Lecture Notes – Sub Queries-FROM Clause

Section 1: Lecture Summary

This lecture focuses on writing **subqueries in the FROM clause**, the final placement of subqueries after coverage of WHERE, HAVING, and SELECT clauses. The key principle is that a subquery in the FROM clause must return a **table structure** (not a scalar value or list) to serve as a derived table that can be joined with other tables. The lecture demonstrates two practical examples from the eCommerceDB: calculating department salary expenses and category revenue, followed by a practice exercise on ranking customers by review count.

Section 2: Key Concepts and Explanations

Subquery in FROM Clause Syntax

A subquery in the FROM clause replaces a table reference. The basic structure is:

```
SELECT columns
FROM (SELECT columns FROM table WHERE condition) AS alias_name
```

The subquery result must produce a complete table structure with rows and columns, not just a single value. Each derived table requires an alias name for reference in the outer query.

When to Use Subqueries in FROM

Subqueries in the FROM clause improve **readability and simplicity** by separating complex operations. Instead of performing all aggregations and joins in a single query, you can:

1. Create an intermediate result set using GROUP BY and aggregate functions

2. Join that result with other tables in the outer query

This approach is particularly useful when you need to group data first, then join the grouped results with dimension tables.

Critical Requirement: Table Alias

Unlike inline views, derived tables from subqueries always require an alias. This alias allows the outer query to reference columns from the subquery result using dot notation (alias.column_name).

Multi-Table Involvement

When using subqueries in FROM, you may work with multiple tables:

- The subquery can join tables internally to produce the intermediate result

- The outer query then joins this result with additional tables

- This creates a logical separation between aggregation and joining operations

Section 3: Example Code and Use Cases

Example 1: Departments with Total Salary Expense

This query shows departments alongside their total salary expenditure by creating an intermediate table of department salary totals:

```
SELECT D.DepartmentID, D.DepartmentName, S.TotalSalary
FROM Departments D
JOIN (
```

```
    SELECT DepartmentID, SUM(Salary) AS TotalSalary
    FROM Employees
    GROUP BY DepartmentID
) S ON D.DepartmentID = S.DepartmentID
```

The subquery (aliased as S) groups employees by department and calculates total salary. The outer query joins this result with the Departments table to display both the department name and its salary total. This approach separates the aggregation step from the joining step, making the query easier to read and maintain.

Example 2: Categories with Total Revenue

This query calculates revenue by category from the eCommerceDB:

```
SELECT C.CategoryID, C.CategoryName, REV.TotalRevenue
FROM Categories C
JOIN (
    SELECT P.CategoryID, SUM(OI.Subtotal) AS TotalRevenue
    FROM OrderItems OI
    JOIN Products P ON OI.ProductID = P.ProductID
    GROUP BY P.CategoryID
) REV ON C.CategoryID = REV.CategoryID
```

The subquery joins OrderItems with Products and groups by CategoryID to calculate revenue per category. The outer query then joins this result with the Categories table. This structure handles the indirect relationship between Categories and OrderItems (they connect through Products) by establishing the join inside the subquery before aggregating.

Example 3: Customer Ranking by Review Count

To rank customers by the number of reviews they wrote:

```
SELECT C.CustomerID, C.FirstName, C.LastName, RC.ReviewCount
FROM Customers C
JOIN (
    SELECT CustomerID, COUNT(*) AS ReviewCount
    FROM Reviews
    GROUP BY CustomerID
) RC ON C.CustomerID = RC.CustomerID
ORDER BY RC.ReviewCount DESC
```

The subquery counts reviews per customer, and the outer query joins this with the Customers table to display customer names with their review counts in descending order.

Section 4: Key Takeaways

Derived Tables Must Have Structure

Subqueries in the FROM clause must return a complete table with multiple columns and rows. A single scalar value cannot serve as a table source.

Alias Naming is Mandatory

Every derived table from a subquery requires an alias. Use meaningful names that reflect the data's content (S for salary totals, REV for revenue) to improve query readability.

Separation of Concerns

Using subqueries in FROM logically separates aggregation operations (GROUP BY, SUM, COUNT) from join operations. This can make complex queries more understandable compared to combining everything in a single statement.

Performance Considerations

The query optimizer generally handles subqueries in FROM efficiently, but direct JOINs followed by GROUP BY may sometimes perform better. The choice between these approaches should prioritize code readability unless performance testing reveals a bottleneck.

Multi-Table Joins Within Subqueries

Subqueries in FROM can internally join multiple tables before returning results. This is useful when intermediate tables lack direct relationships with the outer query's tables.