Lecture Notes – LAG()

Section 1: Lecture Summary

This section covers the **LAG()** window function, a value access function used to retrieve data from the previous row in a result set. The LAG() function enables comparative analysis by displaying the previous value alongside the current value, allowing you to track changes across ordered sequences such as order amounts over time or customer purchase history.

Section 2: Key Concepts and Explanations

LAG() Function Basics

The LAG() function accesses the value from a previous row without using a self-join. It returns NULL for the first row since there is no previous row to reference. The basic syntax is:

```
LAG(column_name) OVER (ORDER BY column_name)
```

Partition By Clause

When analyzing data grouped by categories (such as customers), use the PARTITION BY clause to reset the lag calculation for each group:

```
LAG(column_name) OVER (PARTITION BY grouping_column ORDER BY
ordering_column)
```

This ensures that the lag function only looks back within the same partition, not across different groups.

Order By Clause

The ORDER BY clause within the OVER() function determines the sequence in which rows are evaluated. Ordering by date, amount, or ID ensures that the "previous" value refers to the logically preceding row based on your analysis needs.

Section 3: Example Code and Use Cases

Use Case 1: Compare Each Order with Previous Order Amount

Retrieve all orders and display the previous order's amount for comparison:

```
SELECT
    OrderID,
    TotalAmount,
    LAG(TotalAmount) OVER (ORDER BY OrderID) AS PreviousOrderAmount
FROM Orders;
```

This query returns all orders sorted by OrderID, showing each order's total amount alongside the previous order's amount. The first row displays NULL since no previous order exists.

Use Case 2: Compare Orders Sorted by Amount

Arrange orders by increasing amount to identify how order values change when sorted:

```
SELECT
    OrderID,
    TotalAmount,
    LAG(TotalAmount) OVER (ORDER BY TotalAmount) AS PreviousOrderAmount
FROM Orders
ORDER BY TotalAmount;
```

This reveals the progression of order amounts when viewed in ascending order, making it easy to see value differences between consecutive orders (10,200 → 25,500 → 30,000 → 38,000, etc.).

Use Case 3: Customer-Wise Lag of Order Amount

Analyze each customer's orders independently by partitioning on CustomerID:

```
SELECT
    CustomerID,
    OrderID,
    OrderDate,
    TotalAmount,
    LAG(TotalAmount) OVER (PARTITION BY CustomerID ORDER BY OrderID) AS
PreviousCustomerOrderAmount
FROM Orders;
```

This groups all orders by customer and shows each customer's previous order amount. Each customer's lag sequence resets independently.

Use Case 4: Customer Order History by Date

Track customer orders chronologically to see spending patterns over time:

```
SELECT
    CustomerID,
    OrderID,
    OrderDate,
    TotalAmount,
    LAG(TotalAmount) OVER (PARTITION BY CustomerID ORDER BY OrderDate) AS
PreviousOrderAmount
FROM Orders;
```

This arranges each customer's orders by date and displays the amount from their previous order. You can now observe whether a customer's spending is increasing or decreasing over time.

Section 4: Key Takeaways

LAG() enables temporal and comparative analysis by retrieving previous row values without complex self-joins, making queries more readable and efficient.

PARTITION BY is essential for grouped analysis, allowing you to reset the lag calculation for each group (customer, product category, region, etc.) rather than across the entire result set.

ORDER BY determines the sequence, so choose your ordering logic based on your analysis goal—whether chronological (by date), numerical (by amount), or logical (by ID).

The first row always returns NULL since there is no previous row, which is normal and expected behavior.

LAG() applications extend beyond amounts to any comparable column: dates, quantities, ratings, prices, or any metric where comparing consecutive values provides analytical value.