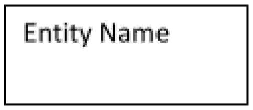
| Experiment No.1 |
|---|
| Identify the case study and detail statement of problem. Design an Entity-Relationship (ER) / Extended Entity-Relationship (EER) Model. |
| Date of Performance:09/01 |
| Date of Submission:23/01 |

**Aim**: Identify the case study and detail statement of problem.
Design an Entity-Relationship (ER) / Extended Entity-Relationship (EER) Model.

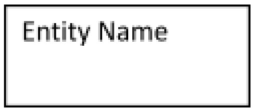
**Objective:** To show the relationships of entity sets attributes and relationships stored in a database.

**Theory**: Summary of ER, EER Diagram Notation

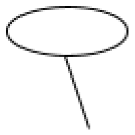
Strong Entities



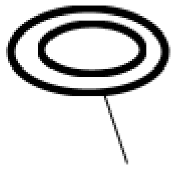
Weak Entities



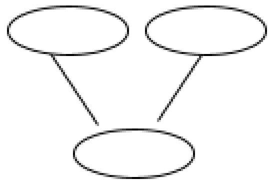
Attributes



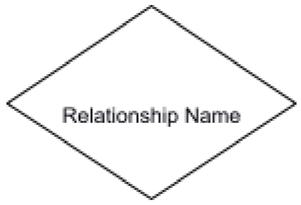
Multi Valued Attributes [Double Ellipse]



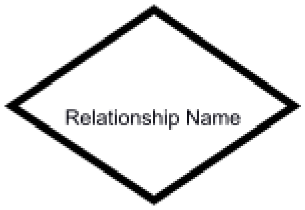
Composite Attributes



Relationships



Identifying Relationships



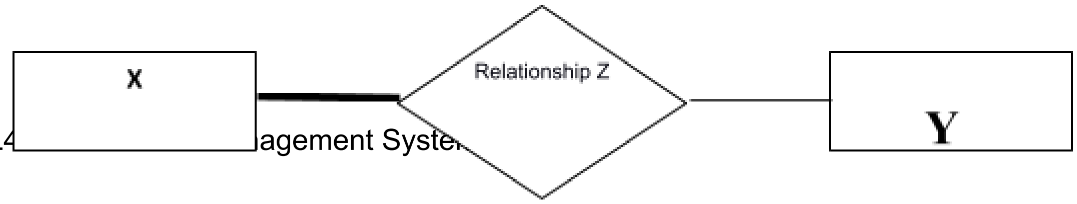
**N-ary relationships**

More than 2 participating entities

Constraints - Participation

. **Total Participation** - entity X has total participation in Relationship Z, meaning that every instance of X takes part in AT LEAST one relationship. (i.e. there are no members of X that do not participate in the relationship.

*Example*: X is Customer, Y is Product, and Z is a 'Purchases' relationship. The figure below indicates the requirement that every customer purchases a product.



. **Partial Participation** - entity Y has partial participation in Relationship Z, meaning that only some instances of Y take part in the relationship.

*Example*: X is Customer, Y is Product, and Z is a 'Purchases' relationship. The figure below indicates the requirement that not every product is purchases by a customer.

**Cardinality:**

. 1:N – One Customer buys many products, each product is purchased by only one customer.

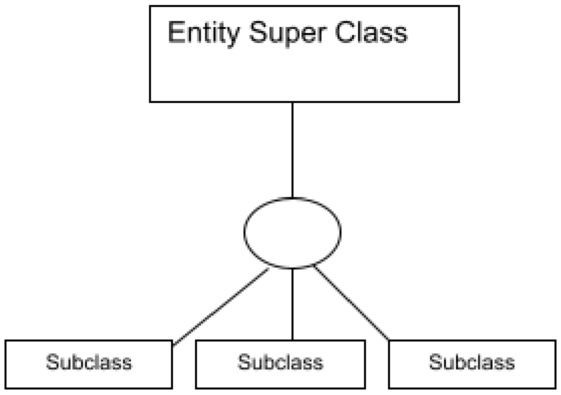
N:1 - Each customer buys at most one product, each product can be purchased by many customers.

1:1 – Each customer purchases at most one product, each product is purchased by only one customer.

M:N – Each customer purchases many products, each product is purchased by many customers.

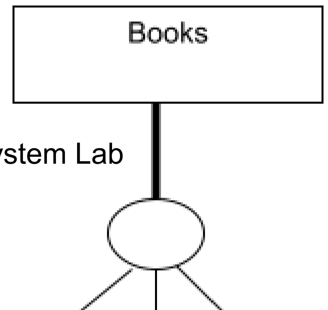
Specialization/Generalization

. Each subclass inherits all relationships and attributes from the super-class.



Constraints on Specialization/Generalization

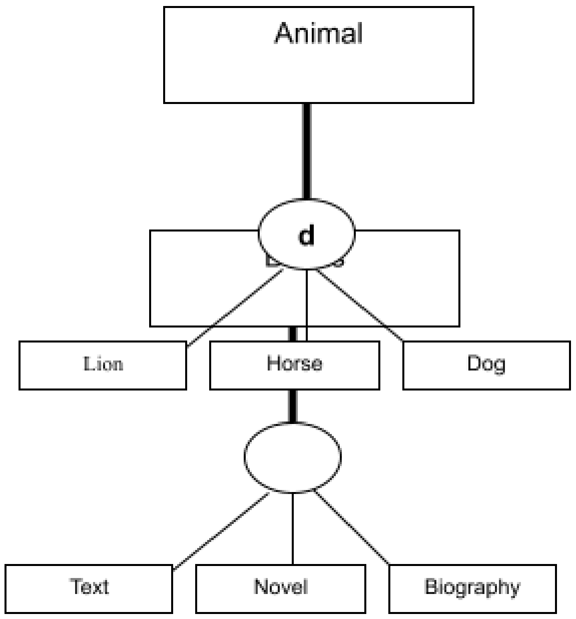
**Total Specialization** – Every member of the super-class must belong to at least one subclass. For example, any book that is not a text book, or a novel can fit into the "Other" category.



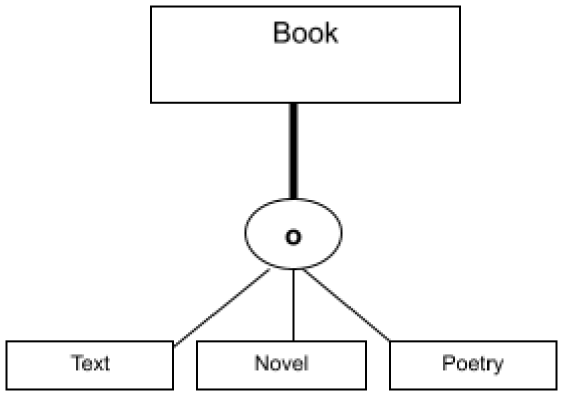
**Partial Specialization** – each member of the super-class may not belong to one of the subclasses. For example, a book on poetry may be neither a text book, a novel or a biography.

Disjointness Constraint

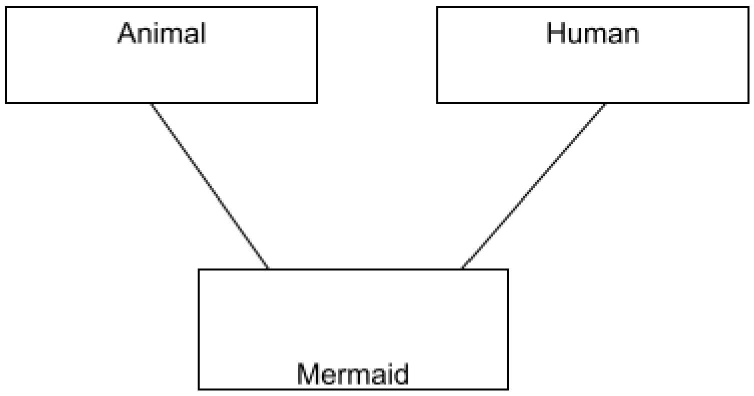
- **Disjoint** – every member of the super-class can belong to at most one of the subclasses. For example, an Animal cannot be a lion and a horse, it must be either a lion, a horse, or a dog.



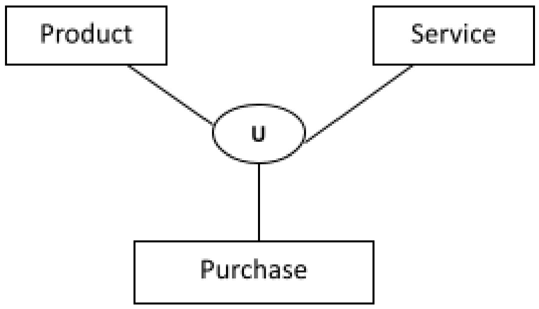
**Overlapping** – every member of the super-class can belong to more than one of the subclasses. For example, a book can be a text book, but also a poetry book at the same time.



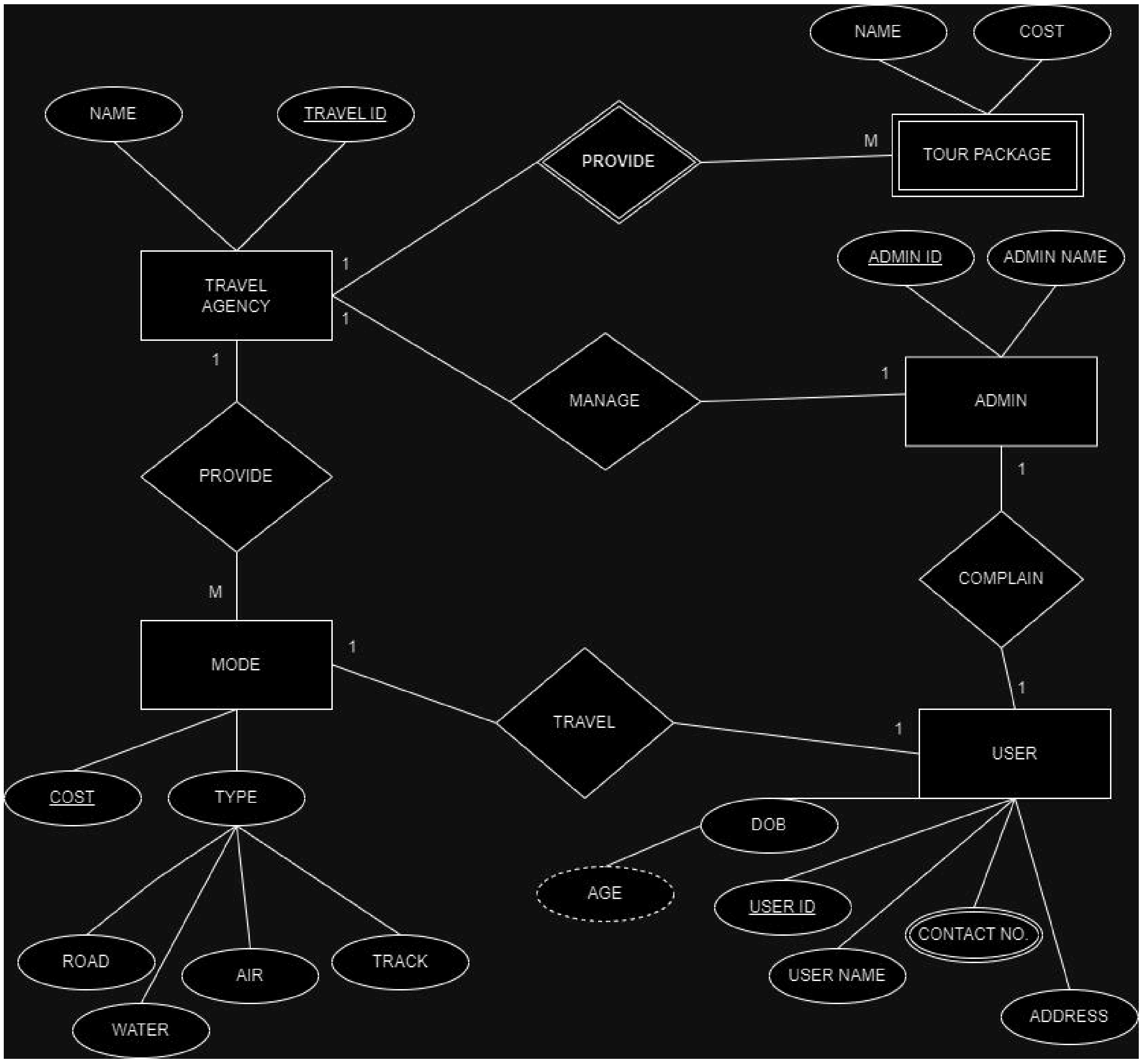
**Multiple Inheritance** – a subclass participates in more than one subclass/super-class relationship, and inherits attributes and relationships from more than one super-class. For example, the subclass Mermaid participates in two subclass/super-class relationships, it inherits attributes and relationships of Animals, as well as attributes and relationships of Humans.



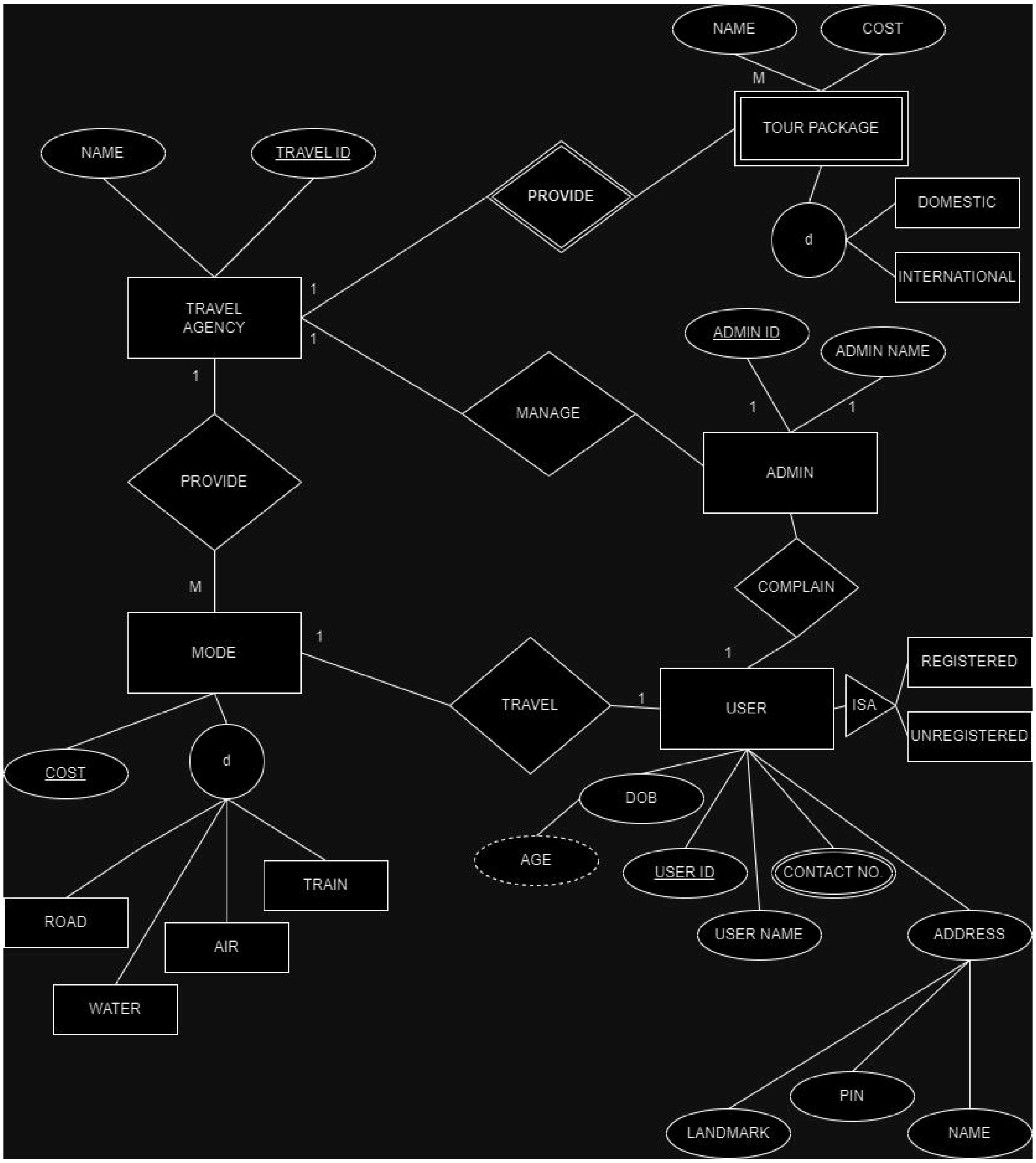
**Union** – a subclass/super-class relationship can have more than one super-class, and the subclass inherits from at most one of the super-classes (i.e. the subclass purchase will inherit the relationships and attributes associated with either service or product, but not both). Each super class may have different primary keys, or the same primary key. All members of the super-classes are not members of the super-class. For example, a purchase can be a product, or a service, but not both. And all products and services are not purchase



**Implementation:(ER)**



**(EER):**



**Conclusion:-** The relationships among entity sets, attributes, and relationships in a database are fundamental components of database design and management. By carefully defining and organizing these elements, database designers can create efficient, scalable, and maintainable database systems that effectively capture and represent the underlying data requirements of an organization or application.

| |
|---|
| Experiment No.2 |
| Mapping ER/EER to Relational schema model. |
| Date of Performance:23/01 |
| Date of Submission:29/01 |

**Aim**: Mapping ER/EER to Relational schema model.

**Objective:** Objective of design is to generate a formal specification of the database schema.

**Theory: Mapping Rules**

**Step 1: Regular Entity Types**
Create an *entity relation* for each strong entity type. Include all single-valued attributes. Flatten composite attributes. Keys become secondary keys, except for the one chosen to be the primary key.

**Step 2: Weak Entity Types**
Also create an entity relation for each weak entity type, similarly including its (flattened) single-valued attributes. In addition, add the primary key of each owner entity type as a foreign key attribute here. Possibly make this foreign key CASCADE.

**Step 3: Binary 1:1 Relationship Types**
Let the relationship be of the form [S]——<R>——[T].

1. **Foreign key approach**: The primary key of T is added as a foreign key in S. Attributes of R are moved to S (possibly renaming them for clarity). If only one of the entities has total participation it's better to call it S, to avoid null attributes. If neither entity has total participation nulls may be unavoidable. *This is the preferred approach in typical cases.*
2. **Merged relation approach**: Both entity types are stored in the same relational table, "pre-joined". If the relationship is not total both ways, there will be null padding on tuples that represent just one entity type. Any attributes of R are also moved to this table.
3. **Cross-reference approach**: A separate relation represents R; each tuple is a foreign key from S and a foreign key from T. Any attributes of R are also added to this relation. Here foreign keys should CASCADE.

| Approach | Join cost | Null-storage cost |
|---|---|---|
| Foreign key | 1 | low to moderate |
| Merged relation | 0 | very high, unless both are total |
| Cross-reference | 2 | None |

**Step 4: Binary 1:N Relationship Types**

Let the relationship be of the form [S]——$^{N}$<R>$^{1}$——[T]. The primary key of T is added as a foreign key in S. Attributes of R are moved to S. This is the foreign key approach. The merged relation approach is not possible for 1:N relationships. (Why?) The cross-reference approach might be used if the join cost is worth avoid null storage.

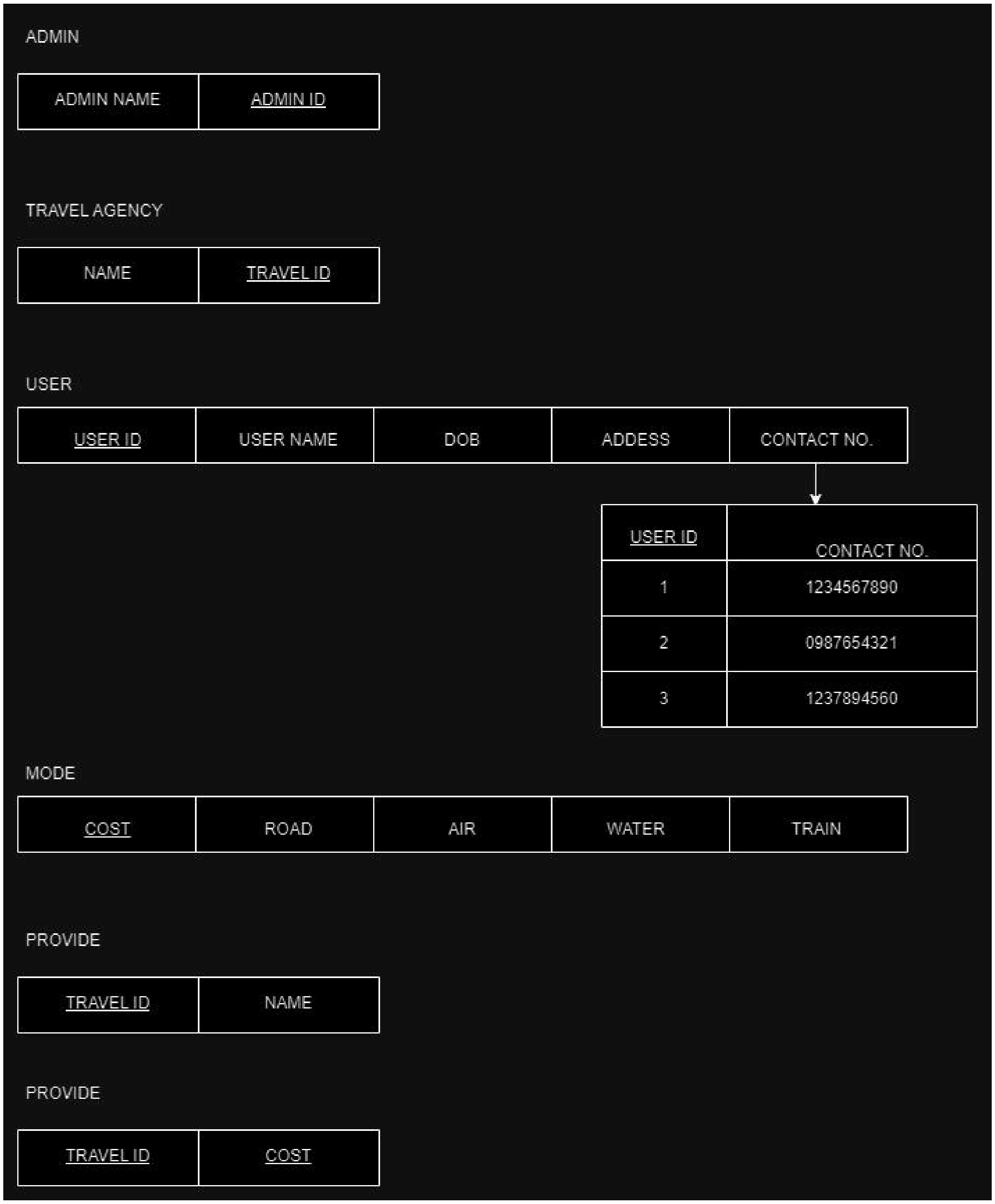
**Step 5: Binary M:N Relationship Types**
Here the cross-reference approach (also called a *relationship relation*) is the only possible way.

**Step 6: Multivalued Attributes**
Let an entity S have multivalued attribute A. Create a new relation R representing the attribute, with a foreign key into S added. The primary key of R is the combination of the foreign key and A. Once again this relation is dependent on an "owner relation" so its foreign key should CASCADE.

**Step 7: Higher-Arity Relationship Types**

Here again, use the cross-reference approach. For each n-ary relationship create a relation to represent it. Add a foreign key into each participating entity type. Also add any attributes of the relationship. The primary key of this relation is the combination of all foreign keys into participating entity types *that do not have a max cardinality of 1.*

**Implementation:**



ADMIN

| ADMIN NAME | ADMIN ID |
|---|---|

TRAVEL AGENCY

| NAME | TRAVEL ID |
|---|---|

USER

| USER ID | USER NAME | DOB | ADDESS | CONTACT NO. |
|---|---|---|---|---|

| USER ID | CONTACT NO. |
|---|---|
| 1 | 1234567890 |
| 2 | 0987654321 |
| 3 | 1237894560 |

MODE

| COST | ROAD | AIR | WATER | TRAIN |
|---|---|---|---|---|

PROVIDE

| TRAVEL ID | NAME |
|---|---|

PROVIDE

| TRAVEL ID | COST |
|---|---|

**Conclusion:** Generating a formal specification of the database schema is crucial for ensuring clarity, standardization, communication, documentation, implementation guidance, and quality assurance in the database design process. It serves as a foundational step towards developing robust, reliable, and efficient database systems that effectively support the data management needs of organizations and applications.

| |
|---|
| Experiment No.3 |
| Create and populate database using Data Definition Language (DDL) and Apply Integrity Constraints for the specified system |
| Date of Performance:29/01 |
| Date of Submission:30/01 |

**Aim**: Create and populate database using Data Definition Language (DDL) and apply Integrity

Constraints for the specified system

**Objective:** DDL or Data Definition Language actually consists of the SQL commands that can be used to define the database schema. It simply deals with descriptions of the database schema and is used to create and modify the structure of database objects in the database.

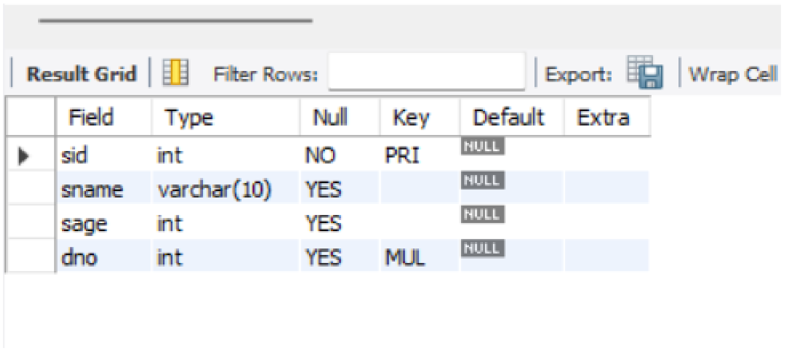
Integrity constraints are used to ensure accuracy and consistency of data in a relational database. Data integrity is handled in a relational database through the concept of referential integrity

**Theory**: DDL Commands

Create a table
Display the table description
Rename the table
Alter the table
Drop the table
Integrity constraints are:

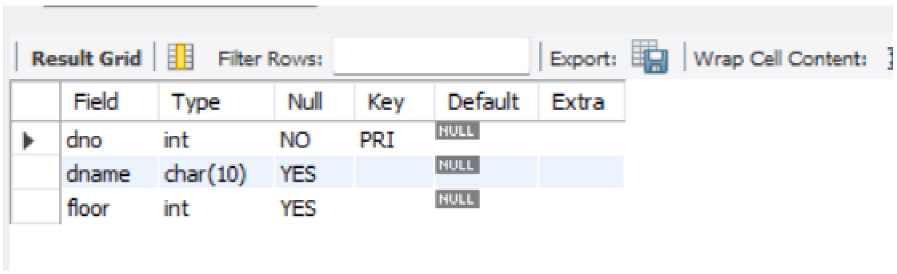
1. PRIMARY KEY CONSTRAINTS
2. FOREIGN KEY CONSTRAINTS
3. NULL CONSTRAINTS
4. NOT NULL CONSTRAINTS
5. CHECK CONSTRAINTS
6. DEFAULT CONSTRAINTS

**Implementation:**



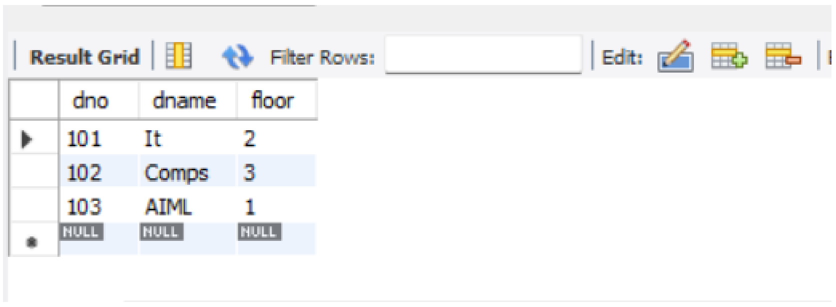
Result Grid | Filter Rows: | Export: | Wrap Cell

| Field | Type | Null | Key | Default | Extra |
|---|---|---|---|---|---|
| sid | int | NO | PRI | NULL | |
| sname | varchar(10) | YES | | NULL | |
| sage | int | YES | | NULL | |
| dno | int | YES | MUL | NULL | |



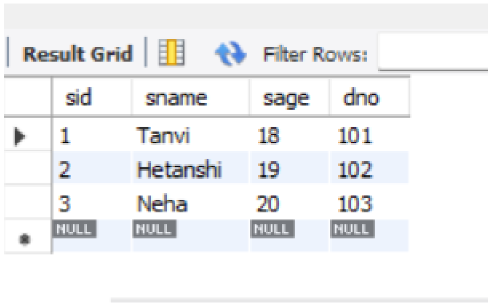
Result Grid | Filter Rows: | Export: | Wrap Cell Content:

| Field | Type | Null | Key | Default | Extra |
|---|---|---|---|---|---|
| dno | int | NO | PRI | NULL | |
| dname | char(10) | YES | | NULL | |
| floor | int | YES | | NULL | |



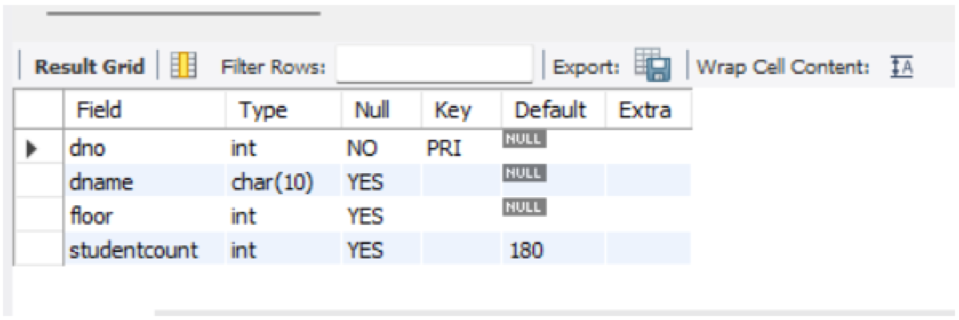
Result Grid | Filter Rows: | Edit:

| dno | dname | floor |
|---|---|---|
| 101 | It | 2 |
| 102 | Comps | 3 |
| 103 | AIML | 1 |
| NULL | NULL | NULL |



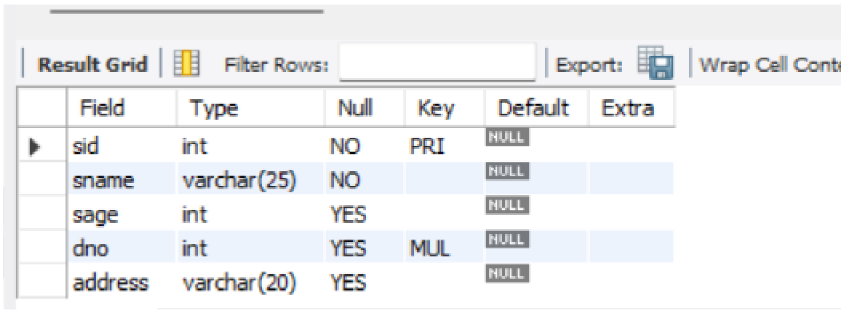
Result Grid | Filter Rows:

| sid | sname | sage | dno |
|---|---|---|---|
| 1 | Tanvi | 18 | 101 |
| 2 | Hetanshi | 19 | 102 |
| 3 | Neha | 20 | 103 |
| NULL | NULL | NULL | NULL |



Result Grid | Filter Rows: | Export: | Wrap Cell Content:

| Field | Type | Null | Key | Default | Extra |
|---|---|---|---|---|---|
| dno | int | NO | PRI | NULL | |
| dname | char(10) | YES | | NULL | |
| floor | int | YES | | NULL | |
| studentcount | int | YES | | 180 | |



Result Grid | Filter Rows: | Export: | Wrap Cell Cont

| Field | Type | Null | Key | Default | Extra |
|---|---|---|---|---|---|
| sid | int | NO | PRI | NULL | |
| sname | varchar(25) | NO | | NULL | |
| sage | int | YES | | NULL | |
| dno | int | YES | MUL | NULL | |
| address | varchar(20) | YES | | NULL | |

```
create table dep

(

dno int primary key,

dname char (10),

floor int check(floor<5)

);



create table std

(

sid int primary key,

sname varchar (10),

sage int check(sage<25),

dno int,

foreign key(dno) references dep(dno)

);

desc std;

desc dep;

insert into std values(1,'Tanvi',18,101);
insert into std values(2,'Hetanshi',19,102);

insert into std values(3,'Neha',20,103);

insert into dep(dno,dname,floor) values(101,'It',2);

insert into dep(dno,dname,floor) values(102,'Comps',3);

insert into dep(dno,dname,floor) values(103,'AIML',1);

select *from dep;

select *from std;

alter table dep

add studentcount int default 180;

desc dep;

alter table std

add address varchar(20) null,

modify sname varchar(25) not null;

alter table std

rename to compstd;

desc compstd;
```

**Conclusion:** A database can be created, populated, and maintained using DDL commands while ensuring data integrity through the application of integrity constraints. Properly defined constraints help in preventing erroneous data entry, maintaining data consistency, and improving overall database reliability. Additionally, thorough testing and periodic updates contribute to the efficient management and usability of the database system.

| |
|---|
| Experiment No.4 |
| Apply DML Commands for your the specified System. |
| Date of Performance:30/01 |
| Date of Submission:06/02 |

**Aim**:- Apply DML Commands for your the specified System

**Objective:** The SQL commands that deals with the manipulation of data present in the database belong to DML or Data Manipulation Language and this includes most of the SQL statements.

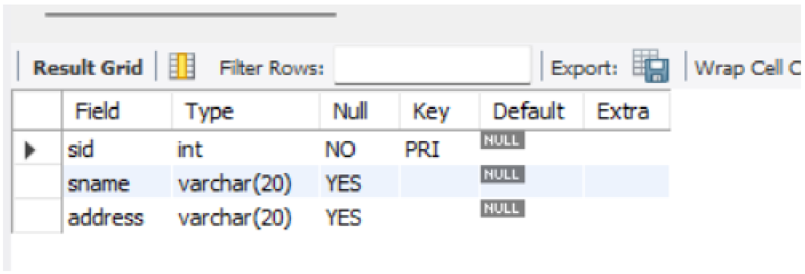
**Theory**:

DML:-DATA MANIPULATION LANGUAGE

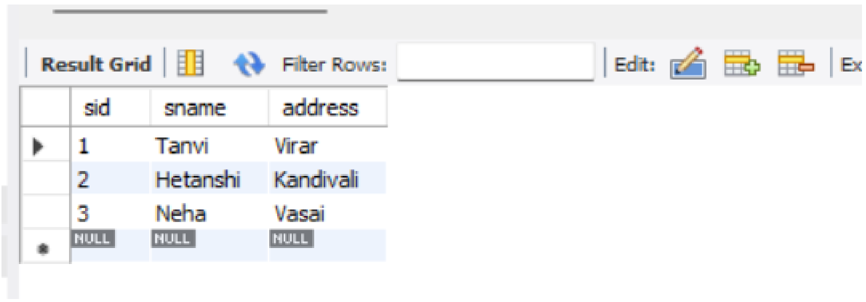
Commands used in DML are

Insert Values
Retrieve all attributes
Update table
Delete table

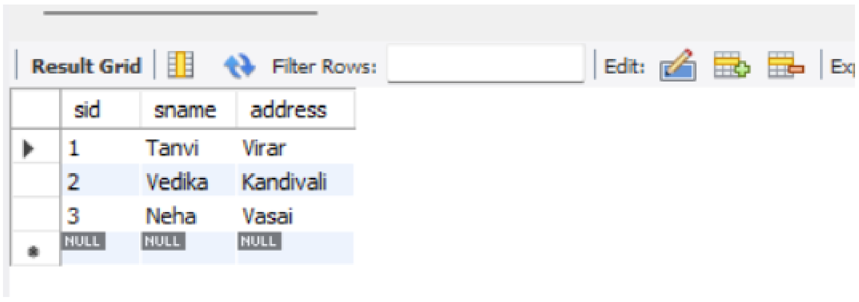
**Implementation:**



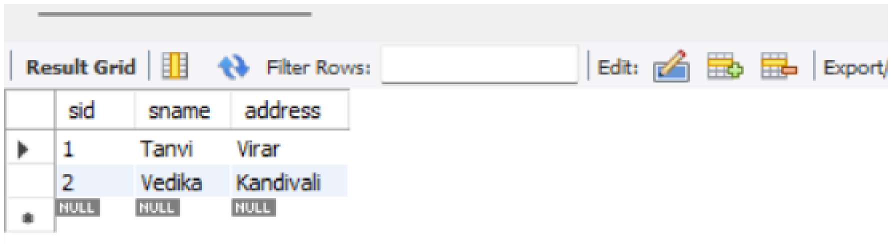
| Field | Type | Null | Key | Default | Extra |
|---|---|---|---|---|---|
| sid | int | NO | PRI | NULL | |
| sname | varchar(20) | YES | | NULL | |
| address | varchar(20) | YES | | NULL | |



| sid | sname | address |
|---|---|---|
| 1 | Tanvi | Virar |
| 2 | Hetanshi | Kandivali |
| 3 | Neha | Vasai |
| NULL | NULL | NULL |



| | sid | sname | address |
|---|---|---|---|
| ▶ | 1 | Tanvi | Virar |
| | 2 | Vedika | Kandivali |
| | 3 | Neha | Vasai |
| * | NULL | NULL | NULL |



| | sid | sname | address |
|---|---|---|---|
| ▶ | 1 | Tanvi | Virar |
| | 2 | Vedika | Kandivali |
| * | NULL | NULL | NULL |

```
create table std
(
sid int primary key,
sname varchar (10),
sage int
);
alter table std
add address varchar(20),
rename to compstd1,
drop column sage,
modify sname varchar(20);
insert into compstd1 values(1,'Tanvi',"Virar");
insert into compstd1 values(2,'Hetanshi',"Kandivali");
insert into compstd1 values(3,'Neha',"Vasai");
select * from compstd1;
```

```
update compstd1

set sname="Vedika"

where sid=2;

desc compstd1;

delete from compstd1

where sid=3;
```

**Conclusion:** DML commands are indispensable for manipulating data within a database system. They facilitate the retrieval, insertion, modification, and deletion of data, allowing for efficient data management and maintenance. Proper use of DML commands ensures data integrity, consistency, and accuracy within the database, contributing to the overall reliability and effectiveness of the database system.

| Experiment No.5 |
|---|
| Perform Simple queries, string manipulation operations and aggregate functions |
| Date of Performance:06/02 |
| Date of Submission:13/02 |

**Aim:-** Perform Simple queries and aggregate functions.

**Objective:** Queries are a way of searching for and compiling data from one or more tables .aggregate functions are used to find Average, Maximum and minimum values, count values from given database

**Theory:**

Student (sid,sname,city,age, Marks)

Department(did, dname, sid)

Q1. Create a table student with given attributes.

Q2. Create a table department with given attributes.

Q3. Insert values into the respective tables & display them.

Q4. Update any row from student relation

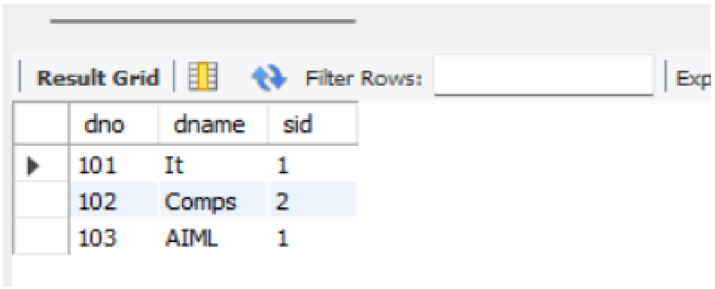
Q5. Delete any row from the department table.

Q6. Give the minimum age of the student relation.

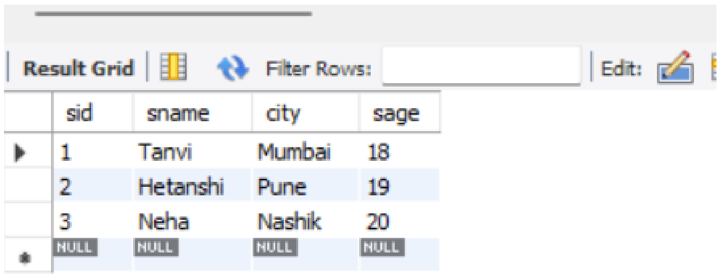
Q7. Find out the avg of marks of the student relation.

Q8. Give the total count of tuples in department relation group by did.

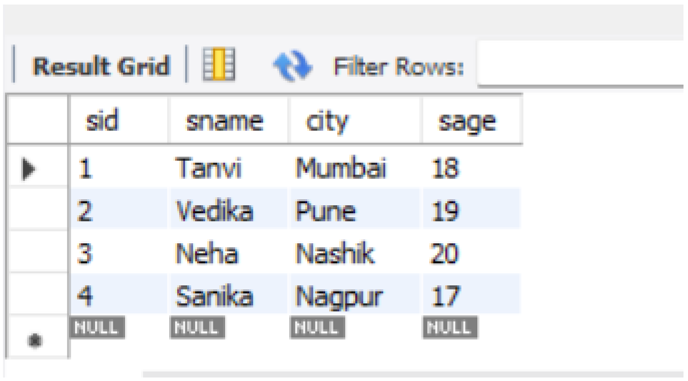
**Implementation:**



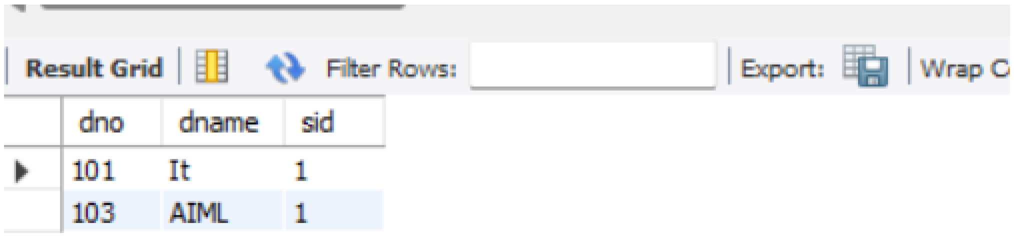
| dno | dname | sid |
|---|---|---|
| 101 | It | 1 |
| 102 | Comps | 2 |
| 103 | AIML | 1 |



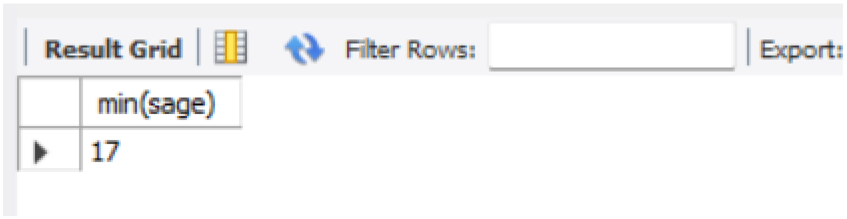
| sid | sname | city | sage |
|---|---|---|---|
| 1 | Tanvi | Mumbai | 18 |
| 2 | Hetanshi | Pune | 19 |
| 3 | Neha | Nashik | 20 |
| NULL | NULL | NULL | NULL |



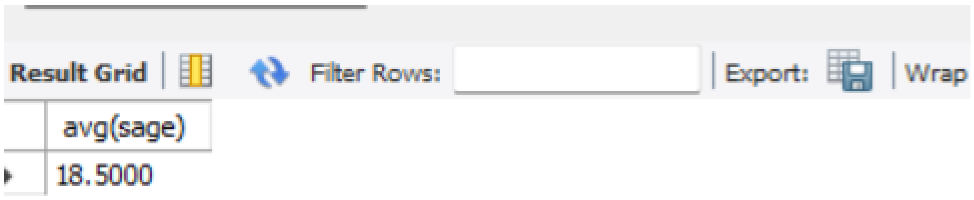
| sid | sname | city | sage |
|---|---|---|---|
| 1 | Tanvi | Mumbai | 18 |
| 2 | Vedika | Pune | 19 |
| 3 | Neha | Nashik | 20 |
| 4 | Sanika | Nagpur | 17 |
| NULL | NULL | NULL | NULL |


Result Grid | Filter Rows: | Export: | Wrap C

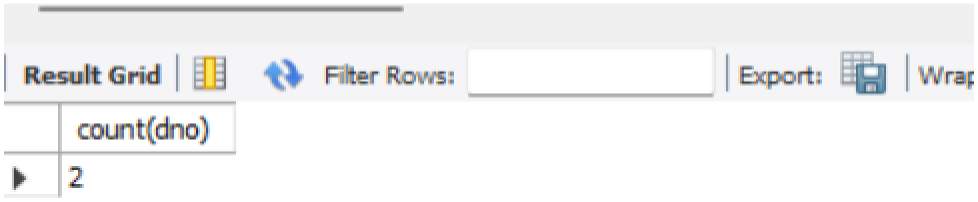
| dno | dname | sid |
|---|---|---|
| 101 | It | 1 |
| 103 | AIML | 1 |


Result Grid | Filter Rows: | Export:

| min(sage) |
|---|
| 17 |


Result Grid | Filter Rows: | Export: | Wrap

| avg(sage) |
|---|
| 18.5000 |


Result Grid | Filter Rows: | Export: | Wrap

| count(dno) |
|---|
| 2 |

```
create table std

(

sid int primary key,

sname varchar (10),

city varchar(10),

sage int check(sage<25)
);


create table dep
(
dno int,
dname char (10),
sid int,
Foreign key(sid) references std (sid)
);




desc std;
desc dep;
insert into std values(1,'Tanvi',"Mumbai",18);
insert into std values(2,'Hetanshi',"Pune",19);
insert into std values(3,'Neha',"Nashik",20);
insert into dep values(101,'It',1);
insert into dep values(102,'Comps',2);
insert into dep values(103,'AIML',1);
select *from dep;
select *from std;
update std set sname="Vedika" where sid=2;
insert std values(4,"Sanika",'Nagpur',17);
```

```
select * from std;

select min(sage) from std;

select avg(sage) from std;

select count(dno) from dep;
```

**Conclusion:** DML commands are indispensable for manipulating data within a database system. They facilitate the retrieval, insertion, modification, and deletion of data, allowing for efficient data management and maintenance. Proper use of DML commands ensures data integrity, consistency, and accuracy within the database, contributing to the overall reliability and effectiveness of the database system.s

| |
|---|
| Experiment No.6 |
| Implement Pattern Matching Queries. |
| Date of Performance:20/02 |
| Date of Submission:27/02 |

**Aim**: **Implement Pattern Matching Queries.**

**Objective:** SQL pattern matching enables you to use _ to match any single character and % to match an arbitrary number of characters (including zero characters).

**Theory:**
SQL pattern matching allows you to search for patterns in data if you don't know the exact word or phrase you are seeking. This kind of SQL query uses wildcard characters to match a pattern, rather than specifying it exactly. For example, you can use the wildcard "C%" to match any string beginning with a capital C.
Using the LIKE Operator

To use a wildcard expression in an SQL query, use the LIKE operator in a WHERE clause, and enclose the pattern within single quotation marks.

Using the % Wildcard to Perform a Simple Search

To search for any employee in your database with a last name beginning with the letter C, use the following Transact-SQL statement:

```
SELECT *
FROM employees
WHERE last_name LIKE 'C%'
```

Omitting Patterns Using the NOT Keyword

Use the NOT keyword to select records that don't match the pattern. For example, this query returns all records whose name last does not begin with C:

```
SELECT *
FROM employees
WHERE last_name NOT LIKE 'C%'
```

Matching a Pattern Anywhere Using the % Wildcard Twice

Use two instances of the % wildcard to match a particular pattern anywhere. This example returns all records that contain a C anywhere in the last name:

```
SELECT *
FROM employees
WHERE last_name LIKE '%C%'
```

Finding a Pattern Match at a Specific Position

Use the _ wildcard to return data at a specific location. This example matches only if C occurs at the third position of the last name column:

```
SELECT *
 FROM employees
 WHERE last_name LIKE '_ _C%'
```

Consider following schema

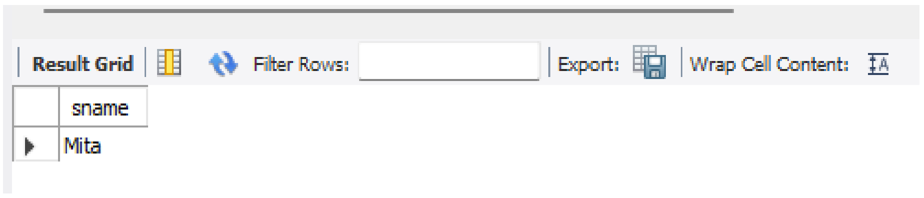
Student (sid,sname,city,age, Marks)
Department(did, dname, sid)

Solve Pattern Matching queries

1. Find sname of student having last letter as M
2. Find dname of student having second letter as O
3. Find sname of student having first letter as G
4. Find dname of student not having letter as A
5. Find dname of student having string as om.

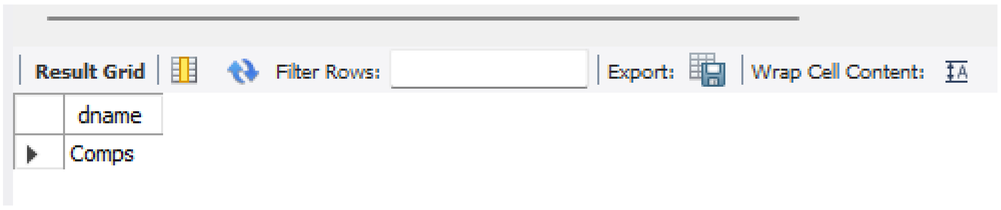
**Implementation:**

1.Find sname of student having last letter as M



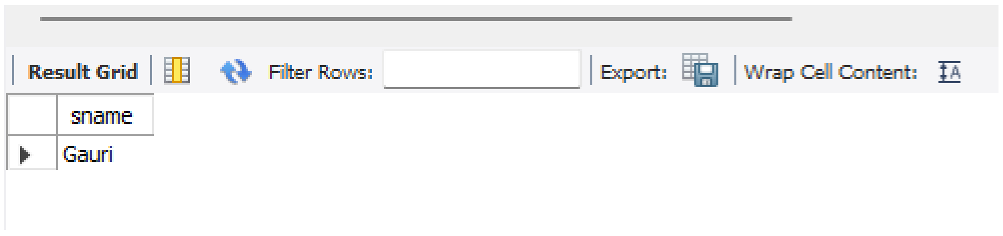
| sname |
|---|
| Mita |

2.Find dname of student having second letter as O



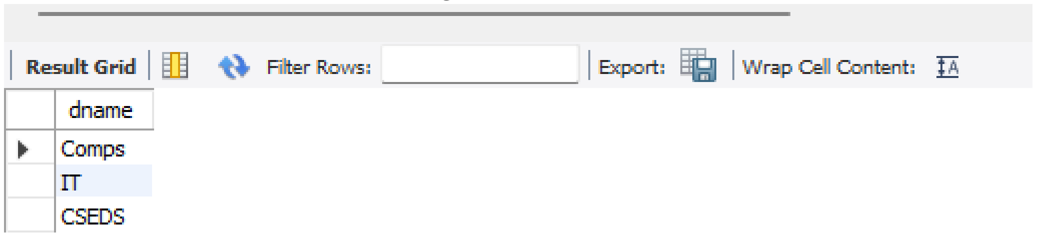
| dname |
|---|
| Comps |

3.Find sname of student having first letter as G


Result Grid | Filter Rows: | Export: | Wrap Cell Content:

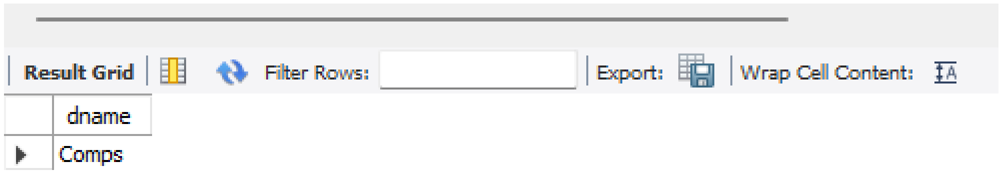
| sname |
|---|
| Gauri |

4.Find dname of student not having letter as A


Result Grid | Filter Rows: | Export: | Wrap Cell Content:

| dname |
|---|
| Comps |
| IT |
| CSEDS |

5.Find dname of student having string as om.


Result Grid | Filter Rows: | Export: | Wrap Cell Content:

| dname |
|---|
| Comps |

```
create table student2
(
sid int primary key,
sname char (10),
city char(10),
marks int(10),
age int (10)
);
insert into student2 values(1,'Tanvi','Mumbai',99,19);
insert into student2 values(2,'Vedika','Mumbai',99,18);
insert into student2 values(3,'Sanika','Pune',99,20);
insert into student2 values(4,'Mita','Pune',92,17);
insert into student2 values(5,'Gauri','Pune',94,17);
select sname from student2 where sname like'M%';
select dname from department2 where dname like'_o%';
select sname from student2 where sname like'G%';
select dname from department2 where dname not like'%A%';
select dname from department2 where dname like'%om%';
```

```
create table department2
(
did int primary key,
dname char (10),
floor int
);
insert into department2 values(1,'Comps',2);
insert into department2 values(2,'IT',2);
insert into department2 values(3,'CSEDS',1);
select * from department2;
```

**Conclusion:** In conclusion, SQL pattern matching provides a powerful tool for searching and filtering data within a database. By using the underscore (_) to match any single character and the percent symbol (%) to match an arbitrary number of characters, SQL developers can construct flexible and precise queries to retrieve the desired information. This capability enhances the versatility of SQL queries, allowing for more efficient data retrieval and manipulation.

| |
|---|
| Experiment No.7 |
| Nested queries and Complex queries |
| Date of Performance:27/02 |
| Date of Submission:05/03 |

**Aim**: Nested queries and Complex queries

**Objective:** In nested queries, a query is written inside a query. The result of inner query is used in execution of outer query

**Theory:**

**Sample table: Salesman**

| | | | |
|---|---|---|---|
| salesman_id | name | city | commission |

**Sample table: Orders**

| | | | | |
|---|---|---|---|---|
| ord_no | purch_amt | ord_date | customer_id | salesman_id |

Questions

1. Write a query to display all the orders from the orders table issued by the salesman 'Paul Adam'.

0. Write a query to display all the orders for the salesman who belongs to the city London.

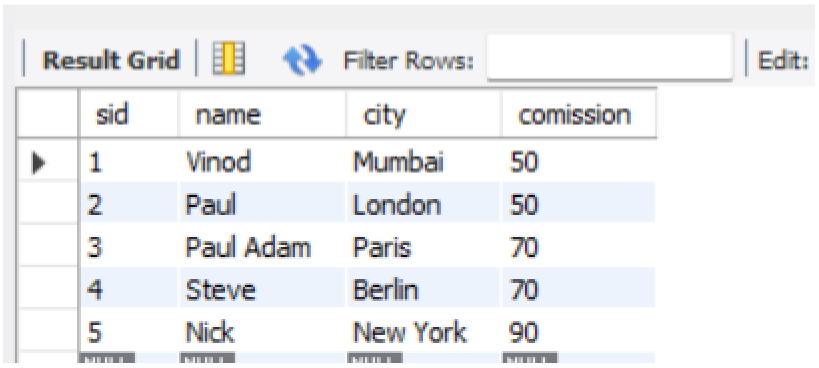
0. Write a query to find all the orders issued against the salesman who may works for customer whose id is 3007

0. Write a query to display all the orders which values are greater than the average order value for 10th October 2012

0. Write a query to find all orders attributed to a salesman in New york.

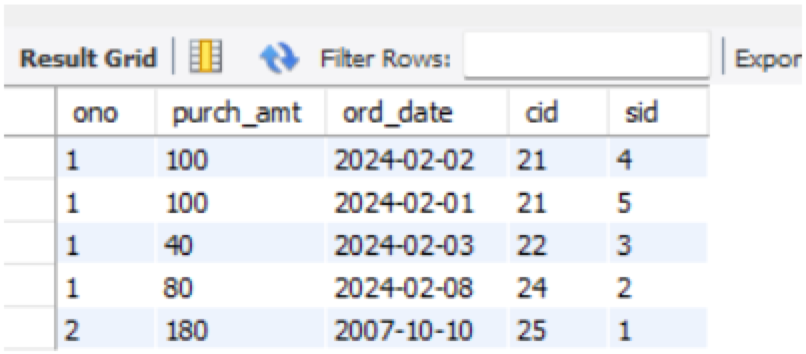
0. Write a query to display the commission of all the salesmen servicing customers in Paris

**Implementation:**



Result Grid | Filter Rows: | Edit:

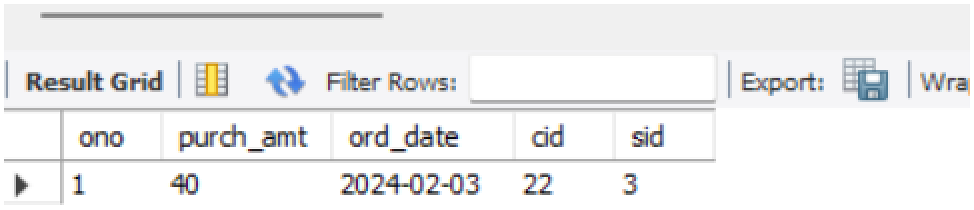
| sid | name | city | comission |
|---|---|---|---|
| 1 | Vinod | Mumbai | 50 |
| 2 | Paul | London | 50 |
| 3 | Paul Adam | Paris | 70 |
| 4 | Steve | Berlin | 70 |
| 5 | Nick | New York | 90 |



Result Grid | Filter Rows: | Expor

| ono | purch_amt | ord_date | cid | sid |
|---|---|---|---|---|
| 1 | 100 | 2024-02-02 | 21 | 4 |
| 1 | 100 | 2024-02-01 | 21 | 5 |
| 1 | 40 | 2024-02-03 | 22 | 3 |
| 1 | 80 | 2024-02-08 | 24 | 2 |
| 2 | 180 | 2007-10-10 | 25 | 1 |

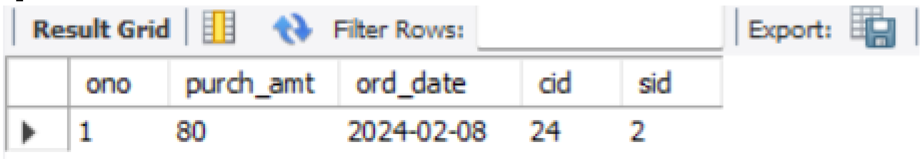
1]



Result Grid | Filter Rows: | Export: | Wra

| ono | purch_amt | ord_date | cid | sid |
|---|---|---|---|---|
| 1 | 40 | 2024-02-03 | 22 | 3 |

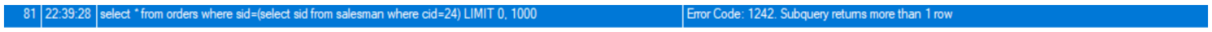
2]



Result Grid | Filter Rows: | Export:

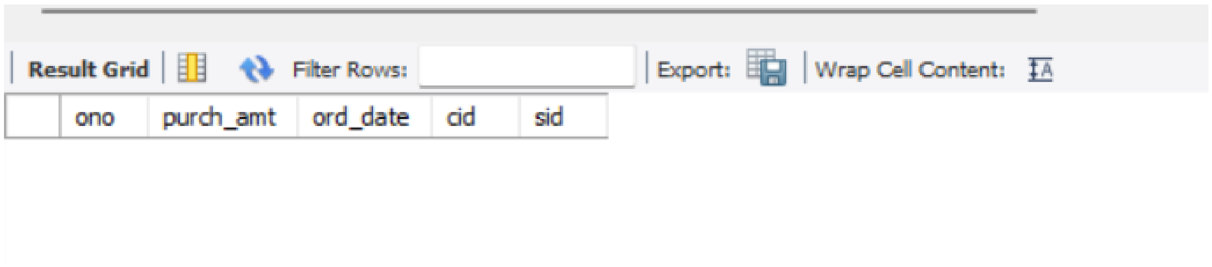
| ono | purch_amt | ord_date | cid | sid |
|---|---|---|---|---|
| 1 | 80 | 2024-02-08 | 24 | 2 |

3]



```
81 | 22:39:28 | select * from orders where sid=(select sid from salesman where cid=24) LIMIT 0, 1000 | Error Code: 1242. Subquery returns more than 1 row
```

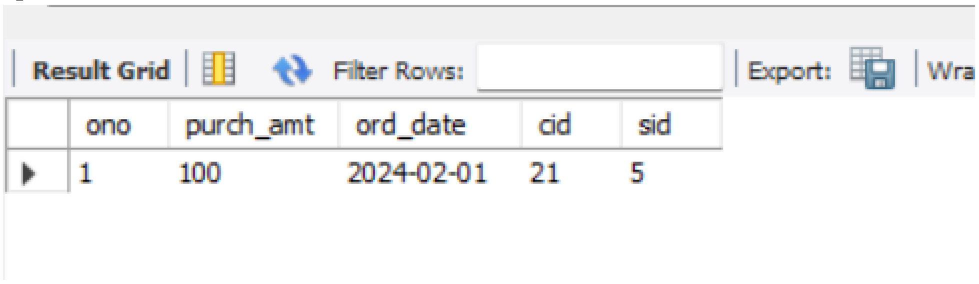
4]



Result Grid | Filter Rows: | Export: | Wrap Cell Content:

| | ono | purch_amt | ord_date | cid | sid |
|---|---|---|---|---|---|

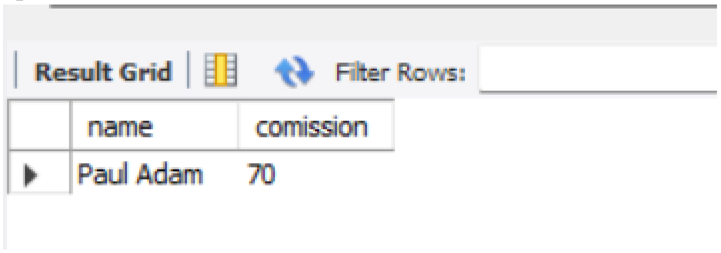
5]



Result Grid | Filter Rows: | Export: | Wra

| | ono | purch_amt | ord_date | cid | sid |
|---|---|---|---|---|---|
| ▶ | 1 | 100 | 2024-02-01 | 21 | 5 |

6]



Result Grid | Filter Rows:

| | name | comission |
|---|---|---|
| ▶ | Paul Adam | 70 |

```
create table salesman(
sid int Primary key,
name varchar(10),
city varchar(10),
comission int
);
create table orders(
ono int,
purch_amt int,
ord_date date,
cid int,
sid int,
foreign key (sid) references salesman(sid)
);
insert into salesman values(1,"Vinod","Mumbai",50);
insert into salesman values(2,"Paul","London",50);
insert into salesman values(3,"Paul Adam","Paris",70);
insert into salesman values(4,"Steve","Berlin",70);
insert into salesman values(5,"Nick","New York",90);
insert into orders values(1,90,'2024-02-02',20,1);
insert into orders values(1,100,'2024-02-02',21,4);
insert into orders values(1,100,'2024-02-01',21,5);
insert into orders values(1,40,'2024-02-03',22,3);
insert into orders values(1,80,'2024-02-08',24,2);
insert into orders values(2,60,'2007-10-10',25,1);
insert into orders values(3,70,'2023-3-14',26,3);
select * from salesman;
select * from orders;
select *
from orders
where sid=(select sid from salesman where name="Paul Adam");
select *
from orders
where sid=(select sid from salesman where city="London");
select *
from orders
where sid=(select sid from salesman where cid=24);
select *
from orders
where purch_amt=(select avg (purch_amt) from orders where ord_date='2012-10-10');
select *
from orders
where sid=(select sid from salesman where city="New York");
select s.name,s.comission
from salesman s
inner join orders o on s.sid=o.sid
where s.city='Paris';
```

**Conclusion:** In conclusion, the output of queries involving nested and complex SQL constructs can vary based on the specific requirements and conditions. These queries provide a powerful mechanism for extracting insights and transforming data, but careful consideration should be given to performance, accuracy, and data integrity. Testing and validation are essential to ensure that the output meets the desired criteria and aligns with the objectives of the analysis.

| |
|---|
| Experiment No.8 |
| Procedures and Functions |
| Date of Performance:05/03 |
| Date of Submission:19/03 |

**Aim**: To implement Functions and procedure.

**Objective:** The function must return a value but in Stored Procedure it is optional. Even a procedure can return zero or n values. Functions can have only input parameters for it whereas Procedures can have input or output parameters

**Theory:**

**Procedure:**

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

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

Where,

- *procedure-name* specifies the name of the procedure.
- [OR REPLACE] option allows the modification of an existing procedure.
- The optional parameter list contains name, mode and types of the parameters. **IN** represents the value that will be passed from outside and OUT represents the parameter that will be used to return a value outside of the procedure.
- *procedure-body* contains the executable part.

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

## 1. Creating a Function

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

```
CREATE [OR REPLACE] FUNCTION function_name
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
RETURN return_datatype
```

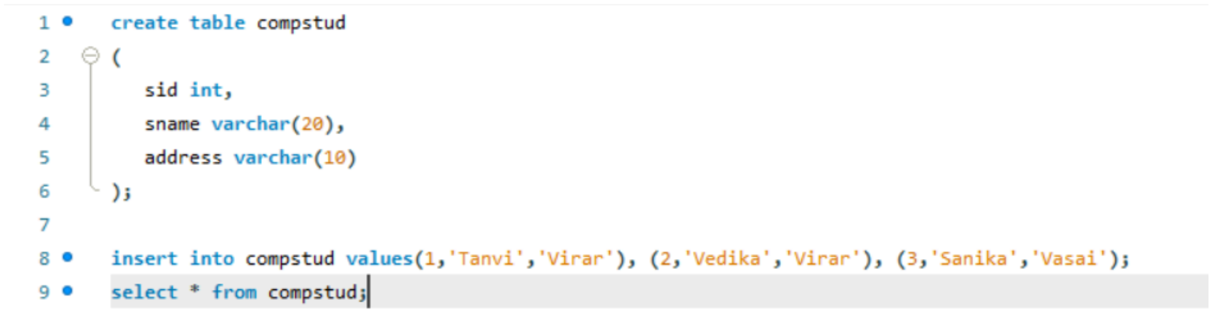
{IS | AS}
BEGIN
  < function_body >
END [function_name];

Where,

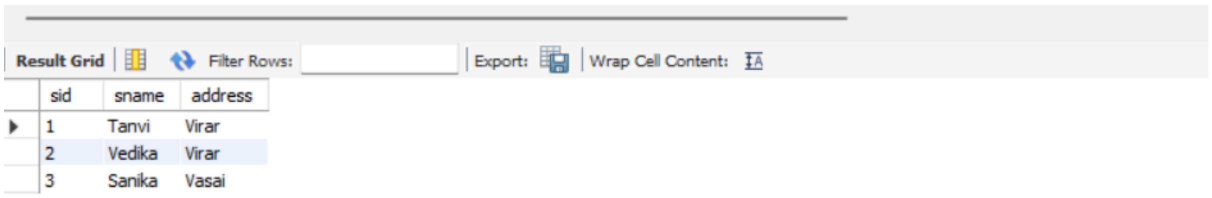
- *function-name* specifies the name of the function.
- [OR REPLACE] option allows the modification of an existing function.
- The optional parameter list contains name, mode and types of the parameters. IN represents the value that will be passed from outside and OUT represents the parameter that will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

**Implementation:**

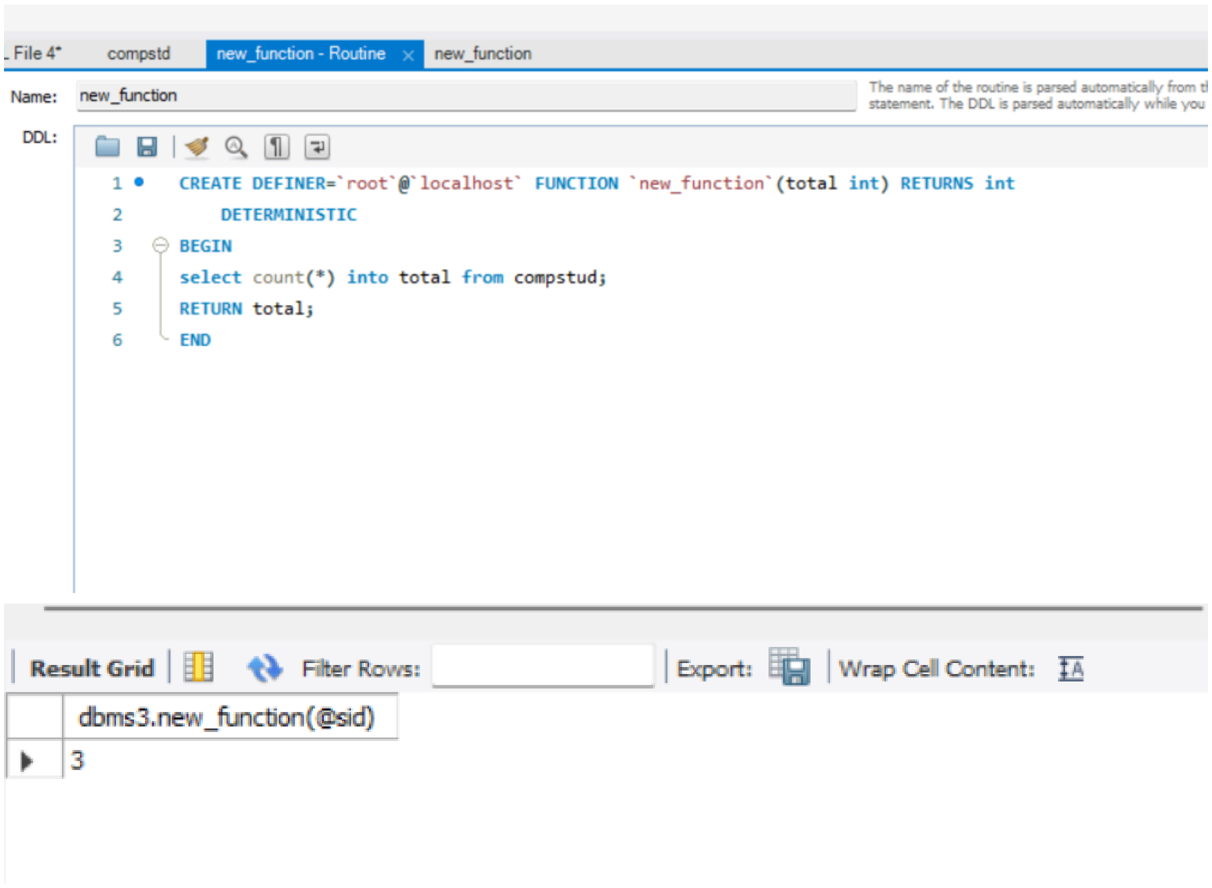
Functions



```
create table compstud
(
   sid int,
   sname varchar(20),
   address varchar(10)
);

insert into compstud values(1,'Tanvi','Virar'), (2,'Vedika','Virar'), (3,'Sanika','Vasai');
select * from compstud;
```



| sid | sname | address |
|---|---|---|
| 1 | Tanvi | Virar |
| 2 | Vedika | Virar |
| 3 | Sanika | Vasai |



- File 4* | compstd | new_function - Routine | new_function

Name: new_function

The name of the routine is parsed automatically from t
statement. The DDL is parsed automatically while you

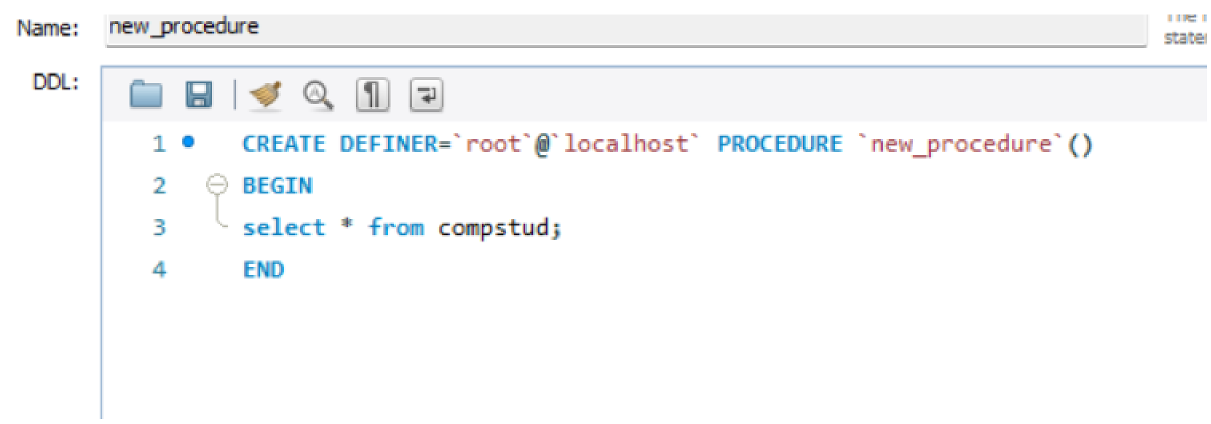
DDL:

```
CREATE DEFINER=`root`@`localhost` FUNCTION `new_function`(total int) RETURNS int
    DETERMINISTIC
BEGIN
select count(*) into total from compstud;
RETURN total;
END
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

```
dbms3.new_function(@sid)
3
```

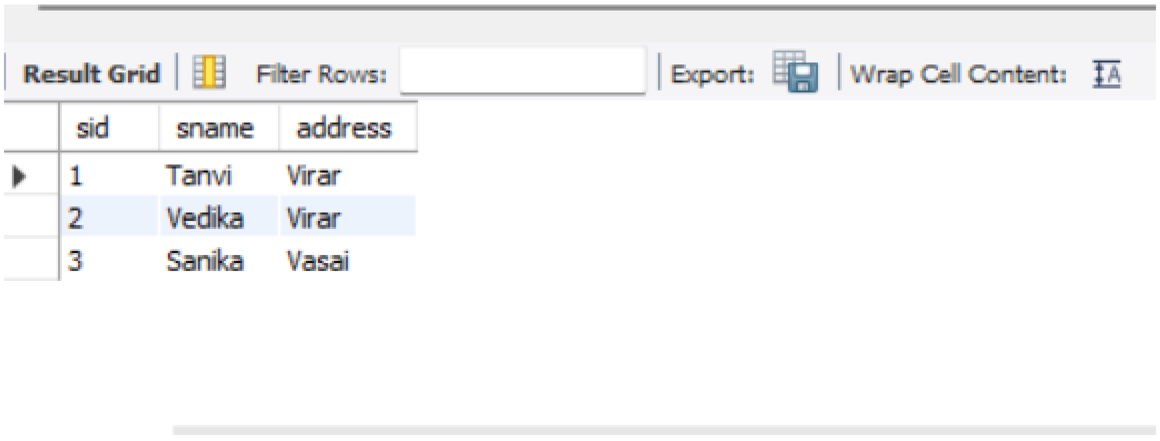
Procedure



Name: new_procedure

DDL:

```
CREATE DEFINER=`root`@`localhost` PROCEDURE `new_procedure`()
BEGIN
select * from compstud;
END
```



Result Grid | Filter Rows: | Export: | Wrap Cell Content:

```
sid  sname   address
1    Tanvi   Virar
2    Vedika  Virar
3    Sanika  Vasai
```

**Conclusion:** Implementing functions and procedures in a DBMS offers numerous benefits, including improved modularity, code reusability, enhanced performance, better security, encapsulation of business logic, effective transaction management, and ease of maintenance. Incorporating these database constructs into development practices can lead to more robust, scalable, and secure database applications.

| |
|---|
| Experiment No.9 |
| Views and Triggers |
| Date of Performance:19/03 |
| Date of Submission:26/03 |

**Aim**: Views and Triggers

**Objective:** Views can join and simplify multiple **tables** into a single virtual table A database trigger is procedural code that is automatically executed in response to certain events on a particular table or view in a database. The trigger is mostly used for maintaining the integrity of the information on the database. For example, when a new record (representing a new worker) is added to the employees table, new records should also be created in the tables of the taxes, vacations and salaries

**Theory:**

VIEWS

A view is nothing more than a SQL statement that is stored in the database with an associated name. A view is actually a composition of a table in the form of a predefined SQL query.

A view can contain all rows of a table or select rows from a table. A view can be created from one or many tables which depends on the written SQL query to create a view.

Views, which are a type of virtual tables allow users to do the following −

- Structure data in a way that users or classes of users find natural or intuitive.
- Restrict access to the data in such a way that a user can see and (sometimes) modify exactly what they need and no more.
- Summarize data from various tables which can be used to generate reports.

**Creating Views**

Database views are created using the **CREATE VIEW** statement. Views can be created from a single table, multiple tables or another view.

To create a view, a user must have the appropriate system privilege according to the specific implementation.

The basic **CREATE VIEW** syntax is as follows −

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

You can include multiple tables in your SELECT statement in a similar way as you use them in a normal SQL SELECT query.

TIGGERS:

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.

**Benefits of Triggers**
Triggers can be written for the following purposes −

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

**Creating Triggers**

The syntax for creating a trigger is −

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

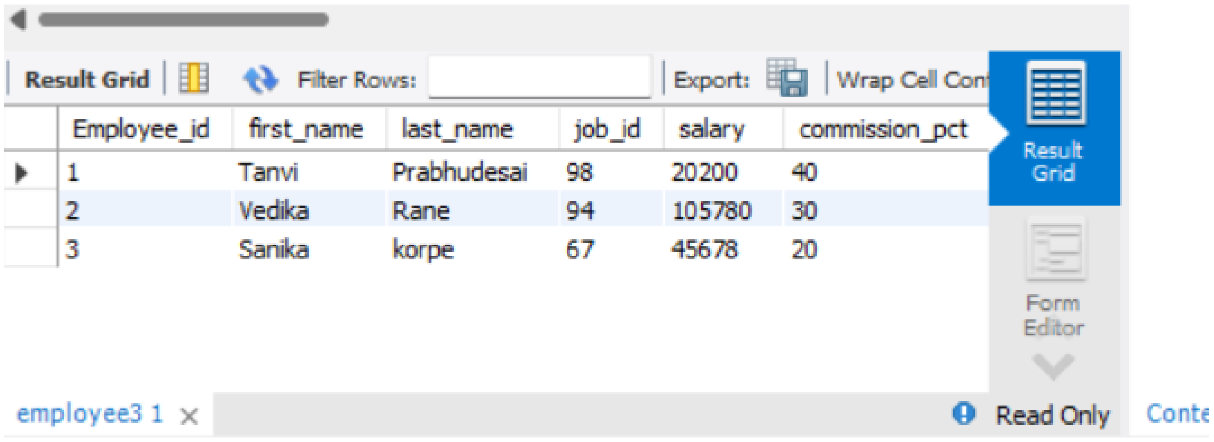
Where,

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

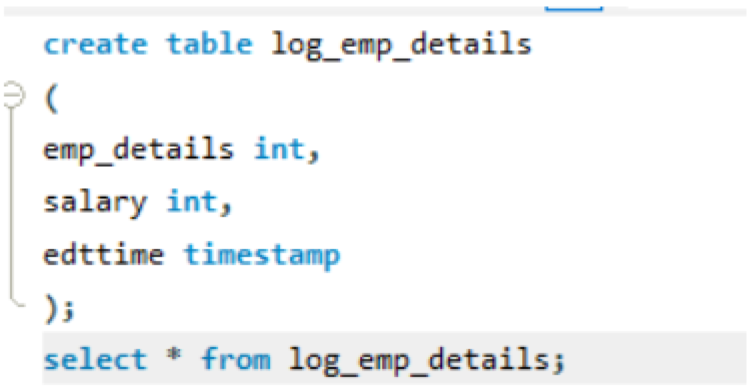
**Implementation:**

Triggers

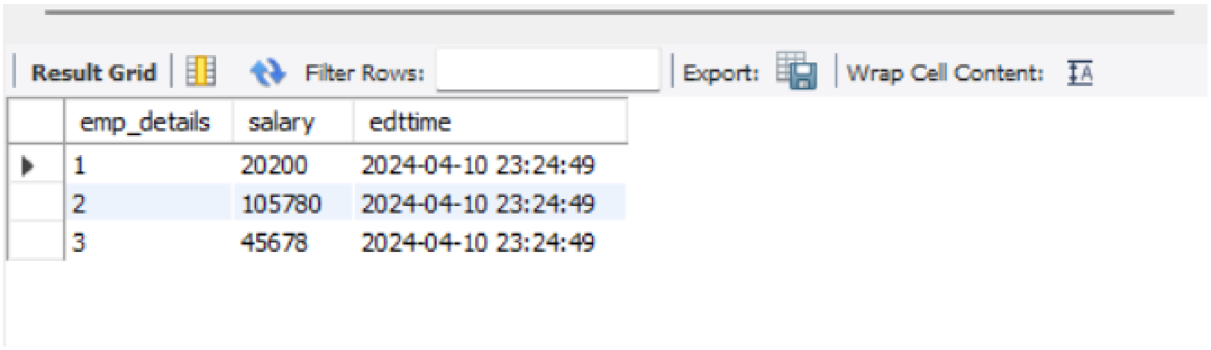
```
• create table employee3
  (
      Employee_id int NOT NULL,
      first_name varchar(35) NOT NULL,
       last_name varchar(35) ,
       job_id int,
       salary int,
      commission_pct int
  );

• insert into employee3 values
  (1,'Tanvi', 'Prabhudesai', 98,20200,40),
  (2,'Vedika', 'Rane', 94,105780,30),
  (3,'Sanika','korpe',67,45678,20);

• select * from employee3;

• select * from log_emp_details;
```



| Employee_id | first_name | last_name | job_id | salary | commission_pct |
|---|---|---|---|---|---|
| 1 | Tanvi | Prabhudesai | 98 | 20200 | 40 |
| 2 | Vedika | Rane | 94 | 105780 | 30 |
| 3 | Sanika | korpe | 67 | 45678 | 20 |

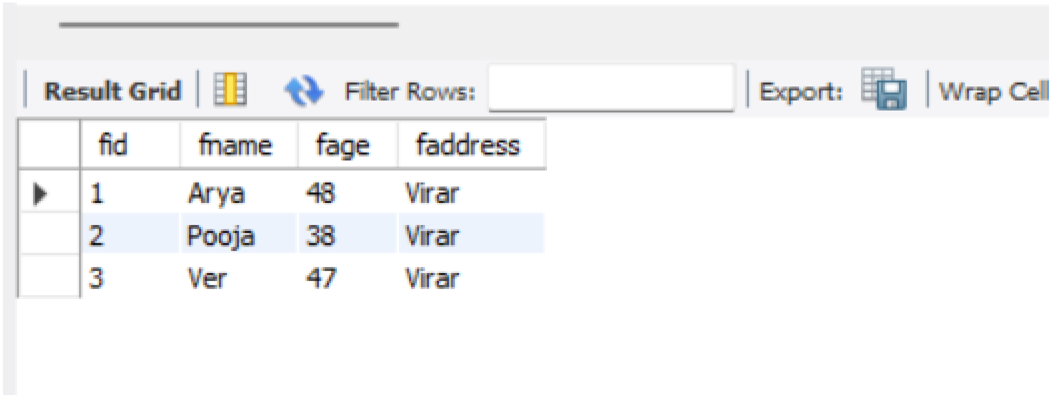


```
create table log_emp_details
(
emp_details int,
salary int,
edttime timestamp
);
select * from log_emp_details;
```

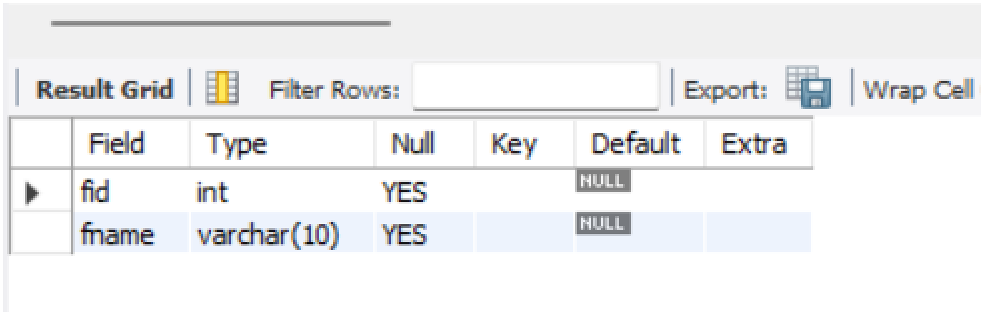


| emp_details | salary | edttime |
|---|---|---|
| 1 | 20200 | 2024-04-10 23:24:49 |
| 2 | 105780 | 2024-04-10 23:24:49 |
| 3 | 45678 | 2024-04-10 23:24:49 |

Views

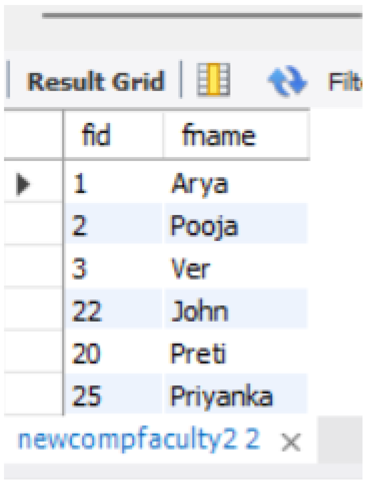


| fid | fname | fage | faddress |
|---|---|---|---|
| 1 | Arya | 48 | Virar |
| 2 | Pooja | 38 | Virar |
| 3 | Ver | 47 | Virar |



| Field | Type | Null | Key | Default | Extra |
|---|---|---|---|---|---|
| fid | int | YES |  | NULL |  |
| fname | varchar(10) | YES |  | NULL |  |

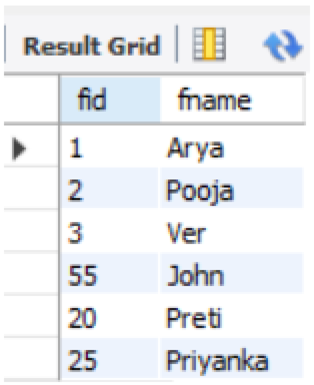
Insert



| fid | fname |
|---|---|
| 1 | Arya |
| 2 | Pooja |
| 3 | Ver |
| 22 | John |
| 20 | Preti |
| 25 | Priyanka |

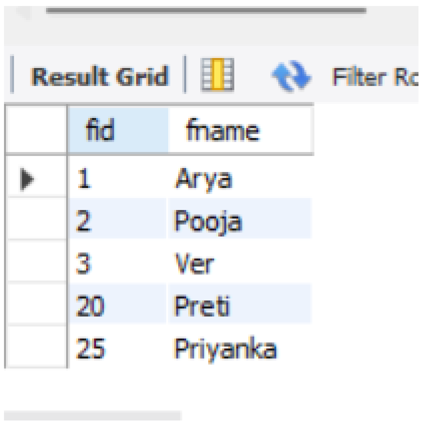
newcompfaculty2 2

Update



| fid | fname |
|---|---|
| 1 | Arya |
| 2 | Pooja |
| 3 | Ver |
| 55 | John |
| 20 | Preti |
| 25 | Priyanka |

Delete



| fid | fname |
|---|---|
| 1 | Arya |
| 2 | Pooja |
| 3 | Ver |
| 20 | Preti |
| 25 | Priyanka |

**Conclusion:**
The Views and Triggers experiment in database management systems (DBMS) offers valuable insights into enhancing database functionality and ensuring data integrity. Through this experiment, it is evident that views provide an efficient mechanism for presenting customized subsets of data to users, thereby simplifying complex queries and improving overall system performance.