Day 3 Assignment: Amaan Shaikh

Total Aggregations, Order of Execution, Rules and restrictions when Grouping and Filtering SQL Data

1.Order of Execution: (FWGH-SOOF)

FROM → WHERE → GROUP BY → HAVING → SELECT → ORDER BY → LIMIT/OFFSET → FETCH NEXT N ROWS ONLY

WHY?

Correct Results: Ensures you get accurate results by knowing how SQL processes the query step-by-step.

Efficient Query Writing: Helps in writing optimised and efficient queries by focusing on the parts that SQL evaluates first.

Debugging: Simplifies troubleshooting by understanding where errors or unexpected results might occur in the query.

Complex Queries: Aids in constructing complex queries, such as those involving subqueries or joins, by knowing the sequence in which they are executed.

Performance Tuning and Resource Optimisation: Improves query performance by allowing you to optimise the order of operations, like filtering data early with WHERE before joining tables.

Logical Flow: Helps in maintaining the logical flow of query execution, ensuring that clauses like GROUP BY, HAVING, and ORDER BY are used correctly.

SQL Query Example:

```
SELECT DepartmentID, AVG(Salary) AS AvgSalary
FROM Employees
WHERE Salary > 50000
GROUP BY DepartmentID
HAVING AVG(Salary) > 60000
ORDER BY AvgSalary DESC
OFFSET 10 ROWS
FETCH NEXT 5 ROWS ONLY;
```

Order of Execution Explanation:

1. **FROM**: The Employees table is selected as the source data.

2. **WHERE**: Filters rows where Salary is greater than 50,000. Only these rows proceed to the next step.
3. **GROUP BY**: Groups the filtered rows by DepartmentID. Aggregates like AVG(Salary) are calculated for each department.
4. **HAVING**: Filters the grouped results to include only those departments where the average salary is greater than 60,000.
5. **SELECT**: Selects the columns to be displayed: DepartmentID and the calculated AvgSalary.
6. **ORDER BY**: Orders the result set by AvgSalary in descending order.
7. **OFFSET**: Skips the first 10 rows of the ordered result set.
8. **FETCH**: Retrieves the next 5 rows after the offset has been applied.

Summary:

- The SQL engine starts by identifying the data source (FROM), filters data (WHERE), groups it (GROUP BY), applies post-group filtering (HAVING), selects and orders the final result (SELECT, ORDER BY), and finally handles pagination with OFFSET and FETCH.

Rules and Restrictions when Grouping and Filtering Data:

Grouping Data (GROUP BY):

1. **Columns in SELECT**: Columns in the SELECT clause must either be in the GROUP BY clause or be used in aggregate functions like SUM, COUNT, etc.
2. **Grouping Order**: The GROUP BY clause must appear after WHERE and before HAVING and ORDER BY.
3. **Grouping Multiple Columns**: You can group by multiple columns, which creates a unique combination for each group.
4. **No Aggregates in GROUP BY**: Aggregate functions cannot be used directly in the GROUP BY clause.
5. **NULL Values**: GROUP BY treats all NULL values as equal, so they are grouped together.

Filtering Data (WHERE and HAVING):

1. **WHERE Before GROUP BY**: The WHERE clause filters rows before grouping, meaning it operates on individual rows.
2. **HAVING After GROUP BY**: The HAVING clause filters groups after the GROUP BY clause has been applied, and it can use aggregate functions.
3. **No Aggregates in WHERE**: Aggregate functions cannot be used in the WHERE clause; they should be used in HAVING.
4. **Filtering Order**: WHERE filters rows before they are grouped, while HAVING filters after grouping has occurred.

General Restrictions:

1. **Column Aliases**: Aliases defined in the SELECT clause cannot be used in the WHERE, GROUP BY, or HAVING clauses.
2. **Non-Aggregated Columns**: Any column in the SELECT list not used in an aggregate function must be included in the GROUP BY clause.

Total Aggregation refers to the process of applying an aggregate function across an entire dataset to produce a single summary value. This is different from grouped aggregation, where data is first grouped by specific columns and then aggregated within those groups.

Example:

Assume we have a table, containing sales data.

SaleID	SaleAmount
1	100
2	150
3	200
4	250
5	300

SELECT SUM(SaleAmount) AS TotalSales FROM Sales;

- Is a query we can use to calculate the total sales amount from all records.

Result:

TotalSales
1000