
	GRT INSTITUTE OF ENGINEERING AND TECHNOLOGY, TIRUTTANI - 631209 <small>Approved by AICTE, New Delhi Affiliated to Anna University, Chennai</small>	
DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND DATA SCIENCE		

PRACTICAL RECORD

Year/Semester: II / 3

Subject Code/Name: AD3381 DATABASE DESIGN AND MANAGEMENT LABORATORY



Name: _____

Register. No: _____

GRT INSTITUTE OF ENGINEERING AND TECHNOLOGY

GRT Mahalakshmi Nagar, Chennai – Thirupathi Highway,
Tiruttani – 631209.

2023 -2024

	GRT INSTITUTE OF ENGINEERING AND TECHNOLOGY, TIRUTTANI - 631209 <small>Approved by AICTE, New Delhi Affiliated to Anna University, Chennai</small>	
DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND DATA SCIENCE		

Bonafide Certificate

NAME: **COURSE:**

REGN.NO: **YEAR/SEM:**

This is to certified to be the bonafide record of work done by the student for **AD3381 DATABASE DESIGN AND MANAGEMENT LABORATORY** in the Department of **ARTIFICIAL INTELLIGENCE AND DATA SCIENCE** in **GRT Institute of Engineering and Technology** during the year **September 2023 to January 2024**.

Submitted for the practical examination held on.....

STAFF IN CHARGE

HEAD OF THE DEPARTMENT

INTERNAL EXAMINER

EXTERNAL EXAMINER

INDEX

Subject Code/Name: AD3381 DATABASE DESIGN AND MANAGEMENT LABORATORY

EX. NO	DATE	NAME OF THE EXPERIMENTS	PAGE NO	MARKS	FACULTY SIGNATURE
1		Database Development Life Cycle: Problem definition and requirement analysis scope and constraints			
2		Database design using conceptual modeling (ER-EER)-Top-down approach Mapping conceptual to relational database validate using normalization.			
3		Implement the database using SQL Data definition with constraints, Views			
4		Query the database using SQL Manipulation			
5)a		Querying/Managing the database using SQL Programming Stored Procedures/Functions			
5)b		Querying/Managing the database using SQL Programming Constraints and security using Triggers			
6		Database design using Normalization - bottom-up approach			
7		Develop database applications using IDE/RAD tools (Eg, Net Beans, Visual Studio)			
8		Database design using EER-to-ODB mapping / UML Class Diagrams			
9		Object features of SQL-UDTs and sub-types Tables using UDTs, Inheritance , Method definition			
10		Querying the Object-relational database using Object Query language			
CONTENT BEYOND SYLLABUS					
11		Mongo DB			
12		SQL Queries on Clustered and Non-Clustered Indexes			
13		Auditing with Trigger			
14		Concepts of Cursor			

EX.NO:1a	Database Development Life Cycle for Library Management System. Problem definition and Requirement analysis Scope and Constraints
DATE:	

AIM:

Procedure:

The Database Development Life Cycle contains six phases: database initial study, database design, implementation and loading, testing and evaluation, operation, and maintenance and evolution.

1.

The Database Initial Study:

In the Database initial study, the designer must examine the current systems operation within the company and determine how and why the current system fails. The overall purpose of the database initial study is to:

- Analyze the company situation.
- Define problems and constraints.
- Define objectives.
- Define scope and boundaries.

2. Database Design:

The second phase focuses on the design of the database model that will support company operations and objectives. This is arguably the most critical Database Development Life cycle phase : making sure that the final product meets user and system requirements.

3. Implementation and Loading:

The output of the database design phase is a series of instructions detailing the creation of tables, attributes, domains, views, indexes, security constraints, and storage and performance guidelines. In this phase, you actually implement all these design specifications.

4. Testing and Evaluation:

In the design phase, decisions were made to ensure integrity, security, performance, and recoverability of the database. During implementation and loading, these plans were put into place. In testing and evaluation, the DBA tests and fine-tunes the database to ensure that it performs as expected. This phase occurs in conjunction with applications programming.

5. Operation

Once the database has passed the evaluation stage, it is considered to be operational. At that point, the database, its management, its users, and its application programs constitute a complete information system. The beginning of the operational phase invariably starts the process of system evolution.

6. Maintenance and Evolution

The database administrator must be prepared to perform routine maintenance activities within the database. Some of the required periodic maintenance activities include:

- Preventive maintenance (backup).
- Corrective maintenance (recovery).
- Adaptive maintenance (enhancing performance, adding entities and attributes, and soon).
- Assignment of access permissions and their maintenance for new and old users.

Database Development Life Cycle for Library Management System. Procedure:

a) Problem definition:

- The library system is a web-based application which is employed for automating a library.
- It allows the librarian to maintain the information about books, magazines, journals, CDs and its users.
- Furthermore, it provides the following facilities to its users
 - Search for items
 - Browse
 - Checkout items
 - Return items
 - Make reservation
 - Remove reservation, etc.
- For borrowing the item from the library, any user must get registered in the system initially.
- The users can search for any item in the library by using the 'search option'.
- If the user finds the item he/she is searching for in the library, he/she can check out the item from the library.
- If the study material is not available in the library at the moment, the user can make reservation for that item.
- The moment the item is available, the user who first reserved for that item is notified first.
- If the user checks out the item from the library, the reservation gets cancelled automatically. The reservation can also be cancelled through an explicit cancellation procedure.
- The librarian is an employee of the library who interacts with the borrowers whose work is supported by the system.

- The system allows the librarian to perform the following functions with a lot of ease
 - Create
 - Update
 - Delete information about titles
 - Borrowers
 - Items and reservations in the system
- The library system can run on popular web-browser platforms like Windows Explorer, Netscape Navigator, etc. It can be easily extended with new functionality.

b) Requirement Analysis:

1. Any library member should be able to search books by their title, author, and subject category as well by the publication date.
2. Each book will have a unique identification number and other details including a rack number which will help to physically locate the book.
3. There could be more than one copy of a book, and library members should be able to check-out and reserve any copy. We will call each copy of a book, a book item.
4. The system should be able to retrieve information like who took a particular book or what are the books checked-out by a specific library member.
5. There should be a maximum limit on how many books a member can check-out.
6. There should be a maximum limit on how many days a member can keep a book.
7. The system should be able to collect fines for books returned after the due date.
8. Members should be able to reserve books that are not currently available.
9. The system should be able to send notifications whenever the reserved books become available, as well as when the book is not returned within the due date.
10. Each book and member card will have a unique barcode. The system will be able to read barcodes from books and members library cards.

c) Scope:

1. To assist the staff in capturing the effort spent on their respective work areas.
2. To utilize the resources in an efficient manner by increasing their productivity through automation.
3. The system generates types of information that can be used for various purposes. Thus, there are information scopes behind developing the “Library Management System” and it reduces a lot of burden of the entry.

d) Constraints:

1. There should be a maximum limit (5) on how many books a member can check-out.
2. There should be a maximum limit (10) on how many days a member can keep a book.
3. The system should be able to collect fines for books returned after the due date.

RESULT:

EX.NO:1b	Database Development Life Cycle for Hotel Management System. Problem definition and Requirement analysis Scope and Constraints
DATE:	

AIM:

Procedure:

a) Problem Definition:

A hotel system manages information about rooms, reservations, customers, and customer billing. A customer can make reservations, change, or cancel reservations through the hotel website. When a customer makes reservations, he/she needs to check if a room the customer wants to reserve is available.

If a room is available, the customer enters his/her information to the system and receives a confirmation number from the web site. A desk clerk checks in a customer with only a prior reservation, change the check out date, and check out the customer. A room is assigned to the customer at check-in time and a customer billing record is created at that time. The customer billing record is updated every night at 12. When a customer checks out, the desk clerk prints the bill. A customer can pay by cash, check, or credit card when he/she checks out.

b) Requirement Analysis:

Requirements analysis is the analysis definition process of requirements, is the start of the planning and development period of the project. Needs analysis task is to thoroughly describe the function and performance of the software, identify limits of software design and software interface details with other elements of the system, defining the effectiveness of software requirements. Requirements analysis includes business requirements, functional requirements and development requirements

Business requirements analysis

(1) Room booking capability can handle customer bookings by various means, such as phone book, online bookings, and reservations at the front desk.

(2) Rooms of the hotel have different grades, and require the system to be able to categorize the room management, and according to book different types of rooms to offer available prices, booking discount timing function settings such as lowest price, easy to fit individual traveler and group reservations.

(3) Check function

(4) Keep abreast of the current state of all whether it is available, such as clean the room, which was checked out, to track whether the guest stay. Based on client occupancy information, you need to provide a guest information management function, in order to achieve unified management to the guests of the hotel information, such as ID number for customer in queries, queries according to the checking dates, change the guest contact information and soon.

(5) A hotel management system should have recognition module in order to distinguish each business (accommodation, registration, cashier, hospitality, and other services) to which staff action is, that means the system should set log in module.

Functional requirements analysis

Hotel management system consists of both background and foreground parts, front office is responsible for booking, reception and cashier services, the background used for administrators to manage systems, such as setting room type, room settings, operator settings, financial management and warehouse management. Cash register function requirements.

Financial management requirements: Finance is the core of hotel Admin, it has the following functional requirements: set up account, initial processing, bookkeeping, certificate, voucher budget, other management review post, enter detailed ledger, inquire function, processing capabilities, a variety of reports and books of output.

Development requirements analysis:

Regardless of hotel scale, its future direction will be to network and information development, its main business needs mainly show in the network business process requirements and hotel information process requirements

. It mainly reflects in: remote query processing capabilities with the business requirements, cooperation with other relevant units of demand, as with other businesses such as hotels, travel agencies, the Ministry of public security, hotel information needs, such as electronic door locks, program-controlled exchange equipment, magnetic card consumer and other business process requirements.

c) Scope:

The software product to be produced is a Hotel Management System which will automate the major hotel operations. The first subsystem is a Reservation and Booking System to keep track of reservations and room availability. The second subsystem is the Tracking and Selling Food System that charges the current room.

The third subsystem is a General Management Services and Automated Tasks System which generates reports to audit all hotel operations and allows modification of subsystem information. These three subsystems' functionality will be described in detail in section 2-Overall Description. There are two end users for the HMS. The end users are the hotel staff (customer service representative) and hotel managers..

The Hotel Management Systems objective is to provide a system to manage a hotel that has increased in size to a total of 100 rooms. Without automation the management of the hotel has become an unwieldy task. The end users' day-to-day jobs of managing a hotel will be simplified by a considerable amount through the automated system.

d) Constraints:

The devised constraints of our system can be to implement the system in such a way that it causes least crashes and the system exceptions are handled properly. The Data security issue must be handled carefully so that privacy is maintained and no one can access the hotel data from outside.

RESULT:

EX.NO:2	Database Design Using Conceptual Modeling (ER-EER) –Top-Down Approach Mapping Conceptual to Relational Database and Validate Using Normalization
DATE:	

AIM:

Procedure:

The mapping of conceptual-model instances to a database schema is fundamentally the same for all conceptual models. A conceptual-model instance describes the relationships and constraints among the various data items.

Many conceptual model instances, however, satisfy stronger than necessary conditions, and it is easy to see that they are canonical. We can see, for example, that an ER model instance is canonical by checking the following criteria.

1. Each attribute is atomic (i.e., not decomposable into component attributes we wish to access in the database).
2. Each entity set has one or more keys (possibly inherited if the entity set is weak or in an ISA hierarchy), but has no other FDs among attributes with left-hand sides that are not keys.
3. Each many-many relationship set has one or more keys, but no other FDs among attributes with left-hand sides that are not keys.
4. Every n-ary relationship set is fully
5. There are no relationship-set cycles, or if there are cycles, then every path from one entity set to another is non-redundant in the sense that we cannot compute any relationship set as combinations of joins and projections of other relationship sets.

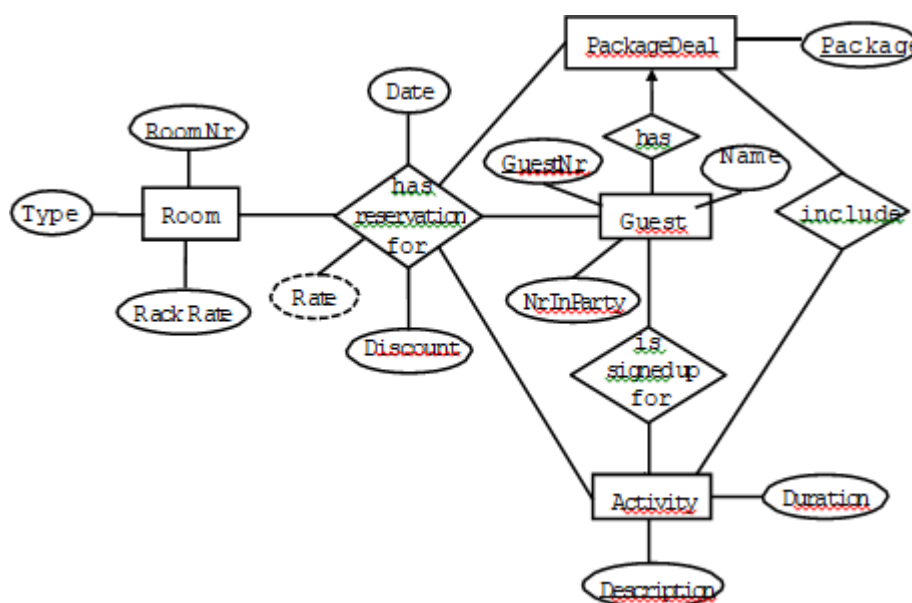
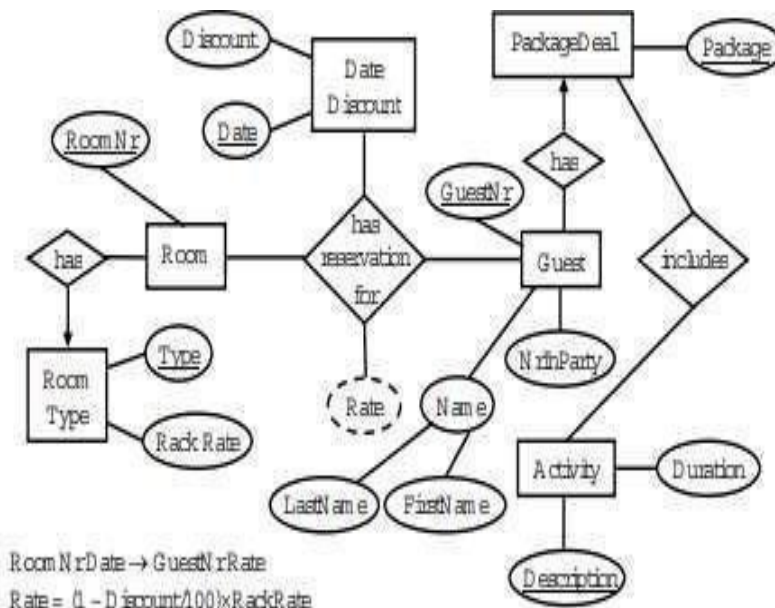


Fig 2.1 ER Diagram with Constraints whose Standard Mapping will Yield Normal Form Violations.



Room(RoomNr, Type, RackRate)
 Guest(GuestNr, Name, NrInParty, Package)
 Activity(Description, Duration)
 HasReservationFor(RoomNr, Date, GuestNr, Rate, Discount, Package, Description)
 IsSignedUpFor(GuestNr, Description)
 Includes(Package, Description)

Fig2.2GeneratedRelationalSchemas–NotNormalized

- Guest is not in 1NF because Name is not atomic. We replace Name by FirstName and LastName in Guest, which yields Guest (GuestNr, FirstName, LastName, NrInParty, Package).
- Has Reservation For is not in 2NF because of DateDiscount. We decompose has Reservation For, which yields Has ReservationFor(RoomNr, Date, GuestNr, Rate, Package, Description) and a new relational schema DateDiscount(Date, Discount).
- Room is not in 3NF because of TypeRackRate. We decompose Room, which yields Room(RoomNr, Type) and a new relational schema RoomType(Type, RackRate).

AfterNormalization:

Fig2.3NormalizedRelationSchemas

Room(RoomNr, Type)
 RoomType(Type, RackRate)
 Guest(GuestNr, FirstName, LastName, NrInParty, Package)
 Activity(Description, Duration)
 HasReservationFor(RoomNr, Date, GuestNr, Rate)
 DateDiscount(Date, Discount)
 Includes(Package, Description)

1. *Non-atomic attributes.* Assuming we wish to have *FirstName* and *LastName* for *Guest*, *Name* is not atomic. We add these attributes, making *Name* a compound attribute as Figure 7.15 shows.
2. *FDs whose left-hand sides are not keys.* Recognizing the FD *Type* \rightarrow *RackRate* as an FD whose left-hand side is not a key, we create a new entity set, *RoomType*. *Type* is a key attribute for *RoomType*, and *RackRate* is a regular attribute. Further, because the FD *Date* \rightarrow *Discount* is another FD whose left-hand side is not a key, we create another new entity set, *DateDiscount*. Its attributes are *Date* and *Discount*, with *Date* being a key attribute.
3. *Reducible n-ary relationship sets.* We can losslessly decompose the relationship set *has reservation for*. After adding the new entity set *DateDiscount* to this relationship set, the relationship set *has reservation for* has become a 5-ary relationship set. We can decompose it losslessly into two binary relationship sets and one ternary relationship set. Since the two new binary relationship sets equate to the existing relationship sets *has* and *is signed up for*, we discard them.
4. *Reducible cycles.* The cycle of relationship sets from *Guest* to *PackageDeal* to *Activity* and back to *Guest* is a reducible cycle. We can remove either *includes* or *is signed up for* because either is computable from the other two relationship sets. We cannot remove both, however, because we need each one to compute the other.

RESULT:

EX.NO:3.a	Implement the Database Using SQL Data Definition with Constraints
DATE:	

Aim:

Procedure:

DDL COMMANDS

1. The Create Table Command:- The CREATE TABLE statement is used to create a new table in a database.

Syntax:

CREATE TABLE table_name (column1datatype,column2datatype,column3datatype,);

Ex: CREATE TABLE TeachersInfo (TeacherIDint, TeacherNamevarchar(255), Addressvarchar(255), Cityvarchar(255), PostalCodeint, Countryvarchar(255), Salaryint);

2. The Alter Table: The ALTER TABLE statement is used to either add, modify or delete constraints and columns from a table.

Syntax:

ALTER TABLE table_name ADD column_name datatype;

Ex: ALTER TABLE Teachers Info ADD DateOfBirthdate;

3. Drop command: This command is used to delete the database, tables or columns.

The 'DROP SCHEMA' Statement: The DROP SCHEMA statement is used to drop the complete schema.

Syntax:

DROP SCHEMA schema_name;

Ex: DROP SCHEMA teachers;

The 'DROPTABLE' Statement: The DROP TABLE statement is used to drop the entire table with all its values.

Syntax:

DROP TABLE table name;

Ex: DROP TABLE TeachersInfo;

4. **Truncate command:** The TRUNCATE statement is used to delete the data which is present inside a table, but the table doesn't get deleted.

Syntax:

TRUNCATETABLEtable_name;

Ex: TRUNCATETABLETeachersInfo;

5. Rename Command: The RENAME statement is used to rename one or more tables or columns.

Syntax:

ALTER TABLE table_name RENAME TO new_table_name;

Ex: ALTER TABLE TeachersInfo RENAME TO InfoTeachers;

ALTER TABLE table_name RENAME COLUMN column_name TO new_column_name;

Ex: ALTER TABLE InfoTeachers RENAME COLUMN dateofbirth TO dob;

CREATION OF TABLE:

SYNTAX:

Create table <tablename> (column1 datatype, column2 datatype...);

EXAMPLE:

postgres> create table td (sno integer(5), sname varchar(20), age integer(5), sdobdate, sm1 integer(10), sm2 integer(10), sm3 integer(10));

Table created.

postgres> insert into std values(101, "AAA", 16, "03-jul-88", 80, 90, 98); 1 row created.

postgres> insert into std values(102, "BBB", 18, "04-aug-89", 88, 98, 90); 1 row created.

ALTER TABLE WITH ADD:

postgres> create table student(id integer(5), name varchar(10), game varchar(20));
Table created.

postgres> insert into student values(1, "mercy", "cricket");
1 row created.

SYNTAX:

alter table <tablename> add (col1 datatype, col2 datatype..);

EXAMPLE:

Postgres> alter table student add (age integer(4));

postgres> insert into student values(2, "sharmi", "tennis", 19);

DROP:

SYNTAX: drop table <tablename>;

EXAMPLE:

postgres> drop table student;

postgres> Table dropped.

TRUNCATE TABLE

SYNTAX: TRUNCATE TABLE<TABLENAME>;

Example: Truncate table stud;

3.a)CONSTRAINTS:

Create table table name (column_name1 data_ type constraints, column_name2 data_ type constraints...)

Example:

Create table company1 (id integer primary key not null, name varchar(20) not null, age integer not null, address varchar(50),salary integer);

a) **NOTNULL Constraint**–Ensures that a column cannot have NULL value.

For example, the following PostgreSQL statement creates a new table called COMPANY1and adds five columns, three of which, ID and NAME and AGE, specify not to accept NULL values

Example:

Create table company1(id integer primary key not null, name varchar(20) not null, age integer not null, address varchar(50), salary integer);

b) **UNIQUE Constraint**–Ensures that all values in a column are different.

For example, the following PostgreSQL statement creates a new table called COMPANY3and adds five columns. Here, AGE column is set to UNIQUE, so that you cannot have two records with same age.

Create table company3(id int primary key not null, name text not null, age integer not null, unique, address char(50),salary integer);

c) **PRIMARY Key** – uniquely identifies each row/record in a database table. If a table has a primary key defined on any field(s), then you cannot have two records having the same value of that field(s).

Create table company4 (id integer primary key not null, name varchar not null, age integer not null, address char(50),salary integer);

d) **Foreign Key:** A foreign key constraint specifies that the values in a column (or a group of columns) must match the values appearing in some row of another table.

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

Create table company6 (id int primary key not null, name integer not null, age integer not null, address varchar(50),salary integer);

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 the table COMPANY6.

Create table department1 (id integer primary key not null, dept varchar(50) not null, emp_id integer references company6(id));

e) **CHECK Constraint**–The CHECK constraint ensures that all values in a column satisfy certain conditions.

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 any SALARY as Zero.

Create table company5 (id int primary key not null, name varchar not null, age integer not null, address varchar(50), salary integer check(salary>0));

Creation of table with constraints:

a) Primary Key Constraints:

we create a new table called **Applicant**, which contains the **four columns**, such as **Id, username, password, and email**. And the **id** is the primary key, which uniquely classifies the id in the Applicant table

```
postgres>Create table applicant (id integer primary key, username
varchar(50),password varchar(30),email varchar(55));
```

Tablecreated

```
Postgres>insert into applicant(id, username, password, email) values (101,'mike',
'mikeross@','rmike@gmail.com'),(102,'john','smith#','johnsmith@gmail.com');
```

1rowcreated.

As we can see in the above screenshot, the **first insert command** will be executed because we have used the unique value for the primary key columns.

But if we want to insert one more value into the **Applicant** table using the same **id** value as 101, then PostgreSQL will issue an error.

```
insert into applicant(id, username, password, email) values (101,'ava jane',
'1234@56','ava@gmail.com);
```

Output:

After executing the above insert command, we will get the following error: The Duplicate key value violates unique constraint "applicant_pkey" as applicant_id=101 values already exist.

b) NotNullConstraint:

We create a new table called **Applicant**, which contains the **four columns**, such as **Id, username, password, and email**. And the **id** is the primary key, which uniquely classifies the id in the Applicant table

```
postgres>Create table applicant (id integer primary key, username varchar(50) not
null, password varchar(30),email varchar(55));
```

Tablecreated

```
Postgres>insert into applicant (id, username, password, email) values
(101,'mike','mikeross@','rmike@gmail.com'),(102,'john','smith#','johnsmith@gmail.
com');
```

1rowcreated.

As we can see in the above screenshot, the **first insert command** will be executed because we have used the unique value for the primary key columns. But if we want to insert one more value into the **Applicant** table using the same **username** value as “null”, then PostgreSQL will issue an error.

Insert into applicant (id, username, password, email) **values**
(103,'null','1234@56','ava@gmail.com);

Output:

After executing the above insert command, we will get the following error:
The username cannot have null values.

c) Unique constraint

```
postgresql>Create table applicant (id integer primary key, username varchar(50)
not null, password varchar(30),email varchar(55),unique (email));
Table created
```

```
Postgresql>insert into applicant (id,username,password,email) values
(101,'mike','mikeross@','rmike@gmail.com'),(102,'john','smith#','johnsmith@gm
ail.com');
```

1rowcreated.

As we can see in the above screenshot, the **first insert command** will be executed because we have used the unique value for the primary key columns.

But if we want to insert one more value into the **Applicant** table using the same **email** as,johnsmith@gmail.com”, to id103,then PostgreSQL will issue an error.

Insert into applicant (id, username, password, email) **values**
(103,'ajay','1234@56','johnsmith@gmail.com);

Output:

After executing the above insert command, we will get the following error:
The email cannot have same values.

d) Foreign Keyconstraint:

```
DROP TABLE IF EXISTS customers;
DROP TABLE IF EXISTS contacts;
```

```
CREATE TABLE customers (Customer_id INT
GENERATED ALWAYS AS IDENTITY, Customer_name VARCHAR(255)
NOT NULL,PRIMARY KEY(customer_id));
```



```
CREATE TABLE contacts (Contact_id INT GENERATED ALWAYS AS
IDENTITY, Customer_id INT Contact_name VARCHAR (255) NOT
NULL, Phone VARCHAR(15), Email VARCHAR(100), PRIMARY KEY
(contact_id), CONSTRAINT fk_customer FOREIGN KEY(customer_id)
REFERENCES customers (customer_id));
```

In this example, the customers table is the parent table and the contacts table is the child table. Each customer has zero or many contacts and each contact belongs to zero or one customer.

The customer_id column in the contacts table is the foreign key column that references the primary key column with the same name in the customers table.

The following foreign key constraint fk_customer in the contacts table defines the customer_id as the foreign key:

```
CONSTRAINT fk_customer FOREIGN KEY(customer_id)
REFERENCES customers(customer_id)
```

Because the foreign key constraint does not have the ON DELETE and ON UPDATE action, they default to NO ACTION.

NOACTION

The following inserts data into the customers and contacts tables:

```
INSERT INTO customers(customer_name)VALUES('BlueBirdInc'),('DolphinLLC');
INSERT
INTO contacts(customer_id,contact_name,phone,email)VALUES(1,'JohnDoe','(408)
-111-1234','john.doe@bluebird.dev'),(1,'JaneDoe','(408)-111-
1235','jane.doe@bluebird.dev'),(2,'DavidWright','(408)-222-
1234','david.wright@dolphin.dev');
```

The following statement deletes the customerid1 from the customerstable:

```
DELETE FROM customers WHERE customer_id= 1;
```

Because of the ON DELETE NO ACTION, PostgreSQL issues a constraint violation because the referencing rows of the customerid1 still exist in the contacts table:

```
ERROR: update or delete on table "customers" violates foreign key constraint
"fk_customer" on table "contacts" DETAIL: Key (customer_id)=(1) is still referenced from
table "contacts".
SQLstate:23503
```

e) Check constraint:

Example: Create table student t(regno integer(6), mark integer (3) constraint check (mark >=0 and mark <=100));

Table created.

Insert into student values (regno, mark)values(101,98),(102,87);
1rowcreated.

If we enter any value greater than 100 or less than 0, the postgresql shows error.

Queries:

Q1.CreateatablecalledEMPwiththefollowingstructure.NameType

EMPNOINTEGER(6)
ENAME
VARCHAR2(20)JOB
VARCHAR2(10)DEP
TNOINTEGER(3)SAL
INTEGER(5)

Allow NULL for all columns excepte name
and job.

Ans:

SQL>create table emp(empno integer(6),ename varchar(20)not null, job varchar(10)
not null, deptno number(3), sal integer(5));
Table created.

Q2:Addacolumnexperientetotheemptable.e
xperienceintegernullallowed.

Ans:SQL>alter table emp add(experience integer(2));
Table altered.

Q3:Createdepttablewiththefollowingstructure.NameType

DEPTNONUMBER(2)
DNAME
VARCHAR2(10)LOC
VARCHAR2(10)
Deptno as the primary key

Ans:

SQL> create table dept (deptno integer(2) primary key, dname
varchar2(10),locvarchar2(10));

Table created.

Q4: create the emp1table with ename and empno, add constraints to check the empno value while entering(i.e)empno>100.

Ans:

```
SQL> create table emp1(ename varchar2(10),empno number(6) constraint  
check(empno>100));
```

Table created.

Q6: drop a column experience to the emp table.

Ans:

```
SQL>alter table emp drop column experience;
```

Table altered.

Q7: Truncate the emp table and drop the dept table

```
Ans:SQL>truncate table emp;
```

Table truncated.

RESULT:

EX.NO:3.b	Implement the Database Using SQL Data Definition with Views
DATE:	

AIM:

Procedure:

Database views are created using the CREATE VIEW statement. Views can be created from a single table, multiple tables, or another view. To create a view, a user must have the appropriate system privilege according to the specific implementation.

```
postgres>CREATE TABLE EMPLOYEE (EMPLOYEE_NAME VARCHAR(10),
EMPLOYEE_NO INTEGER(8), DEPT_NAME VARCHAR(10), DEPT_NO
INTEGER(5), DATE_OF_JOINDATE);
```

Table created.

CREATE VIEW SYNTAX

```
CREATE[ORREPLACE][FORCE]VIEWviewname[(column-name,column-
name)]ASQuery[withcheckoption];
```

Include all not null attribute.

CREATIONOFVIEW

```
postgres>      CREATE      VIEW      EMPVIEW      AS      SELECT
EMPLOYEE_NAME,EMPLOYEE_NO,DEPT_NAME,DEPT_NO,DATE_OF_JOIN
FROMEMPLOYEE;
```

View Created.

DISPLAY VIEW:

```
SQL>SELECT*FROM EMPVIEW;
```

<u>EMPLOYEE</u> <u>E_N</u> -----	<u>EMPLOYEE</u> <u>_NO</u> -----	<u>DEPT_NAME</u> <u>ME</u> -----	<u>DEPT</u> <u>NO</u> -----
RAVI	124	ECE	89
VIJAY	345	CSE	21
RAJ	98	IT	22
GIRI	100	CSE	67

```
postgres>INSERTINTOEMPVIEWVALUES('SRI',120,'CSE', 67);
1ROWCREATED.
```

```
postgres>DROP VIEW EMPVIEW;
view dropped
```

RESULT:

AIM:

DMLCOMMANDS

DML commands are the most frequently used SQL commands and is used to query and manipulate the existing database objects. Some of the commands are Insert, Select, Update, Delete.

Insert Command: This is used to add one or more rows to a table. The values are separated by commas and the data types char and date are enclosed in apostrophes. The values must be entered in the same order as they are defined.

Select Commands: It is used to retrieve information from the table. It is generally referred to as querying the table. We can either display all columns in a table or only specify column from the table.

Update Command: It is used to alter the column values in a table. A single column may be updated or more than one column could be updated.

Delete command: After insert in grow in a table we can also delete them if required. The delete command consists of a from clause followed by an optional where clause.

Q1:Inserta single record in to dept table.

Ans:POSTGRES>insert into dept values(1,'IT','Tholudur');

1row created.

Q2: Insert more than a record into emp table using a single insert command.

Ans: POSTGRES > insert into emp values (&empno, '&ename', '&job',&deptno,&sal);

Enter value for empno:1

Enter value for ename: Mathi

Enter value for job:AP

Enter value for deptno:1

Enter value for sal:10000

old 1: insert into emp values (&empno, '&ename', '&job',&deptno,&sal)

new1:insert into emp values(1,'Mathi','AP',1,10000)

1rowcreated.

POSTGRES>/ Enter value for empno:2

Enter value for ename: Arjun

Enter value forjob:ASP

Enter value for deptno:2

Entervalueforsal:12000

old 1: insert into emp values (&empno, '&ename', '&job',&deptno,&sal)

new1:insert into emp values(2,'Arjun','ASP',2,12000)

1row created.

POSTGRES>/

Entervalueforempno:3

Enter value for ename:Gugan

Enter value for job: ASP

Enter value for deptno: 1

Entervalueforsal:12000

old1:insert into empvalues (&empno, '&ename', '&job',&deptno,&sal) new1:insert into empvalues(3,'Gugan','ASP',1,12000)

1row created.

Q3:Update the emp table to set the salary of all employees to Rs.15000/- who are working as ASP

Ans: POSTGRESQL > select* from emp;

EMPNO ENAME JOB DEPTNO SAL

1	Mathi	AP	1	10000
2	Arjun	ASP	2	12000
3	Gugan	ASP	1	12000

POSTGRESQL>update emp set sal=15000 where job='ASP';

2rows updated

.POSTGRESQLL>select*fromemp;

EMPNO ENAME JOB DEPTNO SAL

1	Mathi	AP	1	10000
2	Arjun	ASP	2	15000
3	Gugan	ASP	1	15000

Q4: select employee name, job from the emp table

Ans: POSTGRESQL > select ename, job from emp;

ENAMEJOB

Mathi AP

Arjun ASP

Gugan ASP

Karthik Prof

Akalya AP

Suresh Lect

6rows selected.

Q5: Delete only those who are working as lecturer

Ans: POSTGRES > select * from emp;

EMPNO ENAME JOB DEPT NO SAL

1	Mathi	AP	1	10000
2	Arjun	ASP	2	15000
3	Gugan	ASP	1	15000
4	Karthik	Prof	2	30000
5	Akalya	AP	1	10000
6	Suresh	Lect	1	8000

6rows selected.

POSTGRES>delete from emp where job='Lect';

1row deleted.

SQL>select*from emp;

EMPNO ENAME JOB DEPTNO SAL

1	Mathi	AP	1	10000
2	Arjun	ASP	2	15000
3	Gugan	ASP	1	15000
4	Karthik	Prof	2	30000
5	Akalya	AP	1	10000

Q6: List the records in the emp table order by salary in ascending order.

Ans:SQL>select*from emp order by sal;

EMPNO ENAME JOB DEPTNO SAL

1	Mathi	AP	1	10000
5	Akalya	AP	1	10000
2	Arjun	ASP	2	15000
3	Gugan	ASP	1	15000
4	Karthik	Prof	2	30000

Q7:List the records in the emp table order by salary in descending order.

Ans: SQL> select * from emp order by sal desc;

EMPNO	ENAME	JOB	DEPTNO	SAL
4	Karthik	Prof	2	30000
2	Arjun	ASP	2	15000
3	Gugan	ASP	1	15000
1	Mathi	AP	1	10000
5	Akalya	AP	1	10000

Q8: Display only those employees whose dept no is 30.

Solution: Use SELECT FROM WHERE syntax.

Ans: SQL> select * from emp where deptno=1;

EMPNO	ENAME	JOB	DEPTNO	SAL
1	Mathi	AP	1	10000
3	Gugan	ASP	1	15000
5	Akalya	AP	1	10000

Q9: Display dept no from the table employee avoiding the duplicated values.

Solution:

1. Use SELECT FROM syntax.
2. Select should include distinct clause for the deptno.
3. Ans: SQL>select distinct deptno from emp;

DEPTNO

1
2

RESULT:

EX.NO:5.a	Querying / Managing the Database Using-Stored Procedures / Functions
DATE:	

AIM:

PROCEDURE:

PostgreSQL **functions**, also known as Stored Procedures, allow you to carry out operations that would normally take several queries and round trips in a single function within the database. Functions allow database reuse as other applications can interact directly with your stored procedures instead of a middle-tier or duplicating code.

Functions can be created in a language of your choice like SQL, PL/pgSQL, C, Python, etc.

SYNTAX:

```
CREATE[ORREPLACE]FUNCTIONfunction_name(arguments)RETURNSreturn_datatypeAS $variable_name$DECLARE declaration;[...]BEGIN <function_body>[...]RETURN{ variable_name|value}END;LANGUAGEplpgsql;
```

Where,

- **function-name** specifies the name of the function.
- [ORREPLACE] option allows modifying an existing function.
- The function must contain a **return** statement.
- **RETURN** clause specifies that data type you are going to return from the function. The **return_datatype** can be a base, composite, or domain type, or can reference the type of a table column.
- **function-body** contains the executable part.
- The AS keyword is used for creating a standalone function.
- **Plpgsql** is the name of the language that the function is implemented in. Here, we use this option for PostgreSQL, it can be SQL, C, internal, or the name of a user-defined procedural language. For backward compatibility, the name can be enclosed by single quotes.

Example:

The following example illustrates creating and calling a standalone function. This function returns the total number of records in the COMPANY table. We will use the COMPANY table, which has the following records–

```
postgres# select * from company; id |name |age|address|salary
```

```
____+____+___+____+_____
1|Paul|32|California|20000
2|Allen|25|Texas    |15000
3|Teddy|23|Norway   |20000
4|Mark|25|Rich-Mond|65000
5|David|27|Texas     |85000
6|Kim|22|South-Hall|45000
7|James|24|Houston  |10000
(7rows)
```

```
POSTGRES# create table company(id integer, name varchar(20),age integer, address
varchar(20),salary integer(10));
```

Table created.

```
POSTGRES#Insertintocompany(id,name,age,salary)company(id,name,age,salary)value
s(1,"Paul",32,"California",20000),(2,"Allen",25,"Texas",15000),(3,"Teddy",23,"Norwa
y",20000),(4,"Mark",25,"RichMond",65000),(5,"David",27,"Texas",85000),(6,"Kim",22,"S
outhHall",45000),(7,"James",24,"Houston,10000);
```

7rows inserted.

Output:

```
POSTGRES# select * from student;
```

```
postgres# select * from company; id |name|age |address|salary
```

```
+++_____+_____
```

```
1|Paul|32|California|20000
2|Allen|25|Texas    |15000
3|Teddy|23|Norway|20000
4|Mark|25|Rich-Mond|65000
5|David|27|Texas    |85000
6|Kim|22|South-Hall|45000
7|James|24|Houston  |10000
```

Function total Records() is as follows–

```
CREATE OR REPLACE FUNCTION total Records()
```

```
RETURNS integer AS $total $declare total integer;
```

```
BEGIN SELECT count(*) in tototal FROM COMPANY;
```

```
RETURN total;
```

```
END;
```

```
$total$ LANGUAGE plpgsql;
```

When the above query is executed, the result would be –

```
POSTGRES# CREATE FUNCTION
```

Now, let us execute a call to this function and check the records in the COMPANY table

```
POSTGRES=#selecttotalRecords();
```

When the above query is executed, the result would be –totalrecords

```
-----
7
```

(1row)

RESULT:

AIM:**PROCEDURE:**

A trigger function is similar to a regular user defined function. However, a trigger function does not take any arguments and has a return value with the type trigger.

To create a new trigger in PostgreSQL, you follow these steps:

- First, create a trigger function using CREATE FUNCTION statement.
- Second, bind the trigger function to a table by using CREATE TRIGGER statement.

Create trigger function syntax

```
CREATE FUNCTION trigger_function()  
RETURNS TRIGGER LANGUAGE PLPGSQL AS  
$$  
BEGIN -- trigger logic  
END;  
$$
```

Notice that you can create a trigger function using any languages supported by PostgreSQL. Here, we will use PL/pgSQL.

For example, OLD and NEW represent the states of the row in the table before or after the triggering event.

Once you define a trigger function, you can bind it to one or more trigger events such as INSERT, UPDATE, and DELETE.

EXAMPLE:

The following statement create new table called employees DROP TABLE IF EXISTS employees;

```
CREATE TABLE employees (id INTEGER, first_name VARCHAR(40) NOT  
NULL, last_name VARCHAR(40) NOT NULL, PRIMARY KEY (id)  
);
```

Suppose that when the name of an employee changes, you want to log the changes in a separate table called employee_audits:

```
CREATE TABLE employee_audits (id INTEGER, employee_id INT NOT
NULL, last_name VARCHAR(40) NOT NULL, changed_on TIMESTAMP(6) NOT
NULL
);
```

First, create a new function called log_last_name_changes:

```
CREATE OR REPLACE FUNCTION log_last_name_changes() RETURNS TRIGGER
LANGUAGE PLPGSQL AS
$$BEGIN
    IF NEW.last_name<>OLD.last_nameTHEN
        INSERT INTO
            employee_audits(employee_id,last_name,changed_on)VALUES(OLD.id,
            OLD.last_name,now());
    ENDIF;
    RETURN NEW;
END;
$$
```

The function inserts the old last name into the employee_audits table including employee id, last name, and the time of change if the last name of an employee changes.

The OLD represents the row before update while the NEW represents the new row that will be updated. The OLD.last_name returns the last name before the update and the NEW.last_name returns the new last name.

Second, **bind the trigger function** to the employees table. The trigger name is last_name_changes.

Before the value of the last_name column is updated, the trigger function is automatically invoked to log the changes.

```
CREATE TRIGGER last_name_changes BEFORE UPDATE ON employees FOR EACH
ROW EXECUTE PROCEDURE log_last_name_changes();
```

Third, insert some rows into the employees table:

```
INSERT INTO employees (first_name, last_name) VALUES ('John','Doe');
INSERT INTO employees (first_name,last_name) VALUES('Lily','Bush');
```

Fourth, examine the contents of the employees table:

```
SELECT*FROM employees;
```

<u>ID</u>	<u>FIRSTNAME</u>	<u>LASTNAM</u> <u>E</u>
1	John	Doe
2	Lily	Bush

Suppose that LilyBush changes her last name to LilyBrown.

Fifth, update Lily's last name to the new one:

```
UPDATE employees SET last_name = 'Brown'WHEREID= 2;
```

Sixth, check if the last name of Lily has been updated:

```
SELECT*FROM employees;
```

<u>ID</u>	<u>FIRSTNAME</u>	<u>LASTNAME</u>
1	John	Doe
2	Lily	Brown

As you can see from the output, Lily's last name has been updated.

Seventh, verify the contents of the employee_audits table:

```
SELECT*FROM employee_audits;
```

id	employee_id	last_name	changed_on
1	1	Brown	2022-05-06 5:30:04

RESULT:

Aim:**Procedure:**

Normalization is one of the key tenets in relational model design. It is the process of removing redundancy in a table so that the table is easier to modify. It usually involves dividing an entity table into two or more tables and defining relationships between the tables. The objective is to isolate data so that additions, deletions, and modifications of an attribute can be made in just one table and then propagated through the rest of the database via the defined relationships.

Normalization utilizes association among attributes within an entity table to accomplish its objective. Since an ERD also utilizes association among attributes as a basis to identify entity type structure, it is possible to apply normalization principles during the conceptual data modeling phase.

Performing normalization during ERD development can improve the conceptual model, and speed its implementation.

Application of normalization:

Application of normalization principles toward ERD development enhances these guidelines. To understand this application (i) representation of dependency concepts in an ERD is outlined, followed by (ii) representation of normal forms toward the development of entity type structure.

Guidelines for identification of various dependencies are avoided in the paper so as to focus more on their application. Only the first four normal forms and the Boyce-Codd normal forms are considered.

Representation of Dependencies

Functional dependency in an entity type occurs if one observes the association among the entity identifier and other attributes as reflected in an entity instance. Each entity instance represents a set of values taken by the non entity identifier attributes for each primary key (entity identifier) value. So, in a way an entity instance structure also reflects an application of the functional dependency concept. For example, the Student entity type of Figure 2 can represent the functional dependency $SID \rightarrow Name, Street, City, Zip$.

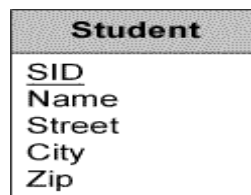


Figure 2

Each entity instance will now represent the functional dependency among the entity attributes as shown in Figure3.

Student	Student	Student
<u>100</u> Jack Taylor 212 S. Page St. Louis 63132	<u>200</u> Tom Larson 444 N. Monroe Springfield 65807	<u>300</u> Kay Beth 4212 S. Normal St. Louis 63132

Figure3

During requirement analysis, some entity types may be identified through functional dependencies, while others may be determined through database relationships. For example, the statement, “A faculty teaches many offerings but an offering is taught by one faculty” defines entity type Faculty and Offerings. Another important consideration is to distinguish when one attribute alone is the entity identifier versus a composite entity identifier. A composite entity identifier is an entity identifier with more than one attribute. A functional dependency in which the determinant contains more than one attribute usually represents a many-to-many relationship, which is more addressed through higher normal forms.

Transitive dependency in an entity type occurs if non entity identifier attributes have dependency among themselves. For example, consider the modified Student entity type as shown in Figure4.

Student
<u>SID</u> Name Street BuildingName Fee

Figure4

In this entity type, suppose there is a functional dependency Building Name? Fee. Existence of Building Name ? Fee dependency implies that the value assigned to the Fee attribute is fixed for distinct Building Name attribute values. In other words, the Fee attribute values are not specific to the SID value of a student, but rather the Building Name value.

Normalized ERD

Now we utilize the representation of dependency concepts in ERD toward their use in the application of normal forms. Each normal form rule and its application is outlined.

First Normal Form (1NF)

The first normal form rule is that there should be no nesting or repeating groups in a table. Now an entity type that contains only one value for an attribute in an entity instance ensures the application of first normal form for the entity type. So in any entity type with an entity identifier is by default in first normal form. For example, the entity type Student in Figure 2 is in first normal form.

Second Normal Form (2NF)

The second normal form rule is that the key attributes determine all non-key attributes. A violation of second normal form occurs when there is a composite key, and part of the key determines some non-key attributes. The second normal form deals with the situation when the entity identifier contains two or more attributes, and the non-key attribute depends on part of the entity identifier. For example, consider the modified entity type Student as shown in Figure 5. The entity type has a composite entity identifier of SID and City attributes.

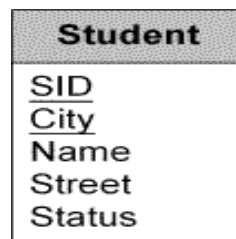


Figure 5

An entity instance of this entity type is shown in Figure 6. Now, if there is a functional dependency City → Status, the entity type structure will violate the second normal form.

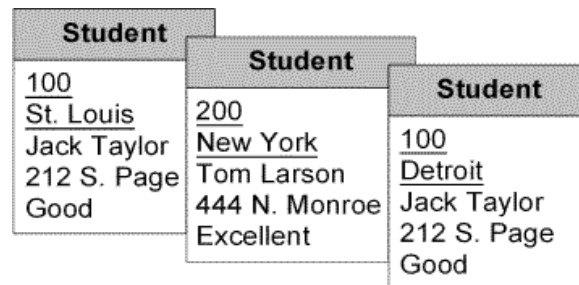


Figure 6

To resolve the violation of the second normal form a separate entity type City with one-to-many relationship is created as shown in Figure 10. The relationship cardinalities can be further modified to reflect organizational working. In general, the second normal form violation can be avoided by ensuring that there is only one attribute as an entity identifier.

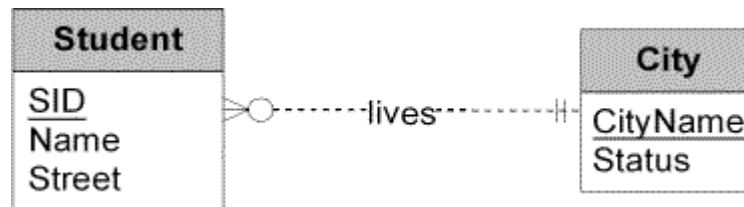


Figure7

Third Normal Form (3NF):

The third normal form rule is that the non-key attributes should be independent. This normal form is violated when there exists a dependency among non-key attributes in the form of a transitive dependency. For example consider the entity type Student as shown in Figure. In this entity type, there is a functional dependency Building Name? Fee that violates the third normal form

Transitive dependency is resolved by moving the dependency attributes to a new entity type with one-to-many relationship. In the new entity type the determinant of the dependency becomes the entity identifier. The resolution of the third normal form is shown in Figure 8. The relationship cardinalities can be further modified to reflect organizational working.



Figure8

Boyce-Codd Normal Form(BCNF):

The Boyce-Codd normal form (BCNF) extends the third normal form. The Boyce-Codd normal form rule is that every determinant is a candidate key. Even though Boyce-Codd normal form and third normal form generally produce the same result, Boyce-Codd normal form is a stronger definition than third normal form. Every table in Boyce-Codd normal form is by definition in third normal form. Boyce-Codd normal form considers two special cases not covered by third normal form:

1. Part of a composite entity identifier determines part of its attribute, and
2. A non entity identifier attribute determines part of an entity identifier attribute.

These situations are only possible if there is a composite entity identifier, and dependencies exist from a non-entity identifier attribute to part of the entity identifier. For example, consider the entity type Student Concentration as shown in Figure 9. The entity type is in third normal form, but since there is a dependency FacultyName? MajorMinor, it is not in Boyce-Codd normalform.

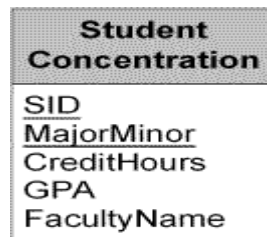


Figure9

To ensure that Student Concentration entity type stays in Boyce-Codd normal form, another entity type Faculty with one-to-many relationship is constructed as shown in Figure10. The relationship cardinalities can be further modified to reflect organizational working.



Figure10

RESULT:

EX.NO:7

DATE:

Develop Database Applications Using IDE / RAD Tools

AIM:

Description:

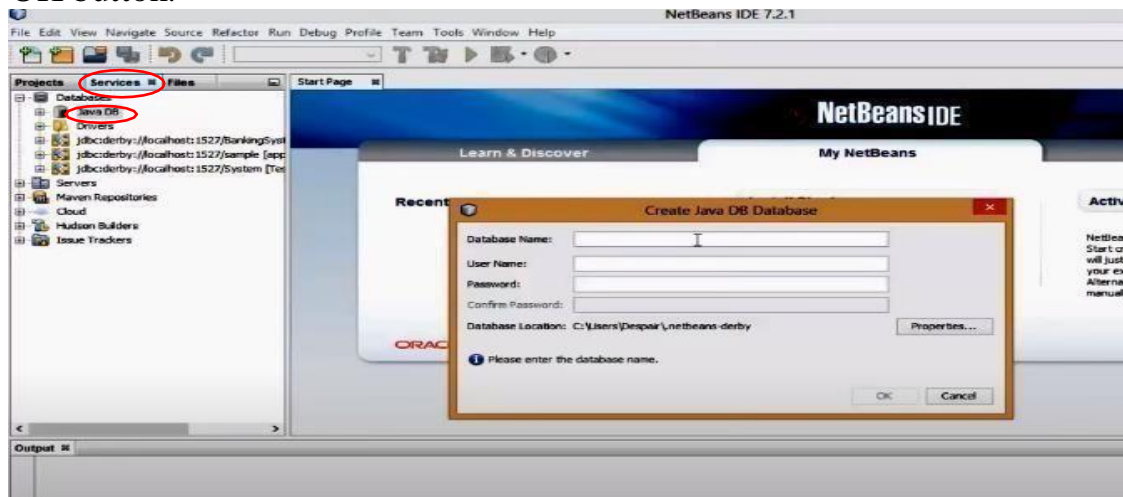
Employee Management System is used to maintain the detail of employees of an organization. HR can view the details department wise. HR can delete the employees. HR can update the employees details. HR can add new employees.

IMPLEMENTATION:

Creating Application using Net beans

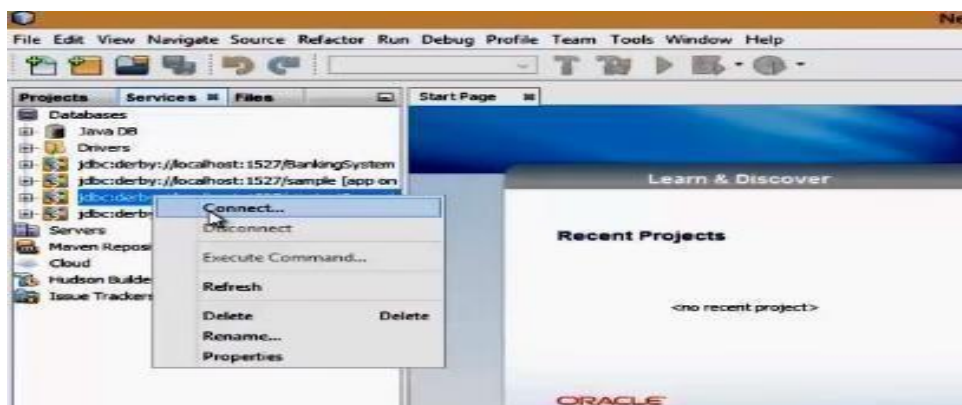
Step1:clickservices>database>javaDB>rightclick>createdatabase. Then create java DB data base box is popped.

Step2:give databasename>give username as PECAI&DS>set password as PECAI&DS>click OK button.



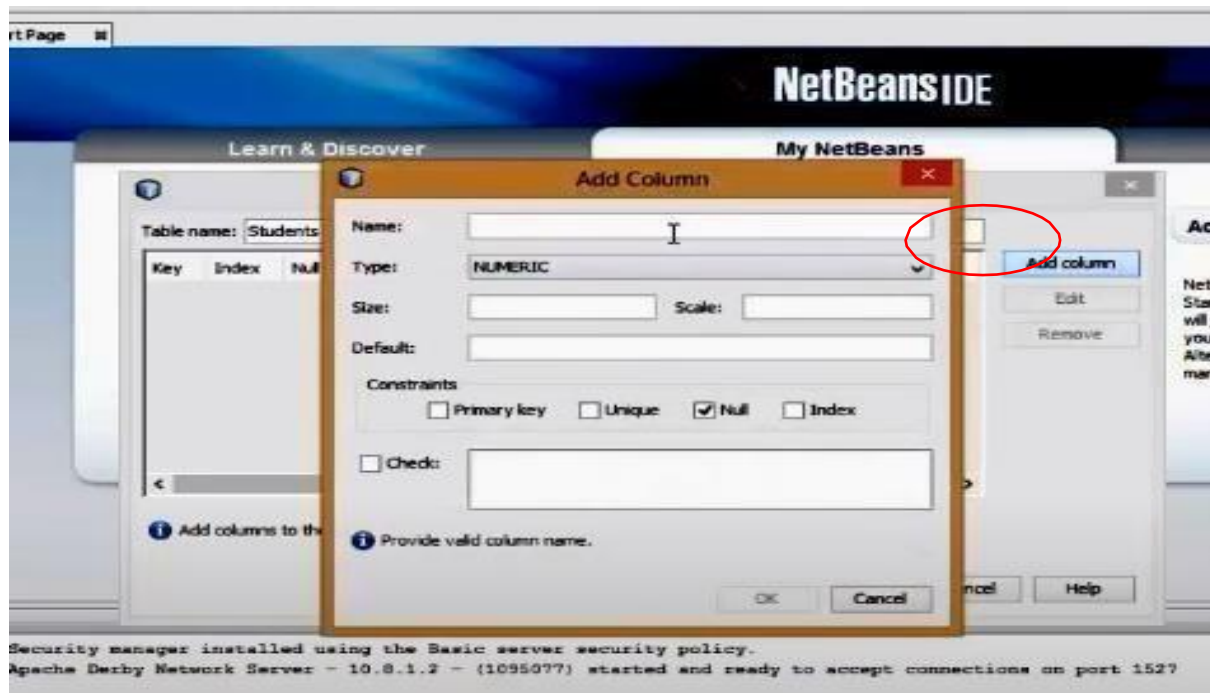
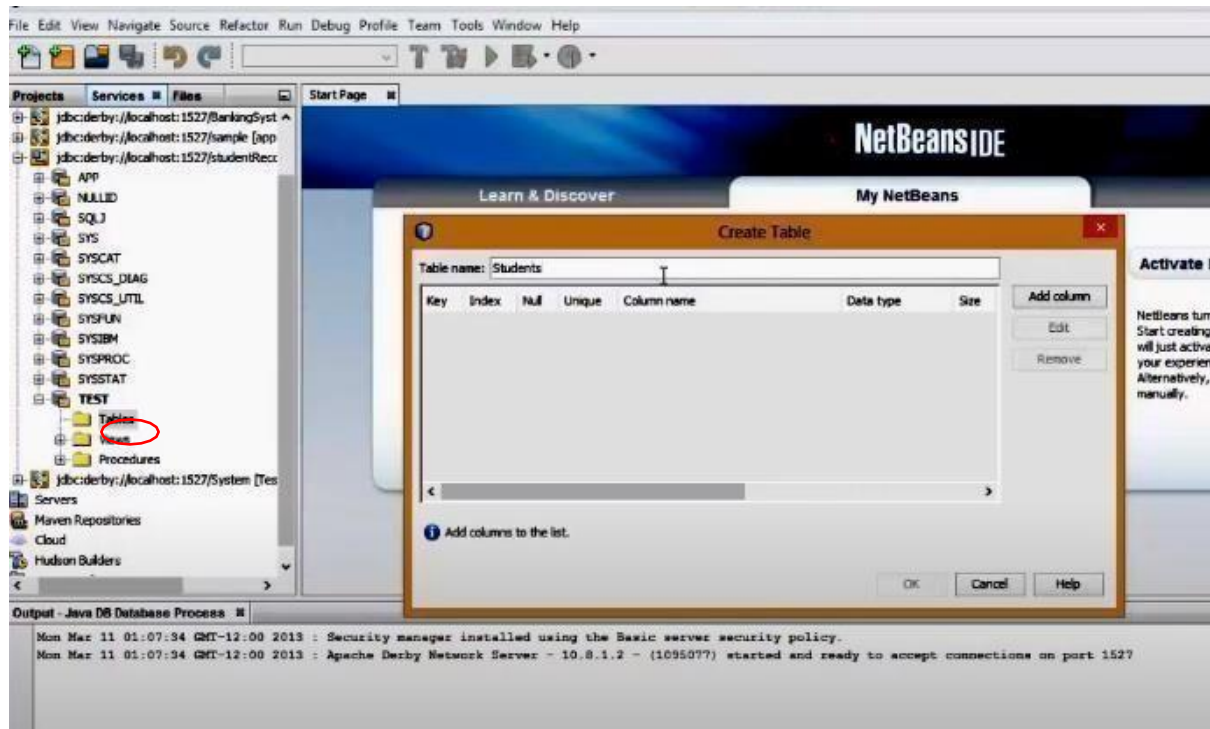
The java DB database will start to process. Once the process is stopped, the javaDB will be created.

Step3:select the created one>rightclick>connect



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

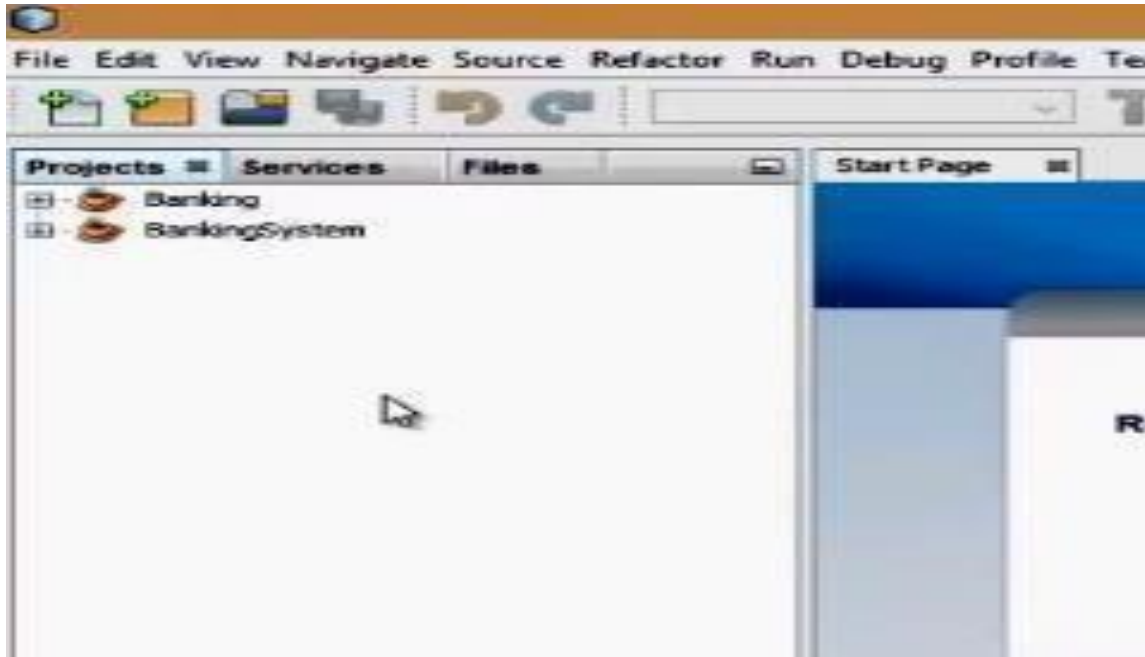
Step4:selecttest>tables>createtable>give tablename



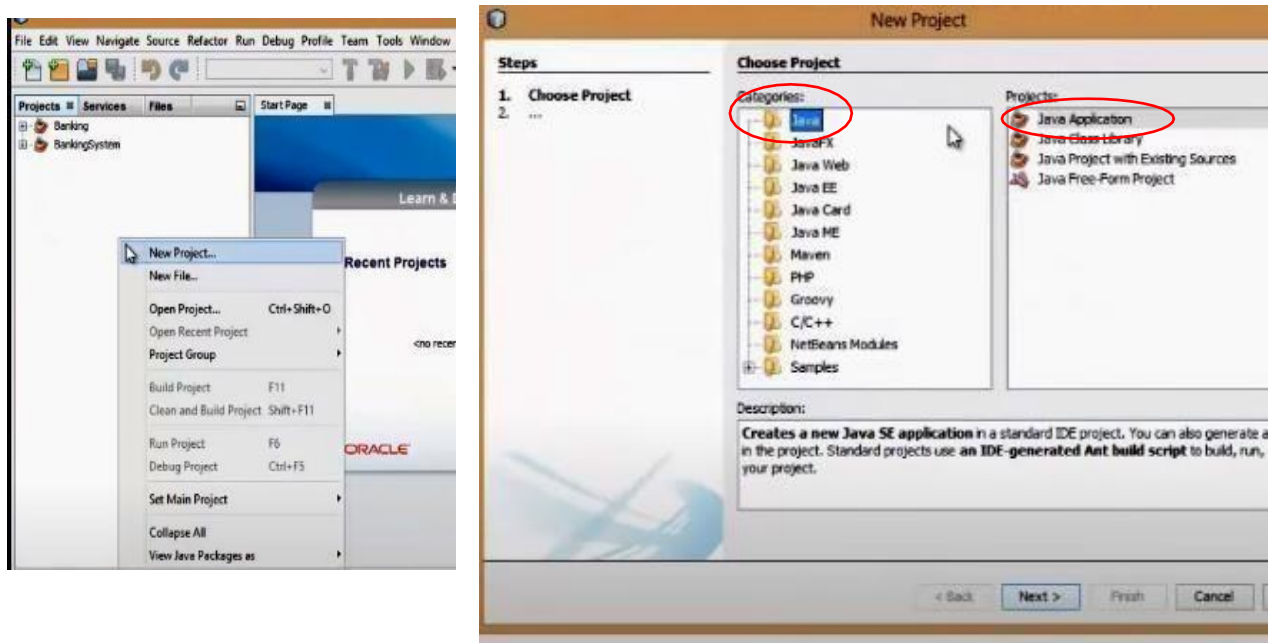
Step5:clickaddcolumn button>give name>select type>give size>then click OK button

If you want more columns, you can repeat this step5 and add columns. Then the table will be created.

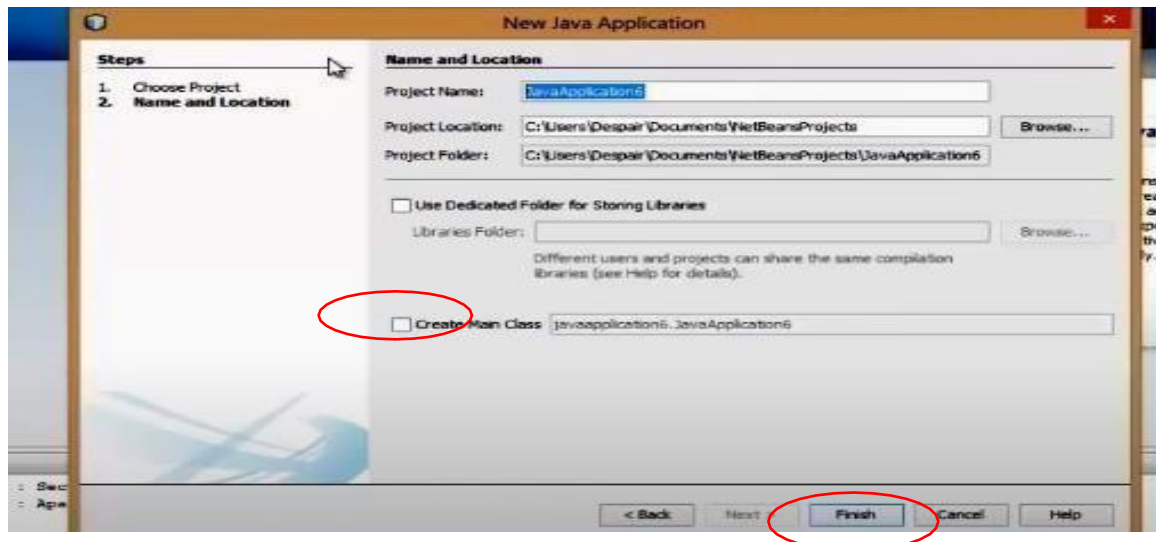
Step 6: at the top corner, select project. In the empty space under project, right click.



Step7: select new project > java > java application > click NEXT

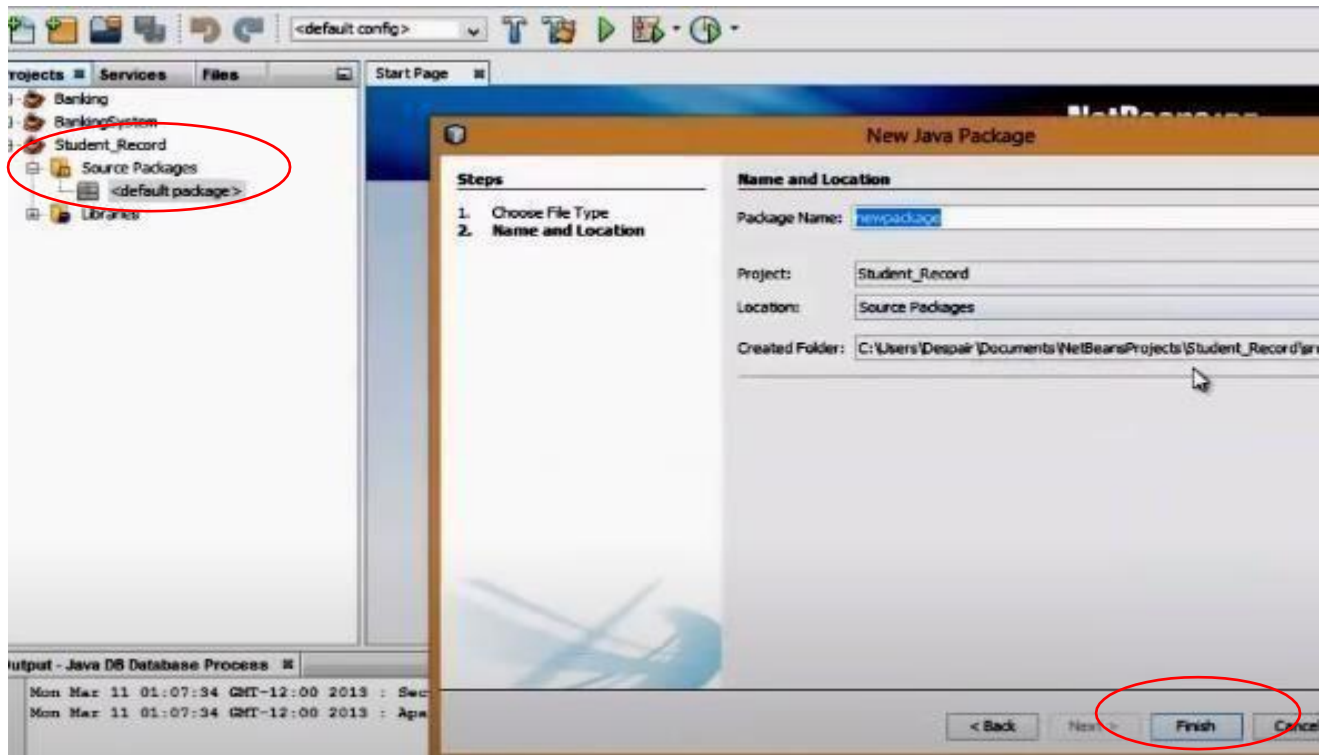


Step8:giveprojectname>removetickincreatemainclass>clickFINISH



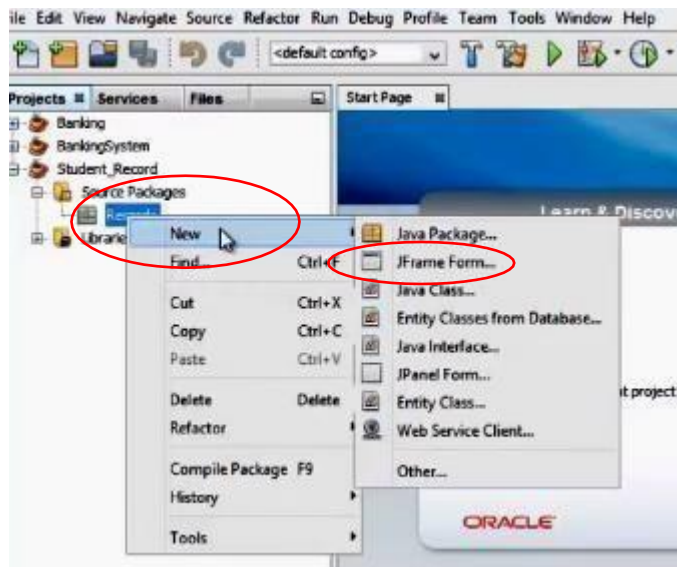
So, the project will be created under the project menu.

Step9:click the project you created. Under that, click source packages>defaultpackage>new>javapackage>give package name> click finish.



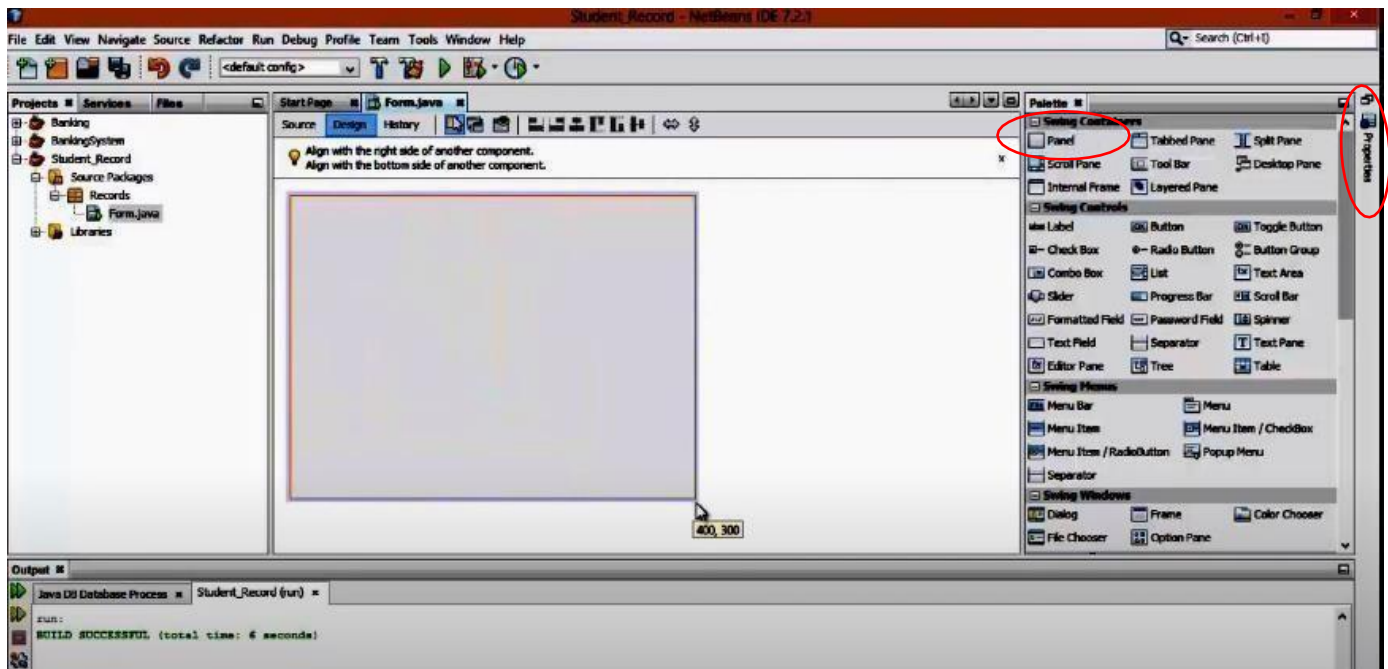
The package will be created in the source package under your project.

Step10:clickthepackagecreatedundersourcepackage>new>JFrameForm>



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

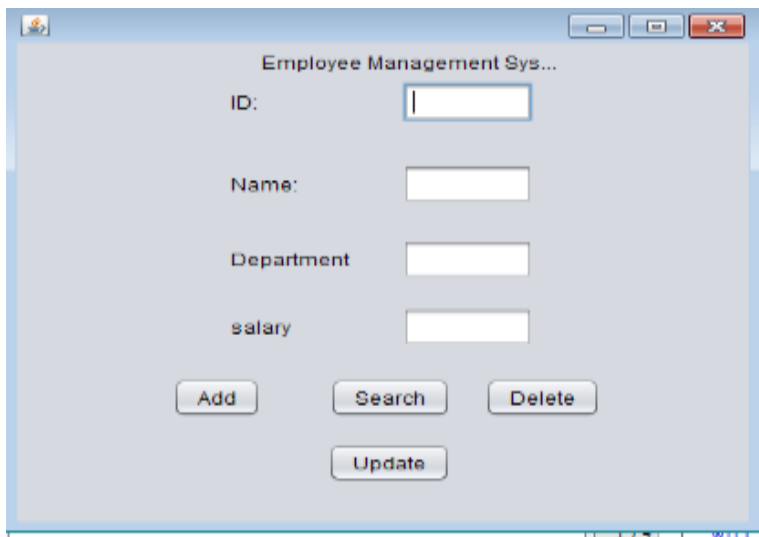
Step11:This window will be shown. Click panel (if you want you can resize it).Then click properties and start doing designing your application.



Tables:

Employee(id integer primary key , name varchar(20) , department varchar(20) , salary number)SQL>select*fromemployee;

ID	NAME	DEPT	SALARY
1	rekha	cse	30000
2	renu	cse	70000
3	elamathi	ece	90000
4	rahul	ece	79990

**Connectivity Coding:**

```
import java.sql.Connection; import java.sql.DriverManager;
public class PostgreSQLJDBC {
    public static void main(String args[]) { Connection c = null;
        try {
            Class.forName("org.postgresql.Driver"); c = DriverManager
                .getConnection("jdbc:postgresql://localhost:5432/testdb",
                    "postgres", "123");
        } catch (Exception e) { e.printStackTrace();
            System.err.println(e.getClass().getName() + ":
                " + e.getMessage()); System.exit(0);
        }
        System.out.println("Opened database successfully");
    }
}
```


Before you compile and run above program, findpg_hba.conf file in your PostgreSQL installation directory and add the following line—

IPv4localconnections:

host all all 127.0.0.1/32 md5

You can start/restart the postgres server incase it is not running using the following command—[root@host] #service postgresql restart

Stopping postgresql service: [OK]

Starting postgresql service: [OK]

Now, let us compile and run the above program to connect with testdb. Here, we are using postgres as user ID and 123 as password to access the database. You can change this as per your database configuration and setup. We are also assuming current version of JDBCdriverpostgresql-9.2-1002.jdbc3.jar is available in the current path.

C:\JavaPostgresIntegration>javacPostgreSQLJDBC.java

C:\JavaPostgresIntegration>java -cp c:\tools\postgresql-9.2-1002.jdbc3.jar;C:\JavaPostgresIntegrationPostgreSQLJDBC

Create a Table

```
import java.sql.Connection; import java.sql.DriverManager;
import java.sql.Statement; public class PostgreSQLJDBC {
    public static void main( String args[] ) { Connection c=
        null;
        Statement stmt=null; try{
            Class.forName("org.postgresql.Driver");
            c=DriverManager.getConnection("jdbc:postgresql://localhost:5432/testdb", "manisha", "123"); System.out.println("Open ed databases successfully");
            stmt=c.createStatement();
            String sql="CREATE TABLE COMPANY"(ID INTEGER PRIMARY KEY NOT NULL, "+"NAME TEXT NOT NULL, "+"ADDRESS CHAR(50), "+"SALARY REAL)";
            stmt.executeUpdate(sql); stmt.close();
            c.close();
        } catch (Exception e) {
            System.err.println(e.getClass().getN ame()+": "+e.getMessage()); System.exit(0)
        }
        System.out.println("Table created successfully");
    }
}
```

ADD Operation

```
import java.sql.Connection;
import java.sql.DriverManager; import java.sql.Statement;
public class PostgreSQLJDBC{
    public static void main(String args[]) {Connection c=
        null;
        Statement stmt=null; try{
            Class.forName("org.postgresql.Driver"); c=DriverManager
                .getConnection("jdbc:postgresql://localhost:5432/testdb",
                    "manisha", "123");
            c.setAutoCommit(false); System.out.println("Opened database
                successfully");
            stmt=c.createStatement();
            String sql="INSERT INTO COMPANY (ID,NAME,ADDRESS,SALARY) "
                +"VALUES (1,'Paul','California',20000.00);";
            stmt.executeUpdate(sql);
            sql="INSERT INTO COMPANY (ID,NAME,ADDRESS,SALARY) "
                +"VALUES (2,'Allen','Texas',15000.00);";
            stmt.executeUpdate(sql);
            sql="INSERT INTO COMPANY (ID,NAME,ADDRESS,SALARY) "
                +"VALUES (3,'Teddy','Norway',20000.00);";
            stmt.executeUpdate(sql);
            sql="INSERT INTO COMPANY (ID,NAME,ADDRESS,SALARY) "
                +"VALUES (4,'Mark','Rich-Mond',65000.00);";
            stmt.executeUpdate(sql);
            stmt.close();
            c.commit();
            c.close();
        } catch (Exception e) {
            System.err.println(e.getClass().getName()+": "+e.getMessage());
            System.exit(0);
        }
        System.out.println("Records created successfully");
    }
}
```

SELECT Operation

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet; import java.sql.Statement;
public class PostgreSQLJDBC{
    public static void main( String args[] ) {Connection c=
        null;
        Statement stmt=null; try{
            Class.forName("org.postgresql.Driver"); c=DriverManager
                .getConnection("jdbc:postgresql://localhost:5432/testdb",
                    "manisha","123"); c.setAutoCommit(false);
            System.out.println("Opened database successfully"); stmt=c.createStatement();
            ResultSet rs=stmt.executeQuery("SELECT *FROM COMPANY;"); while (rs.next()) {
                int id=rs.getInt("id");
                String name=rs.getString("name"); int age =
                    rs.getInt("age");
                String address = rs.getString("address"); float salary
                    = rs.getFloat("salary"); System.out.println(
                        "ID = "+ id); System.out.println("NAME="+name
                    ); System.out.println("ADDRESS
                    =" +address); System.out.println("SALARY="+salary); System
                    .out.println();
            }
            rs.close();
            stmt.close();
            c.close();
        } catch (Exception e) {
            System.err.println( e.getClass().getName()+" : "+
                e.getMessage() ); System.exit(0);
        }
        System.out.println("Operation done successfully");
    }
}
```

UPDATE Operation

```
import java.sql.Connection; import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement; public class PostgreSQLJDBC {
    public static void main( String args[] )
    { Connection c = null;
      Statement stmt = null; try {
        Class.forName("org.postgresql.Driver"); c = DriverManager
          .getConnection("jdbc:postgresql://localhost:5432/testdb",
            "manisha", "123"); c.setAutoCommit(false);
        System.out.println("Opened database successfully"); stmt = c.createStatement();
        String sql = "UPDATE COMPANY SET SALARY=25000.00 WHERE ID=1;";
        stmt.executeUpdate(sql); c.commit();
        ResultSet rs = stmt.executeQuery("SELECT * FROM COMPANY;"); while (rs.next()) {
            int id = rs.getInt("id");
            String name = rs.getString("name"); int age = rs.getInt("age");
            String address = rs.getString("address"); float salary = rs.getFloat("salary");
            System.out.println("ID = " + id); System.out.println("NAME = " + name);
            System.out.println("ADDRESS = " + address); System.out.println("SALARY = " + salary);
            System.out.println();
        }
        rs.close();
        stmt.close();
        c.close();
    } catch (Exception e) {
        System.err.println(e.getClass().getName() + ": " + e.getMessage());
        System.exit(0);
    }
    System.out.println("Operation done successfully");
}
```

DELETE OPERATION:

```
import java.sql.Connection;
import java.sql.DriverManager; import java.sql.ResultSet;
import java.sql.Statement;
public class PostgreSQLJDBC6{
    public static void main( String args[] ) {Connectionc=
        null;
        Statement stmt = null; try{
            Class.forName("org.postgresql.Driver"); c=DriverManager
                .getConnection("jdbc:postgresql://localhost:5432/testdb"
                    , "manisha", "123");
            c.setAutoCommit(false); System.out.println("Opened
                database successfully"); stmt=c.createStatement();
            String sql = "DELETE from COMPANY where ID =
                2;"; stmt.executeUpdate(sql);
            c.commit();
            ResultSet rs=stmt.executeQuery("SELECT
                *FROM COMPANY;"); while (rs.next()){
                int id=rs.getInt("id");
                String name=rs.getString("name"); int age =
                rs.getInt("age");
                String address= rs.getString("address"); float salary
                    = rs.getFloat("salary"); System.out.println(
                    "ID = "+ id); System.out.println( "NAME = " +
                    name ); System.out.println( "ADDRESS = " + address
                    ); System.out.println( "SALARY = " + salary
                    ); System.out.println();
                }
            rs.close();
            stmt.close();
            c.close();
        } catch (Exception e) {
            System.err.println(e.getClass().getName()+" : "+e.getMessag
                e()); System.exit(0);
            System.out.println("Operation done successfully");
        }
    }
```

RESULT:

EX.NO:8	Database Design Using EER-to-ODB Mapping / UML Class Diagrams
DATE:	

AIM:

PROCEDURE:

The UML class diagram is also known as object modeling. It is a static analysis diagram. These diagrams show the static structure of the model. A class diagram is a connection of static model elements, such as classes and their relationships, connected as a graph to each other and to their contents.

A class diagram describes the type of objects in system and various kinds of relationships that exists among them.

Class diagrams and collaboration diagrams are alternate representations of object models.

During analysis, we use class diagram to show roles and responsibilities of entities that provide email client system behaviors design. We use to capture the structure of classes that form the email client system architecture.

A class diagram is represented as:

<<Classname>>

<<Attribute1>>

<<Attributen>>

<<Operation()>>

Relationship used:

A change in one element affects the other

Generalization:

It is a kind of relationship

Diagram of one class:

- Class name in top of box
 - write<<interface>> on top of interfaces' names
 - use *italics* for an *abstract class* name
 - attributes(optional)
 - should include all fields of the object
 - operations/methods(optional)
- omit trivial (get/set) methods
 - a. but don't omit any methods from an interface!
- Should not include inherited methods

Rectangle
- width: int - height: int / area: double
+ Rectangle(width: int, height: int) + distance(r: Rectangle): double

Student
-name:String -id:int <u>-totalStudents:int</u>
#getID():int +getName():String ~getEmailAdress():String +getTotalStudents():int

Class attributes(=fields)

- attributes(fields,instancevariables)
 - *visibilityname:type[count]=default_value*
 - visibility:
 - + public
 - # protected
 - private
 - ~ package(default)
 - / derived
 - underline static attributes
 - **derived attribute**:notstored,butcanbecomputedfromother attributevalues
 - “specificationfields“fromCSE331
 - Attribute example:
 - balance:double=0.00

Rectangle
- width: int - height: int / area: double
+ Rectangle(width: int, height: int) + distance(r: Rectangle): double

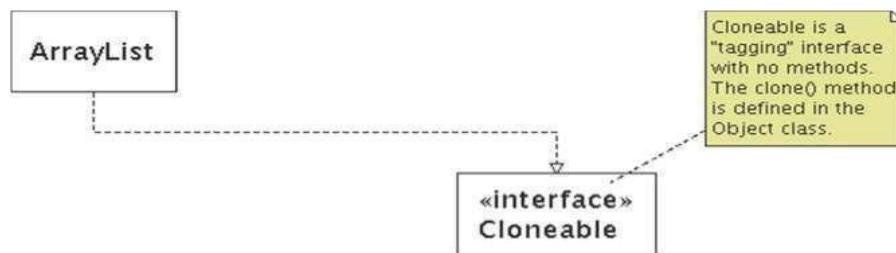
Student
-name:String -id:int <u>-totalStudents:int</u>
#getID():int +getName():String ~getEmailAdress():String +getTotalStudents():int

Class operations/methods

- operations/methods
 - *visibilityname(parameters):return_type*
 - visibility:
 - + public
 - # protected
 - private
 - ~ package(default)
 - underlinestaticmethods
 - parametertypeslistedas(name:type)
 - omitreturn_typeonconstructorsand whenreturntypeisvoid
 - methodexample:
 - +distance(p1:Point,p2:Point): double

Comments

- represented as a folded note, attached to the appropriate class/method/etc by a dashed line.
- generalization: an inheritance relationship

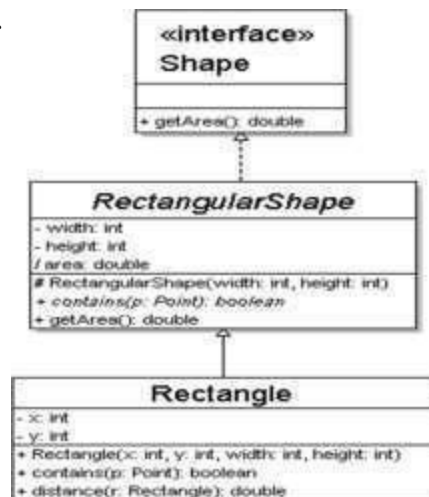


Relationships between classes

- inheritance between classes
- interface implementation
 - association: a usage relationship
- dependency
- aggregation
- composition

Generalization (inheritance) relationships

- hierarchies drawn top-down
- arrows point upward to parent
- line/arrow styles indicate whether parent is a(n):
 - class: solid line, black arrow
 - abstract class: solid line, white arrow
 - interface: dashed line, white arrow
- often omit trivial / obvious generalization relationships, such as drawing the Object class as a parent.



Associational(usage)relationships

multiplicity (how many are used)

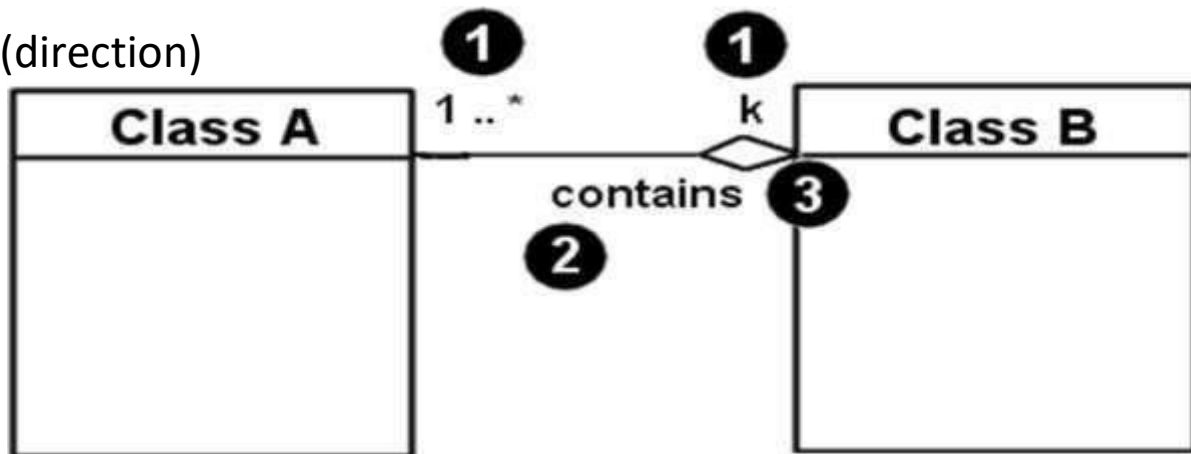
* =>0,1,ormore

1 =>1exactly

2..4=>between2and4,inclusive3..*=>3ormore(alsowrittenas“3..”)

Name(what relationship the objects have)Navigability

(direction)

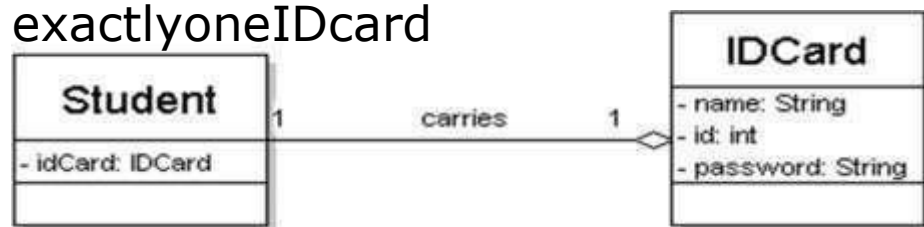


Multiplicity of associations

- one-to-one

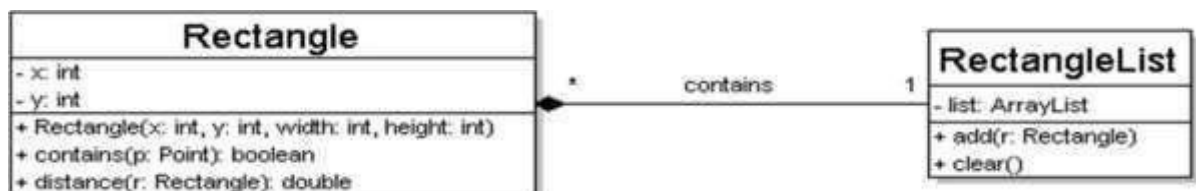
each student must carry

exactly one ID card



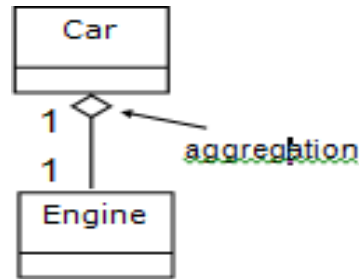
- one-to-many

one rectangle list can contain many rectangles

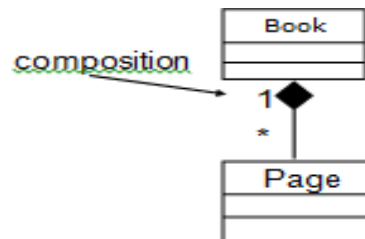


Associationtypes

- **aggregation**: “ispartof”
 - symbolized by a clear white diamond

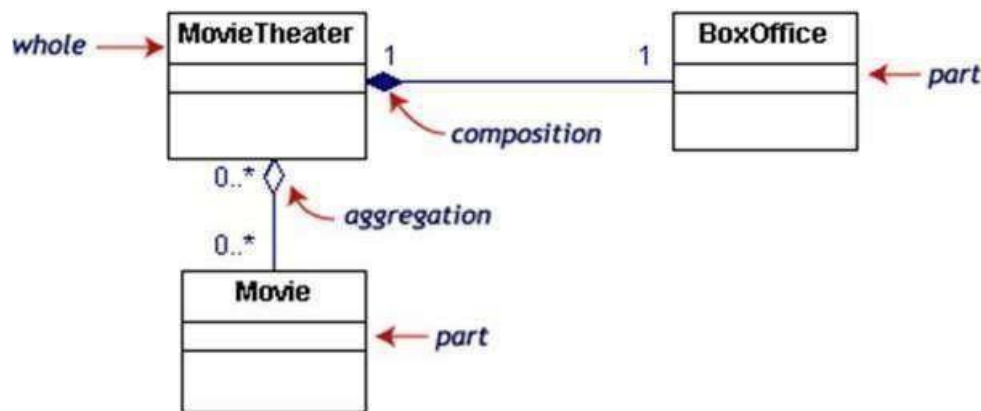


- composition: “is entirely made of”
 - stronger version of aggregation
 - the parts live and die with the whole
 - symbolized by a black diamond



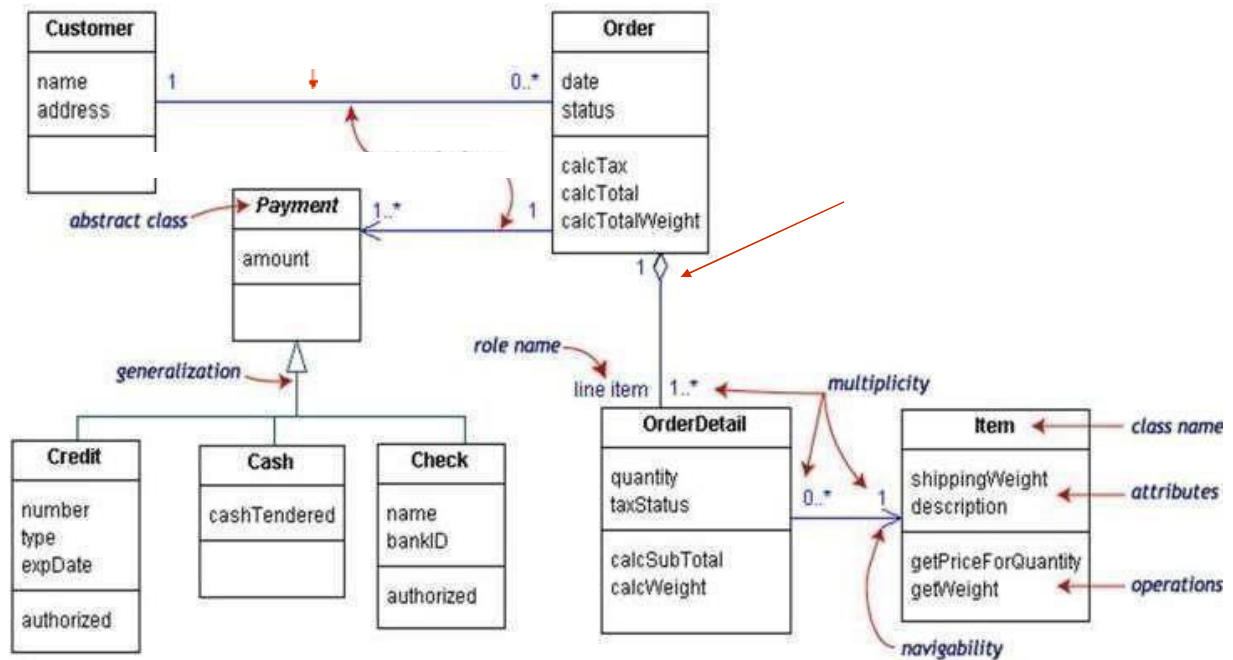
- dependency: “uses temporarily”
 - symbolized by dotted line
 - often is an implementation detail, not an intrinsic part of that object's state

Composition/aggregation example

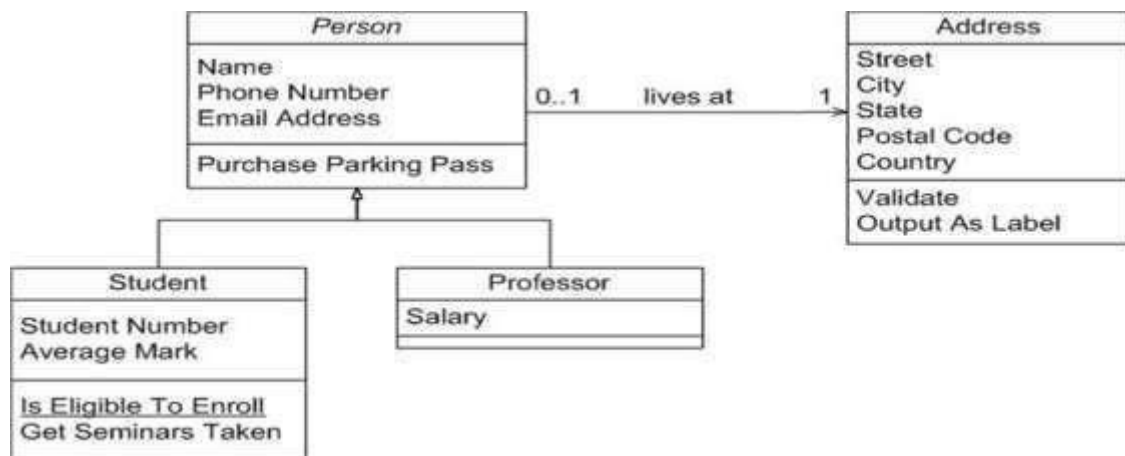


- If the movie theater goes away
 - So does the box office => composition but movies may still exist => aggregation

Class diagram example



Example2: Design Database for the people



RESULT:

EX.NO:9	Object Features of SQL-UDTs and Sub-Types, Tables Using UDTs, Inheritance
DATE:	

AIM:

PROCEDURE:

To create a user-defined type using CREATE DOMAIN and CREATE TYPE statements.

Create Scalar Type - Alias for Built-in Data Type

CREATE DOMAIN allows you to create an alias for a built-in data type and specify the range, optional DEFAULT, NOT NULL and CHECK constraint:

```
CREATE DOMAIN addr VARCHAR(90) NOT NULL DEFAULT 'N/A';
CREATE DOMAIN idx INT CHECK (VALUE > 100 AND VALUE < 999);
```

Create Enumeration Type:

```
CREATE DOMAIN color VARCHAR(10)
CHECK (VALUE IN ('red', 'green', 'blue'));
CREATE TYPE color2 AS ENUM ('red', 'green', 'blue');
```

Table created by using UDT

```
CREATE TABLE colors(
    Color color, -- type created by CREATE DOMAIN color2
    color2 -- type created by CREATE TYPE
);

INSERT INTO colors VALUES ('red', 'red');
INSERT INTO colors VALUES ('blue', 'blue');
```

Output:

```
SELECT * FROM colors ORDER BY color;
--blueblue
--redred
```

Inheritance Definition:

Inheritance is a concept from object-oriented databases.

It opens up interesting new possibilities of database design. Let's create two tables: A table authors and a table distinguished_authors.

Parent Table

Postgres# create table authors(id integer, last_name varchar(10), first_name(10)); Table created.

Postgres# insert into authors(id, last_name, first_name) values(25043, 'Simon', 'Neil'), (20043, 'Nick', 'Jones'); 1 row created.

Child Table:

Postgres# create table distinguished_authors(award text); Table created.

Postgres# insert into distinguished_authors(award varchar(20)) values('Pulitzer Prize'); 1 row created.

Inheritance showing Query:

Postgres# CREATE TABLE distinguished_authors(award text) INHERITS (authors);

RESULT:

EX.NO:10**DATE:****Querying the Object-Relational Database Using Object Query Language****AIM:****Procedure:**

OQL aims at defining a subset of the data in a natural language, while hiding the complexity of the data model and benefit of the power of the object model (encapsulation, inheritance). Its syntax sticks to the syntax of SQL, and its grammar is a subset of SQL. As of now, only SELECT statements have been implemented. Such a statement does return objects of the expected class. The result will be used by programmatic means (to develop an API like ITop).

OQL Statement

There is currently one single type of statement: SELECT

SELECT

class_reference[class_joined][WHERE expression]

Note the absence of FROM clause, because OQL is aimed at returning objects, not values.

Starter**Select Title**

Do return any title existing in the Database. No need to specify the expected columns as we would do in a SQL SELECT clause: OQL do return plain objects.

Joining classes together:

- Relation *course*

<i>course_id</i>	<i>title</i>	<i>dept_name</i>	<i>credits</i>
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

- Relation *prereq*

<i>course_id</i>	<i>prereq_id</i>
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

Outer Join

- Computes the join and then add tuples from one relation that does not match tuples in the other relation to the result of the join. Uses *null* values.

Left Join:

SELECT * FROM course **left join** prereq on course.course_id=prereq.course_id where course.course_id LIKE „BIO-301“

<i>course_id</i>	<i>title</i>	<i>dept_name</i>	<i>credits</i>	<i>prereq_id</i>
BIO-301	Genetics	Biology	4	BIO-101

Right Join:

Select * from course right join prereq on course.course_id=prereq.course_id where prereq.prereq_id LIKE „CS-101“;

Course_id	title	dept_name	credits	prereq_id
CS-190	GameDesign	Comp.Sci	4	CS-101
CS-347	null	null	null	CS-101

Full join:

Select * from course full join prereq on course.course_id=prereq.course_id where course_id LIKE „BIO-101“;

<i>Course_id</i>	<i>title</i>	<i>dept_name</i>	<i>credits</i>	<i>prereq_id</i>
BIO-301	Genetics	Biology	4	BIO-101

RESULT:

CONTENT BEYOND THE SYLLABUS

EX.NO:11	MongoDB
DATE:	

AIM: To understand and demonstrate the MongoDB

MongoDB Overview

MongoDB is a cross-platform, document oriented database that provides, highperformance,highavailability,andeasyscalability.MongoDBworksonconceptofcollecti onanddocument.

Database

Databasaisaphysicalcontainerforcollections.Eachdatabasegetsitsownsetoffiles onthefilesy stem.AsingleMongoDB server typically has multiple databases.

Collection

Collection is a group of MongoDB documents. It is the equivalent of an RDBMS table. A collection exists within a single database. Collections do not enforce a schema. Documents within a collection can have different fields. Typically, all documents in a collection are of similar or related purpose.

Document

A document is a set of key-value pairs. Documents have dynamic schema. Dynamic schema means that documents in the same collection do not need to have the same set of fields or structure, and common fields in a collection's documents may hold different types of data.

The following tables how the relationship of RDBMS terminology with MongoDB.

RDBMS	MongoDB
Database	Database
Table	Collection
Tuple / Row	Document
column	Field
Table Join	Embedded Documents
Primary Key	Key (Default key _id provided by mongodb itself)

Database Server and Client	
Mysqld/Oracle	mongod
mysql/sqlplus	mongo

Sample Document

Following example shows the document structure of a blog site, which is simply a comma separated key value pair.

```
{
  _id:ObjectId(7df78ad8902c)title:'MongoDBOverview',
  description:'MongoDBisnosqldatabase',by:'tutorialspoint',
  url:
  'http://www.tutorialspoint.com',tags:['mongodb','database','NoSQL'],likes:100,
  comments:[
    {
      user:'user1',
      message:'My
      firstcomment',dateCreated:newDate(2011,1,20,2,15),like:0
    },
    {
      user:'user2',
      message:'Mysecondcomments',dateCreated:newDate(2011,1,25,7,45),like:5
    }
  ]
}
```

_id is a 12 bytes hexadecimal number which assures the uniqueness of every document. You can provide **_id** while inserting the document. If you don't provide then MongoDB provides a unique id for every document. These 12 bytes first 4 bytes for the current timestamp, next 3 bytes for machine id, next 2 bytes for process id of MongoDB server and remaining 3 bytes are simple incremental VALUE.

MongoDB—Advantages

Any relational database has a typical schema design that shows number of tables and the relationship between these tables. While in MongoDB, there is no concept of relationship.

Advantages of MongoDB over RDBMS

- ☐ **Schema less:** MongoDB is a document database in which one collection holds different documents. Number of fields, content and size of the document can differ from one document to another.
- ☐ Structure of a single object is clear. No complex joins.
- ☐ Deepquery-ability. MongoDB supports dynamic queries on documents using a document-based query language that's nearly as powerful as SQL.
- ☐ Tuning.
- ☐ Ease of scale-out: MongoDB is easy to scale.
- ☐ Conversion/mapping of application objects to database objects not needed.
- ☐ Uses internal memory for storing the (windowed) working set, enabling faster access of data.

Why Use MongoDB?

- ☐ Document Oriented Storage: Data is stored in the form of JSON style documents. Index on any attribute
- ☐ Replication and high availability Auto-sharding
- ☐ Rich queries
- ☐ Fast in-place updates
- ☐ Professional support by MongoDB

Where to Use MongoDB?

- ☐ Big Data
- ☐ Content Management and Delivery
- ☐ Mobile and Social Infrastructure

MongoDB—Environment

Let us now see how to install MongoDB on Windows.

Install MongoDB on Windows

To install MongoDB on Windows, first download the latest release of MongoDB from <http://www.mongodb.org/downloads>. Make sure you get correct version of MongoDB depending upon your Windows version. To get your Windows version, open command prompt and execute the following command.

```
C:\>wmic os get os architecture OS Architecture 64-bit
C:\>
```

32-bit versions of MongoDB only support databases smaller than 2GB and suitable only for testing and evaluation purposes.

Now extract your downloaded file to c:\ drive or any other location. Make sure the name of the extracted folder is mongodb-win32-i386-[version] or mongodb-win32-x86_64-[version]. Here [version] is the version of MongoDB download.

Next, open the command prompt and run the following command.

```
C:\>move mongodb-win64-*mongodb1 dir(s)moved.
C:\>
```

In case you have extracted the MongoDB at different location, then goto that path by using command **cdFOOLDER/DIR** and now run the above given process.

MongoDB requires a data folder to store its files. The default location for the MongoDB data directory is c:\data\db. So you need to create this folder using the Command Prompt. Execute the following command sequence.

```
C:\>mddataC:\mddata\db
```

If you have to install the MongoDB at a different location, then you need to specify an alternate path for **\data\db** by setting the path **dbpath** in **mongod.exe**. For the same, issue the following commands.

In the command prompt, navigate to the bin directory present in the MongoDB installation folder. Suppose my installation folder is **D:\setup\mongodb**

```
C:\Users\XYZ>d:
D:\>cd "setup"
D:\setup>cd mongodb
D:\setup\mongodb>cd
bin
D:\setup\mongodb\bin>mongod.exe --dbpath "d:\setup\mongodb\data"
```

This will show **waiting for connections** message on the console output, which indicates that the mongod.exe process is running successfully.

Now to run the MongoDB, you need to open another command prompt and issue the following command.

```
D:\setup\mongodb\bin>mongo.exe MongoDB shell version:2.4.6 connecting
to:test
```

```
>db.test.save({a:1})
```

```
>db.test.find()
```

```
{ "_id":ObjectId(5879b0f65a56a454),"a":1 }
```

```
>
```

This will show that MongoDB is installed and run successfully. Next time when you run MongoDB, you need to issue only commands.

```
D:\setup\mongodb\bin>mongod.exe--
```

```
dbpath"d:\setup\mongodb\data"D:\setup\mongodb\bin>mongo.exe
```

Install MongoDB on Ubuntu

Run the following command to import the MongoDB public GPG key–

```
sudoapt-keyadv--keyserverhkp://keyserver.ubuntu.com:80--
```

```
recv7F0CEB10Create/etc/apt/sources.list.d/mongodb.listfileusingthefollowingcomman
d.echo'debhttp://downloads-distro.mongodb.org/repo/ubuntu-upstartdist
```

```
10gen'|sudotee/etc/apt/sources.list.d/mongodb.list
```

Now issue the following command to update the repository–`sudoapt-getupdate`

Next install the MongoDB by using the following command –`apt get install mongodb10gen=2.2.3`

Intheaboveinstallation,2.2.3 is currently released MongoDBversion. Make sure to install the latest version always. NowMongoDB is installed successfully.

StartMongoDB

```
Sudo service mongodb start
```

StopMongoDB

```
Sudo service mongodb stop
```

RestartMongoDB

```
Sudo service mongodb restart
```

To use MongoDB run the following command.`mongo`

This will connect to running MongoDB instance.

MongoDB Help

To get a list of commands, type `db.help()` in MongoDB client. This will give you a list of commands as shown in the following screenshot.

MongoDB Statistics

To get stats about MongoDB server, type the command `db.stats()` in MongoDB client. This will show the database name, number of collection and documents in the database. Output of the command is shown in the following screenshot.

MongoDB-Create Database

Here, we will see how to create database in MongoDB.

The use Command

MongoDB `use DATABASE_NAME` is used to create database. The command will create new database if it doesn't exist, otherwise it will return the existing database.

Syntax

Basic syntax of `use DATABASE` statement is as follows:

Use `DATABASE_NAME`

Example

If you want to create database with name `<mydb>`, then `use DATABASE` statement would be as follows:

```
>usemydb switched to dbmydb
```

To check your currently selected database, use the command `db`

```
>dbmydb
```

If you want to check your databases list, use the command `show dbs`.

```
>showdbs
```

```
local    0.78125GB
```

```
test     0.23012GB
```

Your created database (mydb) is not present in list. To display database, you need to insert at least one document into it.

```
>db.movie.insert({"name":"tutorialspoint"})
```

```
>show dbslocalmydb
```

```
0.78125GB
```

```
0.23012GBtest    0.23012GB
```

In MongoDB default database is test. If you didn't create any database, then collections will be stored in test database.

MongoDB—Drop Database

Here, we will see how to drop a database using MongoDB command.

The drop Database () Method

MongoDB `db.dropDatabase()` command is used to drop a existing database.

Syntax

Basic syntax of **dropDatabase()** command is as follows:`db.dropDatabase()`

This will delete the selected database. If you have not selected any database, then it will delete default 'test' database.

Example

First, check the list of a available databases by using the command, **showdbs**.

```
>show dbs Local mydb test
```

```
>
```

```
0.78125GB
```

```
0.23012GB
```

```
0.23012GB
```

If you want to delete new database<**mydb**>,then**dropDatabase()** command would be as follows:

```
>use mydb switched to db mydb
```

```
>db.dropDatabase()
```

```
>{"dropped" : "mydb", "ok": 1 }
```

```
>
```

Now check list of databases.

```
>show dbs local test0.
```

```
78125GB
```

```
0.23012GB
```

MongoDB—Create Collection

Here, we will see how to create a collection using MongoDB.

The create Collection() Method

MongoDB **db.createCollection(name,options)** is used to create collection.

Syntax

Basic syntax of **createCollection()** command is as follows:`db.create Collection (name,options)`

In the command, **name** is name of collection to be created. **Options** is a document and is used to specify configuration of collection.

Parameter	Type	Description
Name	String	Name of the collection to be created
Options	Document	(Optional) Specify options about memory size and indexing

Options parameter is optional, so you need to specify only the name of the collection. Following is the list of options you can use:

Field	Type	Description
capped	Boolean	(Optional) If true, enables a capped collection. Capped collection is a fixed size collection that automatically overwrites its oldest entries when it reaches its maximum size. If you specify true, you need to specify size parameter also.
autoIndexID	Boolean	(Optional) If true, automatically create index on _id field. Default value is false.
size	number	(Optional) Specifies a maximum size in bytes for a capped collection. If capped is true, then you need to specify this field also.
max	number	(Optional) Specifies the maximum number of documents allowed in the capped collection.

While inserting the document, MongoDB first checks size field of capped collection, then it checks max field.

Examples

Basic syntax of **createCollection()** method without options is as follows:

```
>use test
switched to db test
>db.createCollection("mycollection")
{"ok":1}
>
```

You can check the created collection by using the command **show collections**.

```
>show collections mycollection system.indexes
```


The following example shows the syntax of **createCollection()** method with few important options:

```
>db.createCollection("mycol",{ capped:true,autoIndexID:true,size:6142800,max:10000}  
)  
{"ok":1}  
>
```

In MongoDB, you don't need to create collection. MongoDB creates collection automatically, when you insert some document.

```
>db.tutorialspoint.insert({"name":"tutorialspoint"})  
  
>showcollection  
nsmycolmycoll  
ectionssystem.in  
dexestutorialsp  
oint  
>
```

RESULT: Thus MongoDB was understood was demonstrated

EX.NO:12	SQL Queries on Clustered and Non-Clustered Indexes
DATE:	

AIM: To demonstrate SQL Queries on Clustered and Non-Clustered Indexes

Clustered Indexes in PostgreSQL:

A clustered index is an index structure that determines the physical order of the data on the disk. In PostgreSQL, the clustered index is implemented using the concept of a clustered table or an index-organized table. When a clustered index is created on a table, the data is physically ordered according to the values in the indexed column(s). This can improve query performance, as it allows the database system to retrieve the required data more efficiently.

Example1:

```
CREATE TABLE sales (
  id SERIAL PRIMARY KEY,
  transaction_date DATE,
  customer_name VARCHAR(50),
  product_name VARCHAR(50),
  amount NUMERIC
);
```

```
CREATE INDEX sales_transaction_date_idx ON sales (transaction_date);
```

OUTPUT:

```
CREATE INDEX
```

```
Query returned successfully
```

Example2:

```
Create table Student
( Roll_No int primary key,
  Name varchar(50),
  Gender varchar(30),
  Mob_No bigint );
insert into Student
values (4, 'ankita', 'female', 9876543210 );

insert into Student
values (3, 'anita', 'female', 9675432890 );
```

```
insert into Student
values (5, 'mahima', 'female', 8976453201 );
```

OUTPUT:

Query returned successfully

Non-Clustered Indexes in PostgreSQL:

A non-clustered index is an index structure that stores the indexed column values along with a pointer to the corresponding table rows. In PostgreSQL, a non-clustered index is implemented using a B-tree or GiST index. When a query is executed, the database system uses the non-clustered index to look up the relevant rows in the table, based on the indexed column values.

Example1:

```
CREATE INDEX sales_customer_name_idx ON sales
(customer_name);
```

OUTPUT:

Query returned successfully

Example:

```
Create table Student
( Roll_No int primary key,
Name varchar(50),
Gender varchar(30),
Mob_No bigint );
```

```
insert into Student
values (4, 'afzal', 'male', 9876543210 );
insert into Student
values (3, 'sudhir', 'male', 9675432890 );
insert into Student
values (5, 'zoya', 'female', 8976453201 );
create nonclustered index NIX_FTE_Name
on Student (Name ASC);
```

OUTPUT:

```
CREATE INDEX
Query returned successfully
```

RESULT: Thus SQL Queries on Clustered and Non-Clustered Indexes were demonstrated

EX.NO:13	Auditing with Trigger
DATE:	

AIM: To demonstrate Auditing with Trigger

Example:

```
CREATE TABLE emp (
Empname text NOT NULL,
salaryinteger
);
```

```
CREATE TABLE emp_audit(
operation          char(1)      NOT NULL,
stamp              timestamp NOT NULL,
userid             text         NOT NULL,
empname            text         NOT NULL,
salary             integer
);
```

```
CREATE OR REPLACE FUNCTION process_emp_audit() RETURNS TRIGGER AS
$emp_audit$
BEGIN
    --
    -- Create rows in emp_audit to reflect the operations performed on emp,
    -- making use of the special variable TG_OP to work out the operation.
    --
    IF (TG_OP = 'DELETE') THEN
        INSERT INTO emp_audit
            SELECT 'D', now(), current_user, o.* FROM old_table o;
    ELSIF (TG_OP = 'UPDATE') THEN
        INSERT INTO emp_audit
            SELECT 'U', now(), current_user, n.* FROM new_table n;
    ELSIF (TG_OP = 'INSERT') THEN
        INSERT INTO emp_audit
            SELECT 'I', now(), current_user, n.* FROM new_table n;
    END IF;
    RETURN NULL; -- result is ignored since this is an AFTER trigger
END;
$emp_audit$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER emp_audit_ins  
  AFTER INSERT ON emp  
  REFERENCING NEW TABLE AS new_table  
  FOR EACH STATEMENT EXECUTE FUNCTION process_emp_audit();
```

```
CREATE TRIGGER emp_audit_upd  
  AFTER UPDATE ON emp  
  REFERENCING OLD TABLE AS old_table NEW TABLE AS new_table  
  FOR EACH STATEMENT EXECUTE FUNCTION process_emp_audit();
```

```
CREATE TRIGGER emp_audit_del  
  AFTER DELETE ON emp  
  REFERENCING OLD TABLE AS old_table  
  FOR EACH STATEMENT EXECUTE FUNCTION process_emp_audit();
```

OUTPUT:

Query returned successfully

RESULT: Thus Auditing with Trigger was demonstrated

EX.NO:14	Cursor
DATE:	

AIM: To demonstrate Cursor implementation

Example:

A Cursor in PostgreSQL is used to process large tables. Suppose if a table has 10 million or billion rows. While performing a SELECT operation on the table it will take some time to process the result and most likely give an “out of memory” error and the program will be terminated.

A Cursor can only be declared inside a transaction. The cursor does not calculate the data but only prepares the query so that your data can be created when FETCH is called. In the end, simply commit the transaction.

Syntax:

```
DECLARE
[cursor_name] CURSOR FOR [query]
```

Let's analyze the above syntax:

- Use DECLARE to declare a cursor
- **[cursor_name]** - Give any name to the cursor
- **[query]** - Give a query to the cursor

After declaring a cursor, we can get the data using FETCH. The FETCH gets the next row(s) from the cursor. If no row found, then it returns NULL.

Syntax:

```
FETCH [direction (rows)] FROM [cursor_name];
```

where direction can be empty,
number of rows you want or one of the following:

```
NEXT
PRIOR
FIRST
LAST
ABSOLUTE count
RELATIVE count
count
ALL
FORWARD
```

```
FORWARD count
FORWARD ALL
BACKWARD
BACKWARD count
BACKWARD ALL
```

Lets, create a sample table using the below commands for examples:

```
CREATE TABLE students (
student_id serial PRIMARY KEY,
full_name VARCHAR NOT NULL,
branch_id INT
);
```

Insert data into students table as follows:

```
INSERT INTO students (
student_id,
full_name,
branch_id
)
VALUES
(1, 'M.S Dhoni', NULL),
(2, 'Sachin Tendulkar', 1),
(3, 'R. Sharma', 1),
(4, 'S. Raina', 1),
(5, 'B. Kumar', 1),
(6, 'Y. Singh', 2),
(7, 'Virender Sehwag ', 2),
(8, 'Ajinkya Rahane', 2),
(9, 'Shikhar Dhawan', 2),
(10, 'Mohammed Shami', 3),
(11, 'Shreyas Iyer', 3),
(12, 'Mayank Agarwal', 3),
(13, 'K. L. Rahul', 3),
(14, 'Hardik Pandya', 4),
(15, 'Dinesh Karthik', 4),
(16, 'JaspritBumrah', 7),
(17, 'Kuldeep Yadav', 7),
(18, 'Yuzvendra Chahal', 8),
(19, 'Rishabh Pant', 8),
(20, 'Sanju Samson', 8);
```

Now that the table is ready we can declare our cursor.

```

BEGIN;
DECLARE
my_cursor CURSOR FOR SELECT * FROM students;
Fetch the data.
FETCH 10 FROM my_cursor;

```

OUTPUT:

```

postgres=# FETCH 10 FROM my_cursor;
 student_id |      full_name      | branch_id
-----+-----+-----
          1 | M.S Dhoni           |          1
          2 | Sachin Tendulkar    |          1
          3 | R. Sharma           |          1
          4 | S. Raina            |          1
          5 | B. Kumar            |          1
          6 | Y. Singh            |          2
          7 | Virender Sehwag     |          2
          8 | Ajinkya Rahane      |          2
          9 | Shikhar Dhawan      |          2
         10 | Mohammed Shami      |          3
(10 rows)

```

```

FETCH PRIOR FROM my_cursor;
FETCH PRIOR FROM my_cursor;

```

The above query will give you row 9 and 8 since right now our cursor is at 10;
 FETCH 6 FROM my_cursor;

OUTPUT:

```

postgres=# FETCH 6 FROM my_cursor;
 student_id |      full_name      | branch_id
-----+-----+-----
          9 | Shikhar Dhawan      |          2
         10 | Mohammed Shami      |          3
         11 | Shreyas Iyer         |          3
         12 | Mayank Agarwal       |          3
         13 | K. L. Rahul          |          3
         14 | Hardik Pandya        |          4
(6 rows)

```

```

COMMIT;
Commit the transaction at the end.

```

RESULT: Thus Cursor implementation was demonstrated