Lecture Notes – LEAD()

Section 1: Lecture Summary

The lecture introduces the **LEAD()** window function, which retrieves the next value from a ordered set of rows, contrasting with **LAG()** that retrieves the previous value. It demonstrates queries on the **Orders** table to compare each order's **TotalAmount** with the next order's amount, first without ordering, then ordered by **OrderDate**, partitioned by **CustomerID**, and finally partitioned by **CustomerID** with ordering by **OrderDate** within partitions. Results show **LEAD(TotalAmount)** values aligned by the specified window, with **NULL** for the last row or order in each partition. The function enables analysis of sales trends and customer order improvements over time.

Section 2: Key Concepts and Explanations

LEAD() is a window function that accesses the next row's value in the result set defined by the **OVER** clause. Basic syntax: **LEAD(column)** over the window returns the subsequent row's column value; the last row returns **NULL**. Without **ORDER BY**, it follows the table's natural order. Adding **ORDER BY OrderDate** arranges rows chronologically for time-based comparisons. **PARTITION BY CustomerID** groups rows by customer, restarting the window for each group to analyze per-customer sequences. Combining **PARTITION BY CustomerID ORDER BY OrderDate** orders dates within each customer partition, ideal for tracking individual customer order progression. This supports calculating differences (e.g., current minus next amount) to identify sales growth or decline.

Section 3: Example Code and Use Cases

Using the eCommerceDB **Orders** table:

Basic LEAD without ordering:

```
SELECT OrderID, OrderDate, TotalAmount,
       LEAD(TotalAmount) OVER() AS NextAmount
FROM Orders;
```

Returns **TotalAmount** and the next row's amount in table order, **NULL** for the last row.

LEAD ordered by date:

```
SELECT OrderID, OrderDate, TotalAmount,
       LEAD(TotalAmount) OVER(ORDER BY OrderDate) AS NextAmount
FROM Orders;
```

Arranges by **OrderDate** for chronological next-amount comparison.

Customer-wise LEAD:

```
SELECT OrderID, CustomerID, OrderDate, TotalAmount,
       LEAD(TotalAmount) OVER(PARTITION BY CustomerID) AS NextAmount
FROM Orders;
```

Groups by **CustomerID**, shows next order amount per customer.

Customer-wise with date ordering:

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

Orders dates within each customer for precise progression analysis (e.g., subtract **TotalAmount - NextAmount** for change).

Use cases: Compare sales day-by-day across all orders or per customer to detect improvements.

Section 4: Key Takeaways

LEAD() provides forward-looking row access via **OVER** clause for trend analysis. Specify **PARTITION BY** for grouping, **ORDER BY** for sequencing; last row in window is **NULL**. Apply to **Orders.TotalAmount** with **OrderDate** and **CustomerID** for sales and customer behavior insights. Practice builds analytical queries combining **LEAD** and **LAG**.