BASICS OF SQL

1. What is SQL and its significance in data analysis?

SQL (**Structured Query Language**) is a standard language for **storing, retrieving, and managing** data in databases. It helps analysts **extract insights** from large datasets efficiently.

Use Cases in Data Analysis:

- Filtering and grouping data
- Performing aggregations (SUM(), AVG(), etc.)
- Joining multiple tables
- Handling **big data** with optimized queries

2. Differentiate between SQL and MySQL.

Feature	SQL	MySQL
Definition	A query language	A database management system (DBMS)
Purpose	Used to manage relational databases	Implements SQL to manage data
Vendor	Standardized language	Owned by Oracle Corporation
Scalability	Used in various DBMS	Suitable for **small to medium** applications
Example Usage	SELECT * FROM employees;	MySQL databases store and execute SQL queries

3. Explain SQL joins: INNER JOIN, LEFT JOIN, RIGHT JOIN, and FULL JOIN.

Joins are used to **combine** data from two or more tables.

INNER JOIN (Returns only matching rows)

sql

```
SELECT employees.name, departments.department_name
FROM employees
INNER JOIN departments ON employees.department_id = departments.id;
```

LEFT JOIN (Returns all rows from the left table + matching ones from the right)

sql

```
SELECT employees.name, departments.department_name
FROM employees
LEFT JOIN departments ON employees.department_id = departments.id;
```

RIGHT JOIN (Returns all rows from the right table + matching ones from the left)

sql

```
SELECT employees.name, departments.department_name
FROM employees
RIGHT JOIN departments ON employees.department_id = departments.id;
```

FULL JOIN (Returns all rows when there is a match in either table)

sql

```
SELECT employees.name, departments.department_name
FROM employees
FULL JOIN departments ON employees.department_id = departments.id;
```

4. What are the primary components of a SQL query?

A typical SQL query includes:

1. **SELECT** – Specifies columns
2. **FROM** – Specifies table

3. **WHERE** – Filters records
4. **GROUP BY** – Groups results
5. **HAVING** – Filters grouped data
6. **ORDER BY** – Sorts results

Example Query:

sql

```
SELECT department, COUNT(*)
FROM employees
WHERE salary > 50000
GROUP BY department
HAVING COUNT(*) > 5
ORDER BY department ASC;
```

5. Define the terms: table, row, and column in SQL.

- **Table**: A collection of related data in rows and columns.
- **Row (Record)**: A single **entry** in a table.
- **Column (Field)**: A specific attribute in a table.

Example:

ID	Name	Salary
1	Alice	50000
2	Bob	60000

Here,

- "employees" is the **table**
- (1, 'Alice', 50000) is a **row**
- "Name" is a **column**

6. How do you comment out lines in SQL?

- **Single-line comment**: -- This is a comment

- **Multi-line comment**: /* This is a multi-line comment */

7. What is the purpose of the SELECT statement?

The SELECT statement retrieves **specific data** from a table.

sql

```
SELECT name, salary FROM employees;
```

8. How do you retrieve all columns from a table?

Use * (wildcard):

sql

```
SELECT * FROM employees;
```

9. What is a WHERE clause used for?

Filters records based on conditions.

sql

```
SELECT * FROM employees WHERE salary > 50000;
```

10. Explain the difference between WHERE and HAVING.

Feature	WHERE	HAVING
Used for	Row filtering	Group filtering
Works with	SELECT, UPDATE, DELETE	GROUP BY
Example	WHERE salary > 50000	HAVING COUNT(*) > 5

11. How do you eliminate duplicate records?

Use DISTINCT:

sql

SELECT DISTINCT department FROM employees;

12. Difference between COUNT(*) and COUNT(column_name)

- COUNT(*) counts **all rows**, including NULLs.
- COUNT(column_name) counts **non-NULL values**.

13. GROUP BY vs ORDER BY

- GROUP BY: Groups rows by a column
- ORDER BY: Sorts the result

sql

SELECT department, COUNT(*) FROM employees GROUP BY department ORDER BY department;

14. How do you limit the number of records?

- **MySQL**: LIMIT 10
- **SQL Server**: TOP 10
- **Oracle**: FETCH FIRST 10 ROWS ONLY

15. Purpose of the LIKE operator?

Used for pattern matching.

sql

SELECT * FROM employees WHERE name LIKE 'A%'; -- Names starting with A

16. What is a wildcard character?

- % matches **zero or more** characters
- _ matches **exactly one** character

sql

SELECT * FROM employees WHERE name LIKE '_ohn'; -- Matches John, Mohn

17. How do you perform arithmetic operations?

sql

SELECT name, salary * 1.1 AS NewSalary FROM employees;

18. What is a subquery, and how is it used?

A subquery is a query inside another query.

sql

SELECT name FROM employees WHERE salary > (SELECT AVG(salary) FROM employees);

19. Purpose of the IN operator?

Checks if a value exists in a list.

sql

```
SELECT * FROM employees WHERE department_id IN (1, 2, 3);
```

20. Difference between UNION and UNION ALL

- UNION removes duplicates
- UNION ALL includes duplicates

sql

```
SELECT name FROM employees WHERE department_id = 1
UNION ALL
SELECT name FROM employees WHERE department_id = 2;
```

INTERMEDIATE SQL

1. What are aggregate functions in SQL? Provide examples.

Aggregate functions perform **calculations on multiple rows** and return a **single result**.

Common Aggregate Functions:

- SUM(): Adds values
- AVG(): Finds the average
- COUNT(): Counts rows
- MAX(): Finds the maximum
- MIN(): Finds the minimum

Example:

sql

```
SELECT department, COUNT(*) AS total_employees, AVG(salary) AS avg_salary
FROM employees
GROUP BY department;
```

2. How do you handle NULL values in SQL?

NULL means "no value."

Ways to handle NULL values:

- Use IS NULL or IS NOT NULL:

sql

SELECT * FROM employees WHERE salary IS NULL;

- Use COALESCE() to replace NULL with a default value:

sql

SELECT name, COALESCE(salary, 0) AS salary FROM employees;

3. INNER JOIN vs OUTER JOIN

JOIN Type	Description
INNER JOIN	Returns only matching rows
LEFT JOIN	Returns all left-table rows, and matching right-table rows
RIGHT JOIN	Returns all right-table rows, and matching left-table rows
FULL JOIN	Returns all records from both tables

4. What is a self-join in SQL?

A self-join joins a table to itself.

Example (Find employees with the same manager):

sql

SELECT e1.name AS Employee, e2.name AS Manager
FROM employees e1

```
JOIN employees e2 ON e1.manager_id = e2.id;
```

5. How do you create a new table in SQL?

sql

```
CREATE TABLE employees (
   id INT PRIMARY KEY,
   name VARCHAR(100),
   salary DECIMAL(10,2),
   department_id INT
);
```

6. What is a primary key, and why is it important?

A **primary key** uniquely identifies each row in a table.

- Ensures **data integrity**
- Prevents **duplicate records**

sql

```
CREATE TABLE students (
   student_id INT PRIMARY KEY,
   name VARCHAR(100)
);
```

7. ACID properties in databases

ACID ensures **reliable transactions** in SQL:

- **Atomicity**: All or nothing
- **Consistency**: Data remains valid
- **Isolation**: Transactions are independent

- **Durability**: Data persists after a transaction

8. What is normalization, and why is it important?

Normalization **reduces redundancy** and **improves efficiency**.

Forms of Normalization:

- **1NF**: No duplicate columns
- **2NF**: No partial dependencies
- **3NF**: No transitive dependencies

9. How do you add new data to a table?

sql

```
INSERT INTO employees (id, name, salary, department_id)
VALUES (1, 'Alice', 60000, 2);
```

10. What is a stored procedure, and how is it created?

A stored procedure is a reusable SQL block.

sql

```
CREATE PROCEDURE GetEmployees()
AS
BEGIN
   SELECT * FROM employees;
END;
```

Run it using:

sql

EXEC GetEmployees;

11. View vs Table

Feature	Table	View
Data Storage	Stores data physically	Stores query results dynamically
Updatable?	Yes	Sometimes
Use Case	Permanent storage	Read-only queries

Create a view:

sql

```
CREATE VIEW HighSalaryEmployees AS
SELECT * FROM employees WHERE salary > 70000;
```

12. How do you modify an existing table structure?

- **Add a column**:

sql

```
ALTER TABLE employees ADD age INT;
```

- **Modify a column**:

sql

```
ALTER TABLE employees MODIFY COLUMN salary DECIMAL(12,2);
```

13. What is a transaction in SQL?

A **transaction** is a group of SQL statements executed together.

sql

```
BEGIN TRANSACTION;
UPDATE employees SET salary = salary + 5000 WHERE id = 1;
COMMIT;
```

14. Clustered vs Non-Clustered Index

Index Type	Description
Clustered Index	Data is **physically sorted** (only **one per table**)
Non-Clustered Index	**Pointer-based** indexing (**multiple allowed**)

sql

```
CREATE CLUSTERED INDEX idx_emp_salary ON employees(salary);
CREATE NONCLUSTERED INDEX idx_emp_name ON employees(name);
```

15. How do you handle data concurrency in SQL?

- **Locks** prevent multiple transactions from modifying the same data.
- Use READ COMMITTED isolation level:

sql

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

16. Purpose of the EXPLAIN statement

EXPLAIN shows **query execution plans** to optimize performance.

sql

```
EXPLAIN SELECT * FROM employees WHERE salary > 50000;
```

17. What is a trigger, and when would you use one?

A trigger is an **automatic event** that runs before/after a change.

sql

```
CREATE TRIGGER update_salary
AFTER UPDATE ON employees
FOR EACH ROW
BEGIN
  INSERT INTO salary_log(employee_id, old_salary, new_salary)
  VALUES (OLD.id, OLD.salary, NEW.salary);
END;
```

18. UNION vs JOIN

Feature	UNION	JOIN
Combines	Rows from different queries	Columns from different tables
Removes Duplicates?	Yes (UNION) / No (UNION ALL)	No
Example	SELECT name FROM A UNION SELECT name FROM B;	SELECT A.name, B.salary FROM A JOIN B ON A.id = B.id;

19. How do you perform a case-insensitive search?

Use LOWER() or ILIKE (PostgreSQL).

sql

```
SELECT * FROM employees WHERE LOWER(name) = 'john';
```

20. Indexing in SQL and its advantages

Indexes **improve query performance**.

sql

```
CREATE INDEX idx_salary ON employees(salary);
```

ADVANCED SQL

1. What are Common Table Expressions (CTEs)?

CTEs **store temporary query results**.

sql

```
WITH HighSalary AS (
   SELECT * FROM employees WHERE salary > 70000
)
SELECT * FROM HighSalary;
```

2. OLTP vs OLAP

Feature	OLTP	OLAP
Purpose	Transactions	Analytical Queries
Speed	Fast	Slow
Example	Bank transactions	Data warehouses

3. What is a window function?

Used for **ranking, cumulative sums, and moving averages**.

sql

```
SELECT name, salary,
RANK() OVER (ORDER BY salary DESC) AS salary_rank
FROM employees;
```

4. Purpose of COALESCE()

Replaces NULL values.

sql

```
SELECT name, COALESCE(salary, 0) AS salary FROM employees;
```

5. How to optimize SQL queries?

- Use **indexes**
- Avoid SELECT *
- Use **EXPLAIN** to analyze queries

6. Database Normalization (1NF, 2NF, 3NF)

- **1NF**: Atomic columns
- **2NF**: No partial dependencies
- **3NF**: No transitive dependencies

7. Correlated vs Non-Correlated Subquery

- **Correlated**: Depends on outer query
- **Non-Correlated**: Independent

sql

SELECT name FROM employees WHERE salary > (SELECT AVG(salary) FROM employees);

8. How do you perform data migration in SQL?

Data migration involves **transferring data** from one database to another. This can be done using **SQL commands**, ETL tools, or database migration services.

Steps for SQL-Based Data Migration:

1. **Extract Data** (from the source database):

sql

SELECT * INTO OUTFILE '/path/data.csv' FROM old_database.employees;

2. **Transform Data** (if needed, clean or modify data structure).
3. **Load Data** (into the target database):

sql

LOAD DATA INFILE '/path/data.csv' INTO TABLE new_database.employees;

4. **Verify Data** (compare counts and checksums between source and target).
5. **Optimize Performance** (add indexes, remove duplicates).

Database Denormalization and When to Use It

Denormalization **adds redundancy** to speed up queries.

When to Use Denormalization?

- **Read-heavy workloads** (e.g., analytics dashboards).
- **Fewer JOINs needed** (complex queries with many joins slow performance).
- **Precomputed aggregates** (like total sales, count of users).

Example: Denormalized Table

Instead of using **separate orders and customer tables**, store everything in **one table** for faster access:

sql

```
CREATE TABLE order_summary (
  order_id INT PRIMARY KEY,
  customer_name VARCHAR(255),
  total_amount DECIMAL(10,2)
);
```

9. Steps to Troubleshoot a Slow-Running SQL Query

1. **Use EXPLAIN to Analyze Query Execution Plan**

sql

```
EXPLAIN SELECT * FROM employees WHERE salary > 50000;
```

   a. Check for **full table scans** (bad)
   b. Look for **indexed columns**

2. **Check for Missing Indexes**

sql

```
CREATE INDEX idx_salary ON employees(salary);
```

3. **Avoid SELECT * (Fetch Only Needed Columns)**

sql

```
SELECT name, salary FROM employees WHERE salary > 50000;
```

4. **Optimize WHERE Clause**

    a. Use **indexed columns** in WHERE
    b. Avoid **functions on columns**
    c. Rewrite queries efficiently
5. **Use Joins Efficiently**
    a. Prefer INNER JOIN over OUTER JOIN if possible
    b. Ensure **JOINs use indexed columns**
6. **Limit Results If Possible**

sql

SELECT * FROM orders LIMIT 100;

7. **Analyze Locks and Transactions**

sql

SHOW PROCESSLIST;

8. **Check Query Execution Time**

sql

SET profiling = 1;
SELECT * FROM employees WHERE salary > 50000;
SHOW PROFILES;

10. Steps to Troubleshoot a Slow-Running SQL Query

A slow SQL query can be caused by **poor indexing, inefficient joins, large data scans, or locking issues**. Here's a step-by-step approach to troubleshoot and optimize it:

1. Use EXPLAIN to Analyze Query Execution Plan

- The EXPLAIN statement helps identify how SQL executes a query.

- It shows **table scans, index usage, JOIN strategies, and cost estimation**.

Example:

sql

```
EXPLAIN SELECT * FROM employees WHERE salary > 50000;
```

What to check?

- **"Using Index"** (good) vs. **"Full Table Scan"** (bad)
- Look for **large row scans**

2. Identify Missing or Inefficient Indexes

Indexes **speed up search operations**.

- Use indexes on columns in WHERE, JOIN, ORDER BY.
- Avoid **over-indexing**, as it can slow down inserts/updates.

Example: Creating an Index on salary

sql

```
CREATE INDEX idx_salary ON employees(salary);
```

3. Optimize SELECT Queries (Avoid SELECT *)

Fetching **only necessary columns** speeds up queries.

Bad:

sql

```
SELECT * FROM employees WHERE department_id = 3;
```

Good:

sql

```
SELECT name, salary FROM employees WHERE department_id = 3;
```

4. Optimize WHERE Clause Usage

- Use **indexed columns** in the WHERE clause.
- **Avoid functions on indexed columns**, as they prevent index usage.

Bad (index won't work):

sql

```
SELECT * FROM employees WHERE YEAR(join_date) = 2020;
```

Good (index works):

sql

```
SELECT * FROM employees WHERE join_date BETWEEN '2020-01-01' AND '2020-12-31';
```

5. Optimize Joins (Use Proper Indexing)

Joins are often **the biggest performance bottleneck**.

Bad (No index on foreign keys):

sql

```
SELECT e.name, d.department_name
FROM employees e
JOIN departments d ON e.department_id = d.id;
```

Good (Indexing foreign keys):

sql

```
CREATE INDEX idx_department_id ON employees(department_id);
```

6. Use LIMIT to Reduce Result Set

If only a few records are needed, **limit the output** to avoid unnecessary data fetching.

sql

```
SELECT * FROM orders ORDER BY order_date DESC LIMIT 100;
```

7. Check and Reduce Locking Issues

Long-running transactions can block queries. Use:

sql

```
SHOW PROCESSLIST;
```

- Identify **locked queries**.
- Use **shorter transactions** or COMMIT frequently.

8. Analyze Execution Time with Profiling

Use SQL profiling to measure query performance.

sql

```
SET profiling = 1;
SELECT * FROM employees WHERE salary > 50000;
```

```
SHOW PROFILES;
```

9. Optimize Temporary Tables and Subqueries

Using **joins instead of subqueries** often improves performance.

Bad:

sql

```
SELECT * FROM employees WHERE department_id IN (SELECT id FROM departments
WHERE name = 'HR');
```

Good:

sql

```
SELECT e.* FROM employees e
JOIN departments d ON e.department_id = d.id
WHERE d.name = 'HR';
```

10. Partition Large Tables for Faster Queries

Partitioning **splits large tables into smaller ones** to speed up queries.

sql

```
CREATE TABLE orders (
   id INT,
   order_date DATE,
   amount DECIMAL(10,2),
   PRIMARY KEY (id, order_date)
) PARTITION BY RANGE (YEAR(order_date)) (
   PARTITION p1 VALUES LESS THAN (2022),
   PARTITION p2 VALUES LESS THAN (2023)
```

);

Final Checklist for Query Optimization ☑

✓ Use EXPLAIN to analyze query execution.
✓ Ensure proper **indexing** on WHERE, JOIN, and ORDER BY columns.
✓ Avoid SELECT *; fetch only required columns.
✓ Optimize joins and subqueries.
✓ Check for **locking issues**.
✓ Use LIMIT for large datasets.
✓ Partition large tables if needed.