Data Engineering Batch 1 **Day 6: 24/01/2024**

Name: Pradip Bochare

- Total Aggregation using OVER and PARTITION BY in SQL Queries:
  Over and Partition By clauses in SQL are used in conjunction with window functions to perform calculations on a specific subset of data within a result set. This is useful for creating subtotals and total aggregations.

```
mysql> CREATE TABLE employee (
    ->     employee_id INT PRIMARY KEY,
    ->     first_name VARCHAR(50),
    ->     last_name VARCHAR(50),
    ->     department VARCHAR(50),
    ->     salary DECIMAL(10, 2)
    -> );
Query OK, 0 rows affected (0.04 sec)

mysql> DESC employee;
+-------------+---------------+------+-----+---------+-------+
| Field       | Type          | Null | Key | Default | Extra |
+-------------+---------------+------+-----+---------+-------+
| employee_id | int           | NO   | PRI | NULL    |       |
| first_name  | varchar(50)   | YES  |     | NULL    |       |
| last_name   | varchar(50)   | YES  |     | NULL    |       |
| department  | varchar(50)   | YES  |     | NULL    |       |
| salary      | decimal(10,2) | YES  |     | NULL    |       |
+-------------+---------------+------+-----+---------+-------+
5 rows in set (0.01 sec)

mysql>
```

```
mysql> INSERT INTO employee (employee_id, first_name, last_name, department, salary)
    -> VALUES
    ->     (1, 'Pradip', 'Bochare', 'IT', 60000.00),
    ->     (2, 'Vedant', 'Joshi', 'HR', 55000.00),
    ->     (3, 'Vinay', 'Natkar', 'Finance', 70000.00),
    ->     (4, 'Vikrant', 'Pachbhai', 'Marketing', 62000.00),
    ->     (5, 'Abhishek', 'Zune', 'IT', 58000.00);
Query OK, 5 rows affected (0.02 sec)
Records: 5  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM employee;
+-------------+------------+-----------+------------+----------+
| employee_id | first_name | last_name | department | salary   |
+-------------+------------+-----------+------------+----------+
|           1 | Pradip     | Bochare   | IT         | 60000.00 |
|           2 | Vedant     | Joshi     | HR         | 55000.00 |
|           3 | Vinay      | Natkar    | Finance    | 70000.00 |
|           4 | Vikrant    | Pachbhai  | Marketing  | 62000.00 |
|           5 | Abhishek   | Zune      | IT         | 58000.00 |
+-------------+------------+-----------+------------+----------+
5 rows in set (0.00 sec)
```

- Calculate total salary for each department and overall average salary

```
mysql> SELECT
    ->     employee_id,
    ->     first_name,
    ->     last_name,
    ->     department,
    ->     salary,
    ->     SUM(salary) OVER (PARTITION BY department) AS department_total_salary,
    ->     AVG(salary) OVER () AS overall_avg_salary
    -> FROM
    ->     employee;
+-------------+------------+-----------+------------+----------+--------------------------+--------------------+
| employee_id | first_name | last_name | department | salary   | department_total_salary | overall_avg_salary |
+-------------+------------+-----------+------------+----------+--------------------------+--------------------+
|           3 | Vinay      | Natkar    | Finance    | 70000.00 |                 70000.00 |       61000.000000 |
|           2 | Vedant     | Joshi     | HR         | 55000.00 |                 55000.00 |       61000.000000 |
|           1 | Pradip     | Bochare   | IT         | 60000.00 |                118000.00 |       61000.000000 |
|           5 | Abhishek   | Zune      | IT         | 58000.00 |                118000.00 |       61000.000000 |
|           4 | Vikrant    | Pachbhai  | Marketing  | 62000.00 |                 62000.00 |       61000.000000 |
+-------------+------------+-----------+------------+----------+--------------------------+--------------------+
5 rows in set (0.01 sec)
```

- Calculate Subtotals and total aggregations

```
mysql> SELECT
    ->     department,
    ->     SUM(salary) AS department_total_salary,
    ->     AVG(salary) AS department_avg_salary
    -> FROM
    ->     employee
    -> GROUP BY
    ->     department
    -> WITH ROLLUP;
+------------+--------------------------+----------------------+
| department | department_total_salary | department_avg_salary |
+------------+--------------------------+----------------------+
| Finance    |                 70000.00 |          70000.000000 |
| HR         |                 55000.00 |          55000.000000 |
| IT         |                118000.00 |          59000.000000 |
| Marketing  |                 62000.00 |          62000.000000 |
| NULL       |                305000.00 |          61000.000000 |
+------------+--------------------------+----------------------+
5 rows in set (0.00 sec)
```

Snowflaking& Star schemas

- In data warehousing, snowflaking is a form of dimensional modelling in which dimensions are stored in multiple related dimension tables.
- A snowflake schema is a variation of the star schema that normalizes the dimension tables to increase data integrity, simplify data maintenance and reduce the amount of disk space. In certain situations, it can also improve query performance.
- A star schema consists of a central fact table that references multiple dimension tables.
- Each dimension table is denormalized ("flattened") to avoid the query overhead that comes with a highly normalized schema, which can require a large number of joins to retrieve the necessary data.

❖ **Purpose of the snowflake schema**

- The normalized dimensions in a snowflake schema reduce the amount of redundant data, making them less susceptible to data integrity issues than a star schema.
- Whenever multiple copies of the same data are stored in a database, as is the case with the star schema, there is a greater risk that extract, load and transform (ELT) operations will result in problems with the data.
- With a snowflake schema, data maintenance is easier and less likely to cause data integrity issues. There is less redundant data, and that data is organized into separate tables, simplifying the processes of adding data and updating data.

 Rules and Restrictions to Group and Filter Data in SQL queries

- GROUP BY enables you to use aggregate functions on groups of data returned from a query.
- FILTER is a modifier used on an aggregate function to limit the values used in an aggregation. All the columns in the select statement that aren't aggregated should be specified in a GROUP BY clause in the query.

❖ GROUP BY

Returning to a previous section, when we were working with aggregations, we used the aggregate function AVG to find out the average deal size. If we wanted to know the average value of the deals won by each sales person from highest average to lowest, the query would look like:

```
SELECT sales_agent,
   AVG(close_value)
FROM sales_pipeline
WHERE sales_pipeline.deal_stage = "Won"
GROUP BY sales_agent
ORDER BY AVG(close_value) DESC
```

❖ FILTER

If you wanted to refine your query even more by running your aggregations against a limited set of the values in a column you could use the FILTER keyword. For example, if you wanted to know both the number of deals won by a sales agent and the number of those deals that had a value greater than 1000, you could use the query

```
SELECT sales_agent,
   COUNT(sales_pipeline.close_value) AS total,
   COUNT(sales_pipeline.close_value)
FILTER(WHERE sales_pipeline.close_value > 1000) AS `over 1000`
FROM sales_pipeline
WHERE sales_pipeline.deal_stage = "Won"
GROUP BY sales_pipeline.sales_agent
```

Order of Execution of SQL Queries

- SQL order of execution refers to the sequence in which different clauses and operations within a SQL query are processed by the database management system.
- Each SQL query consists of various components such as SELECT, FROM, WHERE, GROUP BY, HAVING, and ORDER BY clauses, along with functions and operators. Understanding the order in which these components are executed is vital for producing accurate and efficient query results

❖ Importance of SQL Order of Execution

- Accurate Results: Understanding the order in which different components of a SQL query are executed ensures that you retrieve accurate and relevant results.
- Query Optimization: Proficiency in SQL order of execution enables you to optimize your queries for better performance by minimizing the amount of data processed and improving query execution times.
- Efficient Resource Utilization: Efficient queries consume fewer database resources, such as memory and processing power, leading to faster response times and reduced strain on the database server.
- Reduced Query Complexity: A solid grasp of the order of execution helps simplify complex queries. By breaking down a query into its individual steps, you can tackle each part separately, making it easier to understand, troubleshoot, and modify as needed.

❖ Stages of SQL Order of Execution

Clause	Order	Description
FROM	1	The query begins with the FROM clause, where the database identifies the tables involved and accesses the necessary data.
WHERE	2	The database applies the conditions specified in the WHERE clause to filter the data retrieved from the tables in the FROM clause.
GROUP BY	3	If a GROUP BY clause is present, the data is grouped based on the specified columns, and aggregation functions (such as SUM(), AVG(), COUNT()) are applied to each group.
HAVING	4	The HAVING clause filters the aggregated data based on specified conditions.
SELECT	5	The SELECT clause defines the columns to be included in the final result set.
ORDER BY	6	If an ORDER BY clause is used, the result set is sorted according to the specified columns.
LIMIT/OFFSET	7	If LIMIT or OFFSET clause is present, the result set is restricted to the specified number of rows and optionally offset by a certain number of rows.

```
SELECT product_category, AVG(price) AS avg_price
FROM products
WHERE stock_quantity > 0
GROUP BY product_category
HAVING AVG(price) > 50
ORDER BY avg_price DESC
LIMIT 5;
```

Differences Between UNION, EXCEPT and INTERSECT Operators in SQL Server

1. UNION: Combine two or more result sets into a single set without duplicates.
2. UNION ALL: Combine two or more result sets into a single set, including all duplicates.
3. INTERSECT: Takes the data from both result sets, which are in common.
4. EXCEPT: Takes the data from the first result set but not in the second result set (i.e., no matching to each other)

❖ **Rules on Set Operations:**

1. The result sets of all queries must have the same number of columns.
2. In every result set, the data type of each column must be compatible (well-matched) with the data type of its corresponding column in other result sets.
3. An ORDER BY clause should be part of the last select statement to sort the result. The first select statement must find out the column names or aliases.

❖ UNION Operator:

- The Union operator will return all the unique rows from both queries. Notice that the duplicates are removed from the result set.
- Purpose: The UNION operator combines the result sets of two or more SELECT statements into a single result set.
- Distinct Values: It removes duplicate rows between the various SELECT statements.
- Use Case: You would use UNION when listing all distinct rows from multiple tables or queries.

❖ INTERSECT Operator:

- The INTERSECT operator retrieves the common unique rows from the left and right queries. Notice the duplicates are removed.
- Purpose: The INTERSECT operator returns all rows common to both SELECT statements.
- Distinct Values: Like UNION and EXCEPT, INTERSECT also removes duplicates.
- Use Case: You would use INTERSECT when you need to find rows that are shared between two tables or queries.

❖ EXCEPT Operator:

- The EXCEPT operator will return unique rows from the left query that aren't in the right query's results.

- Purpose: The EXCEPT operator returns all rows from the first SELECT statement that are absent in the second SELECT statement's results.
- Distinct Values: It automatically removes duplicates.
- Use Case: EXCEPT is used when you want to find rows in one query that are not found in another. It's useful for finding differences between tables or queries.

❖ Differences Between UNION EXCEPT and INTERSECT Operators in SQL Server

❍ UNION Operator:

- The UNION operator combines the result sets of two or more SELECT statements into a single result set.
- It removes duplicate rows from the combined result set by default.
- The columns in the SELECT statements must have compatible data types, and the number of columns in each SELECT statement must be the same.
- The order of rows in the final result set may not be the same as in the individual SELECT statements unless you use the ORDER BY clause.

❍ EXCEPT Operator:

- The EXCEPT operator retrieves the rows present in the first result set but not in the second result set.
- It returns distinct rows from the first result set that do not have corresponding rows in the second result set.
- The columns in both SELECT statements must have compatible data types, and the number of columns in both statements must be the same.

❍ INTERSECT Operator:

- The INTERSECT operator is used to retrieve the rows that are common to both result sets.
- It returns distinct rows appearing in the first and second result sets.
- The columns in both SELECT statements must have compatible data types, and the number of columns in both statements must be the same.