

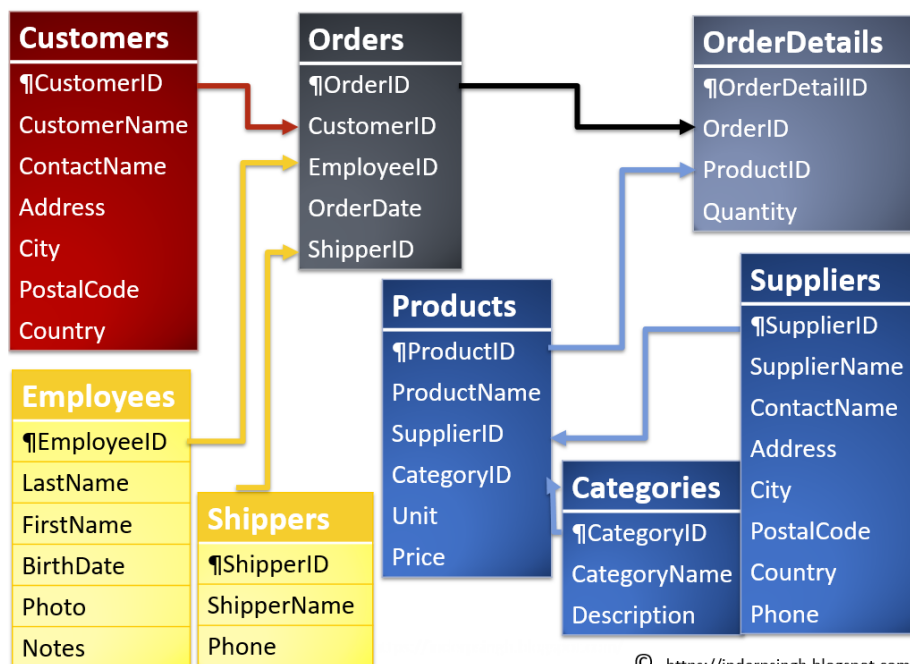
# SQL for SDET, QA Tester and Manual Testers

## SQL for SDET, QA Tester and Manual Testers - explained by Inder P Singh

SQL Concepts (Intro, Basics, Simple Queries, Joins, Grouping, Aggregate Functions and Subqueries )

SQL Queries (Automation Testing, API Testing, Performance Testing, Manual Testing and Data Validation)

SQL in Database Platforms, SQL Questions and Answers, SQL Best Practices and SQL Tips and Tricks



© <https://inderpsingh.blogspot.com/>

Follow Inder P Singh in [LinkedIn](#) for more resources in test automation and software testing.

Download for reference Like Share View SQL Queries video <https://youtu.be/BxMmC77fj9Y>  
View [Database Testing videos playlist](#)

Software and Testing Training YouTube Channel <https://youtube.com/@QA1>

Software Testing Space Blog <https://inderpsingh.blogspot.com/> Copyright © 2024 All Rights Reserved.

1. Introduction to SQL for Testers .....	4
2. Basic SQL Concepts .....	9
3. Writing Simple SQL Queries .....	15
4. Working with Joins and Multiple Tables .....	19
5. Intermediate SQL Concepts .....	23
6. Advanced SQL Concepts .....	27
7. SQL in Different Database Platforms .....	33
8. SQL Queries for Manual Testers .....	40
9. SQL in Automation Testing .....	43
10. SQL for API Testing .....	50
11. SQL Queries for Performance Testing .....	54
12. Data Validation Using SQL .....	59
13. SQL Questions for QA Interview Preparation .....	63
14. SQL Best Practices for Testers .....	65
15. SQL Tips and Tricks for QA Testers .....	68

# 1. Introduction to SQL for Testers

**Question:** What is SQL, and why is it important in QA testing?

**Answer:** SQL (Structured Query Language) is a standard language used to interact with relational databases for storing, retrieving, and manipulating data. In QA testing, [SQL](#) is needed because testers need to validate the data stored in databases, verify data consistency, to test if the application's backend is functioning correctly.

For example, when testing a web application, a tester may need to run [SQL queries](#) to find out if the data entered in the frontend is correctly saved in the database. QA testers frequently use SQL for:

- Verifying if CRUD operations (Create, Read, Update, Delete) are working as expected
- Validating reports or UI data against the database for accuracy
- Checking database constraints (e.g. unique keys, foreign keys) during functional testing.

Example: To verify that user data is correctly inserted into the users table after registration, a QA tester might run: `SELECT * FROM users WHERE username = 'john_doe';`

SQL is needed in different testing approaches:

- **Manual Testing:**
  - a. Manual testers use SQL to manually validate if the data in the database is the same as what is shown in the application's user interface (UI).
  - b. They can write SQL queries to check that new entries, updates, or deletions made through the UI are reflected correctly in the database. **Example:** After updating a user's email address through the application UI, a manual tester may run the following query to verify if the email has been updated: `SELECT email FROM users WHERE username = 'john_doe';`
- **Automation Testing (SDET):**
  - a. In automation, SDETs can write SQL queries within their test scripts to fetch and validate data directly from the database as part of automated test validation.
  - b. SQL can also be used to set up test data before executing test cases or to clean up after tests.
  - c. **Example:** In a Selenium test script, SQL queries might be used to verify database records:

```
String query = "SELECT COUNT(*) FROM orders WHERE user_id = 101 AND status = 'shipped'";
ResultSet rs = statement.executeQuery(query);
rs.next();
int count = rs.getInt(1);
assertEquals(count, 1, "Order not found or not shipped");
```

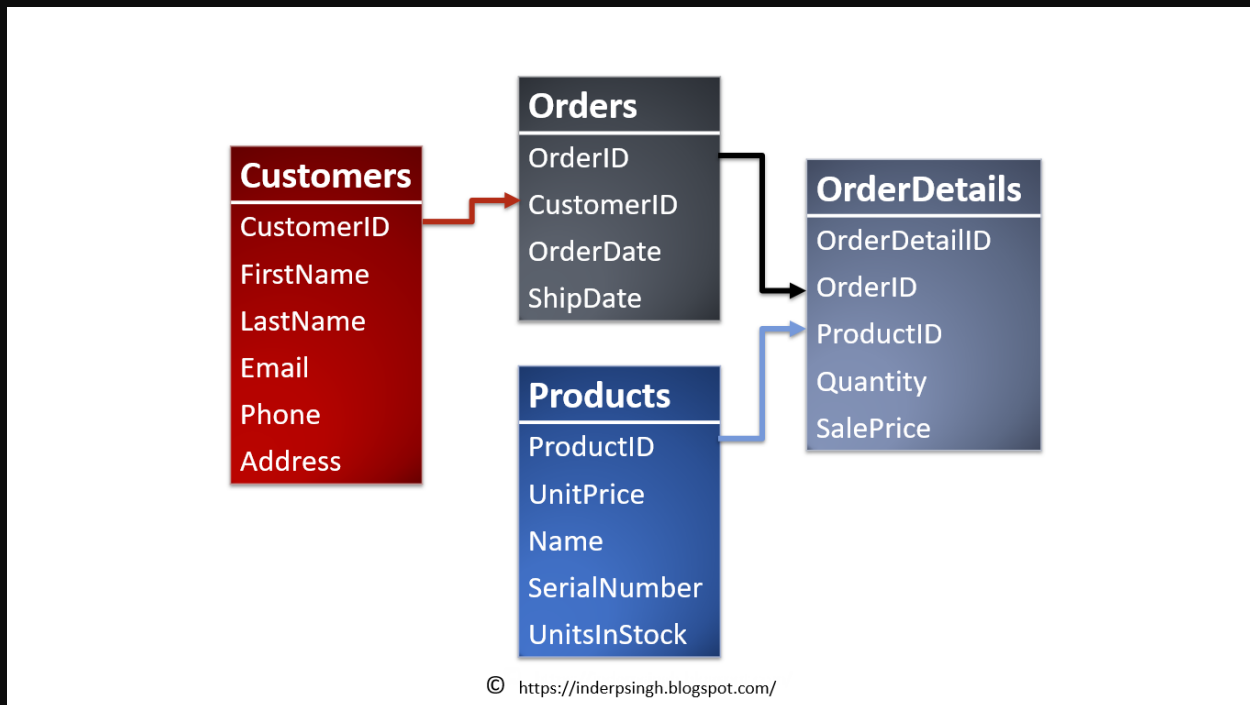
- **API Testing:** Testers use SQL in API testing to validate that the data sent via API calls is correctly inserted or updated in the database. **Example:** After making a POST request to an API that creates a new order, a tester can run SQL to validate the order creation:  
`SELECT * FROM orders WHERE order_id = '12345';`

**Question:** What is the difference between relational databases (SQL) and non-relational databases (NoSQL)?

**Answer:** Relational databases (SQL) and non-relational databases (NoSQL) differ in their structure, use cases, and data handling methods.

Relational Databases (SQL): [SQL databases](#) store data in structured tables with rows and columns. They use SQL queries to perform operations on the data. Data is organized in relations (tables), and each table has a predefined schema. In the example below, Customers, Orders, OrderDetails and Products are the tables. The Customers table has the columns CustomerID, FirstName and so on.

Relational databases are ideal for complex queries, transactions, and applications where data integrity is crucial (e.g. financial applications, ERPs). Examples: MySQL, Oracle, PostgreSQL, SQL Server.



**Non-relational Databases (NoSQL):** NoSQL databases store data in a flexible, schema-less manner, typically as documents, key-value pairs, or wide-column stores. They are suited for handling large volumes of unstructured or semi-structured data, such as [JSON](#), [XML](#), or blobs. NoSQL is used for applications that need scalability, like social media platforms or real-time analytics. Examples: MongoDB (Document-based), Cassandra (Wide-column), Redis (Key-value), Neo4j (Graph database).

**Question:** When should QA testers use SQL databases vs NoSQL databases?

**Answer:** It depends on the type of application, its data structure, and specific project requirements:

- **SQL databases** should be used when:
  - Data integrity and ACID (Atomicity, Consistency, Isolation, Durability) properties are critical.
  - There is a need for complex joins, relationships, and transactional consistency.
  - The data is structured and has well-defined relationships (e.g. e-commerce sites, inventory management).
  - **Example:** If a QA team is testing a banking application, a SQL database like PostgreSQL would be suitable due to the need for data accuracy, complex queries, and strong relationships between tables.

Download for reference Like Share View SQL Queries video <https://youtu.be/BxMmC77fj9Y>  
View [Database Testing videos playlist](#)

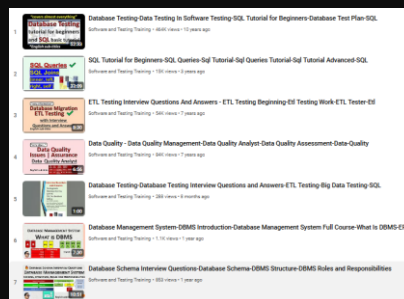
Software and Testing Training YouTube Channel <https://youtube.com/@QA1>

Software Testing Space Blog <https://inderpsingh.blogspot.com/> Copyright © 2024 All Rights Reserved.

- **NoSQL databases** should be used when:
  - The application requires high scalability and performance over large datasets.
  - The data is semi-structured or unstructured (e.g. [JSON](#), [XML](#)).
  - There are no complex relationships or strict schema requirements (e.g., social media, IoT applications).
  - **Example:** For testing a document-heavy application, like a content management system (CMS), a NoSQL database like MongoDB would be appropriate because of the flexibility in storing different document formats.

## Chapter Summary:

- **SQL** is essential for QA testers to learn in order to validate data consistency and accuracy across different types of applications.
- It is widely used by **manual testers** for verifying data, and by **SDETs** for automation purposes, such as validation and data setup.
- The differences between **SQL** (structured, relational) and **NoSQL** (flexible, unstructured) databases help QA testers determine the best approach depending on the type of application being tested.
- You can learn about various aspects of database testing including SQL in my following [Database Testing tutorials](#) playlist (I've published 13 videos in it as of date) at <https://www.youtube.com/playlist?list=PLc3SzDYhhiGVVb76aFOH9AcIMNAW-JuXE>



## 2. Basic SQL Concepts

**Question:** What are databases, tables, rows, and columns in the context of SQL?

**Answer:** In SQL, databases, tables, rows, and columns are components used to store and organize data. I've explained these components and shown examples in my [Database Testing tutorial](https://youtu.be/W_fH6CqiTDU) at [https://youtu.be/W\\_fH6CqiTDU](https://youtu.be/W_fH6CqiTDU)

- **Database:** A collection of organized data that can be accessed, managed, and updated. It acts as a container that holds tables and other database objects such as views, indexes, stored procedures and triggers.
- **Table:** A structured set of data that contains rows and columns. It represents a specific entity in the database, such as customers, orders, or products.
- **Row (Record):** Each row in a table represents a single, complete set of data (i.e. a record) for that entity. For example, a row in a users table would represent one individual user's data (name, email, etc.).
- **Column (Field):** A column represents a specific attribute of the entity being modeled. Each column contains data of a particular type, like VARCHAR for text or INT for numbers.

### Examples:

- **Test automation example:** As an SDET, you may be testing an e-commerce system. You need to verify whether product data is correctly inserted into the products table. Below's an example of a table. Each row represents a product (Laptop, Headphones), and columns represent specific attributes of each product (e.g. product\_id, product\_name, price). To check if the Laptop product exists after an API or UI test, you might run:

```
SELECT * FROM products WHERE product_id = 1;
```

### Table products:

product_id	product_name	price	quantity
1	Laptop	899.99	50
2	Headphones	99.99	200

- **Manual Tester Example:** As a manual tester, after performing a transaction, you should confirm if a user record was correctly inserted. The users table may look like below. You could run the SQL to validate the presence of one user with the username given in the SQL query:

```
SELECT * FROM users WHERE username = 'john_doe';
```


#### Table users:


user_id	username	email	registration_date
1	john_doe	<a href="mailto:john@example.com">john@example.com</a>	2023-01-01
2	jane_smith	<a href="mailto:jane@example.com">jane@example.com</a>	2023-02-01

**Question:** What are common SQL data types, and why are they important?

**Answer:** SQL data types define the kind of data that can be stored in a column. Choosing the correct data type ensures data integrity, optimizes storage, and improves query performance.

SQL Data Type	Description	Example
VARCHAR(size)	Variable-length character string. It is used to store text data.	VARCHAR(50) can store text up to 50 characters long.
INT	Integer number. It is used to store whole numbers (e.g. age, quantity).	INT can store numbers like 42 or 1000.
DATE	Used to store calendar dates (year, month, and day).	DATE stores values like 2024-12-30.
DECIMAL(precision, scale)	Stores decimal numbers with exact precision, useful for monetary values.	DECIMAL(10, 2) stores values like 12345.67.

Download for reference  Like  Share  View SQL Queries video  <https://youtu.be/BxMmC77fj9Y>

View [Database Testing videos playlist](#) 

Software and Testing Training YouTube Channel <https://youtube.com/@QA1>

Software Testing Space Blog <https://inderpsingh.blogspot.com/> Copyright © 2024 All Rights Reserved.



Example 1: If you are verifying if the correct data type is used in a table (e.g. prices in a products table should be decimals), you might check the schema (meaning table structure) with the following SQL. There is more in database schema testing, which I've explained in the [database testing tutorial](#) in my [Software and Testing Training](#) channel.

```
DESCRIBE products;
```

```
product_id  INT
product_name VARCHAR(100)
price       DECIMAL(10,2)
quantity    INT
```

Example 2: As a manual tester, you might perform UI-based actions and then check that data is stored with the right types in the database. For example, after entering a date of birth, you may run the following SQL query. If birthdate is stored as DATE, the result would be formatted correctly, e.g. 1990-04-15.

```
SELECT birthdate FROM users WHERE username = 'jane_smith';
```

If you're enjoying learning SQL, please follow or better, connect with me in [LinkedIn](#) at <https://www.linkedin.com/in/inderpsingh/>

**Question:** What are DDL (Data Definition Language) commands, and how are they used?

**Answer:** DDL (Data Definition Language) commands are used to define, modify, and remove database structures such as tables, schemas, and indexes. These commands do not manipulate the data inside the tables but instead manipulate the schema.

### DDL Commands:

- **CREATE:** Used to create a new database object (e.g. table, index).

```
CREATE TABLE employees (  
    employee_id INT PRIMARY KEY,  
    first_name VARCHAR(50),  
    last_name VARCHAR(50),  
    hire_date DATE  
);
```

- **ALTER:** Used to modify an existing database object (e.g. adding a column to a table). Example: ALTER TABLE employees ADD COLUMN salary DECIMAL(10, 2);
- **DROP:** Used to delete a database object. Example: DROP TABLE employees;

#### Examples:

- **SDET Example:** As an SDET, you may need to verify that a new table is created or modified correctly during automated tests. You might validate this with SQL commands such as CREATE and ALTER within your automation suite:

```
CREATE TABLE test_logs (  
    log_id INT PRIMARY KEY,  
    test_name VARCHAR(100),  
    status VARCHAR(20),  
    execution_date DATE  
);
```

- **Manual Tester Example:** While manual testers typically don't create or alter database structures, you may need to confirm that a new table or column exists after a database migration. You might run the following SQL to check that a column like salary has been successfully added: DESCRIBE employees;

**Question:** What are DML (Data Manipulation Language) commands, and how are they used?

**Answer:** DML (Data Manipulation Language) commands are used to manipulate the data within tables. These commands allow testers to retrieve, insert, update, and delete data in the database.

DML Command	Description	Example
SELECT	Retrieves data from one or more tables.	SELECT first_name, last_name FROM employees WHERE hire_date > '2023-01-01';
INSERT	Adds new records into a table.	INSERT INTO employees (employee_id, first_name, last_name, hire_date) VALUES (101, 'Inder', 'P Singh', '2023-01-01');
UPDATE	Modifies existing record in a table.	UPDATE employees SET salary = 50000 WHERE employee_id = 101;
DELETE	Removes record from a table.	DELETE FROM employees WHERE employee_id = 101;

### Examples:

- **SDET Example:** In automated tests, you might run INSERT, UPDATE, and DELETE commands to verify how the system handles various data manipulations. For example, after adding a record to the employees table, you can check that it was inserted:






```
INSERT INTO employees (employee_id, first_name, last_name, hire_date)
VALUES (102, 'Jane', 'Smith', '2024-10-10');
```

```
SELECT * FROM employees WHERE employee_id = 102;
```

- **Manual Tester Example:** As a manual tester, you may run SELECT queries to validate that updates or deletions performed through the application UI are reflected correctly in the database. For example: SELECT \* FROM employees WHERE first\_name = 'Jane';

### Chapter Summary:

- **Databases** store structured data in tables, and each table consists of rows and columns.
- **Data types** ensure data is stored in a structured manner, and the right type is needed for data accuracy.
- **DDL commands** allow testers to define or modify the structure of databases, whereas **DML commands** allow testers to manipulate the actual data.

Download for reference  Like  Share  View SQL Queries video  <https://youtu.be/BxMmC77fj9Y>  
View [Database Testing videos playlist](#) 

Software and Testing Training YouTube Channel <https://youtube.com/@QA1>

Software Testing Space Blog <https://inderpsingh.blogspot.com/> Copyright © 2024 All Rights Reserved.

- In order to dive deeper, you might view my tutorials on [DBMS](#), [database schema](#), [relational algebra](#) and [relational calculus](#) in my [Software and Testing Training](#) channel.

### 3. Writing Simple SQL Queries

**Question:** How do you SELECT data from a single table?

**Answer:** I've demonstrated many SQL queries in my [SQL queries tutorial](#) at <https://youtu.be/BxMmC77fJ9Y> but, put simply, the SELECT statement is used to query data from a single table in a database. It allows testers to retrieve specific columns or all columns from the table.

1st technique: Retrieve specific columns from the table:

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

2nd technique: If you want to retrieve all columns, use \* instead:

```
SELECT * FROM table_name;
```

**Test Automation Example:** Suppose you're testing an e-commerce system, and you want to validate that the products table contains a specific product. You could run the following query into your test automation script to verify the presence and values of specific products:

```
SELECT product_id, product_name, price FROM products;
```

**Manual Testing Example:** If you want to manually check all customer details in a customers table, you can run the following query. will display all columns (like customer\_id, customer\_name, email, etc.) for all customers in the table, which you can visually inspect to verify.

```
SELECT * FROM customers;
```

**Question:** How do you filter data with WHERE clauses?

**Answer:** The WHERE clause is used to filter records based on specific conditions. It narrows down the results to only those rows that meet the defined criteria. Its syntax is:

```
SELECT column1, column2, ... FROM table_name WHERE condition;
```

**Test automation example 1:** If you need to verify that products with a price above \$100 exist in the products table, run the following query. You can use assertions in your automation code to validate the returned data matches your expectations.

```
SELECT product_id, product_name, price FROM products WHERE price > 100;
```

product_id	product_name	price
1	Laptop	899.99

**Manual Testing Example:** If you want to manually check which customers registered after January 1st, 2024, you can run the following query. It will display only customers who registered after the specified date, allowing you to verify the filtering logic manually.

```
SELECT customer_id, customer_name, registration_date FROM customers  
WHERE registration_date > '2024-01-01';
```

**Question:** How do you sort data using ORDER BY?

**Answer:** The ORDER BY clause is used to sort the result set based on one or more columns. By default, it sorts in ascending order (ASC), but you can specify descending order (DESC) instead. The syntax is:

```
SELECT column1, column2, ... FROM table_name ORDER BY column_name  
[ASC|DESC];
```

**Test Automation Example:** If you want to validate that the products are listed in descending order by price in the products table, you can run the query below. The automation code can verify that the prices appear in the correct order as expected.

```
SELECT product_id, product_name, price FROM products ORDER BY price DESC;
```

product_id	product_name	price
1	Laptop	899.99
2	Mouse	25.50

**Manual Testing Example:** If you want to manually view a list of customers sorted by their registration date, run the query below. This will list all customers in chronological order, making it easier for you to validate registration trends.

```
SELECT customer_id, customer_name, registration_date FROM customers  
ORDER BY registration_date ASC;
```

**Question:** How can you limit results using SQL?

**Answer:** LIMIT (used in MySQL and PostgreSQL) or TOP (used in SQL Server) restricts the number of rows returned by a query, which is useful for testing with a subset of data.

Syntax (for MySQL/PostgreSQL):

```
SELECT column1, column2, ... FROM table_name LIMIT number_of_rows;
```

Syntax (for SQL Server):

```
SELECT TOP number_of_rows column1, column2, ... FROM table_name;
```

**Test Automation Example:** If you need to validate that only the top 3 most expensive products are returned, use the following query. This query can let you test pagination or validate specific subsets of data in the application.

```
SELECT product_id, product_name, price FROM products ORDER BY price  
DESC LIMIT 3;
```

Output:

product_id	product_name	price
1	Laptop	899.99
3	Monitor	299.99
4	Keyboard	120.00

**Manual Testing Example:** To manually check only the first 5 customers in the customers table, use the following query. This will show the top 5 rows in the table, allowing for quick data inspection during manual validation.

```
SELECT * FROM customers LIMIT 5;
```

If the system uses SQL Server, the equivalent query would be:

```
SELECT TOP 5 * FROM customers;
```

### Chapter Summary:

- The SELECT statement retrieves data from a single table.
- WHERE filters data based on conditions, and ORDER BY sorts data in ascending or descending order.
- LIMIT or TOP restricts the number of rows returned, making it useful for quick tests with sample data.
- If you are finding my SQL document useful, you can follow me in [LinkedIn](#) for more practical information in test automation and software testing at the link, <https://www.linkedin.com/in/inderpsingh/>

## 4. Working with Joins and Multiple Tables

**Note:** I've explained and demonstrated SQL joins with multiple tables in my SQL queries tutorial at <https://youtu.be/BxMmC77fJ9Y>

**Question:** What are table relationships in SQL, and what are Primary Key and Foreign Key?

**Answer:** In relational databases, table relationships link data across different tables. The Primary Key and Foreign Key maintain these relationships:

- **Primary Key (PK):** A column (or a set of columns) that uniquely identifies each row in a table. It ensures that there are no duplicate or NULL values in that column.
- **Foreign Key (FK):** A column (or a set of columns) in one table that references the Primary Key of another table. It creates a link between two tables.

These keys allow us to join tables and combine data from multiple tables, ensuring data integrity.

**Test automation example:** You may have a customers table and an orders table. The customer\_id column in customers table would be the Primary Key, and the customer\_id column in orders table would be the Foreign Key. You can join these tables to check if each order is associated with a valid customer.

**Manual Testing Example:** A common database test (view my [data testing](#) tutorial) is validating data integrity between tables. For example, when manually testing a students table and a courses table, the student\_id in the courses table should correspond to a valid student\_id in the students table.

**Question:** What are the different types of joins in SQL?

**Answer:** Joins allow you to retrieve data from multiple tables based on related columns. I've explained and demonstrated joins (inner joins, left joins and self joins) in my [SQL tutorial](#) for beginners at <https://www.youtube.com/watch?v=BxMmC77fJ9Y&t=518s>





The most commonly used joins are:

- **INNER JOIN:** Returns only the rows that have matching values in both tables.
- **LEFT JOIN (LEFT OUTER JOIN):** Returns all rows from the left table and the matching rows from the right table. If no match is found, NULL values are returned from the right table.
- **RIGHT JOIN (RIGHT OUTER JOIN):** Returns all rows from the right table and the matching rows from the left table. If no match is found, NULL values are returned from the left table.
- **FULL JOIN (FULL OUTER JOIN):** Returns all rows from both tables, with NULLs where a match doesn't exist in either table.

**Question:** How do you write SQL queries that combine data from multiple tables using joins?

**Answer:** You can write SQL queries that join two or more tables by specifying the relationship between the tables using the join condition (ON). Here are examples of how different joins work.

### Test automation example

Download for reference Like Share View SQL Queries video <https://youtu.be/BxMmC77fj9Y>  
View [Database Testing videos playlist](#)

Software and Testing Training YouTube Channel <https://youtube.com/@QA1>

Software Testing Space Blog <https://inderpsingh.blogspot.com/> Copyright © 2024 All Rights Reserved.

- **INNER JOIN:** Suppose you want to automate the validation of customer orders. You can write a query to ensure that every order has an associated customer. The query below retrieves only the orders that are linked to valid customers.

```
SELECT customers.customer_name, orders.order_id, orders.order_total
FROM customers
INNER JOIN orders ON customers.customer_id = orders.customer_id;
```

- **LEFT JOIN:** If you're testing for customers who haven't placed any orders yet, you would use a LEFT JOIN to include all customers, even those without orders. The query below retrieves all customers, displaying NULL for orders for customers who haven't made any orders.

```
SELECT customers.customer_name, orders.order_id, orders.order_total
FROM customers
LEFT JOIN orders ON customers.customer_id = orders.customer_id;
```

#### Manual testing example:

- **RIGHT JOIN:** If you're testing a healthcare system and want to verify that every patient has a doctor assigned, you would use a RIGHT JOIN to ensure that no patients are missing doctor assignments. The query below returns all patients, including those who may not yet have an assigned doctor.

```
SELECT doctors.doctor_name, patients.patient_name
FROM doctors
RIGHT JOIN patients ON doctors.doctor_id = patients.doctor_id;
```

- **FULL JOIN:** To manually check for any missing relationships between two tables in a database, you would use a FULL JOIN to get all rows from both tables, including rows with no match in either table. This allows you to detect any missing relationships.

```
SELECT a.column1, b.column2
FROM table_a a
FULL JOIN table_b b ON a.id = b.id;
```

You're welcome to take the [free courses](https://inderpsingh.blogspot.com/p/qa-course.html) (5 free courses until now) on my Software Testing Space blog at <https://inderpsingh.blogspot.com/p/qa-course.html>

**Question:** What are some common test scenarios that need SQL joins?

**Answer:** Joins are used to validate data consistency and integrity between related tables.

**Test automation example:**

- Order validation: In e-commerce applications, SDETs can use INNER JOIN queries to validate that every order in the orders table has a valid customer in the customers table.
- Null data validation: Use LEFT JOIN to ensure no orphaned records exist, such as customers without associated orders or transactions.

**Manual testing example:**

- Data consistency checks: When manually testing a student enrollment system, the tester can use a LEFT JOIN to check that all students in the students table are enrolled in at least one course, or use FULL JOIN to check for students or courses that don't match:

Note: For SQL training or [DBMS training](#) you can contact [Inder P Singh](#) in LinkedIn.

```
SELECT students.student_name, enrollments.course_name
FROM students
LEFT JOIN enrollments ON students.student_id = enrollments.student_id;
```

## 5. Intermediate SQL Concepts

**Question:** How can you group data using GROUP BY and filter groups with HAVING?

**Answer:** GROUP BY is used to group rows that have the same values into summary rows, such as totals or counts. It is often combined with aggregate functions like COUNT, SUM, AVG, etc. The HAVING clause is used to filter records after grouping has been applied, typically based on aggregate results. I've demonstrated GROUP BY and HAVING in my [Software and Testing Training](#) channel's [SQL tutorial](#) from [this](#) time stamp in that video.

**Test automation example:** You want to test that customer orders are grouped correctly by customer in an e-commerce system. The following query groups the orders by customer\_id and calculates the total order value for each customer. It then filters groups to show only customers with a total order value above \$500:

```
SELECT customer_id, SUM(order_total) AS total_spent
FROM orders
GROUP BY customer_id
HAVING SUM(order_total) > 500;
```

**Manual testing example:** In a sales application, you might manually test by checking which sales agents have completed more than 10 orders. You might use the GROUP BY and HAVING clauses as follows:

```
SELECT sales_agent, COUNT(order_id) AS total_orders
FROM orders
GROUP BY sales_agent
HAVING COUNT(order_id) > 10;
```

**Exercise:** By now you should know simple SQL queries, joins and some intermediate SQL concepts. You can take the [SQL Test](#) online on my blog, Software Testing Space at <https://inderpsingh.blogspot.com/2011/10/test-your-knowledge-of-sql-queries.html>

**Question:** What are aggregate functions, and how do you use them in SQL?

**Answer:** Aggregate functions perform calculations on multiple rows of data and return a single value. Common aggregate functions include:

- COUNT: Returns the number of rows.
- SUM: Adds up numeric values.
- AVG: Calculates the average value.
- MIN: Returns the smallest value.
- MAX: Returns the largest value.

**Example 1:** You want to test the total number of orders for each product. You can use the COUNT function to retrieve the total orders for each product\_id:

```
SELECT product_id, COUNT(order_id) AS total_orders
FROM orders
GROUP BY product_id;
```

**Example 2:** When testing a library system, you might need to manually check which book has the highest number of borrowings. Using MAX for this shows the book with the highest borrow count.

```
SELECT book_title, MAX(borrow_count) AS most_borrowed FROM books;
```

**Question:** How and when should you use subqueries (nested queries) in SQL?

**Answer:** You can learn about subqueries from my SQL tutorial for beginners from this [timestamp](#). Basically, subqueries are queries inside another query. They are useful when you need to perform a query that relies on the result of another query.

- In the SELECT clause: To calculate derived columns.
- In the WHERE clause: To filter records based on another query's result.
- In the FROM clause: To create a temporary table for further querying.

**Test automation example:** You want to test whether orders with high totals have the highest-paying customers. You could use a subquery in the WHERE clause to find customers whose total order value exceeds the average:

```
SELECT customer_name FROM customers WHERE customer_id IN (SELECT customer_id FROM orders GROUP BY customer_id HAVING SUM(order_total) > (SELECT AVG(order_total) FROM orders));
```

**Manual testing example:** When testing an inventory system, you want to manually check the products with below-average stock levels. You might use a subquery in the WHERE clause:

```
SELECT product_name
FROM products
WHERE stock_quantity < (SELECT AVG(stock_quantity) FROM products);
```

**Question:** How can you combine query results using UNION and INTERSECT?

Answer: I've explained UNION etc. with examples in my SQL tutorial from this [timestamp](#).

- UNION: Combines the results of two queries and removes duplicates.
- UNION ALL: Combines the results of two queries without removing duplicates.
- INTERSECT: Returns only the rows that are common to both queries.

**Example 1:** Suppose you're testing customer data stored in two different tables: active\_customers and inactive\_customers. You want to generate a list of all unique customers across both tables. Use UNION so that all unique customer records are retrieved from both tables:

```
SELECT customer_id, customer_name FROM active_customers
UNION
SELECT customer_id, customer_name FROM inactive_customers;
```

**Example 2:** You're manually testing an educational platform where you need to find students who are enrolled in both Math and Science courses. You can use INTERSECT to validate the students enrolled in both subjects:

```
SELECT student_id, student_name FROM math_students
INTERSECT
SELECT student_id, student_name FROM science_students;
```

**More resources:** If you want to see more examples, you can view them in my Database Testing tutorials [here](#). You can [subscribe](#) to my Software and Testing Training channel to get updates on new tutorials for SDET, QA and manual testers [here](#).

## 6. Advanced SQL Concepts

**Question:** What are Common Table Expressions (CTEs), and how do you use them in complex queries?

**Answer:** A Common Table Expression (CTE) is a temporary result set defined within the execution of a SELECT, INSERT, UPDATE, or DELETE query. CTEs make complex queries more readable by allowing you to define and reuse subqueries.

**Test automation example:** You want to test an e-commerce system to identify high-value customers, those who have placed orders totaling more than \$1000. You can use a CTE to simplify the query as follows. It makes it easy to break the logic into manageable steps for your automation test.

```
WITH high_value_customers AS (  
    SELECT customer_id, SUM(order_total) AS total_spent  
    FROM orders  
    GROUP BY customer_id  
    HAVING SUM(order_total) > 1000  
)  
SELECT customer_id, total_spent FROM high_value_customers;
```

**Manual testing example:** If you need to manually check the products with sales greater than \$5000, you can use a CTE as follows to first aggregate sales data and then select only high-selling products. It helps in reviewing large datasets step-by-step.

```
WITH sales_data AS (  
    SELECT product_id, SUM(sale_amount) AS total_sales  
    FROM sales  
    GROUP BY product_id  
)  
SELECT product_id, total_sales FROM sales_data  
WHERE total_sales > 5000;
```

In fact, the manual tester can specialize in data quality. In my [Data Quality tutorial](#) below, I've explained the data quality concepts and data quality analyst role with many examples.



Data Quality Concepts for Testers: <https://youtu.be/N9olq42z-AE>

**Question:** How do Window Functions like ROW\_NUMBER(), RANK(), LEAD(), and LAG() work?

**Answer:** Window functions perform calculations across a set of table rows related to the current row. They do not collapse rows into groups like aggregate functions but allow the result of a function to be "windowed" over rows.

- ROW\_NUMBER(): Assigns a unique sequential integer to rows.
- RANK(): Similar to ROW\_NUMBER(), but assigns the same rank to rows with equal values.
- LEAD(): Returns the value of the next row.
- LAG(): Returns the value of the previous row.

**Test automation example:** You want to test that customer orders are ranked by total order value. You can use RANK() as follows to identify the top three customers by order amount. It validates that your application correctly ranks top customers.






```
SELECT customer_id, order_total,  
       RANK() OVER (ORDER BY order_total DESC) AS rank  
FROM orders  
WHERE rank <= 3;
```

**Manual testing example:** To check the order history and identify sequential patterns, you might use LAG() to compare current and previous order dates for each customer. It helps in manually reviewing customer behavior and order trends.

```
SELECT customer_id, order_date,  
       LAG(order_date, 1) OVER (PARTITION BY customer_id ORDER BY  
order_date) AS previous_order  
FROM orders;
```

**Question:** How do you create and use VIEWS for testing purposes?

**Answer:** I've explained how to test database schema objects like TABLE, VIEW, STORED PROCEDURE AND TRIGGERS in my [Database Testing tutorial](#) from [this](#) time stamp. But, as far as a VIEW is concerned, it's a virtual table that consists of a SQL query result. It simplifies complex queries by allowing you to encapsulate them within a reusable object.

Download for reference  Like  Share  View SQL Queries video  <https://youtu.be/BxMmC77fj9Y>  
View [Database Testing videos playlist](#) 

Software and Testing Training YouTube Channel <https://youtube.com/@QA1>

Software Testing Space Blog <https://inderpsingh.blogspot.com/> Copyright © 2024 All Rights Reserved.

**Test automation example:** You need to repeatedly test data on active customer orders. Instead of writing complex queries in each automated test, you can create a VIEW for active orders:

```
CREATE VIEW active_orders AS
SELECT order_id, customer_id, order_total
FROM orders
WHERE order_status = 'Active';
```

You can test this view with a simple SQL query:

```
SELECT * FROM active_orders;
```

**Manual testing example:** In an inventory system, to test low-stock products regularly, create a VIEW like below:

```
CREATE VIEW low_stock_products AS
SELECT product_id, product_name, stock_quantity
FROM products
WHERE stock_quantity < 10;
```

You can now manually check low stock levels using:

```
SELECT * FROM low_stock_products;
```

**Question:** How do you use indexes and optimize queries for large datasets?

**Answer:** Indexes improve the speed of data retrieval by creating a data structure (index) that allows the **DBMS** to find rows more quickly. However, they can slow down INSERT, UPDATE, and DELETE operations.

**Test automation example:** When testing an application that retrieves customer records, performance may degrade as the dataset grows. You can optimize a query using an index on the customer\_id column:

```
CREATE INDEX idx_customer_id ON customers (customer_id);
```

Now your following SQL query will run faster, especially with large datasets:

```
SELECT * FROM customers WHERE customer_id = 123;
```

**Manual testing example:** If manually retrieving product data is slow, you can save your time by adding an index on the product\_name column. It will help improve query speed when searching for product names.

```
CREATE INDEX idx_product_name ON products (product_name);
```

**Question:** How do you handle SQL transactions with COMMIT, ROLLBACK, and SAVEPOINT?

**Answer:** A transaction is a sequence of operations performed as a single logical unit of work. Transactions ensure that either all operations succeed or none at all (atomicity).

- COMMIT: Saves all changes made in the transaction.
- ROLLBACK: Reverts the changes made in the transaction.
- SAVEPOINT: Sets a point within a transaction to which you can roll back partially.

**Test automation example:** You're testing a banking application where funds are transferred between accounts. You use a transaction to ensure that both debit and credit actions occur together:

```
BEGIN TRANSACTION;  
UPDATE accounts SET balance = balance - 100 WHERE account_id = 1;  
UPDATE accounts SET balance = balance + 100 WHERE account_id = 2;  
COMMIT;
```

If there's an issue during the transfer, you can roll back using the following SQL query:

```
ROLLBACK;
```

**Manual testing example:** When testing an inventory system, you may use transactions to ensure that updating stock quantities is atomic. This helps to maintain your test environment's data integrity by ensuring that the stock update is not partially completed:

```
BEGIN TRANSACTION;  
UPDATE products SET stock_quantity = stock_quantity - 1 WHERE product_id = 101;  
-- Error occurs here  
ROLLBACK;
```

## 7. SQL in Different Database Platforms






**Question:** How is SQL used in Oracle, and what are PL/SQL, cursors, and stored procedures?

**Answer:** In Oracle databases, SQL is commonly used with PL/SQL (Procedural Language/SQL), which extends SQL with procedural capabilities like loops, conditionals, and exception handling. This makes PL/SQL suitable for writing complex scripts.

- PL/SQL: A procedural extension of SQL that allows the use of variables, conditions, loops, and error handling in SQL queries.
- Cursors: A pointer that fetches rows one-by-one from a query result set.
- Stored Procedures: Named blocks of PL/SQL code that can be reused and executed on demand.

**Test automation example:** You are testing a financial system that calculates interest for each customer based on their account balance. You can write a stored procedure like the following to automate this calculation. This procedure can be run by an automation script that calls the stored procedure and validates updated account balances.

```
CREATE OR REPLACE PROCEDURE calculate_interest IS
    CURSOR customer_cursor IS
        SELECT customer_id, account_balance FROM customers;
    customer_row customer_cursor%ROWTYPE;
BEGIN
    OPEN customer_cursor;
    LOOP
        FETCH customer_cursor INTO customer_row;
        EXIT WHEN customer_cursor%NOTFOUND;
        -- Logic to calculate and update interest
        UPDATE customers
        SET account_balance = account_balance * 1.05
        WHERE customer_id = customer_row.customer_id;
    END LOOP;
    CLOSE customer_cursor;
END;
```

Download for reference  Like  Share  View SQL Queries video  <https://youtu.be/BxMmC77fj9Y>  
View Database Testing videos playlist 

Software and Testing Training YouTube Channel <https://youtube.com/@QA1>

Software Testing Space Blog <https://inderpsingh.blogspot.com/> Copyright © 2024 All Rights Reserved.

**Manual testing example:** You might want to retrieve a list of all products and their stock levels using the following PL/SQL block. You would then manually validate the product stock levels displayed in the output.

```
DECLARE

    CURSOR product_cursor IS

        SELECT product_name, stock_quantity FROM products;

    product_row product_cursor%ROWTYPE;

BEGIN

    OPEN product_cursor;

    LOOP

        FETCH product_cursor INTO product_row;

        EXIT WHEN product_cursor%NOTFOUND;

        DBMS_OUTPUT.PUT_LINE(product_row.product_name || ' has ' ||
product_row.stock_quantity || ' items in stock.');
```

**Question:** What are PostgreSQL-specific features, like the RETURNING clause?

**Answer:** PostgreSQL has some unique SQL features, one of which is the RETURNING clause. This allows you to retrieve data directly after performing an INSERT, UPDATE, or DELETE operation without the need for an additional SELECT query.

**Test automation example:** You want to insert a new order in the orders table and immediately retrieve the generated order\_id for further automation steps. Using the RETURNING clause, you can capture the order\_id as part of your automated test case and validate it in the same step:

```
INSERT INTO orders (customer_id, order_total)
VALUES (101, 500)
RETURNING order_id;
```

**Manual testing example:** You may want to update customer information and retrieve the updated row to test if changes were applied:

```
UPDATE customers
SET customer_name = 'John Doe'
WHERE customer_id = 101
RETURNING customer_id, customer_name;
```

**Question:** What is T-SQL? How do transactions and error handling work in SQL Server?

**Answer:** T-SQL (Transact-SQL) is the procedural extension of SQL used in SQL Server. It provides control-of-flow language, such as BEGIN, END, IF, WHILE, and error handling mechanisms like TRY...CATCH blocks.

- Transactions in T-SQL ensure atomicity. If all queries in a transaction complete successfully, the changes are committed. If there's an error, you can roll back the transaction.
- Error handling is done using TRY...CATCH, which allows you to gracefully handle SQL errors.

**Test automation example:** You're automating a test that simulates multiple fund transfers in a banking system. T-SQL transactions ensure either all transfers succeed or none. In your automation script, you can validate that both UPDATE operations completed successfully.

```
BEGIN TRANSACTION;
```

```
BEGIN TRY
```

```
    UPDATE accounts SET balance = balance - 100 WHERE account_id = 1;
```

```
    UPDATE accounts SET balance = balance + 100 WHERE account_id = 2;
```

```
    COMMIT;
```

```
END TRY
```

```
BEGIN CATCH
```

```
    ROLLBACK;
```

```
    THROW; -- Re-throws the caught error for logging or reprocessing
```

```
END CATCH;
```

**Manual testing example:** I've explained [database migration testing](#) typically, ETL (Extract-Transform-Load) testing in my tutorial [here](#). For manual database testing, you might manually review transactions to test proper error handling during a bulk update of product prices:

```
BEGIN TRANSACTION;
```

```
BEGIN TRY
```

```
    UPDATE products SET price = price * 1.10 WHERE category = 'Electronics';
```

```
    COMMIT;
```

```
END TRY
```

```
BEGIN CATCH
```



```
    ROLLBACK;
```


```
    PRINT 'Error occurred. Transaction rolled back.';
```

```
END CATCH;
```

**Question:** How is SQL used in NoSQL databases like MongoDB, and what are the basic MongoDB queries using find and aggregate functions?

**Answer:** MongoDB is a NoSQL database, but it uses a query language that resembles SQL in certain aspects. Instead of tables and rows, MongoDB stores data in collections and

Download for reference  Like  Share  View SQL Queries video  <https://youtu.be/BxMmC77fj9Y>

View [Database Testing videos playlist](#) 

Software and Testing Training YouTube Channel <https://youtube.com/@QA1>

Software Testing Space Blog <https://inderpsingh.blogspot.com/> Copyright © 2024 All Rights Reserved.



documents ([JSON](#)-like structures). The find function is similar to SQL's SELECT, and aggregate is used for more complex queries and transformations.

- find: Retrieves documents from a collection.
- aggregate: Performs aggregation operations like grouping, filtering, and sorting.

**Test automation example:** For automated tests, you need to retrieve a specific customer's orders in a MongoDB collection. The query below is similar to SQL's SELECT \* FROM orders WHERE customer\_id = 101. You would assert the results within the automated test.

```
db.orders.find({ customer_id: 101 });
```

For more complex aggregation, such as calculating total sales for each product:

```
db.orders.aggregate([
  { $group: { _id: "$product_id", total_sales: { $sum: "$order_total" } } }
]);
```

**Example 2 (Manual testing):** In manual testing, you might want to retrieve all orders placed in the last 7 days using MongoDB's find:

```
db.orders.find({ order_date: { $gte: new Date(Date.now() - 7*24*60*60*1000) } });
```

For manually testing sales grouped by product, the aggregate function would be used similarly to SQL's GROUP BY:

```
db.orders.aggregate([
  { $match: { order_date: { $gte: new Date("2024-10-01") } } },
  { $group: { _id: "$product_id", total_sales: { $sum: "$order_total" } } }
]);
```

## 8. SQL Queries for Manual Testers

**Note:** If you're a beginner in database testing, I've explained and demonstrated simple SQL queries in my Database Testing and SQL [tutorial](https://youtu.be/W_fH6CqiTDU) at [https://youtu.be/W\\_fH6CqiTDU](https://youtu.be/W_fH6CqiTDU)

**Question:** In manual testing, how can you validate data integrity in databases using SQL?

**Answer:** To validate data integrity in the test environment database, manual testers can write SQL queries to ensure that the data conforms to certain constraints, such as primary keys, foreign keys, and unique values. Data integrity checks include testing for no duplication, correct references between tables, and accurate data types.

**Example:** You are testing a customer management system where each customer should have a unique email address. The following query helps you identify any duplicate email addresses, ensuring uniqueness in the email column. **Tip:** Use GROUP BY and HAVING to find data integrity issues like duplicates and foreign key violations.

```
SELECT email, COUNT(*)  
FROM customers  
GROUP BY email  
HAVING COUNT(*) > 1;
```

**Question:** How can manual testers test CRUD operations with SQL?

**Answer:** Testing the CRUD operations (Create, Read, Update, Delete) is a part of data testing. Manual testers use SQL queries to execute and validate these operations within the test environment database.

- Create (INSERT): Manually insert a new record and verify it.
- Read (SELECT): Retrieve and confirm data is correctly stored.
- Update (UPDATE): Modify a record and check the updated values.
- Delete (DELETE): Remove a record and verify it's no longer in the database.

**Example:**

**Insert Test:** After running the following INSERT SQL query, you can validate the insertion with: `SELECT * FROM orders WHERE order_id = 101;`

```
INSERT INTO orders (order_id, customer_id, order_total)
VALUES (101, 5, 250);
```

Update Test: After running the following UPDATE SQL query, you can validate the updated total with: `SELECT order_total FROM orders WHERE order_id = 101;`

```
UPDATE orders
SET order_total = 300
WHERE order_id = 101;
```

Delete Test: You should be careful while running a Delete test, because the deleted data won't be available post deletion. After running the following DELETE SQL query, you can validate the deletion by getting no rows for: `SELECT * FROM orders WHERE order_id = 101;`

```
DELETE FROM orders WHERE order_id = 101;
```

**Question:** What are practical SQL examples for verifying test data in QA environments?






**Answer:** Practical SQL queries allow manual testers to confirm if the test data in QA environment meets expectations. These queries help validate business rules, relationships, and calculations in the test data.

**Example:**

- Validate a range of orders for a specific customer: This query helps validate that the orders for a customer within a specific date range have the correct totals.

```
SELECT order_id, order_total
FROM orders
WHERE customer_id = 10 AND order_date BETWEEN '2024-01-01' AND '2024-12-31';
```

- **Check product prices after an update:** If a product price update is executed, you can validate the correct pricing using the following SQL query. This query retrieves all products whose price exceeds \$100, ensuring the update succeeded. **Tip:** For complex queries, test various scenarios such as edge cases (e.g. null values and negative values) to test if the system handles them correctly.

Download for reference  Like  Share  View SQL Queries video  <https://youtu.be/BxMmC77fj9Y>  
View [Database Testing videos playlist](#) 

Software and Testing Training YouTube Channel <https://youtube.com/@QA1>

Software Testing Space Blog <https://inderpsingh.blogspot.com/> Copyright © 2024 All Rights Reserved.

```
SELECT product_name, price
FROM products
WHERE price > 100;
```

**Question:** How can manual testers write validation queries for testing processes?

**Answer:** Validation queries are used to test if the application is processing data correctly. Manual testers write SQL queries to check that inputs, outputs, and transformations comply with the expected results.

**Example:** You are testing an e-commerce application where discounts are applied to orders over a certain amount. After applying a discount rule, you can validate it with the following SQL query. This query shows if all orders above \$500 have the discount applied correctly.

**Tip:** Test validation queries cover both positive and negative test cases to test if expected data is processed while invalid data is rejected or handled appropriately.

```
SELECT order_id, order_total, discount_applied
FROM orders
WHERE order_total > 500 AND discount_applied = TRUE;
```

## 9. SQL in Automation Testing

**Question:** How can SQL queries be integrated into automated test scripts in languages like Java or Python?

**Answer:** SQL queries can be integrated into automated test scripts to validate data in the database after performing operations via API or UI automation. In Java, SQL queries are usually executed using JDBC, while Python uses database-specific libraries such as psycopg2 (for PostgreSQL) or sqlite3. **Tip:** Always handle database connections with proper resource management (closing connections after execution) to prevent memory leaks in automation tests.

**Test automation example 1**(Java with JDBC):

```
// Import JDBC classes
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;






public class DatabaseTest {
    public static void main(String[] args) {
        try {
            // Connect to the database
            Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/test_db", "

            // Create a statement object
            Statement stmt = conn.createStatement();

            // Execute a SQL query
            ResultSet rs = stmt.executeQuery("SELECT * FROM users WHERE user_id = 101");

            // Process the result set
            while (rs.next()) {
                System.out.println("Username: " + rs.getString("username"));
            }

            // Close connections
            rs.close();
            stmt.close();
            conn.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Download for reference  Like  Share  View SQL Queries video  <https://youtu.be/BxMmC77fj9Y>  
View Database Testing videos playlist 

Software and Testing Training YouTube Channel <https://youtube.com/@QA1>

Software Testing Space Blog <https://inderpsingh.blogspot.com/> Copyright © 2024 All Rights Reserved.

## Test automation example 2 (Python with psycopg2):

```
import psycopg2

def fetch_data():
    try:
        # Connect to the PostgreSQL database
        conn = psycopg2.connect(
            host="localhost",
            database="test_db",
            user="user",
            password="password"
        )
        cursor = conn.cursor()

        # Execute a SQL query
        cursor.execute("SELECT * FROM users WHERE user_id = 101")

        # Fetch and print the result
        user = cursor.fetchone()
        print("Username:", user[1]) # Assuming username is in the second column

        # Close connections
        cursor.close()
        conn.close()
    except Exception as e:
        print("Error:", e)

fetch_data()
```

**Question:** How can JDBC and database libraries be used in API and UI automation testing?

**Answer:** In API and UI automation, JDBC (Java) and equivalent database libraries in other languages are used to directly interact with the database during testing. For example, you might use SQL queries to validate that an API correctly inserted or updated data in a table or that the UI reflects correct data in the database after performing an action.

## Test automation example (Selenium + JDBC):

```

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;

public class UITestWithSQL {
    public static void main(String[] args) throws Exception {
        WebDriver driver = new ChromeDriver();
        driver.get("https://example.com/login");

        // Perform UI actions to add a new user
        // ...

        // Verify data in the database after UI action
        Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/test_db",
        Statement stmt = conn.createStatement();
        ResultSet rs = stmt.executeQuery("SELECT * FROM users WHERE user_id = 102");

        if (rs.next()) {
            System.out.println("New user added: " + rs.getString("username"));
        }

        // Cleanup
        rs.close();
        stmt.close();
        conn.close();
        driver.quit();
    }
}

```

**Question:** How can you write SQL assertions within Selenium or Rest Assured automation frameworks?

**Answer:** Assertions validated expected results in automation testing. SQL assertions can be integrated into frameworks such as Selenium and Rest Assured by querying the database and using the results for validation. **Tip:** Use database assertions to ensure that API or UI actions have persisted the expected data, adding another layer of verification beyond frontend or API responses.

**Note:** In order to expand your professional network, you're welcome to connect with me (Inder P Singh) in [LinkedIn](https://www.linkedin.com/in/inderpsingh/) at <https://www.linkedin.com/in/inderpsingh/>

## Test automation example (Rest Assured + SQL Assertion):

```
import io.restassured.RestAssured;
import io.restassured.response.Response;
import java.sql.*;

public class APITestWithSQLAssertion {
    public static void main(String[] args) throws SQLException {
        // Send an API request
        Response response = RestAssured.get("https://api.example.com/user/101");
        int statusCode = response.getStatusCode();

        // Assert response status
        assert statusCode == 200;

        // Verify user exists in the database
        Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/test_db", '
        Statement stmt = conn.createStatement();
        ResultSet rs = stmt.executeQuery("SELECT * FROM users WHERE user_id = 101");

        // Assert database data matches
        if (rs.next()) {
            assert rs.getString("username").equals("testuser");
        }

        // Cleanup
        rs.close();
        stmt.close();
        conn.close();
    }
}
```

**Question:** How can you use SQL be for automating data setup and teardown in test environments?

**Answer:** Automating the setup and teardown of test data prepares a clean and consistent environment for each test execution. SQL scripts can be used to insert necessary data before a test runs and remove or reset the data afterward, ensuring tests remain isolated and reproducible.

## Test automation example (Data setup and teardown in Java):



```

public class TestDataManagement {
    // Method to set up data before tests
    public static void setUpTestData() throws SQLException {
        Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/test_db",
            Statement stmt = conn.createStatement();

        // Insert test data
        stmt.executeUpdate("INSERT INTO users (user_id, username) VALUES (101, 'testuser')");

        stmt.close();
        conn.close();
    }

    // Method to clean up data after tests
    public static void tearDownTestData() throws SQLException {
        Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/test_db",
            Statement stmt = conn.createStatement();

        // Delete test data
        stmt.executeUpdate("DELETE FROM users WHERE user_id = 101");

        stmt.close();
        conn.close();
    }
}

```

In your test automation framework (like [JUnit](#) or [TestNG](#)), you can call these methods in the `@BeforeTest` and `@AfterTest` hooks to automate the data setup and teardown. **Tip:** Automating data setup and teardown creates test isolation, preventing data dependency between tests and leading to more reliable outcomes.

**Note:** In the above screenshots, you'll notice that the statements creating the Connection objects are truncated on the right. You'll need to add the username and password arguments to those statements.

## 10. SQL for API Testing

**Question:** How can you use SQL to validate the database state after API calls?

**Answer:** After an API call that performs a create, update, or delete operation, you can use SQL to validate the changes in the database. This tests if the API correctly interacts with the database and that data integrity is maintained.

Resources: API Testing [short tutorial](#), Postman [short tutorial](#), REST Assured [short tutorial](#), SoapUI API Testing [tutorials playlist](#)

**Test automation example** (Validating API state with SQL): Tip: Use SQL queries to validate both the existence and the integrity of the data after an API call.

```
// Perform API request
Response response = RestAssured.post("https://api.example.com/users/create");

// Validate API response
assert response.getStatusCode() == 201;






// Query the database to verify that the new user was added
Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/test_db", '
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM users WHERE username = 'newuser'");

assert rs.next(); // User should exist in the database after API call
assert rs.getString("email").equals("newuser@example.com"); // Validate user data

// Cleanup
rs.close();
stmt.close();
conn.close();
```

**Question:** What are some practical SQL queries for checking API response data?

**Answer:** When testing APIs, SQL queries can be used to compare the actual database values with the results returned by the API. This tests if the API responses match what is stored in the database.

Download for reference  Like  Share  View SQL Queries video  <https://youtu.be/BxMmC77fj9Y>  
View Database Testing videos playlist 

Software and Testing Training YouTube Channel <https://youtube.com/@QA1>

Software Testing Space Blog <https://inderpsingh.blogspot.com/> Copyright © 2024 All Rights Reserved.

### Test automation example (SQL to check API response consistency):

```
// Perform API GET request
Response response = RestAssured.get("https://api.example.com/users/101");
String apiUsername = response.jsonPath().getString("username");

// Query the database
Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/test_db", "user", "pass");
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT username FROM users WHERE user_id = 101");

if (rs.next()) {
    String dbUsername = rs.getString("username");
    assert dbUsername.equals(apiUsername); // Validate that API response matches database value
}

// Cleanup
rs.close();
stmt.close();
conn.close();
```

**Question:** How can you write SQL for API testing scenarios, like checking database consistency?

**Answer:** To test database consistency after API calls, SQL can be used to run multiple validations. For example, after creating a new entity via an API, you might verify if related tables were updated correctly, or if constraints like foreign keys are being enforced.

**Question:** How can you integrate SQL with API automation tools for end-to-end tests?

**Answer:** SQL can be integrated into end-to-end API automation tests to validate the entire flow of data, from frontend API interaction to backend database updates. Tools like Rest Assured (for Java) or requests (for Python) can be integrated with SQL queries to test the API response and the resulting database state.

### Test automation example (End-to-end test using API and SQL):

```

// Perform API call to create a new user
Response createResponse = RestAssured.post("https://api.example.com/users/create");
assert createResponse.getStatusCode() == 201;

// Verify database state
Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/test_db",
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM users WHERE username = 'newuser'");

assert rs.next(); // Ensure user was created in the database
assert rs.getString("email").equals("newuser@example.com");

// Perform API call to fetch user data
Response getResponse = RestAssured.get("https://api.example.com/users/101");
String apiUsername = getResponse.jsonPath().getString("username");

// Validate that the API response matches the database
assert rs.getString("username").equals(apiUsername);

// Cleanup
rs.close();
stmt.close();
conn.close();

```

## 11. SQL Queries for Performance Testing

**Question:** What techniques can you use to optimize SQL queries for performance?

**Answer:** Optimizing SQL queries is needed for higher performance such that the database can handle high loads efficiently. Key techniques include:

- Indexing: Adding indexes to frequently queried columns can improve query performance greatly.
- Query Refactoring: Use efficient query patterns such as avoiding SELECT \* and minimizing subqueries, where possible.
- Batch Processing: Use bulk inserts or updates to process large amounts of data in batches rather than performing row-by-row operations.
- EXPLAIN Plan: Use the EXPLAIN command to analyze and optimize the query execution plan.
- Apply indexes to columns involved in WHERE, JOIN, or ORDER BY clauses for faster retrieval of records.

**Test automation example** (Optimizing SQL query using index):

```
// Execute a performance test with a query that uses an index
String sql = "SELECT employee_name FROM employees WHERE department_id = ?";
PreparedStatement pstmt = conn.prepareStatement(sql);
pstmt.setInt(1, 10);
ResultSet rs = pstmt.executeQuery();

// Measure and log the execution time
long startTime = System.currentTimeMillis();
while (rs.next()) {
    System.out.println(rs.getString("employee_name"));
}
long endTime = System.currentTimeMillis();
System.out.println("Query Execution Time: " + (endTime - startTime) + "ms");
```

**Question:** How can you handle large datasets in performance testing?

**Answer:** When working with large datasets, make sure that the queries can handle the volume of data efficiently without timing out or causing performance degradation. These techniques include:

- Partitioning: Break down large datasets into partitions to make queries faster.
- Pagination: Use SQL LIMIT/OFFSET to fetch data in smaller chunks instead of retrieving all records at once.
- Parallel Queries: For databases supporting parallel processing, enable parallel execution to speed up query processing.

**Test automation example** (Handling large datasets with pagination): Use pagination to break down the retrieval of large datasets into manageable chunks to prevent memory overload during performance tests.

```
// Query large dataset using pagination
int offset = 0;
int limit = 100;
String sql = "SELECT employee_name FROM employees ORDER BY employee_id LIMIT ? OFFSET ?";
PreparedStatement pstmt = conn.prepareStatement(sql);
pstmt.setInt(1, limit);
pstmt.setInt(2, offset);

ResultSet rs = pstmt.executeQuery();

// Process first batch of results
while (rs.next()) {
    System.out.println(rs.getString("employee_name"));
}

// Fetch the next batch of data
offset += limit;
pstmt.setInt(2, offset);
rs = pstmt.executeQuery();
```

**Question:** How can you use SQL to identify bottlenecks in database queries?

**Answer:** SQL performance bottlenecks can be identified by analyzing the query execution plan and measuring the time taken for each part of the query to execute. The following techniques help identify slow-running queries:

- **EXPLAIN Plan Analysis:** Use the EXPLAIN command to get insights into how a query is executed, including which indexes are used, which tables are scanned, and the cost of each step.
- **Query Execution Time Measurement:** Measure the query execution time for each step or part of a query.
- **Indexes and Joins Evaluation:** Determine if missing indexes or inefficient joins are causing bottlenecks.

**Test automation example** (Using EXPLAIN plan to identify bottlenecks): Use EXPLAIN regularly to understand the execution plan and spot potential bottlenecks caused by full table scans or inefficient joins.

// Analyze query performance with EXPLAIN example by Inder P Singh

```
String sql = "EXPLAIN SELECT employee_name FROM employees e JOIN
departments d ON e.department_id = d.department_id WHERE
d.department_name = 'HR'";
```

```
Statement stmt = conn.createStatement();
```

```
ResultSet rs = stmt.executeQuery(sql);
```

```
while (rs.next()) {
```

```
    System.out.println("Query Step: " + rs.getString(1));
```

```
}
```






// Based on EXPLAIN result, add an index to optimize the join condition

```
String indexSql = "CREATE INDEX idx_department_id ON
employees(department_id)";
```

```
stmt.execute(indexSql);
```

**Question:** What are a few practical SQL examples for testing query execution time?

**Answer:** Measuring query execution time is typically done in performance testing to test if the queries are optimized and run within acceptable time limits. You can use SQL commands like NOW(), database-specific functions, or performance testing tools to measure query execution time.

Download for reference  Like  Share  View SQL Queries video  <https://youtu.be/BxMmC77fj9Y>  
View [Database Testing videos playlist](#) 

Software and Testing Training YouTube Channel <https://youtube.com/@QA1>

Software Testing Space Blog <https://inderpsingh.blogspot.com/> Copyright © 2024 All Rights Reserved.

**Test automation example** (Measuring query execution time in Java): **Tip:** Alternatively, you can use the database's built-in functions or performance testing frameworks to benchmark query execution times. Set thresholds to alert when execution time exceeds the expected limits.

```
// Start time before query execution
long startTime = System.currentTimeMillis();

// Execute query
String sql = "SELECT COUNT(*) FROM large_table WHERE condition = ?";
PreparedStatement pstmt = conn.prepareStatement(sql);
pstmt.setString(1, "active");
ResultSet rs = pstmt.executeQuery();

// End time after query execution
long endTime = System.currentTimeMillis();

// Log query execution time
System.out.println("Query Execution Time: " + (endTime - startTime) + "ms");
```



## 12. Data Validation Using SQL

**Question:** How can you use SQL to validate data accuracy between UI and backend databases?

**Answer:** Firstly, you should be aware of various types of data quality issues, that I've explained in my [Data Quality tutorial](#).

Secondly, SQL can be used to compare data displayed in the UI with the actual data stored in the backend database. This tests if the UI reflects the correct values stored in the database and that no discrepancies exist due to system bugs, synchronization issues, or data corruption.

**Manual testing example:** Suppose a web application displays a user's profile information. You can retrieve the user data from the UI and the backend using SQL for comparison:

UI Data:

User's profile shows: Name: Inder P Singh, Profile link:

<https://www.linkedin.com/in/inderpsingh/>

SQL Query: `SELECT name, profile_link FROM users WHERE user_id = 1;`

If both match, the data is accurate. If there's a mismatch, further investigation is needed.

**Test automation example:** In automated tests, you can use SQL as follows to validate UI data with backend values programmatically. Validate data across multiple fields to ensure full data accuracy between the UI and backend.

```

// Fetch user details from UI
String uiName = driver.findElement(By.id("username")).getText();
String uiEmail = driver.findElement(By.id("email")).getText();

// Fetch user details from database
String sql = "SELECT name, email FROM users WHERE user_id = ?";
PreparedStatement pstmt = conn.prepareStatement(sql);
pstmt.setInt(1, 1001);
ResultSet rs = pstmt.executeQuery();

if (rs.next()) {
    String dbName = rs.getString("name");
    String dbEmail = rs.getString("email");

    // Assert UI data matches DB data
    assertEquals(uiName, dbName);
    assertEquals(uiEmail, dbEmail);
}

```






**Question:** How can you write SQL queries for data validation in E2E or end-to-end tests?

**Answer:** End-to-end (E2E) testing validates the complete flow of data from the frontend to the backend. SQL queries help ensure that after actions like form submissions, database updates, or transactions, the correct data has been saved in the backend.

**Manual testing example** (Order creation validation): If a user places an order through the UI, you can validate the order data in the database. After placing the order in the UI, compare the order details in the UI with the corresponding record in the orders table to test if the data is stored correctly.

```
SELECT order_id, product_name, quantity FROM orders WHERE user_id = 1001
AND status = 'placed';
```

**Test automation example** (E2E order validation): Use SQL in E2E tests to validate data changes after significant user actions like form submissions, updates, or deletions.

Download for reference  Like  Share  View SQL Queries video  <https://youtu.be/BxMmC77fj9Y>  
View [Database Testing videos playlist](#) 

Software and Testing Training YouTube Channel <https://youtube.com/@QA1>

Software Testing Space Blog <https://inderpsingh.blogspot.com/> Copyright © 2024 All Rights Reserved.

```
// Place order via API or UI
driver.findElement(By.id("placeOrderButton")).click();

// Fetch order data from DB for validation
String sql = "SELECT order_id, product_name, quantity FROM orders WHERE user_id = ? AND
PreparedStatement pstmt = conn.prepareStatement(sql);
pstmt.setInt(1, 1001);
pstmt.setString(2, "placed");
ResultSet rs = pstmt.executeQuery();

// Verify DB data matches expected order
if (rs.next()) {
    assertEquals("Laptop", rs.getString("product_name"));
    assertEquals(1, rs.getInt("quantity"));
}
```

**Question:** How can you automate data consistency checks with SQL in QA environments?

**Answer:** Data consistency checks test if the data remains consistent across different parts of the application and database. In the QA environment, SQL queries can be integrated into automated test scripts to regularly check for data integrity and consistency issues.

**Test automation example** (Consistency check for user profile data):

```

// Fetch data from the UI
String uiName = driver.findElement(By.id("username")).getText();
String uiEmail = driver.findElement(By.id("email")).getText();

// SQL query to check data consistency in the backend
String sql = "SELECT name, email FROM users WHERE user_id = ?";
PreparedStatement pstmt = conn.prepareStatement(sql);
pstmt.setInt(1, 1001);
ResultSet rs = pstmt.executeQuery();

// Validate UI data against backend data
if (rs.next()) {
    String dbName = rs.getString("name");
    String dbEmail = rs.getString("email");

    assertEquals(uiName, dbName);
    assertEquals(uiEmail, dbEmail);
}

```

Manual testing example (**Cross-environment consistency check**): Suppose your application replicates data between environments (like staging and production). You can manually run SQL queries to test if data is consistent across both environments:

- Staging Query: `SELECT name, email FROM users WHERE user_id = 1001;`
- Production Query (same as Staging Query):

```
SELECT name, email FROM users WHERE user_id = 1001;
```

**Tip:** Regularly schedule automated scripts to run SQL-based data consistency checks to detect any issues early in the QA cycle.

## 13. SQL Questions for QA Interview Preparation

**Question:** What's the difference between INNER JOIN and LEFT JOIN?

**Answer:** INNER JOIN: Returns only the rows that have matching values in both tables.

LEFT JOIN: Returns all rows from the left table, and the matched rows from the right table. If there's no match, the result is NULL on the right side.

**Question:** How would you write a SQL query to find duplicate records in a table?

**Answer:** You can use GROUP BY and HAVING to find duplicate records. The following query groups rows by the column and returns only those with more than one occurrence:

```
SELECT column_name, COUNT(*)  
FROM table_name  
GROUP BY column_name  
HAVING COUNT(*) > 1;
```

**Question:** What is the use of the GROUP BY clause?

**Answer:** GROUP BY is used to group rows that have the same values in specified columns. It is often used with aggregate functions like COUNT, SUM, AVG, etc.

**Question:** How would you optimize a slow-running SQL query?

**Answer:** Use indexes on columns that are frequently filtered or joined. Avoid using SELECT \*, and only retrieve required columns. Use EXPLAIN to analyze query execution and identify bottlenecks. Consider query restructuring (e.g. breaking complex queries into smaller ones).

**Question:** What's the difference between HAVING and WHERE clauses?

**Answer:** WHERE: Filters rows before grouping, used to filter individual rows.

HAVING: Filters groups after GROUP BY, used to filter aggregated data.

**Question:** What are transactions and how COMMIT and ROLLBACK work.

**Answer:** A transaction is a sequence of SQL operations executed as a single unit of work. If any part of the transaction fails, the entire transaction is rolled back.

- COMMIT: Saves all the changes made during the transaction.
- ROLLBACK: Reverts the changes made during the transaction.

**Question:** Write a query to select the second highest salary from a table of employees.

Answer: You can use a subquery to find the second highest salary:

```
SELECT MAX(salary)
FROM employees
WHERE salary < (SELECT MAX(salary) FROM employees);
```

## 14. SQL Best Practices for Testers

**Question:** How do you write reusable and maintainable SQL queries for testing?

**Answer:**

- Use meaningful table and column aliases: This improves readability, especially in complex queries.
- Parameterize queries: Avoid hardcoding values; use variables or placeholders to make queries reusable.
- Comment your queries: Add comments to explain complex logic or reasoning behind certain conditions.
- Organize queries in modular formats: Break down large queries into smaller, logical parts (e.g. using Common Table Expressions or subqueries) for easier maintenance.

**Example:** It uses a WITH clause to break the query into smaller parts, making it more maintainable and modular.

```
WITH filtered_data AS (  
    SELECT employee_id, salary  
    FROM employees  
    WHERE department = 'IT'  
)  
SELECT employee_id, salary  
FROM filtered_data  
WHERE salary > 50000;
```

**Question:** How can testers avoid common SQL mistakes like Cartesian joins or missing indexes?

**Answer:** Avoid Cartesian Joins: Cartesian joins occur when no proper join condition is applied between tables, leading to an exponential number of result rows. Always ensure you use correct JOIN conditions.

**Incorrect Example** (Cartesian join): `SELECT * FROM employees, departments;`

**Correct Example:**

```
SELECT *  
FROM employees e  
JOIN departments d ON e.department_id = d.department_id;
```

**Use Indexes:** Always create indexes on frequently queried columns or join conditions to improve performance. For example, ensure `employee_id` or `department_id` is indexed when used in filtering or joins.

**Question:** What are some best practices for organizing SQL queries in testing projects?

**Answer:** They include:

- Use meaningful naming conventions: Consistent table, column, and alias naming help others understand your queries quickly.
- Organize queries by test cases: For example, group all the SELECT queries used for validation under one file, and INSERT/UPDATE for data setup in another.
- Modular SQL files: Split large queries into logical SQL scripts or files, categorized by functionality (data setup, validation, teardown, etc.).
- Version control your SQL scripts: Use [Git](#) or another version control system to track changes to your queries, especially in long-term testing projects.

**Question:** How can you refactor SQL queries for better performance and scalability?

**Answer:**

- Minimum use of subqueries: Instead of multiple subqueries, use JOINS or CTEs to simplify your query. For example, instead of this subquery example:

```
SELECT employee_name  
FROM employees  
WHERE employee_id IN (SELECT employee_id FROM salaries WHERE salary > 50000);
```

Refactored Example:



```
SELECT e.employee_name
FROM employees e
JOIN salaries s ON e.employee_id = s.employee_id
WHERE s.salary > 50000;
```

- **Avoid using SELECT \*:** Retrieve only the necessary columns to reduce query load and improve performance.
- **Use EXPLAIN PLAN:** Always check your query's execution plan to understand how SQL engine processes it and where optimizations can be made, like creating indexes or optimizing joins.

## 15. SQL Tips and Tricks for QA Testers

**Question:** How can you use SQL functions for string manipulation and date formatting in testing?

**Answer:** String manipulation functions: Functions like CONCAT, SUBSTRING, LOWER, and UPPER are used to handle string data.

**Example:** These functions are helpful when validating data from the database that is presented in different formats in the UI or API responses.

```
SELECT CONCAT(first_name, ' ', last_name) AS full_name
FROM employees;
```

**Date formatting functions:** DATE\_FORMAT in MySQL or TO\_CHAR in Oracle can format dates in a specific pattern.

**Example (MySQL):**

```
SELECT DATE_FORMAT(hire_date, '%Y-%m-%d') AS formatted_date
FROM employees;
```

**Example (Oracle):**

```
SELECT TO_CHAR(hire_date, 'YYYY-MM-DD') AS formatted_date
FROM employees;
```

**Question:** How can you handle null values in SQL queries?

**Answer:** By using IS NULL and IS NOT NULL: To handle null values, use the IS NULL or IS NOT NULL clause in your WHERE conditions. By properly handling nulls, testers can avoid false negatives when validating data integrity.

**Example:**

```
SELECT employee_name
FROM employees
WHERE manager_id IS NULL; -- Selects employees with no manager
```

**Using COALESCE:** This function allows you to replace NULL values with a default value.

```
SELECT employee_name, COALESCE(manager_id, 'No Manager') AS manager_status
FROM employees;
```

**Question:** What are some practical tips for writing efficient SQL queries?

**Answer:**

- Use indexes: Ensure that columns frequently used in WHERE clauses, JOIN conditions, or ORDER BY are indexed for faster query execution.
- Limit the use of DISTINCT: Avoid using DISTINCT unnecessarily, as it can slow down queries. Instead, focus on joins and filtering.
- Avoid unnecessary JOINS: Only join tables that are required for the query. This reduces the overhead of data retrieval.
- Fetch only required columns: Avoid SELECT \* and always specify the columns you need to minimize data transfer.






**Example:** It only fetches the required data and optimize query performance.

```
SELECT first_name, last_name
FROM employees
WHERE department = 'QA'
ORDER BY hire_date;
```

**Question:** How do you handle edge cases in testing with SQL?

**Answer:**

- Missing data: Use LEFT JOIN to detect if there are missing rows in related tables. For example, finding employees without department assignments.

Download for reference  Like  Share  View SQL Queries video  <https://youtu.be/BxMmC77fj9Y>  
View [Database Testing videos playlist](#) 

Software and Testing Training YouTube Channel <https://youtube.com/@QA1>

Software Testing Space Blog <https://inderpsingh.blogspot.com/> Copyright © 2024 All Rights Reserved.

### Example:

```
SELECT e.employee_name
FROM employees e
LEFT JOIN departments d ON e.department_id = d.department_id
WHERE d.department_id IS NULL;
```

- **Duplicate data:** Use GROUP BY and HAVING COUNT > 1 to detect duplicate rows (that might break your application's logic).

### Example:

```
SELECT employee_name, COUNT(*)
FROM employees
GROUP BY employee_name
HAVING COUNT(*) > 1;
```

- **Boundary values:** Test edge cases like zero, negative numbers, or very large values using filters in SQL queries.

### Example:

```
SELECT employee_name, salary
FROM employees
WHERE salary < 0 OR salary > 1000000; -- Checking for salary anomalies
```

**Note:** In order to review SQL queries fast, you should view my SQL Queries tutorial [here](https://youtu.be/BxMmC77fj9Y). To get future updates, you're welcome to connect with me or follow me in LinkedIn at the link, [Inder P Singh](https://www.linkedin.com/in/inderpsingh/). Thank you!