**Basic SELECT Queries and Aggregate Functions (1-10):**

1. **Find the top 10 products with the most sales revenue**
2. **List the customers who have spent more than the average expenditure**
3. Identify the month with the highest number of orders
4. Calculate the total revenue per category for the current year
5. Find the seller with the least number of orders, excluding those with no orders
6. List the products that have never been ordered
7. Determine the customer with the highest number of orders for each year
8. Calculate the difference between the highest and lowest payment received for each payment type
9. Calculate the total revenue generated by each product category

**JOIN Operations (10-40):**

1. Formulate a multi-table INNER JOIN to find customers who have placed orders but have not left reviews.
2. Employ a LEFT JOIN to retrieve all sellers who have never listed a product.
3. Devise a RIGHT JOIN to display all geolocations that have never been associated with a customer address.
4. Implement a FULL OUTER JOIN to show all payment transactions and the corresponding order details, including any mismatches.
5. Generate an INNER JOIN query to correlate customer orders with payments, excluding orders with no payments.
6. Use a LEFT JOIN to find all products that have been listed but never featured in any order.
7. Create a RIGHT JOIN query to list customers with their last order details, including customers with no orders.
8. Construct a FULL OUTER JOIN to compare order items and reviews, showing all possible combinations.
9. Formulate a complex INNER JOIN involving orders, customers, and payments to find discrepancies in payment records.
10. Employ a LEFT JOIN to match order reviews with customer data, including reviews with anonymous users.
11. Devise a RIGHT JOIN to display all shipping information, including orders that have not been shipped.
12. Implement a FULL OUTER JOIN to gather all customer, order, and payment data, highlighting any missing links.
13. Generate an INNER JOIN query to associate sellers with their top-selling products.
14. Use a LEFT JOIN to compile a list of all customers and their total spending, including those who have never made a purchase.
15. Create a RIGHT JOIN query to show all products and their inventory levels, including products that are not in inventory.
16. Construct a FULL OUTER JOIN to reveal all orders and their associated reviews, including orders without reviews.
17. Formulate a multi-table INNER JOIN to find all customers who have made payments but have not received their orders.
18. Employ a LEFT JOIN to identify all orders that have been placed but not reviewed.
19. Devise a RIGHT JOIN to list all products and their order history, including products that have never been ordered.
20. Implement a FULL OUTER JOIN to correlate all customers, orders, and products, regardless of missing data.
21. Generate an INNER JOIN query to find all orders with their corresponding shipment details, excluding unshipped orders.
22. Use a LEFT JOIN to determine which products have been reviewed but not sold.
23. Create a RIGHT JOIN query to list all payment methods and their usage, including unused methods.
24. Construct a FULL OUTER JOIN to compare customer addresses and geolocations, including those without a match.
25. Formulate a complex INNER JOIN involving multiple tables to identify orders with payment discrepancies.
26. Employ a LEFT JOIN to retrieve all seller ratings and comments, including sellers who have not been rated.

**Subqueries and Nested SELECTs (40-50):**

1. Write a query to find the names of customers who have placed orders that are above the average order value for all orders in the database.
2. Create a query to list the products that have a price higher than the average price of all products in their respective category.
3. Construct a query to identify the third highest revenue-generating product using a subquery.
4. Formulate a query that uses a correlated subquery to find all orders that contain the same products as Order ID ‘XYZ’.
5. Develop a query to retrieve the top 3 most frequent customer cities from the orders table using a subquery.
6. Write a query to find the month with the second highest number of orders using a subquery in the WHERE clause.
7. Create a query that lists all customers who have never ordered the same product twice using a correlated subquery.
8. Construct a query to display the seller who has the most products listed in the database without using GROUP BY or LIMIT.
9. Formulate a query to find all customers whose total order amount is greater than the average total order amount of all customers.
10. Develop a query to list all products that have been ordered more than three times in the past month using a subquery in the FROM clause.

**Advanced Filtering (50-60):**

1. Write a query to find all customers whose names contain at least three vowels in a row.
2. Create a query to list all products whose price is within the top 10% of the most expensive products.
3. Construct a query to identify orders placed in the last quarter of any year and have at least one NULL value in any column.
4. Formulate a query that selects all orders where the customer’s last name starts with ‘Mc’ or ‘Mac’.
5. Develop a query to retrieve all products that have not been sold in the current year and have NULL values in the description field.
6. Write a query to find all customers who have placed orders that include product names with ‘new’ anywhere in the name.
7. Create a query to list all orders where the total amount is between the smallest and largest orders by value.
8. Construct a query to display all customers whose first or last name begins with a vowel, using the IN operator.
9. Formulate a query that uses COALESCE to replace any NULL values in the address field with ‘Address not provided’.
10. Develop a query to find all products with a name that ends in ‘ex’ and have been sold in quantities that fall within the second quartile of sales.

**Window Functions(60-70)**

1. Write a query that uses a window function to rank customers based on the total revenue they generated, partitioned by country.
2. Create a query to calculate the running total of sales for each product, partitioned by product category and ordered by sale date.
3. Construct a query that uses a window function to find the three most recent orders for each customer.
4. Formulate a query to determine the average order value for each customer, compared to the average order value of their respective customer segment using a window function.
5. Develop a query that uses a window function to identify the second highest quantity of each product ever sold in a single order.
6. Write a query to display the difference in order amounts between each order and the previous order for the same customer using the LAG window function.
7. Create a query that uses the LEAD window function to predict the next purchase date for each customer based on their order history.
8. Construct a query to calculate the cumulative distribution of product sales over time using a window function.
9. Formulate a query that uses a window function to assign a percentile rank to each order based on the total order amount within each year.
10. Develop a query to compare the total number of orders each customer has made to the average number of orders made by all customers using a window function.

**Analytic Queries (70-80):**

1. Write a query to identify the top 5% of customers by total spend and compare their average order size to the overall average order size.
2. Create a query that calculates the year-over-year growth rate in sales for each product category.
3. Construct a query to determine the month-over-month retention rate of customers based on their first purchase date.
4. Formulate a query that ranks products by the number of times they were purchased within the top 10 orders by total value.
5. Develop a query to find the average time between consecutive orders for each customer and how it compares to the site-wide average.
6. Write a query to calculate the lifetime value (LTV) of customers acquired in the first quarter of any given year.
7. Create a query that segments customers into quartiles based on their order frequency and calculates the average order value for each quartile.
8. Construct a query to analyze the variance in order values for each day of the week to identify trends or anomalies.
9. Formulate a query that uses moving averages to smooth out daily sales data and highlight underlying trends.
10. Develop a query to compare the performance of promotional campaigns by calculating the increase in average order value during the campaign period versus the baseline period.

**Stored Procedures (80-85):**

1. Write a stored procedure that creates a temporary view of the top 10 customers by total spend and then uses that view to update a loyalty status column in the customers table.
2. Develop a stored procedure that dynamically creates indexes on columns frequently used in WHERE clauses of ad-hoc queries, and demonstrate the performance difference with and without the indexes.
3. Create a complex view that combines data from multiple tables and includes subqueries, then write a stored procedure that performs a series of updates based on conditions from this view.
4. Design a stored procedure that monitors index fragmentation over time and automatically rebuilds indexes when they exceed a certain fragmentation threshold.
5. Formulate a stored procedure that logs query performance metrics before and after creating a new index, providing insights into the impact of indexing on complex joins involving multiple tables.

**SQL Functions(85-90)**

1. Write a query that uses a combination of string functions to reverse the names of all products and display only those that are palindromes.
2. Create a query that utilizes date functions to retrieve all orders that were placed on the last day of the month for the past year.
3. Construct a query using numeric functions to find all orders where the total amount is a prime number.
4. Formulate a query that employs window functions to calculate the moving average of daily sales, partitioned by product category.
5. Develop a query that uses conversion functions to change the format of stored dates and display them alongside the original format, ordered by the month and year.

**Table – view – indexes (90 - 100)**

1. Write a query that creates a materialized view based on a complex join of five different tables and includes a calculated field that summarizes data from those tables.
2. Develop a query that demonstrates the performance difference between a regular view and a materialized view when used in a complex join operation.
3. Create a query that uses a combination of table partitioning and indexing to optimize a query that filters data from a large dataset based on a range of dates.
4. Formulate a query that dynamically creates an index on a view that is frequently queried and demonstrate how it affects query performance.
5. Construct a query that alters an existing table to add a new column and then creates an index on that column, showing the before and after performance metrics for a relevant query.
6. Write a query to create a non-clustered index on a table with over 10 million records and measure the performance improvement on a search query.
7. Develop a view that consolidates data from multiple tables with different data types and write a query to report any type mismatches.
8. Create a query that identifies all redundant indexes in a database, including those on foreign key columns where the parent table already has an index.
9. Formulate a query that uses a window function over a view and compare the performance with and without an index on the view.
10. Construct a query to convert an existing table with a high volume of transactions into a partitioned table and demonstrate the performance benefits.
11. Write a query that creates a materialized view from a complex join operation and includes a column for text search using full-text indexing.
12. Design a query to update a table with a unique constraint and observe the behavior when attempting to insert duplicate values.
13. Develop a query that uses indexed views to improve the performance of a recursive CTE operation.
14. Create a query that demonstrates the impact of dropping an index on the execution plan of a complex query involving multiple joins and subqueries.
15. Formulate a query that alters a large table by adding a new column and then creates a filtered index on that column, showing the performance difference for relevant queries.