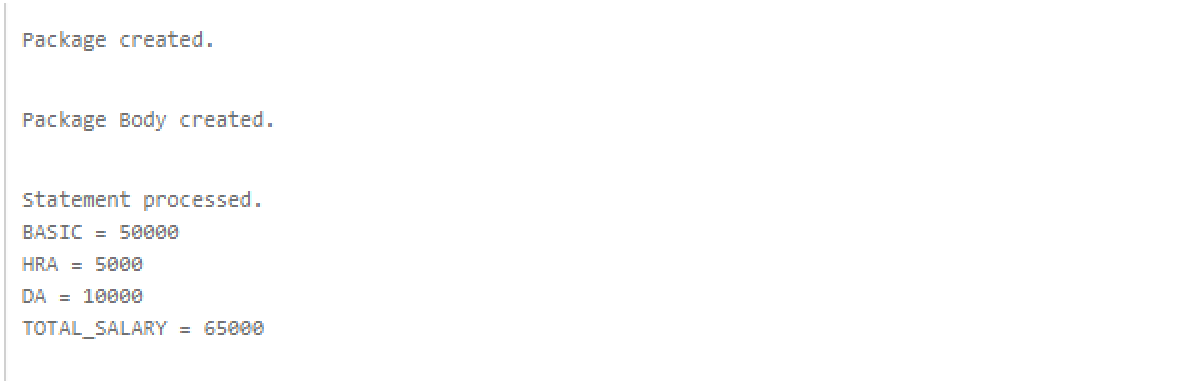
Description:

A package is a schema object that groups logically related PL/SQL types, variables, constants, subprograms, cursors, and exceptions. A package is compiled and stored in the database, where many applications can share its contents.

Query Command:

```
CREATE OR REPLACE PACKAGE TOTAL_SAL
AS
   FUNCTION HRA(BASIC IN INT) RETURN INT;
   FUNCTION DA(BASIC IN INT) RETURN INT;
END;
/
CREATE OR REPLACE PACKAGE BODY TOTAL_SAL
AS
   FUNCTION HRA(BASIC IN INT) RETURN INT
   IS
      H INT;
   BEGIN
      H := BASIC*0.1;
      RETURN H;
   END;
   FUNCTION DA(BASIC IN INT) RETURN INT
   IS
      D INT;
   BEGIN
      D := BASIC * 0.2;
      RETURN D;
   END;
END;
/
DECLARE
   BASIC INT := 50000;
   H INT;
   D INT;
   T INT;
BEGIN
   H := TOTAL_SAL.HRA(BASIC);
   D := TOTAL_SAL.DA(BASIC);
   T := BASIC + H + D;
   DBMS_OUTPUT.PUT_LINE('BASIC = '||BASIC||CHR(10)||'HRA = '|| H || CHR(10)
||'DA = '|| D|| CHR(10) ||'TOTAL_SALARY = '||T);
END;
/
```

Output:



```
Package created.


Package Body created.


Statement processed.
BASIC = 50000
HRA = 5000
DA = 10000
TOTAL_SALARY = 65000
```

Inference:

We will gain an exposure to create packages

WEEK - 12

Experiment 33:

Aim:

Define a cursor which runs through all employees who belong to department with id = 2.

Description:

DECLARE CURSOR defines the attributes of a Transact-SQL server cursor, such as its scrolling behavior and the query used to build the result set on which the cursor operates. The OPEN statement populates the result set, and FETCH returns a row from the result set.

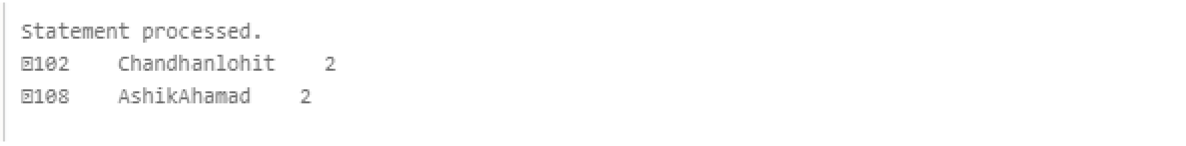
Syntax:

```
Cursor cursor_name [Is/As]
Select Statement;
Declare
Begin
Open cursor_name;
…
…
Close cursor_name;
End;
```

Query Command:

```
DECLARE
CURSOR EMPALL IS SELECT EMPLOYEE_ID, FIRST_NAME, LAST_NAME,
DEPARTMENT_ID FROM EMPLOYEE WHERE DEPARTMENT_ID = 2;
ID EMPLOYEE.EMPLOYEE_ID%TYPE;
F_NAME EMPLOYEE.FIRST_NAME%TYPE;
L_NAME EMPLOYEE.LAST_NAME%TYPE;
DEPT EMPLOYEE.DEPARTMENT_ID%TYPE;
BEGIN
   OPEN EMPALL;
   LOOP
      FETCH EMPALL INTO ID, F_NAME, l_NAME, DEPT;
      EXIT WHEN EMPALL%NOTFOUND;
      dbms_output.put_line(chr(5)|| ID ||'    '|| F_NAME || L_NAME ||'    '|| DEPT);
   END LOOP;
END
```

Output:



```
Statement processed.
⊡102    Chandhanlohit    2
⊡108    AshikAhamad    2
```

Inference:

We will learn about cursors like declaring it, opening and closing of cursor, fetching the rows in a cursor.

Experiment 34:

Aim:

Declare a cursor which runs through all employees who belong to department with id = 2, open the cursor and close the cursor without doing anything.

Description:

DECLARE CURSOR defines the attributes of a Transact-SQL server cursor, such as its scrolling behavior and the query used to build the result set on which the cursor operates. The OPEN statement populates the result set, and FETCH returns a row from the result set.

Syntax:

```
Cursor cursor_name [Is/As]
Select Statement;
Declare
Begin
Open cursor_name;
…
…
Close cursor_name;
End;
```

Query Command:

```
DECLARE
CURSOR EMPALL IS SELECT * FROM EMPLOYEE WHERE DEPARTMENT_ID =
2;
BEGIN
   OPEN EMPALL;
   CLOSE EMPALL;
END;
```

Output:

```
Statement processed.
```

Inference:

We will learn about cursors like declaring it, opening and closing of cursor, fetching the rows in a cursor.

Experiment 35:

Aim:

Declare a cursor which runs through all employees who belong to department with id = 2, open the cursor and fetches one employee at a time, prints the last name and then closes the cursor after all employees are done.

Description:

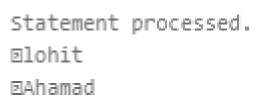
DECLARE CURSOR defines the attributes of a Transact-SQL server cursor, such as its scrolling behavior and the query used to build the result set on which the cursor operates. The OPEN statement populates the result set, and FETCH returns a row from the result set.

```
Syntax:
Cursor cursor_name [Is/As]
Select Statement;
Declare
Begin
Open cursor_name;
LOOP
Fetch … ;
…;
END LOOP;
Close cursor_name;
End;
```

Query Command:

```
DECLARE
CURSOR EMPALL IS SELECT LAST_NAME FROM EMPLOYEE WHERE
DEPARTMENT_ID = 2;
L_NAME EMPLOYEE.LAST_NAME%TYPE;
BEGIN
   OPEN EMPALL;
   LOOP
      FETCH EMPALL INTO l_NAME;
      EXIT WHEN EMPALL%NOTFOUND;
      dbms_output.put_line(chr(5)|| L_NAME);
   END LOOP;
   CLOSE EMPALL;
END;
```

Output:



```
Statement processed.
⊡lohit
⊡Ahamad
```

Inference:

We will learn about cursors like declaring it, opening and closing of cursor, fetching the rows in a cursor.

Experiment 36:

Aim:

Use a cursor to look at each employee who belongs to department with id 10, check the job grade and append NEW_ to all job_grades.

Description:

DECLARE CURSOR defines the attributes of a Transact-SQL server cursor, such as its scrolling behavior and the query used to build the result set on which the cursor operates. The OPEN statement populates the result set, and FETCH returns a row from the result set.

Syntax:

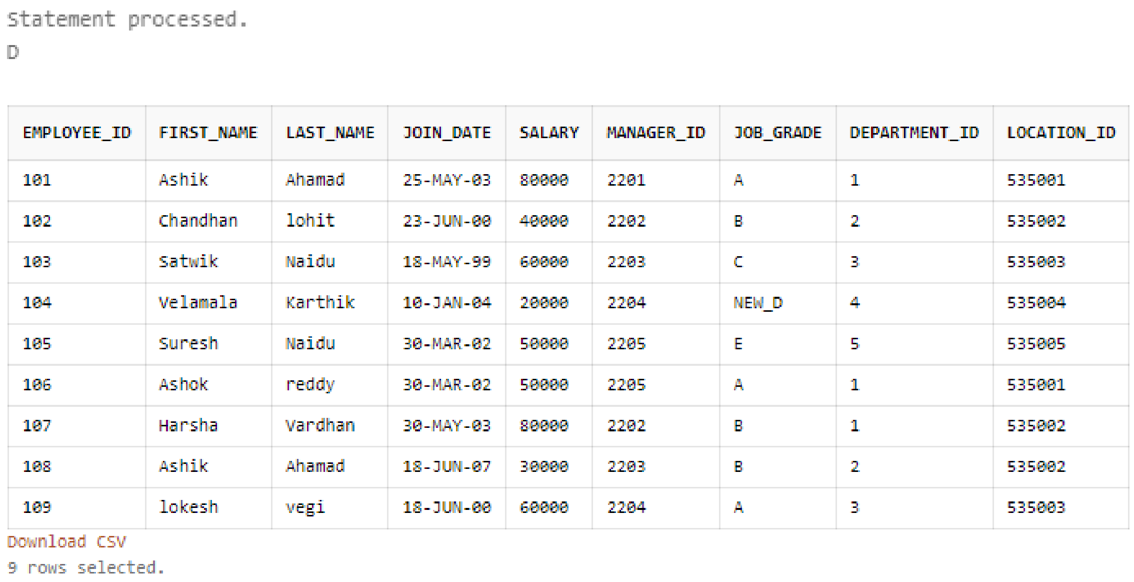
```
Cursor cursor_name [Is/As]
Select Statement;
Declare
Begin
Open cursor_name;
LOOP
Fetch … ;
END LOOP;
Close cursor_name;
End;
```

```
Case
When Condition then
<statements>
…
…
Else
<Statements>
End Case;
```

Query Command:

```
DECLARE
CURSOR EMPALL IS SELECT JOB_GRADE FROM EMPLOYEE WHERE
DEPARTMENT_ID = 4;
JOB_G EMPLOYEE.JOB_GRADE%TYPE;
BEGIN
   OPEN EMPALL;
   LOOP
      FETCH EMPALL INTO JOB_G;
      EXIT WHEN EMPALL%NOTFOUND;
      UPDATE EMPLOYEE SET JOB_GRADE = CONCAT('NEW_',JOB_G) WHERE
DEPARTMENT_ID = 4;
      dbms_output.put_line(JOB_G);
   END LOOP;
   CLOSE EMPALL;
END;
SELECT * FROM EMPLOYEE;
```

Output:



```
Statement processed.
D
```

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	JOIN_DATE	SALARY	MANAGER_ID	JOB_GRADE	DEPARTMENT_ID	LOCATION_ID
101	Ashik	Ahamad	25-MAY-03	80000	2201	A	1	535001
102	Chandhan	lohit	23-JUN-00	40000	2202	B	2	535002
103	Satwik	Naidu	18-MAY-99	60000	2203	C	3	535003
104	Velamala	Karthik	10-JAN-04	20000	2204	NEW_D	4	535004
105	Suresh	Naidu	30-MAR-02	50000	2205	E	5	535005
106	Ashok	reddy	30-MAR-02	50000	2205	A	1	535001
107	Harsha	Vardhan	30-MAY-03	80000	2202	B	1	535002
108	Ashik	Ahamad	18-JUN-07	30000	2203	B	2	535002
109	lokesh	vegi	18-JUN-00	60000	2204	A	3	535003

```
Download CSV
9 rows selected.
```

Inference:

WEEK - 13

Experiment 37:

Aim:

Create a trigger which writes a record called “employees table being changed” with time in a log table whenever anyone attempts to change employees table.

Description:

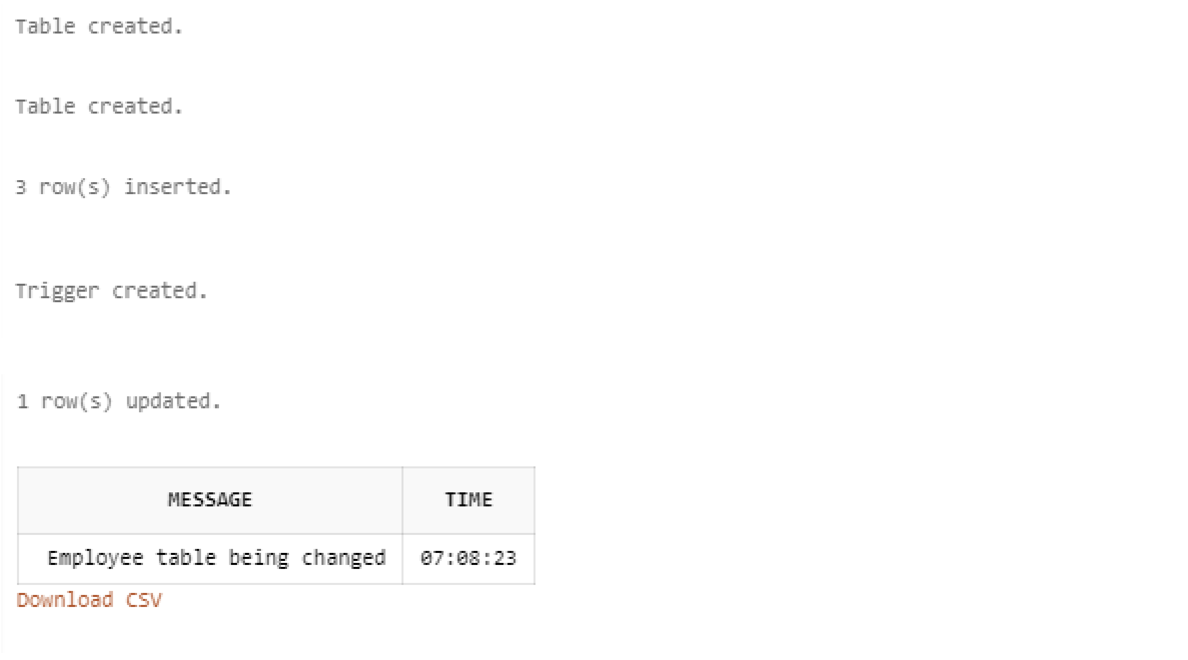
Triggers are stored programs, which are automatically executed or fired when some events occur. Triggers are, in fact, written to be executed in response to any of the following events: A database manipulation (DML) statement (DELETE, INSERT, or UPDATE). A database definition (DDL) statement (CREATE, 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.

Query Command:

```
CREATE TABLE EMPLOYEE (EMP_ID INT PRIMARY KEY, NAME
VARCHAR2(10),SALARY INT);
CREATE TABLE EMPLOYEE_LOG (MESSAGE CLOB, TIME VARCHAR(30));
INSERT ALL
INTO EMPLOYEE VALUES (1, 'ASHIK', 60000)
INTO EMPLOYEE VALUES (2, 'SATWIK', 60000)
INTO EMPLOYEE VALUES (3, 'HARSHA', 60000)
SELECT * FROM DUAL;


CREATE OR REPLACE TRIGGER LOGG
BEFORE INSERT OR UPDATE OR DELETE
ON EMPLOYEE
FOR EACH ROW
DECLARE
   V_MEG CLOB := ' Employee table being changed';
   V_TIME VARCHAR(30) := TO_CHAR(sysdate,'HH24:MI:SS') ;
BEGIN
   IF INSERTING THEN
      INSERT INTO EMPLOYEE_LOG (MESSAGE, TIME) VALUES (V_MEG,
V_TIME);
   ELSIF UPDATING THEN
      INSERT INTO EMPLOYEE_LOG (MESSAGE, TIME) VALUES (V_MEG,
V_TIME);
   ELSIF DELETING THEN
      INSERT INTO EMPLOYEE_LOG (MESSAGE, TIME) VALUES (V_MEG,
V_TIME);
   END IF;
END;
```

```
UPDATE EMPLOYEE SET SALARY = 50000 WHERE NAME LIKE 'SATWIK';
SELECT * FROM EMPLOYEE_LOG;
```

Output:



```
Table created.


Table created.


3 row(s) inserted.



Trigger created.



1 row(s) updated.
```

MESSAGE	TIME
Employee table being changed	07:08:23

Inference:

We have learnt how to Save the logs of a table operations into another table using triggers.

Experiment 38:

Aim:

Create a trigger which writes a record called “employees table has been changed” with time in a log table whenever someone successfully changes the employee table.

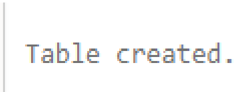
Description:

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.

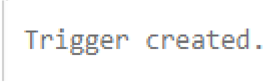
Query Command:

```
CREATE TABLE EMPLOYEE_LOG2 (MESSAGE CLOB, TIME VARCHAR(30));
CREATE OR REPLACE TRIGGER LOGG2
AFTER INSERT OR UPDATE OR DELETE
ON EMPLOYEE
FOR EACH ROW
DECLARE
   V_MEG CLOB :=  'employees table has been changed';
   V_TIME VARCHAR(30) := TO_CHAR(sysdate,'HH24:MI:SS') ;
BEGIN
   IF INSERTING THEN
      INSERT INTO EMPLOYEE_LOG2 (MESSAGE, TIME) VALUES (V_MEG,
V_TIME);
   ELSIF UPDATING THEN
      INSERT INTO EMPLOYEE_LOG2 (MESSAGE, TIME) VALUES (V_MEG,
V_TIME);
   ELSIF DELETING THEN
      INSERT INTO EMPLOYEE_LOG2 (MESSAGE, TIME) VALUES (V_MEG,
V_TIME);
   END IF;
END;

UPDATE EMPLOYEE SET SALARY = 70000 WHERE NAME LIKE 'HARSHA';
SELECT * FROM EMPLOYEE_LOG2;
```

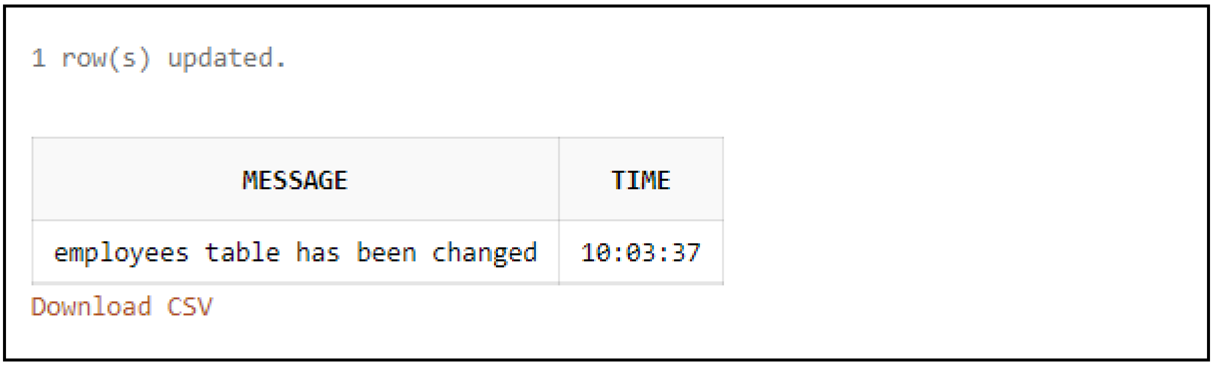
Output:



```
Table created.
```



```
Trigger created.
```



```
1 row(s) updated.
```

MESSAGE	TIME
employees table has been changed	10:03:37

Inference:

We have learnt how to Save the logs of a table operations into another table using triggers for insert, update or delete on a table.

Experiment 39:

Aim:

Implement a solution that would record all attempted updates on salary in the employee table along with the employee id, old salary, new salary and the time when it was attempted in another table

Description:

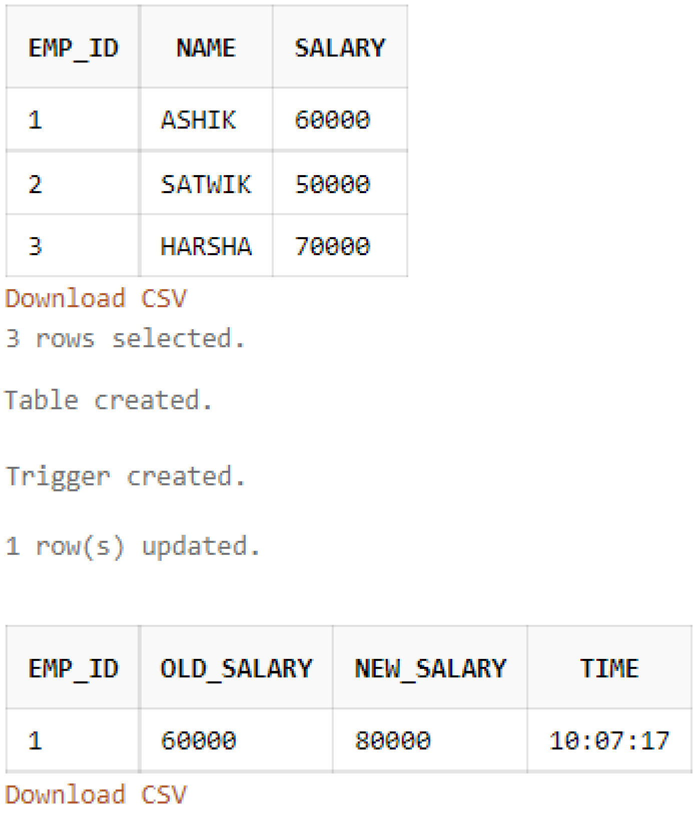
Triggers are stored programs, which are automatically executed or fired when some events occur. Triggers are, in fact, written to be executed in response to any of the following events: A database manipulation (DML) statement (DELETE, INSERT, or UPDATE). A database definition (DDL) statement (CREATE, 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.

Solution for the following will be Using row Before trigger in update operation

Query Command:

```
SELECT * FROM EMPLOYEE;

CREATE TABLE EMPLOYEE_BACKUP (EMP_ID INT PRIMARY KEY,
OLD_SALARY INT, NEW_SALARY INT, TIME VARCHAR(30));

CREATE OR REPLACE TRIGGER EMP_B_U1
BEFORE UPDATE
ON EMPLOYEE
FOR EACH ROW
DECLARE
   V_TIME VARCHAR(30) := TO_CHAR(sysdate,'HH24:MI:SS') ;
BEGIN
   IF UPDATING THEN
      INSERT INTO EMPLOYEE_BACKUP
(EMP_ID,OLD_SALARY,NEW_SALARY,TIME) VALUES (:NEW.EMP_ID,
:OLD.SALARY, :NEW.SALARY, V_TIME);
   END IF;
END;

UPDATE EMPLOYEE SET SALARY = 80000 WHERE NAME LIKE 'ASHIK';
SELECT * FROM EMPLOYEE_BACKUP;
```

Output:



EMP_ID	NAME	SALARY
1	ASHIK	60000
2	SATWIK	50000
3	HARSHA	70000

Download CSV

```
3 rows selected.

Table created.

Trigger created.

1 row(s) updated.
```

EMP_ID	OLD_SALARY	NEW_SALARY	TIME
1	60000	80000	10:07:17

Download CSV

Inference:

We have learned how to use row before trigger in update operation.

Experiment 40:

Aim:

Implement a solution that would record all updates on salary in the employee table along with the employee id, old salary, new salary and the time when it was updated in another table.

Description:

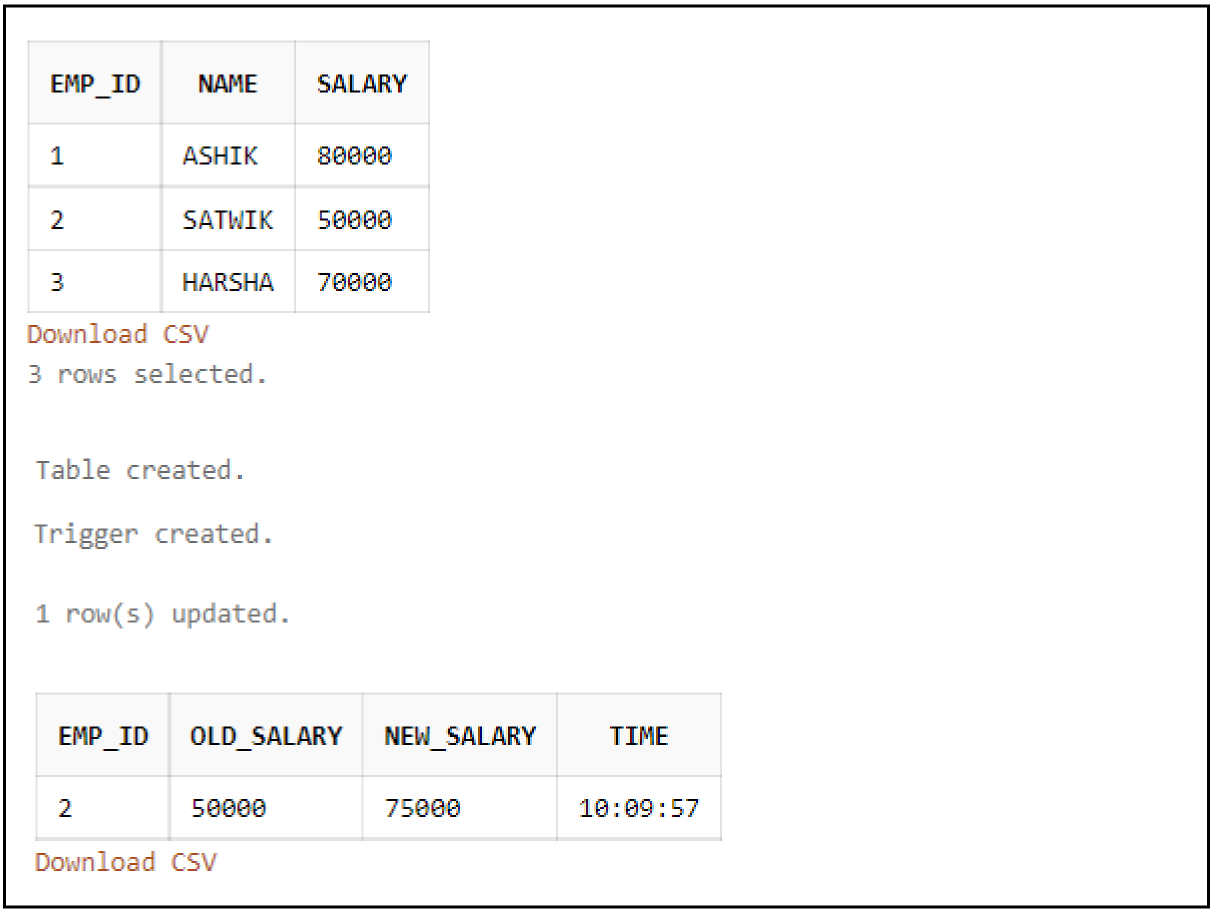
Triggers are stored programs, which are automatically executed or fired when some events occur. Triggers are, in fact, written to be executed in response to any of the following events: A database manipulation (DML) statement (DELETE, INSERT, or UPDATE). A database definition (DDL) statement (CREATE, 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.

Solution for the following will be Using row After trigger in update operation

Query Command:

```
SELECT * FROM EMPLOYEE;

CREATE TABLE EMPLOYEE_BACKUP2 (EMP_ID INT PRIMARY KEY,
OLD_SALARY INT, NEW_SALARY INT, TIME VARCHAR(30));

CREATE OR REPLACE TRIGGER EMP_B_U2
AFTER UPDATE
ON EMPLOYEE
FOR EACH ROW
DECLARE
   V_TIME VARCHAR(30) := TO_CHAR(sysdate,'HH24:MI:SS') ;
BEGIN
   IF UPDATING THEN
      INSERT INTO EMPLOYEE_BACKUP2
(EMP_ID,OLD_SALARY,NEW_SALARY,TIME) VALUES (:NEW.EMP_ID,
:OLD.SALARY, :NEW.SALARY, V_TIME);
   END IF;
END;

UPDATE EMPLOYEE SET SALARY =  75000 WHERE EMP_ID = 2;
SELECT * FROM EMPLOYEE_BACKUP2;
```

Output:



EMP_ID	NAME	SALARY
1	ASHIK	80000
2	SATWIK	50000
3	HARSHA	70000

Download CSV

```
3 rows selected.


Table created.

Trigger created.

1 row(s) updated.
```

EMP_ID	OLD_SALARY	NEW_SALARY	TIME
2	50000	75000	10:09:57

Download CSV

Inference:

We have learnt how to use row after trigger in update operation.

Experiment 41:

Aim:

Implement a solution to insert a log entry in log table whenever salary of employees with more than 20000 is revised.

Description:

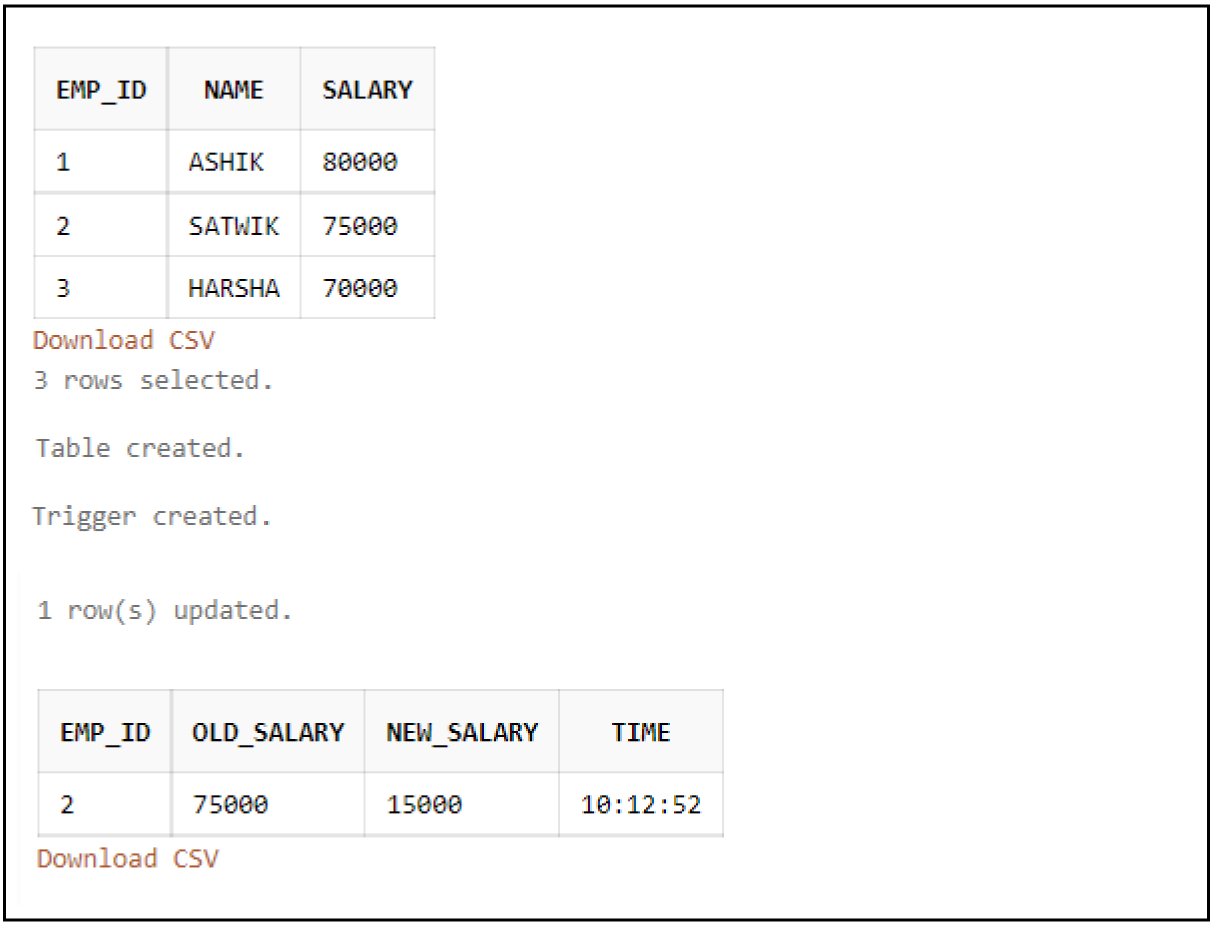
Triggers are stored programs, which are automatically executed or fired when some events occur. Triggers are, in fact, written to be executed in response to any of the following events: A database manipulation (DML) statement (DELETE, INSERT, or UPDATE). A database definition (DDL) statement (CREATE, 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.

Writing a row after trigger on employee record and The trigger should work on the condition that the salary of the record is greater than 20000.

Query Command:

```
SELECT * FROM EMPLOYEE;

CREATE TABLE EMPLOYEE_BACKUP3 (EMP_ID INT PRIMARY KEY,
OLD_SALARY INT, NEW_SALARY INT, TIME VARCHAR(30));

CREATE OR REPLACE TRIGGER EMP_B_U3
AFTER UPDATE
ON EMPLOYEE
FOR EACH ROW
WHEN (OLD.SALARY > 20000)
DECLARE
   V_TIME VARCHAR(30) := TO_CHAR(sysdate,'HH24:MI:SS') ;
BEGIN
   IF UPDATING THEN
      INSERT INTO EMPLOYEE_BACKUP3
(EMP_ID,OLD_SALARY,NEW_SALARY,TIME) VALUES (:NEW.EMP_ID,
:OLD.SALARY, :NEW.SALARY, V_TIME);
   END IF;
END;

UPDATE EMPLOYEE SET SALARY =  15000 WHERE EMP_ID = 2;
SELECT * FROM EMPLOYEE_BACKUP3;
select * from employee;
UPDATE EMPLOYEE SET SALARY =  60000 WHERE EMP_ID = 2;
select * from employee;
SELECT * FROM EMPLOYEE_BACKUP3;
```

Output:

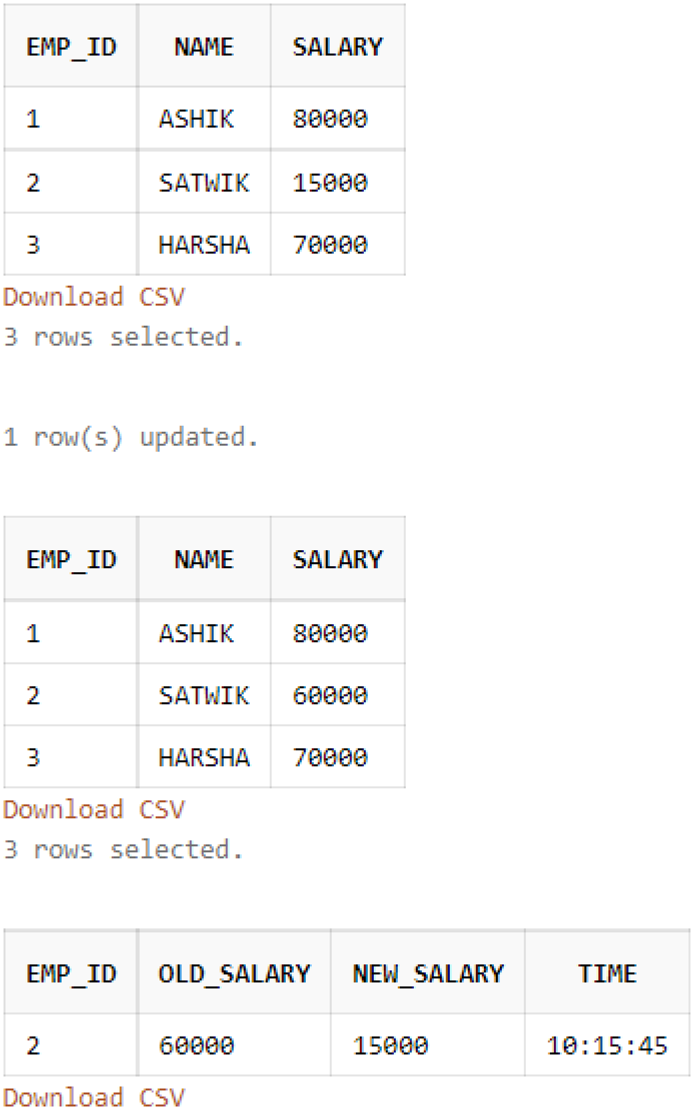


EMP_ID	NAME	SALARY
1	ASHIK	80000
2	SATWIK	75000
3	HARSHA	70000

```
Download CSV
3 rows selected.

Table created.

Trigger created.


1 row(s) updated.
```

EMP_ID	OLD_SALARY	NEW_SALARY	TIME
2	75000	15000	10:12:52

```
Download CSV
```



EMP_ID	NAME	SALARY
1	ASHIK	80000
2	SATWIK	15000
3	HARSHA	70000

Download CSV

3 rows selected.

1 row(s) updated.

EMP_ID	NAME	SALARY
1	ASHIK	80000
2	SATWIK	60000
3	HARSHA	70000

Download CSV

3 rows selected.

EMP_ID	OLD_SALARY	NEW_SALARY	TIME
2	60000	15000	10:15:45

Download CSV

Inference:

We have learnt how to use after insert or update or delete triggers with a condition.

Experiment 42:

Aim:

Implement a solution which would transparently make the user feel like he is updating a view but in reality the view is read-only and a trigger is updating the base tables based on the view update given

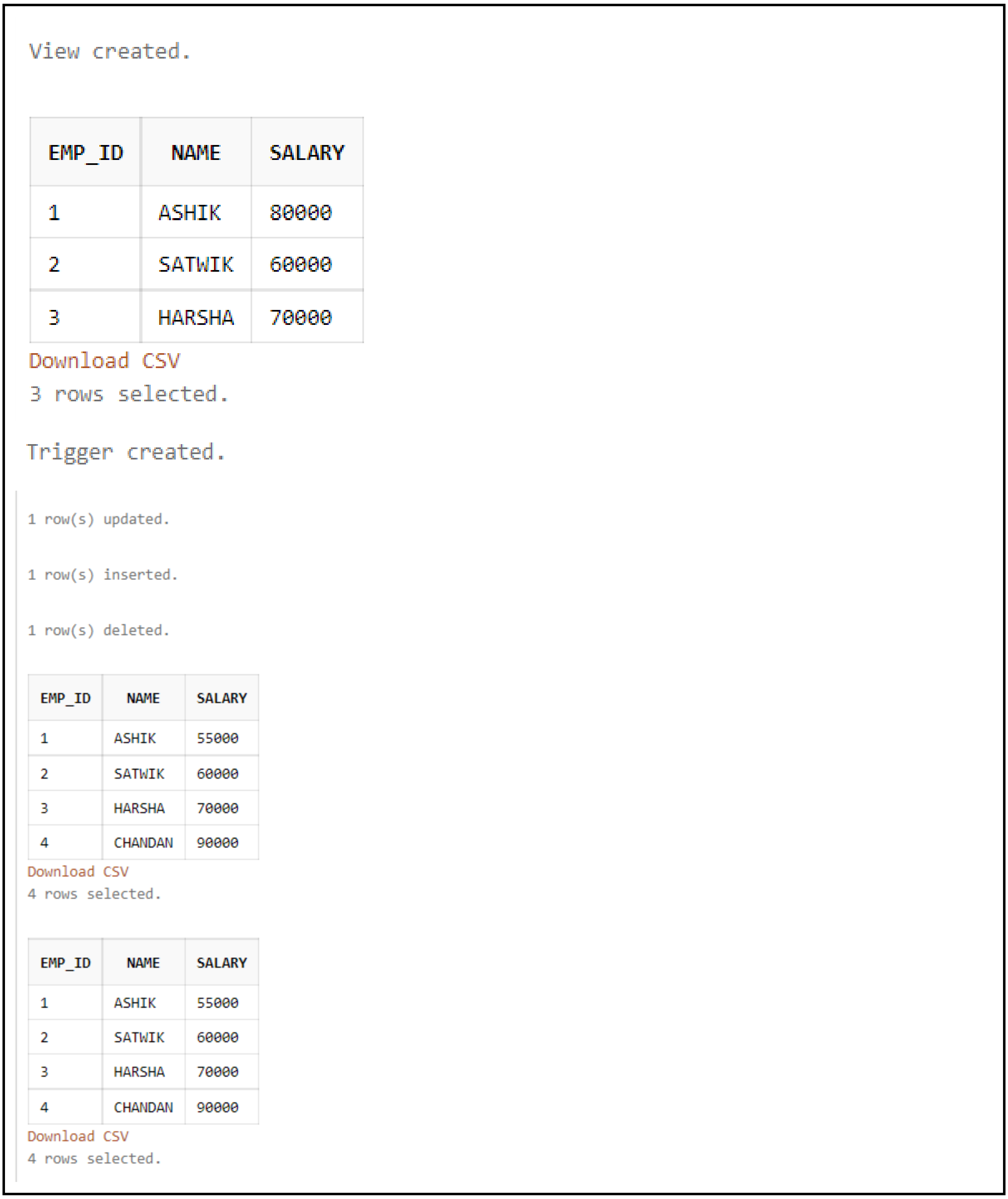
Description:

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.
And here Use the instead of option in triggers to make trigger work in place of view and reproduce the effect intended.

Query Command:

```
CREATE VIEW EMP_VIEW AS SELECT * FROM EMPLOYEE;
SELECT * FROM EMP_VIEW;

CREATE OR REPLACE TRIGGER VIEW_UPDATE
INSTEAD OF UPDATE OR DELETE OR INSERT ON EMP_VIEW
FOR EACH ROW
BEGIN
   IF UPDATING THEN
   UPDATE EMPLOYEE SET SALARY = :NEW.SALARY WHERE EMP_ID =
:NEW.EMP_ID;
   ELSIF DELETING THEN
   DELETE FROM EMPLOYEE WHERE EMP_ID = :NEW.EMP_ID;
   ELSIF INSERTING THEN
   INSERT INTO EMPLOYEE VALUES (:NEW.EMP_ID, :NEW.NAME,
:NEW.SALARY);
   END IF;
END;
UPDATE EMP_VIEW SET SALARY = 55000 WHERE EMP_ID = 1;
INSERT INTO EMP_VIEW VALUES (4, 'CHANDAN',90000);
DELETE FROM EMP_VIEW WHERE EMP_ID = 4;
SELECT * FROM EMP_VIEW;
SELECT * FROM EMPLOYEE;
```

Output:



View created.

EMP_ID	NAME	SALARY
1	ASHIK	80000
2	SATWIK	60000
3	HARSHA	70000

Download CSV

3 rows selected.

Trigger created.

1 row(s) updated.

1 row(s) inserted.

1 row(s) deleted.

EMP_ID	NAME	SALARY
1	ASHIK	55000
2	SATWIK	60000
3	HARSHA	70000
4	CHANDAN	90000

Download CSV

4 rows selected.

EMP_ID	NAME	SALARY
1	ASHIK	55000
2	SATWIK	60000
3	HARSHA	70000
4	CHANDAN	90000

Download CSV

4 rows selected.

Inference:

We have learned how to use triggers on views for updating, inserting and deleting.