Dr. D. Y. Patil Pratishthan's

DR. D. Y. PATIL INSTITUTE OF ENGINEERING, MANAGEMENT & RESEARCH

Approved by A.I.C.T.E, New Delhi , Maharashtra State Government, Affiliated to Savitribai Phule Pune University

Sector No. 29, PCNTDA , Nigidi Pradhikaran, Akurdi, Pune 411044. Phone: 020–27654470, Fax: 020-27656566

Website : www.dypiemr.ac.in Email : principal.dypiemr@gmail.com

---

DEPARTMENT OF COMPUTER ENGINEERING

LAB MANUAL

Database Management System Laboratory

(Third Year Engineering)

Semester – I

Prepared by : Mrs. Rashmi Deshpande
Mrs. Akanksha Kulkarni
Mrs.Priyanka Patil

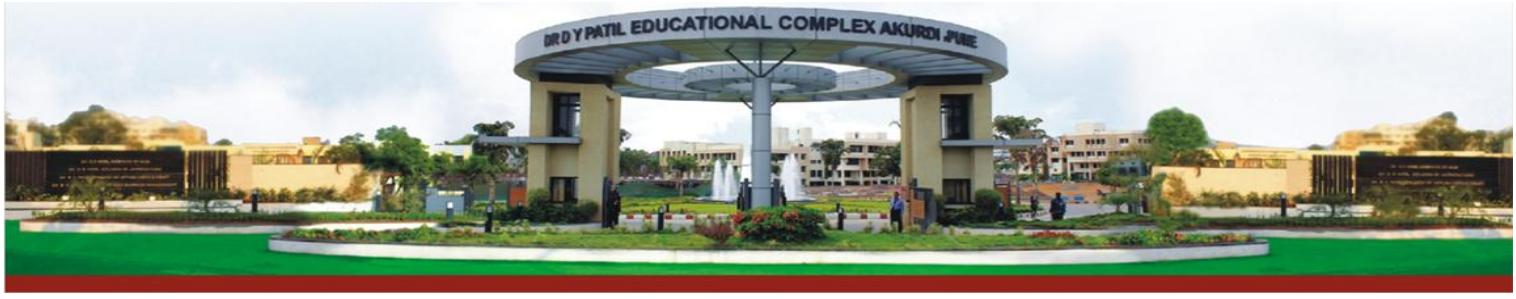


Table of Contents

Assignment No - A1

- **Title:** Study and Draw ER Modelling diagram along with normalization using ERDwin/ERD plus for selected problem statement.
- **Objective:** Study and understand modern tools for ER diagrams.
- **Outcome:** Ability to use modern tools like ERD Plus and ER Win for representing logical model of DBMS.
- **Problem Statement:** Draw ER Modelling digram along with normalization using ERDwin/ERD plus for university database.

The Entity-Relationship Model:

- The entity-relationship (E-R) data model was developed to facilitate database design by allowing specification of an enterprise schema that represents the overall logical structure of a database.
- The E-R model is very useful in mapping the meanings and interactions of real-world enterprises onto a conceptual schema.
- Because of this usefulness, many database-design tools draw on concepts from the E-R model.
- The E-R data model employs three basic concepts: entity sets, relationship sets, and attributes, which we study first.

Entity Sets:

- An entity set is a set of entities of the same type that share the same properties, or attributes.
- The set of all people who are instructors at a given university, for example, can be defined as the entity set instructor.
- Similarly, the entity set student might represent the set of all students in the university.

76766	Crick
45565	Katz
10101	Srinivasan
98345	Kim
76543	Singh
22222	Einstein

instructor

98988	Tanaka
12345	Shankar
00128	Zhang
76543	Brown
76653	Aoi
23121	Chavez
44553	Peltier

student

Figure 7.1 Entity sets *instructor* and *student*.

Attributes:

- An entity is represented by a set of **attributes**.
- Attributes are descriptive properties possessed by each member of an entity set.

- The designation of an attribute for an entity set expresses that the database stores similar information concerning each entity in the entity set; however, each entity may have its own value for each attribute.
- Possible attributes of the *instructor* entity set are *ID*, *name*, *dept name*, and *salary*.

Simple and composite attributes:

- **Composite** attributes, can be divided into subparts (that is, other attributes).
- For example, an attribute *name* could be structured as a composite attribute consisting of *first name*, *middle initial*, and *last name*.

Single-valued and multivalued:

- The attributes in our examples all have a single value for a particular entity.
- For instance, the student ID attribute for a specific student entity refers to only one student ID.
- Such attributes are said to be single valued.
- There may be instances where an attribute has a set of values for a specific entity.

Derived attribute:

- The value for this type of attribute can be derived from the values of other related attributes or entities.
- For instance, let us say that the *instructor* entity set has an attribute *students advised*, which represents how many students an instructor advises.
- We can derive the value for this attribute by counting the number of *student* entities associated with that instructor.
- As another example, suppose that the *instructor* entity set has an attribute *age* that indicates the instructor's age.
- If the *instructor* entity set also has an attribute *date of birth*, we can calculate *age* from *date of birth* and the current date.
- Thus, *age* is a derived attribute. In this case, *date of birth* may be referred to as a *base* attribute, or a *stored* attribute.
- The value of a derived attribute is not stored but is computed when required.

Relationship Set:

- A **relationship** is an association among several entities. For example, we can define a relationship *advisor* that associates instructor Katz with student Shankar.
- This relationship specifies that Katz is an advisor to student Shankar.
- A **relationship set** is a set of relationships of the same type.
- A relationship may also have attributes called **descriptive attributes**.
- Consider a relationship set *advisor* with entity sets *instructor* and *student*.
- We could associate the attribute *date* with that relationship to specify the date when an instructor became the advisor of a student.
- The *advisor* relationship among the entities corresponding to instructor Katz and student Shankar has the value "10 June 2007" for attribute *date*, which means that Katz became Shankar's advisor on 10 June 2007.

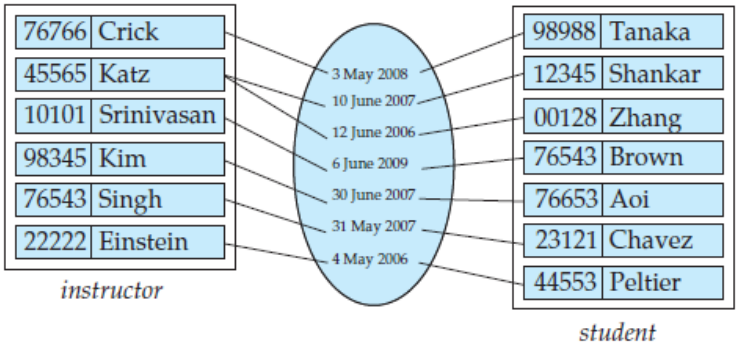


Figure 7.3 *date* as attribute of the *advisor* relationship set.

Mapping Cardinalities:

- Mapping cardinalities, or cardinality ratios, express the number of entities to which another entity can be associated via a relationship set.
- Mapping cardinalities are most useful in describing binary relationship sets, although they can contribute to the description of relationship sets that involve more than two entity sets.
- In this section, we shall concentrate on only binary relationship sets.
- For a binary relationship set *R* between entity sets *A* and *B*, the mapping cardinality must be one of the following:
- The participation of an entity set *E* in a relationship set *R* is said to be **total** if every entity in *E* participates in at least one relationship in *R*.
- If only some entities in *E* participate in relationships in *R*, the participation of entity set *E* in relationship *R* is said to be **partial**.

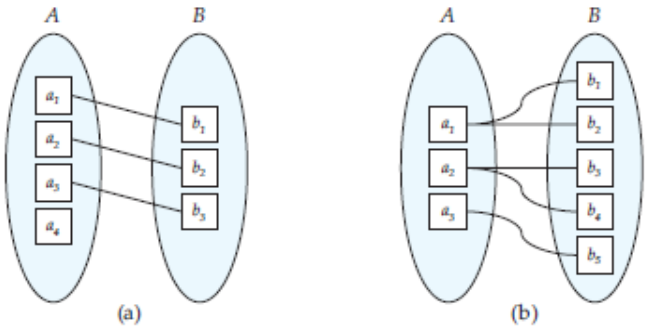


Figure 7.5 Mapping cardinalities. (a) One-to-one. (b) One-to-many.

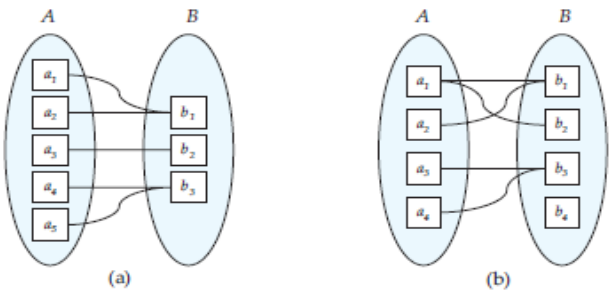


Figure 7.6 Mapping cardinalities. (a) Many-to-one. (b) Many-to-many.

Keys:

- Keys play an important role in the relational database.
- It is used to uniquely identify any record or row of data from the table. It is also used to establish and identify relationships between tables.
- For example, ID is used as a key in the Student table because it is unique for each student. In the PERSON table, passport_number, license_number, SSN are keys since they are unique for each person.

1. Primary key

- It is the first key used to identify one and only one instance of an entity uniquely. An entity can contain multiple keys, as we saw in the PERSON table. The key which is most suitable from those lists becomes a primary key.
- In the EMPLOYEE table, ID can be the primary key since it is unique for each employee. In the EMPLOYEE table, we can even select License_Number and Passport_Number as primary keys since they are also unique.
- For each entity, the primary key selection is based on requirements and developers.

2. Candidate key

- A candidate key is an attribute or set of attributes that can uniquely identify a tuple.
- Except for the primary key, the remaining attributes are considered a candidate key. The candidate keys are as strong as the primary key.
- For example: In the EMPLOYEE table, id is best suited for the primary key. The rest of the attributes, like SSN, Passport_Number, License_Number, etc., are considered a candidate key.

3. Super Key

- Super key is an attribute set that can uniquely identify a tuple. A super key is a superset of a candidate key.
- **For example:** In the above EMPLOYEE table, for(EMPLOEE_ID, EMPLOYEE_NAME), the name of two employees can be the same, but their EMPLYEE_ID can't be the same. Hence, this combination can also be a key.
- The super key would be EMPLOYEE-ID (EMPLOYEE_ID, EMPLOYEE-NAME), etc.

4. Foreign key

- Foreign keys are the column of the table used to point to the primary key of another table.
- Every employee works in a specific department in a company, and employee and department are two different entities. So we can't store the department's information in the employee table. That's why we link these two tables through the primary key of one table.
- We add the primary key of the DEPARTMENT table, Department_Id, as a new attribute in the EMPLOYEE table.
- In the EMPLOYEE table, Department_Id is the foreign key, and both the tables are related.

5. Alternate key

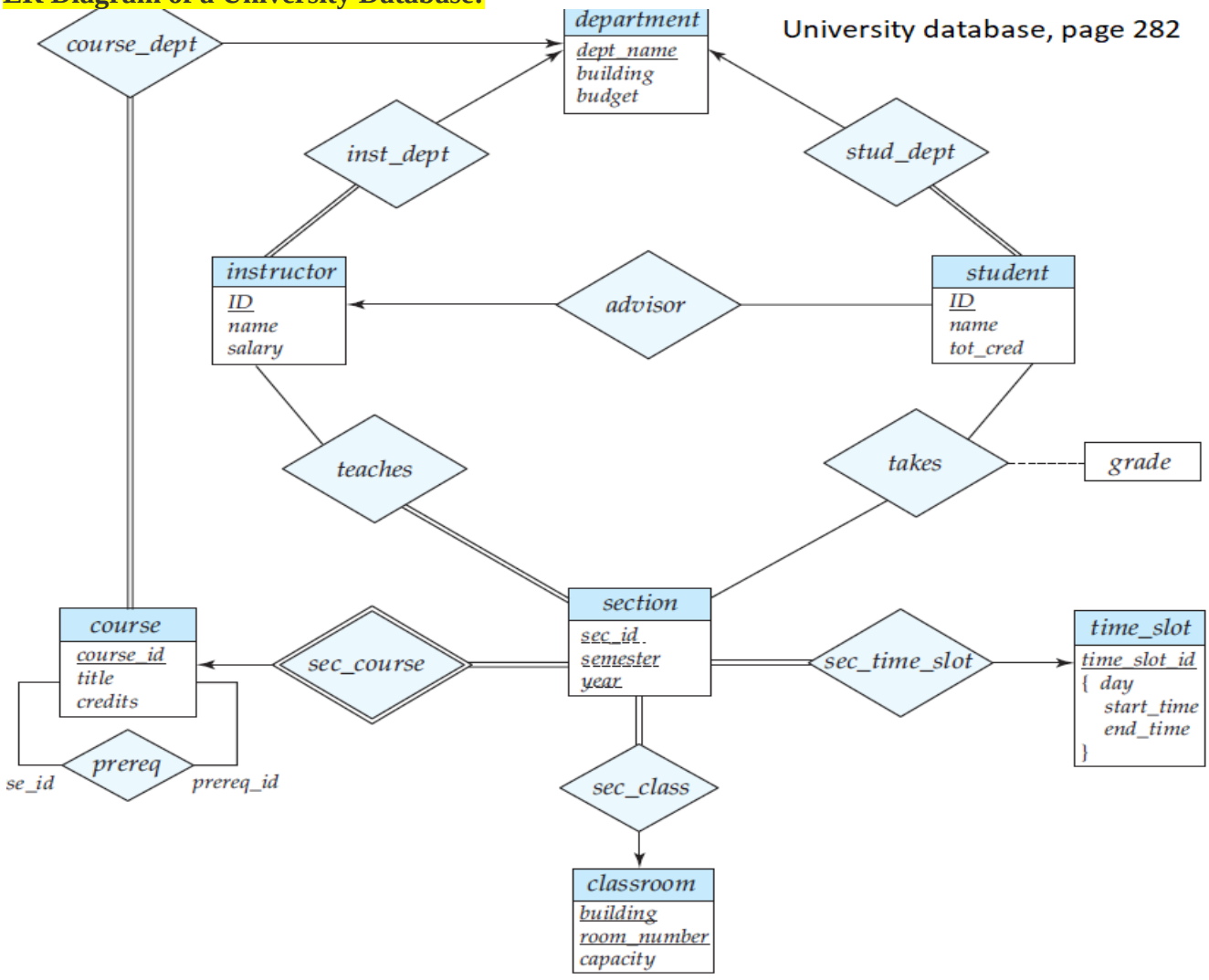
- There may be one or more attributes or a combination of attributes that uniquely identify each tuple in a relation. These attributes or combinations of the attributes are called the candidate keys. One key is chosen as the primary key from these candidate keys, and the remaining candidate key, if it exists, is termed the alternate key. In other words, the total number of the alternate keys is the total number of candidate keys minus the primary key. The alternate key may or may not exist. If there is only one candidate key in a relation, it does not have an alternate key.

- For example, employee relation has two attributes, Employee_Id and PAN_No, that act as candidate keys. In this relation, Employee_Id is chosen as the primary key, so the other candidate key, PAN_No, acts as the Alternate key.

6. Composite key

- Whenever a primary key consists of more than one attribute, it is known as a composite key. This key is also known as Concatenated Key.
- **For example,** in employee relations, we assume that an employee may be assigned multiple roles, and an employee may work on multiple projects simultaneously. So the primary key will be composed of all three attributes, namely Emp_ID, Emp_role, and Proj_ID in combination. So these attributes act as a composite key since the primary key comprises more than one attribute.

ER Diagram of a University Database:



Conclusion:

We have successfully studied and understand the basic theoretical concepts of ER diagrams.

Assignment No - A2 (a)

- **Title:** Design and Develop SQL DDL statements which demonstrate the use of SQL objects such as Table, View, Index, Sequence, Synonym
- **Objective:** Use DDL statements with all constraints, synonym, sequence to design the database and use Insert, Select, Update, Delete, operators, functions, set operators, to solve queries
- **Outcome:** Ability to develop SQL DDL statements which demonstrate the use of SQL objects such as Table, View, Index, Sequence, Synonym
- **Problem Statement:** Write a SQL DDL statements which demonstrate the use of SQL objects such as Table, View, Index, Sequence, Synonym

- **Hardware and Software requirement:**
  1. 64-bit Dell Machine
  2. Linux OS
  3. MySQL

- **Theory:**

To begin with, the table creation command requires the following details

- Name of the table
- Name of the fields
- Definitions for each field

Syntax

Here is a generic SQL syntax to create a MySQL table –

```
CREATE TABLE table_name (column_name column_type);
```

Now, we will create the following table in the **TUTORIALS** database.

```
tutorials_tbl (
  tutorial_id INT NOT NULL AUTO_INCREMENT,
  tutorial_title VARCHAR(100) NOT NULL,
  tutorial_author VARCHAR(40) NOT NULL,
```

```
  submission_date DATE,

  PRIMARY KEY ( tutorial_id )

);
```

Here, a few items need explanation −

- Field Attribute **NOT NULL** is being used because we do not want this field to be NULL. So, if a user will try to create a record with a NULL value, then MySQL will raise an error.
- Field Attribute **AUTO_INCREMENT** tells MySQL to go ahead and add the next available number to the id field.
- Keyword **PRIMARY KEY** is used to define a column as a primary key. You can use multiple columns separated by a comma to define a primary key.

- **Drop MySQL Tables**

It is very easy to drop an existing MySQL table, but you need to be very careful while deleting any existing table because the data lost will not be recovered after deleting a table.

Syntax

Here is a generic SQL syntax to drop a MySQL table −

```
DROP TABLE table_name ;
```

- **Dropping Tables from the Command Prompt**

To drop tables from the command prompt, we need to execute the DROP TABLE SQL command at the mysql> prompt.

Example

The following program is an example which deletes the **tutorials_tbl** −

```
root@host# mysql -u root -p

Enter password:*******

mysql> use TUTORIALS;

Database changed

mysql> DROP TABLE tutorials_tbl

Query OK, 0 rows affected (0.8 sec)
```

```
mysql>
```

MySQL - INDEXES

- A database index is a data structure that improves the speed of operations in a table. Indexes can be created using one or more columns, providing the basis for both rapid random lookups and efficient ordering of access to records.
- While creating index, it should be taken into consideration which all columns will be used to make SQL queries and create one or more indexes on those columns.
- Practically, indexes are also a type of tables, which keep primary key or index field and a pointer to each record into the actual table.
- The users cannot see the indexes, they are just used to speed up queries and will be used by the Database Search Engine to locate records very fast.
- The INSERT and UPDATE statements take more time on tables having indexes, whereas the SELECT statements become fast on those tables. The reason is that while doing insert or update, a database needs to insert or update the index values as well.

Simple and Unique Index

You can create a unique index on a table. A unique index means that two rows cannot have the same index value. Here is the syntax to create an Index on a table.

```
CREATE UNIQUE INDEX index_name
ON table_name ( column1, column2,...);
```

You can use one or more columns to create an index.

For example, we can create an index on **tutorials_tbl** using **tutorial_author**.

```
CREATE UNIQUE INDEX AUTHOR_INDEX
ON tutorials_tbl (tutorial_author)
```

You can create a simple index on a table. Just omit the **UNIQUE** keyword from the query to create a simple index. A Simple index allows duplicate values in a table.

If you want to index the values in a column in a descending order, you can add the reserved word DESC after the column name.

```
mysql> CREATE UNIQUE INDEX AUTHOR_INDEX
ON tutorials_tbl (tutorial_author DESC)
```

ALTER command to add and drop INDEX

There are four types of statements for adding indexes to a table −

- **ALTER TABLE tbl_name ADD PRIMARY KEY (column_list)** − This statement adds a **PRIMARY KEY**, which means that the indexed values must be unique and cannot be NULL.

- **ALTER TABLE tbl_name ADD UNIQUE index_name (column_list)** – This statement creates an index for which the values must be unique (except for the NULL values, which may appear multiple times).
- **ALTER TABLE tbl_name ADD INDEX index_name (column_list)**– This adds an ordinary index in which any value may appear more than once.
- **ALTER TABLE tbl_name ADD FULLTEXT index_name (column_list)** – This creates a special FULLTEXT index that is used for text-searching purposes.

The following code block is an example to add index in an existing table.

```
mysql> ALTER TABLE testalter_tbl ADD INDEX (c);
```

You can drop any INDEX by using the **DROP** clause along with the ALTER command.

Try out the following example to drop the above-created index.

```
mysql> ALTER TABLE testalter_tbl DROP INDEX (c);
```

You can drop any INDEX by using the DROP clause along with the ALTER command.

ALTER Command to add and drop the PRIMARY KEY

You can add a primary key as well in the same way. But make sure the Primary Key works on columns, which are NOT NULL.

The following code block is an example to add the primary key in an existing table. This will make a column NOT NULL first and then add it as a primary key.

```
mysql> ALTER TABLE testalter_tbl MODIFY i INT NOT NULL;

mysql> ALTER TABLE testalter_tbl ADD PRIMARY KEY (i);
```

You can use the ALTER command to drop a primary key as follows –

```
mysql> ALTER TABLE testalter_tbl DROP PRIMARY KEY;
```

To drop an index that is not a PRIMARY KEY, you must specify the index name.

Displaying INDEX Information

You can use the **SHOW INDEX** command to list out all the indexes associated with a table. The vertical-format output (specified by \G) often is useful with this statement, to avoid a long line wraparound –

Try out the following example –

```
mysql> SHOW INDEX FROM table_name\G

........
```

- **Conclusion :**
  We have successfully implemented Control Structures & Exception Handling in Fine Calculation on Book Issue.

A. Sample Code and Output

1. An employee management system needs to record following data about employees – ID, Name, Age, Department, Salary, Experience, AreaOfExperties. Identify columns, their data types and write create statement. Define primary key.
2. Create a view that will display all details of the employee except Salary and AreaOfExperties.
3. Create a sequence to generate employee id.
4. Create an index for the column ID.
5. Create a synonym for the generated table as “EMP” and demonstrate its use.

```
INSERT INTO EMPLOYEE  VALUES(SEQ.NEXTVAL,'AAA','10','COMP','10000','3','TRAINER');
 INSERT INTO EMPLOYEE  VALUES(SEQ.NEXTVAL,'BBB','11','CIVIL','20000','4','MANAGER')
;
 INSERT INTO EMPLOYEE  VALUES(SEQ.NEXTVAL,'CCC','11','IT','30000','5','TEAMLEAD');

----CREATE A VIEW THAT WILL DISPLAY ALL DETAILS OF EMPLOYEE EXCEPT SALARY AND
AREAOFEXPERTISE----
CREATE VIEW EMPLOYEE1 AS
SELECT EMPLOYEE_ID,NAME,AGE,DEPARTMENT,EXPERIENCE FROM EMPLOYEE;

----CREATE A SEQUENCE TO GENERATE EMPLOYEE ID----
CREATE SEQUENCE SEQ
START WITH 1
INCREMENT BY 1;

-----CRAETE AN INDEX FOR COLUMN_ID-----
CREATE INDEX INDEX_ID ON EMPLOYEE(EMPLOYEE_ID);

-----CREATE SYNONYM FOR GENERATED TABLE AS 'EMP' AND DEMONSTRATE ITS USE------
CREATE SYNONYM EMP FOR EMPLOYEE;
SELECT * FROM EMP;
```

Assignment No – A2 (b)

Problem statement: Design at least 10 SQL queries for suitable database application using SQL DML statements: Insert, Select, Update, Delete with operators, functions, and set operator.

Objective: To implement Insert, Select, Update, Delete with operators, functions, and set operator

Outcome: Ability to implement SQL DML Queries.

Problem Statement: Write a SQL query involving Insert, Select, Update, Delete with operators, functions, and set operator.

Solution:

A. Related theory

AGGREGATION:

MySQL aggregate functions retrieve a single value after performing a calculation on a set of values. In general, aggregate functions ignore null values.

COUNT:

The COUNT(column name) function returns the number of values (NULL values will not be counted) of the specified column:SQL COUNT(column name)

Syntax

SELECT COUNT(column name) FROM table name;

ORDER BY CLAUSE:

The ORDER BY keyword is used to sort the result-set by one or more columns. The ORDER BY keyword sorts the records in ascending order by default. To sort the records in a descending order, you can use the DESC keyword.

SQL ORDER BY Syntax:

SELECT * FROM table name

ORDER BY column name ASC or DESC;

DISTINCT KEYWORD:

In a table, a column may contain many duplicate values; and sometimes you only want to list the dfferent (distinct) values. The DISTINCT keyword can be used to return only distinct (different) values.

SQL SELECT DISTINCT Syntax:

SELECT DISTINCT column name,column name FROM table name;

UNION:

The UNION operator is used to combine the result-set of two or more SELECT statements. Notice that each SELECT statement within the UNION must have the same number of columns. The columns must also have similar data types. Also, the columns in each SELECT statement must be in the same order.
SQL UNION Syntax
SELECT column name(s) FROM table1
UNION
SELECT column name(s) FROM table2;

INTERSECTION:
The SQL INTERSECT clause/operator is used to combine two SELECT statements, but returns rows only from the _rst SELECT statement that are identical to a row in the second SELECT statement. This means INTERSECT returns only common rows returned by the two SELECT statements.
SQL INTERSECTION Syntax
SELECT column name(s) FROM table1
INTERSECTION
SELECT column name(s) FROM table2;

IN:
The IN operator allows you to specify multiple values in a WHERE clause.
SQL IN Syntax
SELECT column name(s)
FROM table name
WHERE column name IN (value1,value2,...);

NOTIN:
MySQL NOT IN() makes sure that the expression proceeded does not have any of the values present in the arguments.

SQL NOTIN Syntax:
SELECT column name(s)
FROM table name
WHERE column name NOT IN (value1,value2,...);

Functions performed in banking application are:

1. CREATE BRANCH:

To create a new branch in branch table we use the command:
INSERT into branch values('branch name','branch city',branch assets);
eg: If we want to create a branch named London, having branch city Lawrence and assets of 6000000, we use the command:
INSERT into branch values('London','Lawrence',6000000);

1. CREATE CUSTOMER:

To create a new customer in customer table we use the command:
INSERT into customer values('cust name',cust street','cust city');
eg: If we want to create a branch named London, having branch city Lawrence and assets of 6000000, we use the command:
INSERT into customer values('John',Main street','Glassgow');

2. SHOW:

To display the details of branch table we use the command:
SELECT * FROM branch;
To display the details of customer table we use the command
SELECT * FROM customer;

3. DEPOSIT:

To deposit amout in a particular account we use the command:
UPDATE account set balance = balance + amount WHERE acnt no = account no;
eg: If we want to deposit amount of 25000 in account no 25 we write
the command as:
UPDATE account set balance = balance + 25000
WHERE acnt no = 25;

4. DELETE:

To delete the records of certain customer from customer table we write the command as:
DELETE FROM customer
WHERE cust name='customer name'
eg: If we want to delete the records for the customer named John, we write the command as:
DELETE FROM customer WHERE cust name='John'

B. Conclusion

10 SQL queries for suitable database application using SQL DML statements: Insert, Select, Update, Delete with operators, functions, and set operator.

C. Sample Code and Output

For the following relation schema:
Account(Acc_no, branch_name,balance)
branch(branch_id, branch_name,branch_city,assets)
customer(cust_id, cust_name, cust_street,cust_city)
Depositor(cust_id, acc_no)
Loan(loan_no, branch_id, amount)
Borrower(cust_id, loan_no)

Create above tables and insert few rows in each table. Solve following query:

1. Find the branches where average account balance > 12000.
2. Find all customers who have an account or loan or both at bank.
3. Find all customers who have both account but not loan at bank.
4. Delete all tuples at every branch located in ‘Nigdi’.
5. Find Maximum loan amount in branch ‘Nigdi’
6. Find no. of depositors at each branch.

For all accounts in Akurdi branch increase the balance by 10%.

```
CREATE TABLE ACCOUNT
(
  ACC_NO INTEGER,
  BRANCH_NAME VARCHAR(30),
  BALANCE INTEGER
);
INSERT INTO ACCOUNT VALUES('10','AKURDI','1000');
INSERT INTO ACCOUNT VALUES('11','RAVET','2000');
INSERT INTO ACCOUNT VALUES('12','CHINCHWAD','3000');

CREATE TABLE BRANCH
(
   BRANCH_ID INTEGER,
  BRANCH_NAME VARCHAR(30),
  BRANCH_CITY VARCHAR(20),
  ASSETS VARCHAR(10)
);
INSERT INTO BRANCH VALUES('1','AKURDI','PUNE','HOUSE');
INSERT INTO BRANCH VALUES('2','RAVET','NASHIK','JEWELLERY');
INSERT INTO BRANCH VALUES('3','CHINCHWAD','AMRAVATI','FLAT');
INSERT INTO BRANCH VALUES('4','AKURDI','AMRAVAT','LAT');
INSERT INTO BRANCH VALUES('5','AKURDI','AMRAVA','AT');
INSERT INTO BRANCH VALUES('6','NIGDI','AMRAV','T');

CREATE TABLE CUSTOMER
(
   CUST_ID INTEGER,
  CUST_NAME VARCHAR(30),
  CUST_STREET VARCHAR(20),
  CUST_CITY VARCHAR(10)
);
INSERT INTO CUSTOMER VALUES('20','ABC','LINK ROAD','PUNE');
INSERT INTO CUSTOMER VALUES('21','BCD','LPRO ROAD','NASHIK');
INSERT INTO CUSTOMER VALUES('22','CDE','SHAGUN ROAD','AMRAVATI');

CREATE TABLE DEPOSITOR
(
   CUST_ID INTEGER,
   ACC_NO INTEGER
  );
INSERT INTO DEPOSITOR VALUES('20','10');
INSERT INTO DEPOSITOR VALUES('21','11');
INSERT INTO DEPOSITOR VALUES('22','12');

  CREATE TABLE LOAN
(
   LOAN_NO INTEGER,
   BRANCH_ID INTEGER,
   AMOUNT INTEGER
  );
INSERT INTO LOAN VALUES('100','31','10000');
INSERT INTO LOAN VALUES('101','32','20000');
INSERT INTO LOAN VALUES('102','33','30000');
INSERT INTO LOAN VALUES('103','6','90000');
   CREATE TABLE BORROWERR
```

```
(
   CUST_ID INTEGER,
   LOAN_NO INTEGER
  );

 INSERT INTO BORROWERR VALUES('41','1');
 INSERT INTO BORROWERR VALUES('42','2');
 INSERT INTO BORROWERR VALUES('43','3');

--LIST ALL CUSTOMERS IN ALPHABETICAL ORDER WHO HAVE LOAN IN AKURDI BRANCH--
SELECT CUST_NAME
FROM CUSTOMER,BRANCH
WHERE BRANCH_NAME='AKURDI'
ORDER BY CUST_NAME;

---FIND ALL CUSTOMERS WHO HAVE ACCOUNT OR LOAN OR BOTH AT BANK ---
SELECT CUST_NAME FROM CUSTOMER,DEPOSITOR,BORROWERR
WHERE CUSTOMER.CUST_ID=DEPOSITOR.CUST_ID OR BORROWERR.CUST_ID=CUSTOMER.CUST_ID;

---FIND ALL CUSTOMERS WHO HAVE BOTH ACCOUNT AND LOAN AT BAMK----
SELECT CUST_NAME FROM CUSTOMER,DEPOSITOR,BORROWERR
WHERE CUSTOMER.CUST_ID=DEPOSITOR.CUST_ID AND BORROWERR.CUST_ID=CUSTOMER.CUST_ID;

-----FIND ALL ACCOUNTS IN AKURDI BRANCH INCREASE THE BALANCE BY10%----
UPDATE ACCOUNT
SET BALANCE=BALANCE*1.1
WHERE BRANCH_NAME='AKURDI';

-----FIND AVERAGE ACCOUNT BALANCE AT AKURDI BRANCH-------
SELECT AVG(BALANCE) AS "AVERAGE BALANCE" FROM ACCOUNT
WHERE BRANCH_NAME='AKURDI';

-----FIND MINIMUM ACCOUNT BALANCE AT EACH BRANCH------
SELECT MIN(BALANCE) AS "MINIMUM BALANCE",BRANCH_NAME FROM ACCOUNT
GROUP BY BRANCH_NAME;

----DELETE ALL LOAN WITH LOAN AMOUNT BETWEEN 30000 AND 50000-----
DELETE FROM LOAN
WHERE AMOUNT<50000 AND AMOUNT>30000;


---A3(2)---

----FIND ALL BRANCHES WHERE AVERAGE BALANCE IS GREATER THAN 12000---
select BRANCH_NAME, avg (balance) from account
group by branch_name
having avg (balance) > 12000;

------FIND ALL CUSTOMERS WHO HAVE ACCOUNT BUT NOT LOAN ----
SELECT CUST_NAME FROM CUSTOMER,DEPOSITOR,BORROWERR
WHERE CUSTOMER.CUST_ID=DEPOSITOR.CUST_ID AND BORROWERR.CUST_ID!=CUSTOMER.CUST_ID;

----DELETE ALL TUPLES AT EVERY BRANCH LOCATED IN NIGDI-----
DELETE FROM ACCOUNT
WHERE BRANCH_NAME='NIGDI';

------FIND MAX LOAN AMOUNT IN NIGDI BRANCH-----
```

```
SELECT MAX(LOAN.AMOUNT) AS "MAXIMUM AMOUNT" FROM LOAN,BRANCH
WHERE LOAN.BRANCH_ID = BRANCH.BRANCH_ID AND BRANCH.BRANCH_NAME='NIGDI';

-------FIND NO. OF DEPOSITORS AT EACH BRANCH----
SELECT COUNT(DEPOSITOR.CUST_ID) AS "NO OF CUSTOMERS",ACCOUNT.BRANCH_NAME FROM
DEPOSITOR,ACCOUNT
WHERE DEPOSITOR.ACC_NO=ACCOUNT.ACC_NO
GROUP BY ACCOUNT.BRANCH_NAME;
```

Assignment No - A3

Problem statement: Design at least 10 SQL queries for suitable database application using SQL DML statements: all types of Join, Sub-Query and View.

Objective: To implement : SQL DML statements: all types of Join, Sub-Query and View

Solution:

A. Related theory
SQL Subquery

Subquery or **Inner query** or **Nested query** is a query in a query. SQL subquery is usually added in the WHERE Clause of the SQL statement. Most of the time, a subquery is used when you know how to search for a value using a SELECT statement, but do not know the exact value in the database.

Subqueries are an alternate way of returning data from multiple tables.

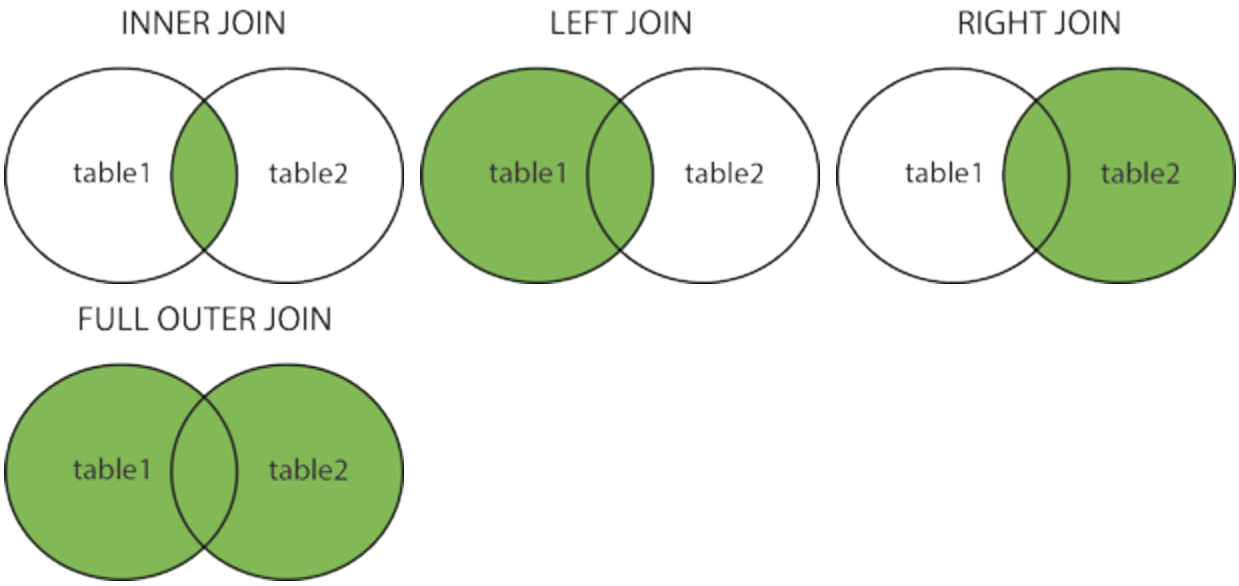
Subqueries can be used with the following SQL statements along with the comparision operators like =, <, >, >=, <= etc.

- SELECT
- INSERT
- UPDATE
- DELETE

Different Types of SQL JOINs

Here are the different types of the JOINs in SQL:

1. (INNER) JOIN: Returns records that have matching values in both tables
2. LEFT (OUTER) JOIN: Return all records from the left table, and the matched records from the right table
3. RIGHT (OUTER) JOIN: Return all records from the right table, and the matched records from the left table
4. FULL (OUTER) JOIN: Return all records when there is a match in either left or right table



SQL Views

A VIEW is a virtual table, through which a selective portion of the data from one or more tables can be seen. Views do not contain data of their own. They are used to restrict access to the database or to hide data complexity. A view is stored as a SELECT statement in the database. DML operations on a view like INSERT, UPDATE, DELETE affects the data in the original table upon which the view is based.

The Syntax to create a sql view is

```
CREATE VIEW view_name
AS
SELECT column_list
FROM table_name [WHERE condition];
```

- ***view_name*** is the name of the VIEW.
- The SELECT statement is used to define the columns and rows that you want to display in the view.

For Example: to create a view on the product table the sql query would be like

```
CREATE VIEW view_product
AS
SELECT product_id, product_name
FROM product;
```

B. Conclusion

10 SQL queries for suitable database application using SQL DML statements: all types of Join, Sub-Query and View are designed.

C. Sample Code

D. Sample Output

For the following relation schema:
employee(employee-name, street, city)
works(employee-name, company-name, salary)
company(company-name, city)
manages(employee-name, manager-name)

Create above tables and insert 5 rows in each table. Give an expression in SQL for each of the following queries:

1. Find the names, street address, and cities of residence for all employees who work for 'First Bank Corporation' and earn more than $10,000.
2. Find the names of all employees in the database who live in the same cities as the companies for which they work.
3. Display employee details that live in cities Pune, Mumbai, and Nasik
4. List employees from ‘First Bank Corporation’ that earn salary more than all employees of ‘Small Bank Corporation’
5. Create a view that will display employee details along with name of his/her manager.
6. Find average salary of employees of ‘First Bank Corporation’.

Give employees of ‘First Bank Corporation’ 15% rise if salary is less than 20000.

```
create table employee(emp_name VARCHAR(100),street VARCHAR(100) ,city VARCHAR(100));
create table work(name VARCHAR(100),company VARCHAR(100),salary int);
create table company(cname VARCHAR(100),city VARCHAR(100));
create table manages(name VARCHAR(100),manager VARCHAR(100));

insert into employee values('Rohit','Pimpri','Pune');
insert into work values('Rohit','SKF',20000);
INSERT INTO COMPANY VALUES('SKF','Pune');
insert into manages values('Rohit','Tejas');
insert into employee values('Rahul','akurdi','Mumbai');
insert into work values('Rahul','tata',20500);
INSERT INTO COMPANY VALUES('tata','Mumbai');
insert into manages values('Rahul','Rohit');
insert into employee values('Pittu','AKURDI','Pune');
insert into work values('Pittu','RKF',5000);
INSERT INTO COMPANY VALUES('RKF','Pune');
insert into manages values('Pittu','Raj');

--A4(1)
--1
select * from employee where city='Pune';

--2
SELECT a.name,a.street,a.city FROM EMPLOYEE a,work b WHERE a.name=b.name and
b.company='SKF' AND b.salary>30000;
```

```
--3
select distinct a.name from employee a,company b where a.city=b.city;

--4
select name from work where company!='SKF';

--5
select name from manages where manager='Tejas';

--6
select avg(salary) from work where company='RKF';

--7
create view mysql as select employee.name,street,city,manager from employee FULL join manages on
employee.name = manages.name;
SELECT * FROM MYsql;

--A4(2)

--1
SELECT a.name,a.street,a.city FROM EMPLOYEE a,work b WHERE a.name=b.name and
b.company='SKF' AND b.salary>10000;

--2
select distinct a.name from employee a,company b where a.city=b.city;

--3
select * from employee where city='Pune' or city='Mumbai' or city='Nashik';

--4
select name from work where COMPANY='tata' and salary > (select max(salary) from work where
company='RKF');

--5
create view my as select employee.name,street,city,manager from employee FULL join manages on
employee.name = manages.name;
SELECT * FROM MY;

--6
select avg(salary) from work where company='RKF';

--7
update work SET salary=(1.15*salary) where company='SKF' and salary<20000
```