📒 SQL Notes for Software Testers (MNC Interview Ready)

✅ 1. Database Basics

What is a Database?

A collection of structured data stored and accessed electronically.

Common DB used in companies

- MySQL
- Oracle
- PostgreSQL
- SQL Server

Important Terms

- **Table** – rows + columns
- **Row (Record)** – single entry
- **Column (Attribute)** – field
- **Primary Key (PK)** – unique identifier
- **Foreign Key (FK)** – links two tables
- **Unique Key** – no duplicates
- **Not Null** – cannot be empty

✅ 2. Most Important SQL Commands for Testers

Software testers mostly use **SELECT queries** to validate data.

👉 SELECT

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

👉 WHERE

```
SELECT * FROM employees WHERE department='QA';
```

👉 Operators

- `=, !=, >, <, >=, <=`
- `AND`
- `OR`
- `LIKE`
- `IN`
- `BETWEEN`
- `IS NULL`

Examples:

```
SELECT * FROM orders WHERE amount > 1000 AND status='paid';
SELECT * FROM users WHERE name LIKE 'S%';
```

✅ 3. Sorting & Filtering

👉 ORDER BY

```
SELECT * FROM employees ORDER BY salary DESC;
```

👉 DISTINCT

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

✅ 4. Aggregate Functions (Used in Testing Reports, Dashboard APIs)

- **COUNT()**
- **SUM()**
- **AVG()**
- **MAX()**
- **MIN()**

Examples:

```
SELECT COUNT(*) FROM users WHERE status='active';
SELECT department, AVG(salary) FROM employees GROUP BY department;
```

✅ 5. GROUP BY & HAVING

Used to test Reporting Modules.

```
SELECT status, COUNT(*)
FROM orders
GROUP BY status
HAVING COUNT(*) > 10;
```

✓ 6. JOINS (Most Important for MNC Interviews)

INNER JOIN

```
SELECT u.name, o.order_date
FROM users u
INNER JOIN orders o ON u.id = o.user_id;
```

LEFT JOIN

```
SELECT u.name, o.order_date
FROM users u
LEFT JOIN orders o ON u.id = o.user_id;
```

RIGHT JOIN

FULL OUTER JOIN (depends on DB support)

☞ Interview Expectation:
You should know **why JOINs are required** – to fetch data from **multiple related tables**.

✓ 7. Subqueries

Used when queries are nested.

```
SELECT name
FROM users
WHERE id IN (
   SELECT user_id FROM orders WHERE amount > 5000
);
```

✓ 8. Views

Companies use read-only views for Testers.

```
SELECT * FROM active_users_view;
```

✅ 9. CRUD (Basic Only for Testers)

INSERT

```
INSERT INTO users(name, email) VALUES ('John', 'john@test.com');
```

UPDATE

```
UPDATE users SET status='inactive' WHERE id=3;
```

DELETE

```
DELETE FROM users WHERE id=10;
```

Constraints Testing – Full Explanation (For Software Testers)

Constraints ensure **data accuracy, consistency, and integrity** in a database.
As testers, you must verify whether the system **correctly follows these rules**.

Below are the **5 most important constraints**, each with:
✓ Explanation
✓ Real-time examples
✓ Test cases
✓ Example SQL queries

◈ 1. PRIMARY KEY (PK)

A **Primary Key** uniquely identifies each row and **cannot be NULL**.

✓ Example Table

```
CREATE TABLE users (
    user_id INT PRIMARY KEY,
    name VARCHAR(50),
    email VARCHAR(50)
);
```

✓ Real-Time Meaning

- Two users **cannot** have the same `user_id`.
- `user_id` must always have a value (not empty).

✓ Tester Scenarios — What You Test

Test Case	Expected Result
Insert duplicate `user_id`	Should FAIL
Insert NULL `user_id`	Should FAIL
Insert unique `user_id`	Should PASS

✓ Queries

```
INSERT INTO users VALUES (1, 'John', 'john@test.com');     -- PASS
INSERT INTO users VALUES (1, 'Sam', 'sam@test.com');       -- FAIL
(duplicate PK)
INSERT INTO users VALUES (NULL, 'Ram', 'ram@test.com');    -- FAIL (NULL PK)
```

◈ 2. FOREIGN KEY (FK)

A **Foreign Key** ensures that a value must exist in another table.

✓ Parent Table

```
CREATE TABLE departments (
    dept_id INT PRIMARY KEY,
    dept_name VARCHAR(50)
);
```

✓ Child Table

```
CREATE TABLE employees (
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(50),
    dept_id INT,
    FOREIGN KEY (dept_id) REFERENCES departments(dept_id)
);
```

✓ Real-Time Meaning

- If `dept_id = 10` does not exist in `departments`,
  you **cannot** insert an employee with `dept_id = 10`.

✓ Tester Scenarios

Test Case	Expected Result
Insert employee with valid dept_id	PASS
Insert employee with dept_id not in department	FAIL
Delete department used by an employee	FAIL (unless cascading allowed)

✓ Queries

```
INSERT INTO departments VALUES (1, 'HR');

INSERT INTO employees VALUES (101, 'John', 1);    -- PASS
INSERT INTO employees VALUES (102, 'Sam', 5);     -- FAIL (no dept_id = 5)
```

◈ 3. UNIQUE Constraint

Ensures that a column contains **unique values**, but **allows one NULL**.

✓ Example

```
CREATE TABLE users (
    id INT PRIMARY KEY,
    email VARCHAR(50) UNIQUE
);
```

✓ Real-Time Meaning

- Two users cannot have the **same email**.
- NULL is allowed (but only one NULL depends on DB; MySQL allows multiple NULLs).

✓ Tester Scenarios

Test Case	Expected Result
Insert duplicate email	FAIL
Insert unique emails	PASS

✓ Queries

```
INSERT INTO users VALUES (1, 'john@test.com');     -- PASS
INSERT INTO users VALUES (2, 'john@test.com');     -- FAIL (duplicate)
INSERT INTO users VALUES (3, NULL);                -- PASS
INSERT INTO users VALUES (4, NULL);                -- PASS or FAIL (DB
dependent)
```

◈ 4. NOT NULL Constraint

Ensures the column **must have a value**.

✓ Example

```
CREATE TABLE products (
    id INT PRIMARY KEY,
    product_name VARCHAR(50) NOT NULL,
    price INT NOT NULL
);
```

✓ Real-Time Meaning

- A product cannot be created without a name.
- A price must always be entered.

✓ Tester Scenarios

Test Case	Expected Result
Insert without product_name	FAIL
Insert with NULL price	FAIL
Insert with all values	PASS

✓ Queries

```
INSERT INTO products VALUES (1, 'Laptop', 50000);     -- PASS
INSERT INTO products VALUES (2, NULL, 30000);         -- FAIL (product_name)
INSERT INTO products VALUES (3, 'Mouse', NULL);       -- FAIL (price)
```

◈ 5. DATA TYPE Constraint

Ensures values **match the expected type** (INT, VARCHAR, DATE, etc.).

✓ Example

```
CREATE TABLE students (
    roll_no INT,
    name VARCHAR(50),
    dob DATE
);
```

✓ Real-Time Meaning

- You cannot insert a string into an INT column.
- You cannot insert `abcd` into a DATE column.

✓ Tester Scenarios

Test Case	Expected Result
Insert text into INT column	FAIL
Insert wrong date format	FAIL
Insert valid formats	PASS

✔ Queries

```
INSERT INTO students VALUES (1, 'Alex', '2024-10-01');  -- PASS
INSERT INTO students VALUES ('abc', 'Sam', '2024-10-05'); -- FAIL (INT)
INSERT INTO students VALUES (2, 'Ram', 'abcd');           -- FAIL (DATE)
```

◈ 6. DEFAULT Constraint

Automatically inserts a **default value** when no value is supplied.

✔ Example

```
CREATE TABLE orders (
    order_id INT PRIMARY KEY,
    amount INT,
    status VARCHAR(20) DEFAULT 'Pending'
);
```

✔ Real-Time Meaning

If the tester does not provide `status`, DB will set `'Pending'`.

✔ Tester Scenarios

Test Case	Expected Result
Insert order without status	status = 'Pending'
Insert with custom status	Use user value

✔ Queries

```
INSERT INTO orders (order_id, amount) VALUES (101, 500);
-- status auto = 'Pending'

INSERT INTO orders VALUES (102, 700, 'Completed');
-- status = 'Completed'
```

🎯 How Testers Validate Constraints

✔ Check through UI + DB

Example:
If UI allows duplicate email but DB rejects → **BUG**.

✔ API + DB validation

POST request inserts data → verify DB follows constraint.

✓ Negative testing

Enter invalid data → DB should reject it.

✓ Boundary testing

Check data limits (e.g., age must not be <18).

✅ What is a Stored Procedure?

A **Stored Procedure (SP)** is a set of SQL statements stored in the database that can be executed as a single unit.

✓ Why Testers need to test SPs?

Because stored procedures run critical **business logic** like:

- Creating orders
- Updating wallet balance
- Validating login
- Generating reports

So testers must ensure the SP **works correctly and returns correct output**.

□ Example Stored Procedure

Let's say we have a table:

```
CREATE TABLE users (
    user_id INT PRIMARY KEY,
    name VARCHAR(50),
    email VARCHAR(50),
    status VARCHAR(20)
);
```

Now we create a stored procedure:

```
CREATE PROCEDURE GetUserDetails @uid INT
AS
BEGIN
    SELECT user_id, name, email, status
    FROM users
    WHERE user_id = @uid;
```

```
END
```

□ How to Execute (Call) the Stored Procedure

As a tester, you normally run:

```
EXEC GetUserDetails 101;
```

OR

```
CALL GetUserDetails(101);   -- In MySQL
```

□ What Happens When You Execute EXEC GetUserDetails 101?

The procedure receives the parameter:

- `@uid = 101`

Then it runs the query inside it:

```
SELECT user_id, name, email, status
FROM users
WHERE user_id = 101;
```

It will return the user with ID **101**.

□ Example Output

If table contains:

user_id	name	email	status
101	Rahul	rahul@test.com	active
102	Anita	anita@test.com	inactive

Then:

```
EXEC GetUserDetails 101;
```

Returns:

user_id	name	email	status
101	Rahul	rahul@test.com	active

🔥 Stored Procedure Testing – What You Test?

▯ 1. Input Parameter Testing

Check how SP behaves when different values are passed.

✔ Test Cases:

Input	Expected Output
Valid user_id (101)	Returns user details
Invalid user_id (999)	Returns zero rows
NULL	Error or no result
Negative values	Should not return data

▯ 2. Output Data Validation

Compare SP output with the actual table data.

✔ Does returned email match DB?
✔ Does the status match UI?
✔ Is the query filtering correctly?

▯ 3. Performance Testing

Stored Procedure must run fast.

✔ Does it take < 2 seconds?
✔ Is it using indexes?
✔ Avoids full table scan?

⍰ 4. Business Logic Testing

For example, if SP calculates discount, tax, status update etc.

✓ Is calculation correct?
✓ Is business rule applied properly?

⍰ 5. Error Handling

Check whether SP handles:

✓ Null inputs
✓ Wrong data types
✓ Invalid parameters
✓ Missing data

□ Real-Time Complex Example

Stored Procedure: Create a new order

```
CREATE PROCEDURE CreateOrder
    @user_id INT,
    @amount DECIMAL(10,2)
AS
BEGIN
    IF EXISTS (SELECT 1 FROM users WHERE user_id=@user_id)
    BEGIN
        INSERT INTO orders(user_id, amount, status)
        VALUES (@user_id, @amount, 'Pending');
    END
    ELSE
    BEGIN
        RAISERROR('User not found', 16, 1);
    END
END
```

□ Tester Responsibilities

✓ Test Case 1: Valid user

```
EXEC CreateOrder 101, 500.00;
```

Expected:

- Row inserted in orders table
- Status = "Pending"

✓ Test Case 2: Invalid user

```
EXEC CreateOrder 999, 200;
```

Expected:

- Error message: **User not found**
- No insertion should happen

✓ Test Case 3: Amount = NULL

Should fail due to NOT NULL constraint

✓ Test Case 4: Large amount

Check system limits

What is a Transaction?

A **transaction** in SQL is a group of SQL statements that must run **together** as one single unit.

☞ **Either ALL operations must succeed (COMMIT)**
☞ **Or ALL must fail (ROLLBACK)**

This ensures **data accuracy**, especially in financial systems.

🔥 Real-Time Example (Bank Wallet Transfer)

Let's say a user transfers ₹100 from Wallet A to Wallet B.

The database must perform 2 steps:

Step 1: Deduct 100 from Wallet A

```
UPDATE wallets SET balance = balance - 100 WHERE user_id = 1;
```

Step 2: Add 100 to Wallet B

```
UPDATE wallets SET balance = balance + 100 WHERE user_id = 2;
```

☞ If ANY step fails, the money should NOT be deducted.

So we wrap these inside a **transaction**:

□ Transaction Example

```
BEGIN TRANSACTION;

UPDATE wallets SET balance = balance - 100 WHERE user_id = 1;
UPDATE wallets SET balance = balance + 100 WHERE user_id = 2;

COMMIT;
```

✔ If both updates succeed → COMMIT

❌ If any error occurs → ROLLBACK

□ What if something goes wrong?

Example:

- Step 1 deducts 100 from User 1
- Step 2 fails because User 2's account doesn't exist

Then:

```
ROLLBACK;
```

💡 Meaning: *Undo the deduction also.*
So User 1 gets his money back.

□ This is why transactions are important in banking!

Without transactions:

- ₹100 deducted from User 1

- Not added to User 2
😱 Money disappears → major bug

◆ Basic Transaction Commands

Command	Meaning
`BEGIN` / `START TRANSACTION`	Start a new transaction
`COMMIT`	Save the changes
`ROLLBACK`	Undo all changes
`SAVEPOINT`	Partial rollback checkpoint

Testing Transactions (Real-Time Scenarios for Testers)

---

1. Money Transfer Test

Scenario: Transfer ₹100

- Wallet A should decrease by 100
- Wallet B should increase by 100
- Both operations should be atomic

Tester Query:

```
SELECT balance FROM wallets WHERE user_id = 1;
SELECT balance FROM wallets WHERE user_id = 2;
```

✓ 2. Negative Test – Transfer When Balance is Low

If Wallet A has only ₹50:

Expected:

- Transaction should fail
- No deduction should occur

3. Booking System Test

Example Transaction:

```
BEGIN;

UPDATE seats SET status='booked' WHERE seat_id=12;
INSERT INTO tickets(user_id, seat_id) VALUES (10, 12);

COMMIT;
```

Tester checks:

- Seat status updated
- Ticket created
- No double booking

4. Rollback Testing

Force an error in second query:

```
BEGIN;

UPDATE wallets SET balance = balance - 100 WHERE user_id = 1;
INSERT INTO wallets(user_id, balance) VALUES (NULL, 100);   -- Error

ROLLBACK;
```

Tester validates:

- Wallet A balance remains unchanged

5. Multi-user Testing

Two users try to book the **same seat** at the same time.

Expected:

- Only one transaction should succeed
- The other should get "seat already booked"

SAVEPOINT Example (Advanced)

Used for partial rollback.

```
BEGIN;

UPDATE accounts SET balance = balance - 100 WHERE id=1;
SAVEPOINT step1;

UPDATE accounts SET balance = balance + 100 WHERE id=2;

ROLLBACK TO step1;   -- Undo step 2 only

COMMIT;
```

ACID Properties (Interview Must-Know)

Transactions follow **ACID:**

Property	Meaning
A – Atomicity	All or nothing
C – Consistency	Data must remain valid
I – Isolation	Simultaneous transactions don’t affect each other
D – Durability	Once committed, data is saved permanently

Interview Question Example

Q: Why are transactions important in banking systems?
A: Transactions ensure that multiple operations like debit and credit happen as a single atomic unit. If any step fails, the entire action is rolled back, preventing money loss or inconsistent balances.