VISION BOARD

AZURE DATA ENGINEERING INTERVIEW QUESTIONS PART-2

DEVIKRISHNA R

LinkedIn: @Devikrishna R Email: visionboard044@gmail.com

WINDOW FUNCTIONS

1. Calculate the Cumulative Sales for Each Product Category

- **Schema**: sales_data(product_category, sale_date, sale_amount)
- **Query**:

```
SELECT
    product_category,
    sale_date,
    sale_amount,
    SUM(sale_amount) OVER (PARTITION BY product_category ORDER BY
        sale_date) AS cumulative_sales
FROM
    sales_data
ORDER BY
    product_category, sale_date;
```

Explanation:

- PARTITION BY product_category: Groups data by product category.
- ORDER BY sale_date: Ensures cumulative calculation is done in order of sale date.

---

2. Find the Top Three Performers in Each Department Based on Their Annual Sales Using Dense Rank

- **Schema**: employee_sales(department, employee_id, annual_sales)
- **Query**:

```
WITH ranked_sales AS (
    SELECT
        department,
        employee_id,
        annual_sales,
        DENSE_RANK() OVER (PARTITION BY department ORDER BY
            annual_sales DESC) AS rank
    FROM
        employee_sales
)
SELECT
    department,
    employee_id,
    annual_sales
FROM
    ranked_sales
WHERE
    rank <= 3
ORDER BY
    department, rank;
```

Explanation:

- DENSE_RANK() ranks employees within each department based on annual_sales in descending order.
- rank <= 3 filters for the top 3 performers.

---

3. Compute the Difference Between the Highest and Lowest Order Value for Each Customer

- **Schema**: orders(customer_id, order_id, order_value)

- **Query**:

```
SELECT
    customer_id,
    MAX(order_value) OVER (PARTITION BY customer_id) - MIN(order_value)
        OVER (PARTITION BY customer_id) AS order_value_difference
FROM
    orders;
```

Explanation:

- MAX() and MIN() calculate the highest and lowest order values per customer.
- Subtracting them gives the required difference.

---

4. Calculate the Running Total of Salaries by Department

- **Schema**: employee_salaries(department, employee_id, salary)
- **Query**:

```
SELECT
    department,
    employee_id,
    salary,
    SUM(salary) OVER (PARTITION BY department ORDER BY employee_id) AS
        running_total_salary
FROM
    employee_salaries
ORDER BY
    department, employee_id;
```

Explanation:

- PARTITION BY department: Groups salaries by department.

- ORDER BY employee_id: Ensures running total is calculated in a consistent order.

---

5. Determine the Average Monthly Sales for Each Region Over the Past Twelve Months

- **Schema**: regional_sales(region, sale_date, sale_amount)
- **Query**:

```
SELECT
    region,
    sale_date,
    AVG(sale_amount) OVER (
        PARTITION BY region
        ORDER BY sale_date
        ROWS BETWEEN 11 PRECEDING AND CURRENT ROW
    ) AS avg_monthly_sales
FROM
    regional_sales
WHERE
    sale_date >= DATEADD(MONTH, -12, GETDATE())
ORDER BY
    region, sale_date;
```

Explanation:

- PARTITION BY region: Groups sales by region.
- ROWS BETWEEN 11 PRECEDING AND CURRENT ROW: Calculates a rolling average for the past 12 months (including the current month).

- The WHERE clause ensures data is limited to the last 12 months.

COMMON TABLE EXPRESSIONS (CTES):

1. Recursive CTE to Calculate the Factorial of Numbers from One to Ten

Schema: No input table required.

Query:

```
WITH RECURSIVE FactorialCTE (n, factorial) AS (
    SELECT 1 AS n, 1 AS factorial  -- Base case: 1! = 1
    UNION ALL
    SELECT n + 1, factorial * (n + 1)
    FROM FactorialCTE
    WHERE n < 10  -- Limit the recursion to numbers up to 10
)
SELECT n, factorial
FROM FactorialCTE;
```

Explanation:

- The recursive part increments n and multiplies the factorial by (n + 1).
- The recursion stops at n = 10.

---

2. CTE to Split Email Addresses into Username and Domain Parts

Schema: users(email_address)

Query:

```
WITH EmailSplit AS (
    SELECT
        email_address,
        LEFT(email_address, CHARINDEX('@', email_address) - 1) AS
            username,
        RIGHT(email_address, LEN(email_address) - CHARINDEX('@',
            email_address)) AS domain
    FROM users
)
SELECT email_address, username, domain
FROM EmailSplit;
```

Explanation:

- LEFT() extracts the substring before the @.
- RIGHT() extracts the substring after the @.

---

3. Identify Gaps in a Sequence of Order IDs Using a CTE

Schema: orders(order_id)

Query:

```
WITH OrderWithGaps AS (
    SELECT
        order_id,
        LEAD(order_id) OVER (ORDER BY order_id) AS next_order_id
    FROM orders
)
SELECT
    order_id AS start_of_gap,
    next_order_id - 1 AS end_of_gap
FROM OrderWithGaps
WHERE next_order_id - order_id > 1;
```

4. CTE to Calculate Total Working Hours of Employees Grouped by Department

Schema: work_logs(employee_id, department, hours_worked)

Query:

```
WITH DepartmentHours AS (
    SELECT
        department,
        SUM(hours_worked) AS total_hours
    FROM work_logs
    GROUP BY department
)
SELECT department, total_hours
FROM DepartmentHours;
```

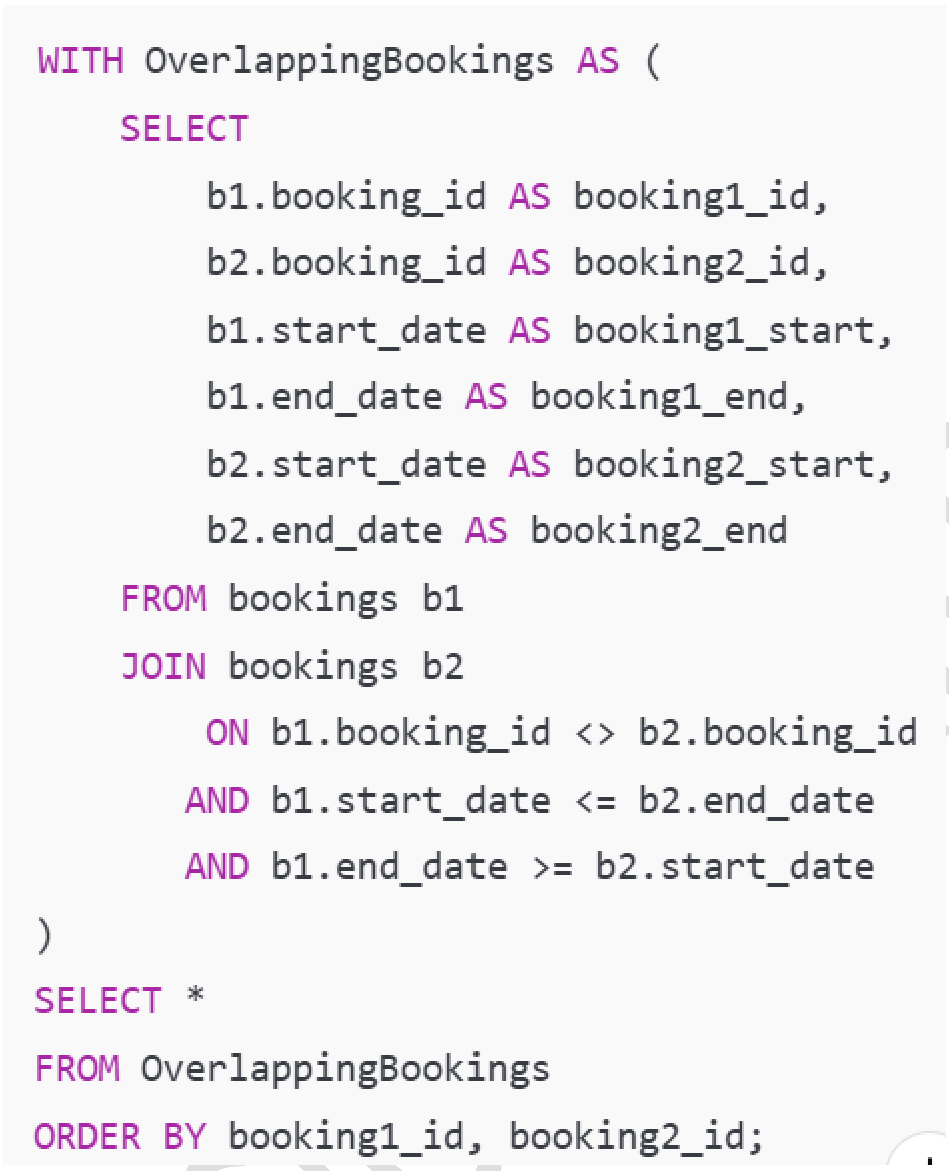
Explanation:

- The CTE aggregates hours_worked by department.

5. Detect Overlapping Date Ranges for Bookings Using a CTE

Schema: bookings(booking_id, start_date, end_date)

Query:



```sql
WITH OverlappingBookings AS (
    SELECT
        b1.booking_id AS booking1_id,
        b2.booking_id AS booking2_id,
        b1.start_date AS booking1_start,
        b1.end_date AS booking1_end,
        b2.start_date AS booking2_start,
        b2.end_date AS booking2_end
    FROM bookings b1
    JOIN bookings b2
        ON b1.booking_id <> b2.booking_id
       AND b1.start_date <= b2.end_date
       AND b1.end_date >= b2.start_date
)
SELECT *
FROM OverlappingBookings
ORDER BY booking1_id, booking2_id;
```

Explanation:

- Joins the bookings table with itself, excluding rows where the IDs are the same (b1.booking_id <> b2.booking_id).
- The overlapping condition is b1.start_date <= b2.end_date AND b1.end_date >= b2.start_date.

JOINS (INNER, OUTER, CROSS, SELF)

1. Retrieve All Employees Along with Their Assigned Projects, Including Employees Without Projects (Right Join)

Schema:

- employees(employee_id, name)
- projects(project_id, employee_id)

Query:

```
SELECT
    e.employee_id,
    e.name AS employee_name,
    p.project_id AS assigned_project
FROM
    employees e
RIGHT JOIN
    projects p
ON
    e.employee_id = p.employee_id;
```

Explanation:

- The **Right Join** ensures all rows from projects are included, even if there is no matching employee_id in employees.

---

2. Find Customers Who Placed Exactly One Order Using a Self-Join

Schema:

- orders(order_id, customer_id)

Query:

```
WITH OrderCounts AS (
    SELECT
        customer_id,
        COUNT(order_id) AS order_count
    FROM
        orders
    GROUP BY
        customer_id
)
SELECT
    customer_id
FROM
    OrderCounts
WHERE
    order_count = 1;
```

Explanation:

- A **Self-Join** isn't needed directly here; instead, use a **CTE or subquery** to count orders and filter for customers with exactly one order.

3. List All Products Along with the Suppliers Who Didn't Supply Them (Outer Join)

Schema:

- products(product_id, product_name)
- suppliers(supplier_id, product_id)

Query:

```
SELECT
    p.product_id,
    p.product_name,
    s.supplier_id
FROM
    products p
LEFT JOIN
    suppliers s
ON
    p.product_id = s.product_id
WHERE
    s.supplier_id IS NULL;
```

Explanation:

- The **Left Join** ensures all products are listed.
- The WHERE s.supplier_id IS NULL clause filters products not supplied by any supplier.

4. Generate All Possible Pairs of Employees and Calculate the Distance Between Their Office Locations Using a Cross Join

Schema:

- employees(employee_id, name, office_location)

Query:

```sql
SELECT
    e1.employee_id AS employee1_id,
    e1.name AS employee1_name,
    e1.office_location AS employee1_location,
    e2.employee_id AS employee2_id,
    e2.name AS employee2_name,
    e2.office_location AS employee2_location,
    -- Example formula to calculate distance (replace with real
        calculation logic)
    SQRT(POWER(e1.office_location_lat - e2.office_location_lat, 2) +
        POWER(e1.office_location_lon - e2.office_location_lon, 2)) AS
        distance
FROM
    employees e1
CROSS JOIN
    employees e2
WHERE
    e1.employee_id <> e2.employee_id;
```

Explanation:

- A **Cross Join** generates all possible combinations of employees.
- Filter e1.employee_id <> e2.employee_id to avoid pairing an employee with themselves.

5. Retrieve a List of Employees Who Share the Same Job Title Using a Self-Join

Schema:

- employees(employee_id, name, job_title)

Query:

```
SELECT
    e1.employee_id AS employee1_id,
    e1.name AS employee1_name,
    e2.employee_id AS employee2_id,
    e2.name AS employee2_name,
    e1.job_title
FROM
    employees e1
JOIN
    employees e2
ON
    e1.job_title = e2.job_title
    AND e1.employee_id <> e2.employee_id;
```

Explanation:

- A **Self-Join** is used to compare rows within the same table.
- The condition e1.job_title = e2.job_title ensures employees share the same title, and e1.employee_id <> e2.employee_id avoids pairing an employee with themselves.

SUBQUERIES

1. List the Products Whose Average Sales Exceed the Overall Average Sales

Schema:

- sales(product_id, sale_amount)

Query:

```
SELECT
    product_id
FROM
    sales
GROUP BY
    product_id
HAVING
    AVG(sale_amount) > (SELECT AVG(sale_amount) FROM sales);
```

Explanation:

- The **subquery** calculates the overall average sales.
- The **HAVING** clause compares the average sales per product to the overall average.

---

2. Find Employees Who Earn More Than the Manager of Their Department

Schema:

- employees(employee_id, name, salary, department_id, is_manager)

Query:

```
SELECT
    e1.employee_id,
    e1.name,
    e1.salary
FROM
    employees e1
WHERE
    e1.salary > (
        SELECT
            e2.salary
        FROM
            employees e2
        WHERE
            e2.department_id = e1.department_id
            AND e2.is_manager = 1
    );
```

Explanation:

- The **subquery** retrieves the manager's salary for the employee's department.
- The main query filters employees earning more than their department's manager.

---

3. Identify Customers Who Have Placed Orders for More Than Five Unique Products

Schema:

- orders(customer_id, product_id)

Query:

```
SELECT
    customer_id
FROM
    orders
GROUP BY
    customer_id
HAVING
    COUNT(DISTINCT product_id) > 5;
```

Explanation:

- No need for a subquery here; using COUNT(DISTINCT product_id) directly in the HAVING clause identifies customers with orders for more than five unique products.

---

4. Retrieve the Top Three Highest-Selling Products for Each Category Using a Correlated Subquery

Schema:

- products(product_id, product_name, category_id, sales_amount)

Query:

```sql
SELECT
    p1.category_id,
    p1.product_id,
    p1.product_name,
    p1.sales_amount
FROM
    products p1
WHERE
    (
        SELECT COUNT(*)
        FROM products p2
        WHERE p2.category_id = p1.category_id
          AND p2.sales_amount > p1.sales_amount
    ) < 3
ORDER BY
    p1.category_id, p1.sales_amount DESC;
```

Explanation:

- The **correlated subquery** counts products in the same category with higher sales.
- Products with fewer than 3 higher sales values are the top three.

---

5. List Employees Working in Departments with Fewer Than Three Employees

Schema:

- employees(employee_id, department_id)

Query:

```
SELECT
    employee_id,
    department_id
FROM
    employees
WHERE
    department_id IN (
        SELECT
            department_id
        FROM
            employees
        GROUP BY
            department_id
        HAVING
            COUNT(employee_id) < 3
    );
```

Explanation:

- The **subquery** identifies departments with fewer than three employees.
- The main query retrieves employees in those departments.

AGGREGATE FUNCTIONS

1. Calculate the Mode (Most Frequently Occurring Value) of Product Prices in Each Category

Schema:

- products(category_id, price)

Query:

```sql
WITH PriceFrequency AS (
    SELECT
        category_id,
        price,
        COUNT(*) AS frequency
    FROM
        products
    GROUP BY
        category_id, price
),
```

```sql
ModePrices AS (
    SELECT
        category_id,
        price,
        RANK() OVER (PARTITION BY category_id ORDER BY frequency DESC) AS rank
    FROM
        PriceFrequency
)
SELECT
    category_id,
    price AS mode_price
FROM
    ModePrices
WHERE
    rank = 1;
```

Explanation:

- The **PriceFrequency** CTE calculates the frequency of each price within each category.
- The **ModePrices** CTE ranks prices by their frequency within each category.
- The main query retrieves the most frequent price (rank = 1).

---

2. Find the Total Number of Orders Placed in Each Quarter and Rank Them by Volume

Schema:

- orders(order_id, order_date)

Query:

```
WITH QuarterlyOrders AS (
    SELECT
        DATEPART(YEAR, order_date) AS order_year,
        DATEPART(QUARTER, order_date) AS order_quarter,
        COUNT(order_id) AS total_orders
    FROM
        orders
    GROUP BY
        DATEPART(YEAR, order_date), DATEPART(QUARTER, order_date)
)
SELECT
    order_year,
    order_quarter,
    total_orders,
    RANK() OVER (ORDER BY total_orders DESC) AS rank
FROM
    QuarterlyOrders;
```

Explanation:

- **DATEPART** extracts the year and quarter from order_date.
- Orders are grouped by quarter, and the total is ranked by volume using RANK().

---

3. Count the Number of Employees Earning Above the Median Salary

Schema:

- employees(employee_id, salary)

Query:

```
WITH MedianSalary AS (
    SELECT
        PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY salary) AS median_salary
    FROM
        employees
)
SELECT
    COUNT(*) AS employees_above_median
FROM
    employees
WHERE
    salary > (SELECT median_salary FROM MedianSalary);
```

Explanation:

- The **PERCENTILE_CONT(0.5)** function calculates the median salary.
- The main query counts employees whose salary is above this median.

---

4. Identify the Month with the Highest Average Sales in Each Region

Schema:

- sales(region_id, sale_date, sale_amount)

Query:

```
WITH MonthlySales AS (
    SELECT
        region_id,
        DATEPART(YEAR, sale_date) AS sale_year,
        DATEPART(MONTH, sale_date) AS sale_month,
        AVG(sale_amount) AS avg_monthly_sales
    FROM
        sales
    GROUP BY
        region_id, DATEPART(YEAR, sale_date), DATEPART(MONTH,
            sale_date)
),
```

```
RankedSales AS (
    SELECT
        region_id,
        sale_year,
        sale_month,
        avg_monthly_sales,
        RANK() OVER (PARTITION BY region_id ORDER BY
            avg_monthly_sales DESC) AS rank
    FROM
        MonthlySales
)
SELECT
    region_id,
    sale_year,
    sale_month,
    avg_monthly_sales
FROM
    RankedSales
WHERE
    rank = 1;
```

Explanation:

- The **MonthlySales** CTE calculates average sales by region and month.
- The **RankedSales** CTE ranks months by average sales for each region.
- The main query retrieves the month with the highest average sales (rank = 1).

---

5. Compute the Variance and Standard Deviation of Sales for Each Product Category

Schema:

- sales(category_id, sale_amount)

Query:

```
SELECT
    category_id,
    VAR(SALE_AMOUNT) AS sales_variance,
    STDEV(SALE_AMOUNT) AS sales_std_dev
FROM
    sales
GROUP BY
    category_id;
```

Explanation:

- **VAR()** computes the variance of sales for each category.
- **STDEV()** computes the standard deviation.

INDEXING AND PERFORMANCE

1. Find the Most Frequently Accessed Rows in a Table Using Indexed Columns

Schema:

- access_logs(log_id, table_id, row_id, access_count)

Query:

```sql
SELECT
    table_id,
    row_id,
    access_count
FROM
    access_logs
WHERE
    access_count = (
        SELECT MAX(access_count)
        FROM access_logs
    )
ORDER BY
    access_count DESC;
```

Explanation:

- This assumes the access_logs table tracks row access counts, and access_count is an indexed column.
- The query identifies rows with the maximum access_count.

- Indexed columns speed up the MAX function and filtering operations.

---

2. Analyze the Impact of Adding an Index on a Large Text Column

Discussion:

- Adding an index to a large text column (e.g., VARCHAR(MAX) or TEXT) can **negatively affect performance** due to the following:
  - **Index Size**: Indexing large columns consumes significant disk space.
  - **Write Overhead**: Every insert/update to the table requires updating the index, increasing latency.
  - **Limited Use Cases**: Text indexes are beneficial only if queries involve searching or filtering large text fields frequently (e.g., LIKE or FULLTEXT queries).

Recommendations:

- Use **FULLTEXT INDEX** for large text fields to optimize search operations.
- If full indexing is unnecessary, consider adding a **computed column index** or indexing a hash of the text column for quick lookups.

---

3. Identify Queries That Perform Full Table Scans and Suggest Indexing Improvements

Query to Identify Full Table Scans (SQL Server Example):

```
SELECT
    q.text AS query_text,
    p.physical_operator,
    p.logical_operator
FROM
    sys.dm_exec_requests r
JOIN
    sys.dm_exec_query_stats s ON r.plan_handle = s.plan_handle
CROSS APPLY
    sys.dm_exec_sql_text(r.sql_handle) AS q
JOIN
    sys.dm_exec_query_plan(s.plan_handle) AS p
WHERE
    p.physical_operator = 'Table Scan';
```

Suggestions for Indexing Improvements:

1. **Filter Columns**: Identify frequently filtered columns in WHERE or JOIN clauses and add indexes on them.
2. **Covering Indexes**: For queries with multiple columns in SELECT, add composite indexes to include all referenced columns.
3. **Query Optimization**: Rewrite queries to avoid conditions like LIKE '%pattern%', which prevents index usage.

---

4. Compare the Performance of Single-Column Index vs. Composite Index on Query Execution Time

Schema:

- sales(order_id, product_id, customer_id, order_date, total_amount)

Test Queries:

1. **Single-Column Index**:

```
CREATE INDEX idx_order_date ON sales(order_date);

SELECT
    order_id, order_date, total_amount
FROM
    sales
WHERE
    order_date BETWEEN '2025-01-01' AND '2025-01-31';
```

2. **Composite Index**:

```
CREATE INDEX idx_order_date_total_amount ON sales(order_date, total_amount);

SELECT
    order_id, order_date, total_amount
FROM
    sales
WHERE
    order_date BETWEEN '2025-01-01' AND '2025-01-31'
    AND total_amount > 100;
```

Comparison:

- **Single-Column Index**: Works well when filtering on a single column (order_date).
- **Composite Index**: Offers better performance when filtering on both order_date and total_amount because it avoids the need for additional lookups.

Analysis: Use composite indexes when queries involve multiple columns in WHERE or ORDER BY.

5. Monitor Index Usage Statistics for a Database Table

SQL Server Example:

```
SELECT
    OBJECT_NAME(i.object_id) AS table_name,
    i.name AS index_name,
    i.type_desc AS index_type,
    user_seeks,
    user_scans,
    user_lookups,
    user_updates
FROM
    sys.dm_db_index_usage_stats s
JOIN
    sys.indexes i ON s.object_id = i.object_id AND s.index_id = i.index_id
WHERE
    OBJECT_NAME(i.object_id) = 'your_table_name';
```

Explanation:

- user_seeks: Number of times the index was used for an efficient lookup.
- user_scans: Indicates table or index scans (less efficient).
- user_updates: Counts index updates due to data changes.

Actionable Insights:

- High user_scans suggests missing or underutilized indexes.
- High user_updates could indicate write-heavy workloads and suggest avoiding excessive indexing.