Lecture Notes – QueriesOnSUM( )

Section 1: Lecture Summary

Window functions combined with the **SUM** aggregate function are used to analyze customer spending progression, payment totals by mode, and running payment sums by city in the eCommerceDB. Queries demonstrate cumulative sums partitioned by customer or city, and totals partitioned by payment mode, using single or joined tables from the schema.

Section 2: Key Concepts and Explanations

Customer Lifetime Value (LTV) Timeline: Tracks cumulative total spending per customer across orders, ordered by order date to show progression. Uses **SUM(totalAmount) OVER (PARTITION BY CustomerID ORDER BY OrderDate)** on the Orders table.

Payment Totals by Mode: Shows each payment with the total amount collected for its payment mode using **SUM(Amount) OVER (PARTITION BY PaymentMode)** on the Payments table.

Running Sum of Payments per City: Calculates cumulative payments by customer city, requiring joins between Customers, Orders, and Payments tables on common keys (CustomerID, OrderID). Uses **SUM(Amount) OVER (PARTITION BY City ORDER BY PaymentDate)**, sourcing Amount from Payments to confirm received payments.

Percentage of Revenue by Category (practice query hint): Join Categories, Products (via CategoryID, ProductID), and OrderItems (via ProductID); compute **SUM(Subtotal)** per category as percentage of total revenue using window function partitioned by category.

Section 3: Example Code and Use Cases

LTV Timeline (cumulative spending per customer):

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

Shows first order amount, then cumulative total (e.g., 30,000 then 105,000 for a customer).

Payments with Total by Mode:

```
SELECT
    PaymentID,
    PaymentMode,
    Amount,
    SUM(Amount) OVER (PARTITION BY PaymentMode) AS TotalByMode
FROM Payments;
```

Groups payments by mode (e.g., Credit Card, UPI) with running total for each.

Running Payments per City:

```
SELECT
    C.City,
    P.Amount,
    P.PaymentDate,
    SUM(P.Amount) OVER (PARTITION BY C.City ORDER BY P.PaymentDate) AS
CityPaymentRunningTotal
FROM Payments P
NATURAL JOIN Orders O
NATURAL JOIN Customers C;
```

Joins on CustomerID/OrderID; running total per city (e.g., Bangalore cumulative from 87,500 onward).

Revenue Percentage by Category (based on hint):

```
SELECT
    Cat.CategoryName,
    SUM(OI.Subtotal) AS CategoryRevenue,
    ROUND(SUM(SUM(OI.Subtotal)) OVER (PARTITION BY Cat.CategoryID) * 100.0
/ SUM(SUM(OI.Subtotal)) OVER (), 2) AS RevenuePercentage
FROM Categories Cat
JOIN Products Prod ON Cat.CategoryID = Prod.CategoryID
JOIN OrderItems OI ON Prod.ProductID = OI.ProductID
GROUP BY Cat.CategoryID, Cat.CategoryName;
```

Section 4: Key Takeaways

Use **OVER (PARTITION BY ... ORDER BY ...)** with **SUM** for cumulative or grouped totals without collapsing rows. Join tables via natural or explicit conditions (e.g., CustomerID, OrderID) for multi-table analysis. Partition by grouping key (CustomerID, PaymentMode, City, CategoryID); order by date for running totals. Source confirmed data like payments over orders to exclude undelivered/canceled items. Practice identifies required tables and expected output first.