**Easy (10 questions)**

1. **Retrieve basic customer information**
   * Show each customer's first\_name, last\_name, city, and state.
2. **List products and their respective categories**
   * Use a join on the products and categories table.
3. **Find all orders with their store name and staff name**
   * Combine orders, stores, and staffs to show order\_date, store\_name, and staff full name.
4. **Display total number of customers per state**
   * Use GROUP BY on state and return the count of customers.
5. **List products that are currently in stock at each store**
   * Use stocks and products join to show store\_id, product\_name, and quantity.
6. **Show the total quantity ordered for each product**
   * Use order\_items grouped by product\_id (or product name if joined to products).
7. **Identify any manager-staff relationship**
   * List each staff\_id, their manager\_id, and manager’s name.
8. **Find the total number of orders processed by each staff**
   * Join orders and staffs, group by staff\_id.
9. **Check if there are customers with no orders**
   * Use a LEFT JOIN between customers and orders and look for NULL in order\_id.
10. **Display orders with their total item count**
    * Using order\_items, group by order\_id and COUNT(item\_id) as total items.

**Intermediate (20 questions)**

1. **Determine each store’s total stock quantity**
   * Combine stores and stocks to show store\_id and the sum of all product quantities.
2. **List customers who made more than 3 orders**
   * Group by customer\_id in orders and use a HAVING clause.
3. **Show the top 5 stores by total quantity in stock**
   * Use an ORDER BY (descending) on the sum of stocks.quantity and TOP 5 (or limit 5).
4. **Calculate the total revenue per order**
   * For each order\_id, sum up (list\_price \* quantity) - discount from order\_items.
5. **Find the latest order date for each customer**
   * Use MAX(order\_date) in conjunction with grouping by customer\_id, or a window function.
6. **Create a *view*** that returns comprehensive order info
   * The view should join orders, order\_items, customers, stores, and products to provide a single result set. (Conceptualize the view—avoid explicit DDL unless your instructor allows a CREATE VIEW statement).
7. **Recommend an *index* strategy** to improve queries filtering by order\_date in the orders table
   * Explain which columns to index (e.g., order\_date or a composite index with customer\_id).
8. **Write a stored procedure** that takes a customer\_id and returns all of that customer’s orders
   * Procedure input: @CustomerID. Output: order headers or full order details.
9. **Use ROW\_NUMBER()** to label each product in each category in ascending order by list\_price
   * Partition by category\_id, order by list\_price.
10. **Use RANK()** to find top 3 best-selling products within each category
    * Based on total quantity sold or total revenue.
11. **Display orders where the discount is higher than the overall average discount**
    * Compare each order\_items.discount to (SELECT AVG(discount) FROM order\_items).
12. **Find customers who have *not* placed any orders in the last 6 months**
    * Compare order\_date with GETDATE() or relevant function minus 6 months.
13. **Use a LEAD() window function** to show each order date and the next order date for that *same* customer
    * Partition by customer\_id and order by order\_date.
14. **Use a LAG() window function** to show difference in discounts for consecutive order items (by product)
    * Partition by product\_id, ordered by order\_id or order\_date.
15. **Create a user-defined function** that returns how many orders a given customer\_id has placed
    * Input: @CustomerID, returns INT.
16. **Determine average list\_price per category** but only for products introduced after a certain model\_year
    * Use WHERE model\_year >= X and GROUP BY category\_id.
17. **Identify products in stock at more than one store**
    * Group by product\_id in stocks and use HAVING COUNT(store\_id) > 1.
18. **Calculate the running total of items sold** for each product
    * Use a window function like SUM(...) OVER (PARTITION BY product\_id ORDER BY order\_date ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW).
19. **Find staff members who handle orders *and* manage other staff**
    * Check orders.staff\_id and also check if the same staff\_id appears as a manager\_id in the staffs table.
20. **Show a monthly breakdown of total revenue** per store
    * Use DATEPART(MONTH, order\_date) or similar function and group by store and month.

**Hard (20 questions)**

1. **Create a trigger** that logs every newly inserted order into a separate logging table
   * The log could store order\_id, customer\_id, order\_date, etc. (Conceptualize the trigger rather than writing full DDL if restricted).
2. **Build a view** that lists each product’s total quantity sold, total revenue, and average discount
   * Combine order\_items and products, group by product\_id.
3. **Write a user-defined function** that returns all order IDs for a given customer\_id as a comma-separated string
   * Demonstrates string aggregation logic.
4. **Identify the *best-selling category* by total revenue for each year**
   * Group by category\_id and YEAR(order\_date), then sort by revenue.
5. **Write a stored procedure** to update the list\_price of all products in a given category by a certain percentage
   * Input parameters: @CategoryID, @Percentage.
6. **Calculate the percentage of on-time orders** where shipped\_date <= required\_date
   * Compare the number of on-time vs total orders.
7. **Use DENSE\_RANK()** to rank staff by the number of orders handled
   * Partition by nothing or by store, ordered by count of orders.
8. **Compute the cumulative sales (revenue) per customer over time**
   * Use a window function for the running total, partitioned by customer\_id, ordered by order\_date.
9. **Find products that appear in orders but are *not* in any store’s stock**
   * Compare order\_items.product\_id with stocks.product\_id.
10. **Identify all orders that have a discount higher than the average discount for *that product***
    * Use a correlated subquery or window function to compare each row’s discount with the product's average discount.
11. **Write a query** to find “repeat customers” who placed more than one order in different months
    * Compare the order dates’ months; group by customer\_id.
12. **Construct a trigger** that prevents any order from being inserted if required\_date < order\_date
    * Ensures no invalid date logic in new rows.
13. **Find the top 3 performing stores** by revenue in the last 12 months
    * Filter order\_date, sum revenue, rank or sort, then pick top 3.
14. **Determine the latest 3 orders for each customer** using ROW\_NUMBER()
    * Partition by customer\_id, order by order\_date desc, and filter row\_number <= 3.
15. **Create a user-defined function** that returns the total number of items for a given order\_id
    * Input: @OrderID. Output: integer count from order\_items.
16. **Identify the order that generated the *highest revenue* overall**, and find which customer placed it
    * Combine orders and order\_items to calculate total revenue, then find the max.
17. **List customers who placed an order *but never received it*** (i.e., shipped\_date is NULL)
    * Join orders and customers, filter by shipped\_date IS NULL.
18. **Show the difference in total revenue between consecutive years**
    * Group by year, use LAG() or LEAD() to compare revenue of the previous year vs current year.
19. **Find any product that is sold at a discount *greater than 25%* for at least one order**
    * Calculate (discount / (list\_price \* quantity)) \* 100 > 25 or adapt if discount is an absolute amount.
20. **Create a view** (or the logic for one) that displays real-time inventory: how many items are in stock *minus* how many are currently ordered but not shipped
    * Combine stocks and unshipped orders + order\_items, subtract in-flight order quantities from stocks.quantity.