Date: Roll No.

4.1 Implement SQL queries on a normalized database schema based on the provided schema. For this exampe: use the schema for a university database, which includes:

- Students (StudentID, StudentName, Major)
- Courses (CourseID, CourseName, Credits)
- Enrollments (StudentID, CourseID, EnrollmentDate)
- Instructors (InstructorID, InstructorName, Phone)
- Course_Instructors (CourseID, InstructorID)

Introduction to Normal Forms

In database management systems (DBMS), the concept of normalization is employed to organize relational databases efficiently and to eliminate redundant data, ensure data dependency, and ensure data integrity. The process of normalization is divided into several stages, called "normal forms." Each normal form has a specific set of rules and criteria that a database schema must meet. Here's a brief overview of the main normal forms:

1. First Normal Form (1NF)

- Each table should have a primary key.
- Atomic values: Each attribute (column) of a table should hold only a single value, meaning no repeating groups or arrays.
- All entries in any column must be of the same kind.

Second Normal Form (2NF)

- It meets all the requirements of 1NF.
- It ensures that non-key attributes are fully functionally dependent on the primary key. In other words, if a table has a composite primary key, then every non-key attribute should be dependent on the full set of primary key attributes.

Third Normal Form (3NF)

- It meets all the requirements of 2NF.
- It ensures that the non-key columns are functionally dependent only on the primary key. This means there should be no transitive dependencies.

Boyce-Codd Normal Form (BCNF)

- Meets all requirements of 3NF.
- For any non-trivial functional dependency, X → Y, X should be a superkey. It's a more stringent version of 3NF.

Fourth Normal Form (4NF)

- Meets all the requirements of BCNF.
- There shouldn't be any multi-valued dependency for a super key. This deals with separating independent multiple relationships, ensuring that you cannot determine multiple sets of values in a table from a single key attribute.

Date: Roll No.

Fifth Normal Form (5NF or Project-Join Normal Form - PJNF)

- It deals with cases where certain projections of your data must be re-creatable from other projections. Normalization often involves trade-offs. While higher normal forms eliminate redundancy and improve data integrity, they can also result in more complex relational schemas and sometimes require more joins, which can affect performance. As such, it's essential to understand the data and the specific application's requirements when deciding the level of normalization suitable for a particular situation. Sometimes, denormalization (intentionally introducing redundancy) is implemented to improve performance, especially in read-heavy databases.

Query to create University database and create table Univ_denorm:

```
create database university;

use university;

CREATE TABLE univ_denorm (
  StudentID INTEGER, StudentName TEXT, Major TEXT,
  CourseID TEXT, CourseName TEXT, Credits INTEGER,
  EnrollmentDate TEXT,
  InstructorID INTEGER, InstructorName TEXT, Phone TEXT
);
```

Query to insert data values into univ_denorm table.

```
INSERT INTO univ_denorm
(StudentID, StudentName, Major, CourseID, CourseName, Credits, EnrollmentDate, InstructorID,
InstructorName, Phone)
VALUES
(1, 'Alice Smith', 'Computer Science', 'C101', 'Intro to CS', 3, '2025-09-01', 10, 'Dr. Adams', '555-
0100'),
(1, 'Alice Smith', 'Computer Science', 'C101', 'Intro to CS', 3, '2025-09-01', 11, 'Dr. Baker', '555-
0111'),
(2, 'Bob Jones', 'Mathematics', 'C101', 'Intro to CS', 3, '2025-09-03', 10, 'Dr. Adams', '555-0100'),
(2, 'Bob Jones', 'Mathematics', 'C101', 'Intro to CS', 3, '2025-09-03', 11, 'Dr. Baker', '555-0111'),
(1, 'Alice Smith', 'Computer Science', 'C102', 'Calculus I', 4, '2025-09-02', 11, 'Dr. Baker', '555-
0111'),
(3, 'Carol Lee', 'Physics', 'C103', 'Physics I', 4, '2025-09-04', 12, 'Dr. Clark', '555-0122');
```

Query to view data in the table:

```
Select * from univ_denorm;
```

Date: Roll No.

Output:

StudentID	StudentName	Major	CourseID	CourseName	Credits	EnrollmentDate	InstructorID	InstructorName	Phone
1	Alice Smith	Computer Science	C101	Intro to CS	3	2025-09-01	10	Dr. Adams	555-0100
1	Alice Smith	Computer Science	C101	Intro to CS	3	2025-09-01	11	Dr. Baker	555-0111
2	Bob Jones	Mathematics	C101	Intro to CS	3	2025-09-03	10	Dr. Adams	555-0100
2	Bob Jones	Mathematics	C101	Intro to CS	3	2025-09-03	11	Dr. Baker	555-0111
1	Alice Smith	Computer Science	C102	Calculus I	4	2025-09-02	11	Dr. Baker	555-0111
3	Carol Lee	Physics	C103	Physics I	4	2025-09-04	12	Dr. Clark	555-0122

Functional Dependencies:

- StudentID → StudentName, Major
- CourseID → CourseName, Credits
- InstructorID → InstructorName, Phone
- (StudentID, CourseID) → EnrollmentDate

Candiadate key identified is:

(StudentID, CourseID, InstructorID) → All other attributes

Normalization to Second Normal Form (2NF)

Students Table:

StudentID	StudentName	Major
1	Alice Smith	Computer Science
2	Bob Jones	Mathematics
3	Carol Lee	Physics

Courses Table

CourseID	CourseName	Credits
C101	Intro to CS	3
C102	Calculus I	4
C103	Physics I	4

Instructors Table

InstructorID	InstructorName	Phone
10	Dr. Adams	555-0100
11	Dr. Baker	555-0111
12	Dr. Clark	555-0122

Enrollments Table

StudentID	CourseID	EnrollmentDate
1	C101	2025-09-01
2	C101	2025-09-03
1	C102	2025-09-02
3	C103	2025-09-04

Date: Roll No.

Queries to generate normalized tables:

```
CREATE TABLE Students AS
  SELECT DISTINCT StudentID, StudentName, Major FROM univ_denorm;

CREATE TABLE Courses AS
  SELECT DISTINCT CourseID, CourseName, Credits FROM univ_denorm;

CREATE TABLE Instructors AS
  SELECT DISTINCT InstructorID, InstructorName, Phone FROM univ_denorm;

CREATE TABLE Enrollments AS
  SELECT DISTINCT StudentID, CourseID, EnrollmentDate FROM univ_denorm;

CREATE TABLE Course_Instructors AS
  SELECT DISTINCT StudentID,CourseID, InstructorID FROM univ_denorm;
```

After splitting into new tables and verifying the normalised tables are in 3NF, BCNF, 4NF.

In order to verify 5NF performing JOIN dependency.

```
SELECT e.StudentID, s.StudentName, s.Major,
       e.CourseID, c.CourseName, c.Credits, e.EnrollmentDate,
       ci.InstructorID, i.InstructorName, i.Phone
FROM Enrollments e
JOIN Students s ON e.StudentID = s.StudentID
JOIN Courses c ON e.CourseID = c.CourseID
JOIN Course_Instructors ci ON c.CourseID = ci.CourseID
JOIN Instructors i ON ci.InstructorID = i.InstructorID
ORDER BY e.StudentID, e.CourseID, ci.InstructorID;
```

Output:

StudentID	StudentName	Major	CourseID	CourseName	Credits	EnrollmentDate	InstructorID	InstructorName	Phone
1	Alice Smith	Computer Science	C101	Intro to CS	3	2025-09-01	10	Dr. Adams	555-0100
1	Alice Smith	Computer Science	C101	Intro to CS	3	2025-09-01	11	Dr. Baker	555-0111
2	Bob Jones	Mathematics	C101	Intro to CS	3	2025-09-03	10	Dr. Adams	555-0100
2	Bob Jones	Mathematics	C101	Intro to CS	3	2025-09-03	11	Dr. Baker	555-0111
1	Alice Smith	Computer Science	C102	Calculus I	4	2025-09-02	11	Dr. Baker	555-0111
3	Carol Lee	Physics	C103	Physics I	4	2025-09-04	12	Dr. Clark	555-0122

Date: Roll No. | | | | | | | | | |

4.2 (A)Implementation of Data Control Language commands – grant and revoke.
(B)Implementation of Transaction Control Language commands - commit, save point, and rollback.

(A) DCL mainly has **GRANT** and **REVOKE**, which are used to control **access rights and permissions** on database objects.

Data Control Language (DCL) in Oracle
DCL commands in Oracle are used to **control access to data and database objects**.
The two main commands are:

- **GRANT** – To provide access rights.
- **REVOKE** – To withdraw access rights.

Step 1: DBA creates a user

```
CREATE USER student_user IDENTIFIED BY password;
GRANT CREATE SESSION TO student_user;
```

Step 2: DBA grants object privileges

```
GRANT SELECT, INSERT ON Students TO student_user;
```

Step 3: student_user accesses Students table

```
SELECT * FROM Students;
INSERT INTO Students VALUES (101, 'John', 'CSE');
```

Step 4: DBA revokes privilege

```
REVOKE INSERT ON Students FROM student_user;
```

Now student_user can still **SELECT**, but cannot **INSERT** into Students.

(B)Implementation of Transaction Control Language commands - commit, save point, and rollback.

Transaction Control Language (TCL)

- TCL commands are used to **manage transactions** in a database.
- A transaction is a sequence of SQL operations performed as a single logical unit of work.
- TCL commands ensure **data integrity and consistency**.

1. COMMIT

- Commits (saves) all the changes made by a transaction permanently to the database.
- After COMMIT, changes cannot be undone.

Syntax:
```
COMMIT;
```

Example:
```
INSERT INTO Students VALUES (101, 'Ravi', 'CSE');

INSERT INTO Students VALUES (102, 'Anita', 'ECE');
COMMIT;  -- permanently saves the above two insertions
```

2. SAVEPOINT

- Creates a checkpoint (marker) within a transaction.
- Allows partial rollback to that point without affecting the entire transaction.
- Multiple savepoints can be created in a single transaction.

Syntax:
```
SAVEPOINT savepoint_name;
```

Example:
```
INSERT INTO Students VALUES (103, 'Mahesh', 'IT');
SAVEPOINT sp1;

INSERT INTO Students VALUES (104, 'Sita', 'EEE');
SAVEPOINT sp2;

DELETE FROM Students WHERE StudentID = 101;
```
Here, we created two savepoints (sp1 and sp2) that allow us to roll back specific parts of the transaction.

3. ROLLBACK

- Rolls back (undoes) all changes made in the current transaction (if no COMMIT has been issued).
- Can also rollback **partially** to a specific **SAVEPOINT**.

Syntax:
```
ROLLBACK;     -- Undo all uncommitted changes
ROLLBACK TO savepoint_name; -- Undo changes after the specified savepoint
```

Example:
```
ROLLBACK TO sp1;   -- undo operations done after savepoint sp1
```

Date: Roll No.

5.1. How to Create a Primary , Secondary Index on a Column?

5.2. How to to Retrieve Data Using an Index?

5.3. How to Insert Data and Update Indexes?

5.4. How to Delete Data and Impact on Indexes?

5.1. (A) Setup: Create database and sample table

```
CREATE TABLE employees (
  emp_id INT,
  first_name VARCHAR(50),
  last_name VARCHAR(50),
  dept VARCHAR(50),
  salary INT,
  hired_date DATE,
  PRIMARY KEY (emp_id) -- creates a primary index on emp_id
) ENGINE=InnoDB;
```

Expected output

```
Query OK, 1 row affected (0.01 sec)
Database changed
Query OK, 0 rows affected (0.02 sec)
```

Note: `PRIMARY KEY (emp_id)` creates a clustered primary index (InnoDB) on `emp_id`.

(B). Creating a Secondary Index

A secondary index is useful for fast lookups on non-primary-key columns.

```
-- Create a non-unique (secondary) index on last_name

CREATE INDEX idx_lastname ON employees(last_name);

-- Create a composite secondary index on (dept, salary)

CREATE INDEX idx_dept_salary ON employees(dept, salary);
```

Expected output

```
Query OK, 0 rows affected (0.01 sec)
Query OK, 0 rows affected (0.01 sec)
```

To list indexes on a table:

```
SHOW INDEX FROM employees;
```

Date: Roll No.

Expected output (example rows)

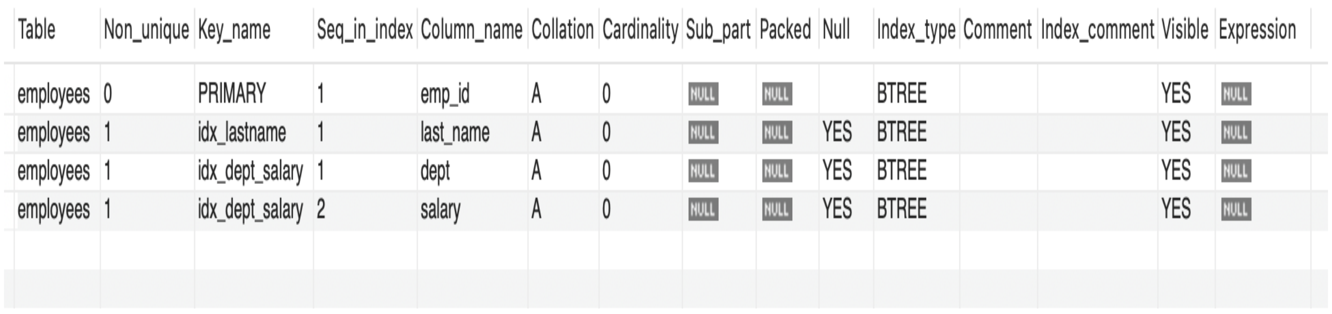


Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment	Index_comment	Visible	Expression
employees	0	PRIMARY	1	emp_id	A	0	NULL	NULL		BTREE			YES	NULL
employees	1	idx_lastname	1	last_name	A	0	NULL	NULL	YES	BTREE			YES	NULL
employees	1	idx_dept_salary	1	dept	A	0	NULL	NULL	YES	BTREE			YES	NULL
employees	1	idx_dept_salary	2	salary	A	0	NULL	NULL	YES	BTREE			YES	NULL

ADITYA UNIVERSITY

Date: Roll No.

5. 2. Retrieve Data Using an Index (and verify with EXPLAIN)

Insert some sample rows first:

```
INSERT INTO employees (first_name, last_name, dept, salary, hired_date) VALUES
('Alice','Kumar','R&D',80000,'2020-03-15'),
('Bob','Sharma','Sales',60000,'2019-08-01'),
('Carol','Kumar','R&D',85000,'2021-02-10'),
('Dave','Patel','Marketing',55000,'2018-11-20'),
('Eve','Sharma','Sales',62000,'2022-01-05');
```

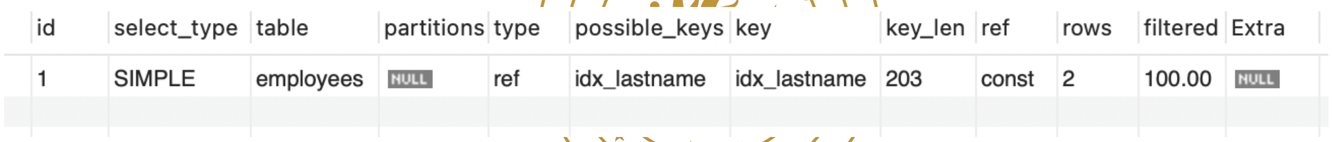
Expected output

```
Query OK, 5 rows affected (0.01 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

Now query using an indexed column `last_name`:

```
EXPLAIN SELECT * FROM employees WHERE last_name = 'Sharma';
SELECT * FROM employees WHERE last_name = 'Sharma';
```

Expected EXPLAIN output (simplified)



id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employees	NULL	ref	idx_lastname	idx_lastname	203	const	2	100.00	NULL

Expected SELECT result

```
+--------+------------+-----------+----------+--------+------------+
| emp_id | first_name | last_name | dept     | salary | hired_date |
+--------+------------+-----------+----------+--------+------------+
|      2 | Bob        | Sharma    | Sales    |  60000 | 2019-08-01 |
|      5 | Eve        | Sharma    | Sales    |  62000 | 2022-01-05 |
+--------+------------+-----------+----------+--------+------------+
```

Explanation

- The `EXPLAIN` shows `key = idx_lastname`, which confirms the DB used the index for the lookup.
- `type = ref` indicates an index lookup returning matching rows.

5.3. (A) How INSERT Affects Indexes

```
INSERT INTO employees (first_name, last_name, dept, salary, hired_date)
VALUES ('Frank','Kumar','R&D',78000,'2024-07-01');
```

Expected output

```
Query OK, 1 row affected (0.01 sec)
```

What happens under the hood

- The row is inserted into the clustered table (InnoDB) according to `emp_id` (PRIMARY KEY).
- Secondary index entries (`idx_lastname`, `idx_dept_salary`) are also inserted/updated automatically to include pointers to the new row.
- Index maintenance is automatic — no manual rebuild required for normal inserts.

Expected rows: Alice, Carol, Frank.

(B) How UPDATE Affects Indexes

Case A: Updating a non-indexed column (e.g., `salary` when salary is not indexed alone) — index impact is minimal.

```
UPDATE employees SET salary = salary + 2000 WHERE last_name = 'Sharma';
```

Expected output

```
Query OK, 2 rows affected (0.01 sec)
Rows matched: 2  Changed: 2  Warnings: 0
```

- Only the table rows change; secondary index entries remain unchanged if indexed columns are not modified.

Case B: Updating an indexed column (e.g., `last_name`) — index entries must be updated.

```
UPDATE employees SET last_name = 'Sharma-Old' WHERE emp_id = 2;
```

Expected output

```
Query OK, 1 row affected (0.01 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

- The DB locates the row (via primary key / clustered index), updates the row, deletes the old secondary index entry for `last_name='Sharma'` and inserts a new index entry for `last_name='Sharma-Old'`.
- This causes more I/O than updating a non-indexed column.

5.4. (A) How DELETE Affects Indexes

```
DELETE FROM employees WHERE emp_id = 4; -- delete Dave
```

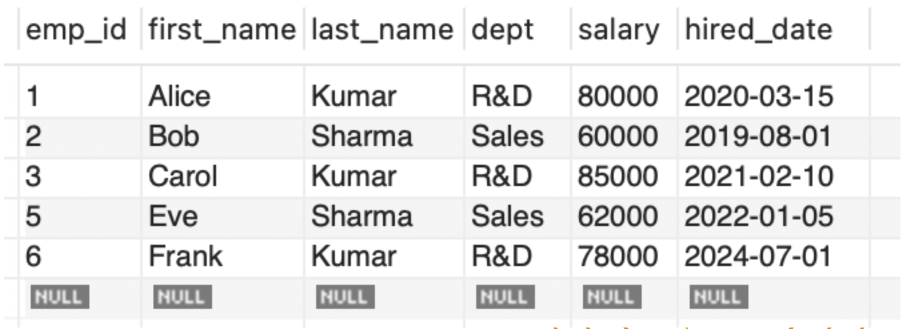
Expected output

```
Query OK, 1 row affected (0.01 sec)
```

What happens

- The row is removed from the clustered storage (PRIMARY index).
- All secondary index entries that referenced that row are also removed automatically.
- Indexes remain present, but their cardinality/statistics change.

Expected `SELECT *` sample



emp_id	first_name	last_name	dept	salary	hired_date
1	Alice	Kumar	R&D	80000	2020-03-15
2	Bob	Sharma	Sales	60000	2019-08-01
3	Carol	Kumar	R&D	85000	2021-02-10
5	Eve	Sharma	Sales	62000	2022-01-05
6	Frank	Kumar	R&D	78000	2024-07-01
NULL	NULL	NULL	NULL	NULL	NULL

Note on performance

- Frequent deletes can cause fragmentation; some DB engines offer `OPTIMIZE TABLE` or `VACUUM` (Postgres) to reclaim space and rebuild indexes.

(B). Dropping and Rebuilding Indexes

```
-- Drop a secondary index
DROP INDEX idx_lastname ON employees;

-- Recreate if needed
CREATE INDEX idx_lastname ON employees(last_name);
```

Expected output

```
Query OK, 0 rows affected (0.01 sec)
Query OK, 0 rows affected (0.01 sec)
```

If you need to rebuild all indexes on a MySQL table:

```
ALTER TABLE employees ENGINE=InnoDB; -- forces rebuild
```