SQL for DATA ANALYTICS

Introduction to SQL and Databases

SQL stands for **Structured Query Language**.
It is a **standard programming language** used to **store, retrieve, manipulate, and manage data** in **relational databases**.

Key Capabilities:

- Query data (`SELECT`)
- Filter and sort results (`WHERE`, `ORDER BY`)
- Combine data from multiple tables (`JOIN`)
- Insert, update, or delete data (`INSERT`, `UPDATE`, `DELETE`)
- Perform aggregations (`SUM`, `AVG`, `COUNT`)

Example:

```
SELECT name, age FROM students WHERE age > 18 ORDER BY age DESC;
```

This retrieves the names and ages of students older than 18, sorted from oldest to youngest.

Role of SQL in Data Analytics

Role of SQL

SQL plays a **central role** by helping analysts:

1. Access Data from Databases:;

Most business data lives in **relational databases** (like MySQL, PostgreSQL). SQL is the primary way to access this data.

```
SELECT * FROM sales WHERE date >= '2024-01-01';
```

2. Clean and Filter Data:

Analysts use SQL to filter invalid, null, or duplicate data.

```
SELECT DISTINCT product_id FROM products WHERE price IS NOT NULL;
```

3. Join and Relate Tables:

Data is often split across multiple tables. SQL lets analysts combine them for meaningful analysis.

```
SELECT customers.name, orders.order_date
FROM customers
JOIN orders ON customers.id = orders.customer_id;
```

4. Generate Reports:

SQL helps compute KPIs, such as monthly sales or customer retention.

```
SELECT MONTH(order_date) AS month, SUM(amount) AS revenue
FROM orders
GROUP BY MONTH(order_date);
```

Types of Databases: Relational vs. Non-relational

Relational Databases (SQL-based):

- **Structured** in rows and columns (like Excel sheets)
- **Data is stored in tables**
- Supports **ACID** properties (Atomicity, Consistency, Isolation, Durability)

Examples: MySQL, PostgreSQL, SQLite, Oracle

```
-- Table: Employees
ID | Name    | Department
--------------------------
1  | Alice   | HR
2  | Bob     | IT
```

Non-relational Databases (NoSQL):

- Used for **semi-structured or unstructured data**
- Data stored in forms like:
  - Key-value pairs (Redis)
  - Documents (MongoDB)
  - Graphs (Neo4j)
- More flexible, scalable for big data & real-time apps

Example (MongoDB JSON document):

```json
{
  "name": "Alice",
  "department": "HR",
  "skills": ["Excel", "Communication"]
}
```

Introduction to RDBMS (Relational Database Management System)

An **RDBMS** is a system/software used to manage **relational databases**.

Features:

- **Tables with relationships**
- **SQL support**
- **Data Integrity & Security**
- Allows **multi-user access**
- Ensures **data consistency** with rules (constraints)

Use Case in Analytics:

An RDBMS helps you:

- Retrieve sales trends
- Track customer history
- Analyze product performance

Basic Terminology

TERM	EXPLANATION
Table	A structured set of data stored in rows and columns. Like a spreadsheet.
Row (Record)	A single entry in a table. Each row = one record of data.
Column (Field)	A specific attribute (like name, date, price)
Primary Key	A **unique identifier** for each row (e.g., student_id)
Foreign Key	A column that links to the **primary key of another table**

Try this on an online SQL tool:

```
CREATE TABLE students (
  student_id INT PRIMARY KEY,
  name TEXT,
  age INT
);

INSERT INTO students VALUES (1, 'Alice', 20), (2, 'Bob', 21);

SELECT * FROM students;
```

You’ll create a table, insert data, and retrieve it — your first hands-on experience with SQL!

Basic SQL Queries

1. SELECT Statement

The SELECT statement is used to **retrieve data** from one or more tables in a database.

Syntax:

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

Example:

```
SELECT name, age FROM students;
```

This retrieves the name and age columns from the students table.

2. `FROM` and `WHERE` Clauses

- **`FROM`** tells SQL which table to get the data from.

- **`WHERE`** filters rows based on conditions.

Example:

```sql
SELECT name, age
FROM students
WHERE age > 18;
```

This fetches the names and ages of students **older than 18** from the `students` table.

3. Filtering with Logical Operators: AND, OR, NOT

These are used in WHERE to combine multiple conditions.

AND — All conditions must be true:

```
SELECT name
FROM students
WHERE age > 18 AND city = 'Delhi';
```

OR — At least one condition must be true:

```
SELECT name
FROM students
WHERE city = 'Delhi' OR city = 'Mumbai';
```

NOT — Reverses a condition:

```
SELECT name
FROM students
WHERE NOT city = 'Delhi';
```

4. Comparison Operators

These are used to **compare values** in a condition within the `WHERE` clause.

OPERATOR	DESCRIPTION	EXAMPLE
`=`	Equal to	`WHERE age = 18`
`>`	Greater than	`WHERE marks > 50`
`<`	Less than	`WHERE age < 30`
`!=` or `<>`	Not equal to	`WHERE city != 'Delhi'`
`BETWEEN`	Between a range (inclusive)	`WHERE marks BETWEEN 60 AND 90`
`IN`	Matches any value in a list	`WHERE city IN ('Delhi', 'Pune')`
`LIKE`	Pattern matching using wildcards	`WHERE name LIKE 'A%'`
`IS NULL`	Checks if a value is missing	`WHERE email IS NULL`

Example: BETWEEN, IN, LIKE, IS NULL

```sql
-- Find students between 18 and 25
SELECT name FROM students
WHERE age BETWEEN 18 AND 25;

-- Find students in specific cities
SELECT name FROM students
WHERE city IN ('Delhi', 'Mumbai');

-- Find students whose name starts with 'A'
SELECT name FROM students
WHERE name LIKE 'A%';

-- Find records where phone number is missing
SELECT name FROM students
WHERE phone IS NULL;
```

5. Sorting Results: `ORDER BY`

Use `ORDER BY` to sort the output in **ascending (ASC)** or **descending (DESC)** order. Default is `ASC`.

Example:

```sql
-- Sort students by age (youngest to oldest)
SELECT name, age
FROM students
ORDER BY age ASC;

-- Sort by marks from highest to lowest
SELECT name, marks
FROM students
ORDER BY marks DESC;
```

6. Renaming Columns Using AS (Aliasing)

The AS keyword is used to give a **temporary name (alias)** to a column or table in the output.

Example:

```
SELECT name AS student_name, marks AS total_score
FROM students;
```

Aliasing for Calculations:

```
SELECT name, marks * 1.1 AS adjusted_score
FROM students;
```

Aliasing in Aggregates:

```
SELECT AVG(marks) AS average_marks
FROM students;
```

Summary

FEATURE	KEYWORD USED	PURPOSE
Select data	`SELECT`	Choose which columns to display
Specify table	`FROM`	Define the table you're querying
Filter data	`WHERE`	Add conditions to limit rows
Combine conditions	`AND`, `OR`, `NOT`	Logic for filtering
Compare values	`=`, `>`, `<`, `!=`, `BETWEEN`, `IN`, `LIKE`, `IS NULL`	Set filtering rules
Sort results	`ORDER BY`	Display data in order
Rename columns	`AS`	Improve readability

Aggregate Functions- COUNT(),SUM(),AVG(),IN(), MAX()

Aggregate functions **operate on sets of rows** and return a **single summary value**.

FUNCTION	DESCRIPTION	EXAMPLE OUTPUT
`COUNT()`	Number of rows or non-NULL values	Count of customers
`SUM()`	Total of values in a column	Total sales amount
`AVG()`	Average value of a numeric column	Average product price
`MIN()`	Minimum value in a column	Earliest order date
`MAX()`	Maximum value in a column	Highest salary

Example Table: `sales`

order_id	customer_id	amount	city
1	C101	500	Delhi
2	C102	700	Mumbai
3	C101	300	Delhi
4	C103	NULL	Kolkata
5	C104	900	Mumbai

Examples of Aggregate Functions:

```sql
-- Count total number of sales (including NULLs)
SELECT COUNT(*) AS total_sales FROM sales;

-- Count only non-NULL amount entries
SELECT COUNT(amount) AS non_null_sales FROM sales;

-- Total sales amount
SELECT SUM(amount) AS total_revenue FROM sales;

-- Average sale amount
SELECT AVG(amount) AS average_amount FROM sales;

-- Minimum and maximum sale
SELECT MIN(amount) AS min_sale, MAX(amount) AS max_sale FROM
sales;
```

2. GROUP BY Clause

Used to **group rows** that have the **same values** in specified columns. It's often used with aggregate functions.

Syntax:

```
SELECT column, AGG_FUNC(column2)
FROM table
GROUP BY column;
```

Example:

```
-- Total sales per city
SELECT city, SUM(amount) AS city_sales
FROM sales
GROUP BY city;
```

Output:

city	**city_sales**
Delhi	800
Mumbai	1600
Kolkata	NULL

3. HAVING Clause for Filtering Aggregates

Unlike WHERE, which filters **before** aggregation, HAVING filters **after aggregation** — useful when using GROUP BY.

Syntax:

```
SELECT column, AGG_FUNC(column2)
FROM table
GROUP BY column
HAVING AGG_FUNC(column2) condition;
```

Example:

```
-- Cities with total sales greater than 1000
SELECT city, SUM(amount) AS city_sales
FROM sales
GROUP BY city
HAVING SUM(amount) > 1000;
```

```
Output:  city                    city_sales
         Mumbai                    1600
```

4. Combining `GROUP BY` and `ORDER BY`

You can **group**, **aggregate**, and **sort** the results using `ORDER BY`.

Example:

```
-- Average sales per city, ordered from highest to lowest
SELECT city, AVG(amount) AS avg_sale
FROM sales
GROUP BY city
ORDER BY avg_sale DESC;
```

Output:

city	avg_sale
Mumbai	800.0
Delhi	400.0
Kolkata	NULL

Key Differences: WHERE vs HAVING

Clause	Used For	Works With Aggregates?
WHERE	Filtering rows	No
HAVING	Filtering groups	Yes

Example:

```
-- Invalid: cannot use SUM() in WHERE
-- SELECT city FROM sales WHERE SUM(amount) > 1000;

-- Correct use with HAVING:
SELECT city, SUM(amount)
FROM sales
GROUP BY city
HAVING SUM(amount) > 1000;
```

Data Filtering and Analysis

1. Using Wildcards with `LIKE` for Pattern Matching

The `LIKE` operator is used to search for a **specific pattern in a column** — typically strings.

Wildcards in SQL:

WILDCARD	DESCRIPTION	EXAMPLE PATTERN
`%`	Matches **any number of characters**	`'A%'` → Starts with A
`_`	Matches **exactly one character**	`'A_'` → A + 1 char

Example Table: `employees`

emp_id	name	department
1	Alice	HR
2	Alok	IT
3	Bob	Finance
4	Anjali	HR
5	John	IT

Example Queries:

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

-- Names ending with 'k'
SELECT name FROM employees
WHERE name LIKE '%k';

-- Names where second letter is 'l'
SELECT name FROM employees
WHERE name LIKE '_l%';
```

Output for `LIKE 'A%'`:

name
Alice
Alok
Anjali

2. Filtering `NULL` Values

`NULL` represents **missing or unknown** data in SQL. It **cannot be compared** using `=`, you must use `IS NULL` or `IS NOT NULL`.

Example Table: `customers`

id	name	email
1	Riya	riya@mail.com
2	Aarav	NULL
3	Kabir	kabir@mail.com
4	Zara	NULL

Example Queries:

```sql
-- Customers without email addresses
SELECT name FROM customers
WHERE email IS NULL;

-- Customers with email addresses
SELECT name FROM customers
WHERE email IS NOT NULL;
```

Output for `IS NULL`:

name
Aarav
Zara

4. Basic Subqueries in WHERE Clause

A **subquery** is a query inside another query, often used in WHERE to **filter data based on another table's result**.

Syntax:

```
SELECT column1 FROM table1
WHERE column2 IN (SELECT column3 FROM table2 WHERE condition);
```

Example Tables:

customers

id	name
C101	Alice
C102	Bob
C103	John

Orders

order_id	customer_id	product
1	C101	Phone
2	C102	Laptop

Example Query:

```
-- Find customers who have placed an order
SELECT name
FROM customers
WHERE id IN (
    SELECT customer_id
    FROM orders
);
```

Output:

name
Alice
Bob

Another Example with NOT IN:

```
-- Customers who have not placed any orders
SELECT name
FROM customers
WHERE id NOT IN (
    SELECT customer_id
    FROM orders
);
```

Output:

name
John

Summary Table

CONCEPT	DESCRIPTION	EXAMPLE SYNTAX
`LIKE`, `%`, `_`	Match patterns in text	`WHERE name LIKE 'A%'`
`IS NULL`, `IS NOT NULL`	Handle missing values	`WHERE email IS NULL`
`DISTINCT`	Remove duplicates in results	`SELECT DISTINCT product`
`Subquery in WHERE`	Use result of one query in another	`WHERE id IN (SELECT id FROM ...)`

Data Joins

1. Understanding Relationships Between Tables

In **Relational Databases**, data is normalized and stored in **multiple related tables** instead of one large table.

Why?

- To **reduce redundancy**
- Ensure **data consistency**
- Improve **data organization and retrieval**

Common Relationship Types:

RELATIONSHIP TYPE	EXAMPLE
One-to-One	Each person has one passport
One-to-Many	One customer places many orders
Many-to-Many	Students enrolled in multiple courses

Example:

Let’s say we have two tables:

Customers Table:

customer_id	name	city
C101	Alice	Delhi
C102	Bob	Mumbai
C103	John	Kolkata

Orders Table:

order_id	customer_id	product	amount
1	C101	Phone	500
2	C102	Laptop	700
3	C101	Tablet	300

The `customer_id` in `Orders` references the primary key in `Customers`. This is a **One-to-Many** relationship (one customer can place many orders).

2. `INNER JOIN`

An `INNER JOIN` returns only **matching rows** from both tables — based on the join condition.

Syntax:

```
SELECT columns
FROM table1
INNER JOIN table2
ON table1.key = table2.key;
```

Example:

```
SELECT customers.name, orders.product, orders.amount
FROM customers
INNER JOIN orders
ON customers.customer_id = orders.customer_id;
```

name	product	amount
Alice	Phone	500
Bob	Laptop	700
Alice	Tablet	300

3. `LEFT JOIN` and `RIGHT JOIN`

- `LEFT JOIN`: All rows from the **left table**, with matching rows from the right table. If no match, shows `NULL`.
- `RIGHT JOIN`: All rows from the **right table**, with matching rows from the left.

Example (LEFT JOIN):

```
SELECT customers.name, orders.product
FROM customers
LEFT JOIN orders
ON customers.customer_id = orders.customer_id;
```

Output:

name	product
Alice	Phone
Alice	Tablet
Bob	Laptop
John	NULL

Example (RIGHT JOIN):

```
SELECT customers.name, orders.product
FROM customers
RIGHT JOIN orders
ON customers.customer_id = orders.customer_id;
```

Output is same as `INNER JOIN` here because all orders have matching customers. But it will **also include unmatched orders** if any exist.

4. FULL OUTER JOIN (If Supported)

A FULL OUTER JOIN returns **all records** from both tables. If no match, returns NULL.

Not supported in MySQL, but works in PostgreSQL and SQL Server.

Example:

```
SELECT customers.name, orders.product
FROM customers
FULL OUTER JOIN orders
ON customers.customer_id = orders.customer_id;
```

Output:

name	product
Alice	Phone
Alice	Tablet
Bob	Laptop
John	NULL
NULL	Charger

5. Joining More Than Two Tables

You can chain multiple `JOINs` to pull data from 3 or more tables.

Tables:

Products:

product_id	product_name	price
P1	Phone	500
P2	Laptop	700
P3	Tablet	300

Orders (Updated):

order_id	customer_id	product_id
1	C101	P1
2	C102	P2
3	C101	P3

Query: Join Customers + Orders + Products

```
SELECT customers.name, products.product_name, products.price
FROM customers
JOIN orders ON customers.customer_id = orders.customer_id
JOIN products ON orders.product_id = products.product_id;
```

Output:

name	product_name	price
Alice	Phone	500
Bob	Laptop	700
Alice	Tablet	300

6. **Practical Use Cases in Analytics**

USE CASE	TABLES TO JOIN	PURPOSE
Sales Reporting	`orders + customers`	Revenue by region or customer type
Product Performance	`orders + products`	Analyze best-selling products
Customer Order History	`customers + orders`	Lifetime value, frequency of purchase
Marketing Campaign Effectiveness	`campaigns + leads + sales`	Conversion rate per campaign
Inventory Tracking	`products + inventory + sales`	Stock level after orders

Summary: Join Types Comparison

JOIN TYPE	DESCRIPTION
`INNER JOIN`	Only matching rows
`LEFT JOIN`	All from left table + matches from right
`RIGHT JOIN`	All from right table + matches from left
`FULL OUTER JOIN`	All rows from both sides, with NULLs if no match
Multiple JOINs	Used for complex data relationships