Summary

Q1: Theory and Practice Topics

1. Query Planning

- **Definition:** The process of deciding the most efficient way to execute a database query.
- **Key Concepts:**
  - **Cost-Based Optimization:** Database chooses the most efficient query execution plan based on statistics (e.g., index usage, table size).
  - **Steps in Query Planning:** Parsing, Planning, Statistics, Execution.
  - **Plan Types:** Sequential scans, index scans, joins (nested-loop, hash join, merge join).

    ▼ **Query Plans and Operations**

    ### 1. Sequential Scan

    - **What it does:** Scans all rows of a table one by one.
    - **When used:** No suitable index exists, or the query needs most/all rows.
    - **Efficiency:** Inefficient for large datasets but optimal for small tables or when retrieving most rows.

    ### 2. Index Scan

    - **What it does:** Uses an index to quickly locate specific rows based on conditions (e.g., WHERE clauses).
    - **When used:** A matching index exists, and the query targets a small subset of rows.
    - **Efficiency:** Faster than a sequential scan for selective queries but can be slower if fetching many rows.

    ### 3. Nested Loop Join

- **What it does:** For each row in the outer table, scans rows in the inner table to find matches.
- **When used:** Small datasets or when the join condition includes indexed columns.
- **Efficiency:** Simple but can be slow for large datasets.

4. Hash Join

- **What it does:** Creates a hash table of one table and probes it with rows from the other table.
- **When used:** Equality joins on large datasets.
- **Efficiency:** Faster than nested loops for large, unordered datasets, but requires memory for the hash table.

5. Merge Join

- **What it does:** Merges two sorted datasets by scanning them sequentially and matching rows.
- **When used:** Both tables are sorted on the join key.
- **Efficiency:** Efficient for large datasets with sorted input but requires sorting if data is unsorted.

6. Aggregation

- **What it does:** Groups rows and computes summary values (e.g., SUM, AVG, COUNT).
- **When used:** Queries with GROUP BY or aggregate functions.
- **Efficiency:** Performance depends on dataset size and indexes but can use parallel processing.

7. Sorting

- **What it does:** Orders rows based on specified columns (e.g., ORDER BY).
- **When used:** Queries that require ordered results.
- **Efficiency:** Sorting large datasets can be resource-intensive but is optimized with indexes.

▼ **Subplans and Subquery Scans**

1. Subplans

- **Definition:** A subplan is a part of the overall query execution plan that handles a **subquery**. It is created when the query planner decides to execute a subquery separately and integrate its results into the main query.
- **How they work:**
  - The main query references the subplan like a virtual table.
  - Subplans are used when the subquery cannot be directly optimized into a join or inline operation.
- **Example:**

```
SELECT Name FROM Employees
WHERE Salary > (SELECT AVG(Salary) FROM Employees);
```

  - The **subplan** will compute `AVG(Salary)` and provide the result to the outer query.

When Used:

- Scalar subqueries (returning a single value).
- Correlated subqueries (dependent on the outer query).
- Independent subqueries that cannot be flattened.

---

2. Subquery Scans

- **Definition:** A subquery scan occurs when the planner treats a subquery as a derived table and scans its output rows.
- **How they work:**
  - The subquery is treated as a temporary result set or table.
  - The outer query scans this result set to retrieve rows.
- **Example:**

```
SELECT e.Name, sub.AvgSalary
FROM Employees e
JOIN (SELECT DeptID, AVG(Salary) AS AvgSalary F
ROM Employees
GROUP BY DeptID) sub
ON e.DeptID = sub.DeptID;
```

- Here, `(SELECT DeptID, AVG(Salary)...)` is a subquery. The planner uses a **subquery scan** to fetch rows from this temporary result.

Key Differences

Feature	Subplans	Subquery Scans
Purpose	Executes subqueries separately.	Scans the output of a subquery as a derived table.
Execution	Can execute independently of the main query.	Treated like a table within the outer query.
Example Usage	Scalar and correlated subqueries.	Subqueries in `FROM` clause (common table expressions).

Optimization Notes

- Subplans and subquery scans can be inefficient if the subquery is executed multiple times.
- The query planner may flatten subqueries into joins or other optimized operations to improve performance where possible.

- **Improving Query Performance:**
  - Use indexes appropriately.
  - Optimize SQL queries (e.g., avoid SELECT *; use WHERE clause).
  - Partitioning and clustering tables.

2. Stored Procedures, Functions, Triggers

- **Definitions:**
  - **Stored Procedures:** Precompiled collections of SQL statements executed as a unit.
  - **Functions:** Similar to stored procedures but must return a value.
  - **Triggers:** Code executed automatically in response to specific database events (INSERT, UPDATE, DELETE).
- **Use Cases:**
  - Automating repetitive tasks (triggers for audit logs).
  - Encapsulating business logic in stored procedures.
  - Reusable calculations in functions.

3. SQL vs NoSQL

- **SQL:**
  - Relational, structured schemas.
  - Use cases: Banking, Business Analytics.
- **NoSQL:**
  - Flexible, schema-less design, document-based (e.g., MongoDB).
  - Use cases: Big data, real-time analytics, IoT.
- **When to Use:**
  - SQL for structured data, strong ACID compliance.
  - NoSQL for high scalability and unstructured data.

4. Creating Schema

- **Scenario-Based Schema Design:**
  - Analyze the use case (e.g., e-commerce site, inventory system).
  - Define entities, attributes, and relationships.
  - Choose between SQL (relational model) or NoSQL (document model) based on the scenario.

Q2: Scenario-Based Questions

1. Files and Schemas

- **Design Steps:**
  - Identify entities and their attributes.
  - Normalize the schema to reduce redundancy.
  - For NoSQL: Decide on collections, documents, and nesting levels.

2. MongoDB Basics

- **MongoDB Advantages:**
  - Horizontal scalability.
  - Schema-less data storage.
- **Key Differences with SQL:**
  - MongoDB stores JSON-like documents; SQL uses tables.
  - No Joins in MongoDB (aggregation pipelines instead).

3. Relational Systems and Normalization

- **1NF (First Normal Form):** No repeating groups or arrays.
- **2NF (Second Normal Form):** 1NF + no partial dependency (all non-key attributes depend on the whole primary key).
- **3NF (Third Normal Form):** 2NF + no transitive dependency.

4. ERD (Entity-Relationship Diagram)

- **Key Components:**
  - Entities (e.g., tables).
  - Relationships (1:1, 1:N, M:N).
  - Attributes (columns).

5. PostgreSQL

- Practice SQL queries:

```
CREATE TABLE Customers (
    ID SERIAL PRIMARY KEY,
    Name VARCHAR(100),
```

```
    Email VARCHAR(100) UNIQUE
);

INSERT INTO Customers (Name, Email) VALUES ('John Doe',
'john@example.com');
SELECT * FROM Customers;
```

Q3: Advanced Topics

1. Types of NoSQL Databases

- **Document Databases:** store data in flexible JSON-like documents, ideal for applications with complex and varying data structures.
- **Key-Value Stores:** key-value pair model, optimised for fast data retrieval and caching and session management.
- **Column-Family Stores:** organise data by columns for big data and analytical workloads that require efficient querying of large datasets.
- **Graph Databases:** focus on relationships between data, use nodes and edges to represent entities and connections, for social networks.

CAP Theorem: Consistency, Availability, Partition Tolerance. Impossible to achieve all three.

- NoSQL focus on AP (Availability + Partition Tolerance)
- **BASE Properties:**
    - **Basic Availability:** system available most of the time.
    - **Soft State:** data can be in intermediate state (unsynchronized changes).
    - **Eventual Consistency:** data will eventually become consistent.

2. Query Planning in Detail

- **Sequential Scans:** Reads the entire table row by row.
- **Sub Plans:** Break queries into smaller, reusable pieces.
- **Index Scans:** Use indexes to locate data faster.

3. Transactions

- **ACID Properties:**
  - **Atomicity:** All-or-nothing.
  - **Consistency:** Maintain integrity (from a valid state to another).
  - **Isolation:** Concurrent transactions don't interfere.
  - **Durability:** Changes persist even after crashes.
- **Isolation Levels:**
  - Read Uncommitted, Read Committed, Repeatable Read, Serializable.
  - Implement using `SET TRANSACTION ISOLATION LEVEL`.

4. Deadlocks

- **Definition:** Two transactions wait indefinitely for each other.
- **Prevention in PostgreSQL:**
  - Use timeouts or ordered access to resources.

5. MongoDB vs SQL Queries

- **SQL Insert:**

```
INSERT INTO Customers (Name, Email) VALUES ('John Doe',
'john@example.com');
```

- **MongoDB Insert:**

```
db.Customers.insertOne({ Name: "John Doe", Email: "john@
example.com" });
```

6. Subqueries vs Joins

- **Subquery:**
  - Nested query inside another.
  - Used when you need a temporary result that depends on another query.
  - **When to Use:** for aggregate or filtering data that's difficult to express as a join.

```
SELECT Name FROM Employees
WHERE Salary > (SELECT AVG(Salary) FROM Employees);
```

- **Join:**
  - Combines rows from two tables.
  - Combines data from multiple tables based on a condition.

```
SELECT A.Name, B.Department
FROM Employees A
JOIN Departments B ON A.DeptID = B.ID;
```

7. Query Refinement

- **Methods:**
  - Add indexes.
  - Use LIMIT to restrict rows.
  - Optimize WHERE clauses (e.g., avoid LIKE '%abc').

Procedures, Functions, Triggers and Cursors

1. Procedure

- A procedure is a block of SQL code stored on the database server and executed as a unit.
- **Syntax:**

```
CREATE OR REPLACE PROCEDURE procedure_name (param1 datatyp
e, param2 datatype)
LANGUAGE plpgsql AS $$
BEGIN
    -- Procedure logic here
END;
$$;
```

- **Example:**

```
CREATE OR REPLACE PROCEDURE add_employee(p_name VARCHAR, p_
salary INT)
LANGUAGE plpgsql AS $$
BEGIN
    INSERT INTO employees (name, salary) VALUES (p_name, p_
salary);
END;
$$;
```

2. Trigger

- A trigger is code that runs automatically when an event (INSERT, UPDATE, DELETE) occurs on a table.
- **Syntax:**

```
CREATE OR REPLACE FUNCTION trigger_function_name()
RETURNS TRIGGER AS $$
BEGIN
    -- Trigger logic here
    RETURN NEW; -- or OLD for DELETE
$$ LANGUAGE pgplsql;

CREATE TRIGGER trigger_name
AFTER INSERT ON table_name
FOR EACH ROW
EXECUTE FUNCTION trigger_function_name();
```

- **Example:**

```
CREATE OR REPLACE FUNCTION log_insert()
RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO auditLog (tableName, operation) VALUES ('Em
ployees', 'INSERT');
    RETURN NEW;
$$ LANGUAGE pgplsql;
```

```
CREATE TRIGGER after_employee_insert
AFTER INSERT ON Employees
FOR EACH ROW
EXECUTE FUNCTION log_insert();
```

3. Function

- A function is similar to a procedure but must return a value.
- **Syntax:**

```
CREATE OR REPLACE FUNCTION function_name (param1 datatype)
RETURNS return_datatype AS $$
LANGUAGE plpgsql AS $$
BEGIN
    -- Function logic here
END;
$$ LANGUAGE plpgsql;
```

- **Example:**

```
CREATE OR REPLACE FUNCTION calculate_bonus (salary INT)
RETURNS INT AS $$
BEGIN
    RETURN salary * 0.1;
END;
$$ LANGUAGE plpgsql;
```

4. Cursor

- A cursor allows row-by-row processing of query results.
- **Syntax:**

```
-- 1. Declare cursor: define cursor and the query to execut
e
DECLARE cursor_name CURSOR FOR query;
-- 2. Open cursor: automatically done after declaration
```

```
-- 3. Fetch data from cursor: retrieve row or set of rows
FETCH cursor_name INTO variable;
-- 4. Close cursor: release cursor and free resources
CLOSE cursor_name;
```

Transactions

- **Definition:** a transaction groups multiple SQL operations into a single logical unit. It is all-or-nothing: either all changes are committed or none are applied.
- **Syntax:**

```
BEGIN;
    -- SQL statements
COMMIT; -- or ROLLBACK;
```

- **Example:**

```
BEGIN;
    INSERT INTO Orders (CustomerID, ProductID, Quantity) VA
LUES (1, 101, 3);
    UPDATE Inventory SET Stock = Stokc - 3 WHERE ProductID
= 101;
COMMIT;
```

Deadlocks and Deadlock Levels

Deadlocks

- **Definition:** a deadlock occurs when two or more transactions are waiting for each other to release locks, preventing them from proceeding.
- **Example Scenario:**
    1. Transaction A locks Table1 and waits for Table2.
    2. Transaction B locks Table2 and waits for Table1.

How to Prevent Deadlocks:

- Always access tables in the same order.
- Minimize transaction scope.
- Set timeouts.

Isolation Levels

- **Definition:** control the visibility of changes made by one transaction to other concurrent transactions.
- **Levels (from least to most strict):**
    1. **Read Uncommitted:** dirty reads are allowed (seeing uncommitted changes).
    2. **Read Committed:** only committed data is visible.
    3. **Repeatable Read:** prevents non-repeatable reads (two equal reads obtain different results) but allows phantom reads (new rows added by other transactions).
    4. **Serializable:** full isolation, acts like transactions are executed sequentially.

NoSQL Code Examples

Insert Document

```
db.Employees.insertOne({
    Name: "John Doe",
    Salary: 50000,
    Department: "HR"
});
```

Find Documents

```
db.Employees.find({ Salary: { $gt: 40000 } });
```

Update Document

```
db.Employees.updateOne(
    { Name: "John Doe" },
    { $set: { Salary: 55000 } }
);
```

Delete Document

```
db.Employees.deleteOne({ Name: "John Doe" });
```

Aggregation Example

```
db.Employees.aggregate([
    { $group: { _id: "$Department", AvgSalary: { $avg: "$Sa
lary" } } }
]);
```

Stored Procedure

A stored procedure is a named set of SQL and procedural logic that can be executed explicitly by calling it.

- Must be invoked using the CALL statement.
- Does not necessarily return a value.
- Used for performing operations like updating multiple tables, managing transactions, or executing complex business logic.
- Can contain transaction control commands (COMMIT, ROLLBACK).
- Can have side effects like modifying data.

```
CREATE OR REPLACE PROCEDURE update_inventory(stock_id INT,
quantity INT)
LANGUAGE plpgsql AS $$
BEGIN
```

```
    UPDATE inventory
    SET stock = stock + quantity
    WHERE inventory_id = stock_id;

END;
$$;

-- To execute the stored procedure
CALL update_inventory(1, 10);
```

Stored Function

A function is a set of SQL and procedural code that performs a specific task ad returns a value.

- Invoked with SQL statements (SELECT, INSERT, UPDATE).
- Must return a value.
- Used for calculations, data transformations or returning query results.
- Functions cannot contain transaction control commands (COMMIT, ROLLBACK).
- Operations without side effects, but can update data in some cases.

```
CREATE OR REPLACE FUNCTION calculate_square(num INT)
RETURNS INT AS $$
BEGIN

    RETURN num * num;

END;
$$ LANGUAGE plpgsql;

-- Use the function
SELECT calculate_square(5);
```

Trigger

A trigger is an automatic mechanism that executes a function in response to certain database events (INSERT, UPDATE, DELETE) on a table.

- Automatically executed before or after specified database events.
- Does not return a value explicitly to the user but can return NEW or OLD records internally for table operations.
- Used for enforcing business rules, auditing changes or automating data updates.
- Operates within the transaction of the operation that fires the trigger.
- Can modify the data being acted upon.