1. Concepts of Databases, Tables, and Schemas

1.1 Definition and Basic Concept

- **Database:**
  A structured collection of data organized for efficient storage, retrieval, and management.
  - Example: MySQL, PostgreSQL, Oracle DB.
- **Table:**
  A table is a collection of rows and columns in a database where data is stored in a structured manner.
  - Row = Record, Column = Field/Attribute.
- **Schema:**
  The logical structure of a database, defining tables, columns, relationships, views, and constraints.

1.2 Purpose and Need

- Centralized storage and retrieval of data.
- Enforce data integrity and relationships.
- Facilitate multi-user access.
- Organize data efficiently for queries.

1.3 Architecture and Components

Database Architecture Layers:

```
+---------------------------+
|      Application Layer    | <-- Queries from users/applications
+---------------------------+
|      Database Engine      | <-- SQL parser, Query optimizer
+---------------------------+
| Storage Layer / Data Files | <-- Physical storage
+---------------------------+
```

Components:

1. **Tables** – Store data.
2. **Views** – Virtual tables for abstraction.
3. **Indexes** – Improve query speed.
4. **Constraints** – Ensure data integrity (PRIMARY KEY, FOREIGN KEY, UNIQUE).
5. **Stored Procedures / Triggers** – Automate tasks.

1.4 Working / Workflow

1. User sends SQL commands (via CLI or app).
2. Query parser interprets commands.
3. Optimizer selects the best execution plan.
4. Engine executes commands, interacting with storage files.
5. Result returned to the user.

1.5 Types and Classifications

- **Databases by model:**
  - Relational (RDBMS) – MySQL, PostgreSQL
  - NoSQL – MongoDB, Cassandra
  - NewSQL – CockroachDB
- **Tables:**
  - Base tables, temporary tables, partitioned tables
- **Schemas:**
  - Logical schema, physical schema

1.6 Features

- Structured organization
- Data integrity and constraints
- Concurrency control
- Backup and recovery

1.7 Advantages and Disadvantages

Advantages	Disadvantages
Efficient data management	Requires learning SQL
Multi-user support	Maintenance overhead
Data integrity	Can be resource-heavy
Query optimization	Security risks if misconfigured

1.8 Real-world Use Cases

- Bank transaction systems
- Inventory management systems
- Hospital patient record management

1.9 Important Exam Points

- RDBMS vs NoSQL
- Schema definition vs table definition
- Keys and constraints

---

2. Syntax and Examples for Creating Databases

2.1 Definition and Concept

- Creating a database means defining a new container to hold tables and related objects.

2.2 Purpose

- Segregates data for applications, users, or departments.
- Organizes data logically.

2.3 Syntax

```
CREATE DATABASE database_name;
```

Example:

```
CREATE DATABASE CollegeDB;
```

2.4 Features

- Optional: Character set and collation

```
CREATE DATABASE CollegeDB CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci;
```

2.5 Advantages

- Easy separation of datasets
- Simplifies permissions management

2.6 Important Exam Points

- `CREATE DATABASE`
- Optional: `IF NOT EXISTS` to avoid errors

---

3. Best Practices for Organizing and Naming Databases

3.1 Purpose

- Ensures clarity, maintainability, and avoids conflicts.

3.2 Guidelines

1. Use **lowercase letters** and underscores: `college_db`.
2. Avoid special characters or spaces.
3. Use descriptive names: `student_records` instead of `db1`.
4. Consistency in naming conventions across the project.
5. Separate databases for **different modules** of an application.

3.3 Advantages

- Easier collaboration and maintenance
- Avoids naming collisions

3.4 Exam Keywords

- Naming conventions
- Modularity
- Logical organization

---

4. Syntax and Examples for Dropping Databases

4.1 Definition

- Dropping a database deletes the database and all its objects permanently.

4.2 Syntax

```
DROP DATABASE database_name;
```

Example:

```
DROP DATABASE CollegeDB;
```

- Optional safety check:

```
DROP DATABASE IF EXISTS CollegeDB;
```

4.3 Advantages and Disadvantages

Advantages	Disadvantages
Removes unused databases	Permanent deletion
Frees storage	Data loss if not backed up

4.4 Exam Keywords

- `DROP DATABASE`
- `IF EXISTS`

---

5. Create a MySQL User

5.1 Purpose

- Manage access and permissions.
- Enhance security.

5.2 Syntax

```
CREATE USER 'username'@'host' IDENTIFIED BY 'password';
```

Example:

```
CREATE USER 'pankaj'@'localhost' IDENTIFIED BY 'Pass123!';
```

5.3 Granting Privileges

```
GRANT ALL PRIVILEGES ON CollegeDB.* TO 'pankaj'@'localhost';
FLUSH PRIVILEGES;
```

5.4 Advantages

- Controlled access
- Role-based security
- Better auditing

5.5 Exam Keywords

- `CREATE USER`
- `GRANT`, `REVOKE`
- `FLUSH PRIVILEGES`

---

6. Create Database, Add Tables, and Insert Data

6.1 Workflow

1. Create a database.
2. Create tables with columns and data types.
3. Insert records using `INSERT INTO`.

6.2 Example

```
-- Step 1: Create Database
CREATE DATABASE CollegeDB;

-- Step 2: Use Database
USE CollegeDB;

-- Step 3: Create Table
CREATE TABLE Students (
  student_id INT PRIMARY KEY,
  name VARCHAR(50),
  course VARCHAR(50),
  age INT
);

-- Step 4: Insert Data
INSERT INTO Students (student_id, name, course, age)
VALUES (1, 'Alice', 'CS', 21),
       (2, 'Bob', 'IT', 22);
```

6.3 Exam Keywords

- `CREATE TABLE`, `INSERT INTO`
- Data types: `INT`, `VARCHAR`, `DATE`, `FLOAT`

---

7. Delete / Modify a Table or Database

7.1 Drop a Table

```
DROP TABLE table_name;
```

Example:

```
DROP TABLE Students;
```

7.2 Alter a Table

- Add a column:

```
ALTER TABLE Students ADD COLUMN email VARCHAR(50);
```

- Modify column:

```
ALTER TABLE Students MODIFY COLUMN age SMALLINT;
```

- Drop column:

```
ALTER TABLE Students DROP COLUMN email;
```

7.3 Drop Database

```
DROP DATABASE CollegeDB;
```

7.4 Advantages

- Modify structure without losing all data
- Clean removal when obsolete

7.5 Exam Keywords

- `ALTER TABLE`, `DROP TABLE`, `DROP DATABASE`

Comparison Table: CREATE vs DROP

Operation	Purpose	Syntax	Permanent
CREATE DATABASE	Create new database	`CREATE DATABASE dbname;`	Yes
DROP DATABASE	Delete database	`DROP DATABASE dbname;`	Yes
CREATE TABLE	Create new table	`CREATE TABLE tbl(...);`	Yes
DROP TABLE	Delete table	`DROP TABLE tbl;`	Yes

☑ Important Exam Tips and Keywords

- SQL commands: `CREATE`, `DROP`, `ALTER`, `INSERT`

- `PRIMARY KEY`, `FOREIGN KEY`, `UNIQUE`, `INDEX`
- Schema vs database
- Data types and constraints
- Best practices in naming
- User creation and privileges
- Workflow: database → table → insert data → modify/delete

1. Writing SELECT Statements to Fetch Data

1.1 Definition and Basic Concept

- **SELECT statement:** SQL command used to retrieve data from one or more tables in a database.
- Basic syntax:

```
SELECT column1, column2, ...
FROM table_name;
```

1.2 Purpose and Need

- To query and fetch relevant information from a database.
- Enables analysis, reporting, and decision-making.

1.3 Architecture / Components

```
+--------------------+
|   Query Processor  | <-- Parses SQL query
+--------------------+
|   Optimizer        | <-- Selects efficient execution plan
+--------------------+
|   Storage Engine   | <-- Fetches data from tables/files
+--------------------+
|   Result Set       | <-- Returns data to user/application
+--------------------+
```

Components in SQL query:

- `SELECT` → Columns to retrieve
- `FROM` → Table(s) to query
- Optional clauses: `WHERE`, `ORDER BY`, `GROUP BY`, etc.

1.4 Working / Workflow

1. User executes `SELECT` query.
2. Query parser checks syntax.

3. Optimizer generates the best execution plan.
4. Storage engine retrieves data from tables.
5. Result set returned to user.

1.5 Types and Classifications

- **Basic SELECT:** Fetch all rows

```
SELECT * FROM Students;
```

- **Column-specific SELECT:** Fetch specific columns

```
SELECT name, age FROM Students;
```

- **DISTINCT SELECT:** Fetch unique values

```
SELECT DISTINCT course FROM Students;
```

1.6 Features

- Fetch specific columns
- Retrieve all or filtered data
- Works with single or multiple tables

1.7 Advantages

- Quick retrieval of data
- Supports analytical operations
- Flexible selection of columns

1.8 Disadvantages

- Requires knowledge of table structure
- Large tables may result in performance overhead

1.9 Real-world Use Cases

- Retrieve all student records from a college database
- Fetch product details for an e-commerce website

1.10 Exam Keywords

- `SELECT`, `DISTINCT`, `*`, column selection
- Query execution steps

---

2. Using WHERE Clause for Filtering Data

2.1 Definition

- **WHERE clause:** Used to filter records based on specific conditions.

2.2 Purpose

- Extract only relevant rows from a table.
- Reduce unnecessary data retrieval.

2.3 Syntax

```
SELECT column1, column2
FROM table_name
WHERE condition;
```

Example:

```
SELECT name, course FROM Students
WHERE age > 21;
```

2.4 Types of Conditions

1. Comparison operators: `=`, `!=`, `<`, `>`, `<=`, `>=`
2. Logical operators: `AND`, `OR`, `NOT`
3. Pattern matching: `LIKE`
4. Range: `BETWEEN ... AND ...`
5. Set membership: `IN`

2.5 Workflow

1. SQL engine scans table rows.
2. Evaluates each row against the WHERE condition.
3. Only rows that satisfy the condition are returned.

2.6 Features

- Supports multiple conditions
- Works with all SQL data types
- Can filter numeric, string, and date data

2.7 Advantages

- Efficient data retrieval

- Reduces result set size
- Supports complex conditions

2.8 Disadvantages

- Complex conditions may reduce readability
- Incorrect conditions may return wrong results

2.9 Example Use Case

- Fetch employees earning above a certain salary
- Retrieve students enrolled in a specific course

2.10 Exam Keywords

- `WHERE`, `AND`, `OR`, `NOT`, `LIKE`, `BETWEEN`, `IN`

---

3. Sorting Data Using ORDER BY

3.1 Definition

- **ORDER BY clause:** Sorts retrieved data in ascending (`ASC`) or descending (`DESC`) order.

3.2 Purpose

- Organize query results for reporting or analysis.

3.3 Syntax

```
SELECT column1, column2
FROM table_name
ORDER BY column1 [ASC|DESC];
```

Example:

```
SELECT name, age FROM Students
ORDER BY age DESC;
```

3.4 Workflow

1. SQL engine fetches the result set.
2. Sorting algorithm (quick sort, merge sort) applied on specified column(s).
3. Sorted data returned to user.

3.5 Types

- Single-column sorting
- Multi-column sorting:

```
SELECT name, age, course FROM Students
ORDER BY course ASC, age DESC;
```

3.6 Features

- Supports ascending/descending order
- Multiple columns
- Works with numbers, text, and dates

3.7 Advantages

- Easy to analyze sorted data
- Useful for ranking and reporting

3.8 Disadvantages

- Sorting large datasets may impact performance

3.9 Exam Keywords

- `ORDER BY`, `ASC`, `DESC`, multi-column sorting

---

4. Basic JOIN Operations to Combine Data from Multiple Tables

4.1 Definition

- **JOIN:** Combines rows from two or more tables based on a related column.

4.2 Purpose

- Retrieve related data from multiple tables without redundancy.

4.3 Types / Classifications

JOIN Type	Description
INNER JOIN	Returns matching rows in both tables
LEFT JOIN	Returns all rows from left table + matching rows from right table
RIGHT JOIN	Returns all rows from right table + matching rows from left table
FULL OUTER JOIN	Returns all rows from both tables, NULL if no match
CROSS JOIN	Cartesian product of two tables

4.4 Syntax

```
SELECT Students.name, Courses.course_name
FROM Students
INNER JOIN Courses
ON Students.course_id = Courses.course_id;
```

4.5 Workflow

1. SQL engine identifies the JOIN type.
2. Matches rows using the ON condition.
3. Combines results and returns final dataset.

4.6 Features

- Combines data logically
- Supports multiple JOIN conditions
- Works with filtering (`WHERE`) and sorting (`ORDER BY`)

4.7 Advantages

- Reduces data redundancy
- Supports complex relational queries
- Efficient data retrieval across tables

4.8 Disadvantages

- Complex joins may reduce performance
- Requires proper indexing for large tables

4.9 Real-world Use Cases

- Fetch student names along with enrolled course names
- Display orders along with customer information

4.10 Exam Keywords

- `INNER JOIN`, `LEFT JOIN`, `RIGHT JOIN`, `FULL JOIN`

- Join conditions, ON clause

---

5. Using Aggregate Functions (SUM, COUNT, AVG)

5.1 Definition

- Aggregate functions perform calculations on a set of values and return a single value.

5.2 Purpose

- Summarize large datasets for analysis.

5.3 Syntax

```
SELECT COUNT(student_id) AS TotalStudents,
       SUM(age) AS TotalAge,
       AVG(age) AS AverageAge
FROM Students;
```

5.4 Workflow

1. SQL engine groups rows if `GROUP BY` is used.
2. Performs aggregate function calculation.
3. Returns a single summarized value or group-wise summary.

5.5 Types

- `SUM()` → Total sum
- `COUNT()` → Number of rows
- `AVG()` → Average value
- `MAX()`, `MIN()` → Maximum and minimum values

5.6 Advantages

- Efficient summarization
- Supports analytics and reporting

5.7 Disadvantages

- Cannot return individual row data when aggregated

5.8 Exam Keywords

- Aggregate functions
- `SUM`, `COUNT`, `AVG`, `MIN`, `MAX`

---

6. GROUP BY and HAVING Clauses for Grouped Data

6.1 Definition

- **GROUP BY:** Groups rows sharing a common value for aggregation.
- **HAVING:** Filters groups based on aggregate conditions.

6.2 Syntax

```
SELECT course, COUNT(student_id) AS StudentCount
FROM Students
GROUP BY course
HAVING COUNT(student_id) > 10;
```

6.3 Workflow

1. SQL engine groups rows based on `GROUP BY`.
2. Aggregate functions applied to each group.
3. `HAVING` filters groups.

6.4 Features

- Works with aggregate functions
- Filters grouped data

6.5 Advantages

- Summarize data per category
- Advanced filtering on groups

6.6 Exam Keywords

- `GROUP BY`, `HAVING`, aggregate filtering

---

7. Subqueries and Nested Queries

7.1 Definition

- **Subquery:** A query nested inside another query to provide results for the outer query.

7.2 Purpose

- Break complex queries into smaller, manageable queries.
- Dynamically feed results to outer queries.

7.3 Syntax

```
SELECT name, age
FROM Students
WHERE course_id = (SELECT course_id
                   FROM Courses
                   WHERE course_name='CS');
```

7.4 Types

- Single-row subquery
- Multi-row subquery
- Correlated subquery (depends on outer query)

7.5 Features

- Can be used in `SELECT`, `FROM`, `WHERE`, `HAVING`
- Supports dynamic and conditional queries

7.6 Advantages

- Handles complex filtering
- Improves query modularity

7.7 Disadvantages

- Can impact performance if overused
- More complex to debug

7.8 Exam Keywords

- Subquery, Nested query, Correlated subquery

Comparison Table: WHERE vs HAVING

Clause	Purpose	Applied To	Example
WHERE	Filter rows before aggregation	Individual rows	`WHERE age>20`
HAVING	Filter after aggregation	Grouped rows	`HAVING COUNT(student_id)>10`

☑ Important Exam Keywords

- `SELECT`, `FROM`, `WHERE`, `ORDER BY`, `GROUP BY`, `HAVING`
- Aggregate functions: `SUM`, `COUNT`, `AVG`, `MAX`, `MIN`
- Joins: `INNER`, `LEFT`, `RIGHT`, `FULL OUTER`
- Subquery types: single-row, multi-row, correlated
- Logical operators: `AND`, `OR`, `NOT`
- Pattern matching: `LIKE`, `IN`, `BETWEEN`