Database Management Systems Lab

LIST OF EXPERIMENTS

1. Concept design with E-R Model

2. Relational Model

3. Normalization

4. Practicing DDL commands

5. Practicing DML commands

6. Querying (using ANY, ALL, IN, Exists, NOT EXISTS, UNION, INTERSECT, Constraints etc.)

7. Queries using Aggregate functions, GROUP BY, HAVING and Creation and dropping of Views.

8. Triggers (Creation of insert trigger, delete trigger, update trigger)

9. Procedures

10. Usage of Cursors

EXPERMENT 1: E-R model

Analyze the problem and come with the entities in it. Identify what Data has to be persisted in the databases. The Following are the entities and its attributes.

a. Bus:

1. Bus_No: varchar (10) (primary key)
2. Source: varchar (20)
3. Destination: varchar (20)

b. Passenger:

1. PNR_No: Number (9) (primary key)
2. Ticket_No: Number (9)
3. Name: varchar (15)
4. Age: integer (4)
5. Sex: char (10); Male/Female
6. P_PNO: varchar (15)

c. Reservation:

1. PNR_No : number(9) (foreign key)
2. Journey date : date
3. No_of_seats : integer(8)
4. Address : varchar(50)
5. Contact_No : Number(9)
6. Status : Char(2)

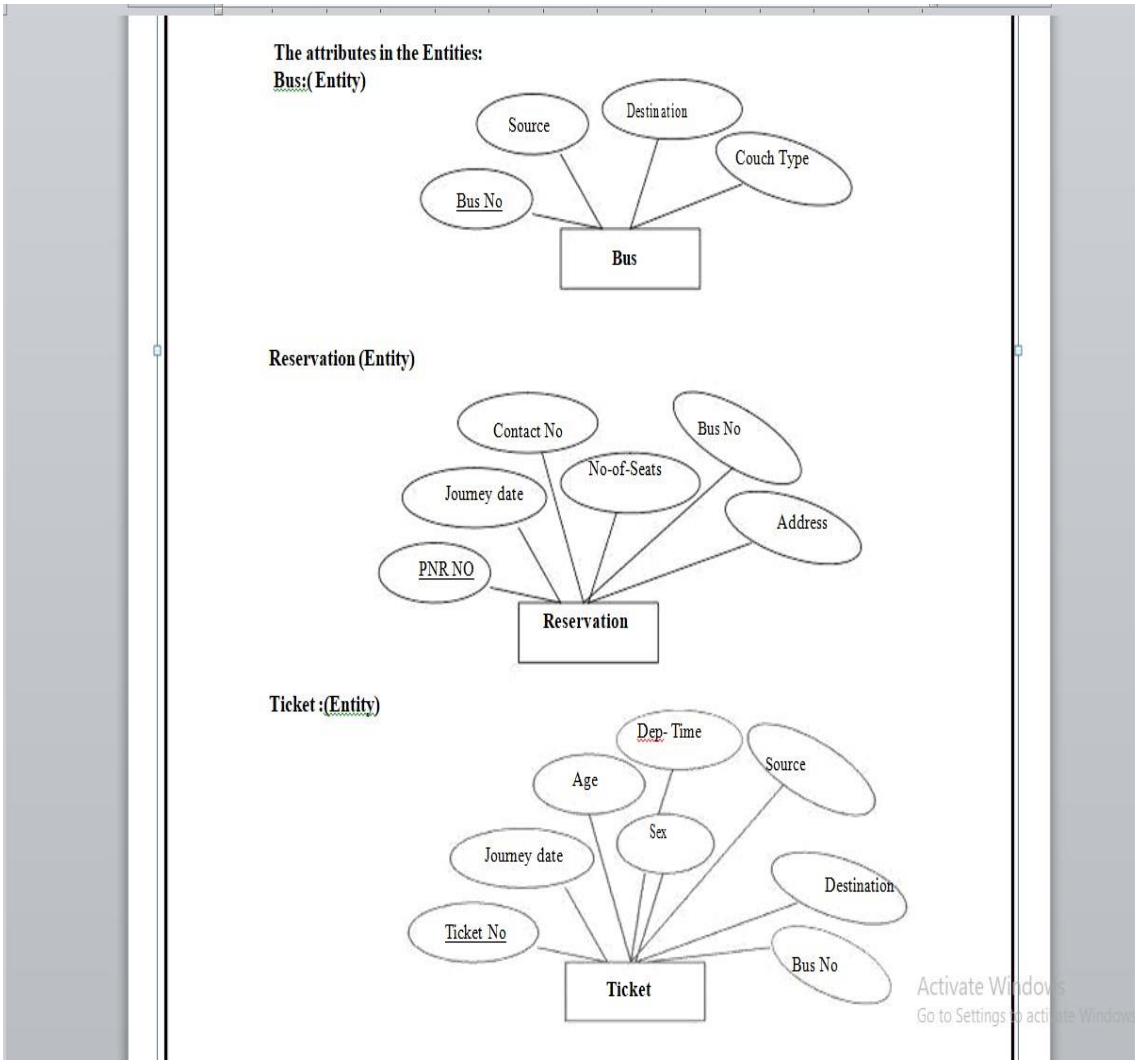
d. Cancellation:

1. PNR_No : number(9)(foreign key)
2. Journey_date : date
3. No_of_seats : integer(8)
4. Address : varchar(50)
5. Contact_No : Number(10)
6. Status : Char(2)

e. Ticket:

1. Ticket_No : number(9)(primary key)
2. Journey_date : date
3. Age : int(4)
4. Sex : Char(10)
5. Source : varchar(50)
6. Destination :varchar(50)
7. Dep_time : varchar(50)

The attributes in the Entities:

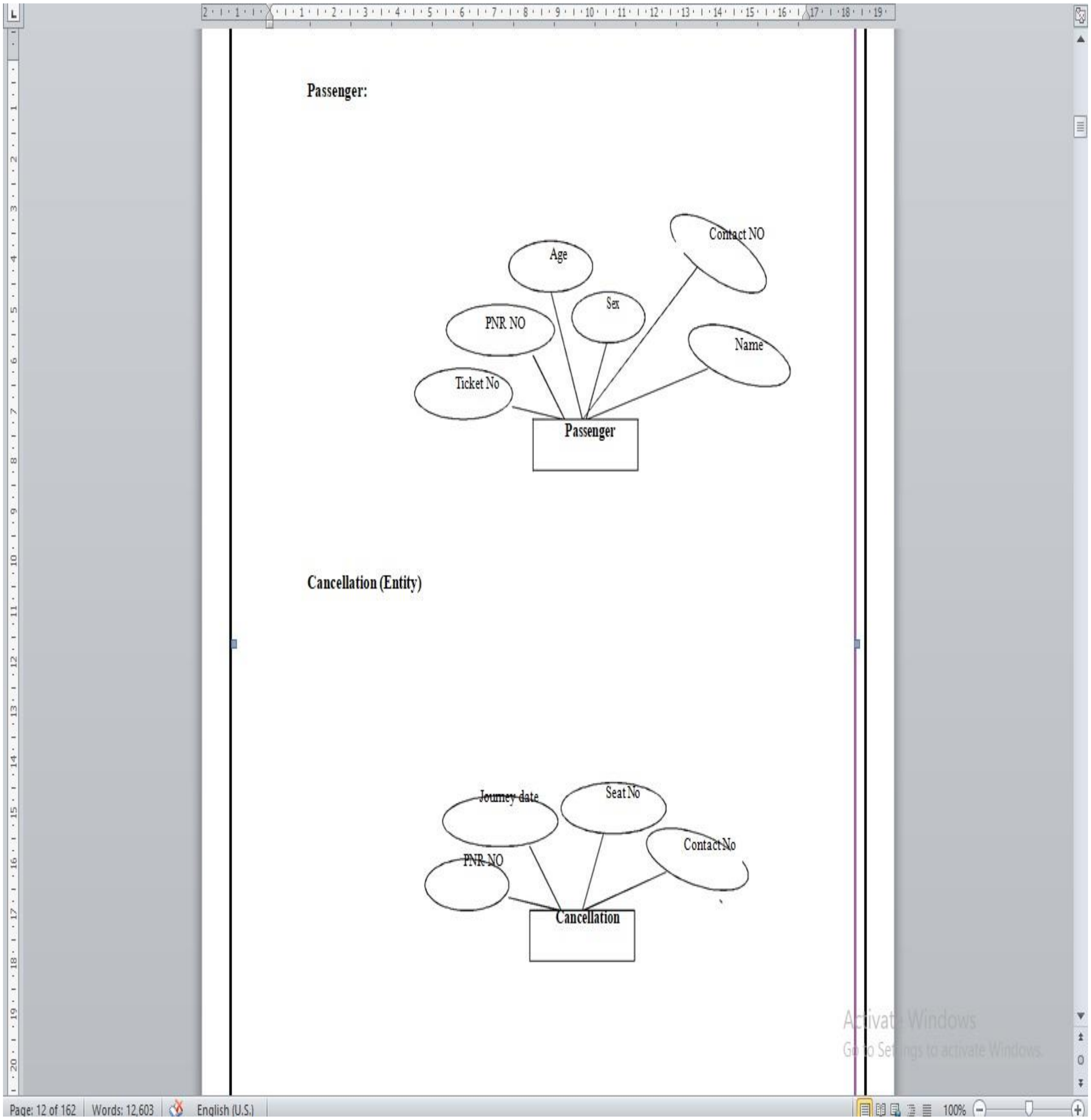


The attributes in the Entities:

Bus:(Entity)

Reservation (Entity)

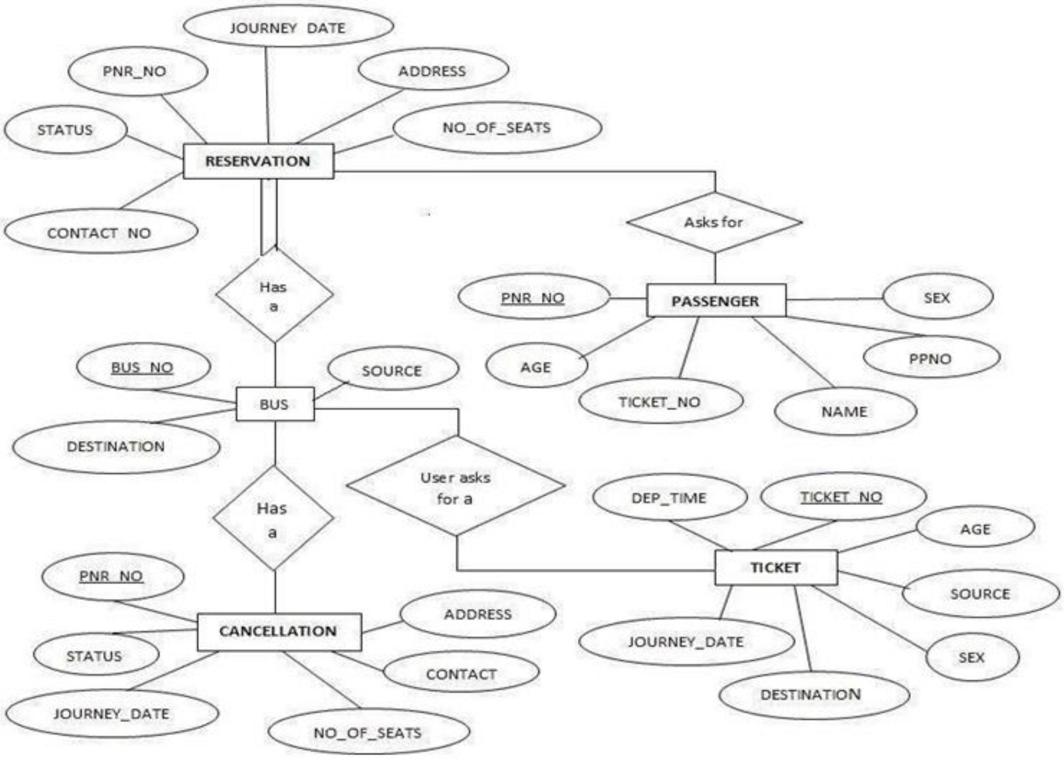
Ticket :(Entity)



Passenger:

Cancellation (Entity)

Relate the entities appropriately. Apply cardinalities for each relation



EXPERIMENT 2: Relational Model

Represent all entities in a tabular fashion. Represent all relationships in a tabular fashion. The fallowing is tabular representation of the above entities and relationships

BUS:

Bus_no	**Source**	**Destination**
TA07AZ6789	Hyderabad	Goa

COLOUMN NAME	**DATA TYPE**	**CONSTRAINT**
Bus No	varchar2(10)	**Primary Key**
Source	varchar2(20)	
Destination	varchar2(20)	

PASSENGER:

Pnr_No	**Ticket_no**	**Name**	**Age**	**Sex**	**P_PNO**
7456558	TS1234568	Raj	35	Male	9898989000

COLOUMN NAME	**DATA TYPE**	**CONSTRAINT**
PNR No	Number(9)	**Primary Key**
Ticket No	Number(9)	Foreign key
Name	varchar2(15)	
Age	integer(4)	
Sex	char(10)	(Male/Female)

P_PNo	Number(9)	Should be equal to 10 numbers and not allow other than numeric

RESERVATION:

Pnr_No	**Journey_date**	**No_of_seats**	**Address**	**Contact_No**	**Status**
5458661	14-06-2023	2	LB Nagar	9845676659	CNF

COLOUMN NAME	**DATA TYPE**	**CONSTRAINT**
PNRNo	number(9)	**Primary Key**
Journey date	Date	
No-of-seats	integer(8)	
Address	varchar2(50)	
Contact No	Number(9)	Should be equal to 10 numbers and not allow other than numeric
BusNo	varchar2(10)	**Foreign key**
Seat no	Number(10)	

CANCELLATION:

Pnr_No	Journey_date	No_of_seats	Address	Contact_No	Status
5458661	14-06-2023	2	LB Nagar	9845676659	CNF

COLOUMN NAME	**DATA TYPE**	**CONSTRAINT**
PNR No	Number(9)	Foriegn-key
Journey-date	Date	
Address	Varchar2(100)	
Seat no	Number(9)	
Contact_No	Number(9)	Should be equal to 10 numbers and not allow other than numeric

TICKET:

Ticket_ No	Journey_date	Age	sex	source	Destination	Dep_time

COLOUMN NAME	**DATA TYPE**	**CONSTRAINT**
Ticket_No	number(9)	**Primary Key**
Journey date	Date	
Age	Number(4)	
Sex	Varchar2(10)	
Source	varchar2(10)	
Destination	varchar2(10)	
Dep-time	varchar2(10)	
Bus No	Number2(10)	

EXPERIMENT 3 . **Normalization**

Normalization is the process of minimizing **redundancy** from a relation or set of relations. Redundancy in relation may cause insertion, deletion, and update anomalies. So, it helps to minimize the redundancy in relations. **Normal forms** are used to eliminate or reduce redundancy in database tables.

Introduction:

In database management systems (DBMS), normal forms are a series of guidelines that help to ensure that the design of a database is efficient, organized, and free from data anomalies. There are several levels of normalization, each with its own set of guidelines, known as normal forms.

Example

We'll be using a **student database** as an example in this article, which records student, class, and teacher information.

Let's say our student database looks like this:

Student ID	Student Name	Fees Paid	Course Name	Class 1	Class 2	Class 3
1	John Smith	200	Economics	Economics 1	Biology 1	
2	Maria Griffin	500	Computer Science	Biology 1	Business Intro	Programming 2
3	Susan Johnson	400	Medicine	Biology 2		
4	Matt Long	850	Dentistry			

This table keeps track of a few pieces of information:

- The student names
- The fees a student has paid
- The classes a student is taking, if any

This is **not** a normalised table, and there are a few issues with this.

Insert Anomaly

An insert anomaly happens when we try to insert a record into this table without knowing all the data we need to know. For example, if we wanted to add a new student but did not know their course name.

The new record would look like this:

Student ID	**Student Name**	**Fees Paid**	**Course Name**	**Class 1**	**Class 2**	**Class 3**
1	John Smith	200	Economics	Economics 1	Biology 1	
2	Maria Griffin	500	Computer Science	Biology 1	Business Intro	Programming 2
3	Susan Johnson	400	Medicine	Biology 2		
4	Matt Long	850	Dentistry			
5	**Jared Oldham**	**0**	**?**			

We would be adding incomplete data to our table, which can cause issues when trying to analyze this data.

Update Anomaly

An update anomaly happens when we want to update data, and we update some of the data but not other data. For example, let’s say the class Biology 1 was changed to “Intro to Biology”. We would have to query all of the columns that could have this Class field and rename each one that was found.

Student ID	**Student Name**	**Fees Paid**	**Course Name**	**Class 1**	**Class 2**	**Class 3**
1	John Smith	200	Economics	Economics 1	**Intro to Biology**	
2	Maria Griffin	500	Computer Science	**Intro to Biology**	Business Intro	Programming 2
3	Susan Johnson	400	Medicine	Biology 2		
4	Matt Long	850	Dentistry			

There’s a risk that we miss out on a value, which would cause issues.

Ideally, we would only update the value once, in one location.

Delete Anomaly

A delete anomaly occurs when we want to delete data from the table, but we end up deleting more than what we intended.

For example, let’s say Susan Johnson quits and her record needs to be deleted from the system. We could delete her row:

Student ID	Student Name	Fees Paid	Course Name	Class 1	Class 2	Class 3
1	John Smith	200	Economics	Economics 1	Biology 1	
2	Maria Griffin	500	Computer Science	Biology 1	Business Intro	Programming 2
3	**Susan Johnson**	**400**	**Medicine**	**Biology 2**		
4	Matt Long	850	Dentistry			

But, if we delete this row, we lose the record of the Biology 2 class, because it’s not stored anywhere else. The same can be said for the Medicine course.

We should be able to delete one type of data or one record without having impacts on other records we don’t want to delete.

Experiment 4: Practicing DDL commands

- To practice sql commands in computer use the following applications.
- Download the Oracle Application Server 10g or higher releases and install on computer.
- Download the Install SQL Server 2014 or higher releases and install on computer.
- Please make note that create user name and password where it is applicable.

CREATE It is used to create a new table in the database.

a) Passenger Table

```
Create table passenger (PNR_NO int(9) primary key , Ticket_NO int(9), Name
varchar(20), Age int(4), Sex char(10), PPNO varchar(15));

Desc Passenger;
```

b) Reservation Table

```
Create table reservation (PNR_NO int(9), No_of_seats int(8), Address
varchar(50), Contact_No int(9), Status char(3));

Desc Reservation;
```

c) Bus Table

```
Create table Bus (Bus_No varchar (5) primary key, source varchar (20), destination
varchar (20));

Desc Bus;
```

d) Cancellation Table

```
Create table cancellation (PNR_NO int(9), No_of_seats int(8), Address
varchar(50), Contact_No int(9), Status char(3));

Desc Cancellation;
```

e) Ticket Table

```
Create table ticket (Ticket_No int(9) primary key, age int(4), sex char(4) Not null, source
varchar(20),destination varchar(20), dep_time varchar(4));
```

Desc Ticket;

ALTER: It is used to alter the structure of the database. This change could be either to modify the characteristics of an existing attribute or probably to add a new attribute.

Alter table Bus ADD (Bus_Model varchar2(20));

Desc Bus;

TRUNCATE: It is used to delete all the rows from the table and free the space containing the table.

Truncate table Bus;

Desc bus;

DROP: It is used to delete both the structure and record stored in the table.
Drop table Bus;

Experiment 5: Practicing DML commands

DML commands are used to modify the database. It is responsible for all form of changes in the database.

INSERT: The INSERT command is used to insert data into the row of a table.

Note: in previous experiment we created table structure now in this experiment we can insert data into those tables.

PASSENGER:

```
INSERT INTO passenger (PNR_NO, Ticket_NO, Name, Age, Sex, PPNO) VALUES (1, 101, 'name_1', 13, 'm', 'pp01');
```

Result: 1row affected

```
INSERT INTO passenger (PNR_NO, Ticket_NO, Name, Age, Sex, PPNO ) VALUES (2, 102, 'name_2', 14, 'f', 'pp02');
```

Result: 1row affected

```
INSERT INTO passenger (PNR_NO, Ticket_NO, Name, Age, Sex, PPNO ) VALUES (3, 103, 'name_3', 15, 'm', 'pp03');
```

Result: 1row affected

```
INSERT INTO passenger (PNR_NO, Ticket_NO, Name, Age, Sex, PPNO ) VALUES (4, 104, 'name_4', 16, 'f', 'pp04');
```

Result: 1row affected

```
INSERT INTO passenger (PNR_NO, Ticket_NO, Name, Age, Sex, PPNO ) VALUES (5, 105, 'name_5', 17, 'm', 'pp05');
```

Result: 1row affected

```
INSERT INTO passenger (PNR_NO, Ticket_NO, Name, Age, Sex, PPNO ) VALUES (6, 106, 'name_6', 18, 'f', 'pp06');
```

Result: 1row affected

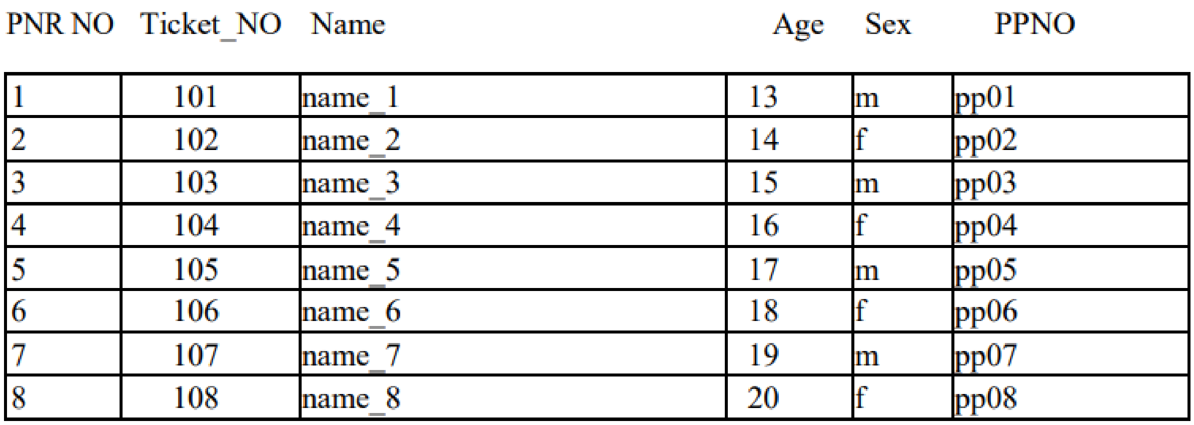
```
INSERT INTO passenger (PNR_NO, Ticket_NO, Name, Age, Sex, PPNO ) VALUES (7, 107, 'name_7', 19, 'm', 'pp07');
```

Result: 1row affected

```
INSERT INTO passenger (PNR_NO, Ticket_NO, Name, Age, Sex, PPNO ) VALUES (8, 108, 'name_8', 20, 'f', 'pp08');
```

Result: 1row affected

Select * from passenger;



PNR NO	Ticket_NO	Name	Age	Sex	PPNO
1	101	name_1	13	m	pp01
2	102	name_2	14	f	pp02
3	103	name_3	15	m	pp03
4	104	name_4	16	f	pp04
5	105	name_5	17	m	pp05
6	106	name_6	18	f	pp06
7	107	name_7	19	m	pp07
8	108	name_8	20	f	pp08

TICKET:

INSERT INTO ticket (Ticket_No, age, sex, source, destination, dep_time) VALUES (101, 13, 'm', 'src1', 'des1', '0830');
Result: 1row affected

INSERT INTO ticket (Ticket_No, age, sex, source, destination, dep_time) VALUES (102, 14, 'f', 'src2', 'des2', '1030');
Result: 1row affected

INSERT INTO ticket (Ticket_No, age, sex, source, destination, dep_time) VALUES (103, 15, 'm', 'src3', 'des3', '1230');
Result: 1row affected

INSERT INTO ticket (Ticket_No, age, sex, source, destination, dep_time) VALUES (104, 16, 'f', 'src4', 'des4', '1430');
Result: 1row affected

INSERT INTO ticket (Ticket_No, age, sex, source, destination, dep_time) VALUES (105, 17, 'm', 'src5', 'des5', '1630');
Result: 1row affected

INSERT INTO ticket (Ticket_No, age, sex, source, destination, dep_time) VALUES (106, 18, 'f', 'src6', 'des6', '1830');
Result: 1row affected

INSERT INTO ticket (Ticket_No, age, sex, source, destination, dep_time) VALUES (107, 19, 'm', 'src7', 'des7', '2030');
Result: 1row affected

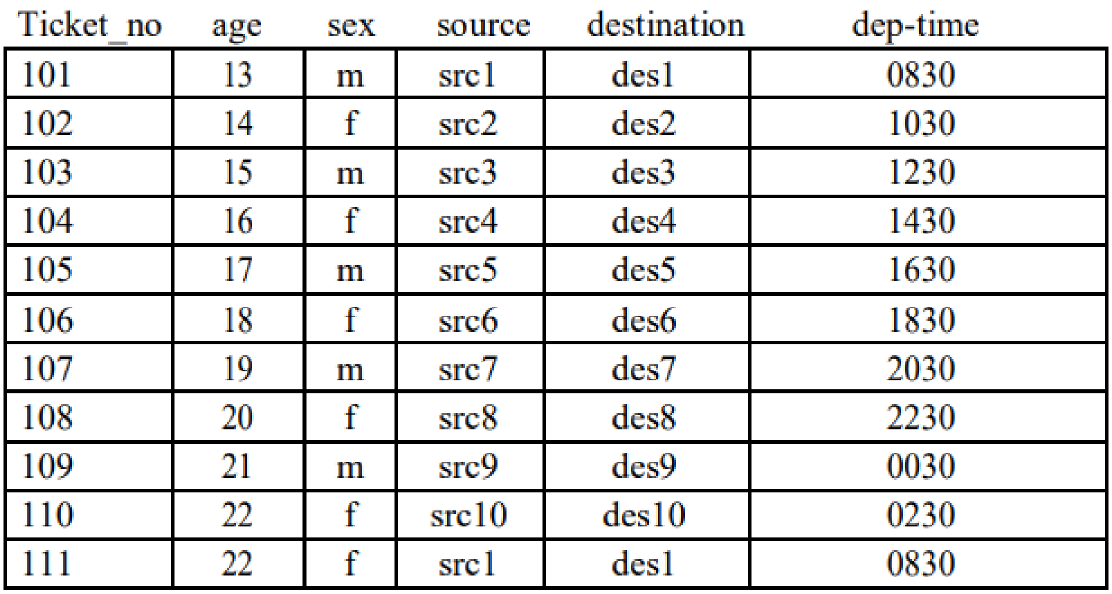
INSERT INTO ticket (Ticket_No, age, sex, source, destination, dep_time) VALUES (108, 20, 'f', 'src8', 'des8', '2230');
Result: 1row affected

INSERT INTO ticket (Ticket_No, age, sex, source, destination, dep_time) VALUES (109, 21, 'm', 'src9', 'des9', '0030');
Result: 1row affected

INSERT INTO ticket (Ticket_No, age, sex, source, destination, dep_time) VALUES (110, 22, 'f', 'src10', 'des10', '0230');
Result: 1row affected

INSERT INTO ticket (Ticket_No, age, sex, source, destination, dep_time) VALUES (111, 22, 'f', 'src1', 'des1', '0830');
Result: 1row affected

Select * from ticket;



Ticket_no	age	sex	source	destination	dep-time
101	13	m	src1	des1	0830
102	14	f	src2	des2	1030
103	15	m	src3	des3	1230
104	16	f	src4	des4	1430
105	17	m	src5	des5	1630
106	18	f	src6	des6	1830
107	19	m	src7	des7	2030
108	20	f	src8	des8	2230
109	21	m	src9	des9	0030
110	22	f	src10	des10	0230
111	22	f	src1	des1	0830

RESERVATION:

INSERT INTO reservation (PNR_NO, No_of_seats, Address, Contact_No, status) VALUES (1, 1, 'adrs_1', 9891, 's');
Result: 1row affected

INSERT INTO reservation (PNR_NO, No_of_seats, Address, Contact_No, status) VALUES (2, 1, 'adrs_2', 9892, 's');
Result: 1row affected

INSERT INTO reservation (PNR_NO, No_of_seats, Address, Contact_No, status) VALUES (3, 1, 'adrs_3', 9893, 's');
Result: 1row affected

INSERT INTO reservation (PNR_NO, No_of_seats, Address, Contact_No, status) VALUES (4, 1, 'adrs_4', 9894, 's');
Result: 1row affected

INSERT INTO reservation (PNR_NO, No_of_seats, Address, Contact_No, status) VALUES (5, 1, 'adrs_5', 9895, 's');
Result: 1row affected

INSERT INTO reservation (PNR_NO, No_of_seats, Address, Contact_No, status) VALUES (6, 1, 'adrs_6', 9896, 's');
Result: 1row affected

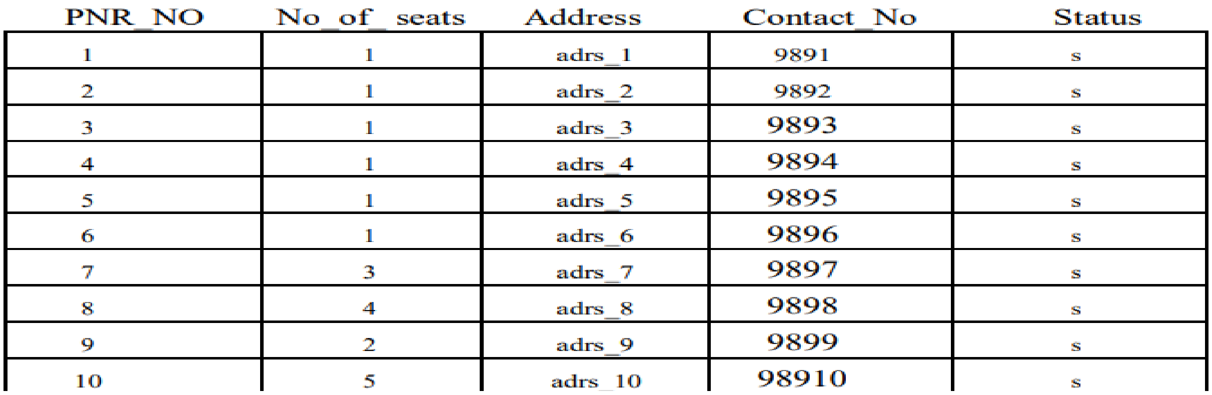
INSERT INTO reservation (PNR_NO, No_of_seats, Address, Contact_No, status) VALUES (7, 3, 'adrs_7', 9897, 's');
Result: 1row affected

INSERT INTO reservation (PNR_NO, No_of_seats, Address, Contact_No, status) VALUES (8, 4, 'adrs_8', 9898, 's');
Result: 1row affected

INSERT INTO reservation (PNR_NO, No_of_seats, Address, Contact_No, status) VALUES (9, 2, 'adrs_9', 9899, 's');
Result: 1row affected

INSERT INTO reservation (PNR_NO, No_of_seats, Address, Contact_No, status) VALUES (10, 5, 'adrs_10', 98910, 's');
Result: 1row affected

Select * from reservation;



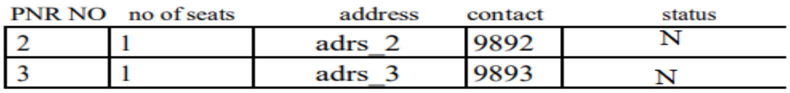
PNR_NO	No_of_ seats	Address	Contact_No	Status
1	1	adrs_1	9891	s
2	1	adrs_2	9892	s
3	1	adrs_3	9893	s
4	1	adrs_4	9894	s
5	1	adrs_5	9895	s
6	1	adrs_6	9896	s
7	3	adrs_7	9897	s
8	4	adrs_8	9898	s
9	2	adrs_9	9899	s
10	5	adrs_10	98910	s

CANCELLATION:

INSERT INTO cancellation (PNR_NO, No_of_seats, Address, Contact_No, Status) VALUES (2, 1, 'adrs_2', 9892, 'N');
Result: 1row affected

INSERT INTO cancellation (PNR_NO, No_of_seats, Address, Contact_No, Status) VALUES (3, 1, 'adrs_3', 9893, 'N');
Result: 1row affected

Select * from cancellation;



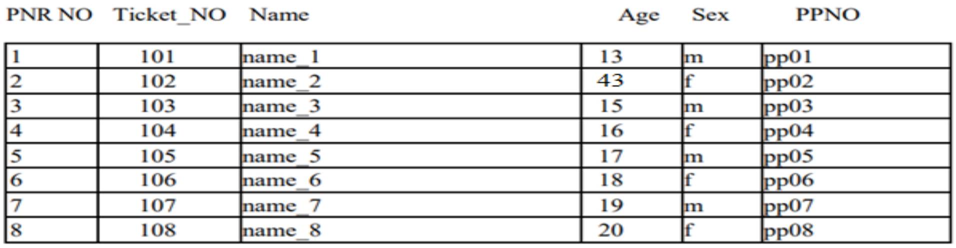
PNR NO	no of seats	address	contact	status
2	1	adrs_2	9892	N
3	1	adrs_3	9893	N

UPDATE: This command is used to update or modify the value of a column in the table.

Update passenger set age='43' where PNR_NO='2';

Result: 1row affected

Select * from passenger;



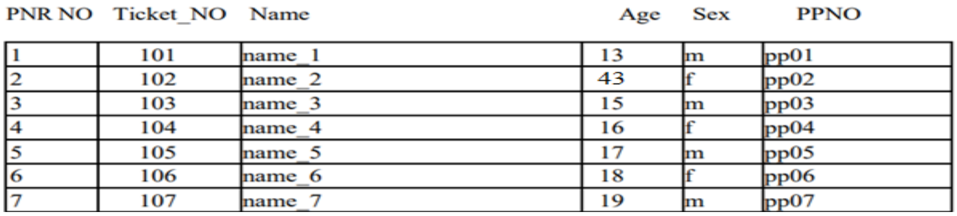
PNR NO	Ticket_NO	Name	Age	Sex	PPNO
1	101	name_1	13	m	pp01
2	102	name_2	43	f	pp02
3	103	name_3	15	m	pp03
4	104	name_4	16	f	pp04
5	105	name_5	17	m	pp05
6	106	name_6	18	f	pp06
7	107	name_7	19	m	pp07
8	108	name_8	20	f	pp08

DELETE: It is used to remove one or more row from a table.

```
delete from passenger where Name ='name_8';
```

Result: 1 row affected

Select * from passenger;



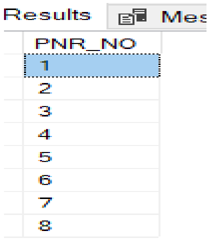
PNR NO	Ticket_NO	Name	Age	Sex	PPNO
1	101	name_1	13	m	pp01
2	102	name_2	43	f	pp02
3	103	name_3	15	m	pp03
4	104	name_4	16	f	pp04
5	105	name_5	17	m	pp05
6	106	name_6	18	f	pp06
7	107	name_7	19	m	pp07

Experiment 6: Querying (using ANY, ALL, IN, Exists, NOT EXISTS, UNION, INTERSECT, Constraints etc.)

1. Display Unique PNR_NO of all Passengers?

```
Select PNR_NO from Passenger;
```

Result:

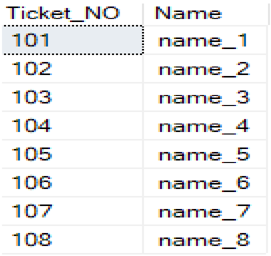


PNR_NO
1
2
3
4
5
6
7
8

2. Display Ticket numbers and names of all Passengers?

```
Select Ticket_NO, Name from Passenger;
```

Result:

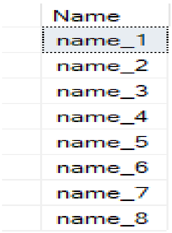


Ticket_NO	Name
101	name_1
102	name_2
103	name_3
104	name_4
105	name_5
106	name_6
107	name_7
108	name_8

3. ALL

ALL means that the condition will be true only if the operation is true for all values in the range.

```
SELECT ALL Name FROM passenger;
```



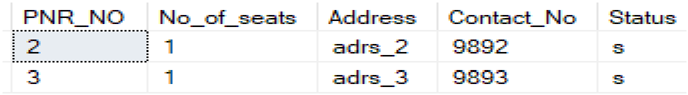
Name
name_1
name_2
name_3
name_4
name_5
name_6
name_7
name_8

4. ANY

ANY means that the condition will be true if the operation is true for any of the values in the range.

```
SELECT * FROM reservation where Contact_No=ANY(SELECT Contact_No FROM cancellation);
```

Result:



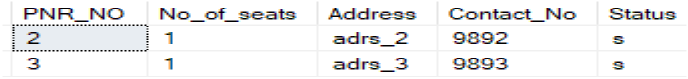
PNR_NO	No_of_seats	Address	Contact_No	Status
2	1	adrs_2	9892	s
3	1	adrs_3	9893	s

5. IN

The IN operator allows you to specify multiple values in a WHERE clause.

```
SELECT * FROM reservation where Contact_No IN (SELECT Contact_No FROM
cancellation);
```

RESULT:



PNR_NO	No_of_seats	Address	Contact_No	Status
2	1	adrs_2	9892	s
3	1	adrs_3	9893	s

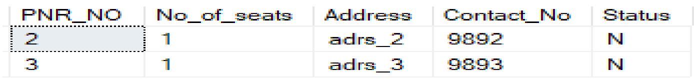
6. EXISTS

The EXISTS operator is used to test for the existence of any record in a subquery.

it returns TRUE if the subquery returns one or more records.

```
SELECT * FROM reservation where EXISTS (SELECT Contact_No FROM cancellation where
Contact_No=9892);
```

Result:

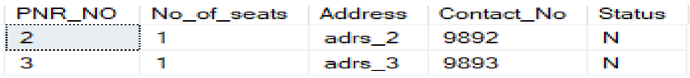


PNR_NO	No_of_seats	Address	Contact_No	Status
2	1	adrs_2	9892	N
3	1	adrs_3	9893	N

7. NOT EXISTS

NOT EXISTS allows locating records that don’t match the subquery.

```
SELECT * FROM cancellation where NOT EXISTS (SELECT Contact_No FROM reservation where
Contact_No=9890);
```



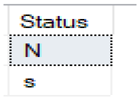
PNR_NO	No_of_seats	Address	Contact_No	Status
2	1	adrs_2	9892	N
3	1	adrs_3	9893	N

8. UNION

The UNION operator is used to combine the result-set of two or more SELECT statements.

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

```
SELECT Status FROM reservation UNION SELECT Status from cancellation;
```

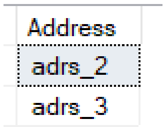


Status
N
s

9. INTERSECT

The INTERSECT is an operator in Structured Query Language that combines the rows of two SELECT statements and returns only those rows from the first SELECT statement, which are the same as the rows of the second SELECT statement.

```
SELECT Address FROM reservation INTERSECT SELECT Address from cancellation;
```



Address
adrs_2
adrs_3

Experiment 7:. Queries using Aggregate functions, GROUP BY, HAVING and Creation and dropping of Views.

SQL Aggregate Functions

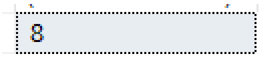
SQL aggregation function is used to perform the calculations on multiple rows of a single column of a table. It returns a single value.

COUNT FUNCTION

- COUNT function is used to Count the number of rows in a database table. It can work on both numeric and non-numeric data types.
- COUNT function uses the COUNT(*) that returns the count of all the rows in a specified table. COUNT(*) considers duplicate and Null.

```
select count(*) from passenger;
```

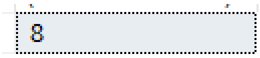
Result:



```
8
```

```
select count(Name) from passenger;
```

Result:



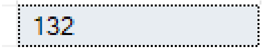
```
8
```

SUM Function

Sum function is used to calculate the sum of all selected columns. It works on numeric fields only.

```
select sum(Age) from passenger;
```

Result:



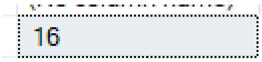
```
132
```

AVG function

The AVG function is used to calculate the average value of the numeric type. AVG function returns the average of all non-Null values.

```
select avg(Age) from passenger;
```

Result:



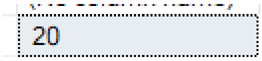
```
16
```

MAX Function

MAX function is used to find the maximum value of a certain column. This function determines the largest value of all selected values of a column.

```
select max(Age) from passenger;
```

Result:



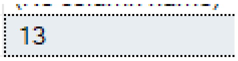
20

MIN Function

MIN function is used to find the minimum value of a certain column. This function determines the smallest value of all selected values of a column.

```
select min(Age) from passenger;
```

Result:



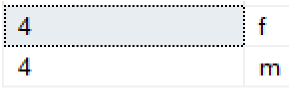
13

GROUP BY

In SQL, The Group By statement is used for organizing similar data into groups.

```
select count(Ticket_NO),Sex from passenger group by Sex;
```

Result:



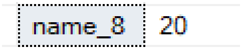
4	f
4	m

HAVING

The HAVING is a keyword in SQL which selects the rows filtered by the GROUP BY keyword based on the particular single or multiple conditions.

```
select Name, max(Age) from passenger group by Name having max(Age)>19;
```

Result:



name_8	20

SQL View

The **View** in the Structured Query Language is considered as the virtual table, which depends on the result-set of the predefined SQL statement.

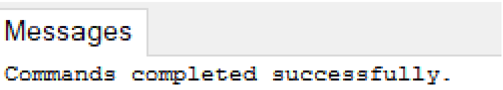
Like the SQL tables, Views also store data in rows and columns, but the rows do not have any physical existence in the database.

Create a SQL View

You can easily create a View in Structured Query Language by using the CREATE VIEW statement. You can create the View from a single table or multiple tables.

```
create view passenger_view as select Name, Age, Sex, Ticket_NO from passenger where
Age>16;
```

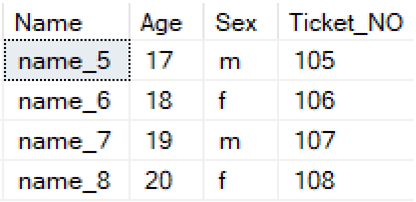
Result:



```
Commands completed successfully.
```

```
select * from passenger_view;
```

Result:



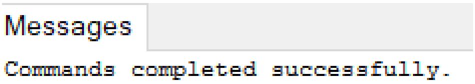
Name	Age	Sex	Ticket_NO
name_5	17	m	105
name_6	18	f	106
name_7	19	m	107
name_8	20	f	108

Create a View from Multiple tables

Let us consider two tables passenger and reservation, we can use these two tables we can create view from multiple talbes

```
create view passenger_reservation_view as select Name, Age, Sex, Ticket_NO,Address,
Contact_No from passenger,reservation;
```

Result:



```
Commands completed successfully.
```

```
Select * from passenger_reservation_view;
```

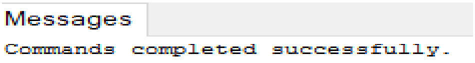
Result: (only practice it in lab its shows many rows and select colums)

Drop a View

We can also delete the existing view from the database if it is no longer needed. The following SQL DROP statement is used to delete the view:

```
drop view passenger_reservation_view;
```

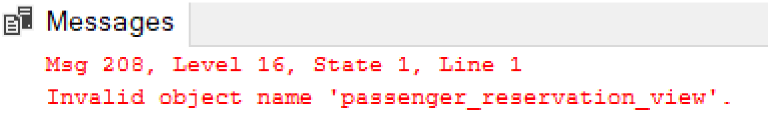
Result:



```
Commands completed successfully.
```

```
Select * from passenger_reservation_view;
```

Result:



```
Msg 208, Level 16, State 1, Line 1
Invalid object name 'passenger_reservation_view'.
```

Experiment 7: Triggers (Creation of insert trigger, delete trigger, update trigger)

Trigger in SQL

A **Trigger** in Structured Query Language is a set of procedural statements which are executed automatically when there is any response to certain events on the particular table in the database. Triggers are used to protect the data integrity in the database.

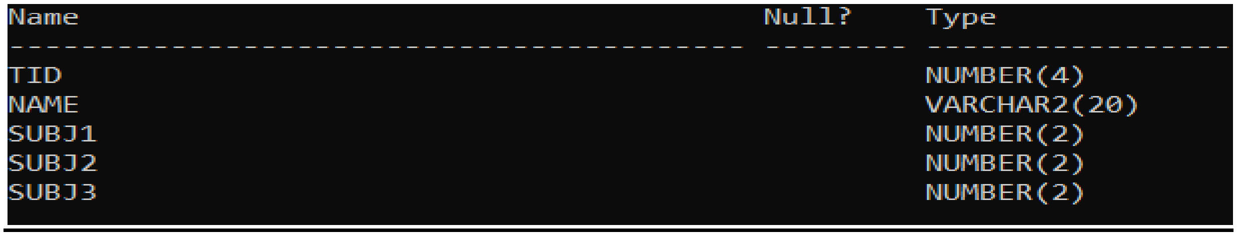
Creating a Trigger table

create table stud (tid number(4), name varchar(20), subj1 number(2),subj2 number(2),subj3 number(2));

Result: Table is created

desc stud;(In sql use this command: `exec sp_help stud;`

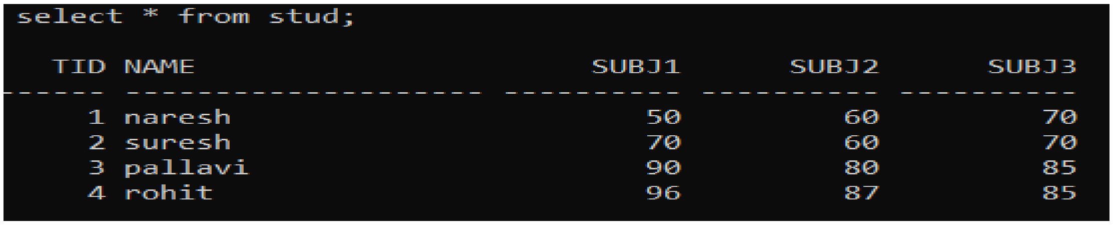
Result:



```
Name                                      Null?    Type
----------------------------------------- -------- ----------------
TID                                                NUMBER(4)
NAME                                               VARCHAR2(20)
SUBJ1                                              NUMBER(2)
SUBJ2                                              NUMBER(2)
SUBJ3                                              NUMBER(2)
```

insert into stud values(1, 'naresh', 50,60,70);
Result: 1 row effected
insert into stud values(2, 'suresh', 70,60,70);
Result: 1 row effected
insert into stud values(3, 'pallavi', 90,80,85);
Result: 1 row effected
insert into stud values(4, 'rohit', 96,87,85);
Result: 1 row effected

select * from stud;



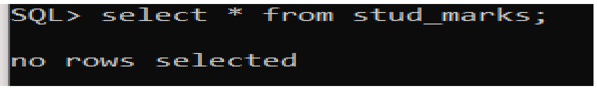
```
select * from stud;

  TID NAME                      SUBJ1      SUBJ2      SUBJ3
----- -------------------- ---------- ---------- ----------
    1 naresh                       50         60         70
    2 suresh                       70         60         70
    3 pallavi                      90         80         85
    4 rohit                        96         87         85
```

Creating a Trigger backup table

create table stud_marks (tid number(4), name varchar(20), subj1 number(2),subj2 number(2),subj3 number(2));

Result: Table is created

desc stud_marks;

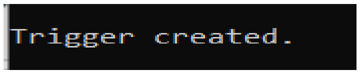


```
SQL> select * from stud_marks;

no rows selected
```

Now, we will create a trigger that stores student marks of each insert operation on the **stud** table into the **stud_marks**. Here we are going to create the insert trigger using the below statement:

```
create or replace trigger t1
 before delete on stud
 for each row
 begin
 insert into stud_marks values(:old.tid, :old.name,:old.subj1,:old.subj2,:old.subj3);
 end;
 /
```

Result:



```
Trigger created.
```

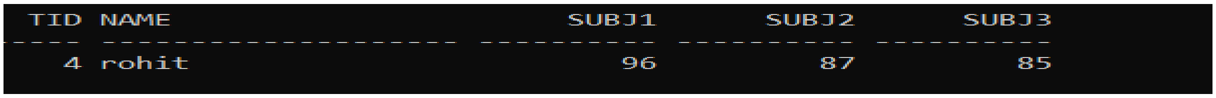
Now, delete one record from stud table by using delete command..

delete from stud where tid=4;

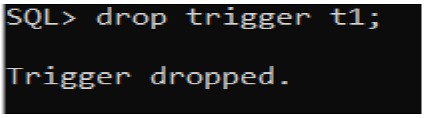
Result: ***1 row deleted***

Finally check the backup table(stud_marks) it will be updated missing data from stud_marks by using trigger t1.

Select * from stud_marks;



```
  TID NAME                      SUBJ1      SUBJ2      SUBJ3
----- -------------------- ---------- ---------- ----------
    4 rohit                        96         87         85
```



```
SQL> drop trigger t1;

Trigger dropped.
```

Experiment 9: Procedures

A **Procedure** in PL/SQL is a subprogram unit that consists of a group of PL/SQL statements that can be called by name. Each procedure in PL/SQL has its own unique name by which it can be referred to and called. This subprogram unit in the Oracle database is stored as a database object.

Creating a Procedure

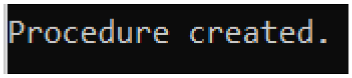
A procedure is created with the **CREATE OR REPLACE PROCEDURE** statement.

Example

The following example creates a simple procedure that displays the string 'Hello World!' on the screen when executed.

```
CREATE OR REPLACE PROCEDURE greetings
AS
BEGIN
  dbms_output.put_line('Hello World!');
END;
/
```

Result:

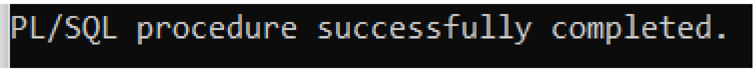


```
Procedure created.
```

The above procedure named 'greetings' can be called with the EXECUTE keyword as −

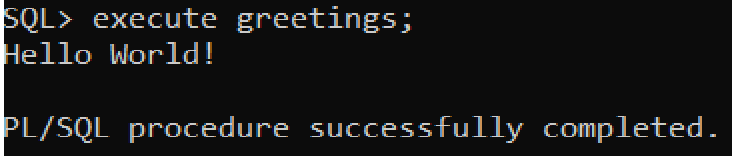
EXECUTE greetings;

Result:



```
PL/SQL procedure successfully completed.
```

If we see hello world in the above output use command as ***set serveroutput on*** then execute it.



```
SQL> execute greetings;
Hello World!

PL/SQL procedure successfully completed.
```

Example:

First create one emp table with name and salary then we can increment salary by using procedures.

```
create table emp (eid number(5) primary key, name varchar(20), sal number(10));
```

Result: table created

insert into emp values (1, 'ashwin', 10000);

Result: 1 row created.

insert into emp values (2,'bumrha', 12000);

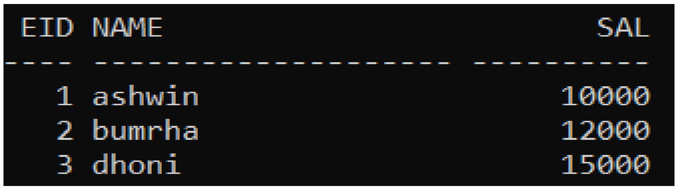
Result: 1 row created.

insert into emp values (3,'dhoni', 15000);

Result: 1 row created.

Select * from emp;

Result:

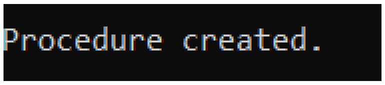


```
 EID NAME                        SAL
---- -------------------- ----------
   1 ashwin                    10000
   2 bumrha                    12000
   3 dhoni                     15000
```

Now, create a procedure query to increment salary.

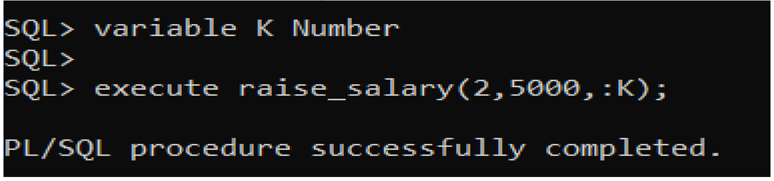
```
   CREATE OR REPLACE PROCEDURE
   raise_salary(E IN NUMBER, AMT IN NUMBER, S OUT NUMBER)
  IS
  BEGIN
  UPDATE emp SET sal=sal+AMT
  where eid=E;
  commit;
 SELECT sal INTO S FROM emp WHERE eid=E;
 END;
  /
```

Result:



```
Procedure created.
```

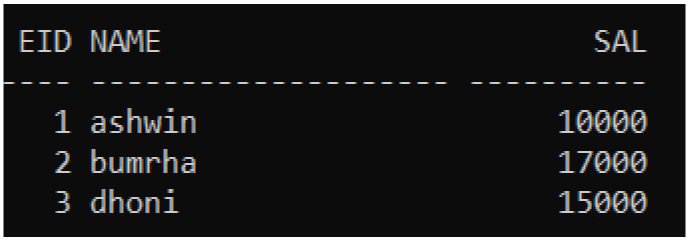
Now follow steps to assign one more variable and how much salary is incrementing



```
SQL> variable K Number
SQL>
SQL> execute raise_salary(2,5000,:K);

PL/SQL procedure successfully completed.
```

select * from emp;

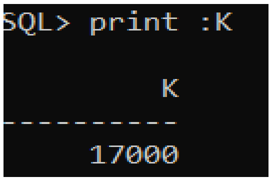
Result:



```
EID NAME                        SAL
---- -------------------- ----------
   1 ashwin                    10000
   2 bumrha                    17000
   3 dhoni                     15000
```

Or

Print updated salary by using declared variable



```
SQL> print :K

         K
----------
     17000
```

Experiment 10: **Usage of Cursors**

Whenever DML statements are executed, a temporary work area is created in the system memory and it is called a cursor. A cursor can have more than one row, but processing wise only 1 row is taken into account. Cursors are very helpful in all kinds of databases like Oracle, SQL Server, MySQL, etc. They can be used well with DML statements like Update, Insert and Delete. Especially Implicit cursors are there with these operations.

In PL/SQL, two different types of cursors are available.

- Implicit cursors
- Explicit cursors

Implicit cursors

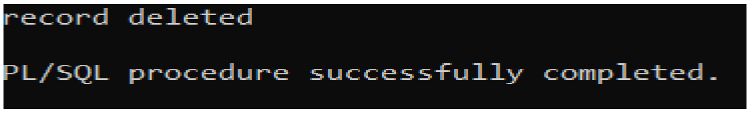
Orcale provides some attributes known as Implicit cursor's attributes to check the status of DML operations. Some of them are: %FOUND, %NOTFOUND, %ROWCOUNT and %ISOPEN.

Let us practice these commands with previous emp table

```
declare
cursor1 emp.eid%type;
begin
 cursor1:= &eid;
delete from emp where eid=cursor1;
if SQL%found then
dbms_output.put_line('record deleted');
else
dbms_output.put_line(' no record');
end if;
commit;
end;
 /
```

```
Enter value for eid: 3
old   4: cursor1:= &eid;
new   4: cursor1:= 3;
```

Result:



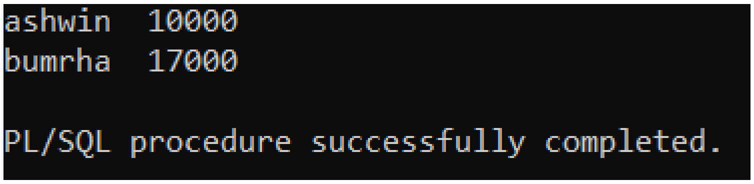
```
record deleted

PL/SQL procedure successfully completed.
```

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.

```
SQL> declare
  2  cursor c1 is select name, sal from emp;
  3  vname emp.name%type;
  4  vsal emp.sal%type;
  5  begin
  6  open c1;
  7  loop
  8  fetch c1 into vname, vsal;
  9  exit when c1%notfound;
 10  dbms_output.put_line(vname||'  '||vsal);
 11  end loop;
 12  close c1;
 13  end;
 14  /
```

Result:



```
ashwin  10000
bumrha  17000

PL/SQL procedure successfully completed.
```