Module-4

1. **What is RDBMS?**

* In **software testing, RDBMS** stands for **Relational Database Management System.**
* It’s not exclusive to testing — it’s a core concept in software and database management — but in testing, it becomes very important.

**An** **RDBMS** **is software that manages** **databases** **based on the** **relational model** **In an RDBMS**:

Data is stored in **tables** (rows and columns).

Each table can relate to others through **keys** (like primary key and foreign key).

You use **SQL (Structured Query Language)** to create, read, update, and delete data (CRUD operations).

**Examples of RDBMS:**

MySQL

Oracle Database

Microsoft SQL Server

PostgreSQL

**In Software Testing**, **RDBMS matters because**:

* **Data Validation**: Testers check whether the application saves, updates, deletes, or retrieves the correct data from the database.
* **Database Testing**: It involves writing SQL queries to verify data integrity, business rules, and relationships between tables.
* **Performance Testing**: Testers may check how quickly the database responds under heavy loads.
* **Security Testing**: Testers verify that sensitive data in the database is protected.

Common tasks involving RDBMS in testing:

* Writing and executing **SQL queries** to verify test results.
* Checking **data consistency** across multiple tables.
* Validating **transactions** (e.g., rollback if an operation fails).
* Ensuring **constraints** (like "NOT NULL", "FOREIGN KEY") are properly enforced.

1. **What is SQL?**

* **SQL = Structured Query Language**
* SQL is used to **communicate** with a database.
* It helps testers **check, add, update, or delete** data.
* It is important for **Database Testing**.

**Why SQL is used in Testing**

* To **verify** if correct data is saved in the database.
* To **fetch data** and compare it with what is shown on the app.
* To **check data integrity** (no missing, duplicate, or wrong data).
* To **test performance** of database queries.
* To **validate business rules** (like discounts, login validations, etc.)

**Simple SQL Commands for Testing**

|  |  |  |
| --- | --- | --- |
| Command | Purpose | Example |
| SELECT | Get data from database | SELECT \* FROM users; |
| INSERT | Add new data | INSERT INTO users (name, email) VALUES (Khushi, khushi@gmail.com'); |
| UPDATE | Change existing data | UPDATE users SET name=’khishi’WHERE id=1; |
| DELETE | Remove data | DELETE FROM users WHERE id=1; |

1. **Write SQL Commands**

|  |  |  |
| --- | --- | --- |
| SQL Command | Purpose | Example |
| SELECT | To **fetch data** from a table | SELECT \* FROM employees; |
| WHERE | To **filter data** based on a condition | SELECT \* FROM employees WHERE department = 'HR'; |
| INSERT | To **add new data** into a table | INSERT INTO employees (name, department, salary) VALUES ('soham', 'Sales', 50000); |
| UPDATE | To **change existing data** in a table | UPDATE employees SET salary = 55000 WHERE name = 'John'; |
| DELETE | To **remove data** from a table | DELETE FROM employees WHERE name = 'John'; |
| ORDER BY | To **sort data** (ascending/descending) | SELECT \* FROM employees ORDER BY salary DESC; |
| JOIN | To **combine data** from two tables | SELECT employees.name, departments.name FROM employees JOIN departments ON employees.dept\_id = departments.id; |
| GROUP BY | To **group data** and perform operations (like COUNT, SUM) | SELECT department, COUNT(\*) FROM employees GROUP BY department; |
| LIKE | To **search** for a pattern | SELECT \* FROM employees WHERE name LIKE 'A%'; |
| IN | To **match** multiple values | SELECT \* FROM employees WHERE department IN ('HR', 'Sales'); |

1. **What is join?**

* **JOIN** is a **SQL command** used to **combine data from two or more tables** based on a related column between them.
* In software testing, JOIN helps testers fetch and check related data from different tables together.

**Why JOIN is important in Testing?**

* To **verify** that **linked data** is stored correctly across tables.
* To **test relationships** like **customer orders, user profiles**, etc.
* To **find missing or wrong linked data.**

**Types of JOIN:**

|  |  |
| --- | --- |
| JOIN Type | Meaning |
| INNER JOIN | Only matching data from both tables. |
| LEFT JOIN | All data from the left table + matching data from the right table. |
| RIGHT JOIN | All data from the right table + matching data from the left table. |
| FULL JOIN | All data from both tables (matches and non-matches). |

1. **Write type of joins?**

* In **SQL**, there are **4 main types of joins.**
* They help testers **combine and check data** from multiple tables.

1. **INNER JOIN**

* **Returns** only **matching** rows from both tables
* **If no match, no data** is shown.

**For example;**

SELECT Customers.name, Orders.order\_id

FROM Customers

INNER JOIN Orders

ON Customers.id = Orders.customer\_id;

1. **LEFT JOIN**

* **Returns** all rows from the **left table**, even if there's **no match** in the right table.
* If no match, right table fields are **NULL**.

**For example;**

SELECT Customers.name, Orders.order\_id

FROM Customers

LEFT JOIN Orders

ON Customers.id = Orders.customer\_id;

1. **RIGHT JOIN**

* **Returns** all rows from the **right table**, even if there's **no match** in the left table.
* If no match, left table fields are **NULL.**

**For example;**

SELECT Customers.name, Orders.order\_id

FROM Customers

RIGHT JOIN Orders

ON Customers.id = Orders.customer\_id;

1. **FULL JOIN**

* **Returns** all rows from **both** tables.
* If no match, missing side is shown as **NULL**.

**For example;**

SELECT Customers.name, Orders.order\_id

FROM Customers

FULL JOIN Orders

ON Customers.id = Orders.customer\_id;

**Summary table:-**

|  |  |
| --- | --- |
| JOIN Type | What it shows |
| INNER JOIN | Only matched data |
| LEFT JOIN | All left table + matching right |
| RIGHT JOIN | All right table + matching left |
| FULL JOIN | All data from both sides |

**6. How Many constraint and describes it self.**

* Constraints in software testing are the limitations or restrictions that affect the testing process.
* These constraints can arise from various factors and impact the scope, depth, and efficiency of testing efforts.

Here are some common constraints in software testing:

* **Time Constraints:** Tight deadlines often limit the amount of time available for thorough testing, potentially leading to incomplete test coverage.
* **Budget Constraints:** Limited financial resources can restrict the availability of testing tools, infrastructure, and the number of testers.
* **Resource Constraints:** This includes the availability of skilled testers, suitable testing environments, hardware, software licenses, and test data.
* **Scope Constraints:** The defined scope of the project or a specific testing phase might limit the features or areas of the software that can be tested.
* **Data Constraints:** The availability, quality, and management of test data can pose significant challenges. Creating realistic and sufficient test data can be time-consuming and complex.
* **Technical Constraints:** Limitations in the technology being used, such as compatibility issues with testing tools or the complexity of the system under test, can restrict testing activities.
* **Documentation Constraints:** Incomplete or missing documentation can make it difficult for testers to understand the system requirements and design effective test cases.
* **Communication Issues:** Poor communication between development, testing, and other teams can lead to misunderstandings, delays, and inefficiencies in the testing process.
* **Skill and Expertise Constraints:** The lack of specific technical skills or domain knowledge within the testing team can limit the types and effectiveness of testing that can be performed.
* **Access Constraints:** Restrictions on accessing certain parts of the system, third-party integrations, or specific environments can hinder testing efforts.
* **Quality Constraints:** Sometimes, there might be a perceived constraint where speed of delivery is prioritized over thorough testing and achieving a high level of quality.

Self-Describing Constraints

* The concept of "self-describing constraints" in software testing isn't a widely established or formally defined term. However, we can interpret it in a few ways:

1. **Constraints Documented Within Test Artifacts:** This could refer to constraints that are explicitly documented within test plans, test cases, or test scripts themselves. For example, a test case might specify:

* The specific environment required to run the test.
* Any prerequisite data or system states.
* Performance targets or acceptable response times.
* Dependencies on other test cases or system components. By including these descriptions directly within the test artifacts, the constraints become self-evident to anyone examining or executing the tests.

1. **Constraints Inherent in the Test Design or Automation:** In some cases, the way tests are designed or automated can inherently reflect constraints. For instance:

* **Performance Tests:** The design of a load test script to simulate a specific number of concurrent users directly describes the load constraint being tested.
* **Security Tests:** Automated vulnerability scans are constrained by the tools used and the specific security rules they are configured to check.
* **Platform-Specific Tests:** Test suites designed to run only on a particular operating system or browser implicitly describe the platform constraint.

1. **Constraints Expressed as Assertions or Checks:** Within automated tests, assertions or checks can act as self-describing constraints. For example, an assertion that verifies a database record count is less than 100 directly describes a constraint on the data volume.

**7. Difference between RDBMS vs DBMS**

|  |  |  |
| --- | --- | --- |
| **Feature** | **DBMS Testing** | **RDBMS Testing** |
| Focus | Data integrity within individual files/structures, data retrieval mechanisms specific to the DBMS. | Data integrity across related tables, adherence to the relational model, correctness and performance of SQL queries, enforcement of relationships and constraints. |
| Data Integrity | Verifying data accuracy and consistency within the storage format used. | Verifying referential integrity (foreign keys), domain integrity (data types, constraints), entity integrity (primary keys), and data consistency across related tables. |
| Relationships | Testing how the application manages relationships between data (if any). | Rigorously testing the defined relationships between tables (using joins), ensuring data consistency when related data is modified or deleted (referential integrity). |
| Querying | Testing the specific commands or APIs used to retrieve and manipulate data. | Primarily focused on testing the correctness and performance of SQL queries (SELECT, INSERT, UPDATE, DELETE), including complex joins, subqueries, and stored procedures. |
| Transactions | Testing transaction management based on the DBMS's capabilities (if any). | Thoroughly testing ACID properties of transactions: Atomicity (all or nothing), Consistency (valid state after transaction), Isolation (concurrent transactions don't interfere), Durability (permanent changes). |
| Normalization | Less emphasis as DBMS often doesn't enforce normalization. | Ensuring the database schema is properly normalized to reduce redundancy and improve data integrity. |
| Security | Testing security features provided by the specific DBMS. | Testing user roles, permissions, and access controls at the database level. Ensuring data is protected from unauthorized access and modification. |
| Tools & Skills | Testers need to be proficient in the data manipulation methods specific to the DBMS being used. | Testers need strong SQL skills and familiarity with RDBMS concepts, database testing tools, and potentially performance testing tools for SQL queries. |
| Complexity | Can vary depending on the type of DBMS. Simpler file-based systems might be less complex to test in some aspects. | Often more complex due to the structured nature, relationships between tables, and the need to test SQL queries and transactions. |

**8. What is an SQL alias?**

* the context of SQL and software testing that involves databases, an **SQL alias** is a temporary, alternative name given to a table or a column within a single SQL query.
* This temporary name makes the query more readable and easier to understand, especially when dealing with complex queries involving multiple tables or lengthy column names.
* **Think of it like giving a nickname to something for a specific conversation.** The original name still exists, but you use the nickname for convenience within that particular context.

Here's why SQL aliases are important in software testing:

* **Improved Readability of Test Queries:** When writing SQL queries to verify data during testing, especially for complex scenarios involving joins across multiple tables, aliases make the queries much clearer.

**For example,**

-- Without aliases

SELECT Orders.OrderID, Customers.CustomerName

FROM Orders

INNER JOIN Customers ON Orders.CustomerID = Customers.CustomerID;

-- With aliases

SELECT o.OrderID, c.CustomerName

FROM Orders AS o

INNER JOIN Customers AS c ON o.CustomerID = c.CustomerID;

* **Avoiding Ambiguous Column Names:** When joining multiple tables that have columns with the same name, aliases are essential to specify which table a particular column belongs to. This prevents ambiguity and ensures the correct data is retrieved for testing.

**For example,**

SELECT e.EmployeeID, d.DepartmentID, e.Name AS EmployeeName, d.Name AS DepartmentName

FROM Employees AS e

INNER JOIN Departments AS d ON e.DepartmentID = d.DepartmentID;

* **Simplifying Complex Calculations:** When performing calculations on columns, aliases can provide more descriptive names for the resulting computed columns, making the test results easier to interpret.

**For example,**

SELECT Price \* Quantity AS TotalAmount

FROM OrderItems;

* **Facilitating Self-Joins:** When a table is joined with itself (a self-join), aliases are crucial to differentiate between the two instances of the same table. This is often used in testing hierarchical data or comparing rows within the same table.

**For example,**

SELECT e1.EmployeeName AS Employee, e2.EmployeeName AS Manager

FROM Employees AS e1

INNER JOIN Employees AS e2 ON e1.ManagerID = e2.EmployeeID;

**9. Write a query to create the table in Structured Query Language.**

* create table student (id int,firstname varchar(50),lastname varchar(50),email varchar(100), mobile bigint, address varchar(250));

**10. Write a query to insert data into table.**

* student(id,firstname,lastname,email,mobile,address)
* values(1,"godhani","khushi","khushi@gmail.com",9173816590,"nikol");
* [INSERT](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/insert.html) INTO student (id,firstname,lastname,email,mobile,address)  [VALUES](mailto:VALUES)(2,"pooja","sarvaiya","pooja@gmail.com",9925016590,"nikol");
* [INSERT](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/insert.html) INTO student (id,firstname,lastname,email,mobile,address)  [VALUES](mailto:VALUES)(3,"yatrik","prajapati","yatrik@gmail.com",9925206590,"nikol");
* [INSERT](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/insert.html) INTO student (id,firstname,lastname,email,mobile,address)  [VALUES](mailto:VALUES)(4,"nirali","gajera","nirali@gmail.com",8925206590,"nikol");
* [INSERT](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/insert.html) INTO student (id,firstname,lastname,email,mobile,address)

[VALUES](mailto:VALUES)(5,"jenish","vaghasiya","jenish@gmail.com",8925206220,"nikol");

* [INSERT](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/insert.html) INTO student (id,firstname,lastname,email,mobile,address)

[VALUES](mailto:VALUES)(6,"divya","hirpara","divya@gmail.com",8916506220,"nikol");

* [INSERT](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/insert.html) INTO student (id,firstname,lastname,email,mobile,address)

[VALUES](mailto:VALUES)(7,"senha","patel","sneha@gmail.com",8916506250,"nikol");

* [INSERT](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/insert.html) INTO student (id,firstname,lastname,email,mobile,address)

[VALUES](mailto:VALUES)(8,"amruta","jotkar","amruta@gmail.com",8989506250,"surta");

* [INSERT](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/insert.html) INTO student (id,firstname,lastname,email,mobile,address)
* [VALUES](mailto:VALUES)(9,"jenish","vaghasiya","jenish@gmail.com",8989503050,"manali");
* [INSERT](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/insert.html) INTO student (id,firstname,lastname,email,mobile,address)

[VALUES](mailto:VALUES)(10,"tanvi","dhamliya","tanvi@gmail.com",8989403050,"nikol");

**11. Write a query to update data into table with validations.**

* [update](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/update.html) student [set](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/set.html) mobile= 9925016590 where id=6;

**12. Write a query to delete data from table with validations.**

* [delete](http://localhost/phpmyadmin/url.php?url=https://dev.mysql.com/doc/refman/8.0/en/delete.html) from student where id=10;

**13. Write a query to insert new column in existing table.**

* ALTER TABLE Employees ADD COLUMN Department VARCHAR(50);

**14. Write a query to drop table and database.**

* -- Drop Table DROP TABLE Employees;
* -- Drop Database DROP DATABASE IF EXISTS MyDatabase;

**15. Write a query to find max and min value from table.**

-- Find the maximum and minimum salary from the Employees table

SELECT MAX(Salary) AS MaxSalary, MIN(Salary) AS MinSalary

FROM Employees;

**16. Create two tables named Seller and Product apply foreign key in product table Fetch data from both table using different joins.**

-- Create the Seller table

CREATE TABLE Seller (

    SellerID INT PRIMARY KEY,

    SellerName VARCHAR(255) NOT NULL,

    SellerLocation VARCHAR(255)

);

-- Create the Product table with a foreign key referencing SellerID

CREATE TABLE Product (

    ProductID INT PRIMARY KEY,

    ProductName VARCHAR(255) NOT NULL,

    SellerID INT,

    Price DECIMAL(10, 2) NOT NULL,

    FOREIGN KEY (SellerID) REFERENCES Seller(SellerID)

);

-- Insert sample data into Seller

INSERT INTO Seller (SellerID, SellerName, SellerLocation) VALUES

    (1, 'TechCorp', 'New York'),

    (2, 'GlobalElectronics', 'London'),

    (3, 'BookHaven', 'Chicago'),

    (4, 'FashionStyle', 'Los Angeles');

-- Insert sample data into Product

INSERT INTO Product (ProductID, ProductName, SellerID, Price) VALUES

    (101, 'Laptop', 1, 1200.00),

    (102, 'Smartphone', 1, 800.00),

    (103, 'Office Chair', 2, 150.00),

    (104, 'Harry Potter', 3, 25.00),

    (105, 'T-Shirt', 4, 15.00),

    (106, 'Data Book', NULL, 30.00);

-- 1. INNER JOIN: Get products and their sellers where there's a match in both tables

SELECT Product.ProductName, Seller.SellerName

FROM Product

INNER JOIN Seller ON Product.SellerID = Seller.SellerID;

-- 2. LEFT JOIN (or LEFT OUTER JOIN): Get all products and their sellers, including products without a seller

SELECT Product.ProductName, Seller.SellerName

FROM Product

LEFT JOIN Seller ON Product.SellerID = Seller.SellerID;

-- 3. RIGHT JOIN (or RIGHT OUTER JOIN): Get all sellers and their products, including sellers without any products.

SELECT Product.ProductName, Seller.SellerName

FROM Product

RIGHT JOIN Seller ON Product.SellerID = Seller.SellerID;

-- 4. FULL OUTER JOIN: Get all products and all sellers, showing all possible combinations (Note: FULL OUTER JOIN is not supported in MySQL)

-- SELECT Product.ProductName, Seller.SellerName

-- FROM Product

-- FULL OUTER JOIN Seller ON Product.SellerID = Seller.SellerID;

-- 5. CROSS JOIN: Get all possible combinations of products and sellers (Cartesian product)

SELECT Product.ProductName, Seller.SellerName

FROM Product

CROSS JOIN Seller;

**17. What is API Testing**

* API (Application Programming Interface) testing is a type of software testing that focuses on verifying the functionality, reliability, performance, and security of 1 the application programming interfaces (APIs).
* In software development, APIs act as intermediaries, allowing different software systems to communicate and exchange data with each other without needing to know the underlying implementation details.

Key aspects of API testing:

* **Focus:** Testing the business logic layer of an application.

Scope:

* Verifying data accuracy and completeness.
* Checking error codes, response times, and authentication.
* Validating the API's behavior under different conditions.
* Types of API Testing:
* **Functional Testing:** Validates that the API endpoints are working as expected.
* **Performance Testing:** Measures the API's response time, throughput, and stability under load.
* **Security Testing:** Identifies vulnerabilities.
* **Contract Testing:** Verifies that an API meets the expectations of its consumers.

**18. Types of API Testing**

* In software testing, API testing involves various approaches to ensure the functionality, performance, and security of APIs. Here's a breakdown of the key types:
* **Functional Testing**: This is the most common type, focusing on validating that the API endpoints work as expected. It involves sending requests to the API with different inputs and verifying the responses against the expected outcomes. This includes checking:

Correctness of the response data.

Status codes.

Error messages.

Headers.

* **Performance Testing**: This evaluates the API's performance under various conditions, including normal and peak loads. It helps to identify bottlenecks, measure response times, and assess the API's scalability. Common types of performance tests include:

Load testing.

Stress testing.

Endurance testing.

* **Security Testing**: This focuses on identifying vulnerabilities and ensuring that the API is protected against unauthorized access and security threats. Key aspects include:

Authentication and authorization mechanisms.

Data encryption.

Input validation.

Vulnerability scanning.

* **Integration Testing**: This verifies that the API works correctly with other systems or applications. It ensures that data exchange and workflows between different components function as expected.
* **Contract Testing**: This verifies that an API meets the expectations of its consumers. It ensures that the API's responses adhere to a predefined contract or schema, ensuring compatibility between the API provider and consumer.
* **End-to-End Testing**: While not exclusively API testing, this involves testing the entire workflow of an application, including its API interactions, to ensure that the system as a whole functions correctly.