PANIMALAR INSTITUTE OF TECHNOLOGY

(A CHRISTIAN MINORITY INSTITUTION)
JAISAKTHI EDUCATIONAL TRUST
BANGALORE TRUNK ROAD, VARADHARAJAPURAM
NASARATHPET, POONAMALLEE, CHENNAI 600 123

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND DATA SCIENCE

AD8411 – DATABASE DESIGN AND MANAGEMENT LABORATORY

ACADEMIC YEAR: 2021 – 22 (EVEN SEMESTER)

Name of the Student	:	
Register Number	:	
Roll Number	:	
Year & Semester	:	

PANIMALAR INSTITUTE OF TECHNOLOGY

REGISTER NUMBER:

Certified that this is a bonafide record of practical work done by

____

of II Year / IV Semester of B.Tech Artificial Intelligence and Data Science in AD8411 – DATABASE DESIGN AND MANAGEMENT LABORATORY during the academic year 2021 - 22.

STAFF IN-CHARGE **HEAD OF THE DEPARTMENT**

Submitted for the Anna University practical examination held on _____________ at Panimalar Institute of Technology, Chennai – 600 123.

INTERNAL EXAMINER **EXTERNAL EXAMINER**

TABLE OF CONTENTS

Ex. No.	Date	Name of the Experiment	Page No.	Marks	Staff Sign
Content Beyond Syllabus – Additional Experiments					

Ex. No. :	**DATABASE DEVELOPMENT LIFECYCLE**
Date:	

The different phases of database development life cycle (DDLC) in the Database Management System (DBMS) are explained below −

- Requirement analysis.
- Database design.
- Evaluation and selection.
- Logical database design.
- Physical database design.
- Implementation.
- Data loading.
- Testing and performance tuning.
- Operation.
- Maintenance.

REQUIREMENT ANALYSIS

- The most important step in implementing a database system is to find out what is needed i.e what type of a database is required for the business organization, daily volume of data, how much data needs to be stored in the master files etc.
- In order to collect all this information, a database analyst spends a lot of time within the business organization talking to people, end users and getting acquainted with the day-to-dayprocess.

DATABASE DESIGN

- In this phase the database designers will make a decision on the database model that perfectly suits the organization‘s requirement. The database designers will study the documents prepared by the analysis in the requirement analysis stage and then start development of a system model that fulfils the needs.

EVALUATION AND SELECTION

- In this phase, we evaluate the diverse database management systems and choose the one which perfectly suits the requirements of the organization.
- In order to identify the best performing database, end users should be involved.

LOGICAL DATABASE DESIGN

- Once the evaluation and selection phase is completed successfully, the next step is logical database design.
- This design is translated into internal model which includes mapping of all objects i.e designof tables, indexes, views, transaction, access privileges etc.,

PHYSICAL DATABASE DESIGN

- This phase selects and characterizes the data storage and data access of the database.
- The data storage depends on the type of devices supported by the hardware, the data accessmethods.
- Physical design is very vital because of bad design which results in poor performance.

IMPLEMENTATION

- Database implementation needs the formation of special storage related constructs.
- These constructs consist of storage groups, table spaces, data files, tables etc.

DATA LOADING

- Once the database has been created, the data must be loaded into the database.
- The data required to be converted, if the loaded date is in a different format.

OPERATIONS

- In this phase, the database is accessed by the end users and application programs.
- This stage includes adding of new data, modifying existing data and deletion of absolutedata.
- This phase provides useful information and helps management to make a business decision.

MAINTENANCE

- It is one of the ongoing phases in DDLC.
- The major tasks included are database backup and recovery, access management, hardwaremaintenance etc.

ADVANTAGES OF DATABASE DEVELOPMENT

There are innumerable benefits of the database system for a business. Using a database for gathering, storing, and managing the data gives your business a distinct advantage over your competitors. Here are a few advantages of database development for your business.

- Data redundancy is prevented in databases. Any changes are reflected immediately thus,zero chance of encountering data duplication.
- The database makes the sharing of data among its users more streamlined. Also, there arevarious authorization levels to access and share data.
- The data stored in the database stay accurate and consistent which helps maintain the integrity of data.
- Unauthorized users are not allowed to access the data from the database. Only authorizedusers with valid user credentials can access the data.
- Databases have different levels of access which allows specific users or groups of users toaccess only specific types of data.

- The database management system (DBMS) has in-built backup and recovery. Thus, no needfor periodic backup of the database.
- Data restoration is possible and DBMS restores the database to its previous condition aftercrash or failure.
- Because of no data redundancy, all the data in the database appear consistently for all usershaving access to that database.

ORACLE :

Oracle is an Object-Relational Database Management System.

ORACLE DATABASE :

Every Oracle Database Contains Logical and Physical Structures. Logical Structures are table spaces, Schema objects, extents and segments. Physical Structures are Datafiles, Redo Log Files, Control File.

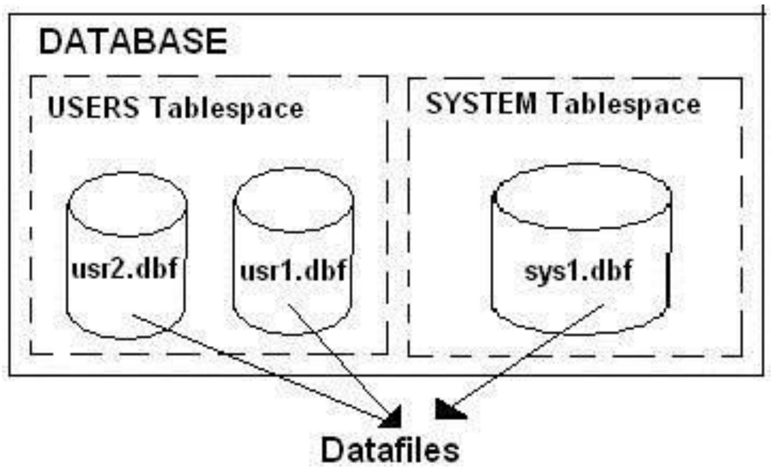


Table :

✓ A table is the data structure that holds data in a relational database. A table is composed ofrows and columns.

All the tables and other objects in Oracle are stored in tablespace logically, but physically they arestored in the data files associated with the tablespace.

CODD'S RULES :

1. **Information Rule**: All information in a relational database including table names,column names are represented by values in tables.
2. **Guaranteed Access Rule**: Every piece of data in a relational database, can be accessed by using combination of a table name, a primary key value that identifies the row and column name which identified a cell.
3. **Systematic Treatment of Nulls Rule**: The RDBMS handles records that have unknown or inapplicable values in a pre-defined fashion.
4. **Active On-line catalog based on the relational model**: The description of a database and in its contents are database tables and therefore can be queried on-line via the data manipulation language.

5. **Comprehensive Data Sub-language Rule**: A RDBMS may support several languages. But at least one of them should allow user to do all of the following: define tables and views, query and update the data, set integrity constraints, set authorizations and define transactions.
6. **View Updating Rule**: Any view that is theoretically updateable can be updated using the RDBMS.
7. **High-Level Insert, Update and Delete**: The RDBMS supports insertions, updation and deletion at a table level.
8. **Physical Data Independence**: The execution of adhoc requests and application programs is not affected by changes in the physical data access and storage methods.
9. **Logical Data Independence**: Logical changes in tables and views such adding/deleting columns or changing fields lengths need not necessitate modifications in the programs orin the format of adhoc requests.
10. **Integrity Independence**: Like table/view definition, integrity constraints are stored in the on-line catalog and can therefore be changed without necessitating changes in the application programs.
11. **Distribution Independence**: Application programs and adhoc requests are not affected by change in the distribution of physical data.
12. **No subversion Rule**: If the RDBMS has a language that accesses the information of a record at a time, this language should not be used to bypass the integrity constraints. This is necessary for data integrity.

SQL (Structured Query Language) :

SQL (Structured Query Language) is a database computer language designed for the retrieval and management of data in relational database management systems (RDBMS), database schema creation and modification, and database object access control management. SQL is a standard supported by all the popular relational database management systems in the market place.
The basis data structure in RDBMS is a ***table***. SQL provides you the features to define tables, define constraints on tables, query for data in table, and change the data in table by adding, modifying, and removing data. SQL also supports grouping of data in multiple rows, combining tables and other features. All these put together, SQL is a high-level query language standard to access and alter data in RDBMS.
Various Data Types :

1. **Character Datatypes:**

✓ Char – fixed length character string that can varies between 1-2000 bytes
✓ Varchar / Varchar2 – variable length character string, size ranges from 1-4000 bytes.
✓ Long - variable length character string, maximum size is 2 GB

2. **Number Datatypes :**

✓ Number – {p=38,s=0}
✓ Number(p) - fixed point
✓ Number(p,s) –floating point (p=1 to 38,s= -84 to 127)

3. **Date Datatype: used to store date and time in the table.**

✓ DB uses its own format of storing in fixed length of 7 bytes for century, date, month, year, hour, minutes, and seconds.
✓ Default data type is ―dd-mon-yy‖

4. **Raw Datatype: used to store byte oriented data like binary data and byte string.**

5. **Other :**

✓ CLOB – stores character object with single byte character.
✓ BLOB – stores large binary objects such as graphics, video, sounds.
✓ BFILE – stores file pointers to the LOB's.

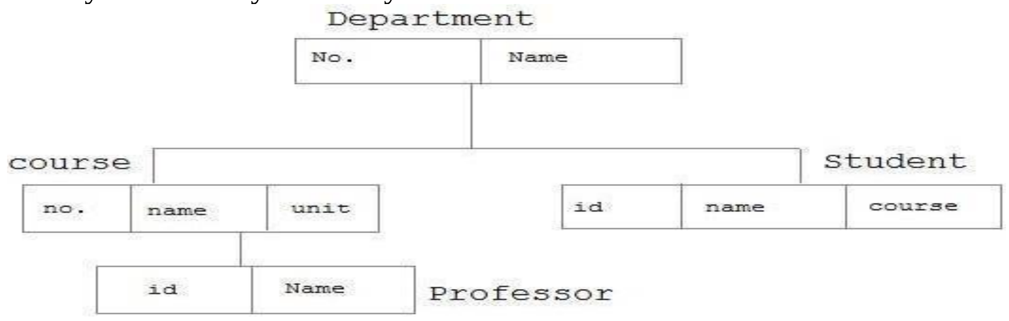
Database Model

A Database model defines the logical design of data. The model describes the relationships between different parts of the data. Historically, in database design, three models are commonly used. They are,

- Hierarchical Model
- Network Model
- Relational Model

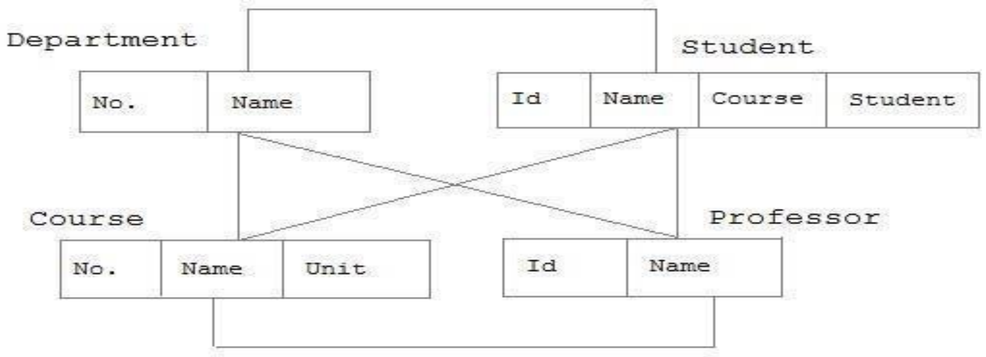
Hierarchical Model

In this model each entity has only one parent but can have several children . At the top ofhierarchy there is only one entity which is called **Root**.



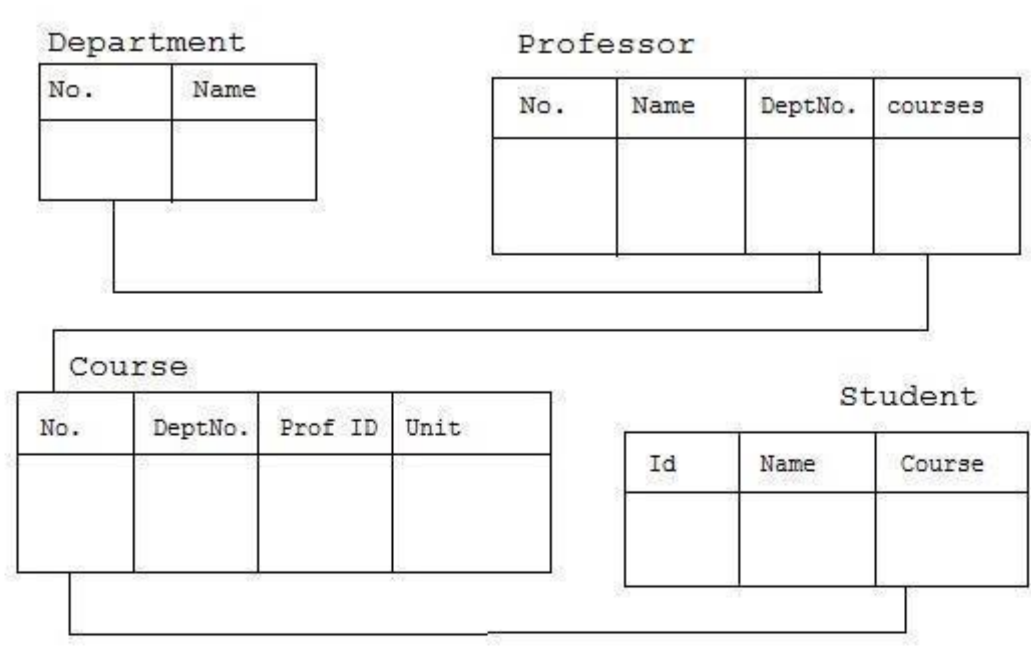
Network Model

In the network model, entities are organized in a graph, in which some entities can be accessed through several path



Relational Model

In this model, data is organized in two-dimensional tables called **relations**. The tables orrelation are related to each other.



SQL LANGUAGE STATEMENTS:

1. Data Definition Language (DDL) Commands In RDBMS.

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

The commands used are:

- ✓ **CREATE** – Create a new Table, database, schema
- ✓ **ALTER** – Alter existing table, column description
- ✓ **DROP** – Delete existing objects from database
- ✓ **Truncate** – Delete all the records from table .
- ✓ **Rename** - To rename a table

The Create Table Command :

The create table command defines each column of the table uniquely. Each column has minimum of three attributes.

- Name
- Data type

☐ Size(column width).

For example ,

Table Name : Student		
Column name	**Data type**	**Size**
Reg_no	varchar2	10
Name	char	30
DOB	date	
Address	varchar2	50

Command :

```
CREATE TABLE Student (Reg_no varchar2(10) ,Name char(30), DOB date, Address
varchar2(50));
```

Data Constraint :

✓ Oracle allows constraints for attaching in the table columns via SQL syntax that checksdata for integrity.

✓ The constraints is a rule which works into the table, it disallows the that data which are notfollowing the rule of data constraints.

INTEGRITY CONSTRAINT

An integrity constraint is a mechanism used by oracle to prevent invalid data entry into thetable. It has enforcing the rules for the columns in a table. The types of the integrity constraints are:

a) Domain Integrity - Not Null , Check

b) Entity Integrity - Unique , Primary Key

c) Referential Integrity - Foreign Key , References Key

General Types of Data Constraints:Input/output Constraints

1. The Primary key Constraint
2. The Foreign key Constraint

Business Rule Constraints

1. Check Constraint
2. Unique Constraint
3. NOT NULL Constraint

Primary key :

- A primary key is one or more column(s) in a table used to uniquely identify each row in thetable. A primary key column in a table has special attributes :
- A primary key is the combination of (NOT NULL+UNIQUE) key constraints.
- A single column primary key is called *Simple key.* And a multicolumn primary key is calleda *Composite primary key*.

PRIMARY KEY constraint

defined at the column level :

Syntax :

<col_name> <datatype>(<size>) PRIMARY KEY

For example ,

```
CREATE TABLE Sales_Order(Order_no varchar2(6) PRIMARY KEY, Order_Date date,
Client_no varchar2(6), Dely_addr varchar2(25), Order_status
```

PRIMARY KEY Constraint defined at

the table level:Syntax:

Primary key (<col_name,<col_name>,)

Example:

```
CREATE TABLE sales_order_details (Detlorder_no varchar2(6), Product_no varchar2(7),
product_rate number(8,2),primary key (Detlorder_no, Product_no));
```

Foreign key :

Foreign keys represents the relationships between tables. A foreign key is a column(or groupof columns) whose values are derived from the primary or unique key of the some other tables.

The table in which the foreign key is defined is called a ***foreign table or detail table***. The table that defines the primary key or unique key and is referenced by the foreign key is called ***primary table or master table***.

Syntax:

<col_name> <datatype> (<size>) REFERENCES <table_name> [(<col_name>)]

Example:

```
Create table sales_order_details(Detlorder_no varchar2(6), product_no varchar2(6),
product_rate number(8,2) ,foreign key(detlorder_no) references sales_order.order_no);
```

Principal of Foreign Key/References:

- Rejects an INSERT or UPDATE of a value, if a corresponding value does not exist currently in the **master key** table or you can say Primary key table from where foreign key has brought out.
- If ON DELETE CASCADE option is set, a DELETE operation in the master table will trigger a delete operation for corresponding records in all detail table.
- Rejects a DELETE for the master table if corresponding records in the DETAIL table exist or you can say if value exist in foreign table and you want to DELETE it from Primary key table then it will rejected, First you have to DELETE from FOREIGN KEY TABLE then PRIMARY KEY table.Foreign key table also called **Detail Table.**
- It must be matching Data Type for both Primary key and Foreign key attribute.

The CHECK constraint:

CHECK constraint must be specified as a logical expression that evaluates either to TRUE or FALSE

Syntax:

<col_name> <data_Type> (<size>) CHECK (<logical expression>)

Example :

```
CREATE TABLE Client_Master(client_no varchar2(6) CHECK (client_no like‘c%‘),
Name varchar2(20) CHECK (name=upper(name)), city varchar2(15) CHECK (city
N(_Allahabad‘,‘Lucknow‘)));
```

Unique key :

The Unique key disallows the similar value in same column of a table.

Syntax :

<col_name>

<data_type>(<size>)UNIQU

EFor Example,

```
CREATE TABLE Student (Reg_no varchar2(10)  PRIMARY  KEY,Name varchar2(30),
Phone number(10) UNIQUE, Address varchar2(50));
```

NULL Value :

A NULL value is different from a blank or a zero.

NOT NULL Constraint

When a column is defined as NOT NULL , then that column becomes a mandatory column.it must be enter a value into that column.

For Example,

```
CREATE TABLE Student(Reg_no varchar2(10) PRIMARY KEY, Name varchar2(30) NOT
NULL, Phone number(10) UNIQUE,Address varchar2(50)NOT NULL , DOB date);
```

Creating a Database:

create database database-name;

Example,

create database Test;

ALTER TABLE Command:

ALTER TABLE Command we can **add** (or) **modify** field of our exiting table.

Syntax:

alter table tablename add/modify (attribute datatype(size));Adding New Columns:

Syntax:

ALTER TABLE <table_name> ADD (<NewColumnName> <Data_Type>(<size>), ...n);

Example: ALTER TABLE Student ADD (Age number(2), Marks number(3));

DROPPING A COLUMN from the Table:

Syntax:

ALTER TABLE <table_name> DROP COLUMN <column_name>

Example:

ALTER TABLE Student DROP COLUMN Age;

Modifying Existing Table :

Syntax:

ALTER TABLE <table_name> MODIFY (<column_name>

<NewDataType>(<NewSize>))

Example:

```
ALTER TABLE Student MODIFY (Name Varchar2(40));
```

Restriction on the ALTER TABLE :

Using the ALTER TABLE clause the following tasks cannot be performed.

- Change the name of the table
- Change the name of the column
- Decrease the size of a column if table data exists

Example:

1. Alter table emp add (phone_no char (20));
2. Alter table emp modify(phone_no number (10));
3. ALTER TABLE EMP ADD CONSTRAINT Pkey1 PRIMARY KEY (EmpNo);

DROP TABLE:

It will destroy the table and all data which will be recorded in it.

Syntax:

DROP TABLE <table_name>

Example: DROP TABLE Student;

TRUNCATE TABLE

Truncate statement to delete all the rows from table permanently . But this command will notdestroy the table's structure.

Syntax:
TRUNCATE TABLE <Table_name>
Example:

```
TRUNCATE TABLE Student;
```

RENAME query :

The rename command is used to rename a table.

Syntax,

rename table old-table-name to new-table-name

For Example ,

```
rename table Student to Student-record;
```

DESC

This is command used to view the structure of the table.

Example:

```
SQL > desc emp;
```

Name	**Null?**	**Type**
EmpNo	NOT NULL	number(5)
EName		VarChar(15)
Job	NOT NULL	Char(10)
DeptNo	NOT NULL	number(3)
PHONE_NO		number (10)

DML command

Data Manipulation Language (DML) statements are used for managing data in database. DML commands are not auto-committed. It means changes made by DML command are not permanent to database, it can be rolled back.

- ✓ INSERT : Use to Add Rows to existing table.
- ✓ UPDATE : Use to Edit Existing Rows in tables.
- ✓ DELETE : Use to Delete Rows from tables.

INSERT command

Insert command is used to insert data into a table.

INSERT into table-name values(data1,data2,..)

For example ,

```
// Insert all field values
INSERT into Student values(102,'Alex',null);
// Insert particular Field values
INSERT into Student(id,name) values(102,'Alex');
```

Inserting data into a table from another table:

Example :

Suppose you want to insert data from course table to university table then use this example:

```
Ex.: 1   INSERT INTO  university SELECT course_id, course_name FROM course;

Ex.: 2   Insert into emp (empno, ename, sal) select empno, ename, sal from old_emp;
```

UPDATE command:

Update command is used to update a row of a table.

Syntax:

UPDATE table-name set column-name = value where condition;Example,

```
Eg : 1 update Student set age=18 where s_id=102;

Eg : 2 update emp set sal=sal+500 where empno = 104;

Eg : 3 update emp set sal=sal+(sal*5/100);

Eg : 4 update student set total=maths+phy+chem, average=(maths+phy+chem)/3;

// update the city field value in emp table from same city field value in dept table which match bothdeptno is same
Eg. : 5         update emp set city=(select city from dept where deptno= emp.deptno);
```

Rollno	Name	Maths	Phy	Chem	Total	Average
101 102 103	Sami Scott Smith	99 34 45	90 77 82	89 56 43		

```
//Salary will be automatically increased by 10% of salary.
Eg.:6 UPDATE employee SET salary= salary+salary*0.1 WHERE employee_id='E101';
```

Update multiple columns:

UPDATE Student set s_name='Abhi',age=17 where s_id=103;

Delete command:

Delete command is used to delete data from a table. Delete command can also be used with condition to delete a particular row.

Syntax:

DELETE from table-name; Delete all Records from a Table:

```
DELETE from Student;
```

To Delete a particular Record from a Table:

```
DELETE from Student where  s_id=103;
```

Select (Data Query) Statement :

Use a SELECT statement or subquery to retrieve data from one or more tables, object tables,views, object views, or materialized views

Eg. :1 select empno, ename, sal from emp; // to retrieve all rows from emp table.Eg. :2 select * from emp; // * means select all columns
Eg. :3 select * from emp where sal > 5000; // Retrieve all values those who are all sal > 5000

// Using Logical Conditions

Eg. :4 select * from emp where not (sal **IS** NULL);
Eg. :5 select * from emp where not (salary **BETWEEN** 1000 **AND** 2000);
Eg. :6 select * from employees where ename ='SAMI' **AND** sal=3000;

// Using Membership conditions

Eg. :8 select * from emp where deptno **IN** (10,20);
Eg. :9 select * from emp where deptno **in** (select deptno from dept where city=‘hyd‘)
Eg. :10 select * from emp where ename **not in** ('scott', 'smith');
Eg. :11 select ename from emp where deptno is **null**;

// Using EXISTS Conditions

In EXISTS condition tests for existence of rows in a subquery.

Eg. :12 Select deptno from dept d where **exists** (select * from emp e where d.deptno = e.deptno);

// Using LIKE Conditions
The LIKE conditions specify a test involving pattern matching.
Note :

- The percent sign (%) matches any string.
- The underscore (_) matches any single character

Eg. :13 Select * from emp where ename **like _S%‘** ; // whose name starts with "S"
Eg. :14 select * from emp where ename **like „%d‟**; // whose name ends with ―d‖
// whose name starts with _A‘ and ends with _d‘
Eg. :15 select * from emp where ename **like _A%d‟**;
Eg. :16 select * from emp where ename like **„%a%‘**; // whose name contains character _a‘ anywhere in the string.
// List whose name contains _a‘ in second position
Eg. :17 select * from emp where ename **like „_a%‘**;
// List whose name contains _a‘ as last second character
Eg. :18 select * from emp where ename **like „%a_‘**;

Aggregate Functions :

Aggregate functions return a single value based on groups of rows, rather than single valuefor each row.

The important Aggregate functions are :

1. Avg 2. Sum 3.Max 4. Min 5.Count 6. Stddev 7. Variance

Eg. : 19 Select **avg**(sal) –Average Salary‖ from emp;

Eg. : 20 Select **sum**(sal) –Total Salary‖ from emp;

Eg. : 21 Select **max**(sal) –Max Salary‖ from emp;

Eg. : 22 Select **min**(sal) –Min Salary‖ from emp;

Eg. : 23 Select **count**(*) from emp; // return no. of rows in table

// Counts the number of employees whose salary is not null.

Eg. :24 Select **count**(sal) from emp;

Eg. :25 Select **stddev**(sal) from emp;

Eg. :26 Select **variance**(sal) from emp;For example ,

SELECT deptname, **MAX(salary)** AS "Highest salary" FROM employees **GROUP BY** deptname;

// To display emp name with the minimum salary from each department

SELECT ename, department, **MIN(salary)** AS "Lowest salary" FROM employees **GROUP BY** department;

HAVING Clause: (Must have atleast One Aggregate function + GROUP BY Clause)

The Oracle HAVING clause is used in combination with the GROUP BY clause to restrictthe groups of returned rows.

For example,

// To display departments with sales greater than $25,000 will be returned.

SQL > SELECT department, SUM(sales) AS "Total sales" FROM order_details GROUP BY department HAVING SUM(sales) > 25000;

// To display number of employees in departments with salary less than $50,000 and havingmore than 10 employee will be returned.

SQL > SELECT department, COUNT(*) AS "Number of employees" FROM employees WHEREsalary < 50000 GROUP BY department HAVING COUNT(*) > 10;

Sorting DATA:

The Rows retrieved from the table will be sorted in either **Ascending** or **Descending** orderdepending on the condition specified in select statement, the Keyword has used **ORDER BY.**

```
Eg. :27        SELECT * FROM Student ORDER BY First_Name ASC;
Eg. :28        SELECT first_name, city,pincode FROM Student ORDER BY First_name
DESC;
//     Sorting By Multiple Columns
SELECT first_name, city,pincode FROM Student ORDER BY First_name, city DESC;
// The TOP clause allows us to specify how many rows to
return.SQL > SELECT TOP 3 * FROM emp;
SQL > SELECT TOP 40 PERCENT * FROM emp;
SQL > SELECT TOP 40 PERCENT * FROM emp ORDER BY  2  DESC;  // 2- Second
field
```

Eliminating Duplicates:(to find unique values in a column**)**

A table could hold duplicate rows. In such a case, you can eliminate duplicates.

Syntax:

SELECT DISTINCT col, col, .., FROM table_name;

```
Eg. :29 SELECT DISTINCT * FROM Student;
Eg. :30 SELECT DISTINCT first_name, city, pincode FROM Student;
```

Date Functions and Operators:

To manipulate the Date values.

```
Eg. :31 select sysdate from dual;
```

Sample Output :

```
SYSDATE
    --------
7-MAR-22
```

```
Eg. 32: alter session set NLS_DATE_FORMAT=‘DD-MON-YYYY HH:MIpm‘;

Eg. 33: select sysdate from dual;
```

Sample Output :

```
SYSDATE
    ---------------------
7-MAR-2022 02:05pm
```

Note : The default setting of NLS_DATE_FORMAT is DD-MON-YY

```
Eg. 34: SELECT SESSIONTIMEZONE, CURRENT_DATE FROM DUAL;
```

Sample Output :

```
SESSION TIME ZONE        CURRENT_DATE
      -----------------          -----------------------

+05:30                   7-MAR-2022 14:15:03
```

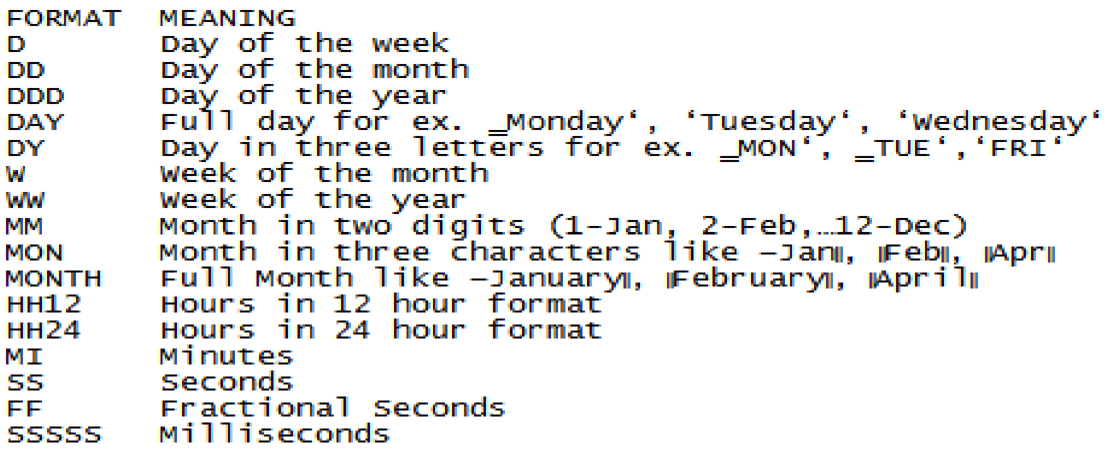
DATE FORMAT MODELS :

To translate the date into a different format string you can use TO_CHAR function withdate format.

```
Eg. 35:Select to_char(sysdate,‘DAY‘)‖Today‖ FROM DUAL;
```

Sample Output :

TODAY MONDAY



FORMAT	MEANING
D	Day of the week
DD	Day of the month
DDD	Day of the year
DAY	Full day for ex. ‗Monday‘, ‘Tuesday‘, ‘Wednesday‘
DY	Day in three letters for ex. ‗MON‘, ‗TUE‘,‘FRI‘
W	Week of the month
WW	Week of the year
MM	Month in two digits (1-Jan, 2-Feb,…12-Dec)
MON	Month in three characters like ―Jan‖, ‖Feb‖, ‖Apr‖
MONTH	Full Month like ―January‖, ‖February‖, ‖April‖
HH12	Hours in 12 hour format
HH24	Hours in 24 hour format
MI	Minutes
SS	Seconds
FF	Fractional Seconds
SSSSS	Milliseconds

```
Eg. 36: select to_char(sysdate,‘Day, ddth Month, yyyy‘)‖Today‖ from dual;

Eg. 37: select to_char(hire_date,‘Day, ddth Month, yyyy‘) from emp;
```

TO_DATE function :

To_Date function is used to convert strings into date values.

```
Eg. 38: select to_char(to_date (‘15-aug-1947‘,‘dd-mon-yyyy‘),‘Day‘)   from dual;

Eg. 39: select sysdate+45 from dual;
```

ADD_MONTHS :

```
Eg. 40:          Select ADD_MONTHS(SYSDATE,6) from dual;
```

MONTHS_BETWEEN:

```
Eg. 41:          select months_between(sysdate,to_date(‘15-aug-1947‘)) from dual;
            // To eliminate the decimal value use truncate function
Eg. 42:          select trunc(months_between(sysdate,to_date(‘15-aug-1947‘)))  from
dual;
```

LAST_DAY :

Eg. 43: select LAST_DAY(sysdate) from dual; // last date of the month of a given date

NEXT_DAY :

Eg. 44: select next_day(sysdate) from dual;

EXTRACT :

Syntax :

EXTRACT (YEAR / MONTH / WEEK / DAY / HOUR / MINUTE / TIMEZONE FROM DATE)

Eg. 45: select extract(year from sysdate) from dual;

Character Functions:

Eg. 46: select **LOWER**(‗SAMI‘) from dual; // Return lower case string

Eg. 47: select **UPPER**(‗Sami‘) from dual; // Return lower case string

Eg. 48: select **INITCAP**(‗med sami‘) from dual; // Initial letter in capital.

Eg. 49: select **length**(‗mohammed sami‘) from dual; // Return length of the string

Eg. 50: select **substr**('mohammed sami',10,3) from dual; // Return Substring 3 character start with 10 position
Eg. 51 :select **replace**('hell mohd kfshan','mohd','mohammed') from dual;
Eg. 52 :select **rpad**(ename,'*',10) from emp;
Eg. 53 :select **lpad**(ename,'*',10) from emp;
Eg. 54 :select **ltrim**(' Interface ') from dual;
Eg. 55 :select **rtrim**(' Interface ') from dual

SUBQUERIES :

A query nested within a query is known as subquery. Here the sub query will first computethe average salary and then main query will execute.

Eg. 55: Select * from emp where sal > (select avg(sal) from emp);
// To display the name and empno of that employee whose salary is maximum.
Eg. 56: Select * from emp where sal = (select max(sal) from emp);
// To see second maximum salary
Eg. 57: Select max(sal) from emp where sal < (select max(sal) from emp);
// To see the Third highest salary.
Eg. 58: Select max(sal) from emp where sal < (select max(sal) from emp where sal < (select max(sal) from emp));

Eg. 59: **// To List the top 5 salaries from employees table**
Select sal from (select sal from emp order sal desc) where rownum <= 5;

Eg. 60: **//To display the sum of salary by deptwise**

Select sum(sal) from emp group by deptno;

Eg. 61: // **To display the average total salary deptwise**

Select avg(depttotal) from (select sum(sal) as depttotal from emp group by deptno);

Eg. 62 : // **To display the sum of salary of all employees dept wise**

Select deptno,sum(sal) from emp group by deptno;

Eg. 63 : // **To display the Average of salary of all employees dept wise**

SQL> Select deptno,avg(sal) from emp group by deptno;

Eg. 64: **// To see the maximum salary in each department.**

SQL> Select deptno,max(sal) from emp group by deptno;

Eg. 65 : **// To see total salary department wise where the dept wise total salary is above 5000.**

SQL> Select deptno,sum(sal) from emp group by deptno having sum(sal) >= 5000;

Eg. 66 : **// To display deptname and average salary of them.**

Select dname,avg(sal) from emp,dept where emp.deptno=dept.deptno group by dname;Eg.

67: // **To display deptname and sum salary of them.**

Select dname,sum(sal) from emp,dept where emp.deptno=dept.deptno group by dname;

SQL Alias :

Alias is used to give an alias name to a table or a column. This is quite useful in case of largeor complex queries.

Syntax :

SELECT column-name from table-name **as** alias-name // Table alias Name

SELECT column-name **as** alias-name from table-name // Column Alias Name

For Eg. : SELECT C.ID, C.NAME, C.AGE, O.AMOUNT FROM **CUSTOMERS AS C**, **ORDERSAS O** WHERE C.ID = O.CUSTOMER_ID;

Transaction Control Language (TCL) :

Transaction control statements manage changes made by DML statements.

A transaction is a set of SQL statements which Oracle treats as a Single Unit. i.e. all thestatements should execute successfully or none of the statements should execute.

TCL Commands are ,

✓ **COMMIT**: Make changes done in transaction permanent.

✓ **ROLLBACK** : Rollbacks the state of database to the last commit point.

✓ **SAVEPOINT** : Use to specify a point in transaction to which later you can rollback.

✓ **COMMIT :**

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

Syntax :

COMMIT [WORK] ;

For Example

```
SQL> insert into emp (empno,ename,sal) values (101,‘Abid‘,2300);SQL> commit;
```

ROLLBACK :

To rollback the changes done in a transaction give rollback statement. Rollback restore thestate of the database to the last commit point.

Syntax :

rollback to savepoint-name;

For example,

```
SQL> delete from emp;SQL> rollback;
```

SAVEPOINT :

Specify a point in a transaction to which later you can roll back.

Syntax :

savepoint savepoint-name;

For Example,

```
SQL> insert into emp (empno,ename,sal) values (109,‘Sami‘,3000);SQL> savepoint a;
SQL> insert into dept values (10,‘Sales‘,‘Hyd‘);SQL> savepoint b;
SQL> insert into salgrade values (_III‘,9000,12000);SQL> rollback to a;
```

Join Types :

Depending on your requirements, you can do an "inner" join or an "outer" join. These aredifferent in a subtle way

Inner join

This will only return rows when there is at least one row in both tables that match the joincondition.

Syntax :

SELECT * FROM table_name1 INNER JOIN table_name2 ON table_name1.column_name = table_name2.column_name;

For Example ,

```
SQL> SELECT * FROM emp INNER JOIN customer ON emp.deptid =
customer.deptid;
```

Left outer join (or left join)

This will return rows that have data in the left table (left of the JOIN keyword), even ifthere's no matching rows in the right table.

Syntax :

SELECT * FROM table_name1 LEFT JOIN table_name2 ON table_name1.column_name =table_name2.column_name;

For Example ,

```
SQL> SELECT * FROM emp LEFT JOIN customer ON emp.deptid =
customer.deptid;
```

Right outer join (or right join)

This will return rows that have data in the right table (right of the JOIN keyword), even if there's no matching rows in the left table.

Syntax :

SELECT * FROM table_name1 RIGHT JOIN table_name2 ON table_name1.column_name =table_name2.column_name;

For Example ,

```
SQL> SELECT * FROM emp RIGHT JOIN customer ON emp.deptid =
customer.deptid;
```

Full outer join (or full join)

This will return all rows, as long as there's matching data in one of the tables.

Syntax :

SELECT * FROM table_name1 FULL JOIN table_name2 ON table_name1.column_name =table_name2.column_name;

For Example ,

```
SQL> SELECT * FROM emp FULL JOIN customer ON emp.deptid = customer.deptid
```

VIEWS :

A view in SQL is a logical subset of data from one or more tables. View is used to restrictdata access. Views are known as logical tables. They represent the data of one of more tables.

You can Query, Insert, Update and delete from views, just as any other table.

Syntax :

CREATE or REPLACE view view_name AS SELECT column_name(s) FROM table_nameWHERE condition

For example ,

// To Create View

```
SQL> CREATE or REPLACE view sale_view as select * from Sale where customer =
'Alex';
```

```
SQL>create view emp_det as select e.empno,e.ename,e.sal,e.deptno,d.dname,d.loc
from emp e,dept d where e.deptno=d.deptno;
```

// To display view

```
SQL> SELECT * from sale_view;
SQL> SELECT * from emp_det;
```

Force View Creation :

This keyword force to create View even if the table does not exist. After creating a force View if we create the base table and enter values in it, the view will be automatically updated.

For Example ,

```
SQL> CREATE or REPLACE force view sale_view1 as select * from Sale1 where
customer ='Abhi';
```

Read-Only View :

We can create a view with read-only option to restrict access to the view.

For Example .

```
SQL> CREATE or REPLACE view sale_view2 as select * from Sale where
customer ='ABC' with read-only;
```

SEQUENCES :

A sequence is used to generate numbers in sequence.

Syntax :

CREATE Sequence sequence-name start with initial-value increment by increment-value maxvaluemaximum-value cycle | nocycle

For Example ,

```
SQL> create sequence bills start with 1 increment by 1 minvalue 1 maxvalue 100
cyclecache 10;
```

Accessing Sequence Numbers :

To generate Sequence Numbers you can use NEXTVAL and CURRVAL.

```
SQL> Select bills.nextval from dual;

SQL> insert into sales (billno,custname,amt) values (bills.nextval,‘Sami‘,2300);

 // Creating Table with sequences and default :
```

```
SQL> create table invoices (invoice_no number(10) default bills.nextval, invoice_date date default sysdate,customer varchar2(100),invoice_amt number(12,2));
```

Altering Sequences :

To alter sequences use ALTER SEQUENCE statement.

For Example ,

SQL> ALTER SEQUENCE BILLS MAXVALUE 200;

Dropping Sequences :

SQL> drop sequence bills;

Listing Information About Sequences :

SQL> select * from user_sequences;

SYNONYMS:

A synonym is an alias or alternative name for objects like a table, view, snapshot, sequence, procedure, function, or package.

Two types of SYNONYMS are,

- ✓ Public Synonym
- ✓ Private Synonym

Syntax:

CREATE [OR REPLACE] [PUBLIC] SYNONYM [schema.] synonym_name FOR [schema.]object_name [@ dblink];

For Example,

```
SQL> create synonym employee for scott.emp;

SQL>create public synonym suppliers for scott.suppliers; // accessible to all users
```

View the Synonym :

```
SQL> select * from employee;
```

Dropping Synonyms :

```
SQL> drop synonym employee;
```

Listing information about synonyms INDEX:

```
SQL> select * from user_synonyms;
```

An index is a performance-tuning method of allowing faster retrieval of records. An index creates an entry for each value that appears in the indexed columns. By default, Oracle creates B-tree indexes.

Syntax :

CREATE INDEX index_name ON table_name (column_name);

For Example ,

```
SQL> CREATE INDEX empIndex ON emp(LastName); // Index on Single Column
SQL> CREATE INDEX supplier_idx ON supplier (supplier_name, city);
SQL>create index empno_ind on emp (empno);
SQL>create index empdept_ind on emp (empno,deptno);
```

Rename an Index :

Syntax :

ALTER INDEX index_name RENAME TO new_index_name;

For Example ,

```
SQL> ALTER INDEX supplier_idx RENAME TO supplier_index_name;
```

Drop an Index :

Syntax :

DROP INDEX index_name;

For example:

```
SQL>  DROP INDEX supplier_idx;
SQL>select * from user_indexes;
```

CLUSTERS :

A cluster is a group tables that share the same data blocks i.e. all the tables are physicallystored together.

Creating a cluster :

- To create clustered tables.
- First, create a cluster and create index on it.
- Then create tables in it.For Example ,

```
// Create a cluster
SQL>create cluster emp_dept (deptno number(2));
// Then create index on it.
SQL>create index on cluster emp_dept;
// create table in the cluster
SQL> create table dept (deptno number(2),name varchar2(20),loc
varchar2(20))cluster emp_dept(deptno);
SQL> create table emp (empno number(5),name varchar2(20),sal
number(10,2),deptno number(2))cluster emp_dept (deptno)
```

Dropping Clusters :

```
SQL> drop cluster emp_dept;
SQL> drop cluster emp_dept including tables;
```

Listing Information about Clusters :

```
SQL> select * from user_clusters;
```

PL/SQL :

PL/SQL stands for Procedural Language/SQL. PL/SQL extends SQL by adding constructsfound in procedural languages,

General Structure :

```
DECLARE
//Declarative section: variables, types, and local subprograms
BEGIN
//Executable section: procedural and SQL statements go here.
//This is the only section of the block that is required.
EXCEPTION
//Exception handling section: error handling statements go here.
      END;
```

`Ex.:

```
DECLARE
        a integer := 30;b integer := 40;c integer;
        f real;
BEGIN
        c := a + b;
        dbms_output.put_line('Value of c: ' || c);f := 100.0/3.0;
        dbms_output.put_line('Value of f: ' || f);
END;
```

-- Add two number

Sample Output :

```
SQL> set serveroutput on
Value of c: 70

Value of f: 33.33333333333333333333
PL/SQL procedure successfully completed.
```

Ex. : // Check the given number is Even or Add

```
DECLARE
        x NUMBER := 100;BEGIN
        FOR i IN 1..10 LOOP
IF MOD(i,2) = 0 THEN    -- i is even
     INSERT INTO temp VALUES (i, x, 'i is even');
ELSE
     INSERT INTO temp VALUES (i, x, 'i is odd');END IF;
     x := x + 100;END LOOP;
  COMMIT;
END;
```

Output Table :

SQL > SELECT * FROM temp ORDER BY col1;

Note : Create temp table three field

Ex. : PL/SQL Program to Find Factorial of a Number

```
declare
n number;
fac number:=1;i number;
begin
n:=&n;
for i in 1..nloop
fac:=fac*i;
end loop; dbms_output.put_line('factorial='||fac);
end;
/
```

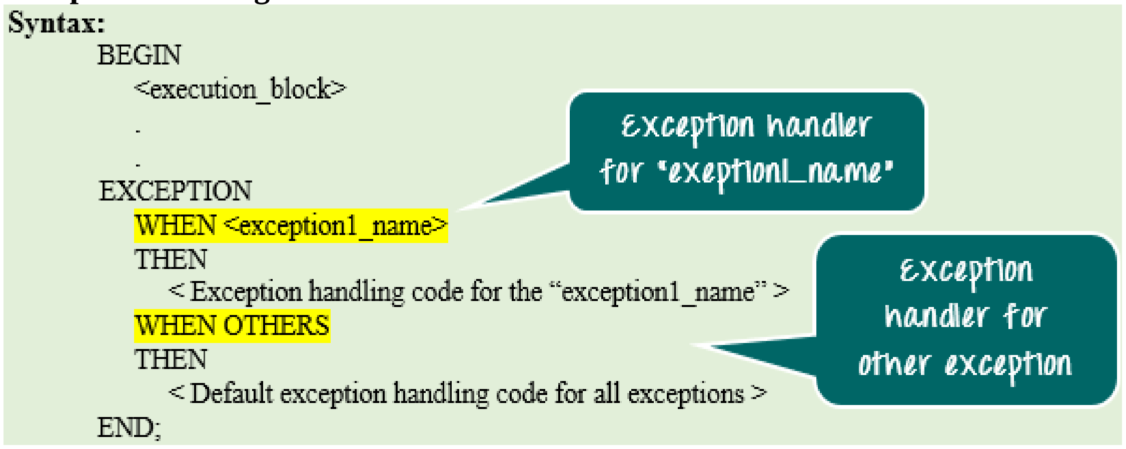
Sample Output :

```
Enter value for n: 10old 7: n:=&n;
new 7: n:=10; factorial=3628800
```

Ex. : PL/SQL Program to Reverse a String

```
Declare
str1 varchar2(50):='&str';str2 varchar2(50);
len number;i number;
begin
len:=length(str1); for i in reverse 1..lenloop
str2:=str2 || substr(str1,i,1);
end loop;
dbms_output.put_line('Reverse of String is:'||str2);
end;
```

Exception handling :



Syntax:

```
        BEGIN
            <execution_block>
            .
            .
        EXCEPTION
            WHEN <exception1_name>
            THEN
                < Exception handling code for the "exception1_name" >
            WHEN OTHERS
            THEN
                < Default exception handling code for all exceptions >
        END;
```

Exception handler for 'exeption1_name'
Exception handler for other exception

Ex. : PL/SQL Program using Exception handling

```
CREATE OR REPLACE PROCEDURE add_new_order (order_id_in IN
NUMBER, sales_in INNUMBER)
IS
no_sales EXCEPTION;        // user defined exception
BEGIN
IF sales_in = 0 THEN
RAISE no_sales;
EXCEPTION
WHEN no_sales THEN

raise_application_error (-20001,'You must have sales in order to submit the order.');
WHEN DUP_VAL_ON_INDEX THEN

raise_application_error (-20001,'You have tried to insert a duplicate order_id.');
WHEN OTHERS THEN raise_application_error (-20002,'An error has occurred
inserting an order.');

END;
/ELSE
INSERT INTO orders (order_id, total_sales )  VALUES ( order_id_in, sales_in );
END IF;
```

Ex. : (Switch case)

```
DECLARE
grade char(1) := 'A';
BEGIN
CASE grade
when 'A' then dbms_output.put_line('Excellent');
when 'B' then dbms_output.put_line('Very good');
when 'C' then dbms_output.put_line('Good');
when 'D' then dbms_output.put_line('Average');
when 'F' then dbms_output.put_line('Passed with Grace');
else dbms_output.put_line('Failed');
END CASE;
END;
```

Sample Output :

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

Types of PL/SQL Loops

There are 4 types of PL/SQL Loops.

1. Basic Loop / Exit Loop
2. While Loop
3. For Loop
4. Cursor For Loop

Ex. :

```
DECLARE
    VAR1 NUMBER;
BEGIN
    VAR1:=10;
    FOR VAR2 IN 1..10 // or FOR VAR2 IN REVERSE 1..10LOOP
            DBMS_OUTPUT.PUT_LINE (VAR1*VAR2);
    END LOOP;
END;
```

PL/SQL PROCEDURE :

The PL/SQL stored procedure or simply a procedure is a PL/SQL block which performs one or more specific tasks. The procedure contains a header and a body.

- **Header:** The header contains the name of the procedure and the parameters or variablespassed to the procedure.
- **Body:** The body contains a declaration section, execution section and exception sectionsimilar to a general PL/SQL block.

How to pass parameters in procedure:

There is three ways to pass parameters in procedure:

1. **IN parameters:** The IN parameter can be referenced by the procedure or function. Thevalue of the parameter cannot be overwritten by the procedure or the function.
2. **OUT parameters:** The OUT parameter cannot be referenced by the procedure or function,but the value of the parameter can be overwritten by the procedure or function.
3. **INOUT parameters:** The INOUT parameter can be referenced by the procedure or functionand the value of the parameter can be overwritten by the procedure or function.

Syntax for creating procedure:

```
CREATE [OR REPLACE] PROCEDURE procedure_name[ (parameter [,parameter]) ]
IS
[declaration_section]
BEGIN
executable_section[EXCEPTION
exception_section]
END [procedure_name];
```

Table creation:

```
SQL > create table user(id number(10) primary key,name varchar2(100));
```

Now write the procedure code to insert record in user table.

Procedure Code:

```
create or replace procedure "INSERTUSER" (id IN NUMBER, name IN
VARCHAR2) isbegin
   insert into user
values(id,name);end;
/
```

Sample Output:

```
Procedure created.
        SQL > EXEC procedure_name;
        SQL > EXEC insertuser(101,'Rahul');
```

PL/SQL program to call procedure

Let's see the code to call above created procedure.

```
  BEGIN
                insertuser(101,'Rahul');
                dbms_output.put_line('record inserted successfully');
  END;
```

PL/SQL FUNCTION :

The PL/SQL Function is very similar to PL/SQL Procedure. The main difference between procedure and a function is, a function must always return a value, and on the other hand a procedure may or may not return a value.

Syntax to create a function:

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

Ex. :

C**reate a function** :

```
create or replace function adder(n1 in number, n2 in number)
return number
is
n3 number(8);
begin
n3 :=n1+n2;
return n3;
end;
```

To **call the function** :

```
DECLARE
   n3 number(2);
BEGIN
      n3 := adder(11,22);
      dbms_output.put_line('Addition is: ' || n3);
END;
```

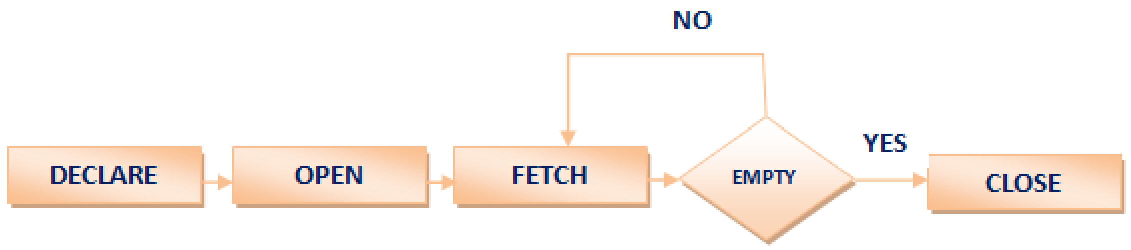
Output:

Addition is: 33 Statement processed.

0.05 seconds

PL/SQL Implicit Cursor :

A PL/SQL cursor is a pointer that points to the result set of an SQL query against database tables.



Create customers table and have records:

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	23	Allahabad	20000
2	Suresh	22	Kanpur	22000
3	Mahesh	24	Ghaziabad	24000
4	Chandan	25	Noida	26000
5	Alex	21	Paris	28000
6	Sunita	20	Delhi	30000

Let's execute the following program to update the table and increase salary of each customer by 5000. Here, SQL%ROWCOUNT attribute is used to determine the number of rows affected:

Create Cursor procedure:

```
DECLARE
total_rows number(2);
BEGIN
UPDATE customers SET salary = salary + 5000;
IF sql%notfound  THEN
          dbms_output.put_line('no customers updated');
ELSIF sql%found THEN
          total_rows := sql%rowcount;
          dbms_output.put_line( total_rows || ' customers updated ');
           END IF;
END;
```

Sample Output:

```
6 customers updated
PL/SQL procedure successfully completed.
```

PL/SQL Explicit Cursors:

Steps:

You must follow these steps while working with an explicit cursor.

1. Declare the cursor to initialize in the memory.
2. Open the cursor to allocate memory.
3. Fetch the cursor to retrieve data.
4. Close the cursor to release allocated memory.

Create Cursor procedure:

Execute the following program to retrieve the customer name and address.

```
DECLARE
c_id customers.id%type; c_name customers.name%type;c_addr
customers.address%type;
CURSOR c_customers is  SELECT id, name, address FROM customers;
BEGIN
OPEN c_customers;LOOP
FETCH c_customers into c_id, c_name, c_addr; EXIT WHEN
c_customers%notfound; dbms_output.put_line(c_id || ' ' || c_name || ' ' || c_addr);
END LOOP;
CLOSE c_customers;
END;
/
```

Sample Output:

```
1   Ramesh  Allahabad
2   Suresh  Kanpur
3   Mahesh  Ghaziabad
4   Chandan  Noida
5   Alex Paris
6   Sunita  Delhi
PL/SQL procedure successfully completed.
```

PL/SQL TRIGGER :

Trigger is invoked by Oracle engine automatically whenever a specified event occurs.Trigger is stored into database and invoked repeatedly, when specific condition match.

Advantages of Triggers

These are the following advantages of Triggers:

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

General Syntax :

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

Ex. : Create and display trigger for changing the salary on customer table

```
CREATE OR REPLACE TRIGGER display_salary_changes BEFORE
DELETE OR INSERT OR UPDATE ON customersFOR EACH ROW
WHEN (NEW.ID > 0)
DECLARE
   sal_diff number;
BEGIN
   sal_diff := :NEW.salary - :OLD.salary;
   dbms_output.put_line('Old salary: ' || :OLD.salary);
    dbms_output.put_line('New salary: ' || :NEW.salary);
   dbms_output.put_line('Salary difference: ' || sal_diff);
END;
/
```

Execute and call the trigger :

Check the salary difference by procedure:

Use the following code to get the old salary, new salary and salary difference after thetrigger created.

```
DECLARE
            total_rows number(2);
BEGIN
           UPDATE customers  SET salary = salary + 5000;
             IF sql%notfound THEN
                           dbms_output.put_line('no customers updated');
             ELSIF sql%found THEN
                           total_rows := sql%rowcount;
             dbms_output.put_line( total_rows || '
             customers updated ');
  END IF;
END;
/
```

Ex. No. :	**DATABASE DESIGN USING CONCEPTUAL MODELING (ER-EER)**
Date :	

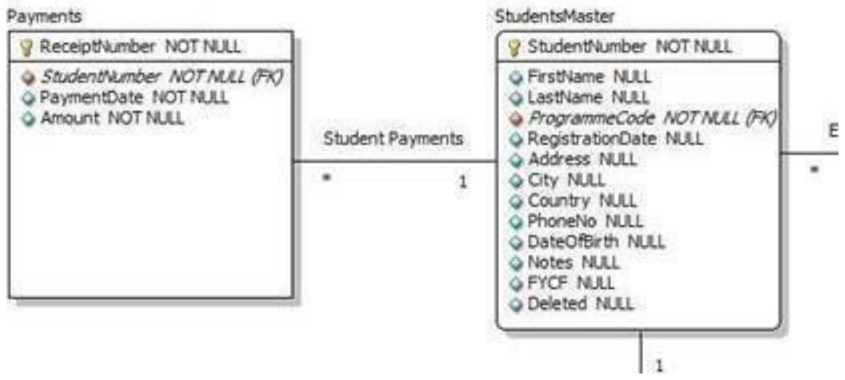
AIM: To create database using conceptual modeling using (ER-EER) model using SQL workbench**.**

PROCEDURE:

ENHANCED ENTITY RELATIONSHIP (EER) MODEL:

Enhanced Entity Relationship (EER) Model is a high level data model which provides extensions to original **Entity Relationship** (ER) model. EER Models supports more details design. EER Modeling emerged as a solution for modeling highly complex databases.

EER uses UML notation. UML is the acronym for Unified Modeling Language; it is a general purpose modeling language used when designing object oriented systems. Entities are represented as class diagrams. Relationships are represented as associations between entities. The diagram shown below illustrates an ER diagram using the UML notation.



Why use ER Model?
Now you may think why use ER modeling when we can simply create the database and all of its objects without ER modeling? One of the challenges faced when designing database is the fact thatdesigners, developers and end-users tend to view data and its usage differently. If this situation is left unchecked, we can end up producing a database system that does not meet the requirements of the users.

Communication tools understood by all stakeholders(technical as well non-technical users) are critical in producing database systems that meet the requirements of the users. ER models are examples of such tools.

ER diagrams also increase user productivity as they can be easily translated into relational tables.

Case Study: ER diagram for "MyFlix" Video Library

Let's now work with the MyFlix Video Library database system to help understand the concept of ER diagrams. We will using this database for all hand-on in the remainder of this tutorials

MyFlix is a business entity that rents out movies to its members. MyFlix has been storing itsrecords manually. The management now wants to move to a DBMS

Let's look at the steps to develop EER diagram for this database-

1. Identify the entities and determine the relationships that exist among them.
2. Each entity, attribute and relationship, should have appropriate names that can be easilyunderstood by the non-technical people as well.
3. Relationships should not be connected directly to eachother. Relationships should connectentities.
4. Each attribute in a given entity should have a unique name.

Entities in the "MyFlix" library

The entities to be included in our ER diagram are;

- **Members** - this entity will hold member information.
- **Movies** - this entity will hold information regarding movies
- **Categories** - this entity will hold information that places movies into different categoriessuch as "Drama", "Action", and "Epic" etc.
- **Movie Rentals** - this entity will hold information that about movies rented out to members.
- **Payments** - this entity will hold information about the payments made by members.

Defining the relationships among entities Members and movies

The following holds true regarding the interactions between the two entities.

A member can rent a more than movie in a given period.
A movie can be rented by more than one member in a given period.

From the above scenario, we can see that the nature of the relationship is many-to-many. **Relational databases do not support many-to-many relationships. We need to introduce a junction entity**. This is the role that the MovieRentals entity plays. It has a one-to-many relationship with the members table and another one-to-many relationship with movies table.

Movies and categories entities

The following holds true about movies and categories.

- A movie can only belong to one category but a category can have more than one movie.

We can deduce from this that the nature of the relation between categories and movies table is one-to- many.

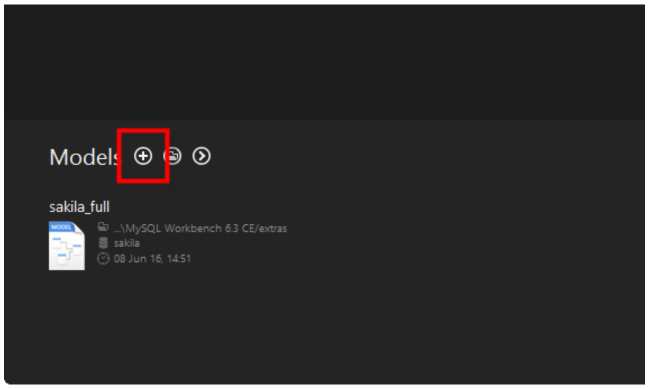
Members and payments entities

The following holds true about members and payments

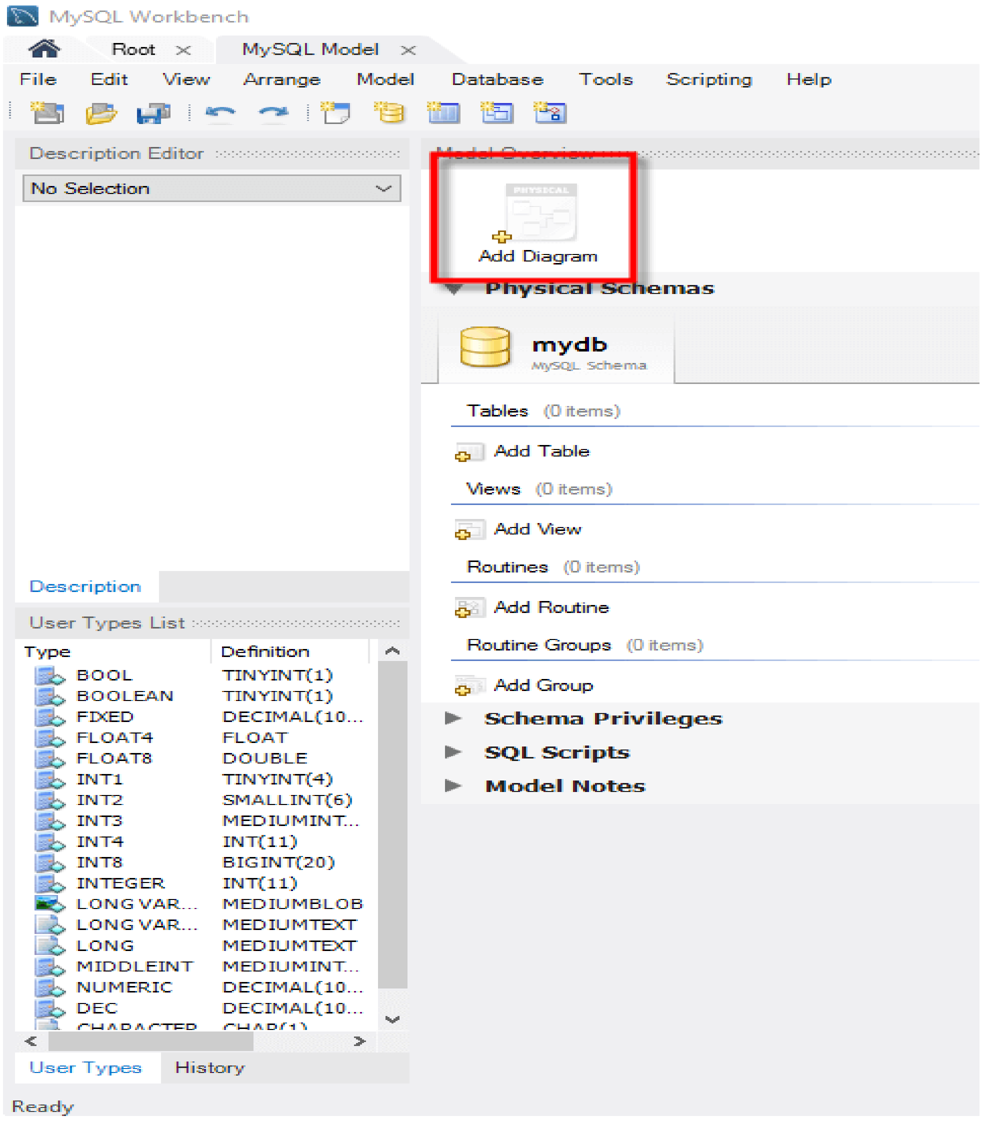
- A member can only have one account but can make a number of payments.

We can deduce from this that the nature of the relationship between members and payments entities is one-to-many.

Now lets create EER model using MySQL Workbench In the MySQL workbench , Click - "+"Button

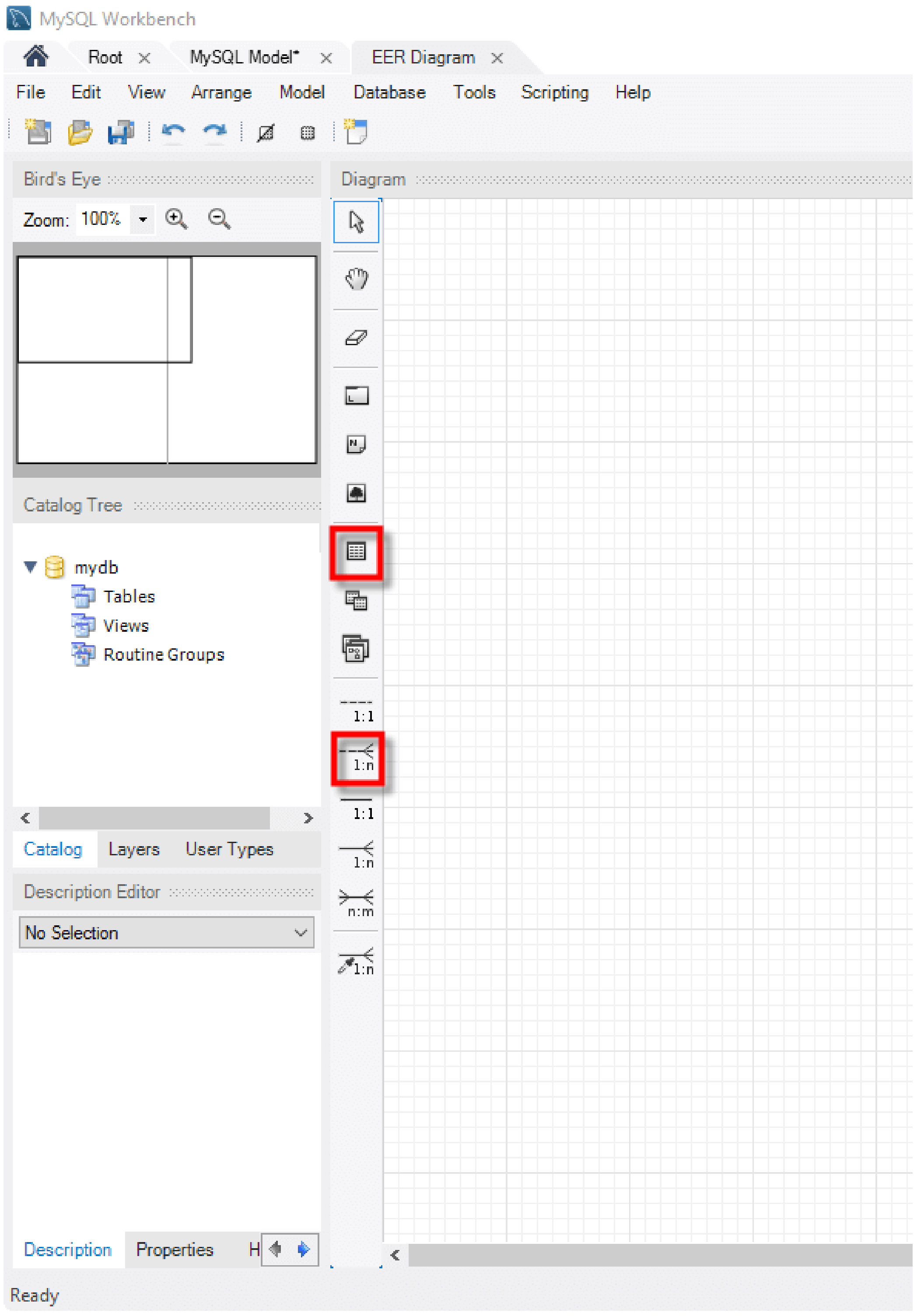


Double click on Add Diagram button to open the workspace for ER diagrams.



Type	Definition
BOOL	TINYINT(1)
BOOLEAN	TINYINT(1)
FIXED	DECIMAL(10...
FLOAT4	FLOAT
FLOAT8	DOUBLE
INT1	TINYINT(4)
INT2	SMALLINT(6)
INT3	MEDIUMINT...
INT4	INT(11)
INT8	BIGINT(20)
INTEGER	INT(11)
LONG VAR...	MEDIUMBLOB
LONG VAR...	MEDIUMTEXT
LONG	MEDIUMTEXT
MIDDLEINT	MEDIUMINT...
NUMERIC	DECIMAL(10...
DEC	DECIMAL(10...
CHARACTER	CHAR(1)

Following window appears



Let's look at the two objects that we will work with.

The table object allows us to create entities and define the attributes associated with the particular entity.

The place relationship button allows us to define relationships between entities.

The **members'** entity will have the following attributes

- Membership number
- Full names
- Gender
- Date of birth
- Physical address
- Postal address

Let's now create the members table

1. Drag the table object from the tools panel
2. Drop it in the workspace area. An entity named table 1 appears
3. Double click on it. Theproperties window shown below appears

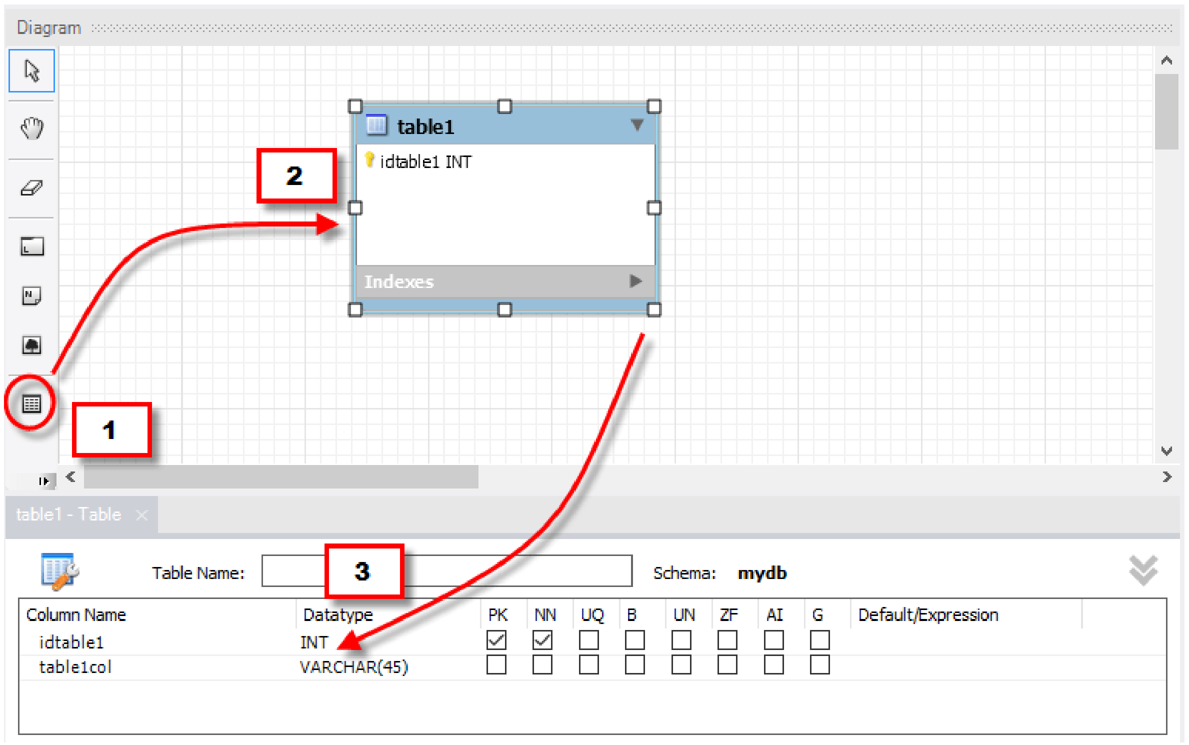


Table Name:

Schema: mydb

Column Name	Datatype	PK	NN	UQ	B	UN	ZF	AI	G	Default/Expression
idtable1	INT	☑	☑	☐	☐	☐	☐	☐	☐	
table1col	VARCHAR(45)	☐	☐	☐	☐	☐	☐	☐	☐	

Next ,

1. Change table 1 to Members
2. Edit the default idtable1 to membership_number
3. Click on the next line to add the next field
4. Do the same for all the attributes identified in members' entity.

Your properties window should now look like this.

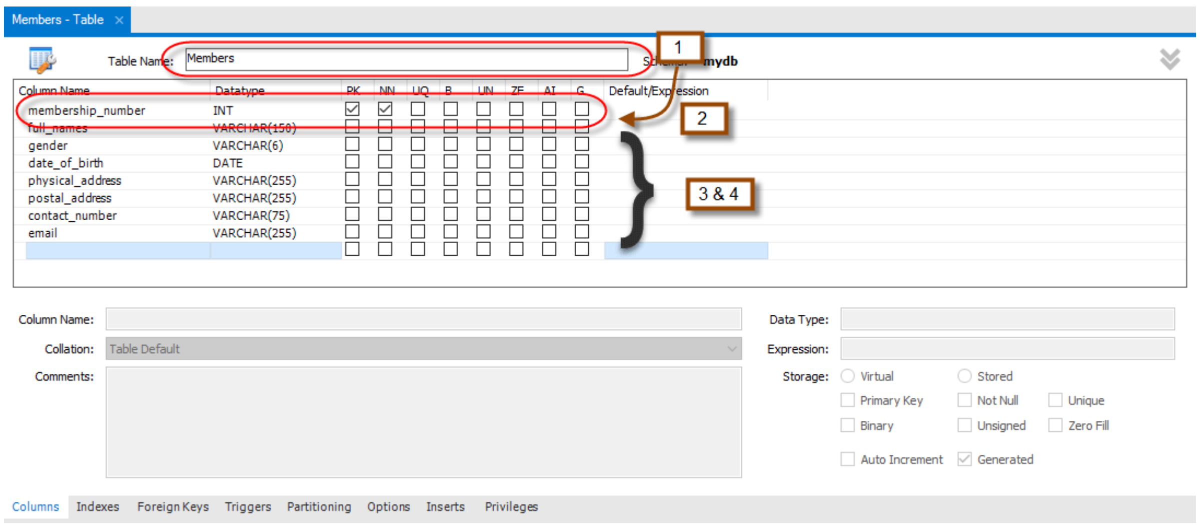


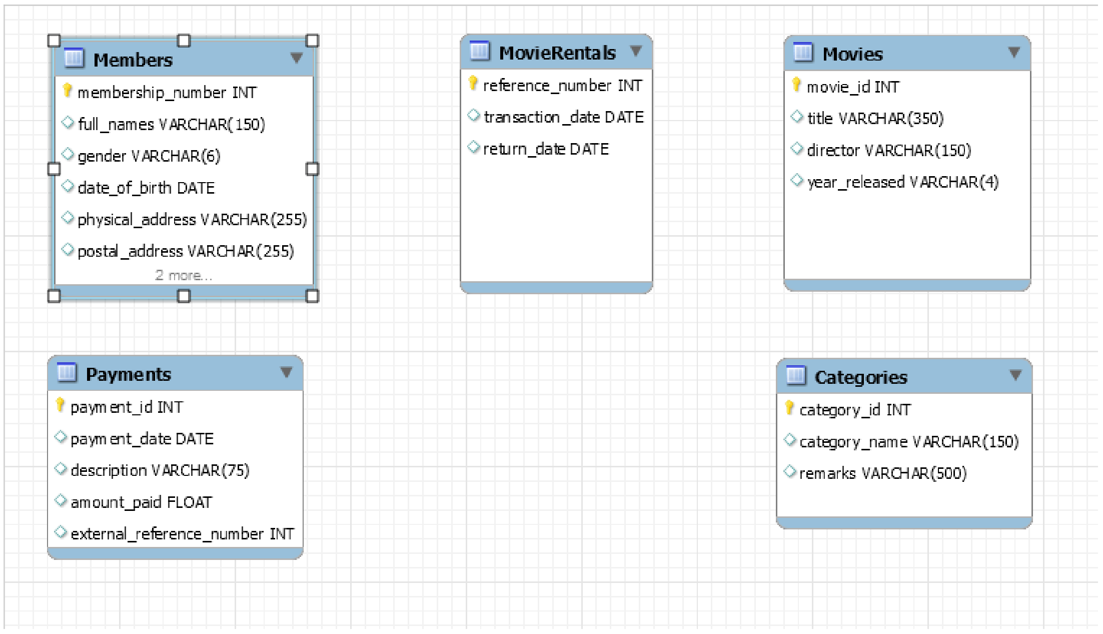
Table Name: Members Schema: mydb

Column Name	Datatype	PK	NN	UQ	B	UN	ZF	AI	G	Default/Expression
membership_number	INT	✓	✓							
full_names	VARCHAR(150)									
gender	VARCHAR(6)									
date_of_birth	DATE									
physical_address	VARCHAR(255)									
postal_address	VARCHAR(255)									
contact_number	VARCHAR(75)									
email	VARCHAR(255)									

Column Name:
Collation: Table Default
Comments:
Data Type:
Expression:
Storage: Virtual Stored
Primary Key Not Null Unique
Binary Unsigned Zero Fill
Auto Increment [✓ Generated]

Repeat the above steps for all the identified entities.

Your diagram workspace should now look like the one shown below.



Members
- membership_number INT
- full_names VARCHAR(150)
- gender VARCHAR(6)
- date_of_birth DATE
- physical_address VARCHAR(255)
- postal_address VARCHAR(255)
- 2 more...

MovieRentals
- reference_number INT
- transaction_date DATE
- return_date DATE

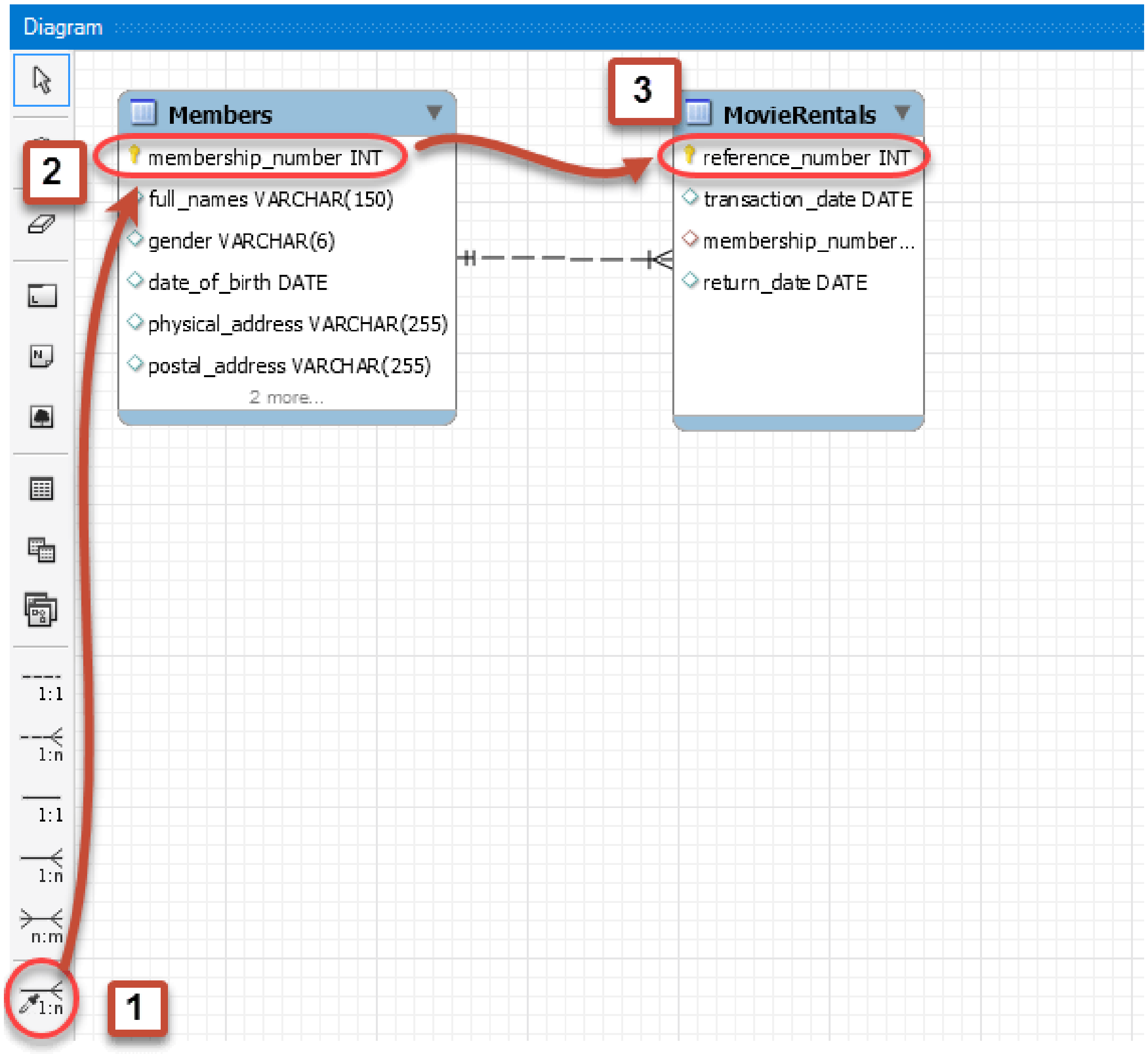
Movies
- movie_id INT
- title VARCHAR(350)
- director VARCHAR(150)
- year_released VARCHAR(4)

Payments
- payment_id INT
- payment_date DATE
- description VARCHAR(75)
- amount_paid FLOAT
- external_reference_number INT

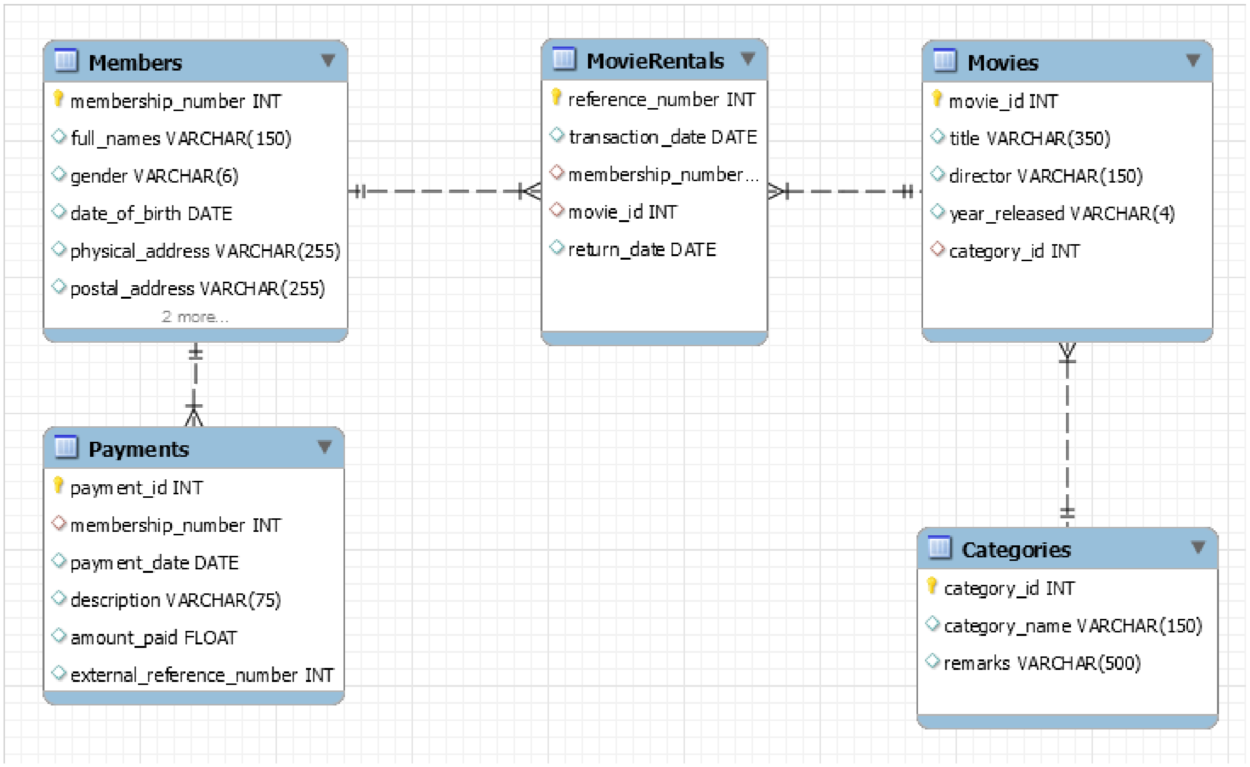
Categories
- category_id INT
- category_name VARCHAR(150)
- remarks VARCHAR(500)

Lets create relationship between Members and Movie Rentals

1. Select the place relationship using existing columns too
2. Click on membership_number in the Members table
3. Click on reference_number in the MovieRentals table



Repeat above steps for other relationships. Your ER diagram should now look like this –



Summary

- ✓ ER Diagrams play a very important role in the database designing process. They serve as anon-technical communication tool for technical and non-technical people.
- ✓ Entities represent real world things; they can be conceptual as a sales order or physical such asa customer.
- ✓ All entities must be given unique names.
- ✓ ER models also allow the database designers to identify and define the relations that existamong entities.

Ex. No. :	IMPLEMENT THE DATABASE USING SQL DATA DEFINITION WITH CONSTRAINTS, VIEWS
Date :	

AIM:

To create table and Execute Data Definition Commands with Constraints and Views.

DESCRIPTION:

Data Definition Language:

DDL (Data Definition Language) statements are used to create, change the objects of a database. Typically, a database administrator is responsible for using DDL statements or production databases in a large database system.

The commands used are:

Create - It is used to create a table.

Alter - This command is used to add a new column, modify the existing column definition and to include or drop integrity constraint.

Drop - It will delete the table structure provided the table should be empty.
Truncate - If there is no further use of records stored in a table and the structure has to be retained,and then the records alone can be deleted.

Desc - This is used to view the structure of the table.

Most commonly used data types for Table columns

Here we have listed some of the most commonly used data types used for columns in tables.

Datatype	**Use**
integer	used for columns which will store integer values. (-2147483648 to +2147483647)
numeric	used for columns which will store user-specified precision, exact.
real	used for columns which will store variable-precision, inexact. (6 decimal digits precision)
money	used for columns which will store a currency amount with a fixed fractional precision
varchar	used for columns which will be used to store characters and integers, variable-length with limit.
char	used for columns which will store char values (fixed-length, blank padded).
text	used for columns which will store char values (variable unlimited length)
date	used for columns which will store date values. (4713 BC to 5874897 AD)

CREATING BASE TABLE

Description: CREATE TABLE query in PostgreSQL

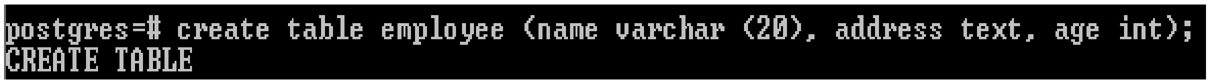
CREATE TABLE is a keyword that will create a new, initially empty table in the database.

Syntax:
CREATE TABLE <table_name> (column1 datatype, column2 datatype ...);

create table command will tell the database system to create a new table with the given table nameand column information.

Example:

postgres=# create table employee (name varchar (20), address text, age int);



```
postgres=# create table employee (name varchar (20), address text, age int);
CREATE TABLE
```

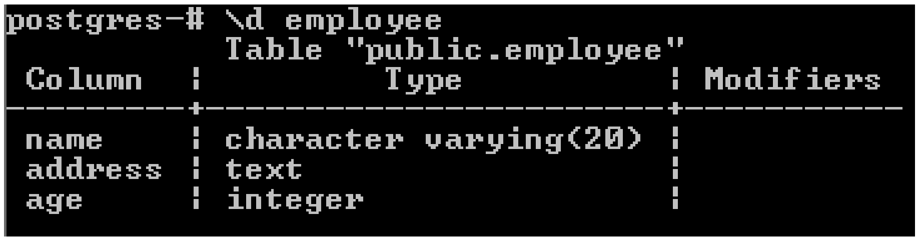
DESCRIBING THE TABLE CREATED

Description: To view the table /shows the description of the table / format of the table

Syntax: \d <table name>;

Example:

postgres=# \d employee

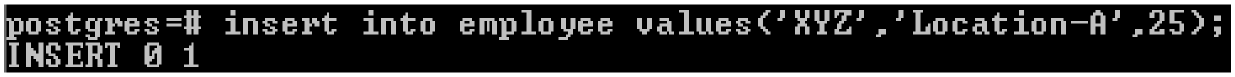


```
postgres=# \d employee
           Table "public.employee"
 Column  |         Type          | Modifiers
---------+-----------------------+-----------
 name    | character varying(20) |
 address | text                  |
 age     | integer               |
```

INSERT query in PostgreSQL

The INSERT command is used to insert data into a table:

postgres=# insert into <table name> values ('XYZ', 'location-A', 25);



```
postgres=# insert into employee values('XYZ','Location-A',25);
INSERT 0 1
```

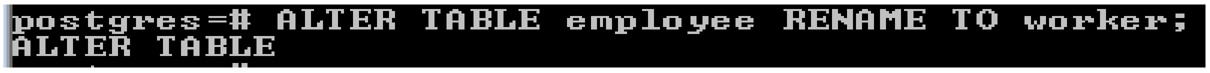
CHANGING NAME OF THE TABLE:

PostgreSQL has a RENAME clause that is used with the ALTER TABLE statement to rename thename of an existing table.

Syntax: ALTER TABLE table_name RENAME TO new_table_name;

ALTER TABLE:

postgres=# alter table employee rename to worker;



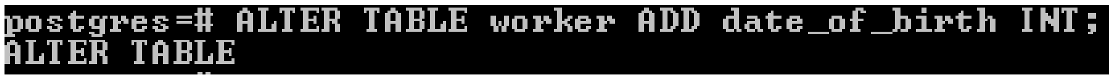
```
postgres=# ALTER TABLE employee RENAME TO worker;
ALTER TABLE
```

PostgreSQL ALTER Table: ADD Column, Rename Column/Table

The ALTER TABLE command is used to alter the structure of a PostgreSQL table. It is the commandused to change the table columns or the name of the table.
Syntax for the PostgreSQL ALTER TABLE command:

ALTER TABLE table-name action;
The table-name parameter is the name of the table that you need to change.
The action parameter is the action that you need to perform, such as changing the name of a column,changing the data type of a column, etc.

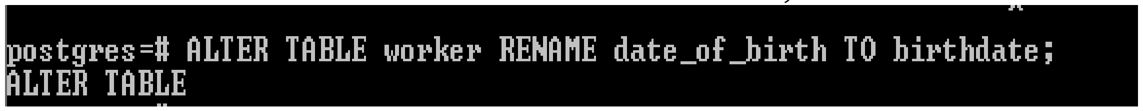
Adding a New column
To add a new column to a PostgreSQL table, the ALTER TABLE command is used with the following syntax:
ALTER TABLE table-name
ADD new-column-name column-definition;



```
postgres=# ALTER TABLE worker ADD date_of_birth INT;
ALTER TABLE
```

Renaming a Column:
ALTER TABLE table-name

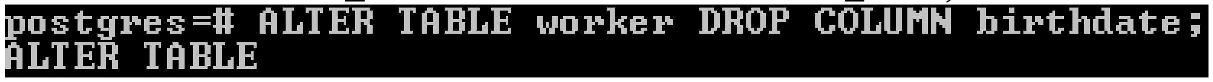
RENAME old-column-name To new-column-name;



```
postgres=# ALTER TABLE worker RENAME date_of_birth TO birthdate;
ALTER TABLE
```

Dropping a Column:

To drop a column of a table, you use the DROP COLUMN clause in the ALTER TABLE statementas follows:
ALTER TABLE table_name DROP COLUMN column_name;



```
postgres=# ALTER TABLE worker DROP COLUMN birthdate;
ALTER TABLE
```

If the column that you want to remove is used in other database objects such as views, triggers, stored procedures, etc., you cannot drop the column because other objects are depending on it. In this case, you need to add the CASCADE option to the DROP COLUMN clause to drop the column and all of its dependent objects:

ALTER TABLE table_name
DROP COLUMN column_name CASCADE;

If you remove a column that does not exist, PostgreSQL will issue an error. To remove a column onlyif it exists, you can add the IF EXISTS option as follows:

ALTER TABLE table_name
DROP COLUMN IF EXISTS column_name;

In this form, if you remove a column that does not exist, PostgreSQL will issue a notice instead of an
error.

If you want to drop multiple columns of a table in a single command, you use multiple DROPCOLUMN clause like this:

ALTER TABLE table_name DROP COLUMN column_name1, DROP COLUMN column_name2,
...;

CREATE VIEW query in PostgreSQL

The CREATE VIEW command is used to generate views. Views are pseudo-tables, which are used topresent a full table, subset, or select columns from the underlying table:
create or replace view vi as select * from <table name>;

TRUNCATE:

The truncate table

- Removes all rows from a table

- Release the storage space used by that table

Syntax: truncate table <table name>;

<u>Example:</u>

postgres=# Truncate table employee;Table truncated.

DROP TABLE:

All data and structure in the table is deletedAny pending transactions are committed.
All indexes are dropped.
Syntax: drop table <table name>;postgres=# drop table employee; Table dropped.

Constraints are the rules enforced on data columns on table. These are used to prevent invalid data from being entered into the database. This ensures the accuracy and reliability of the data in the database.

The following are commonly used constraints available in PostgreSQL.

- **NOT NULL Constraint** − Ensures that a column cannot have NULL value.
- **UNIQUE Constraint** − Ensures that all values in a column are different.
- **PRIMARY Key** − Uniquely identifies each row/record in a database table.
- **FOREIGN Key** − Constrains data based on columns in other tables.
- **CHECK Constraint** − The CHECK constraint ensures that all values in a column satisfycertain conditions.

NOT NULL Constraint

By default, a column can hold NULL values. If you do not want a column to have a NULL value, then you need to define such constraint on this column specifying that NULL is now not allowed for that column. A NOT NULL constraint is always written as a column constraint.

A NULL is not the same as no data; rather, it represents unknown data.

Example

For example, the following PostgreSQL statement creates a new table called COMPANY1 and adds five columns, three of which, ID and NAME and AGE, specify not to accept NULL values −

```
CREATE TABLE COMPANY1(NAME TEXT NOT NULL, AGE INT NOT NULL,
ADDRESS CHAR(50), SALARY REAL);
```

UNIQUE Constraint

The UNIQUE Constraint prevents two records from having identical values in a particular column. In the COMPANY table, for example, you might want to prevent two or more people from having identical age.

Example

For example, the following PostgreSQL statement creates a new table called COMPANY3 and adds five columns. Here, AGE column is set to UNIQUE, so that you cannot have two records with same age –

```
CREATE TABLE COMPANY3(NAME TEXT NOT NULL, AGE INT NOT NULL
UNIQUE, ADDRESS CHAR(50), SALARY REAL DEFAULT 50000.00);
```

PRIMARY KEY Constraint

The PRIMARY KEY constraint uniquely identifies each record in a database table. There can be more UNIQUE columns, but only one primary key in a table. Primary keys are important when designing the database tables. Primary keys are unique ids.

A table can have only one primary key, which may consist of single or multiple fields. Whenmultiple fields are used as a primary key, they are called a **composite key**.

Example

CREATE TABLE COMPANY4(NAME TEXT NOT NULL, AGE INT NOT NULL, ADDRESS CHAR(50), SALARY REAL);

FOREIGN KEY Constraint

A foreign key constraint specifies that the values in a column (or a group of columns) must matchthe values appearing in some row of another table. We say this maintains the referential integrity between two related tables. They are called foreign keys because the constraints are foreign; that is, outside the table. Foreign keys are sometimes called a referencing key.

Example

For example, the following PostgreSQL statement creates a new table called COMPANY5 and add five columns.

CREATE TABLE COMPANY6 (NAME TEXT NOT NULL, AGE INT NOT NULL, ADDRESS CHAR(50));

For example, the following PostgreSQL statement creates a new table called DEPARTMENT1, which adds three columns. The column EMP_ID is the foreign key and references the ID field of thetable COMPANY6.

CREATE TABLE DEPARTMENT1 (DEPT CHAR(50) NOT NULL, EMP_ID INT references COMPANY6(ID));

CHECK Constraint

The CHECK Constraint enables a condition to check the value being entered into a record. If the condition evaluates to false, the record violates the constraint and is not entered into the table.

Example

For example, the following PostgreSQL statement creates a new table called COMPANY5 and adds five columns. Here, we add a CHECK with SALARY column, so that you cannot have anySALARY as Zero.

CREATE TABLE COMPANY5 (ID INT PRIMARY KEY NOT NULL, NAME TEXT NOT NULL, AGE INT NOT NULL, ADDRESS CHAR(50), SALARY REAL CHECK(SALARY >0) ;

Dropping Constraints

To remove a constraint, we need to know its name. If the name is known, it is easy to drop. Else, we need to find out the system-generated name. The psql command \d table name can be helpful here. The general syntax is −

```
ALTER TABLE table_name DROP CONSTRAINT some_name;
```

Ex. No. :	QUERY THE DATABASE USING SQL MANIPULATION
Date :	

INSERTION, DELETION, MODIFYING, ALTERING, UPDATING AND VIEWING RECORDS BASED ON CONDITIONS

AIM:

To implement Insertion, Deletion, Modifying, Altering, Updating and Viewing recordsbased on conditions using SQL commands (DML & DCL).

PROCEDURE

STEP 1: Start.

STEP 2: Create the table with its essential attributes.STEP 3: Insert the record into table.

STEP 4: Update the existing records into the table.STEP 5: Delete the records in to the table.

STEP 6: Alter the existing records into the table.STEP 7: Viewing the records from the table.

STEP 8: End.

IMPLEMENTATION :

Problem 2.1: Insert 5 records into dept table.Solution:

1. Decide the data to add in dept.
2. Add to dept one row at a time using the insert into syntax.

```
Ans: CREATE TABLE dept (DNAME int(2), DEPTNO varchar(15), DLOC varchar(15));
INSERT INTO dept VALUES (10, 'MANAGEMENT', 'MAIN BLOCK');
INSERT INTO dept VALUES (20, 'Manufacturing', null);
INSERT INTO dept VALUES (30, 'MAINTAIN ANC', 'MAIN BLOCK');
INSERT INTO dept VALUES (40, 'TRANSPORT', 'ADMIN BLOCK');
INSERT INTO dept VALUES (50, 'SALES', 'HEAD OFFICE');
SELECT * FROM dept;
```

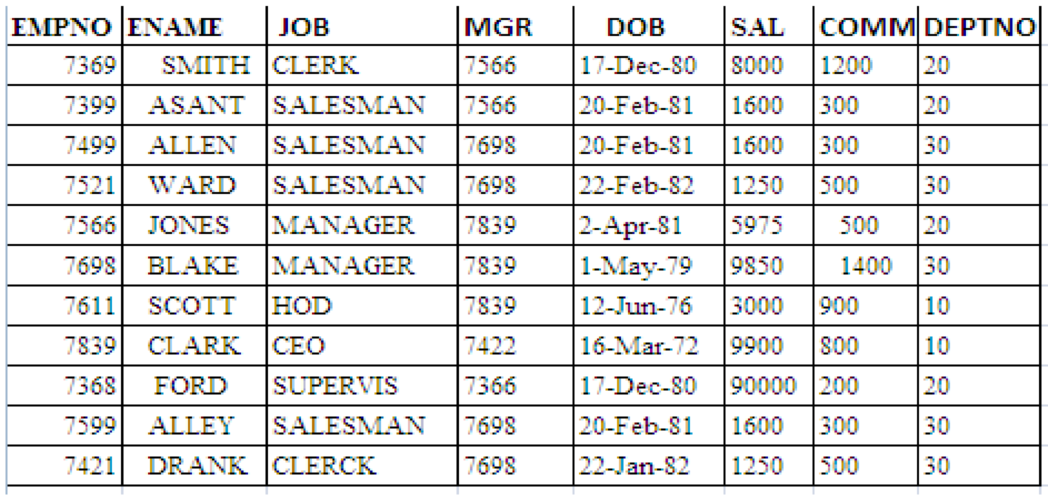
DNAME	DEPTNO	DLOC
10	MANAGEMENT	MAIN BLOCK
20	Manufacturing	
30	MAINTAINANC	MAIN BLOCK
40	TRANSPORT	ADMIN BLOCK
50	SALES	HEAD OFFICE

Problem 2.2: Insert 11 records

into emp table.Solution:

1. Decide the data to add in emp.
2. Add to emp one row at a time using the insert into syntax.

```
Ans: CREATE TABLE emp (EMPNO int(4),ENAME varchar(20),JOB varchar(15),MGR int(4),DOB
date,SAL int(7), COMM int(5),DEPTNO int(2));
INSERT INTO emp VALUES (7369, 'SMITH', 'CLERK', 7566, '17-Dec-80', 8000, 1200, 20);
INSERT INTO emp VALUES(7399, 'ASANT', 'SALESMAN', 7566, '20-Feb-81', 1600, 300, 20);
INSERT INTO emp VALUES (7499, 'ALLEN', 'SALESMAN', 7698, '20-Feb-81', 1600, 300, 30);
INSERT INTO emp VALUES(7611, 'SCOTT', 'HOD', 7839, '12-June-76', 3000, 900, 10);
INSERT INTO emp VALUES (7368, 'FORD', 'SUPERVIS', 7366, '17-Dec-80', 90000, 200, 20);
INSERT INTO emp VALUES(7421, 'DRANK', 'CLERK', 7698, '22-Jan-82', 1250, 500, 30);
INSERT INTO emp VALUES(7521, 'WARD', 'SALESMAN', 7698, '22-Feb-82', 1250, 500, 30);
INSERT INTO emp VALUES (7566, 'JONES', 'MANAGER', 7839, '2-Apr-81', 5975, 500, 20);
INSERT INTO emp VALUES(7698, 'BLAKE', 'MANAGER', 7839, '1-May-79', 9850, 1400, 30);
INSERT INTO emp VALUES (7839, 'CLARK', 'CEO', 7422, '16-Mar-72', 9900, 800, 10);
INSERT INTO emp VALUES (7599, 'ALLEY', 'SALESMAN', 7698, '20-Feb-81', 1600, 300, 30);
select * FROM emp;
```



EMPNO	ENAME	JOB	MGR	DOB	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7566	17-Dec-80	8000	1200	20
7399	ASANT	SALESMAN	7566	20-Feb-81	1600	300	20
7499	ALLEN	SALESMAN	7698	20-Feb-81	1600	300	30
7521	WARD	SALESMAN	7698	22-Feb-82	1250	500	30
7566	JONES	MANAGER	7839	2-Apr-81	5975	500	20
7698	BLAKE	MANAGER	7839	1-May-79	9850	1400	30
7611	SCOTT	HOD	7839	12-Jun-76	3000	900	10
7839	CLARK	CEO	7422	16-Mar-72	9900	800	10
7368	FORD	SUPERVIS	7366	17-Dec-80	90000	200	20
7599	ALLEY	SALESMAN	7698	20-Feb-81	1600	300	30
7421	DRANK	CLERCK	7698	22-Jan-82	1250	500	30

Problem 2.3: Update the emp table to set the default commission of all employees to Rs

1000/-who are working as managers

Solution:

1. Learn update table syntax.
2. Update the default commission as Rs 1000/-

```
Ans: UPDATE emp
SET SAL = 1000

WHERE JOB = 'MANAGER';
SELECT * FROM emp;
```

Problem 2.4: Create a pseudo table employee with the same structure as the table emp andinsert rows into the table using select clauses.

Solution:

1. Create a table employee as emp.
2. Use Select clause to perform this.

```
Ans: CREATE TABLE employee AS
SELECT * FROM emp;
SELECT * FROM employee;
```

Problem : Delete only those who are working as supervisors.Solution:

1. Delete the employee whose job is supervisor.Ans:

```
DELETE FROM emp WHERE JOB = 'SUPERVIS';
SELECT * FROM emp;
```

Problem : Delete the rows whose empno is 7599.Solution:

1. Delete the employee whose empno is 7599.

```
DELETE FROM emp
WHERE EMPNO = 7599;
SELECT * FROM emp;
```

Problem: List the records in the emp table orderby salary in ascending order.Solution:

1. Use the orderby function in select clause.

```
Ans: SELECT * FROM emp
   ORDER BY SAL;
```

Problem : List the records in the emp table orderby salary in descending order.Solution:

1. Use the orderby function in select clause.

```
Ans: SELECT * FROM emp
   ORDER BY SAL DESC;
```

Problem : Display only those employees whose deptno is 30

Solution:

1. Use SELECT FROM WHERE syntax.
2. Select should include all in the given format.
3. from should include employee
4. Where should include condition on deptn0 = 30.

```
Ans: SELECT * FROM employee
        WHERE DEPTNO = 30;
```

Problem : Display deptno from the table employee avoiding the duplicated values.
Solution:

1. Use SELECT FROM syntax.
2. select should include distinct clause for the deptno.
3. from should include employee

```
Ans: SELECT DISTINCT(DEPTNO) FROM employee;
```

Problem : List the records in sorted order of their employees
Solution:

1. Use SELECT FROM syntax.
2. Select should include all in the given format.
3. from should include employee
4. Use the order by function empname.

```
Ans: SELECT * FROM employee
        ORDER BY ENAME;
```

Problem : List the employee names whose commission is null.

```
Ans: SELECT ENAME FROM emp
   WHERE COMM = null;
```

Consider the following Table :

Table Name : Salesperson

ID	Name	Age	Salary
1	Abe	61	140000
2	Bob	34	44000
5	Chris	34	40000
7	Dan	41	52000
8	Ken	57	115000
11	Joe	38	38000

Table Name : Customer

ID	Name	City	Industry Type
4	Samsonic	Pleasant	J
6	Panasung	oaktown	J
7	Samony	Jackson	B
9	Orange	Jackson	B

Table Name : Orders

Number	order_date	cust_id	salesperson_id	Amount
10	8/2/96	4	2	540
20	1/30/99	4	8	1800
30	7/14/95	9	1	460
40	1/29/98	7	2	2400
50	2/3/98	6	7	600
60	3/2/98	6	7	720
70	5/6/98	9	7	150

Given the tables above, find the following:

a. **The names of all salespeople that have an order with Samsonic.**
b. **The names of all salespeople that do not have any order with Samsonic.**
c. **The names of salespeople that have 2 or more orders.**

d. Write a SQL statement to insert rows into a table called highAchiever (Name, Age), where asalesperson must have a salary of 100,000 or greater to be included in the table

Ans :

a. SElECT Salesperson.s_Name FROM Salesperson, Orders WHERE Salesperson.ID = Orders.salesperson_id AND cust_id = '4';

b. SELECT Salesperson.s_Name FROM Salesperson
WHERE Salesperson.ID NOT IN(
SELECT Orders.salesperson_id FROM Orders, Customer
WHERE Orders.cust_id = Customer.ID
AND Customer.c_Name = 'Samsonic');

c. SELECT s_Name
FROM Orders, Salesperson
WHERE Orders.salesperson_id = Salesperson.id
GROUP BY s_Name, salesperson_id
HAVING COUNT(salesperson_id) >1;

d. CREATE TABLE highAchiever(ID INT primary key NOT NULL auto_increment,H_name varchar(50),Age INT);

insert into highAchiever (H_name, age)
(select s_Name, age from salesperson where salary >= 100000);

Ex. No. :	**QUERYING / MANAGING THE DATABASE USING SQL PROGRAMMING**
Date :	• **STORED PROCEDURES/FUNCTIONS** • **CONSTRAINTS AND SECURITY USING TRIGGERS**

AIM: To manage database by procedures and triggers using SQL programming.

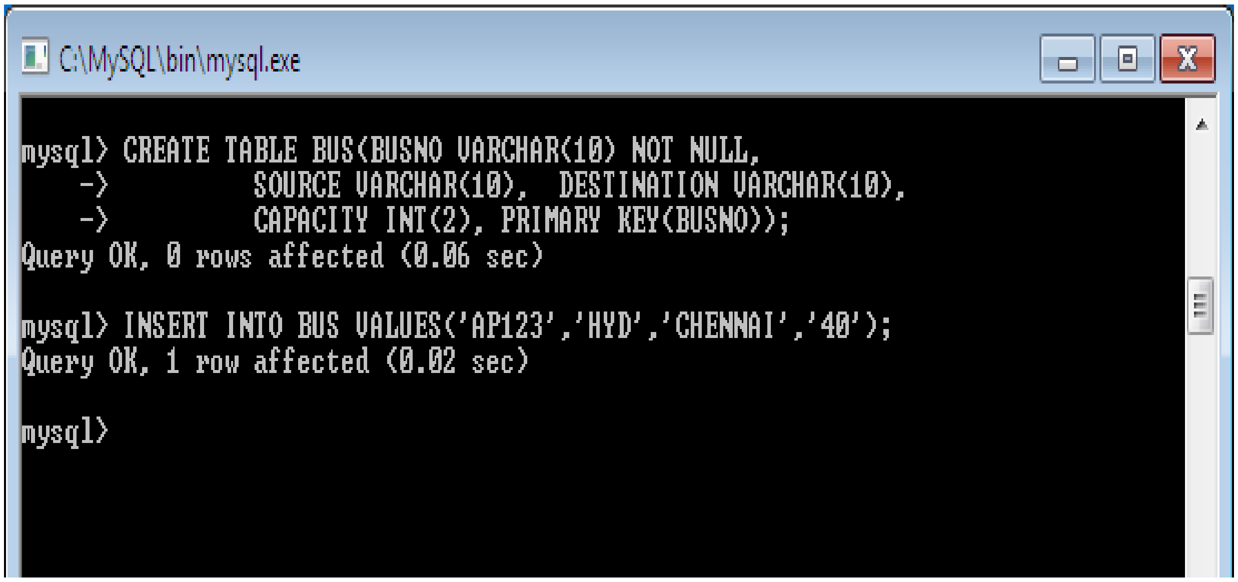
PROCEDUE:

PL/SQL Program Units

- PL/SQL block
- Function
- Package
- Package body
- Procedure
- Trigger
- Type
- Type body

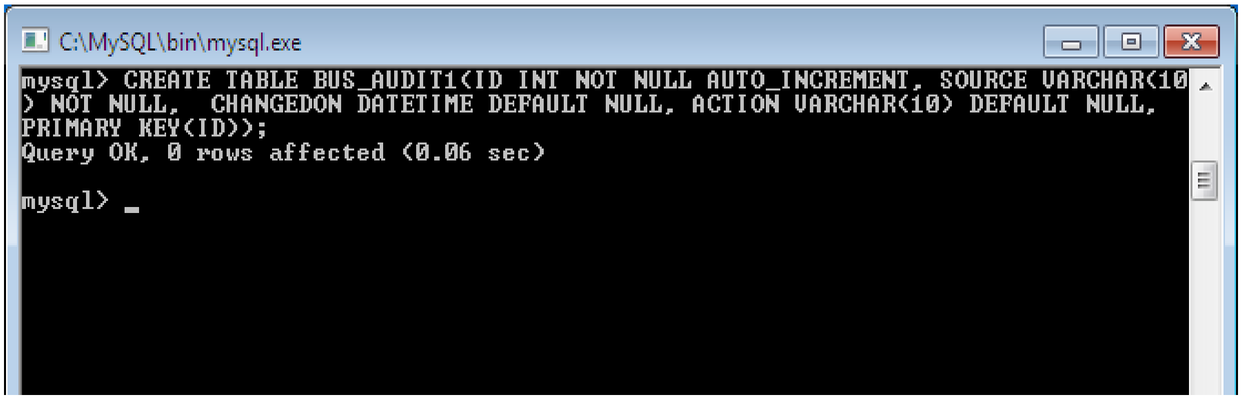
TRIGGER: Creation of insert trigger, delete trigger and update trigger.

MySQL>CREATE TABLE BUS(BUSNO VARCHAR(10) NOT NULL, SOURCE VARCHAR(10), DESTINATION VARCHAR(10), CAPACITY INT(2), PRIMARY KEY(BUSNO));
MySQL>INSERT INTO BUS VALUES('AP123','HYD','CHENNAI','40');



```
mysql> CREATE TABLE BUS(BUSNO VARCHAR(10) NOT NULL,
    ->          SOURCE VARCHAR(10),  DESTINATION VARCHAR(10),
    ->          CAPACITY INT(2), PRIMARY KEY(BUSNO));
Query OK, 0 rows affected (0.06 sec)

mysql> INSERT INTO BUS VALUES('AP123','HYD','CHENNAI','40');
Query OK, 1 row affected (0.02 sec)

mysql>
```

CREATE TABLE BUS_AUDIT1(ID INT NOT NULL AUTO_INCREMENT, SOURCE VARCHAR(10) NOT NULL, CHANGEDON DATETIME DEFAULT NULL, ACTION VARCHAR(10) DEFAULT NULL, PRIMARY KEY(ID));

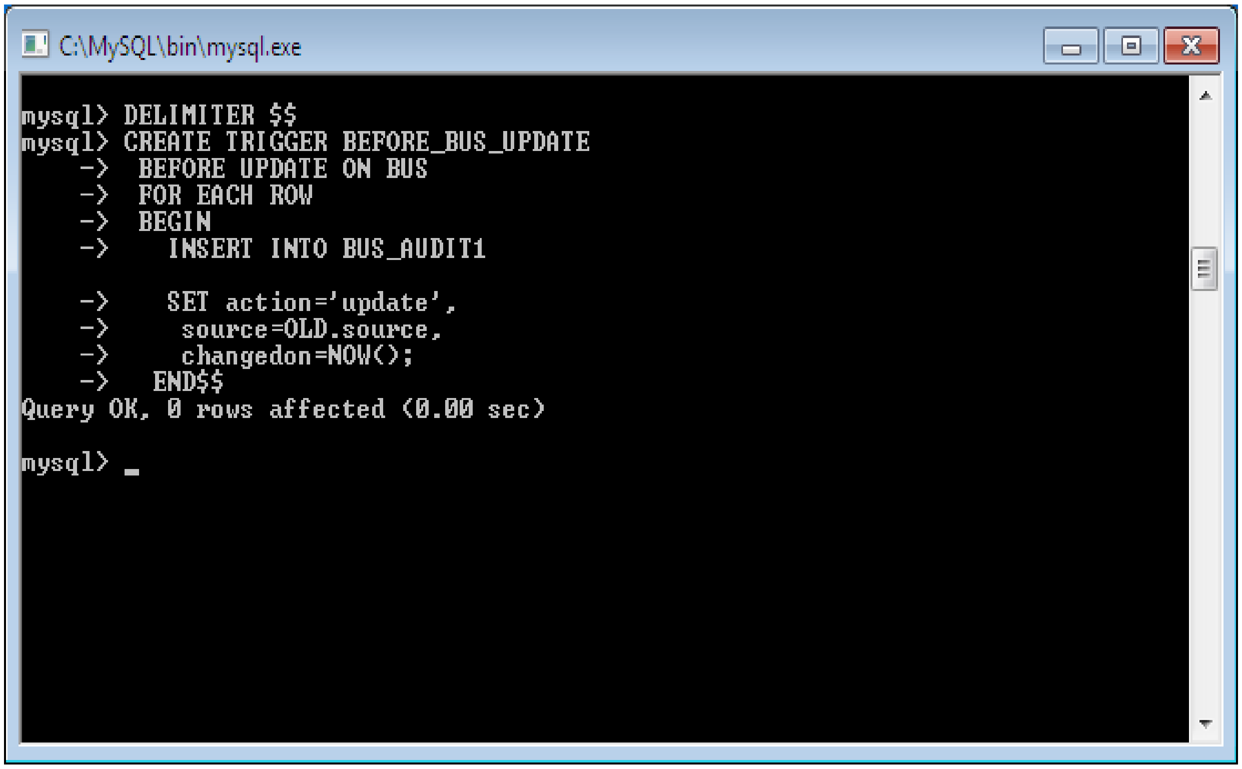


```
mysql> CREATE TABLE BUS_AUDIT1(ID INT NOT NULL AUTO_INCREMENT, SOURCE VARCHAR(10
) NOT NULL,  CHANGEDON DATETIME DEFAULT NULL, ACTION VARCHAR(10) DEFAULT NULL,
PRIMARY KEY(ID));
Query OK, 0 rows affected (0.06 sec)

mysql> _
```

CREATE TRIGGER BEFORE_BUS_UPDATE BEFORE UPDATE ON BUS FOR EACH ROW BEGIN

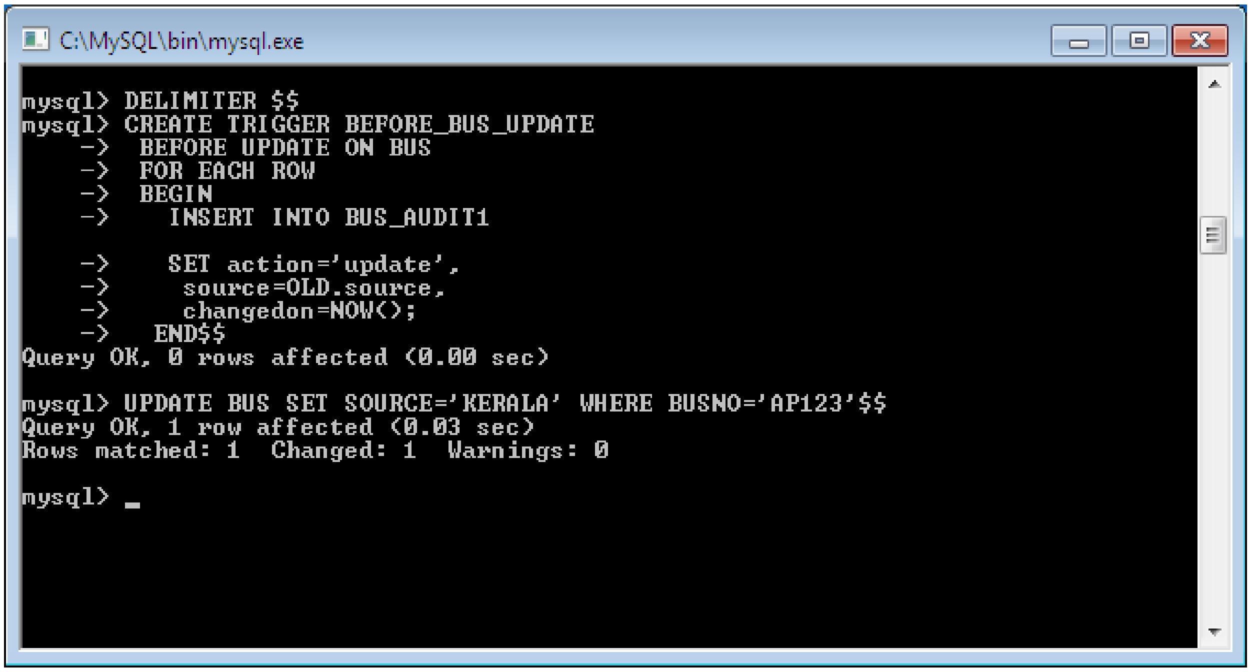
INSERT INTO BUS_AUDIT1

SET action='update', source=OLD.source, changedon=NOW(); END$$



```
mysql> DELIMITER $$
mysql> CREATE TRIGGER BEFORE_BUS_UPDATE
    ->  BEFORE UPDATE ON BUS
    ->  FOR EACH ROW
    ->  BEGIN
    ->    INSERT INTO BUS_AUDIT1

    ->    SET action='update',
    ->     source=OLD.source,
    ->     changedon=NOW();
    ->   END$$
Query OK, 0 rows affected (0.00 sec)

mysql> _
```

UPDATE :



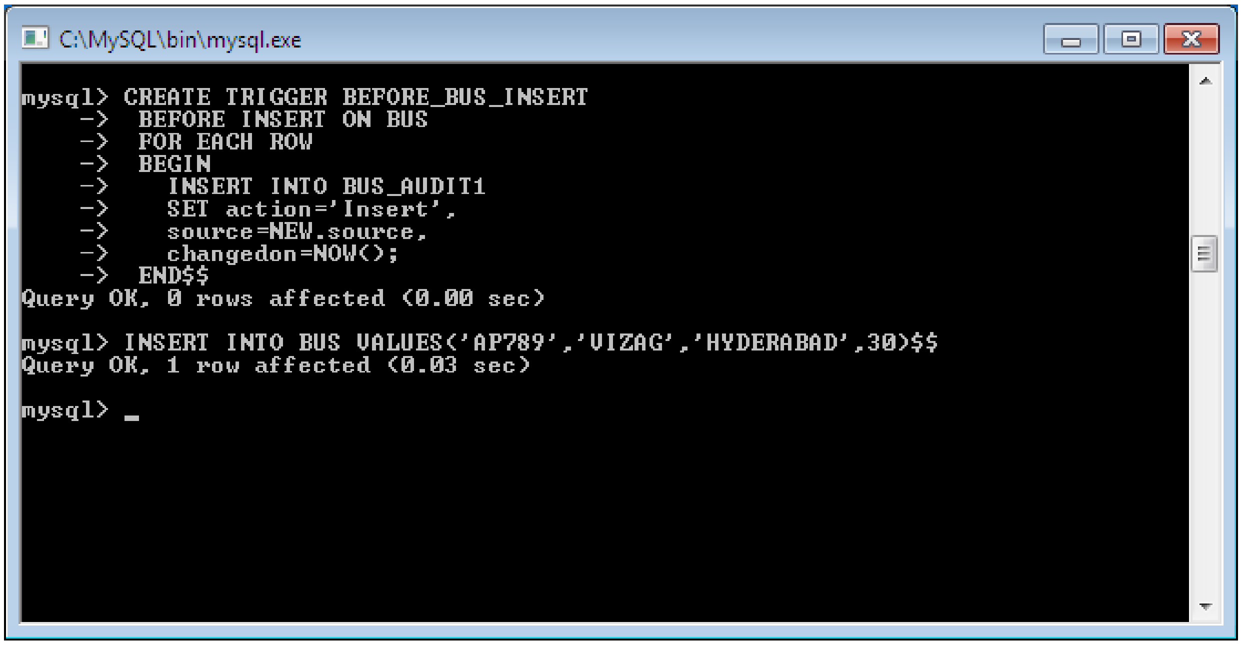
```
mysql> DELIMITER $$
mysql> CREATE TRIGGER BEFORE_BUS_UPDATE
    ->  BEFORE UPDATE ON BUS
    ->  FOR EACH ROW
    ->  BEGIN
    ->    INSERT INTO BUS_AUDIT1

    ->    SET action='update',
    ->     source=OLD.source,
    ->     changedon=NOW();
    ->   END$$
Query OK, 0 rows affected (0.00 sec)

mysql> UPDATE BUS SET SOURCE='KERALA' WHERE BUSNO='AP123'$$
Query OK, 1 row affected (0.03 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> _
```

MySQL>UPDATE BUS SET SOURCE='KERALA' WHERE BUSNO='AP123'$$

S.No.	Source	Changedon	Action
1	Banglore	2014:03:23 12:51:00	Insert
2	Kerela	2014:03:25:12:56:00	Update
3	Mumbai	2014:04:26:12:59:02	Delete

INSERT:
CREATE TRIGGER BEFORE_BUS_INSERT BEFORE INSERT ON BUS FOR EACH ROW BEGIN
INSERT INTO BUS_AUDIT1

S No	Source	Changedon	Action
1	Banglore	2014:03:23 12:51:00	Insert
2	Kerela	2014:03:25:12:56:00	Update
3	Mumbai	2014:04:26:12:59:02	Delete



```
mysql> CREATE TRIGGER BEFORE_BUS_INSERT
    ->  BEFORE INSERT ON BUS
    ->  FOR EACH ROW
    ->  BEGIN
    ->    INSERT INTO BUS_AUDIT1
    ->    SET action='Insert',
    ->    source=NEW.source,
    ->    changedon=NOW();
    ->  END$$
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO BUS VALUES('AP789','VIZAG','HYDERABAD',30)$$
Query OK, 1 row affected (0.03 sec)

mysql> _
```

SET action='Insert', source=NEW.source, changedon=NOW(); END$$

MYSQL>INSERT INTOBUS VALUES('AP789','VIZAG','HYDERABAD',30)$$

CREATE TRIGGER BEFORE_BUS_DELETE BEFORE DELETE ON BUS FOR EACH

ROWBEGIN

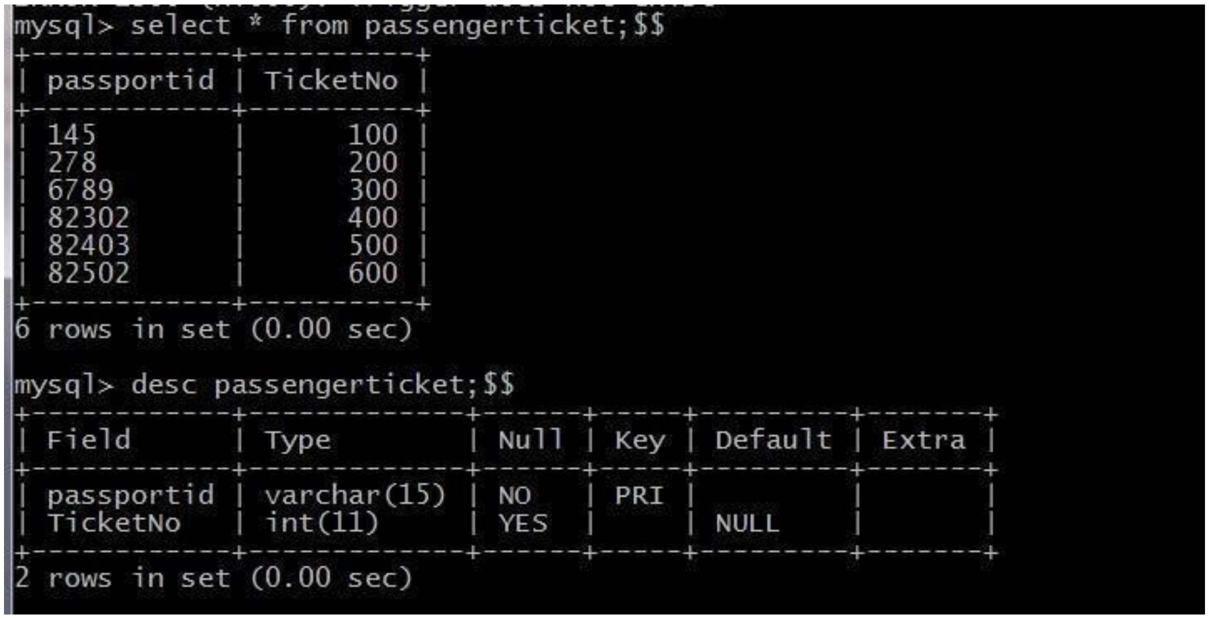
DELETE FROM BUS_AUDIT1
SET action='Insert', source=NEW.source, changedon=NOW(); END$$ DELETE FROM

BUSWHERE SOURCE=‘HYDERABAD‘$$

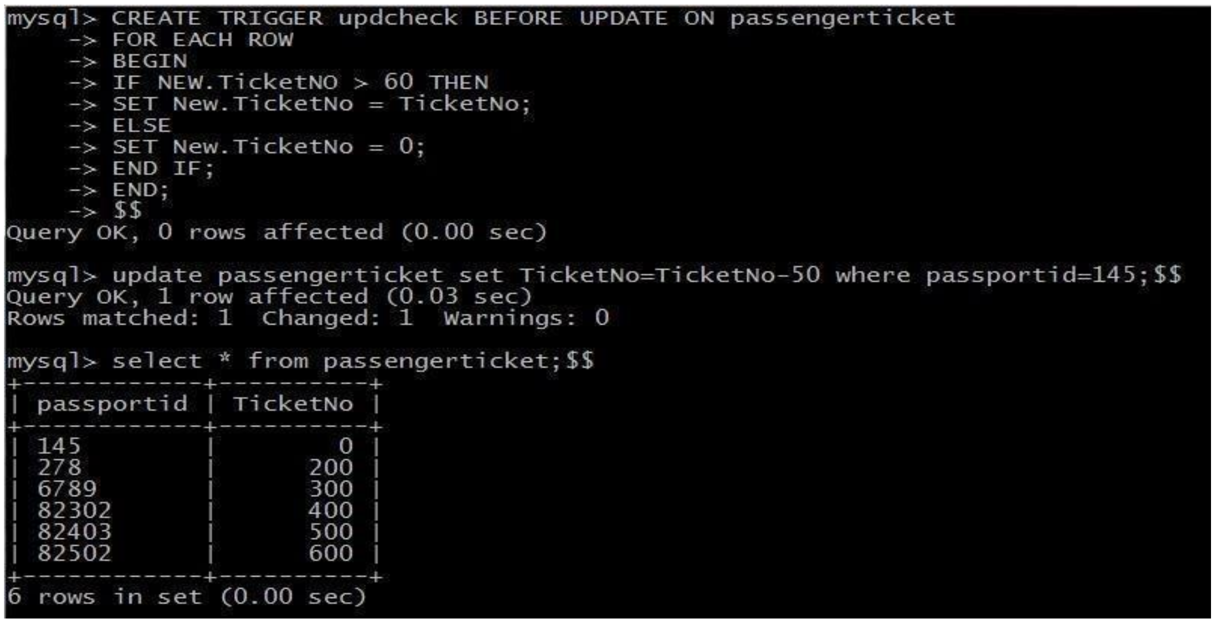
S. No.	Source	Changedon	Action
1	Banglore	2014:03:23 12:51:00	Insert
2	Kerela	2014:03:25:12:56:00	Update
3	Mumbai	2014:04:26:12:59:02	Delete

Examples:

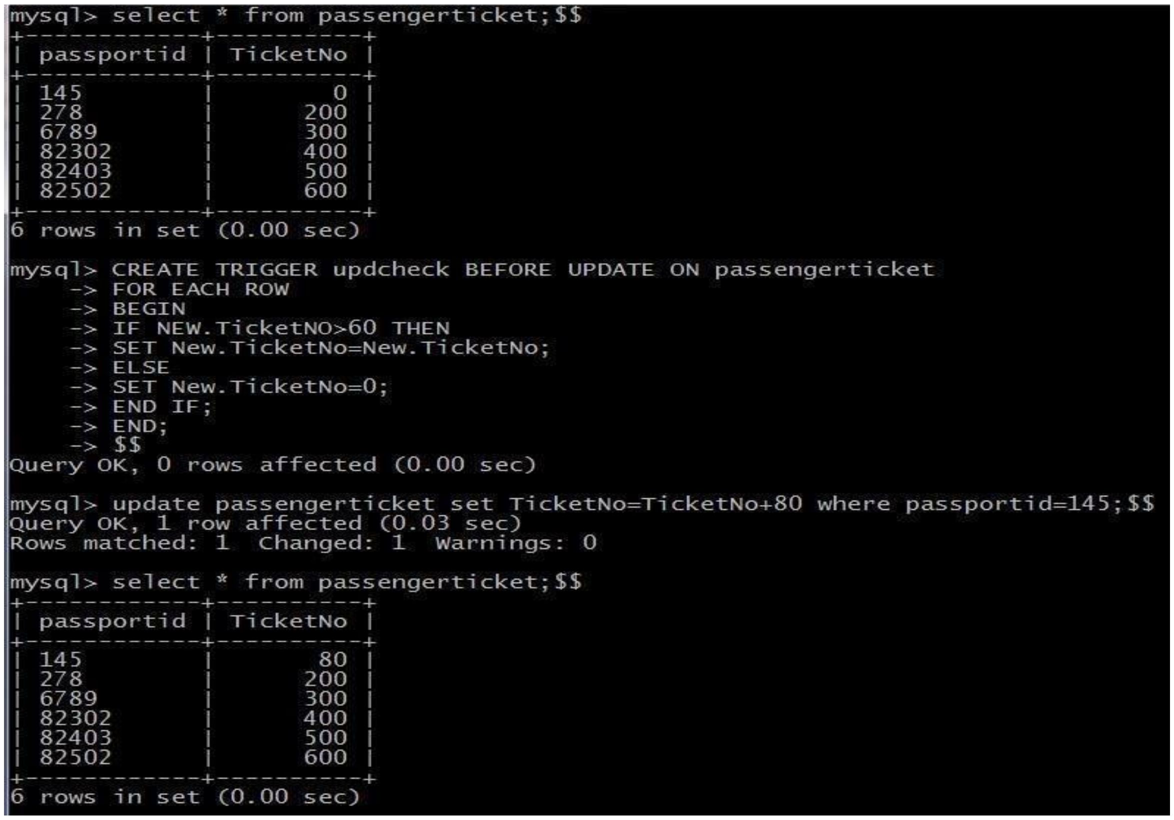
CREATE TRIGGER updcheck1 BEFORE UPDATE ON passengerticket FOR EACH
ROWBEGIN
IF NEW.TicketNO > 60 THEN
SET New.TicketNo = New.TicketNo; ELSE SET New.TicketNo = 0;END IF;
END;



```
mysql> select * from passengerticket;$$
+------------+----------+
| passportid | TicketNo |
+------------+----------+
| 145        |      100 |
| 278        |      200 |
| 6789       |      300 |
| 82302      |      400 |
| 82403      |      500 |
| 82502      |      600 |
+------------+----------+
6 rows in set (0.00 sec)

mysql> desc passengerticket;$$
+------------+-------------+------+-----+---------+-------+
| Field      | Type        | Null | Key | Default | Extra |
+------------+-------------+------+-----+---------+-------+
| passportid | varchar(15) | NO   | PRI |         |       |
| TicketNo   | int(11)     | YES  |     | NULL    |       |
+------------+-------------+------+-----+---------+-------+
2 rows in set (0.00 sec)
```



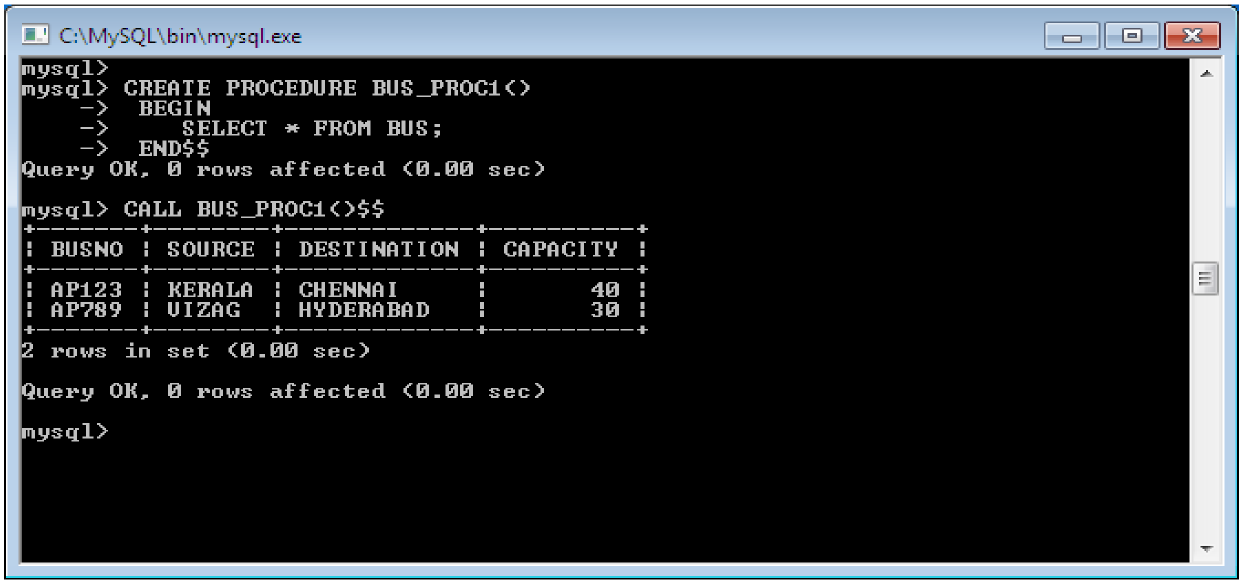
```
mysql> CREATE TRIGGER updcheck BEFORE UPDATE ON passengerticket
    -> FOR EACH ROW
    -> BEGIN
    -> IF NEW.TicketNO > 60 THEN
    -> SET New.TicketNo = TicketNo;
    -> ELSE
    -> SET New.TicketNo = 0;
    -> END IF;
    -> END;
    -> $$
Query OK, 0 rows affected (0.00 sec)

mysql> update passengerticket set TicketNo=TicketNo-50 where passportid=145;$$
Query OK, 1 row affected (0.03 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> select * from passengerticket;$$
+------------+----------+
| passportid | TicketNo |
+------------+----------+
| 145        |        0 |
| 278        |      200 |
| 6789       |      300 |
| 82302      |      400 |
| 82403      |      500 |
| 82502      |      600 |
+------------+----------+
6 rows in set (0.00 sec)
```



```
mysql> select * from passengerticket;$$
+------------+----------+
| passportid | TicketNo |
+------------+----------+
| 145        |        0 |
| 278        |      200 |
| 6789       |      300 |
| 82302      |      400 |
| 82403      |      500 |
| 82502      |      600 |
+------------+----------+
6 rows in set (0.00 sec)

mysql> CREATE TRIGGER updcheck BEFORE UPDATE ON passengerticket
    -> FOR EACH ROW
    -> BEGIN
    -> IF NEW.TicketNO>60 THEN
    -> SET New.TicketNo=New.TicketNo;
    -> ELSE
    -> SET New.TicketNo=0;
    -> END IF;
    -> END;
    -> $$
Query OK, 0 rows affected (0.00 sec)

mysql> update passengerticket set TicketNo=TicketNo+80 where passportid=145;$$
Query OK, 1 row affected (0.03 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> select * from passengerticket;$$
+------------+----------+
| passportid | TicketNo |
+------------+----------+
| 145        |       80 |
| 278        |      200 |
| 6789       |      300 |
| 82302      |      400 |
| 82403      |      500 |
| 82502      |      600 |
+------------+----------+
6 rows in set (0.00 sec)
```

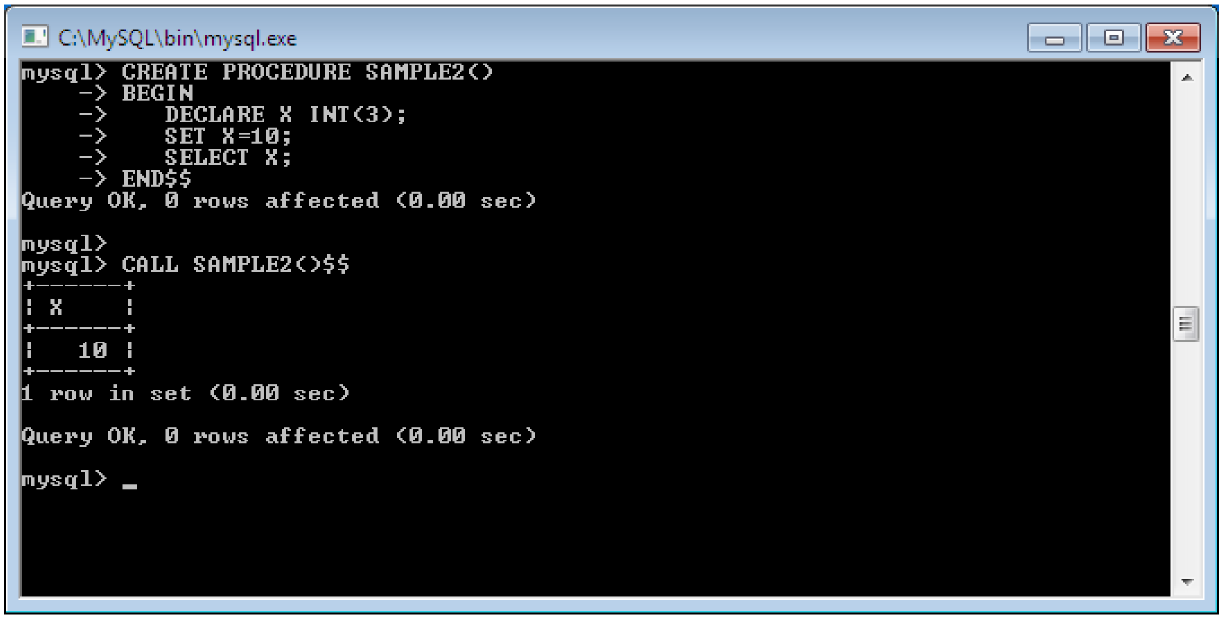
PROCEDURES: Creation of stored Procedures and Execution of Procedures and Modification ofProcedures.

Eg 1:
CREATE PROCEDURE BUS_PROC1() BEGIN SELECT * FROM BUS;
END$$
CALL BUS_PROC1()$$**Eg 2:**



```
mysql>
mysql> CREATE PROCEDURE BUS_PROC1()
    ->  BEGIN
    ->     SELECT * FROM BUS;
    ->  END$$
Query OK, 0 rows affected (0.00 sec)

mysql> CALL BUS_PROC1()$$
+-------+--------+-------------+----------+
| BUSNO | SOURCE | DESTINATION | CAPACITY |
+-------+--------+-------------+----------+
| AP123 | KERALA | CHENNAI     |       40 |
| AP789 | VIZAG  | HYDERABAD   |       30 |
+-------+--------+-------------+----------+
2 rows in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

mysql>
```

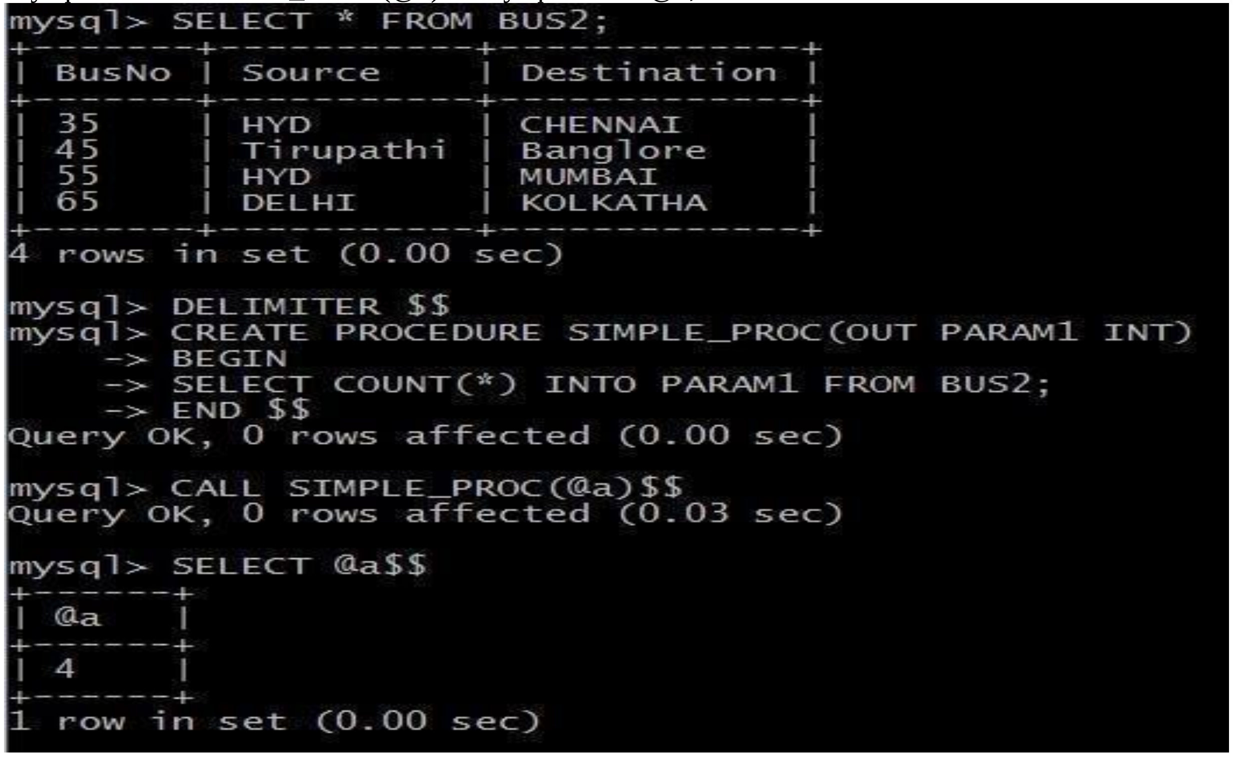
CREATE PROCEDURE SAMPLE2() BEGIN DECLARE X INT(3); SET X=10;SELECT X;
END$



```
mysql> CREATE PROCEDURE SAMPLE2()
    -> BEGIN
    ->     DECLARE X INT(3);
    ->     SET X=10;
    ->     SELECT X;
    -> END$$
Query OK, 0 rows affected (0.00 sec)

mysql>
mysql> CALL SAMPLE2()$$
+-------+
| X     |
+-------+
|   10  |
+-------+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

mysql> _
```

Mysql> CALL SAMPLE2()$$

Ex3:
CREATE PROCEDURE SIMPLE_PROC(OUT PARAM1 INT) BEGIN SELECT COUNT(*)
INTOPARAM1 FROM BUS;
END$$
Mysql> CALL SIMPLE_PROC(@a)$$ Mysql> select @a;



```
mysql> SELECT * FROM BUS2;
+-------+-----------+-------------+
| BusNo | Source    | Destination |
+-------+-----------+-------------+
| 35    | HYD       | CHENNAI     |
| 45    | Tirupathi | Banglore    |
| 55    | HYD       | MUMBAI      |
| 65    | DELHI     | KOLKATHA    |
+-------+-----------+-------------+
4 rows in set (0.00 sec)

mysql> DELIMITER $$
mysql> CREATE PROCEDURE SIMPLE_PROC(OUT PARAM1 INT)
    -> BEGIN
    -> SELECT COUNT(*) INTO PARAM1 FROM BUS2;
    -> END $$
Query OK, 0 rows affected (0.00 sec)

mysql> CALL SIMPLE_PROC(@a)$$
Query OK, 0 rows affected (0.03 sec)

mysql> SELECT @a$$
+------+
| @a   |
+------+
| 4    |
+------+
1 row in set (0.00 sec)
```

Ex. No. :	DATABASE DESIGN USING NORMALIZATION
Date :	

AIM: Apply the database Normalization techniques for designing relational database tables to minimize duplication of information like 1NF, 2NF, 3NF, BCNF and for creating relationship between the databases as tables using SQL commands.

PROCEDURE

Normalization is a database design technique which organizes tables in a manner that reducesredundancy and dependency of data.

It divides larger tables to smaller tables and links them using relationships.

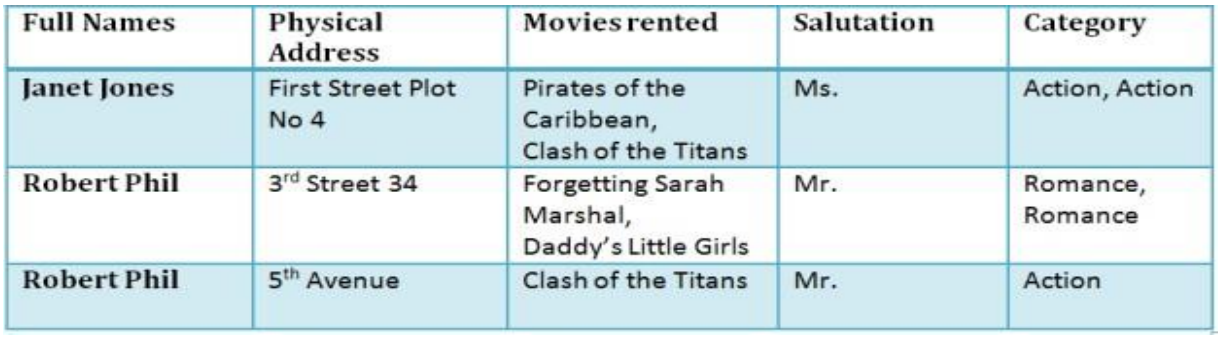
The inventor of the relational model Edgar Codd proposed the theory of normalization with the introduction of First Normal Form, and he continued to extend theory with Second and Third Normal Form. Later he joined with Raymond F. Boyce to develop the theory of Boyce-Codd Normal Form.

Theory of Data Normalization in SQL is still being developed further. For example, there are discussions even on 6thNormal Form. **However, in most practical applications, normalization achieves its best in 3rd Normal Form**. The evolution of Normalization theories is illustrated below-

1st Normal Form
2nd Normal Form
3rd Normall Form
Boyce-Codd NF
4th Normal Form
5th Normal Form
6th Normal Form

Database Normalization Examples -

Assume a video library maintains a database of movies rented out. Without any normalization, allinformation is stored in one table as shown below.



Full Names	Physical Address	Movies rented	Salutation	Category
Janet Jones	First Street Plot No 4	Pirates of the Caribbean, Clash of the Titans	Ms.	Action, Action
Robert Phil	3rd Street 34	Forgetting Sarah Marshal, Daddy's Little Girls	Mr.	Romance, Romance
Robert Phil	5th Avenue	Clash of the Titans	Mr.	Action

Table 1

Here you see Movies Rented column has multiple values. Database Normal Forms

Now let's move into 1st Normal Forms

1NF (First Normal Form) Rules

- Each table cell should contain a single value.
- Each record needs to be unique.

The above table in 1NF-*1NF Example*

FULL NAMES	PHYSICAL ADDRESS	MOVIES RENTED	SALUTATION
Janet Jones	First Street Plot No 4	Pirates of the Caribbean	Ms.
Janet Jones	First Street Plot No 4	Clash of the Titans	Ms.
Robert Phil	3rd Street 34	Forgetting Sarah Marshal	Mr.
Robert Phil	3rd Street 34	Daddy's Little Girls	Mr.
Robert Phil	5th Avenue	Clash of the Titans	Mr.

Table 1: In 1NF Form

Before we proceed let's understand a few things --

What is a KEY?

A KEY is a value used to identify a record in a table uniquely. A KEY could be a single column orcombination of multiple columns

Note: Columns in a table that are NOT used to identify a record uniquely are called non-keycolumns.
What is a Primary Key?

A primary is a single column value used to identify a database recorduniquely.

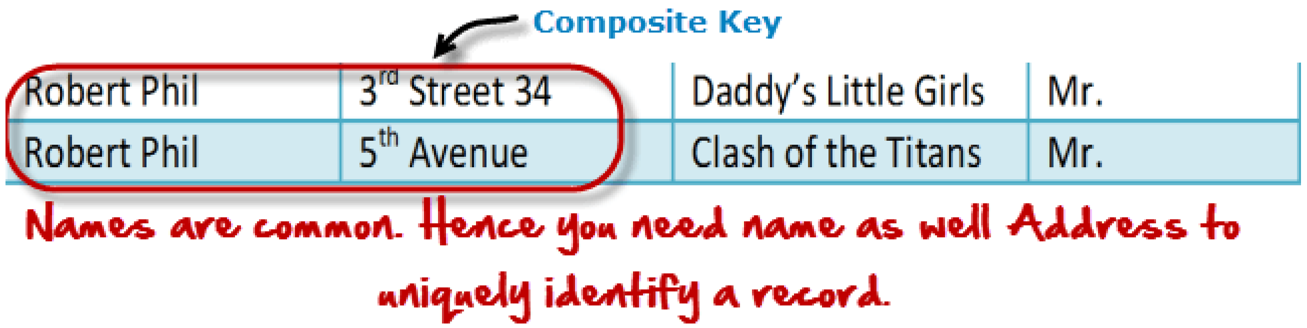
It has following attributes

A primary key cannot be NULL
- ✓ A primary key value must be unique
- ✓ The primary key values cannot be changed
- ✓ The primary key must be given a value when a new record is inserted.

What is Composite Key?

A composite key is a primary key composed of multiple columns used to identify a recorduniquely

In our database, we have two people with the same name Robert Phil, but they live in differentplaces.


Composite Key

Robert Phil	3rd Street 34	Daddy's Little Girls	Mr.
Robert Phil	5th Avenue	Clash of the Titans	Mr.

Names are common. Hence you need name as well Address to uniquely identify a record.

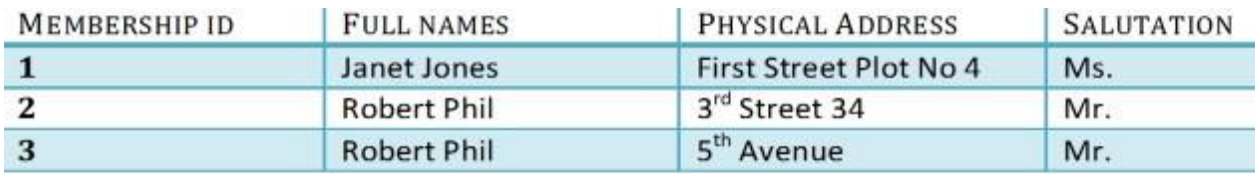
Hence, we require both Full Name and Address to identify a record uniquely. That is a compositekey.

Let's move into second normal form 2NF

2NF (Second Normal Form) Rules

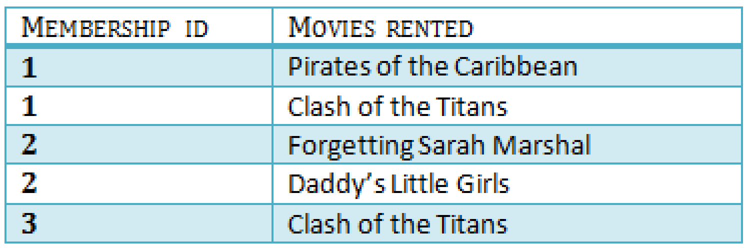
- Rule 1- Be in 1NF
- Rule 2- Single Column Primary Key

It is clear that we can't move forward to make our simple database in 2nd Normalization formunless we partition the table above.



MEMBERSHIP ID	FULL NAMES	PHYSICAL ADDRESS	SALUTATION
1	Janet Jones	First Street Plot No 4	Ms.
2	Robert Phil	3rd Street 34	Mr.
3	Robert Phil	5th Avenue	Mr.

Table 1



MEMBERSHIP ID	MOVIES RENTED
1	Pirates of the Caribbean
1	Clash of the Titans
2	Forgetting Sarah Marshal
2	Daddy's Little Girls
3	Clash of the Titans

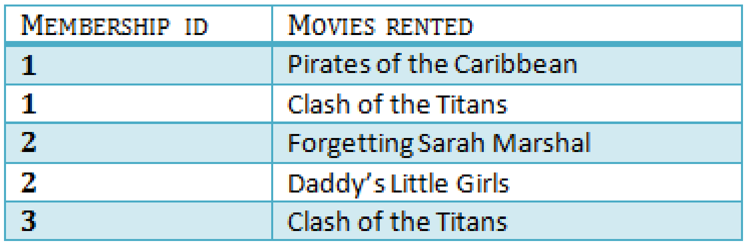
Table 2

We have divided our 1NF table into two tables viz. Table 1 and Table2. Table 1 contains memberinformation. Table 2 contains information on movies rented.

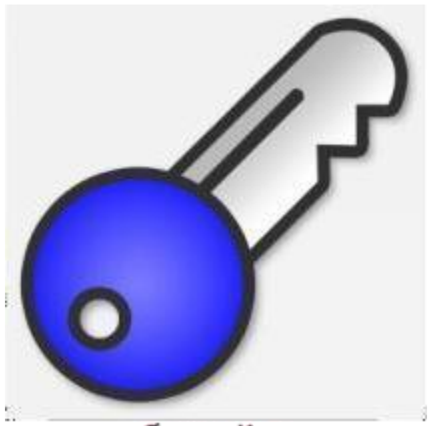
We have introduced a new column called Membership_id which is the primary key for table 1.Records can be uniquely identified in Table 1 using membership id.

Database - Foreign Key

In Table 2, Membership_ID is the Foreign Key



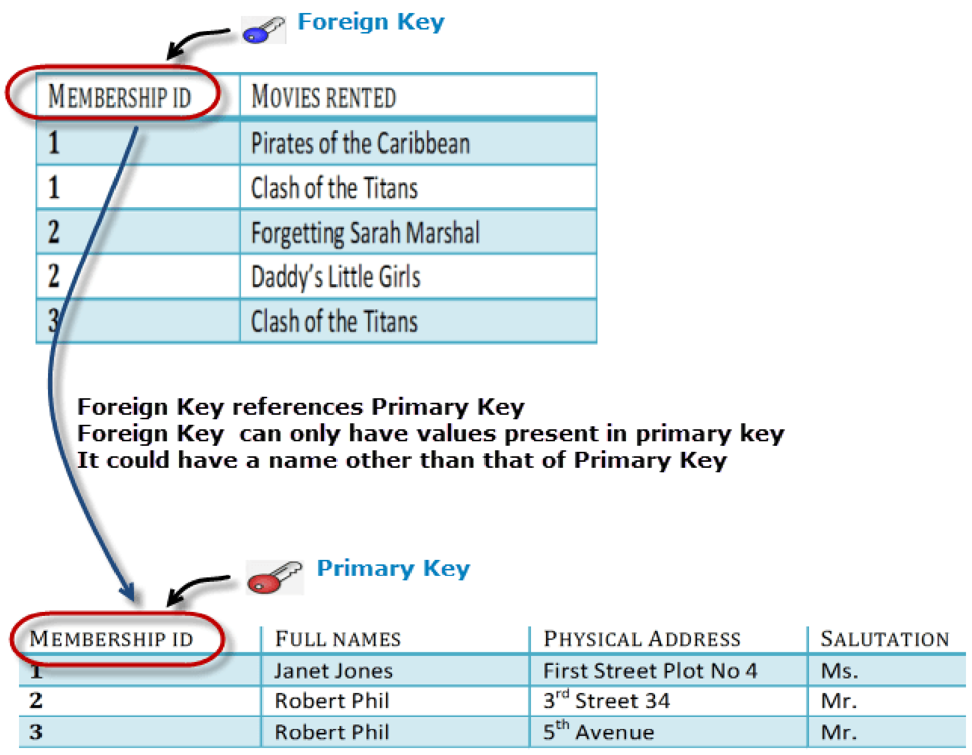
Membership ID	Movies rented
1	Pirates of the Caribbean
1	Clash of the Titans
2	Forgetting Sarah Marshal
2	Daddy's Little Girls
3	Clash of the Titans



Foreign Key

Foreign Key references the primary key of another Table! Ithelps connect your Tables

- A foreign key can have a different name from its primary key
- It ensures rows in one table have corresponding rowsin another
- Unlike the Primary key, they do not have to be unique. Most often they aren't
- Foreign keys can be null even though primary keys can not



Membership ID	Movies rented
1	Pirates of the Caribbean
1	Clash of the Titans
2	Forgetting Sarah Marshal
2	Daddy's Little Girls
3	Clash of the Titans

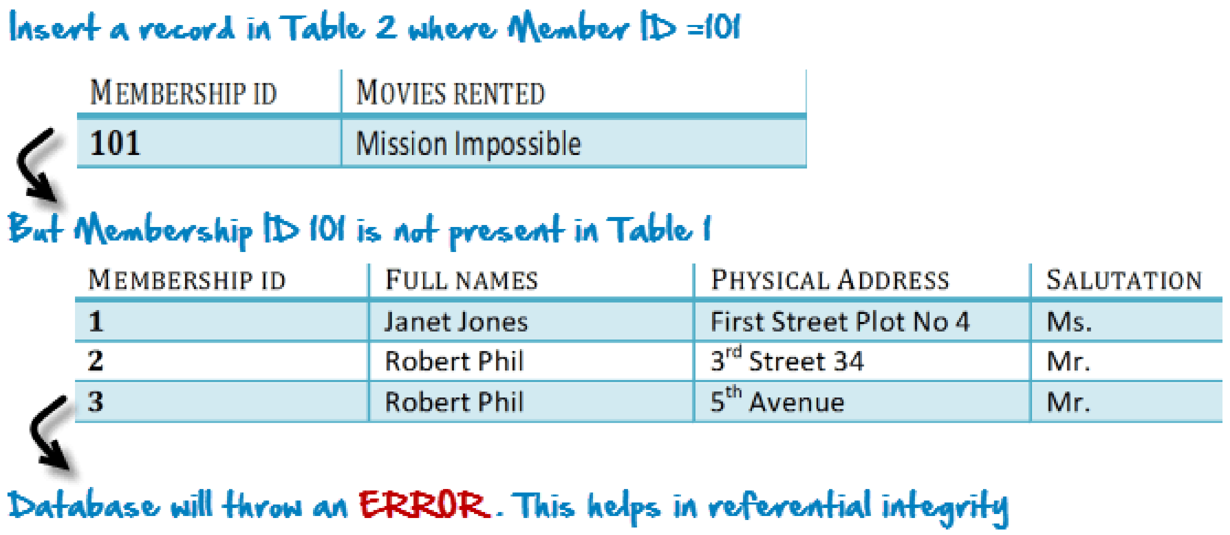
Foreign Key references Primary Key
Foreign Key can only have values present in primary key
It could have a name other than that of Primary Key

Membership ID	Full Names	Physical Address	Salutation
1	Janet Jones	First Street Plot No 4	Ms.
2	Robert Phil	3rd Street 34	Mr.
3	Robert Phil	5th Avenue	Mr.

Why do you need a foreign key?

Suppose an idiot inserts a record in Table B such as

You will only be able to insert values into your foreign key that exist in the unique key in theparent table. This helps in referential integrity.


Insert a record in Table 2 where Member ID =101
But Membership ID 101 is not present in Table 1
Database will throw an ERROR. This helps in referential integrity

MEMBERSHIP ID	MOVIES RENTED
101	Mission Impossible

MEMBERSHIP ID	FULL NAMES	PHYSICAL ADDRESS	SALUTATION
1	Janet Jones	First Street Plot No 4	Ms.
2	Robert Phil	3rd Street 34	Mr.
3	Robert Phil	5th Avenue	Mr.

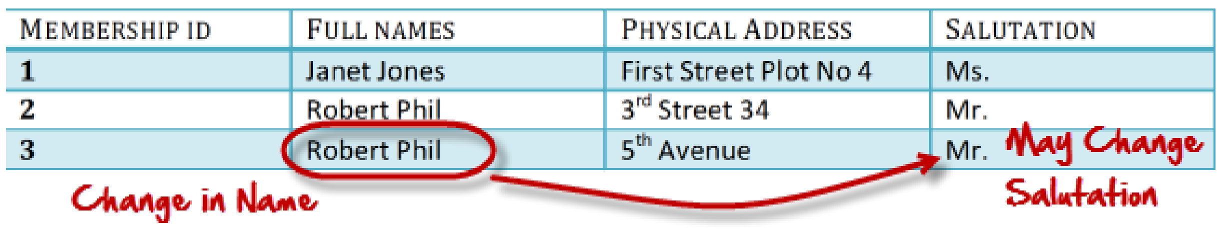
The above problem can be overcome by declaring membership id from Table2 as foreign key ofmembership id from Table1

Now, if somebody tries to insert a value in the membership id field that does not exist in the parenttable, an error will be shown!

What are transitive functional dependencies?

A transitive functional dependency is when changing a non-key column, might cause any of theother non-key columns to change

Consider the table 1. Changing the non-key column Full Name may change Salutation.


Change in Name
May Change Salutation

MEMBERSHIP ID	FULL NAMES	PHYSICAL ADDRESS	SALUTATION
1	Janet Jones	First Street Plot No 4	Ms.
2	Robert Phil	3rd Street 34	Mr.
3	Robert Phil	5th Avenue	Mr.

Let's move into 3NF

3NF (Third Normal Form) Rules

- Rule 1- Be in 2NF
- Rule 2- Has no transitive functional dependencies

To move our 2NF table into 3NF, we again need to again divide our table.

3NF Example

MEMBERSHIP ID	FULL NAMES	PHYSICAL ADDRESS	SALUTATION ID
1	Janet Jones	First Street Plot No 4	2
2	Robert Phil	3rd Street 34	1
3	Robert Phil	5th Avenue	1

TABLE 1

MEMBERSHIP ID	MOVIES RENTED
1	Pirates of the Caribbean
1	Clash of the Titans
2	Forgetting Sarah Marshal
2	Daddy's Little Girls
3	Clash of the Titans

Table 2

SALUTATION ID	SALUTATION
1	Mr.
2	Ms.
3	Mrs.
4	Dr.

Table 3

We have again divided our tables and created a new table which stores Salutations. There are no transitive functional dependencies, and hence our table is in 3NF

In Table 3 Salutation ID is primary key, and in Table 1 Salutation ID is foreign to primary key in
Table 3

Now our little example is at a level that cannot further be decomposed to attain higher forms of normalization. In fact, it is already in higher normalization forms. Separate efforts for moving into next levels of normalizing data are normally needed in complex databases. However, we will be discussing next levels of normalizations in brief in the following.

Boyce-Codd Normal Form (BCNF)

Even when a database is in 3rd Normal Form, still there would be anomalies resulted if it has morethan one **Candidate** Key.

Sometimes is BCNF is also referred as **3.5 Normal Form. 4NF (Fourth Normal Form) Rules**

If no database table instance contains two or more, independent and multivalued data describingthe relevant entity, then it is in 4th Normal Form.

5NF (Fifth Normal Form) Rules

A table is in 5th Normal Form only if it is in 4NF and it cannot be decomposed into any number ofsmaller tables without loss of data.

6NF (Sixth Normal Form) Proposed

6th Normal Form is not standardized, yet however, it is being discussed by database experts for some time. Hopefully, we would have a clear & standardized definition for 6th Normal Form in thenear future...

CREATING RELATIONSHIP BETWEEN THE DATABASES

Database Normalizations :

Database Normalizations is a technique of organizing the data in the database. Normalization is a systematic approach of decomposing tables to eliminate data redundancy and undesirable characteristics like Insertion, Update and Deletion Anomalies.
Normalization is used for mainly two purpose,

- Eliminating redundant(useless) data.
- Ensuring data dependencies make sense i.e data is logically stored.Table Name : **Student**

S_id	**S_Name**	**S_Address**	**Subject_opted**
401	Adam	Noida	Bio
402	Alex	Panipat	Maths
403	Stuart	Jammu	Maths
404	Adam	Noida	Physics

Problem Without Normalization :
Insertion, Updation and Deletion Anamolies are very frequent if Database is not Normalized.

- **Updation Anamoly :** To update address of a student who occurs twice or more than twice in a table, we will have to update **S_Address** column in all the rows, else data will become inconsistent.
- **Insertion Anamoly :** Suppose for a new admission, we have a Student id(S_id), name and address of a student but if student has not opted for any subjects yet then we have to insert **NULL** there, leading to Insertion Anamoly.
- **Deletion Anamoly :** If (S_id) 401 has only one subject and temporarily he drops it, when we delete that row, entire student record will be deleted along with it.

There are many approaches to the design of a database as given below:

- *Bottom-up database approach:* This approach starts fundamental level of attributes, that is properties of the entities and relationships. It then combines or add to these abstractions, which are grouped into relations that represent types of entities and relationships between entities. New relations among entity types may be added as the design progresses.
- *Top-down database approach:* This approach commences with the development of the data models that contains high level abstractions. Then the successive top-down refinement are applied to identify lower-level entities, relationships and the associated attributes. The Entity Relationshipmodel is an example of top-down approach and is more suitable for the design of complex databases.
- *Inside-out database design approach:* This approach begins with the identification of set of major entities and then spreading out to consider other entities, relationships and attributes associated with those first identified. The inside-out database design approach is special case of a bottom-up approach, where attention is focused at a central set of concepts that are most evident and then spreading outward by considering others in the vicinity of existing ones.
- *Mixed strategy database design approach:* This approach uses both the bottom-up and top-down approach instead of following any particular approach for various parts of the data model before finally combining all parts together. In this case, the requirements are partitioned according to a top-down approach and part of the schema is each partition according to a bottom-up approach.

Normalization Rule

Normalization rule are divided into following normal form.

1. First Normal Form - **Tables Must Not Contain Repeating Groups Of Data**
2. Second Normal Form - **Eliminations of Redundant data**
3. Third Normal Form - **Eliminate columns Not Dependant on the key .**
4. Fourth Normal Form - **Isolate Independent Multiple Relations .**
5. BCNF

First Normal Form (1NF)

- As per First Normal Form, no two Rows of data must contain repeating group of information i.e each set of column must have a unique value, such that multiple columns cannot be used to fetch the same row.
- It should hold only atomic(Single) values.

Example: Suppose a company wants to store the names and contact details of its employees. It creates a table that looks like this:

Emp_id	Emp_name	Emp_address	Emp_mobile
101	Herschel	New Delhi	8912312390
102	Jon	Kanpur	8812121212 9900012222
103	Ron	Chennai	7778881212
104	Lester	Bangalore	9990000123 8123450987

Two employees (Jon & Lester) are having two mobile numbers so the company stored them in the same field as you can see in the table above.

This table is **not in 1NF** as the rule says –each attribute of a table must have atomic (single) values‖, the emp_mobile values for employees Jon & Lester violates that rule.

To make the table complies with 1NF we should have the data like this:

Emp_id	**Emp_name**	**Emp_address**	**Emp_mobile**
101	Herschel	New Delhi	8912312390
102	Jon	Kanpur	8812121212
102	Jon	Kanpur	9900012222
103	Ron	Chennai	7778881212
104	Lester	Bangalore	9990000123
104	Lester	Bangalore	8123450987

Second normal form (2NF)

A table is said to be in 2NF if both the following conditions hold:

- **Table is in 1NF (First normal form)**
- **No non-prime attribute is dependent on the proper subset of any candidate key of table.**

An attribute that is not part of any candidate key is known as non-prime attribute.

Example: Suppose a school wants to store the data of teachers and the subjects they teach. They create a table that looks like this: Since a teacher can teach more than one subjects, the table can have multiple rows for a same teacher.

Teacher_id	**Subject**	**Teacher_age**
111	Maths	38
111	Physics	38
222	Biology	38
333	Physics	40
333	Chemistry	40

Candidate Keys:{teacher_id,subject}**Non prime attribute**: teacher_age

The table is in 1 NF because each attribute has atomic values. However, it is not in 2NF because non prime attribute teacher_age is dependent on teacher_id alone which is a proper subset of candidate key.

This violates the rule for 2NF as the rule says –**no** non-prime attribute is dependent on the proper subset of any candidate key of the table‖.

To make the table complies with 2NF we can break it in two tables like this:

teacher_details table:

Teacher_id	**Teacher_age**
111	38
222	38
333	40

teacher_subject table:

Teacher_id	Subject
111	Maths
111	Physics
222	Biology
333	Physics
333	Chemistry

Now the tables comply with Second normal form (2NF).

Third Normal form (3NF)

A table design is said to be in 3NF if both the following conditions hold:

- Table must be in 2NF
- **Transitive functional dependency** of non-prime attribute on any super key should be removed.

An attribute that is not part of any **candidate key** is known as non-prime attribute.

In other words 3NF can be explained like this: A table is in 3NF if it is in 2NF and for eachfunctional dependency X-> Y at least one of the following conditions hold:

- X is a **super key** of table
- Y is a prime attribute of table

An attribute that is a part of one of the candidate keys is known as prime attribute.

Example: Suppose a company wants to store the complete address of each employee, they create atable named employee_details that looks like this:

Emp_id	**Emp_name**	**Emp_zip**	**Emp_state**	**Emp_city**	**Emp_district**
1001	John	282005	UP	Agra	Dayal Bagh
1002	Ajeet	222008	TN	Chennai	M-City
1006	Lora	282007	TN	Chennai	Urrapakkam
1101	Lilly	292008	UK	Pauri	Bhagwan
1201	Steve	222999	MP	Gwalior	Ratan

Super keys: {emp_id}, {emp_id, emp_name}, {emp_id, emp_name, emp_zip}…so on
Candidate Keys: {emp_id}

Non-prime attributes: all attributes except emp_id are non-prime as they are not part of any candidate keys.

Here, emp_state, emp_city & emp_district dependent on emp_zip. And, emp_zip is dependent on emp_id that makes non-prime attributes (emp_state, emp_city & emp_district) transitively dependent on super key (emp_id). This violates the rule of 3NF.

To make this table complies with 3NF we have to break the table into two tables to remove the transitive dependency:

Employee table:

Emp_id	**Emp_name**	**Emp_zip**
1001	John	282005
1002	Ajeet	222008
1006	Lora	282007
1101	Lilly	292008
1201	Steve	222999

Employee_zip table:

Emp_zip	**Emp_state**	**Emp_city**	**Emp_district**
282005	UP	Agra	Dayal Bagh
222008	TN	Chennai	M-City
282007	TN	Chennai	Urrapakkam
292008	UK	Pauri	Bhagwan
222999	MP	Gwalior	Ratan

Boyce Codd normal form (BCNF)

It is an advance version of 3NF that‟s why it is also referred as 3.5NF. BCNF is stricter than 3NF. A table complies with BCNF if it is in 3NF and for every **functional dependency** X->Y, X should be the super key of the table.

Example: Suppose there is a company wherein employees work in **more than one department**. They store the data like this:

Emp_id	**Emp_nationality**	**Emp_dept**	**Dept_type**	**Dept_no_of_emp**
1001	Austrian	Production and planning	D001	200
1001	Austrian	stores	D001	250
1002	American	design and technical support	D134	100
1002	American	Purchasing department	D134	600

Functional dependencies in the table above:emp_id -> emp_nationality

emp_dept -> {dept_type, dept_no_of_emp}

Candidate key: {emp_id, emp_dept}

The table is not in BCNF as neither emp_id nor emp_dept alone are keys.

To make the table comply with BCNF we can break the table in three tables like this:

Emp_Nationality Table:

Emp_id	**Emp_nationality**
1001	Austrian
1002	American

emp_dept table:

Emp_dept	**Dept_type**	**Dept_no_of_emp**
Production and planning	D001	200
stores	D001	250
design and technical support	D134	100
Purchasing department	D134	600

emp_dept_mapping table:

Emp_id	**Emp_dept**
1001	Production and planning
1001	stores
1002	design and technical support
1002	Purchasing department

Functional dependencies: emp_id -> emp_nationality

emp_dept -> {dept_type, dept_no_of_emp}

Candidate keys:

For first table: emp_id

For second table: emp_dept

For third table: {emp_id, emp_dept}

This is now in BCNF as in both the functional dependencies left side part is a key.

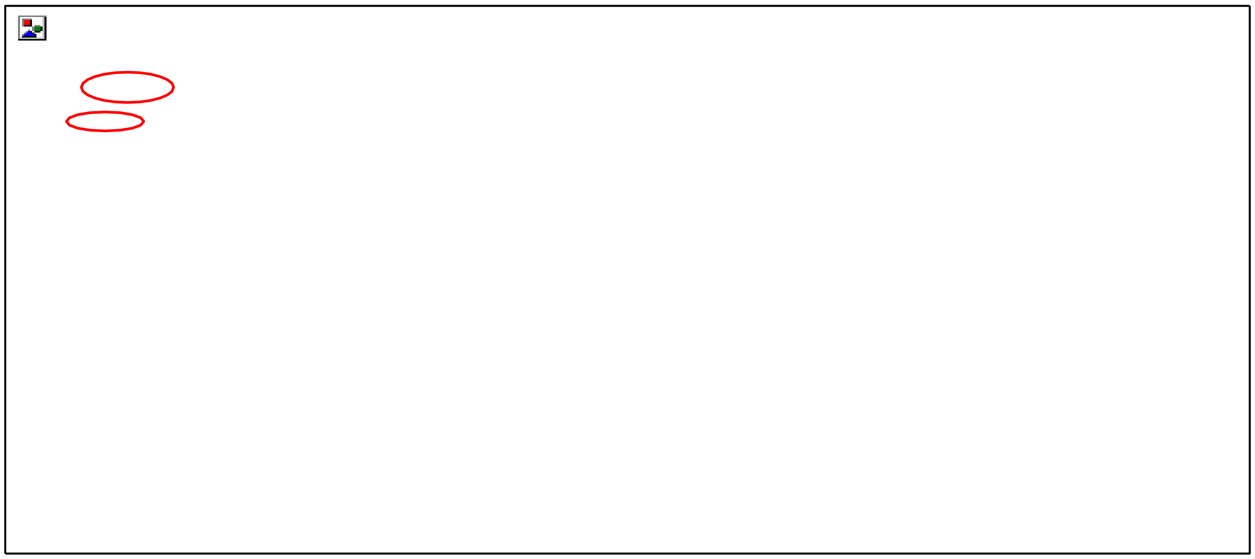
Ex. No. :	**DEVELOPMENT OF DATABASE APPLICATIONS**
Date :	

AIM

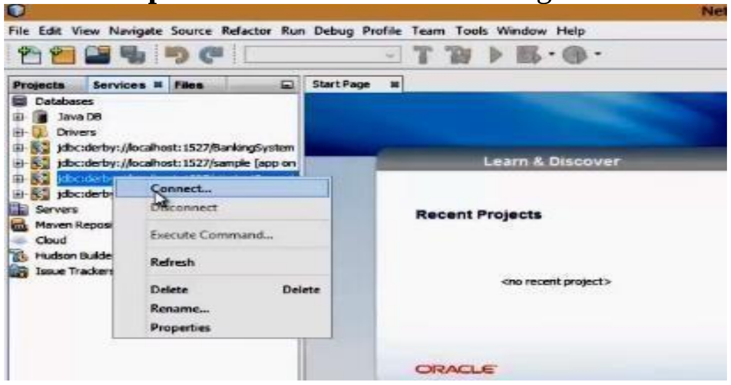
To develop a database application using NetBeans.

IMPLEMENTION

Creating Application using Netbeans
Step 1: click services > database > java DB > right click > create database. Then create java DBdatabase box is popped.
Step 2: give database name > give user name as PECAI&DS > set password as PECAI&DS > clickOK button.

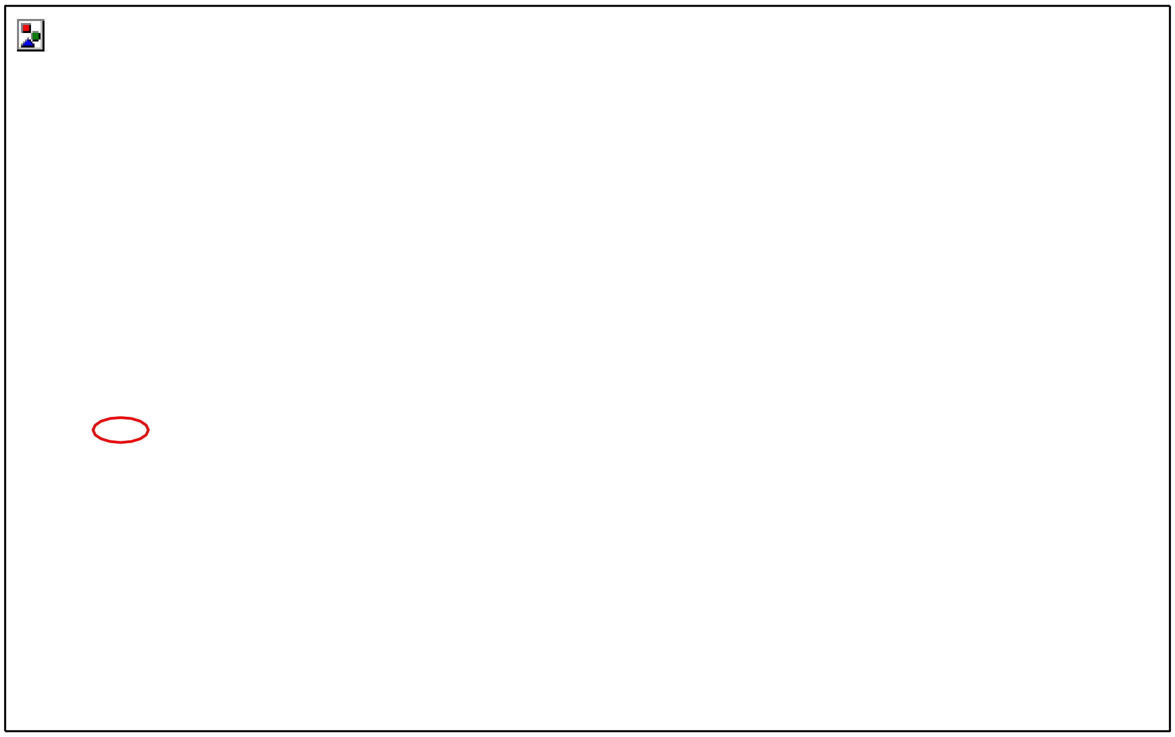


The java DB database will start to process. Once the process is stopped, the java DB will be created.**Step 3:** select the created one > right click > connect



Once you are connected, you can see the database list.

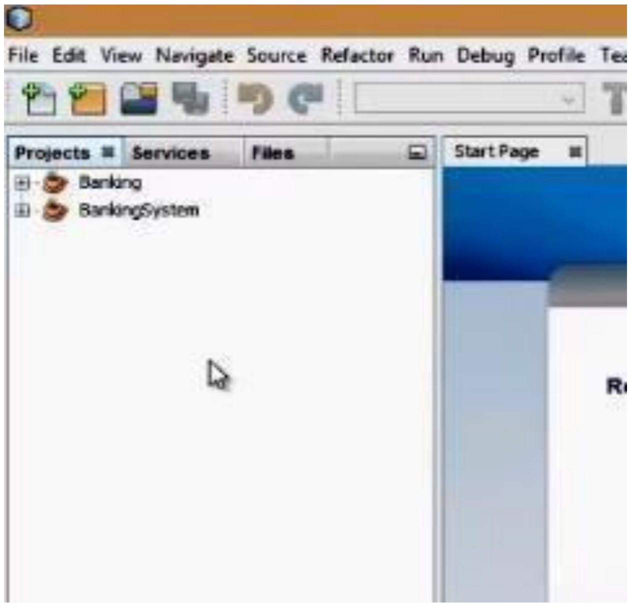
Step 4: select test > tables > create table > give table name



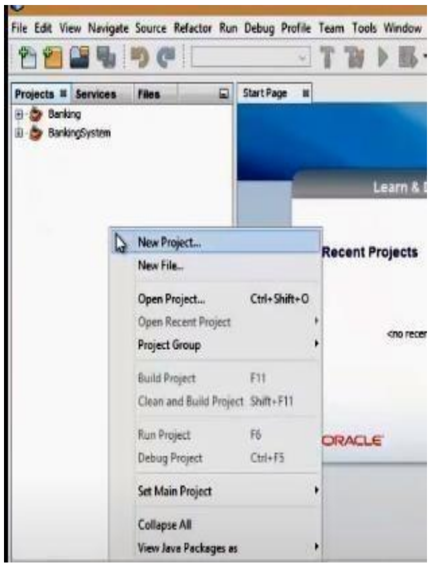
Step 5: click add column button > give name > select type > give size > then click OK button

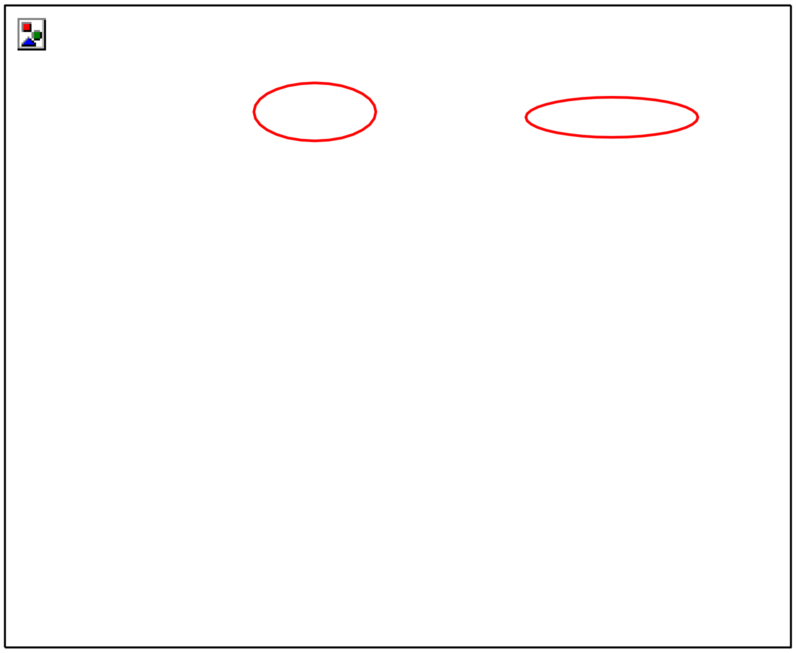
If you want more columns, you can repeat this step 5 and add columns.Then the table will be created.

Step 6: at the top corner, select project. In the empty space under project, right click.

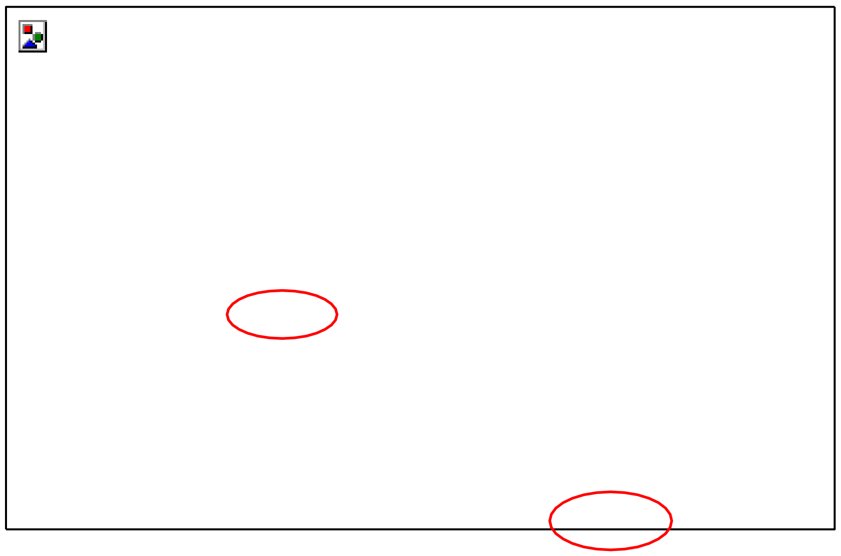


Step 7: select new project > java > java application > click NEXT



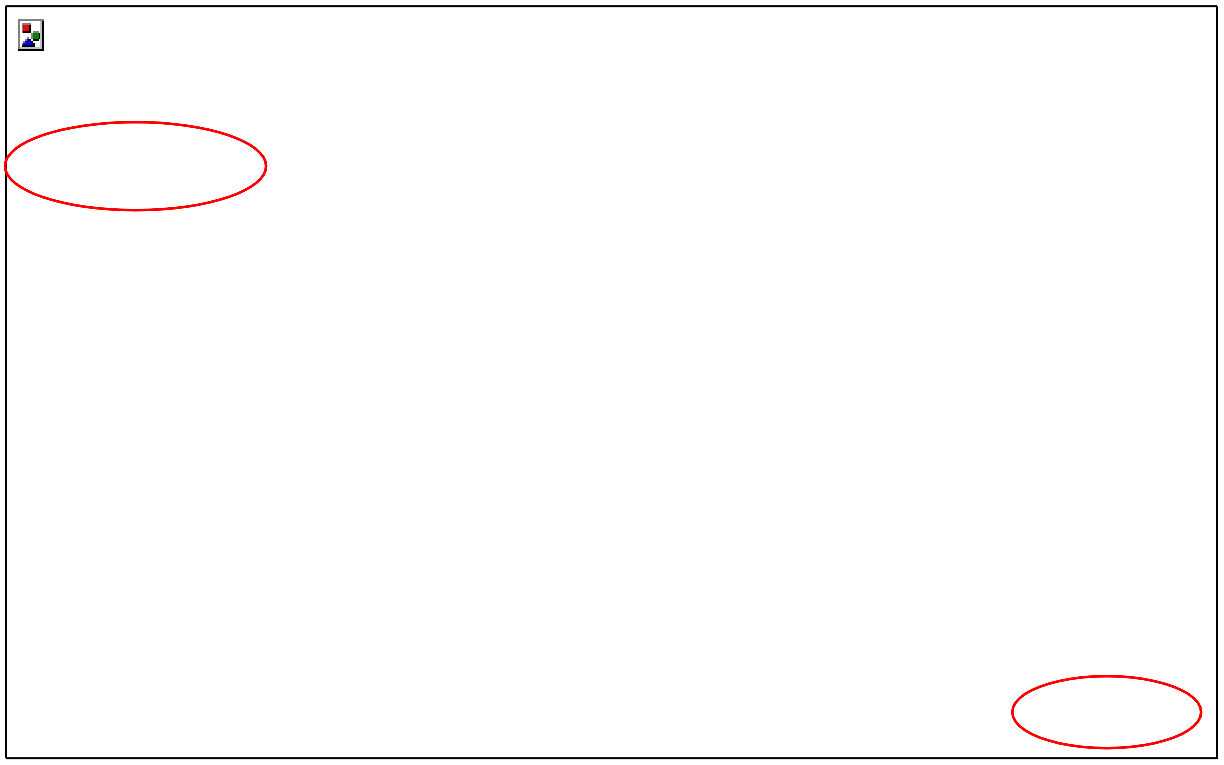


Step 8: give project name > remove tick in create main class > click FINISH



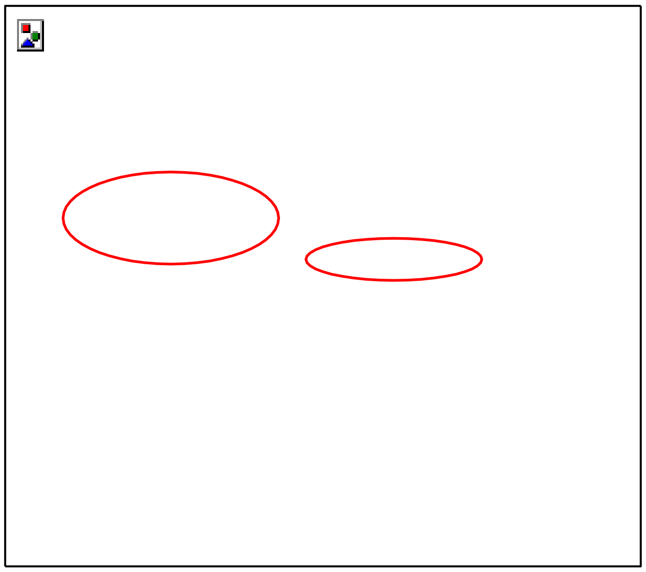
So, the project will be created under the project menu.

Step 9: click the project you created. Under that, click source packages > default package > new >java package> give package name > click finish.



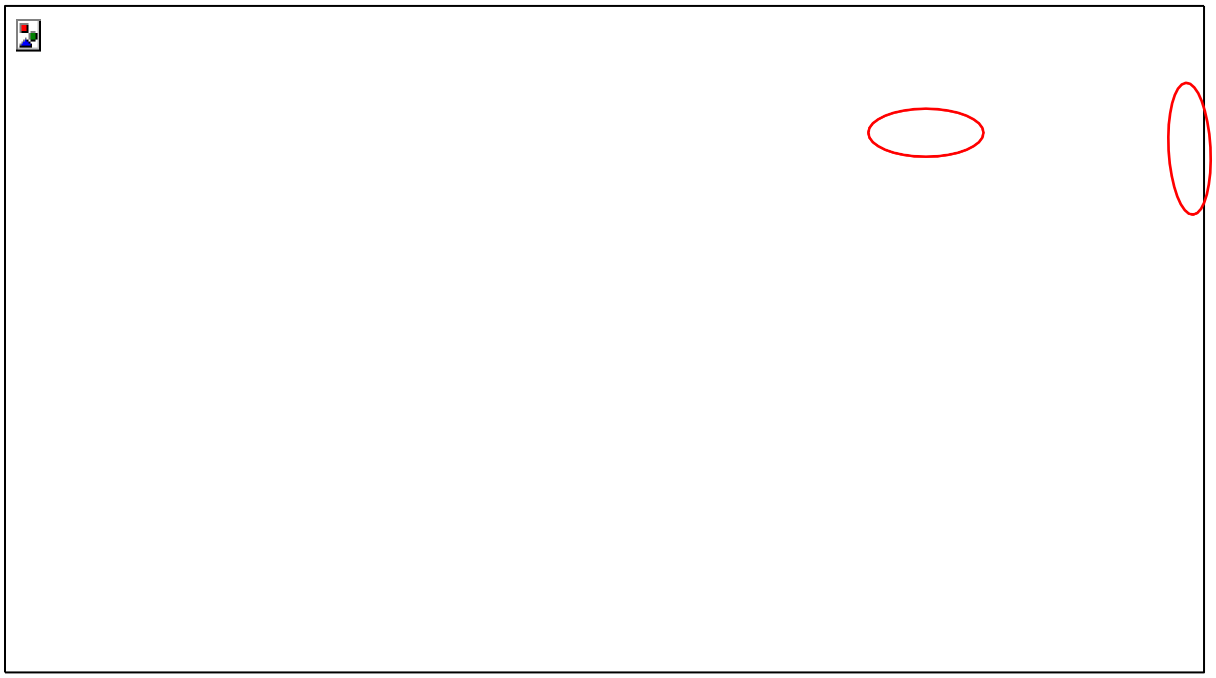
The package will be created in the source package under your project.

Step 10: click the package created under source package > new > JFrame Form >



A new JFrame form window will be popped. Give class name and click finish.

Step 11: This window will be shown. Click panel (if you want you can resize it). Then clickproperties and start doing designing your application.



Ex. No. :	DATABASE DESIGN USING EER-TO-ODB MAPPING / UML CLASS DIAGRAMS
Date :	

AIM: To create ER/ UML diagram of a database using MySQL workbench

PROCEDURE:

Step 1: First make sure you have a Database and Tables created on the MySQLserver.
For Example :-

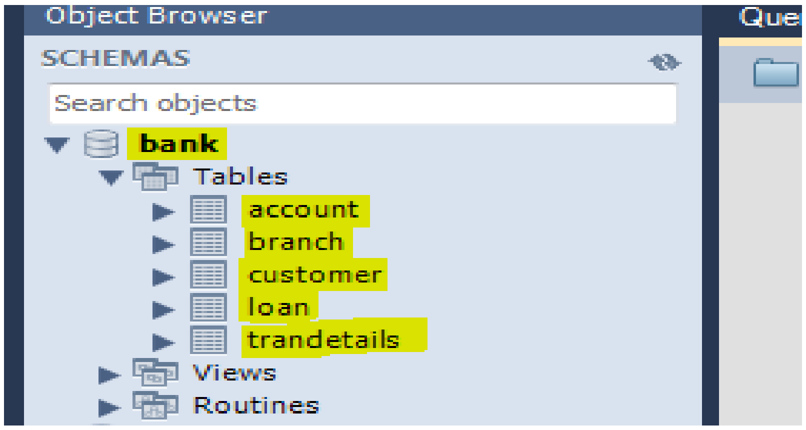
Database - *bank.*

Tables - *account, branch, customer, loan, trandetails.*

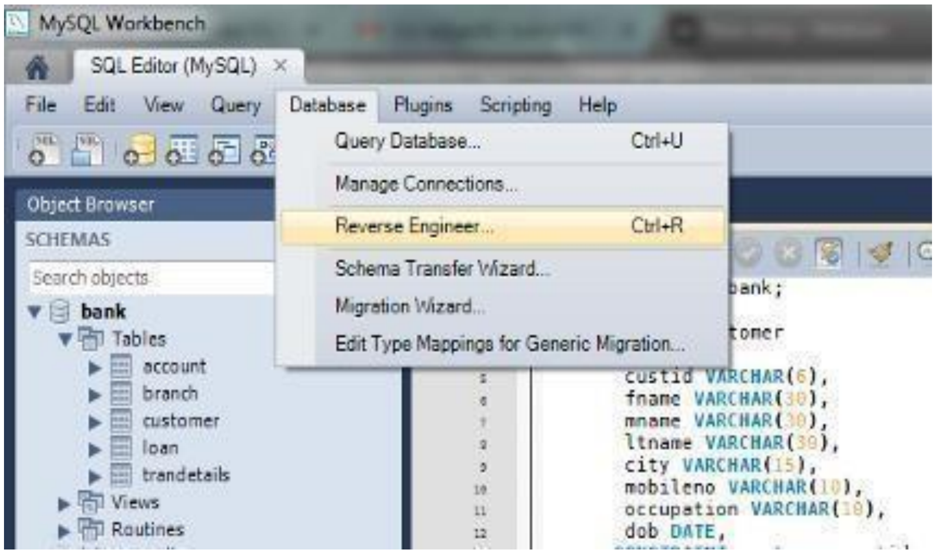
QUERYING/MANAGING THE DATABASE USING SQL PROGRAMMING

STORED PROCEDURES/FUNCTIONS

CONSTRAINTS AND SECURITY USING TRIGGERS

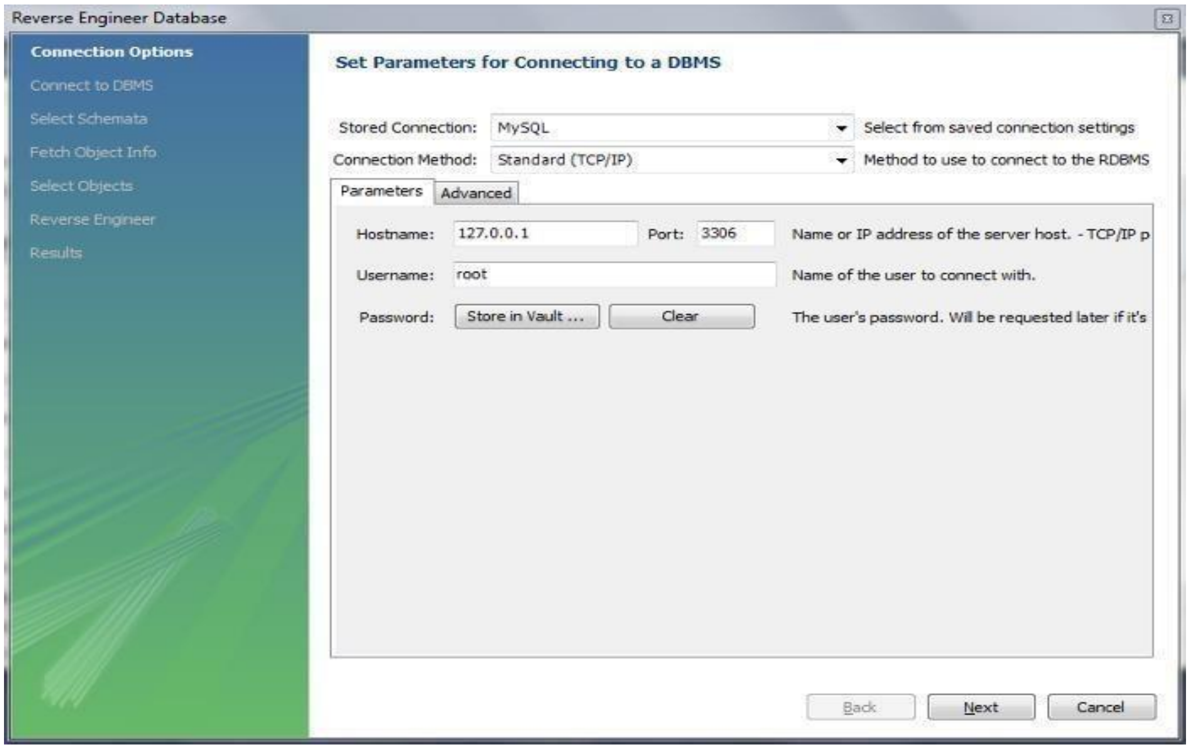


Step 2: Click on **Database -> Reverse Engineer**.



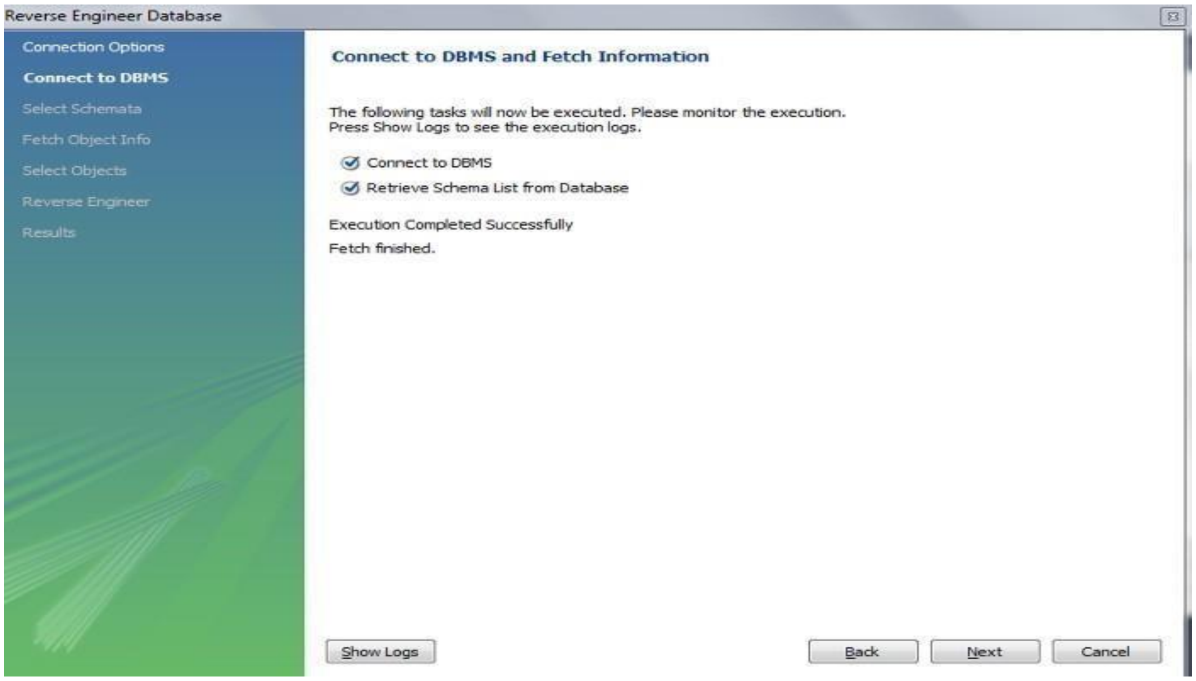
```
bank;
tomer
custid VARCHAR(6),
fname VARCHAR(30),
mname VARCHAR(30),
ltname VARCHAR(30),
city VARCHAR(15),
mobileno VARCHAR(10),
occupation VARCHAR(10),
dob DATE,
```

Step 3: Select your **stored connection** *(for connecting to your MySQL Server in whichdatabase is present)* from the dropdown. Then click **Next**.



Stored Connection: MySQL — Select from saved connection settings
Connection Method: Standard (TCP/IP) — Method to use to connect to the RDBMS
Hostname: 127.0.0.1 Port: 3306 — Name or IP address of the server host. - TCP/IP p
Username: root — Name of the user to connect with.
Password: — The user's password. Will be requested later if it's

Step 4: After the execution gets completed successfully *(connection to DBMS)*, click **Next.**



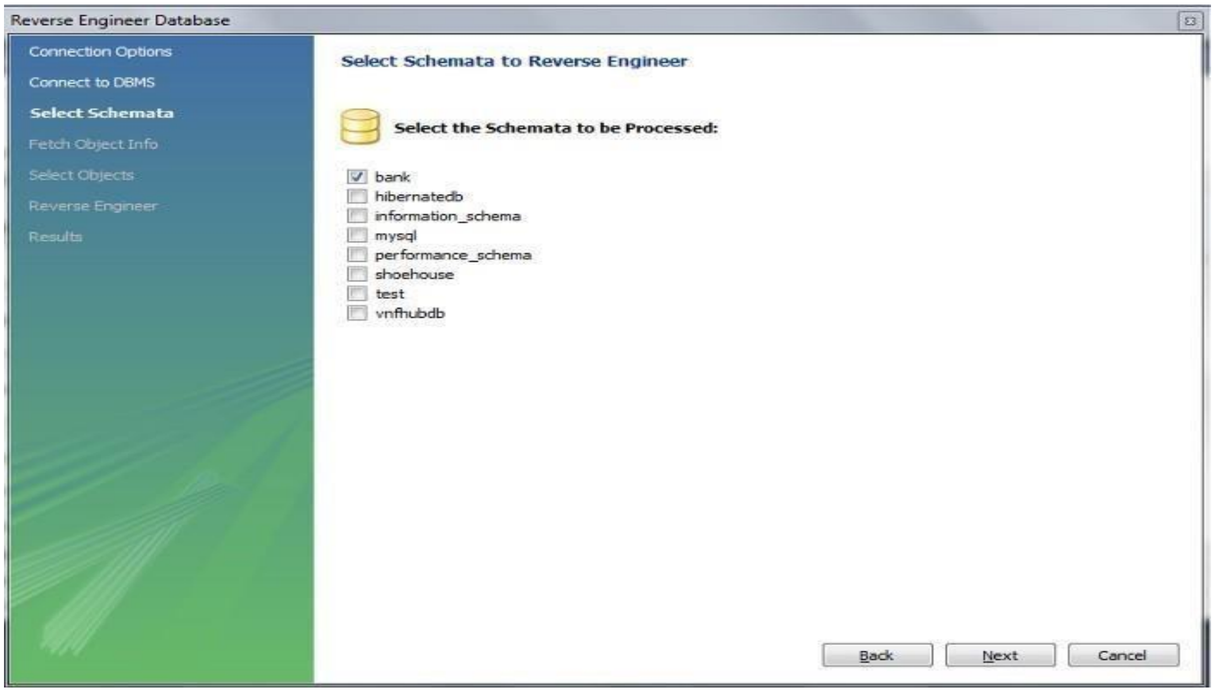
Connect to DBMS and Fetch Information

The following tasks will now be executed. Please monitor the execution.
Press Show Logs to see the execution logs.

✓ Connect to DBMS
✓ Retrieve Schema List from Database

Execution Completed Successfully
Fetch finished.

Step 5: Select your Database from the MySQL Server for which you want to create the ER Diagram(*in our case the database name is **"bank"**)*, then click **Next**

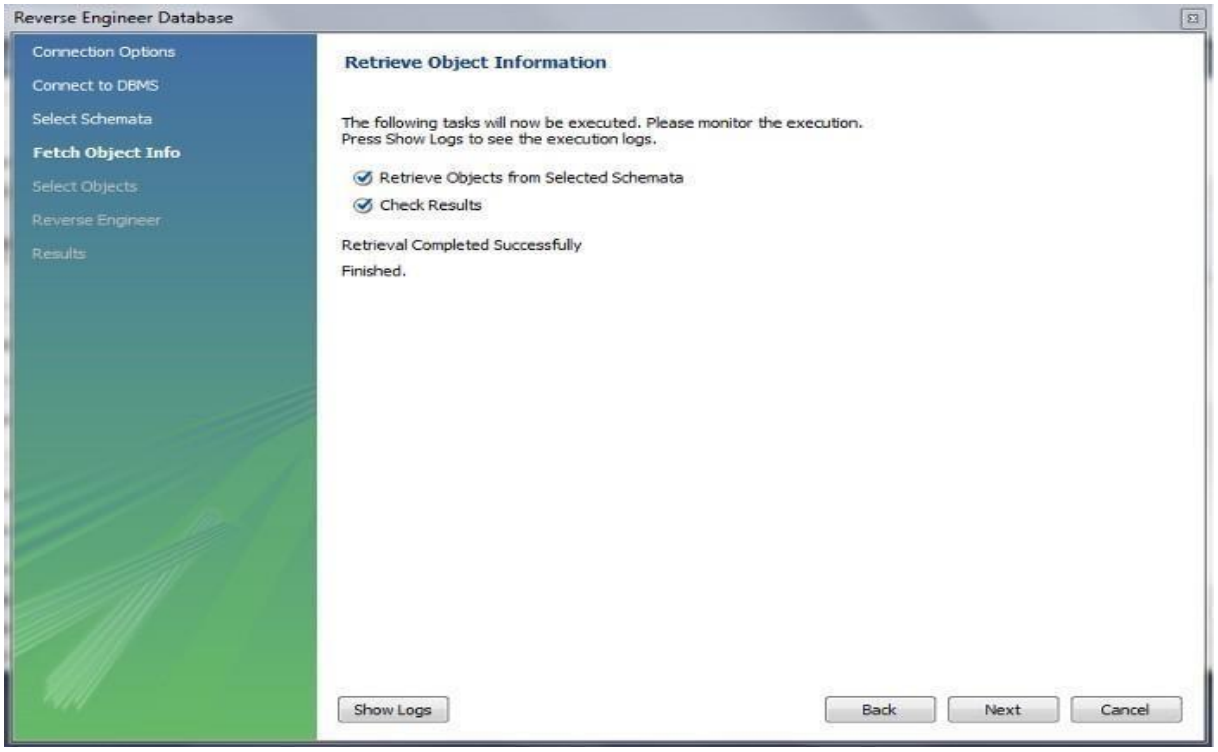


Select Schemata to Reverse Engineer

Select the Schemata to be Processed:

- [✓] bank
- [ ] hibernatedb
- [ ] information_schema
- [ ] mysql
- [ ] performance_schema
- [ ] shoehouse
- [ ] test
- [ ] vnfhubdb

Step 6: After the retrieval gets **completed** successfully for the selected Database, click **Next.**



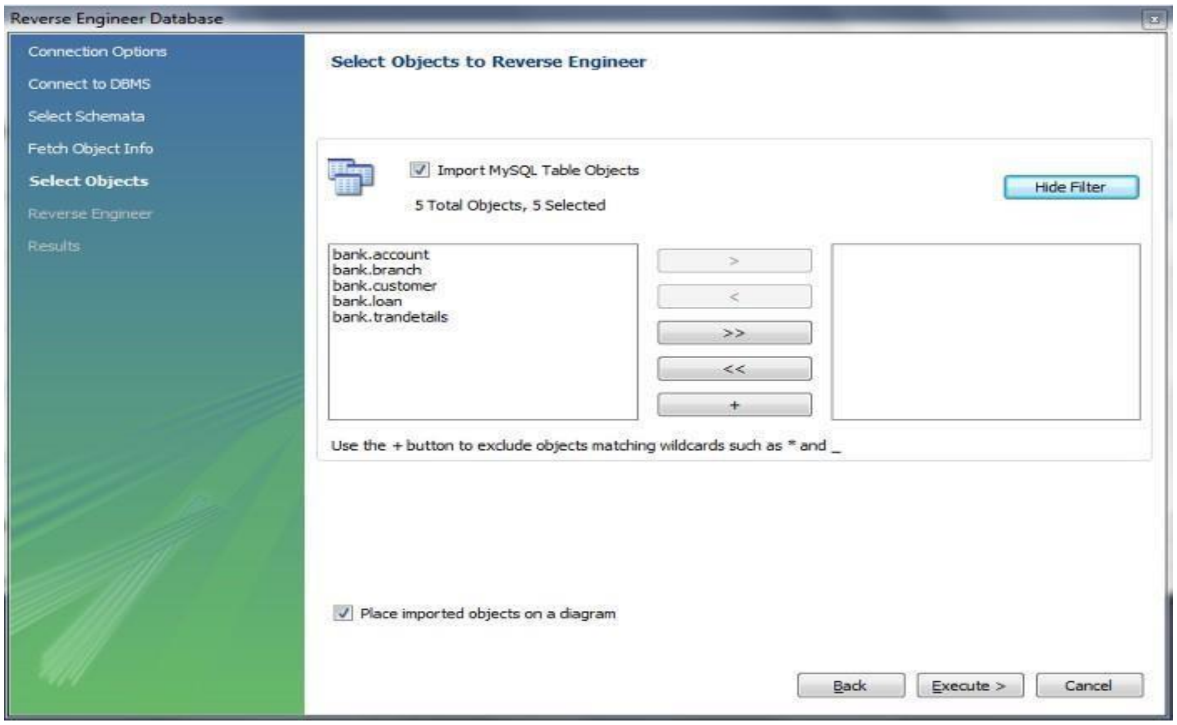
Retrieve Object Information

The following tasks will now be executed. Please monitor the execution.
Press Show Logs to see the execution logs.

✓ Retrieve Objects from Selected Schemata
✓ Check Results

Retrieval Completed Successfully
Finished.

Step 7: Select the Tables of the Database which you want to be visible on the ER Diagram *(In thiscase I am importing all the tables of the DB),* then click **Execute>.**



Select Objects to Reverse Engineer

☑ Import MySQL Table Objects

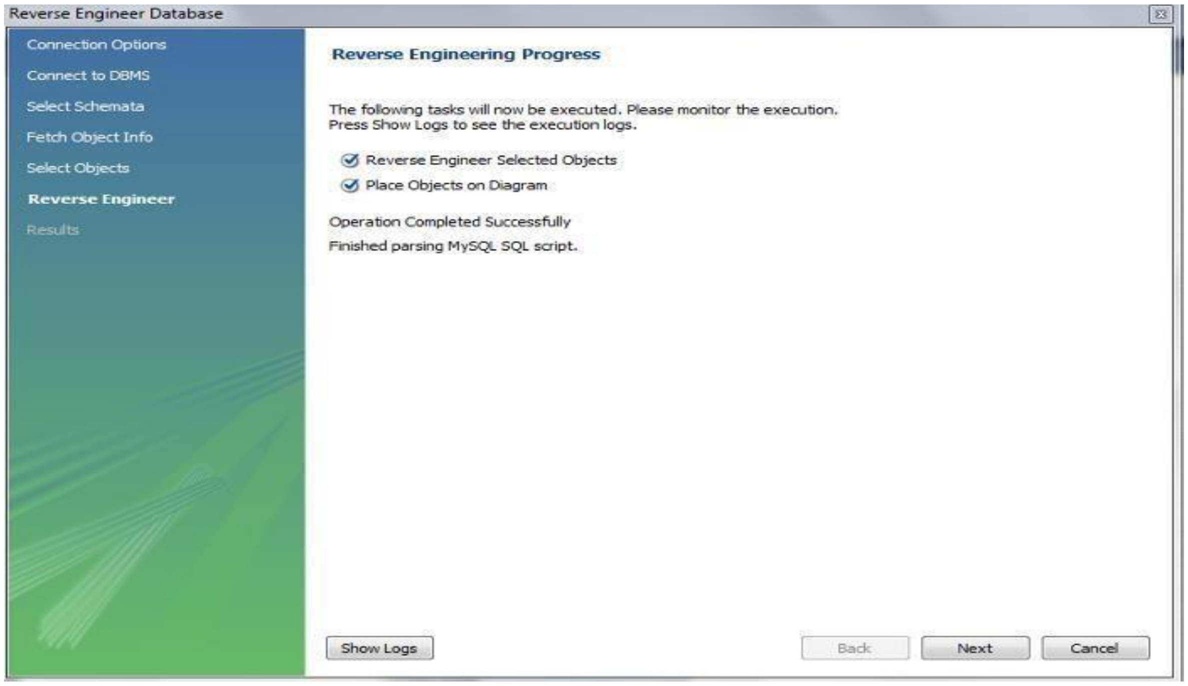
5 Total Objects, 5 Selected

bank.account
bank.branch
bank.customer
bank.loan
bank.trandetails

Use the + button to exclude objects matching wildcards such as * and _

☑ Place imported objects on a diagram

Step 8: After the Reverse Engineering Process gets completed successfully, click **Next.**



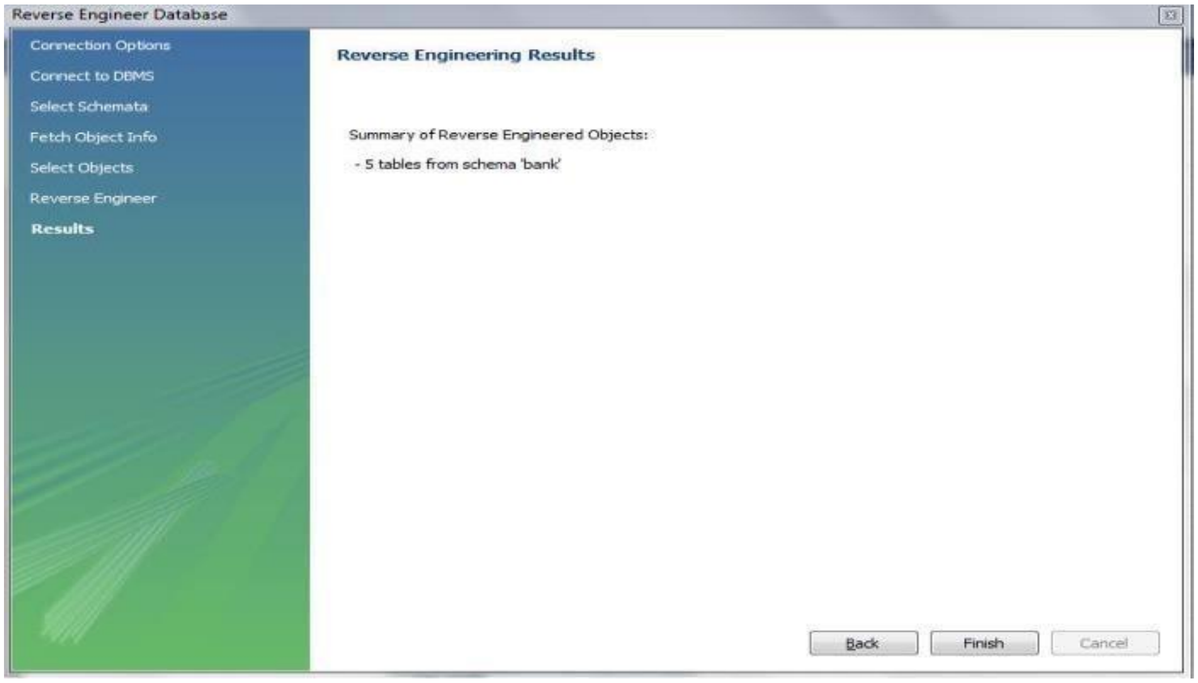
Reverse Engineering Progress

The following tasks will now be executed. Please monitor the execution.
Press Show Logs to see the execution logs.

✓ Reverse Engineer Selected Objects
✓ Place Objects on Diagram

Operation Completed Successfully
Finished parsing MySQL SQL script.

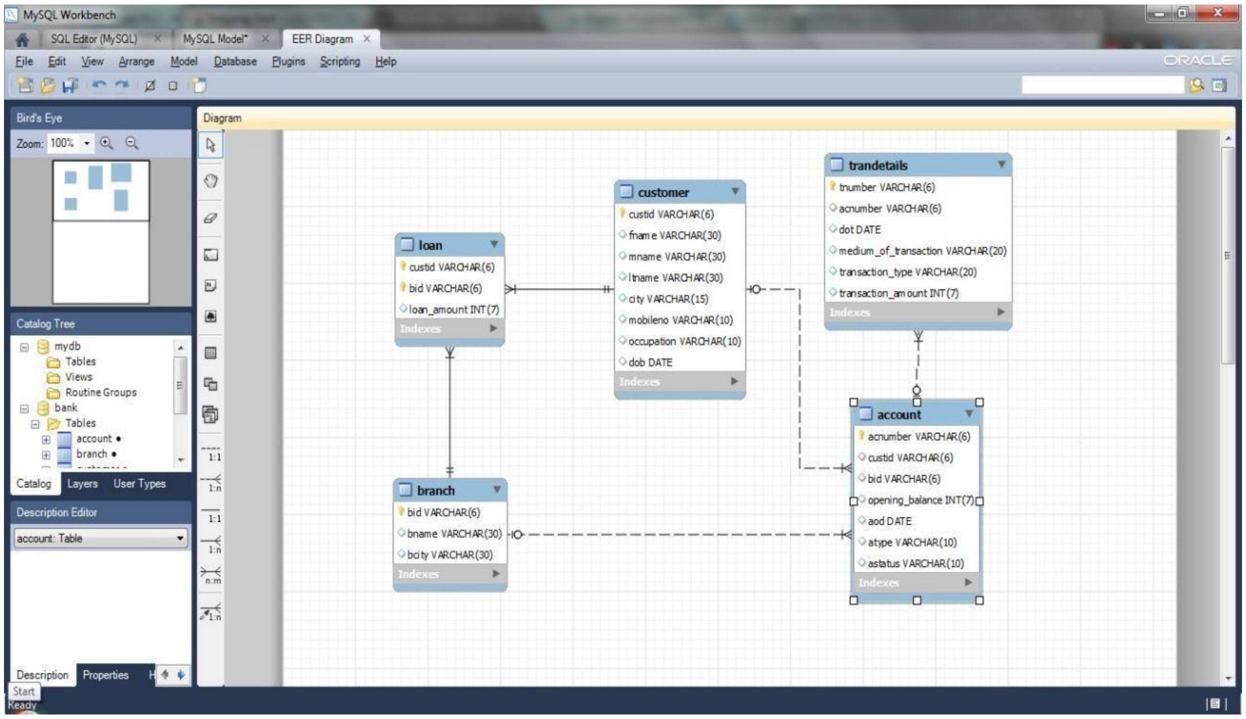
Step 9: Click **Finish.**



Reverse Engineering Results

Summary of Reverse Engineered Objects:
- 5 tables from schema 'bank'

✓ Now you can see the ER Diagram/UML diagram of the Database.



Ex. No. :	**USER-DEFINED TYPES IN SQL SERVER**
Date :	

AIM: Creating datas using User-Defined Types (UDTs) in SQL server.

PROCEDURE:

You can access user-defined type (UDT) functionality in Microsoft SQL Server from the Transact- SQL language by using regular query syntax. UDTs can be used in the definition of database objects, as variables in Transact-SQL batches, in functions and stored procedures, and as arguments in functions and stored procedures.

- **Defining UDT Tables and Columns**

Describes how to use Transact-SQL to create a UDT column in a table.

- **Manipulating UDT Data**

Describes how to use Transact-SQL to work with UDT data in SQL Server.

- **Defining UDT Tables and Columns**

- **CREATING TABLES WITH UDTS**

There is no special syntax for creating a UDT column in a table. You can use the name of the UDT in a column definition as though it were one of the intrinsic SQL Server data types. The following CREATE TABLE Transact-SQL statement creates a table named **Points**, with a column named **ID,** which is defined as an **int** identity column and the primary key for the table. The second column is named **PointValue**, with a data type of **Point**. The schema name used in this example is **dbo**. Note that you must have the necessary permissions to specify a schema name. If you omitthe schema name, the default schema for the database user is used.

CREATE TABLE dbo.Points
(ID int IDENTITY(1,1) PRIMARY KEY, PointValue Point)

- **Manipulating UDT Data**

- **Inserting Data in a UDT Column**

The following Transact-SQL statements insert three rows of sample data into the **Points** table. The **Point** data type consists of X and Y integer values that are exposed as properties of the UDT. You must use either the CAST or CONVERT function to cast the comma-delimited X and Y values to the **Point** type. The first two statements use the CONVERT function to convert a string value to the **Point** type, and the third statement uses the CAST function:

INSERT INTO dbo.Points (PointValue) VALUES (CONVERT(Point, '3,4'));INSERT INTO dbo.Points (PointValue) VALUES (CONVERT(Point, '1,5')); INSERT INTO dbo.Points (PointValue) VALUES (CAST ('1,99' AS Point));

- **Selecting Data**

The following SELECT statement selects the binary value of the UDT.

SELECT ID, PointValue FROM dbo.Points

To see the output displayed in a readable format, call the **ToString** method of the **Point** UDT, whichconverts the value to its string representation.

SELECT ID, PointValue.ToString() AS PointValueFROM dbo.Points;

This produces the following results.

```
ID PointValue
------------

1 3, 4
2 1, 5
3 1, 99
```

You can also use the Transact-SQL CAST and CONVERT functions to achieve the same results.

SELECT ID, CAST (PointValue AS varchar)FROM dbo.Points;

SELECT ID, CONVERT (varchar, PointValue) FROM dbo.Points;

The **Point** UDT exposes its X and Y coordinates as properties, which you can then select individually. The following Transact-SQL statement selects the X and Y coordinates separately:

SELECT ID, PointValue.X AS xVal, PointValue.Y AS yValFROM dbo.Points;

The X and Y properties return an integer value, which is displayed in the result set.

```
ID xVal yVal
         --   ---   ----
          1    3    4
          2    1    5
          3    1    99
```

➢ **WORKING WITH VARIABLES**

You can work with variables using the DECLARE statement to assign a variable to a UDT type. The following statements assign a value using the Transact-SQL SET statement and display the results by calling the UDT's **ToString** method on the variable:

DECLARE @PointValue Point;
SET @PointValue = (SELECT PointValue FROM dbo.PointsWHERE ID = 2);

SELECT @PointValue.ToString() AS PointValue;

The result set displays the variable value:

PointValue

-1,5

The following Transact-SQL statements achieve the same result using SELECT rather than SET forthe variable assignment:

DECLARE @PointValue Point;
SELECT @PointValue = PointValue FROM dbo.PointsWHERE ID = 2;

SELECT @PointValue.ToString() AS PointValue;

The difference between using SELECT and SET for variable assignment is that SELECT allows youto assign multiple variables in one SELECT statement, whereas the SET syntax requires each variable assignment to have its own SET statement.

- **COMPARING DATA**

You can use comparison operators to compare values in your UDT if you have set the **IsByteOrdered** property to **true** when defining the class.

SELECT ID, PointValue.ToString() AS PointsFROM dbo.Points
WHERE PointValue > CONVERT(Point, '2,2');

You can compare internal values of the UDT regardless of the **IsByteOrdered** setting if the values themselves are comparable. The following Transact-SQL statement selects rows where X is greater than Y:

SELECT ID, PointValue.ToString() AS PointValueFROM dbo.Points
WHERE PointValue.X < PointValue.Y;

You can also use comparison operators with variables, as shown in this query that searches for a matching PointValue.

DECLARE @ComparePoint Point;
SET @ComparePoint = CONVERT(Point, '3,4'); SELECT ID, PointValue.ToString() AS MatchingPoint

FROM dbo.Points
WHERE PointValue = @ComparePoint;

- **INVOKING UDT METHODS**

You can also invoke methods that are defined in your UDT in Transact-SQL. The **Point** class contains three methods, **Distance**, **DistanceFrom**, and **DistanceFromXY**. For the code listings defining these three methods, see Coding User-Defined Types.

The following Transact-SQL statement calls the **PointValue.Distance** method:

```
SELECT ID, PointValue.X AS [Point.X], PointValue.Y AS [Point.Y],
PointValue.Distance() AS DistanceFromZero
FROM dbo.Points;
```

The results are displayed in the **Distance** column:

```
ID  X   Y           Distance
--  --  --    ----------------
 1  3   4            5
 2  1   5     5.09901951359278
 3  1   99    99.0050503762308
```

The **DistanceFrom** method takes an argument of **Point** data type, and displays the distance from thespecified point to the PointValue:

```
SELECT ID, PointValue.ToString() AS Pnt,
PointValue.DistanceFrom(CONVERT(Point, '1,99')) AS DistanceFromPoint
FROM dbo.Points;
```

The results display the results of the **DistanceFrom** method for each row in the table:

```
ID Pnt  DistanceFromPoint
      --   -------------------

      1 3,4   95.0210502993942
      2 1,5             94
      3 1,9             90
```

The **DistanceFromXY** method takes the points individually as arguments:

```
SELECT ID, PointValue.X as X, PointValue.Y as Y, PointValue.DistanceFromXY(1,
99) AS DistanceFromXYFROM dbo.Points
```

The result set is the same as the **DistanceFrom** method.

➢ UPDATING DATA IN A UDT COLUMN

To update data in a UDT column, use the Transact-SQL UPDATE statement. You can also use a method of the UDT to update the state of the object. The following Transact-SQL statement updates a single row in the table:

```
UPDATE dbo.Points
SET PointValue = CAST('1,88' AS Point)WHERE ID = 3
```

You can also update UDT elements separately. The following Transact-SQL statement updates onlythe Y coordinate:

UPDATE dbo.Points SET PointValue.Y = 99WHERE ID = 3

If the UDT has been defined with byte ordering set to **true**, Transact-SQL can evaluate the UDT column in a WHERE clause.

UPDATE dbo.Points SET PointValue = '4,5'
WHERE PointValue = '3,4';

- **UPDATING LIMITATIONS**

You cannot update multiple properties at once using Transact-SQL. For example, the following UPDATE statement fails with an error because you cannot use the same column name twice in one UPDATE statement.

UPDATE dbo.Points
SET PointValue.X = 5, PointValue.Y = 99WHERE ID = 3

To update each point individually, you would need to create a mutator method in the Point UDT assembly. You can then invoke the mutator method to update the object in a Transact-SQL UPDATE statement, as in the following:

UPDATE dbo.Points
SET PointValue.SetXY(5, 99)WHERE ID = 3

- **DELETING DATA IN A UDT COLUMN**

To delete data in a UDT, use the Transact-SQL DELETE statement. The following statement deletesall rows in the table that match the criteria specified in the WHERE clause. If you omit the WHERE clause in a DELETE statement, all rows in the table will be deleted.

DELETE FROM dbo.Points
WHERE PointValue = CAST('1,99' AS Point)

Use the UPDATE statement if you want to remove the values in a UDT column while leaving other row values intact. This example sets the PointValue to null.

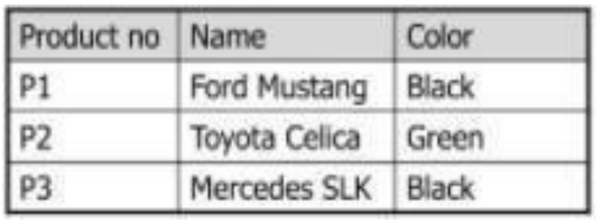
UPDATE dbo.Points SET PointValue = nullWHERE ID = 2

Ex. No. :	QUERYING THE OBJECT-RELATIONAL DATABASE USING OBJECT QUERY LANGUAGE
Date :	

AIM: Querying object-relational database using object query language.

PROCEDURE:
OQL is ap powerful and easy-to-use SQL-like query language with special features dealing with complex objects, values and methods.

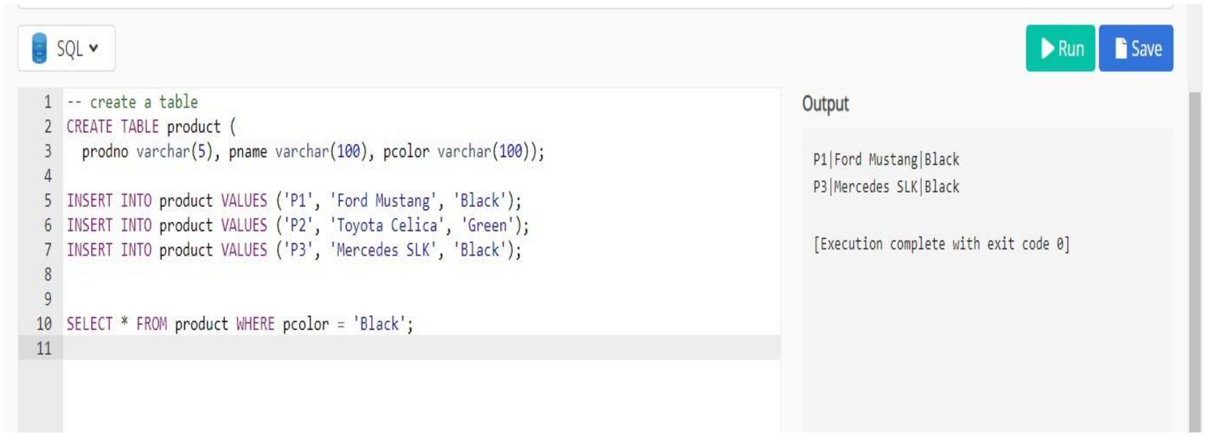
Example:



Product no	Name	Color
P1	Ford Mustang	Black
P2	Toyota Celica	Green
P3	Mercedes SLK	Black

The following is the sample query:

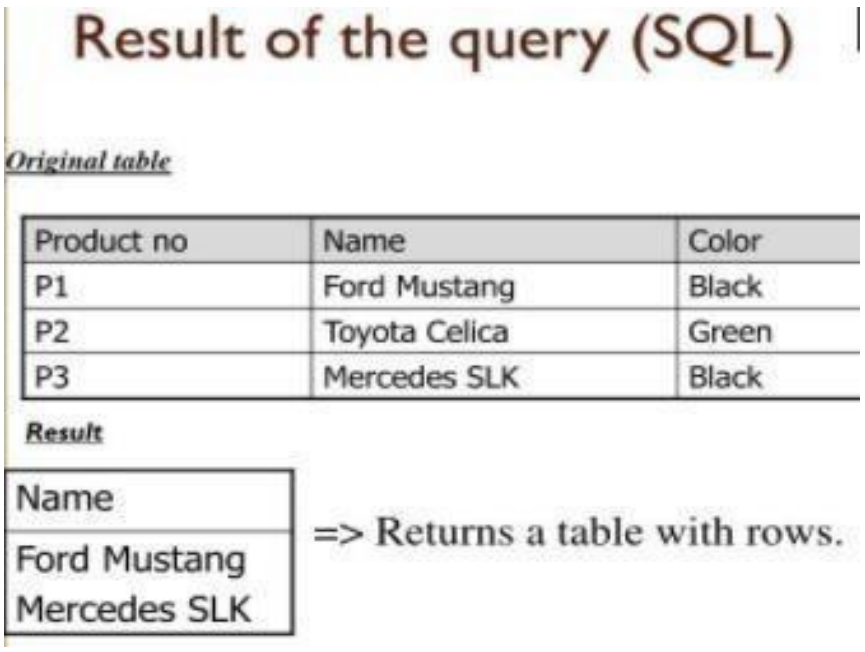
“what are the names of the black product?”Firstly, create table and insert columns.
Then, Select distinct p.name From products p Where p.color=”black”



```
-- create a table
CREATE TABLE product (
  prodno varchar(5), pname varchar(100), pcolor varchar(100));

INSERT INTO product VALUES ('P1', 'Ford Mustang', 'Black');
INSERT INTO product VALUES ('P2', 'Toyota Celica', 'Green');
INSERT INTO product VALUES ('P3', 'Mercedes SLK', 'Black');


SELECT * FROM product WHERE pcolor = 'Black';
```

```
P1|Ford Mustang|Black
P3|Mercedes SLK|Black

[Execution complete with exit code 0]
```

The above is valid for both SQL and OQL, but the results are different.



Result of the query (SQL)

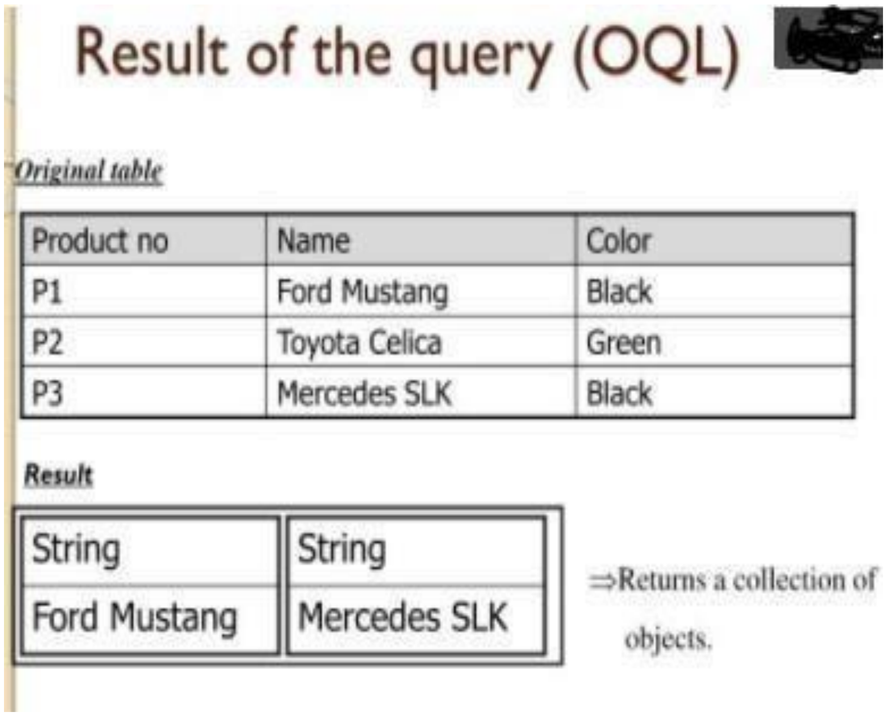
Original table

Product no	Name	Color
P1	Ford Mustang	Black
P2	Toyota Celica	Green
P3	Mercedes SLK	Black

Result

Name
Ford Mustang
Mercedes SLK

=> Returns a table with rows.



Result of the query (OQL)

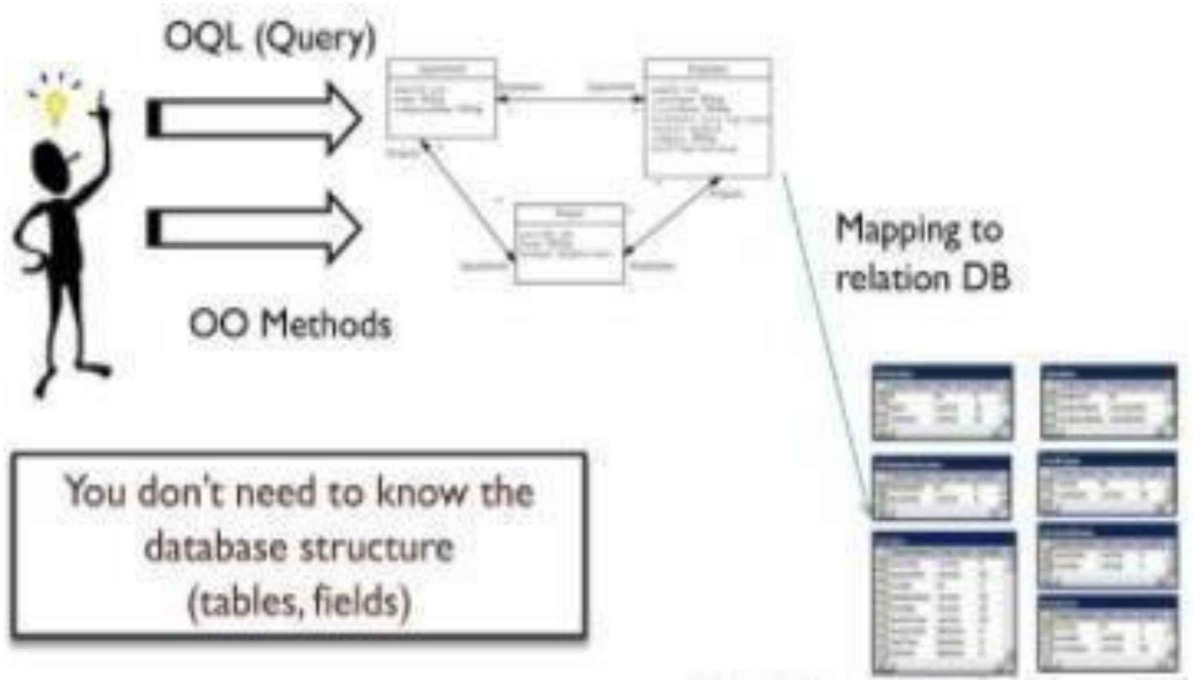
Original table

Product no	Name	Color
P1	Ford Mustang	Black
P2	Toyota Celica	Green
P3	Mercedes SLK	Black

Result

String	String
Ford Mustang	Mercedes SLK

⇒Returns a collection of objects.


OQL (Query)
OO Methods
Mapping to relation DB
You don't need to know the database structure (tables, fields)

We will discuss the syntax of simple OQL queries and the concept of using named objects or extentsas database entry points. Then, we will discuss the structure of query results and the use of path expressions to traverse relationships among objects.

1. Simple OQL Queries, Database Entry Points, and Iterator Variables

The basic OQL syntax is a select ... from ... where ... structure, as it is for SQL. For example, the query to retrieve the names of all departments in the college of ‗Engineering' can be written as follows:

Q0: select *D*.Dname from *D* in DEPARTMENTS where *D*.College = „Engineering";

In general, an **entry point** to the database is needed for each query, which can be any *named persistent object.* For many queries, the entry point is the name of the extent of a class. Recall that the extent name is considered to be the name of a persistent object whose type is a collection (in most cases, a set) of objects from the class. Looking at the extent names in Figure 11.10, the named object DEPARTMENTS is of type set <DEPARTMENT>; PERSONS is of type set<PERSON>; FACULTY is of type set<FACULTY>; and so on.

The use of an extent name—DEPARTMENTS in Q0—as an entry point refers to a persistent collection of objects. Whenever a collection is referenced in an OQL query, we should define an **iterator variable**—*D* in Q0—that ranges over each object in the collection. In many cases, as in Q0, the query will select certain objects from the collection, based on the conditions specified in the where clause. In Q0, only persistent objects *D* in the collection of DEPARTMENTS that satisfy the condition *D*.College = ‗Engineering' are selected for the query result. For each selected object *D*, the value of *D*.Dname is retrieved in the query result. Hence, the *type of the result* for Q0 is bag<string> because the type of each Dname value is string (even though the actual result is a set because Dname is a key attribute). In general, the result of a query would be of type bag for select ... from ... and of type set for select distinct ... from ... , as in SQL (adding the keyword distinct eliminates duplicates).

Using the example in Q0, there are three syntactic options for specifying iterator variables:

D* in DEPARTMENTSDEPARTMENTS *D* DEPARTMENTS AS *D
We will use the first construct in our examples.

The named objects used as database entry points for OQL queries are not limited to the names of extents. Any named persistent object, whether it refers to an atomic (single) object or to a collection object, can be used as a database entry point.

2. Query Results and Path Expressions
In general, the result of a query can be of any type that can be expressed in the ODMG object model. A query does not have to follow the select ... from ... where ...

structure; in the simplest case, any persistent name on its own is a query, whose result is a reference to that persistent object. For example, the query

Q1: DEPARTMENTS;
returns a reference to the collection of all persistent DEPARTMENT objects, whose type is set<DEPARTMENT>. Similarly, suppose we had given (via the database bind operation, see Figure 11.8) a persistent name CS_DEPARTMENT to a single DEPARTMENT object (the Computer Science department); then, the query

Q1A: CS_DEPARTMENT;
returns a reference to that individual object of type DEPARTMENT. Once an entry point is specified, the concept of a **path expression** can be used to specify a *path* to related attributes and objects. A path expression typically starts at a *persistent object name,* or at the iterator variable that ranges over individual objects in a collection. This name will be followed by zero or more relationship names or attribute names connected using the *dot notation.* For example, referring to the UNIVERSITY data-base in Figure 11.10, the following are examples of path expressions, which are also valid queries in OQL:

Q2: CS_DEPARTMENT.Chair; Q2A: CS_DEPARTMENT.Chair.Rank;
Q2B: CS_DEPARTMENT.Has_faculty;

The first expression Q2 returns an object of type FACULTY, because that is the type of the attribute Chair of the DEPARTMENT class. This will be a reference to the FACULTY object that is related to the DEPARTMENT object whose persistent name is CS_DEPARTMENT via the attribute Chair; that is, a reference to the FACULTY object who is chairperson of the Computer Science department. The second expression Q2A is similar, except that it returns the Rank of this FACULTY object (the Computer Science chair) rather than the object reference; hence, the type returned by Q2A is string, which is the data type for the Rank attribute of the FACULTY class.

Path expressions Q2 and Q2A return single values, because the ttributes Chair (of DEPARTMENT) and Rank (of FACULTY) are both single-valued and they are applied to a single object. The third expression, Q2B, is different; it returns an object of type set<FACULTY> even when applied to a single object, because that is the type of the relationship Has_faculty of the DEPARTMENT class. The collection returned will include references to all FACULTY objects that are related tothe DEPARTMENT object whose persistent name is CS_DEPARTMENT via the relationship Has_faculty; that is, references to all FACULTY objects who are working in the Computer Science department. Now, to return the ranks of Computer Science faculty, we *cannot* write

Q3 : CS_DEPARTMENT.Has_faculty.Rank;
because it is not clear whether the object returned would be of type set<string> or bag<string> (the latter being more likely, since multiple faculty may share the same rank). Because of this type of ambiguity problem, OQL does not allow expressions such as Q3. Rather, one must use an iterator variable over any collections, as in Q3A or Q3B below:

Q3A: select F.Rank
from F in CS_DEPARTMENT.Has_faculty;

Q3B: select distinct F.Rank
from F in CS_DEPARTMENT.Has_faculty;

Here, Q3A returns bag<string> (duplicate rank values appear in the result), whereas Q3B returns set<string> (duplicates are eliminated via the distinct keyword). Both Q3A and Q3B illustrate how an iterator variable can be defined in the from clause to range over a restricted collection specified in the query. The variable *F* in Q3A and Q3B ranges over the elements of the collection CS_DEPARTMENT.Has_faculty, which is of type set<FACULTY>, and includes only those faculty who are members of the Computer Science department.

In general, an OQL query can return a result with a complex structure specified in the query itself byutilizing the struct keyword. Consider the following examples:

Q4: CS_DEPARTMENT.Chair.Advises;
Q4A: select struct (name: struct (last_name: *S*.name.Lname, first_name: *S*.name.Fname),degrees:(select struct (deg: *D*.Degree, yr: *D*.Year,

college: *D*.College) from *D* in *S*.Degrees)) from *S* in CS_DEPARTMENT.Chair.Advises;

Here, Q4 is straightforward, returning an object of type set<GRAD_STUDENT> as its result; this is the collection of graduate students who are advised by the chair of the Computer Science department. Now, suppose that a query is needed to retrieve the last and first names of these graduate students, plus the list of previous degrees of each. This can be written as in Q4A, where the variable *S* ranges over the collec-tion of graduate students advised by the chairperson, and the variable *D* ranges over the degrees of each such student *S*. The type of the result of Q4A is a collection of (first-level) structs where each struct has two components: name and degrees.
The name component is a further struct made up of last_name and first_name, each being a single string. The degrees component is defined by an embedded query and is itself a collection of further (second level) structs, each with three string compo-nents: deg, yr, and college.

Note that OQL is *orthogonal* with respect to specifying path expressions. That is, attributes, relationships, and operation names (methods) can be used interchange-ably within the path expressions, as long as the type system of OQL is not compro-mised. For example, one can write thefollowing queries to retrieve the grade point average of all senior students majoring in Computer Science, with the result ordered by GPA, and within that by last and first name:

Q5A: select struct (last_name: S.name.Lname, first_name: S.name.Fname,gpa: S.gpa) from S in CS_DEPARTMENT.Has_majors where S.Class = „senior‟

order by gpa desc, last_name asc, first_name asc;

Q5B: select struct (last_name: S.name.Lname, first_name: S.name.Fname,gpa: S.gpa)

from S in STUDENTS where S.Majors_in.Dname = „Computer Science‟ and S.Class = „senior‟
order by gpa desc, last_name asc, first_name asc;
Q5A used the named entry point CS_DEPARTMENT to directly locate the reference to the Computer Science department and then locate the students via the relation-ship

Has_majors, whereas Q5B searches the STUDENTS extent to locate all students majoring in that department. Notice how attribute names, relationship names, and operation (method) names are all used interchangeably (in an orthogonal manner) in the path expressions: gpa is an operation; Majors_in and Has_majors are relation-ships; and Class, Name, Dname, Lname, and Fname are attributes. The implementa-tion of the gpa operation computes the grade point average and returns its value as a float type for each selected STUDENT.

The order by clause is similar to the corresponding SQL construct, and specifies in which order the query result is to be displayed. Hence, the collection returned by a query with an order by clause is of type *list.*

ADDITIONAL LAB EXPERIMENTS

Ex. No. :	AUTOMATIC BACK UP FILES AND RECOVERY FILES (GLA University)
Date :	

Aim:
To Implement a Query using automatic Back up files and Recovery Files.

Procedure:
Specifying a Backup File by Using Its Physical Name

The basic BACKUP syntax for specifying a backup file by using its physical device name is:

```
BACKUP DATABASE database_name
TO DISK = {‘physical_backup_device_name' | @physical_backup_device_name_var }
```

For example:

```
BACKUP DATABASE ACCTG_SRVR\SQLEXPRESS
TO DISK = 'Z:\SQLServerBackups\ ACCTG_SRVR\SQLEXPRESS.bak';
GO
```

To specify a physical disk device in a Restore statement, the basic syntax is:

```
RESTORE {DATABASE | LOG} database name
FROM DISK = {‘physical_backup_device_name' | @physical_backup_device_name_var }
```

For example,

```
RESTORE DATABASE ACCTG_SRVR\SQLEXPRESS
FROM DISK = 'Z:\SQLServerBackups\ ACCTG_SRVR\SQLEXPRESS.bak';
```

Example:

```
Use <database>
SELECT execquery.last_execution_time AS [Date Time], execsql.text AS [Script] FROM
sys.dm_exec_query_stats AS execquery
CROSS APPLY sys.dm_exec_sql_text(execquery.sql_handle) AS execsql
ORDER BY execquery.last_execution_time DESC
```

Result:
Thus the implementation of automatic Back up files and recovery Files program is developed and the results are verified

Ex. No. :	**EMBEDDED AND DYNAMIC SQL (NIT-Trichy)**
Date :	

Aim:
Write PL/SQL program for Embedded SQL.

Procedure:
To create the table of an example of most common elements of Embedded SQL applications.
Concepts

1. The programming language in which the SQL statements are embedded is called the host language. The SQL statements and host language statements make the source program which is fed to a SQL precompiler for processing the SQL statements.
2. The host programming languages variables can be referenced in the embedded SQL statements, which allows values calculated by the programs to be used by SQL statements.
3. There are some special program variables which are used to assign null values to database columns. These program variables support the retrieval of null values from the database.

Format

```
/* Begin program */
EXEC SQL INCLUDE SQLCA;
EXEC SQL BEGIN DECLARE SECTION
host_customername character_string(20)
host_cust_number integer
EXEC SQL END DECLARE SECTION
EXEC SQL WHENEVER SQLERROR STOP
EXEC SQL CONNECT frans
/* Formulate query, something like: */
EXEC SQL SELECT customername cust_number
INTO host_customername host_cust_number
FROM customer
WHERE cust_number = 10001
/* Print host_customername and host_cust_number */
EXEC SQL DISCONNECT
/* End program */
```

Result:
Thus the implementation of Embedded SQL program is developed and the results are verified.

Ex. No. :	**REPORTS USING FORM DESIGN (NIT-Trichy)**
Date :	

Aim:
To generate report for customer details using VB.

Procedure:
Step1: select data project from new project dialog box.

Step2: right click connection one of the data environment and set the property as Microsoft jet 4.00 OLEDB provider and database name as empdb

Step3: check weather the connection is established using test connection button.

Step4: add command for the connection1 and set database object as table and object name as emptable.

Step5: drag command,objects and drop it in the data reports detail section. Two items will appear for each object.the first label for heading information. Move this label into page header section. The label is used to store the actual value of the object.

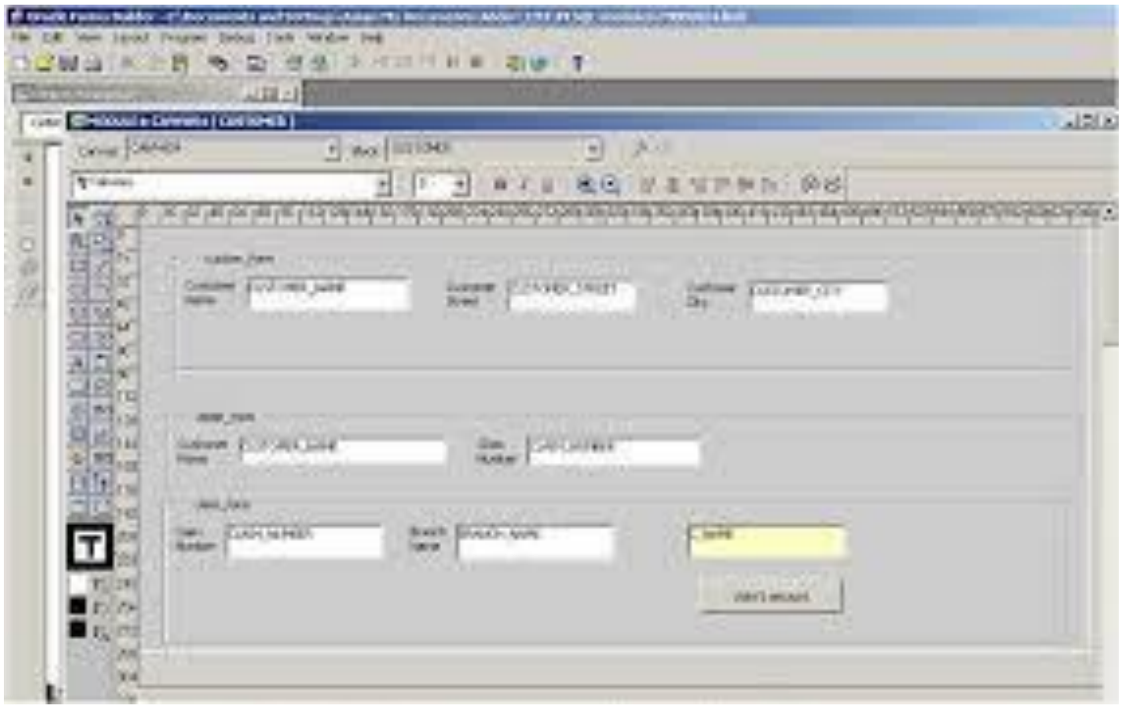
Step6: set data source and data report1 as data environment1 and the data member as command1

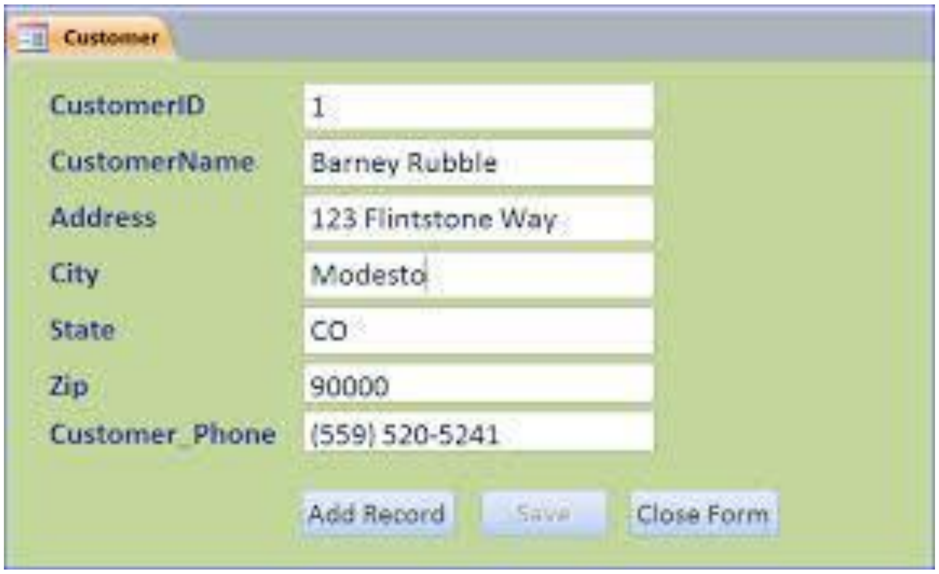
Step7: format the report as required.

Step8: place a button in the form and connect it to the data report to view the report.

PROGRAM:
```
Private Sub Command1_Click()
Dbase.recordset.addnew
End Sub
Private Sub Command1_Click()
Dbase.recordset.delete
Dbase.recordset.movefirst
End Sub
Private Sub Command1_Click()
Datareport1.show
End Sub
Private Sub Command1_Click()
Unload me
End Sub
```

OUTPUT:





Customer

CustomerID	1
CustomerName	Barney Rubble
Address	123 Flintstone Way
City	Modesto
State	CO
Zip	90000
Customer_Phone	(559) 520-5241

Result:
Thus the implementation Customer details using VB successfully completed

Ex. No. :	**REPORTS USING SQL (NIT-Roorkee)**
Date :	

Aim:
Prepare a Reports Using Table in SQL.

EXAMPLE:

```
SQL> ttitle 'Customer details'
SQL> btitle "End”
SQL> column mark heading 'total' \format$999
SQL> clear computes
computes cleared
SQL> break on title skip 2
SQL> compute sum of total on title
SQL> select * from customer order by custno,custname,purchaseitem,total;
                         Customer Details
Custno    Total
---------- ---------
223       250
554       100
154       200
134       450
-----------
Sum     $1000
```

Result:

Thus the Reports in SQL successfully Completed.