Disicussion-5 The GROUP BY and HAVING clauses

Discussion Topic:

Using the database you installed from Module 1, provide an example query using both a group by clause and a having clause. Show no more than ten rows of your query result. Discuss if the query you wrote can be rewritten without those clauses.

My Post:

Hello Class,

Structured Query Language (SQL) is used within Relational Database Management Systems (RDBMS) to create and manipulate databases. MySQL is one of the various RDBMS frameworks that supports *the GROUP BY* and *HAVING* clauses for data aggregation. Data aggregation, in the context of RDBMS, is the process of taking multiple rows of data and summarizing them into a single result row, based on certain stated conditions or criteria. The `GROUP BY` and `HAVING` clauses are used in conjunction with aggregate functions like `COUNT()`, `SUM()`, `AVG()`, `MAX()`, and `MIN()`.

The GROUP BY Clause

The `GROUP BY` clause is used to partition identical data (records) into groups (DataCamp, n.d.a). It is often used in conjunction with aggregate functions that perform calculations on the groups created by the clause. For example:

```
SELECT product_category, SUM(sale_amount) AS total_sales
FROM sales
GROUP BY product_category
```

Output Example:

product_category	total_sales
electronics	3500
books	120
clothing	150

Explanation of the Code:

1. `SELECT product_category, SUM(sale_amount) AS total_sales`
   - `product_category`: The `product_category` column will be included in the results.
   - `SUM(sale_amount) AS total_sales`: Aggregate function that calculates the sum of values in the `sale_amount` column and assigns to results to column with the alias `total_sales`
2. `FROM sales`
   - The table from which the data is retrieved.
3. `GROUP BY product_category`
   - Partition the records (rows) with identical `product_category` values into groups. For example, if there are multiple sales records for "Electronics" (`product_category`), all these

records will form one group; all sales records for "Books" (`product_category`) will form another group, and so on.

- Then the aggregate function `SUM(sale_amount)` will compute the sum of the `sale_amount` and the results are stored in the column `total_sales` (`AS total_sales`).

The HAVING Clause

The `HAVING` Clause is used to filter identical data partitioned by the `Group By` clause (DataCamp, n.d.b). Note that the clause after For example:

```
SELECT product_category, SUM(sale_amount) AS total_sales
FROM sales
GROUP BY product_category
HAVING SUM(sale_amount) > 200
```

Output Example:

product_category	total_sales
electronics	3500

Explanation of the Code:

1. `SELECT product_category, SUM(sale_amount) AS total_sales`
   - `product_category`: The `product_category` column will be included in the results.
   - `SUM(sale_amount) AS total_sales`: Aggregate function that calculates the sum of values in the `sale_amount` column and assigns to results to column with the alias `total_sales`
2. `FROM sales`
   - The table from which the data is retrieved.
3. `GROUP BY product_category`
   - Partition the records (rows) with identical `product_category` values into groups. For example, if there are multiple sales records for "Electronics" (`product_category`), all these records will form one group; all sales records for "Books" (`product_category`) will form another group, and so on.
   - Then the aggregate function `SUM(sale_amount)` will compute the sum of the `sale_amount` and the results are stored in the column `total_sales` (`AS total_sales`).
4. `HAVING SUM(sale_amount) > 200`
   - The `HAVING` clause filters the groups that were created by the `GROUP BY` clause. The clause is applied after the grouping has occurred and the sum of the `sale_amount` has been computed for each group.
   - `SUM(sale_amount) > 200` is the condition. Only the groups (`product_categories`) with a total sum (`sale_amount`) greater than `200` will be included in the final result.

Rewriting Example

In this section, the provided example below using the GROUP BY and HAVING clauses is rewritten by utilizing first correlated subqueries and then MySQL window functions to showcase approaches to achieve the same data aggregation but different methods.

On a side note: the ORDER BY clause is a SQL command used to sort query results in either ascending or descending order based on one or more columns (Kartik, 2024). MySQL query statement utilizing GROUP BY clause in conjunction with aggregate functions often uses the ORDER BY to arrange the aggregated results in a more meaningful way, for example, by ranking the groups based on their computed aggregate results (e.g., showing product categories with the highest SUM(sales) first, or customers with the COUNT(orders) in descending order). Below is an example illustrating the combined use of the GROUP BY, HAVING, and ORDER BY clauses. Here we want to identify and rank customers who have multiple orders by the number of orders they made.

```
SELECT
    customer_id,
    COUNT(order_id) AS total_orders
FROM
    orders
GROUP BY
    customer_id
HAVING
    COUNT(order_id) > 1
ORDER BY
    total_orders DESC
LIMIT 10;
```

Output Example:

customer_id	total_orders
1	2
4	2

Explanation of the Code:

1. SELECT customer_id, COUNT(order_id) AS total_orders
   - Selects the customer_id column.
   - COUNT(order_id) counts the number of orders for each customer.
   - AS total_orders gives an alias to the calculated count column
2. FROM orders
   - The table from which the data is retrieved.
3. GROUP BY customer_id
   - Partition the records (rows) with identical customer_id values into groups.
   - Then the aggregate function COUNT(order_id) will count the number of rows (records) in the order_id column, including the row with NULL values.
4. HAVING COUNT(order_id) > 1

   - The `HAVING` clause filters the groups that were created by the `GROUP BY` clause. The clause is applied after grouping has occurred and the count of the `total_orders` has been computed for each group.
   - `COUNT(order_id) > 1` is the condition. Only the groups (`total_orders`) with a total count superior then `1` will be included in the final result

5. `ORDER BY total_orders DESC`
   - Sorts the `total_orders` results in descending order.
6. `LIMIT 10`
   - This limits the number of `total_orders` results output to a maximum of ten rows.

Rewriting The Query Without the GROUP BY and HAVING Clauses

By Using Correlated Subqueries

It is possible to rewrite the last example without using the `GROUP BY` and `HAVING` clauses. For example, by using subqueries (subqueries, in MySQL, are queries embedded in other queries):

```
SELECT
    customer_id,
    total_orders
FROM
    (   -- Subquery substitute for the GROUP BY clause
        SELECT
            cust_with_orders.customer_id,
            (   -- Sub-subquery-2 counts the total number of orders for each distinct customer id
                SELECT COUNT(order_id)
                FROM orders
                WHERE customer_id = cust_with_orders.customer_id
            ) AS total_orders
        FROM
            (   -- Sub-subquery-1 queries all unique customer ids that exist in the orders table
                SELECT DISTINCT customer_id
                FROM orders
            ) AS cust_with_orders
    ) AS group_orders_by_customer_id  -- output table
WHERE -- substitute for the HAVE clause
    total_orders > 1
ORDER BY
    total_orders DESC
LIMIT 10;
```

Output Example:

customer_id	total_orders
1	2
4	2

Explanation of the Code: (For the sake of length, the code explanation is summarized)

1. Sub-subquery-1, a list (`cust_with_orders`) of all distinct customer ids from the `orders` table is created. Note that the created list is a table consisting of one column (`customer_id`).
2. Then, sub-subquery-2, counts for each distinct customer id (`cust_with_orders.customer_id`), their number of orders from the `orders` table is counted, and outputs a list of total orders (`total_orders`). Note that the output is a column.
3. These results (`total_orders`) and their related customer ids (`cust_with_orders.customer_id`) are merged in a table with the alias `group_orders_by_customer_id`.
   Note that the combination of the previous steps simulates the `GROUP BY` clause functionality.
4. Then the values of the `group_orders_by_customer_id` are filtered using a `WHERE` clause with the condition `total_orders > 1`. Note that the `WHERE` clause is the substitute for the `HAVE` clause.

Although the query using the `GROUP BY` and `HAVING` clauses can be written by using correlated subqueries is more verbose, making it more difficult to read, and is more often than not less performant for aggregation operations than the `GROUP BY` clause. Another alternative is to use window functions, which are supported by MySQL.

Using Window Functions to Rewrite the Query

Window functions can perform aggregations, rankings, distributions, and more without collapsing the entire result set down to a single row (PlanetScale, 2025). It is possible to rewrite the query example without using the `GROUP BY` and `HAVING` clauses. For example:

```sql
-- Window function
-- The function counts the number of orders per customer id in the order table
-- it outputs a table where each original row (containing a customer_id)
-- also contains the total number of orders for that specific customer_id
WITH customer_order_counts AS (
    SELECT
        customer_id,
        -- The 'orders' table's data is partitioned by customer_id
        -- and each occurrence of the customer_id is associated with
        --  the total number of orders for that specific customer_id
        -- computed by the COUNT(order_id) clause
        COUNT(order_id) OVER (PARTITION BY customer_id) AS total_orders
    FROM
        orders
)
-- -------------------------------------------------------------------
-- This is the Main query
SELECT DISTINCT
    customer_id, -- Only select distinct customer_id
    total_orders
FROM
```

```
    customer_order_counts
WHERE
    total_orders > 1
ORDER BY
    total_orders DESC
LIMIT 10;
```

Output Example:

customer_id	total_orders
1	2
4	2

Explanation of the Code: (For the sake of length, the code explanation is summarized)

1. The query first calculates the total number of orders for every customer ids from the `orders` table and attaches the respective totals to each occurrent `customer_id` using the `customer_order_counts` window function. Not that this function creates duplicate rows
2. Then, it filters distinct customer ids (from the `customer_order_counts` table) with their total order counts, then filters out customers who have only made 1 order, sorts the remaining customers by their total order count (highest first), and finally, it only returns the top 10 row results.

Similarly to using the correlated subqueries, the query can be rewritten using window functions. The query is more verbose than the one using `GROUP BY` clause, making it less readable and concise. It is also less performant, as it computes the total order count for every individual row of the `orders` table, creating duplicates. Window functions should be used in scenarios like calculating per-row values like ranks, running totals, or lead/lag comparisons, and where keeping the individual row context is needed.

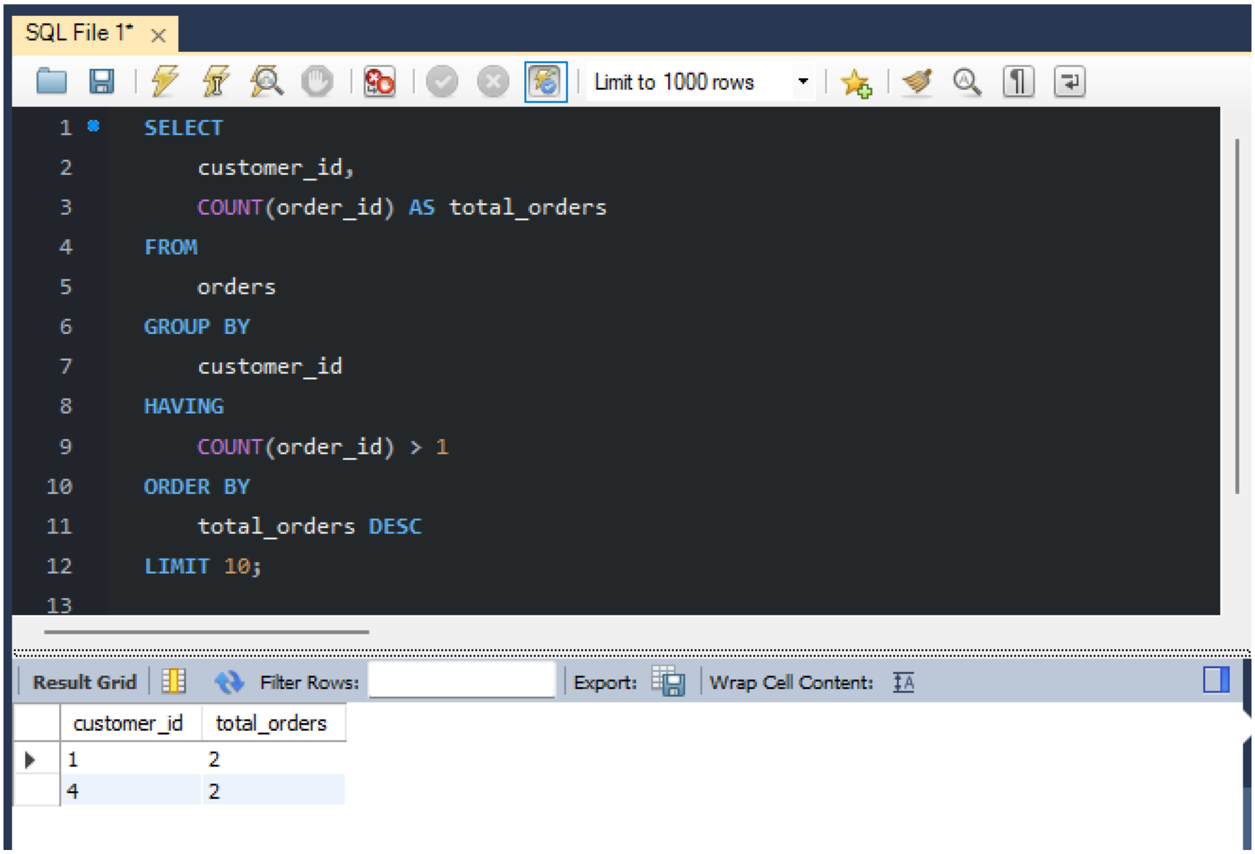
-Alex

References:

DataCamp (n.d.a). *MySQL GROUP BY clause.* DataCamp. https://www.datacamp.com/doc/mysql/mysql-group-by

DataCamp (n.d.b). *MySQL HAVING clause.* DataCamp. https://www.datacamp.com/doc/mysql/mysql-having

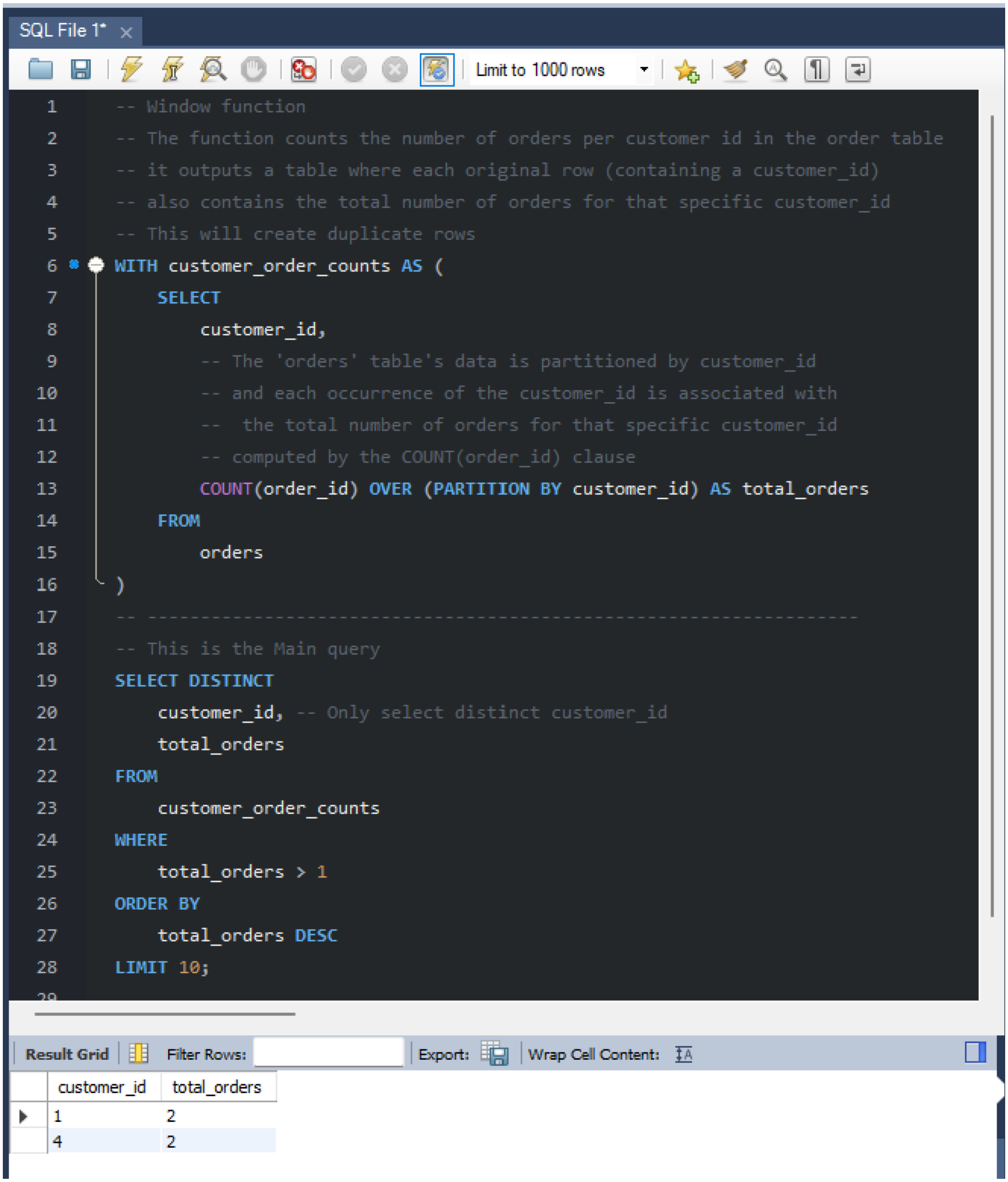
PlanetScale (2025). 3.15 Window functions. MySQL for developers. PlanetScale. https://planetscale.com/learn/courses/mysql-for-developers/queries/window-functions?autoplay=1



```sql
SELECT
    customer_id,
    COUNT(order_id) AS total_orders
FROM
    orders
GROUP BY
    customer_id
HAVING
    COUNT(order_id) > 1
ORDER BY
    total_orders DESC
LIMIT 10;

```

customer_id	total_orders
1	2
4	2



```sql
-- Window function
-- The function counts the number of orders per customer id in the order table
-- it outputs a table where each original row (containing a customer_id)
-- also contains the total number of orders for that specific customer_id
-- This will create duplicate rows
WITH customer_order_counts AS (
    SELECT
        customer_id,
        -- The 'orders' table's data is partitioned by customer_id
        -- and each occurrence of the customer_id is associated with
        --  the total number of orders for that specific customer_id
        -- computed by the COUNT(order_id) clause
        COUNT(order_id) OVER (PARTITION BY customer_id) AS total_orders
    FROM
        orders
)
-- -------------------------------------------------------------------
-- This is the Main query
SELECT DISTINCT
    customer_id, -- Only select distinct customer_id
    total_orders
FROM
    customer_order_counts
WHERE
    total_orders > 1
ORDER BY
    total_orders DESC
LIMIT 10;
```

customer_id	total_orders
1	2
4	2