SQL Notes with Examples

🔹 What is Data?

Data is raw, unprocessed facts and figures that need interpretation to be meaningful. It can be numbers, text, images, etc.

Key Points:

- Data is meaningless without context.
- Becomes useful only after processing (turns into information).
- Types: Structured (tabular), Unstructured (images, audio, etc.)

Example:

- 80 → just a number.
- Student scored 80 in Maths → now it's meaningful (information).

🔹 What is a Database?

A database is a structured collection of related data stored electronically for easy access, management, and update.

Key Points:

- Stores data in tables (rows and columns).
- Reduces redundancy, ensures integrity.
- Can be queried using SQL.

🔹 What is RDBMS?

RDBMS (Relational Database Management System) stores data in related tables using the relational model.

Key Features:

- Each table has a **primary key**.
- Tables can have **foreign keys** to define relationships.
- Supports **normalization** to reduce redundancy.
- Enforces **data integrity**.
- Uses SQL for data operations.

Examples: MySQL, Oracle, PostgreSQL, SQL Server

◈ SQL vs MySQL

Feature	SQL	MySQL
Type	Language	RDBMS
Usage	Used in all RDBMS	Specific to MySQL system
Ownership	ISO standard	Oracle Corporation

Summary:
SQL is the standard language; MySQL is a popular system that uses it.

◈ Data Types in MySQL

1. Numeric: INT, BIGINT, FLOAT, DOUBLE, DECIMAL
2. String: CHAR, VARCHAR, TEXT, ENUM, SET
3. Date/Time: DATE, DATETIME, TIME, YEAR, TIMESTAMP
4. Boolean: BOOLEAN (alias of TINYINT(1))

Example:

```
CREATE TABLE students (
  id INT,
  name VARCHAR(50),
  dob DATE,
  is_active BOOLEAN
);
```

◈ DDL (Data Definition Language)

Used to define and modify database structure.

Commands:

- **CREATE**: Create tables, schemas.
- **ALTER**: Modify structure (add/remove columns).
- **DROP**: Delete table or schema permanently.
- **TRUNCATE**: Remove all data from a table (structure intact).

- **RENAME**: Rename table or columns.

Examples:

```
CREATE TABLE employees (id INT, name VARCHAR(50));

ALTER TABLE employees ADD salary INT;

DROP TABLE employees;

TRUNCATE TABLE employees;

RENAME TABLE employees TO staff;
```

◈ DML (Data Manipulation Language)

Used to manage the data inside tables.

- **INSERT**: Add new records.
- **UPDATE**: Modify existing data.
- **DELETE**: Remove data.

Examples:

```
INSERT INTO employees (id, name) VALUES (1, 'John');

UPDATE employees SET name = 'David' WHERE id = 1;

DELETE FROM employees WHERE id = 1;
```

◈ DQL (Data Query Language)

Used to retrieve data.

- **SELECT**: Fetch data from one or more tables.

Examples:

```
SELECT * FROM employees;

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

◈ DCL (Data Control Language)

Used to control access to the database.

- **GRANT**: Give privileges.
- **REVOKE**: Take away privileges.

Examples:

GRANT SELECT, INSERT ON employees TO 'user1';

REVOKE INSERT ON employees FROM 'user1';

◈ TCL (Transaction Control Language)

Used to manage database transactions.

- **COMMIT**: Save changes.
- **ROLLBACK**: Undo changes.
- **SAVEPOINT**: Set a rollback point.

Example:

START TRANSACTION;

UPDATE employees SET salary = 70000 WHERE id = 1;

SAVEPOINT before_bonus;

UPDATE employees SET salary = 80000 WHERE id = 2;

ROLLBACK TO before_bonus;

COMMIT;

◈ Constraints in SQL

Ensure data accuracy and integrity.

- NOT NULL: No null values.
- UNIQUE: No duplicate values.
- PRIMARY KEY: Uniquely identifies rows (NOT NULL + UNIQUE).
- FOREIGN KEY: Links to another table.
- CHECK: Validates values.
- DEFAULT: Sets a default value.

Example:

```
CREATE TABLE users (
  id INT PRIMARY KEY,
  name VARCHAR(50) NOT NULL,
```

```
  age INT CHECK (age > 0),
  city VARCHAR(50) DEFAULT 'Delhi'
);
```

◈ Composite Primary Key

Combining two or more columns as a unique identifier.

Example:

```
CREATE TABLE enrollments (
  student_id INT,
  course_id INT,
  PRIMARY KEY (student_id, course_id)
);
```

◈ AUTO_INCREMENT

Automatically generates unique ID values.

Example:

```
CREATE TABLE products (
  id INT AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(100)
);
```

◈ DISTINCT

Removes duplicates from the result.

Example:

```
SELECT DISTINCT department FROM employees;
```

◈ ORDER BY

Sorts results in ascending or descending order.

Examples:

SELECT * FROM employees ORDER BY salary DESC;

SELECT * FROM employees ORDER BY name;

◈ GROUP BY

Groups rows for aggregation.

Example:

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

◈ HAVING

Filters grouped data (used with GROUP BY).

Example:

SELECT department, COUNT(*) AS emp_count

FROM employees

GROUP BY department

HAVING emp_count > 5;

◈ LIKE Operator

Used for pattern matching in WHERE clause.

- % matches any number of characters.
- _ matches exactly one character.

Examples:

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

SELECT * FROM employees WHERE name LIKE '%n'; **-- Ends with n**

SELECT * FROM employees WHERE name LIKE '_a%'; **-- Second letter is a**

◈ Aggregate Functions (With Explanation)

1. **COUNT()** – Counts total rows.

SELECT COUNT(*) FROM employees;

2. **SUM()** – Adds values of a column.

SELECT SUM(salary) FROM employees;

3. **MIN()/MAX()** – Gets smallest/largest values.

SELECT MIN(salary), MAX(salary) FROM employees;

4. **AVG()** – Calculates average.

SELECT AVG(salary) FROM employees;

🔷 All SQL Joins (With Explanation)

1. **INNER JOIN**
    - Returns only matching rows from both tables.
    - Most used join.

    SELECT e.name, d.name

    FROM employees e

    INNER JOIN departments d ON e.dept_id = d.id;

2. **LEFT JOIN**
    - Returns all rows from left table, matched from right.
    - NULL if no match in right.

    SELECT e.name, d.name

    FROM employees e

    LEFT JOIN departments d ON e.dept_id = d.id;

3. **RIGHT JOIN**
    - All rows from right + matching from left.

    SELECT e.name, d.name

    FROM employees e

    RIGHT JOIN departments d ON e.dept_id = d.id;

4. **FULL OUTER JOIN** (MySQL workaround with UNION)
    - All rows from both tables.

    SELECT e.name, d.name

    FROM employees e

    LEFT JOIN departments d ON e.dept_id = d.id

UNION

SELECT e.name, d.name

FROM employees e

RIGHT JOIN departments d ON e.dept_id = d.id;

5. **CROSS JOIN**
    - Cartesian product (every row with every row).
    - Typically the number of rows returned by this join is equal to the product of rows in 2 tables.

SELECT e.name, d.name

FROM employees e

CROSS JOIN departments d;

6. **SELF JOIN**
    - A table joined with itself.
    - Here there is only a single table and we join within the same table using a common column.

SELECT A.name AS Employee, B.name AS Manager

FROM employees A

JOIN employees B ON A.manager_id = B.id;

7. **NATURAL JOIN**
    - Automatically joins using same column names.
    - This join is not recommended because this will not work as expected when column names doesn't match.

SELECT * FROM employees

NATURAL JOIN departments;

Stored Procedures

🔷 What is a Stored Procedure?

A **Stored Procedure** is a **precompiled collection of SQL statements** stored in the database. It performs a specific task and can be **executed repeatedly** with different inputs.

🔷 Key Characteristics

- Stored **inside the database**.
- Improves **performance** (compiled once, used many times).
- Supports **input**, **output**, and **INOUT** parameters.
- Helps in implementing **business logic** at the database level.
- Can include **control flow statements** like IF, CASE, WHILE, etc.

🔷 Benefits of Stored Procedures

1. **Reusability**: Write once and call multiple times.
2. **Maintainability**: Easy to update logic at one place.
3. **Performance**: Compiled and cached by the database.
4. **Security**: Can control access by granting EXECUTE permission only.
5. **Modularity**: Break complex operations into reusable units.
6. **Reduced network traffic**: One call replaces many SQL queries.

🔷 Simple Example: No Parameters

```
DELIMITER //

CREATE PROCEDURE GetAllEmployees()

BEGIN

   SELECT * FROM employees;

END //

DELIMITER ;


-- To call:

CALL GetAllEmployees();
```

◈ Example: IN Parameter

```
DELIMITER //
CREATE PROCEDURE GetEmployeeById(IN empId INT)
BEGIN
   SELECT * FROM employees WHERE id = empId;
END //
DELIMITER ;

-- To call:
CALL GetEmployeeById(2);
```

◈ Example: OUT Parameter

```
DELIMITER //
CREATE PROCEDURE GetEmployeeCount(OUT empCount INT)
BEGIN
   SELECT COUNT(*) INTO empCount FROM employees;
END //
DELIMITER ;

-- To call:
CALL GetEmployeeCount(@total);
SELECT @total;
```

◈ Example: INOUT Parameter

```
DELIMITER //
CREATE PROCEDURE IncrementSalary(INOUT sal INT)
BEGIN
   SET sal = sal + 1000;
END //
```

```
DELIMITER ;

-- To call:
SET @mysalary = 40000;
CALL IncrementSalary(@mysalary);
SELECT @mysalary;
```

◈ Using Control Statements in Procedures

```
DELIMITER //
CREATE PROCEDURE CheckAge(IN age INT)
BEGIN
   IF age >= 18 THEN
      SELECT 'Adult';
   ELSE
      SELECT 'Minor';
   END IF;
END //
DELIMITER ;

CALL CheckAge(20); -- Output: 'Adult'
```

◈ Altering and Dropping Stored Procedures

Drop Procedure:

```
DROP PROCEDURE IF EXISTS procedure_name;
```

Alter Procedure (MySQL does not support ALTER, recreate it instead):

```
DROP PROCEDURE IF EXISTS procedure_name;
-- then recreate it
```

◈ Show All Procedures in a Database

```
SHOW PROCEDURE STATUS WHERE Db = 'your_database_name';
```

◈ When to Use Stored Procedures?

- When repetitive operations are performed on the data.
- When business logic should be placed closer to the data.
- For report generation, auditing, and bulk operations.
- To encapsulate complex joins and operations.