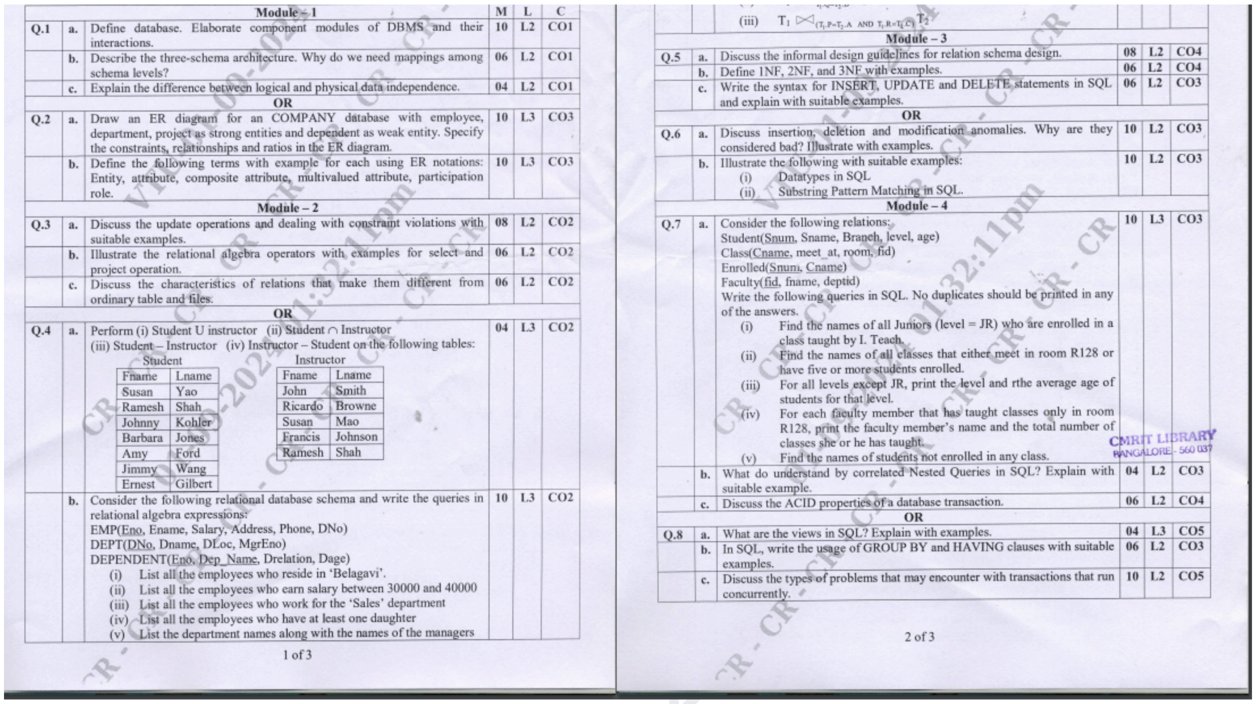
DBMS-BCS403 SOLUTION JUNE/JULY 2024



		Module – 1	M	L	C
Q.1	a.	Define database. Elaborate component modules of DBMS and their interactions.	10	L2	CO1
	b.	Describe the three-schema architecture. Why do we need mappings among schema levels?	06	L2	CO1
	c.	Explain the difference between logical and physical data independence.	04	L2	CO1
		OR			
Q.2	a.	Draw an ER diagram for an COMPANY database with employee, department, project as strong entities and dependent as weak entity. Specify the constraints, relationships and ratios in the ER diagram.	10	L3	CO3
	b.	Define the following terms with example for each using ER notations: Entity, attribute, composite attribute, multivalued attribute, participation role.	10	L3	CO3
		Module – 2			
Q.3	a.	Discuss the update operations and dealing with constraint violations with suitable examples.	08	L2	CO2
	b.	Illustrate the relational algebra operators with examples for select and project operation.	06	L2	CO2
	c.	Discuss the characteristics of relations that make them different from ordinary table and files.	06	L2	CO2
		OR			
Q.4	a.	Perform (i) Student U instructor (ii) Student ∩ Instructor (iii) Student – Instructor (iv) Instructor – Student on the following tables:	04	L3	CO2

Student

Fname	Lname
Susan	Yao
Ramesh	Shah
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert

Instructor

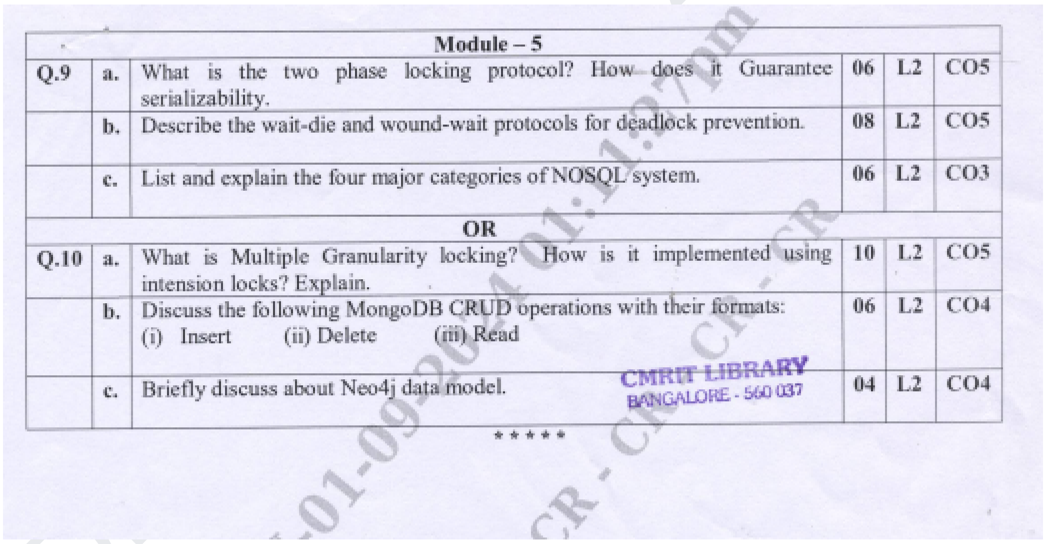
Fname	Lname
John	Smith
Ricardo	Browne
Susan	Mao
Francis	Johnson
Ramesh	Shah

	b.	Consider the following relational database schema and write the queries in relational algebra expressions: EMP(Eno, Ename, Salary, Address, Phone, DNo) DEPT(DNo, Dname, DLoc, MgrEno) DEPENDENT(Eno, Dep_Name, Drelation, Dage) (i) List all the employees who reside in 'Belagavi'. (ii) List all the employees who earn salary between 30000 and 40000 (iii) List all the employees who work for the 'Sales' department (iv) List all the employees who have at least one daughter (v) List the department names along with the names of the managers	10	L3	CO2

1 of 3

		(iii) $T_1 \bowtie_{(T_1.P=T_2.A \text{ AND } T_1.R=T_2.C)} T_2$			
		Module – 3			
Q.5	a.	Discuss the informal design guidelines for relation schema design.	08	L2	CO4
	b.	Define 1NF, 2NF, and 3NF with examples.	06	L2	CO4
	c.	Write the syntax for INSERT, UPDATE and DELETE statements in SQL and explain with suitable examples.	06	L2	CO3
		OR			
Q.6	a.	Discuss insertion, deletion and modification anomalies. Why are they considered bad? Illustrate with examples.	10	L2	CO3
	b.	Illustrate the following with suitable examples: (i) Datatypes in SQL (ii) Substring Pattern Matching in SQL.	10	L2	CO3
		Module – 4			
Q.7	a.	Consider the following relations: Student(Snum, Sname, Branch, level, age) Class(Cname, meet_at, room, fid) Enrolled(Snum, Cname) Faculty(fid, fname, deptid) Write the following queries in SQL. No duplicates should be printed in any of the answers. (i) Find the names of all Juniors (level = JR) who are enrolled in a class taught by I. Teach. (ii) Find the names of all classes that either meet in room R128 or have five or more students enrolled. (iii) For all levels except JR, print the level and rthe average age of students for that level. (iv) For each faculty member that has taught classes only in room R128, print the faculty member's name and the total number of classes she or he has taught. (v) Find the names of students not enrolled in any class.	10	L3	CO3
	b.	What do understand by correlated Nested Queries in SQL? Explain with suitable example.	04	L2	CO3
	c.	Discuss the ACID properties of a database transaction.	06	L2	CO4
		OR			
Q.8	a.	What are the views in SQL? Explain with examples.	04	L3	CO5
	b.	In SQL, write the usage of GROUP BY and HAVING clauses with suitable examples.	06	L2	CO3
	c.	Discuss the types of problems that may encounter with transactions that run concurrently.	10	L2	CO5

2 of 3



		Module – 5			
Q.9	a.	What is the two phase locking protocol? How does it Guarantee serializability.	06	L2	CO5
	b.	Describe the wait-die and wound-wait protocols for deadlock prevention.	08	L2	CO5
	c.	List and explain the four major categories of NOSQL system.	06	L2	CO3
		OR			
Q.10	a.	What is Multiple Granularity locking? How is it implemented using intension locks? Explain.	10	L2	CO5
	b.	Discuss the following MongoDB CRUD operations with their formats: (i) Insert (ii) Delete (iii) Read	06	L2	CO4
	c.	Briefly discuss about Neo4j data model.	04	L2	CO4

* * * * *

MODULE 1

Q1a. Define database. Elaborate component modules of DBMS and their interactions.

A **database** is a collection of logically coherent data with some inherent meaning, representing aspects of the real world, organized in such a way that it can be easily accessed, managed, and updated.

A **DBMS** consists of the following major components:

- **Storage Manager**: Manages the storage of data and meta-data.
- **Query Processor**: Converts user queries into low-level instructions for efficient access.
- **Transaction Manager**: Manages the execution of transactions while ensuring ACID properties (Atomicity, Consistency, Isolation, and Durability).
- **Concurrency Control Manager**: Ensures transactions do not interfere with each other.

These modules interact to process user queries, ensure data consistency, manage concurrent access, and store the data efficiently .

1. Interactive Query Interface

- Casual or occasional users interact with the database through an **interactive query interface**.
- Queries go through a **query compiler** that parses and validates them for correctness (syntax, file names, etc.), and converts them into an internal form.
- The **query optimizer** improves the query by rearranging operations, eliminating redundancies, and choosing efficient search algorithms, using statistics from the system catalog. This results in **executable code** that is passed to the runtime processor.

2. Application Programs

- **Application programmers** write programs using languages like Java, C, or C++ which interact with the database.
- A **precompiler** extracts **DML (Data Manipulation Language)** commands from the program, which are compiled separately by the **DML compiler**.
- The rest of the program is compiled by the host language compiler, and the two parts (DML commands and the main program) are linked together to form a **canned transaction**.

- **Canned transactions** are executed repeatedly by users who supply parameters (e.g., in a bank payment transaction, account number, payee, and amount are parameters). This approach is becoming popular with scripting languages like PHP and Python.

3. Runtime Database Processor

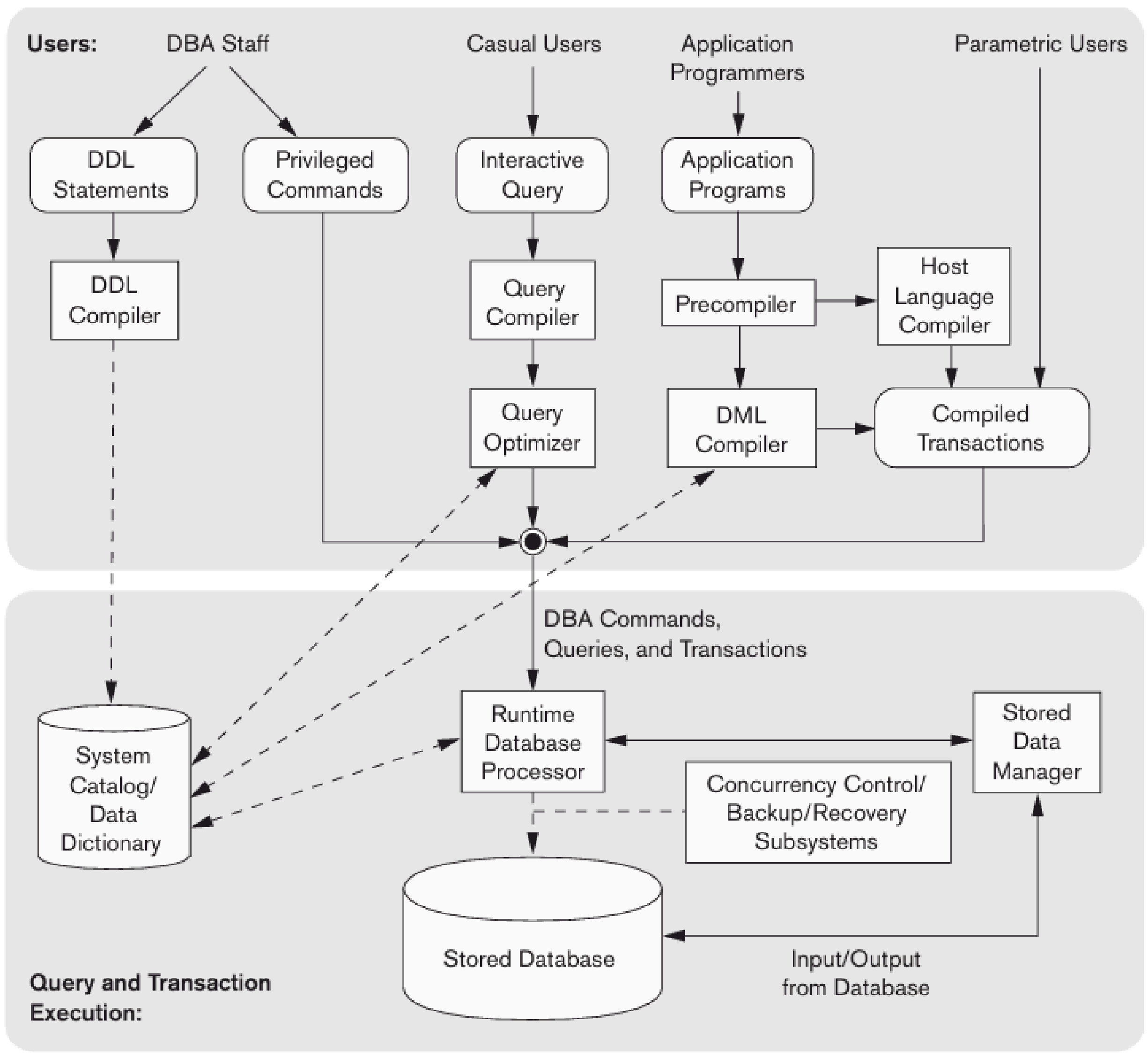
- The **runtime database processor** executes **privileged commands**, **executable query plans**, and **canned transactions** with parameters.
- It interacts with the **system catalog** and may update statistics related to the database. It also works with the **stored data manager**, which handles I/O operations between the disk and main memory.
- **Buffer management** is handled either by the DBMS or the operating system. The DBMS also includes **concurrency control** and **backup/recovery modules** for transaction management.

4. Client-Server Architecture

- Typically, the **client program** runs on a separate computer (client) that accesses the database, which resides on a **database server**.
- In some cases, the client accesses a **middle-tier application server**, which in turn accesses the database server.
- The **operating system** schedules disk accesses and DBMS processing if the system is shared by many users. If dedicated, the DBMS may control memory buffering of disk pages directly.

5. DBMS Interaction with the OS

- The DBMS interacts with the operating system when it needs to perform **disk accesses** (for the database or catalog).
- The DBMS can also interface with **compilers** for host programming languages and with client programs through the **system network interface**.



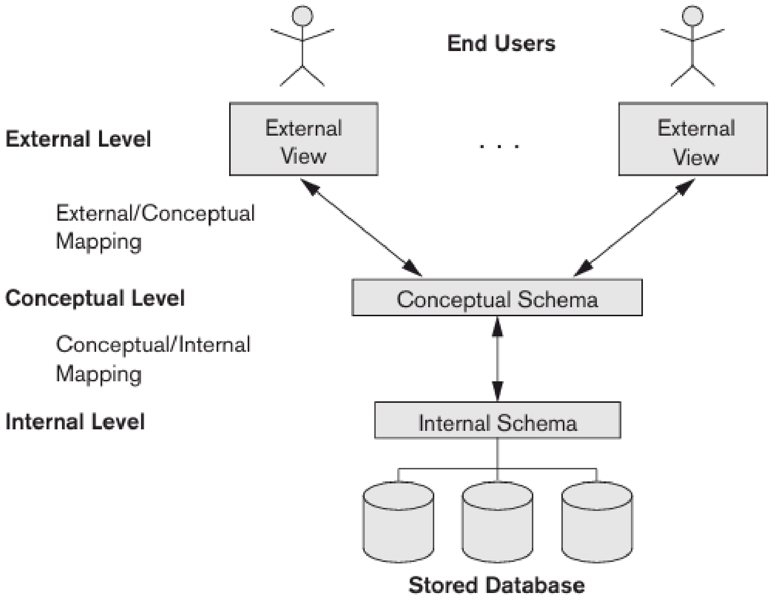
Q1b. Describe the three-schema architecture. Why do we need mappings among schema levels?

The **three-schema architecture** separates the user's view from the physical database and defines three levels:

1. **Internal Level**: Describes the physical storage structure.
2. **Conceptual Level**: Describes the structure of the database for a community of users.
3. **External Level**: Defines different views for users.

Mappings among these schemas are essential to transform a request from an external schema to the conceptual schema and then to the internal schema. This ensures data independence at different levels.

Figure 2.2
The three-schema architecture.



[9]This is also known as the ANSI/SPARC (American National Standards Institute/ Standards Planning And Requirements Committee) architecture, after the committee that proposed it (Tsichritzis & Klug, 1978).

Q1c. Explain the difference between logical and physical data independence.

- **Logical data independence** allows changing the conceptual schema without affecting the external schemas or application programs.
- **Physical data independence** allows changing the internal schema without affecting the conceptual schema

Q2a. Draw an ER diagram for a company database with employee, department, project as strong entities, and dependent as a weak entity.

Here is a simplified explanation for the ER diagram:

- **Entities**: Employee, Department, Project (Strong), Dependent (Weak)
- **Relationships**: Employee works in a Department, Employee works on a Project, Dependent is related to Employee
- **Constraints**: Employee has one Department, and works on multiple projects, Dependent weak entity relies on Employee.

The diagram would look similar to the one in the textbook showing employees, departments, and dependents.

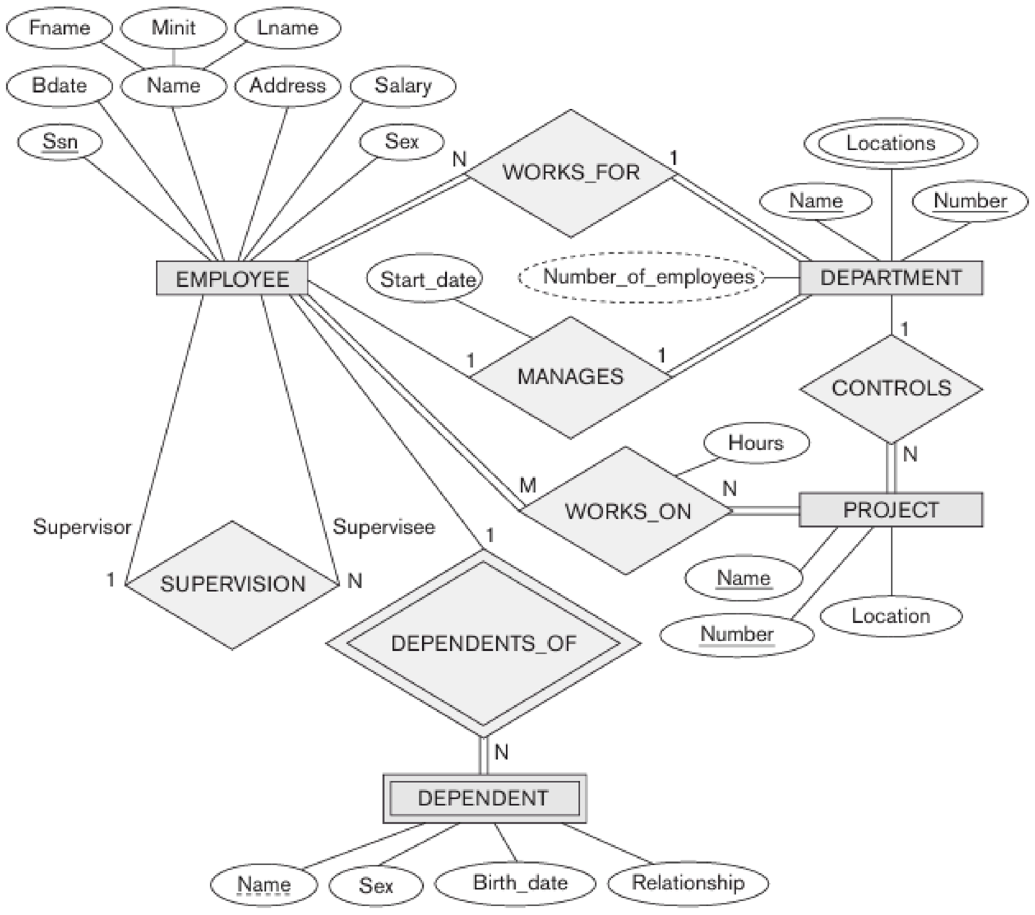


Figure 3.2
An ER schema diagram for the COMPANY database. The diagrammatic notation is introduced gradually throughout this chapter and is summarized in Figure 3.14.

Q2b. Define the following terms with examples using ER notations: Entity, attribute, composite attribute, multivalued attribute, participation role.

- **Entity**: Represents a real-world object, e.g., Employee.
- **Attribute**: Property of an entity, e.g., Employee's name.
- **Composite Attribute**: Attribute with components, e.g., Name (First Name, Last Name).
- **Multivalued Attribute**: Attribute that can hold multiple values, e.g., Phone Numbers.
- **Participation Role**: Refers to an entity's role in a relationship, e.g., Employee *works in* Department

MODULE 2

Q3a. Discuss update operations and constraint violations with examples.

Update operations include Insert, Delete, and Update. These can violate constraints such as:

1. Relational Model Operations

- The operations of the relational model are categorized into **retrievals** and **updates**.
- **Retrieval operations** are typically expressed using **relational algebra** or **relational calculus** (introduced in later chapters). Relational algebra forms a new relation by applying algebraic operators on existing relations, which helps in querying the database.
- **Update operations** modify the database state, including the **Insert**, **Delete**, and **Update (Modify)** operations.

2. Insert Operation

- **Insert** adds new tuples (records) to a relation. However, integrity constraints must be maintained:
  - **Domain constraints** ensure that each value belongs to the proper domain (data type or value range).
  - **Key constraints** prevent duplicate primary key values.

    - **Entity integrity** ensures that primary key attributes cannot be NULL.
    - **Referential integrity** ensures that foreign keys reference valid tuples in other relations.
- **Examples:**
    - Inserting a tuple with a **NULL primary key** violates entity integrity and is rejected.
    - Inserting a tuple with an existing **primary key value** violates key constraints and is rejected.
    - Inserting a tuple with an invalid **foreign key** violates referential integrity.
    - A valid insert operation satisfies all constraints.

3. Delete Operation

- **Delete** removes tuples from a relation, and it can only violate **referential integrity**.
    - **Referential integrity violation** occurs if other tuples reference the tuple being deleted. For example, deleting an **EMPLOYEE** tuple could cause issues if other relations like **WORKS_ON** or **DEPENDENT** reference it.
- **Options for Handling Violations:**
    - **Restrict:** Reject the deletion.
    - **Cascade:** Delete the referencing tuples as well (e.g., deleting related **WORKS_ON** tuples).
    - **Set Null or Set Default:** Modify the referencing attributes, either by setting them to NULL or to a default value.

4. Update Operation

- **Update** modifies attribute values in existing tuples. It can violate several constraints:
    - **Domain constraints** if the new value is not of the correct type.
    - **Key constraints** if updating the primary key duplicates an existing value.
    - **Referential integrity** if modifying a foreign key references a non-existing tuple.
- **Examples:**
    - Updating an employee’s salary or department number is acceptable as long as it doesn't violate any integrity constraints.

    - Updating a primary key or foreign key might require checking for consistency with other tuples or relations.

5. Handling Constraint Violations

- The DBMS can reject operations that violate integrity constraints by default or provide corrective options such as asking for valid inputs.
- Similar handling options for **Update** as with **Delete**, like **restrict**, **cascade**, or **set null**.

Q3b. Relational algebra operators for select and project operations.

- **Select (σ)**: Retrieves rows that satisfy a condition, e.g., σ(Salary > 30000)(Employee).
- **Project (π)**: Retrieves specific columns, e.g., π(Name, Salary)(Employee).

Q3c. Characteristics of relations that make them different from ordinary data.

1. Tabular Structure (Relations are Tables)

- A relation is represented as a table of **rows** and **columns**. Each row represents a **tuple** (or record), and each column represents an **attribute** (or field).
- **Tuples** in a relation are ordered conceptually but not physically, meaning the order of rows does not affect the relation.

2. Attributes and Domains

- Each column (attribute) in a relation is associated with a **domain**, which defines the permissible values the attribute can take. For example, a domain for a "Date of Birth" column might only accept valid date values.
- Attributes must maintain **atomicity**, meaning every value in a relation must be atomic or indivisible (not a set or list of values).

3. Uniqueness of Tuples

- In a relational model, **each tuple must be unique**. No two rows can have identical values for all attributes. This is enforced by the **primary key**, a unique identifier for each tuple.

- **Primary keys** ensure the integrity of the data and allow the database to reference individual records precisely.

4. No Duplicates

- Relations do not allow **duplicate tuples**. Each tuple must be distinct, as a relation is a set of tuples and sets do not contain duplicate elements.

5. Unordered Tuples and Attributes

- **Tuples** (rows) in a relation are not ordered. Unlike arrays or lists in programming, the order of tuples is irrelevant, and the DBMS does not enforce a particular order.
- **Attributes** (columns) are also unordered, meaning the left-to-right order of columns does not affect the definition or behavior of the relation.

6. Integrity Constraints

- **Relations** are subject to various **integrity constraints** that ensure the correctness and consistency of the data:
  - **Domain constraints**: Values in each column must come from the attribute's domain.
  - **Key constraints**: The **primary key** must be unique for every tuple.
  - **Referential integrity**: A foreign key in one relation must reference a valid tuple in another relation.
  - **Entity integrity**: The primary key must not be NULL.

7. Relational Operations

- Relations support specific operations defined by **relational algebra**, including:
  - **Selection**: Retrieving rows based on conditions.
  - **Projection**: Retrieving specific columns.
  - **Join**: Combining two relations based on a related attribute.
  - **Union, Intersection, and Difference**: Set-based operations to combine or compare relations.

8. Data Independence

- Relations are abstract representations of data, which provides **data independence**. Changes to the physical storage of data do not affect the logical structure of relations, meaning users interact with data at a higher level of abstraction.

9. Set-Based Theory

- Relations are fundamentally based on **set theory** and **predicate logic**. They are considered sets of tuples where set operations like union, intersection, and difference can be performed. This makes relations different from ordered data structures in programming.

10. Null Values

- Relations can accommodate **NULL** values, which represent unknown, missing, or inapplicable information. However, certain constraints, like primary keys, may disallow NULL values.

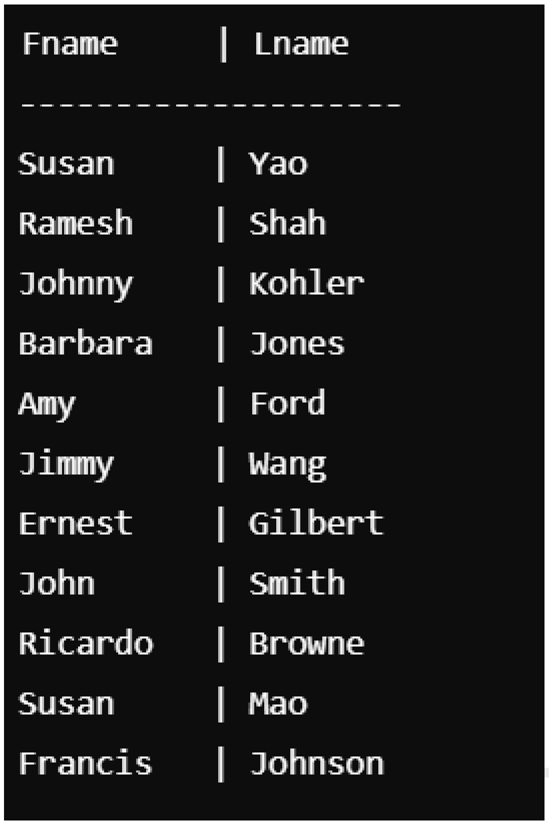
11. Normalization

- Relations can be **normalized** to reduce redundancy and improve data integrity. Normalization organizes data into smaller, related relations to eliminate anomalies during insertion, deletion, or update operations.
-

a. Perform the following:
(i) Student U Instructor
(ii) Student ∩ Instructor
(iii) Student - Instructor (iv) Instructor - Student on the following tables:

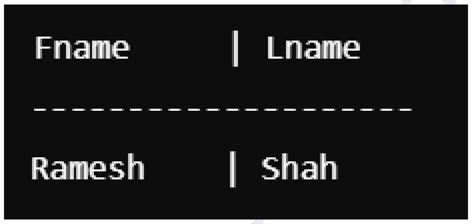
- **Student: Fname, Lname**
- **Instructor: Fname, Lname**

i)



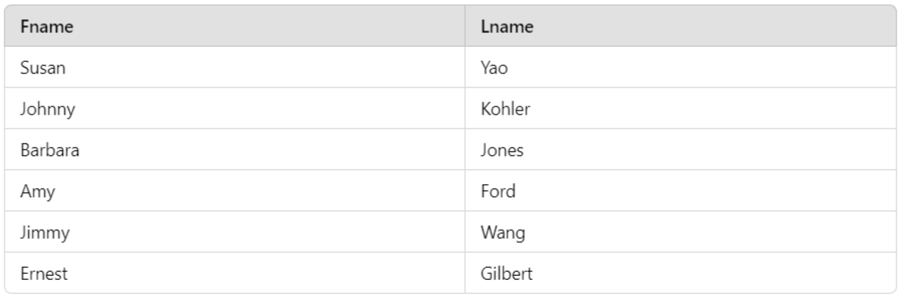
```
Fname     | Lname
--------------------
Susan     | Yao
Ramesh    | Shah
Johnny    | Kohler
Barbara   | Jones
Amy       | Ford
Jimmy     | Wang
Ernest    | Gilbert
John      | Smith
Ricardo   | Browne
Susan     | Mao
Francis   | Johnson
```

ii)



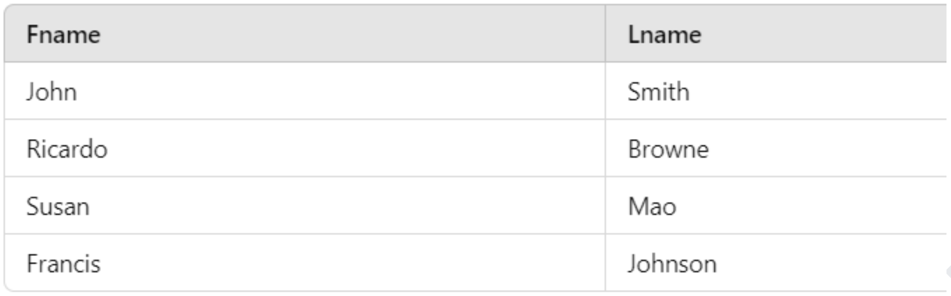
```
Fname     | Lname
--------------------
Ramesh    | Shah
```

iii)



Fname	Lname
Susan	Yao
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert

iv)



Fname	Lname
John	Smith
Ricardo	Browne
Susan	Mao
Francis	Johnson

b. Consider the following relational database schema and write the queries in relational algebra expressions:
EMP(Eno, Ename, Salary, Address, Phone, DNo)
DEPT(DNo, Dname, Dloc, MgrEno)
DEPENDENT(Eno, DepName, Relation, Dage)

1. **List all the employees who reside in 'Belagavi'.**
2. **List all the employees who earn between 30000 and 40000.**
3. **List all the employees who work for the 'Sales' department.**
4. **List the employees who have at least one dependent.**
5. **List the department names along with the names of the managers.**

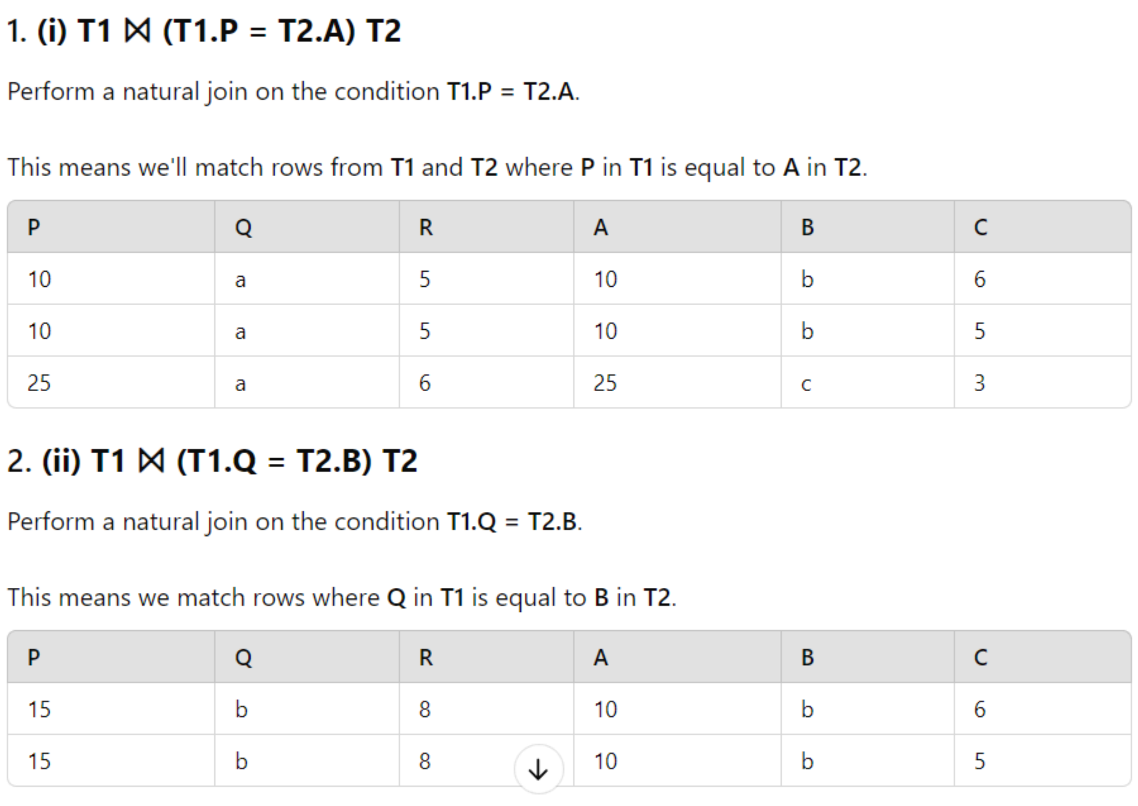
$$\sigma_{\text{Address}='Belagavi'}(\text{EMP})$$

$$\sigma_{30000\leq\text{Salary}\leq 40000}(\text{EMP})$$

$$\pi_{\text{Ename}}(\sigma_{\text{Dname}='Sales'}(\text{DEPT}) \bowtie \text{EMP})$$

$$\pi_{\text{Ename}}(\text{EMP} \bowtie \text{DEPENDENT})$$

$$\pi_{\text{Dname,Ename}}(\text{DEPT} \bowtie_{\text{MgrEno}=\text{Eno}} \text{EMP})$$



1. (i) T1 ⋈ (T1.P = T2.A) T2

Perform a natural join on the condition **T1.P = T2.A**.

This means we'll match rows from **T1** and **T2** where **P** in **T1** is equal to **A** in **T2**.

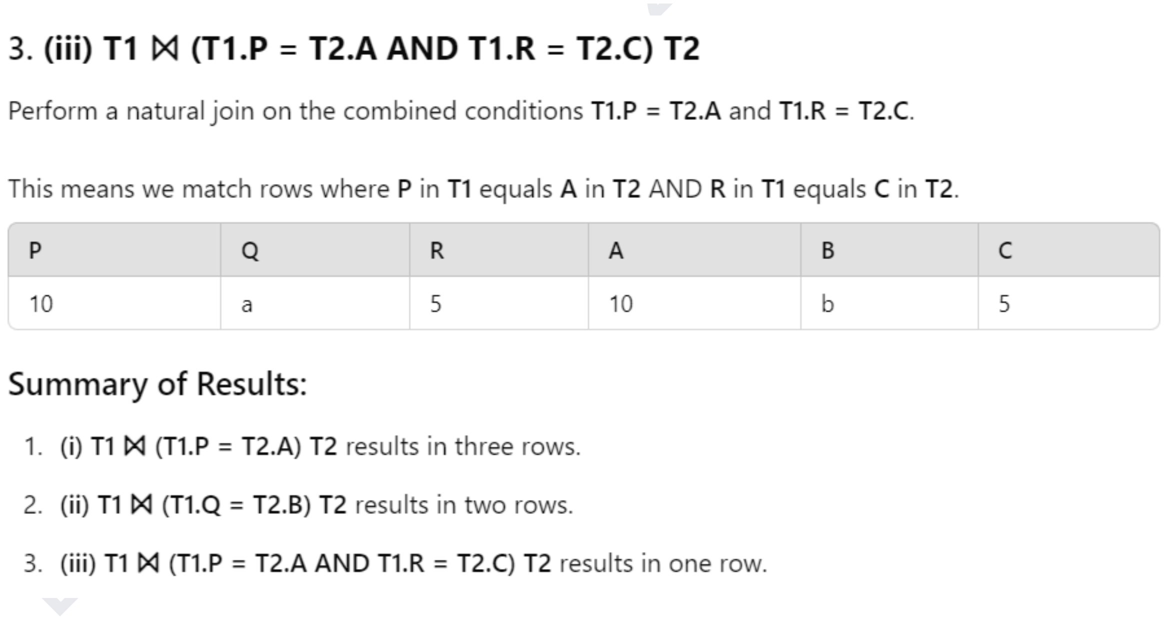
P	Q	R	A	B	C
10	a	5	10	b	6
10	a	5	10	b	5
25	a	6	25	c	3

2. (ii) T1 ⋈ (T1.Q = T2.B) T2

Perform a natural join on the condition **T1.Q = T2.B**.

This means we match rows where **Q** in **T1** is equal to **B** in **T2**.

P	Q	R	A	B	C
15	b	8	10	b	6
15	b	8	10	b	5



3. (iii) T1 ⋈ (T1.P = T2.A AND T1.R = T2.C) T2

Perform a natural join on the combined conditions **T1.P = T2.A** and **T1.R = T2.C**.

This means we match rows where **P** in **T1** equals **A** in **T2** AND **R** in **T1** equals **C** in **T2**.

P	Q	R	A	B	C
10	a	5	10	b	5

Summary of Results:

1. **(i) T1 ⋈ (T1.P = T2.A) T2** results in three rows.
2. **(ii) T1 ⋈ (T1.Q = T2.B) T2** results in two rows.
3. **(iii) T1 ⋈ (T1.P = T2.A AND T1.R = T2.C) T2** results in one row.

MODULE 3

Module 3:

Q5a. Informal design guidelines for relation schema design.

- Making sure that the semantics of the attributes is clear in the schema
- Reducing the redundant information in tuples
- Reducing the NULL values in tuples
- Disallowing the possibility of generating spurious tuples

14.1.1 Clear Semantics

- **Definition**: Relation attributes should have clear real-world meanings.
- **Importance**: A well-designed relation schema is easier to explain and interpret.
- **Example**: The EMPLOYEE schema clearly represents employee data, with attributes like name and department number, indicating an implicit relationship with the DEPARTMENT schema.
- **Guideline**: Design schemas to ensure each relation corresponds to a single entity or relationship to avoid semantic ambiguity.

14.1.2 Redundant Information and Update Anomalies

- **Redundancy**: Grouping attributes can lead to repeated information across tuples, wasting space.
- **Anomalies**:
    - **Insertion Anomalies**: Difficulty adding new data (e.g., a new employee) without needing to fill in all department details.
    - **Deletion Anomalies**: Removing a record can unintentionally lose essential data (e.g., deleting the last employee removes department info).
    - **Modification Anomalies**: Changes require updates in multiple places, risking inconsistencies.
- **Guideline**: Structure schemas to prevent these anomalies.

14.1.3 NULL Values in Tuples

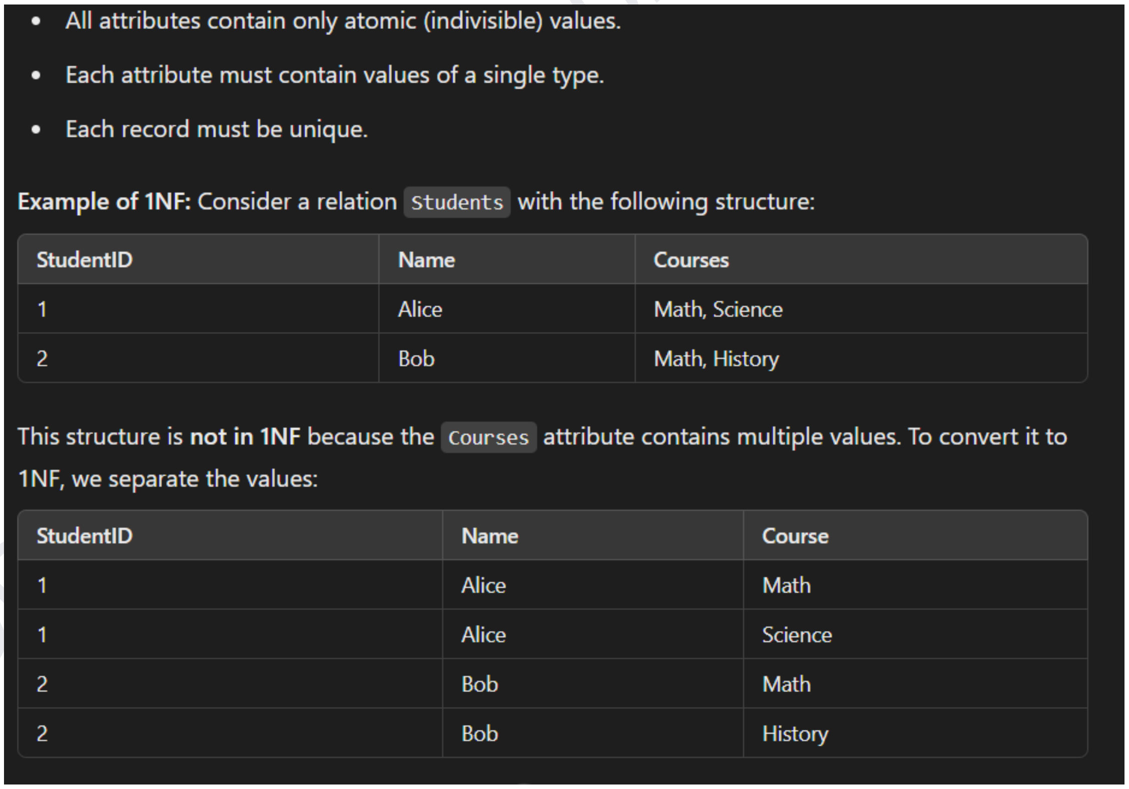
- **Problem**: Excessive NULL values can complicate understanding and operations.
- **Guideline**: Minimize attributes with frequent NULL values and consider separate relations for those attributes to maintain clarity.

14.1.4 Generation of Spurious Tuples

- **Concern**: Poorly designed relations can lead to unexpected tuples in results when joining or querying.
- **Example**: Combining multiple entities or relationships can create tuples that do not accurately represent the intended relationships.
- **Guideline**: Ensure design avoids configurations that could lead to spurious data.

Q5b.Explain 1NF, 2NF, and 3NF with examples.

- **1NF (First Normal Form)**: Ensures atomicity of values.
- **2NF (Second Normal Form)**: Ensures no partial dependency, every non-key attribute must depend on the whole primary key.
- **3NF (Third Normal Form)**: Ensures no transitive dependency, i.e., non-key attributes do not depend on other non-key attributes.



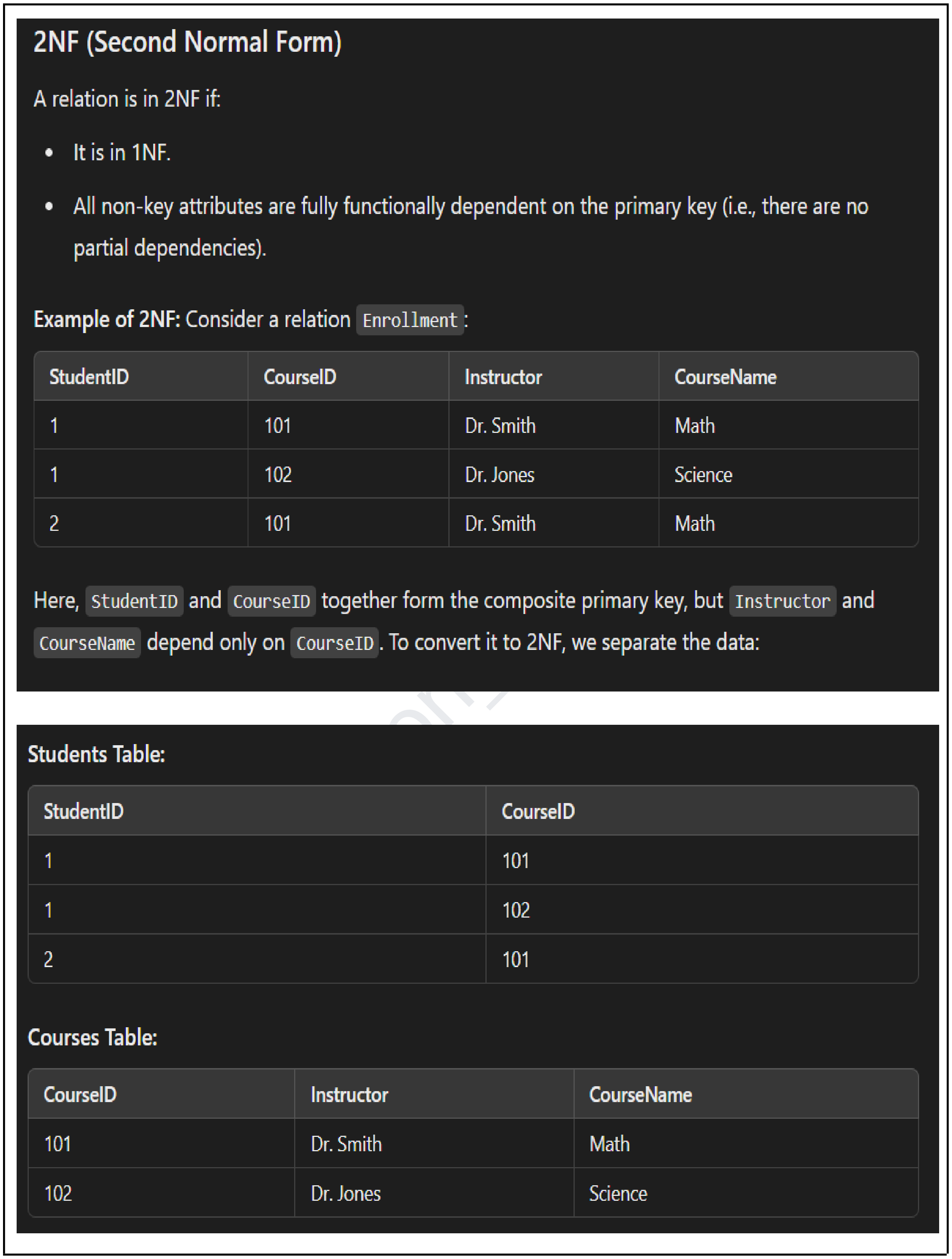
- All attributes contain only atomic (indivisible) values.
- Each attribute must contain values of a single type.
- Each record must be unique.

Example of 1NF: Consider a relation `Students` with the following structure:

StudentID	Name	Courses
1	Alice	Math, Science
2	Bob	Math, History

This structure is **not in 1NF** because the `Courses` attribute contains multiple values. To convert it to 1NF, we separate the values:

StudentID	Name	Course
1	Alice	Math
1	Alice	Science
2	Bob	Math
2	Bob	History



2NF (Second Normal Form)

A relation is in 2NF if:

- It is in 1NF.
- All non-key attributes are fully functionally dependent on the primary key (i.e., there are no partial dependencies).

Example of 2NF: Consider a relation `Enrollment`:

StudentID	CourseID	Instructor	CourseName
1	101	Dr. Smith	Math
1	102	Dr. Jones	Science
2	101	Dr. Smith	Math

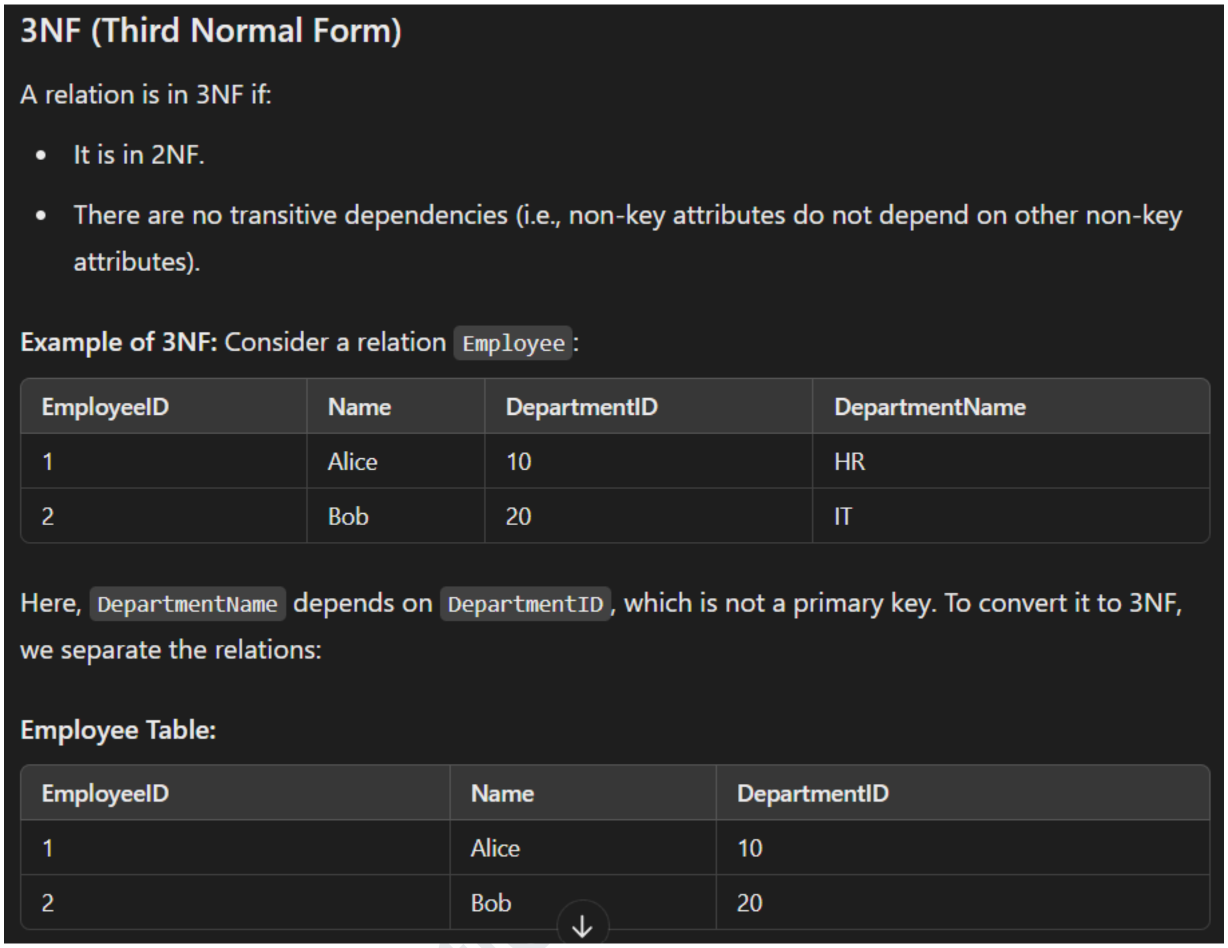
Here, `StudentID` and `CourseID` together form the composite primary key, but `Instructor` and `CourseName` depend only on `CourseID`. To convert it to 2NF, we separate the data:

Students Table:

StudentID	CourseID
1	101
1	102
2	101

Courses Table:

CourseID	Instructor	CourseName
101	Dr. Smith	Math
102	Dr. Jones	Science



3NF (Third Normal Form)

A relation is in 3NF if:

- It is in 2NF.
- There are no transitive dependencies (i.e., non-key attributes do not depend on other non-key attributes).

Example of 3NF: Consider a relation `Employee`:

EmployeeID	Name	DepartmentID	DepartmentName
1	Alice	10	HR
2	Bob	20	IT

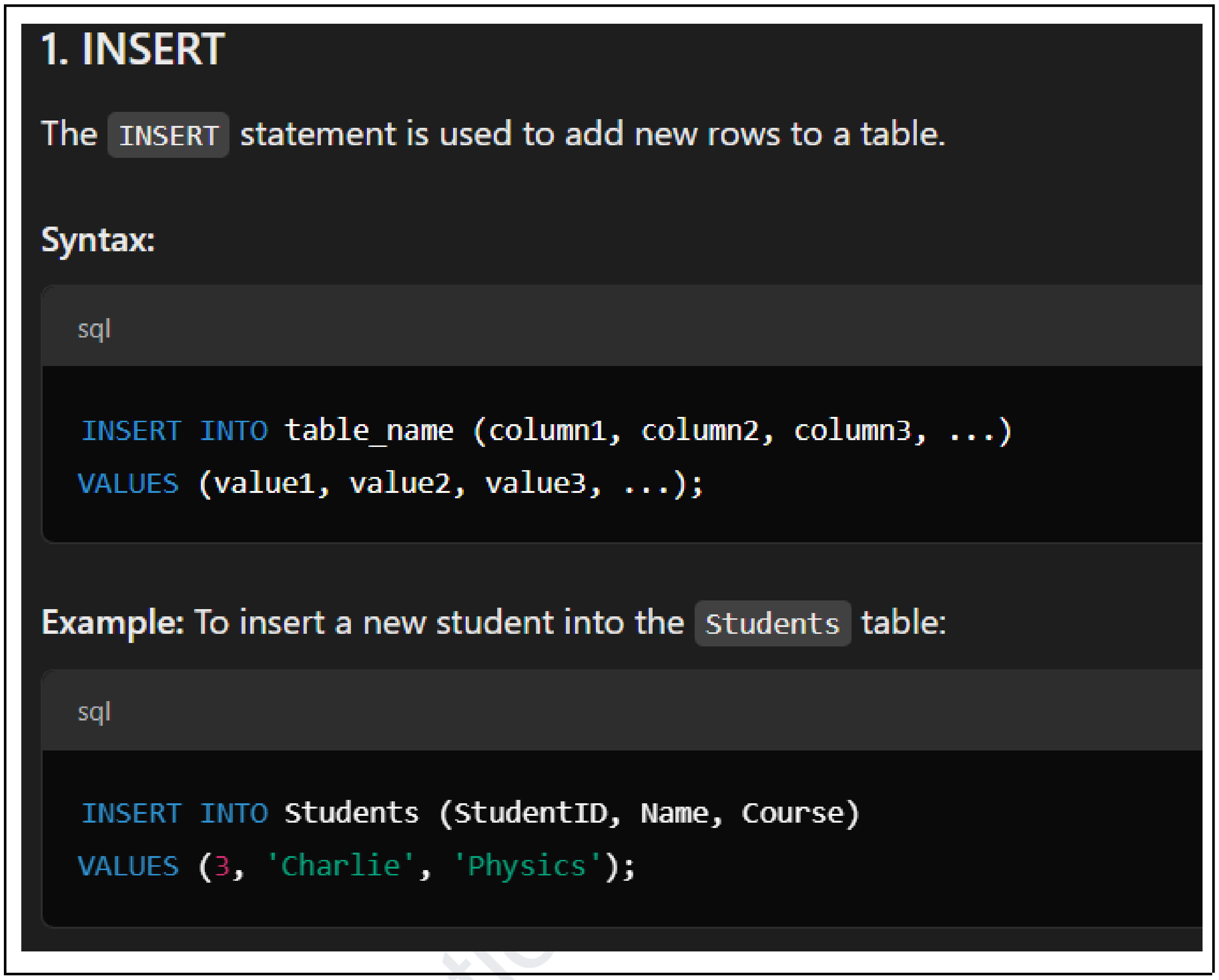
Here, `DepartmentName` depends on `DepartmentID`, which is not a primary key. To convert it to 3NF, we separate the relations:

Employee Table:

EmployeeID	Name	DepartmentID
1	Alice	10
2	Bob	20

Q5c. SQL Commands for INSERT, UPDATE, and DELETE.

- **INSERT**: INSERT INTO Employee VALUES ('John', 'Doe', 30000);
- **UPDATE**: UPDATE Employee SET Salary = 35000 WHERE Name = 'John Doe';
- **DELETE**: DELETE FROM Employee WHERE Name = 'John Doe';.


1. INSERT

The `INSERT` statement is used to add new rows to a table.

Syntax:

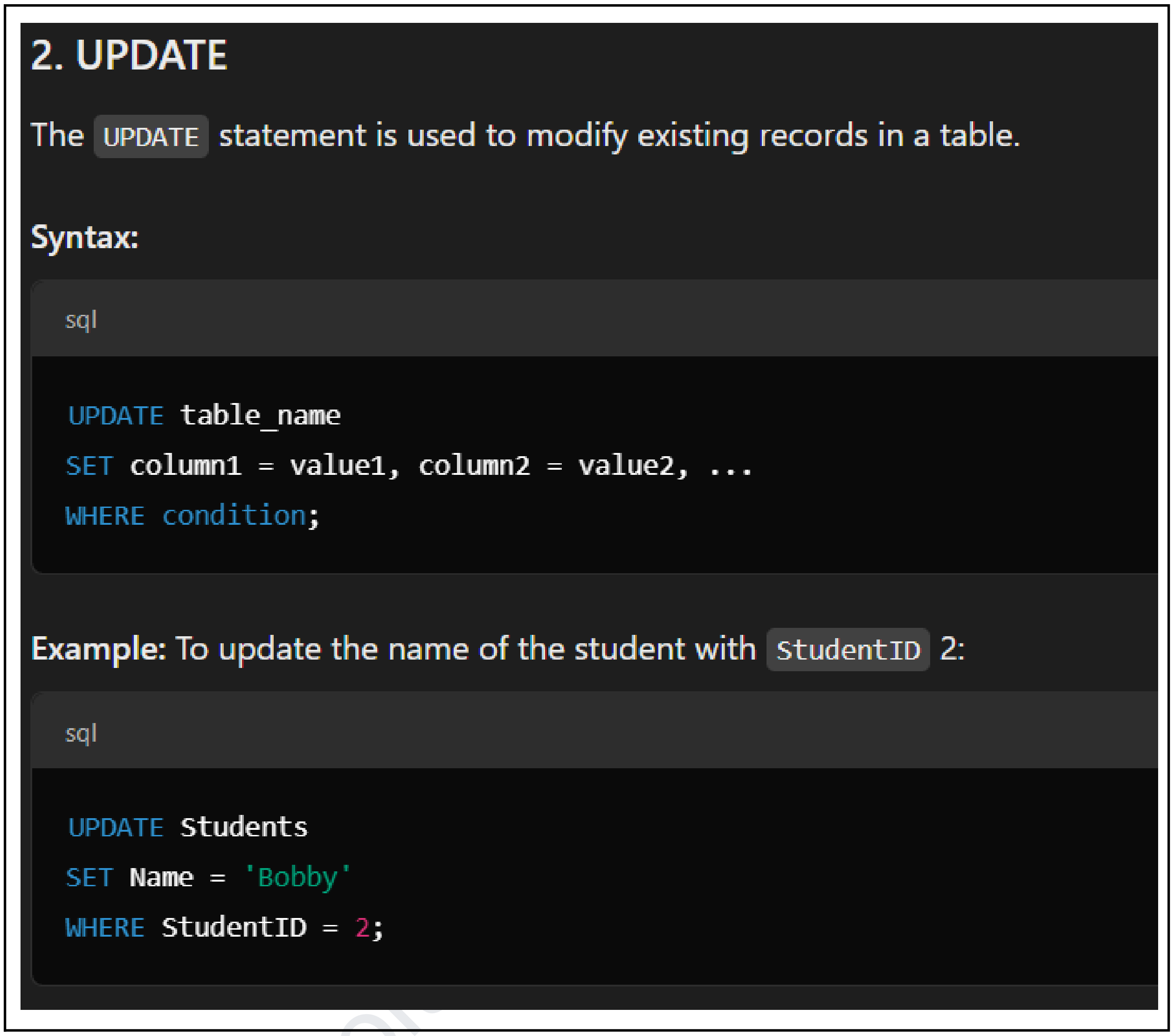
sql

```sql
INSERT INTO table_name (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...);
```

Example: To insert a new student into the `Students` table:

sql

```sql
INSERT INTO Students (StudentID, Name, Course)
VALUES (3, 'Charlie', 'Physics');
```


2. UPDATE

The `UPDATE` statement is used to modify existing records in a table.

Syntax:

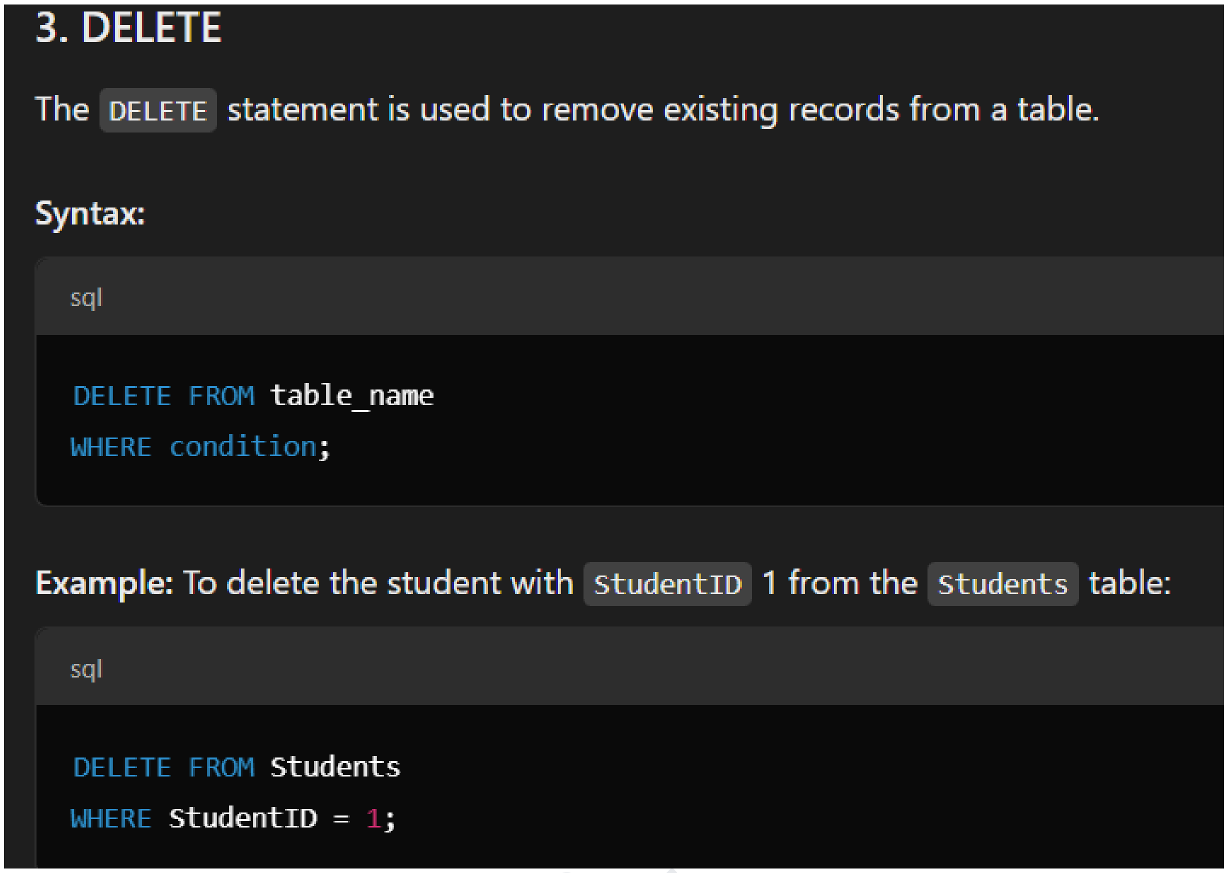
sql

```sql
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

Example: To update the name of the student with `StudentID` 2:

sql

```sql
UPDATE Students
SET Name = 'Bobby'
WHERE StudentID = 2;
```


3. DELETE

The `DELETE` statement is used to remove existing records from a table.

Syntax:

```sql
DELETE FROM table_name
WHERE condition;
```

Example: To delete the student with `StudentID` 1 from the `Students` table:

```sql
DELETE FROM Students
WHERE StudentID = 1;
```

6a). Discuss insertion, deletion, and modification anomalies. Why are they considered bad? Illustrate with examples.

In relational databases, anomalies occur when data is stored redundantly in a poorly designed schema, typically in a non-normalized table. These anomalies can be classified into three main types: insertion, deletion, and modification anomalies. They are considered bad because they compromise the integrity of the data and may result in inconsistencies or the loss of important information.

1. Insertion Anomaly:

An insertion anomaly occurs when it is impossible to insert data into the table without also inserting unrelated or redundant information. This happens in a non-normalized schema where certain columns depend on each other and require values even when the actual information is not available.

Example: Consider a table that stores employee and department information:

EmpID	EmpName	DeptID	DeptName
101	John	10	HR

If you want to add a new department (say "Finance") without assigning any employee to it, you cannot insert it into this table unless you also provide some employee information. This dependency creates an insertion anomaly, as you can't insert department data without inserting employee data.

---

2. Deletion Anomaly:

A deletion anomaly occurs when deleting data about one entity inadvertently causes loss of data about another entity. This is common in non-normalized tables where data is stored redundantly.

Example: Using the same table from above:

EmpID	EmpName	DeptID	DeptName
101	John	10	HR
102	Susan	20	Finance

If you delete the row for employee Susan, you also lose information about the Finance department (DeptID 20), even though that department may still exist.

---

3. Modification Anomaly:

A modification anomaly occurs when data is stored redundantly and needs to be updated in multiple places. If you update it in one place and forget to update it elsewhere, the database will have inconsistent data.

Example: If the department name HR changes to Human Resources, you would need to update all rows where DeptID is 10. If you forget to update some rows, the database will have inconsistent data, where DeptID 10 refers to both HR and

Human Resources.

EmpID	EmpName	DeptID	DeptName
101	John	10	HR
103	Mary	10	Human Resources

Why are these anomalies considered bad?

- Data Redundancy: Storing the same information multiple times leads to redundancy, which wastes storage space and makes the system harder to manage.
- Data Inconsistency: If the same information is stored in multiple places, there's a risk that it will become inconsistent over time due to modification anomalies.
- Difficulty in Data Management: Anomalies make it harder to insert, update, or delete records without affecting unrelated data. This increases the complexity of the database and the risk of errors.
- Loss of Information: Deletion anomalies can lead to the unintended loss of important data, compromising the integrity of the database.

Normalization as a Solution:

Normalization, particularly to 3NF (Third Normal Form) or higher, is used to eliminate these anomalies. By organizing the data into separate, well-defined tables with appropriate keys and relationships, insertion, deletion, and modification anomalies can be avoided, ensuring consistency and integrity of the data.

6b)Illustrate the following with suitable examples:

- (i) Datatypes in SQL
- (ii) Substring Pattern Matching in SQL

(i) SQL provides various data types to define the type of data that can be stored in a column of a table. Here are some common SQL datatypes:

INT: Stores integer values (e.g., 100, -45).

```
CREATE TABLE Employees (
  ID INT,
  Name VARCHAR(50),
  Age INT
);
```

-

VARCHAR(size): Stores variable-length string data (e.g., John, Database).

```
CREATE TABLE Students (
  StudentID INT,
  Name VARCHAR(100)
);
```

-

FLOAT: Stores floating-point numbers (e.g., 10.5, -2.75).

```
CREATE TABLE Products (
  ProductID INT,
  Price FLOAT
);
```

-

DATE: Stores date values in the format YYYY-MM-DD (e.g., 2023-09-20).

```
CREATE TABLE Orders (
  OrderID INT,
  OrderDate DATE
);
```

BOOLEAN: Stores true/false values.

```
CREATE TABLE Flags (
  FlagID INT,
  IsActive BOOLEAN
        );
```

(ii)

SQL provides the LIKE operator for pattern matching using wildcards, and it is commonly used to search for substrings in string data. The two main wildcards used with LIKE are:

- **%**: Represents zero or more characters.
- **_**: Represents a single character.

  Examples:

  Find names that start with 'A':

  ```
  SELECT Name
  FROM Students
  WHERE Name LIKE 'A%';
  ```

- This query retrieves all names that begin with the letter 'A'.

  Find names that end with 'son':

SELECT Name

FROM Employees

WHERE Name LIKE '%son';

- **Find names where the second letter is 'a'**:

  SELECT Name

  FROM Employees

  WHERE Name LIKE '_a%';

- **Find names that contain 'dat'**:

  SELECT Name

  FROM Employees

- WHERE Name LIKE '%dat%';

MODULE 4

Q7a. Consider the following relations:

- **Student**(Snum, Sname, Branch, level, age)
- **Class**(Cname, meet_at, room, fid)
- **Enrolled**(Snum, Cname)
- **Faculty**(fid, fname, deptid)

Write the following queries in SQL. No duplicates should be printed in any of the answers.

(i) Find the names of all Juniors (level = JR) who are enrolled in a class

taught by I. Teach.

```
SELECT DISTINCT S.Sname
FROM Student S, Enrolled E, Class C, Faculty F
WHERE S.Snum = E.Snum
AND E.Cname = C.Cname
AND C.fid = F.fid
AND S.level = 'JR'
AND F.fname = 'I. Teach';
```

This query joins the Student, Enrolled, Class, and Faculty tables to find the students who are at the junior level and enrolled in a class taught by a specific faculty member.

(ii) Find the names of all classes that either meet in room R128 or have five or more students enrolled.

```
SELECT DISTINCT C.Cname
FROM Class C
WHERE C.room = 'R128'
UNION
SELECT C.Cname
FROM Class C, Enrolled E
WHERE C.Cname = E.Cname
GROUP BY C.Cname
HAVING COUNT(E.Snum) >= 5;
```

This query uses a UNION to combine the results of two conditions: classes meeting in room R128 or classes with five or more students enrolled.

(iii) For all levels except JR, print the level and the average age of students for that level.

```
SELECT S.level, AVG(S.age) AS Avg_Age
FROM Student S
WHERE S.level <> 'JR'
GROUP BY S.level;
```

This query calculates the average age of students for each level except for juniors.

(iv) For each faculty member that has taught classes only in room R128, print the faculty member's name and the total number of classes he or she has taught.

```
SELECT F.fname, COUNT(C.Cname) AS Total_Classes
FROM Faculty F, Class C
WHERE F.fid = C.fid
AND C.room = 'R128'
GROUP BY F.fname
HAVING COUNT(DISTINCT C.room) = 1;
```

This query finds faculty members who have only taught classes in room R128 and returns their name and the total number of classes they have taught.

(v) Find the names of students not enrolled in any class.

```
SELECT S.Sname
FROM Student S
WHERE S.Snum NOT IN (SELECT E.Snum FROM Enrolled E);
```

This query retrieves the names of students who are not enrolled in any class by using a NOT IN subquery.

Q7b. What do you understand by correlated and nested queries in SQL? Explain with a suitable example.

- **Correlated Query**: A correlated query is a subquery that refers to a column from the outer query. It is executed repeatedly, once for each row selected by the outer query.

Example:

```
SELECT Sname, age
FROM Student S
WHERE age > (SELECT AVG(age) FROM Student WHERE Branch =
S.Branch);
```

Here, the inner query refers to the outer query's Branch, and for each student, the average age of students in that branch is calculated.

- **Nested Query**: A nested query is a query within another query, where the inner query is executed once and its result is passed to the outer query.

Example:

```
SELECT Sname, age
FROM Student
WHERE age > (SELECT AVG(age) FROM Student);
```

Here, the inner query calculates the average age once, and the result is used to compare against the age of each student.

Q7c. Discuss the ACID properties of a database transaction.

The **ACID** properties ensure that database transactions are processed reliably. They are:

1. **Atomicity**: Ensures that all operations within a transaction are completed. If any part of the transaction fails, the entire transaction is rolled back, and the database remains unchanged.
   Example: If a bank transfer transaction debits one account but fails to

credit another, the entire transaction is reversed.

2. **Consistency**: Ensures that a transaction brings the database from one valid state to another. The database must meet all the rules and constraints after the transaction.
   Example: If a transaction violates any constraints, such as a foreign key constraint, the database should be rolled back.
3. **Isolation**: Ensures that transactions are executed in isolation from one another. Intermediate results of a transaction are not visible to other transactions until the transaction is committed.
   Example: If two users try to update the same account balance simultaneously, isolation ensures that both transactions are handled in a serializable manner.
4. **Durability**: Ensures that once a transaction is committed, it remains so, even in the event of a system crash. The results of the transaction are permanently stored in the database.
   Example: Once a bank transfer transaction is committed, it is reflected in the database even if the system crashes immediately afterward.

Q8a. What are the views in SQL? Explain with examples.

Views in SQL are virtual tables based on the result of an SQL query. They do not store data themselves but provide a way to look at and manipulate the result of a query as if it were a table.

Example:

```
CREATE VIEW HighSalaryEmployees AS
SELECT EmpID, EmpName, Salary
FROM Employees
WHERE Salary > 50000;
```

In this example, a view called **HighSalaryEmployees** is created based on the result of a query that selects employees with a salary greater than 50,000. Once the view is created, you can query it as if it were a table:

```
SELECT * FROM HighSalaryEmployees;
```

Benefits of Views:

1. **Simplicity**: They simplify complex queries by encapsulating them in a view.
2. **Security**: You can restrict access to certain data by allowing users to query a view instead of the base table.
3. **Data Abstraction**: Views provide an abstraction layer over the database schema.

Q8b. In SQL, write the usage of GROUP BY and HAVING clauses with suitable examples.

- GROUP BY: The GROUP BY clause is used to group rows that have the same values in specified columns. It is often used with aggregate functions (COUNT, SUM, AVG, etc.).
- HAVING: The HAVING clause is used to filter the groups based on certain conditions, similar to WHERE but for groups.

Example:

```
SELECT DeptID, COUNT(EmpID) AS EmployeeCount

FROM Employees

GROUP BY DeptID

HAVING COUNT(EmpID) > 5;
```

- GROUP BY DeptID groups the rows by department.
- HAVING COUNT(EmpID) > 5 filters the groups to only those departments with more than 5 employees.

In this query, we count the number of employees in each department and return only the departments with more than 5 employees.

Q8c. Discuss the types of problems that may encounter with transactions that run concurrently.

When transactions run concurrently, several problems can arise, affecting the consistency and isolation of the database. These problems include:

1. Lost Update: Occurs when two transactions simultaneously update the same data. The update of one transaction may overwrite the other, leading to lost data.
   Example:
   - Transaction A reads a value of 100 from the database.
   - Transaction B also reads the value 100, increments it to 110, and writes it back.
   - Transaction A increments its original value (100) to 105 and writes it back, thus "losing" the update made by Transaction B.
2. Dirty Read: Occurs when a transaction reads data that has been written by another transaction but not yet committed. If the second transaction is rolled back, the first transaction has read invalid data.
   Example:
   - Transaction A updates a row, but before committing, Transaction B reads that updated data.
   - If Transaction A rolls back, the data read by Transaction B is invalid.
3. Non-Repeatable Read: Happens when a transaction reads the same data multiple times and gets different values due to modifications by other concurrent transactions.
   Example:
   - Transaction A reads a row.
   - Transaction B updates that row and commits.
   - Transaction A reads the same row again and finds different data.
4. Phantom Read: Occurs when a transaction reads a set of rows that satisfy a certain condition, but when it repeats the query, new rows satisfying the condition appear due to insertions by other transactions.
   Example:
   - Transaction A reads all rows where Salary > 50000.
   - Transaction B inserts a new row with Salary = 60000.

- When Transaction A re-executes the query, it gets different results due to the newly inserted row.

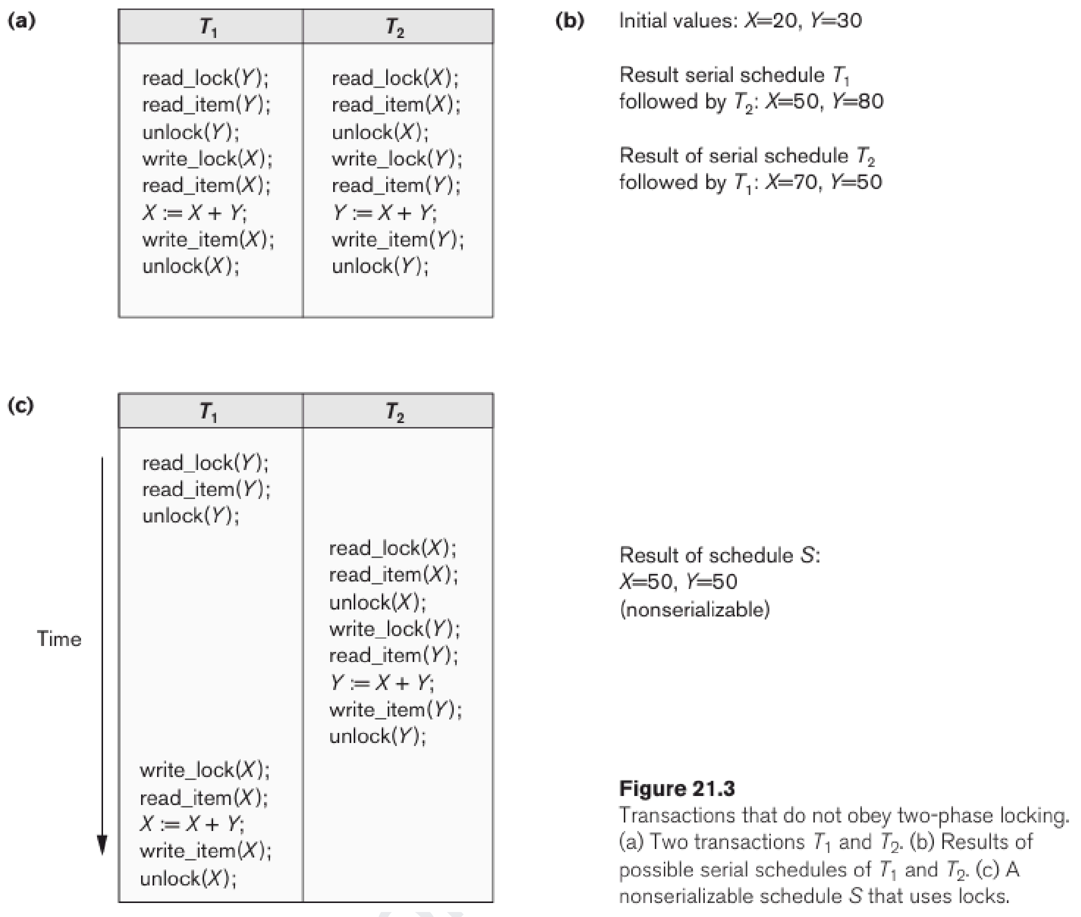
MODULE 5

a. What is the two-phase locking protocol? How does it guarantee serializability?

The two-phase locking (2PL) protocol is a concurrency control method used to ensure serializability of database transactions. A transaction is said to follow the two-phase locking protocol if all locking operations (acquiring locks) precede the first unlock operation (releasing locks). This divides the transaction into two phases:

- **Growing Phase**: The transaction acquires all required locks and cannot release any lock.
- **Shrinking Phase**: The transaction releases the locks and cannot acquire any more.

By enforcing that no new locks can be acquired after the release of any lock, 2PL guarantees that the transactions will be executed in a serializable manner, i.e., the result of concurrently running transactions will be the same as if they were executed sequentially



(a)

T_1	T_2
read_lock(Y);	read_lock(X);
read_item(Y);	read_item(X);
unlock(Y);	unlock(X);
write_lock(X);	write_lock(Y);
read_item(X);	read_item(Y);
$X := X + Y$;	$Y := X + Y$;
write_item(X);	write_item(Y);
unlock(X);	unlock(Y);

(b) Initial values: X=20, Y=30

Result serial schedule T_1 followed by T_2: X=50, Y=80

Result of serial schedule T_2 followed by T_1: X=70, Y=50

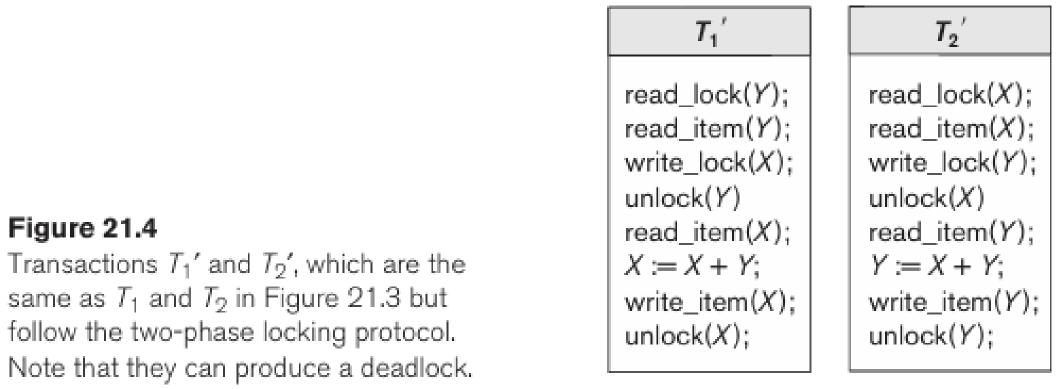
(c)

Time

T_1	T_2
read_lock(Y);	
read_item(Y);	
unlock(Y);	
	read_lock(X);
	read_item(X);
	unlock(X);
	write_lock(Y);
	read_item(Y);
	$Y := X + Y$;
	write_item(Y);
	unlock(Y);
write_lock(X);	
read_item(X);	
$X := X + Y$;	
write_item(X);	
unlock(X);	

Result of schedule S:
X=50, Y=50
(nonserializable)

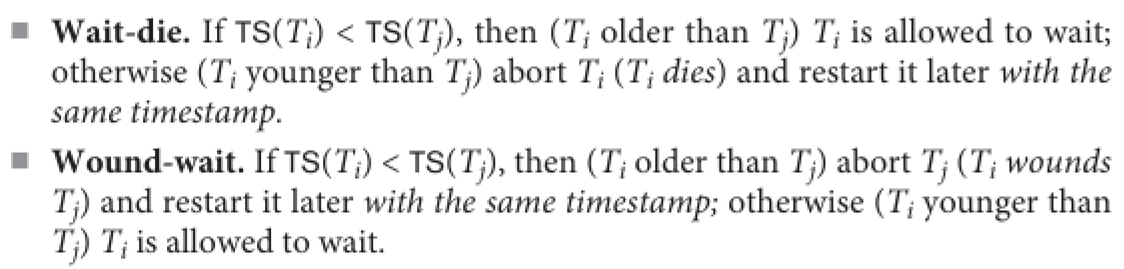
Figure 21.3
Transactions that do not obey two-phase locking. (a) Two transactions T_1 and T_2. (b) Results of possible serial schedules of T_1 and T_2. (c) A nonserializable schedule S that uses locks.



T_1'	T_2'
read_lock(Y);	read_lock(X);
read_item(Y);	read_item(X);
write_lock(X);	write_lock(Y);
unlock(Y)	unlock(X)
read_item(X);	read_item(Y);
$X := X + Y$;	$Y := X + Y$;
write_item(X);	write_item(Y);
unlock(X);	unlock(Y);

Figure 21.4
Transactions T_1' and T_2', which are the same as T_1 and T_2 in Figure 21.3 but follow the two-phase locking protocol. Note that they can produce a deadlock.

b. Describe the wait-die and wound-wait protocols for deadlock prevention.



- **Wait-die.** If $\mathrm{TS}(T_i) < \mathrm{TS}(T_j)$, then ($T_i$ older than T_j) T_i is allowed to wait; otherwise (T_i younger than T_j) abort T_i (T_i *dies*) and restart it later *with the same timestamp*.
- **Wound-wait.** If $\mathrm{TS}(T_i) < \mathrm{TS}(T_j)$, then ($T_i$ older than T_j) abort T_j (T_i *wounds* T_j) and restart it later *with the same timestamp;* otherwise (T_i younger than T_j) T_i is allowed to wait.

Wait-Die Protocol: In this protocol, if an older transaction requests a lock held by a younger transaction, the older transaction waits. If a younger transaction requests a lock held by an older transaction, the younger transaction is aborted ("dies") and restarted later with the same timestamp.
Wound-Wait Protocol: In this scheme, if an older transaction requests a lock held by a younger transaction, the younger transaction is aborted ("wounded") and restarted later. However, if a younger transaction requests a lock held by an older transaction, the younger transaction waits

c. List and explain the four major categories of NoSQL systems.

The four major categories of NoSQL systems are:

1. **Document-based NoSQL Systems**: Store data as documents in formats like JSON or BSON, accessible via document IDs.
2. **Key-Value Stores**: Simple data models that store data as key-value pairs for fast access.
3. **Column-Based (Wide Column) NoSQL Systems**: Partition data into columns or families of columns, optimized for queries across a few columns.
4. **Graph-Based NoSQL Systems**: Represent data in graphs, where nodes are entities, and edges represent relationships between entities. These systems are optimal for relationship-heavy queries.

Document-Based NoSQL:

```
{
  "_id": 1,
  "name": "John",
  "age": 30,
  "email": "john@example.com"
}
```

Key-Value Store:

```
{
  "John": {"age": 30, "email": "john@example.com"}
}
```

Column-Based Store:

```
| Row Key | Name  | Age | Email              |
|---------|-------|-----|--------------------|
| 1       | John  | 30  | john@example.com   |
```

Graph-Based (Neo4j):

```
(John)-[:FRIEND_OF]->(Jane)
```

a. What is Multiple Granularity Locking? How is it implemented using intention locks? Explain.

Multiple granularity locking allows a database system to lock data items at different levels of granularity (e.g., entire databases, files, pages, or records) efficiently. The system uses **intention locks** to indicate at higher levels of the hierarchy the intention to lock data at finer levels.

- **Intention Shared (IS)**: Indicates that a shared lock is requested on a lower-level data item.

- **Intention Exclusive (IX)**: Indicates that an exclusive lock is requested on a lower-level data item.
- **Shared-Intention Exclusive (SIX)**: Indicates that a shared lock is held on a higher-level item, with the intention of locking lower-level items exclusively

.

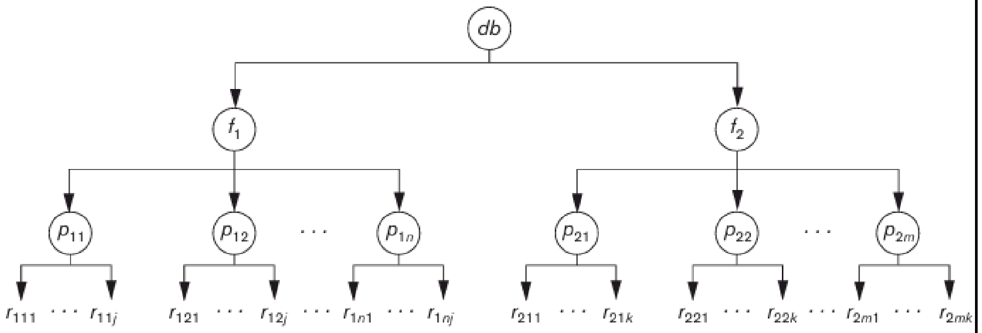
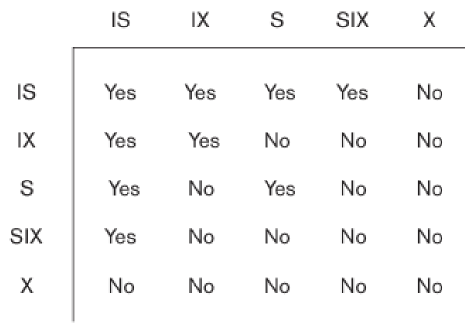


Figure 21.7
A granularity hierarchy for illustrating multiple granularity level locking.



	IS	IX	S	SIX	X
IS	Yes	Yes	Yes	Yes	No
IX	Yes	Yes	No	No	No
S	Yes	No	Yes	No	No
SIX	Yes	No	No	No	No
X	No	No	No	No	No

Figure 21.8
Lock compatibility matrix for multiple granularity locking.

The compatibility table of the three intention locks, and the actual shared and exclusive locks, is shown in Figure 21.8. In addition to the three types of intention locks, an appropriate locking protocol must be used. The multiple granularity locking (MGL) protocol consists of the following rules:

1. The lock compatibility (based on Figure 21.8) must be adhered to.

2. The root of the tree must be locked first, in any mode.

3. A node N can be locked by a transaction T in S or IS mode only if the parent node N is already locked by transaction T in either IS or IX mode.

4. A node N can be locked by a transaction T in X, IX, or SIX mode only if the parent of node N is already locked by transaction T in either IX or SIX mode.

5. A transaction T can lock a node only if it has not unlocked any node (to enforce the 2PL protocol).

6. A transaction T can unlock a node, N, only if none of the children of node N are currently locked by T

b. Discuss the following MongoDB CRUD operations with their formats:
Insert: Adds new documents into a collection using the command:

db.collection.insert(document)

Delete: Removes documents from a collection using the command:

db.collection.remove(query)

Read: Retrieves documents from a collection using the command:

db.collection.find(query)

Each operation corresponds to creating, reading, updating, or deleting data in the MongoDB database.
update, delete).
Documents can be created and inserted into their collections using the insert operation, whose format is:
db..insert() The parameters of the insert operation can include either a single document or an array of documents.
The delete operation is called remove, and the format is:
db..remove()
The documents to be removed from the collection are specified by a Boolean condition on some of the fields in the collection documents. There is also an update operation, which has a condition to select certain documents, and a $set clause to specify the update.

It is also possible to use the update operation to replace an existing document with another one but keep the same ObjectId. For read queries, the main command is called find, and the format is: db..find() General Boolean conditions can be specified as , and the documents in the collection that return true are selected for the query result. For a full discussion of the MongoDb CRUD operations, see the MongoDB online documentation in the chapter references.

c. Briefly discuss about Neo4j data model.

The Neo4j data model is based on a **graph structure** where:

- **Nodes**: Represent entities such as people, places, or objects, and can have properties.
- **Relationships**: Represent connections between nodes and are directed. They can also have properties.
- **Properties**: Key-value pairs associated with nodes or relationships.
- **Labels**: Group nodes into sets for querying. Nodes can have zero or multiple labels.

The graph-based model is flexible and well-suited for complex queries involving relationships, as seen in social networks, fraud detection, and recommendation systems

1. Labels and Properties

- **Labels** categorize nodes. For example, in Figure 24.4(a), the nodes are labeled as **EMPLOYEE, DEPARTMENT, PROJECT,** and **LOCATION**.
- **Properties** are attributes stored in curly brackets {...}. Nodes can have multiple labels, such as PERSON:EMPLOYEE:MANAGER. This is similar to an entity and its subclasses in entity-relationship models (ER and EER).

2. Relationships and Relationship Types

- Relationships define connections between nodes. For example, Figure 24.4(b) shows relationships like **WorksFor, Manager, LocatedIn,** and **WorksOn**.
- **WorksOn** has properties, such as Hours, which specify more details about the relationship.
- Relationships can be traversed in either direction, even if they are shown

with arrows (→) indicating a specific direction.

3. Paths

- A **path** is a sequence of nodes and relationships that form part of a query. It helps to navigate the graph and retrieve data matching a specific pattern.
- The concept is similar to path expressions in object databases (OQL) or XML query languages (XPath, XQuery).

4. Optional Schema

- Neo4j can function without a predefined schema. However, version 2.0 introduced schema-related functions like **indexes** and **constraints**. For instance, a key constraint can ensure that a certain property (e.g., EmpId) is unique within a label (e.g., EMPLOYEE).

5. Indexing and Node Identifiers

- When a node is created, Neo4j assigns it a unique system-defined identifier.
- Users can create indexes based on node properties for efficient retrieval. For example, EmpId could be used to index **EMPLOYEE** nodes, and Dno for **DEPARTMENT** nodes.