SUMMARY: SUBQUERIES

SESSION OVERVIEW:

By the end of this session, the students will be able to:

- Understand the use cases of CTE in SQL.
- Understand the subqueries and nested queries.

KEY TOPICS AND EXAMPLES:

Understanding the use cases of CTE:

CTE is the short form for Common Table Expressions. CTE is one of the most powerful tools of SQL, and it also helps to clean the data. It is the concept of SQL used to simplify coding and help to get the result as quickly as possible. CTE is the temporary table used to reference the original table. If the original table contains too many columns and we require only a few of them, we can make CTE (a temporary table) containing the required columns only.

Functions of CTE:

- **Simplification of Complex Queries:** CTEs break down complex queries into simpler, more manageable parts. This helps in understanding and debugging SQL code.
- **Improved Readability:** By using descriptive names for CTEs, the purpose and logic of each part of the query become clearer. This makes the overall query easier to read and maintain.
- **Reusability within Queries:** A CTE can be referenced multiple times within the main query, avoiding the need to repeat the same subquery logic multiple times. This reduces redundancy and potential errors.
- **Modularity:** Queries can be structured more modularly, where each CTE represents a distinct step or logic. This modular approach aligns with best practices in software development, making the query easier to manage and extend.
- **Support for Recursive Queries:** CTEs support recursion, allowing for the processing of hierarchical or tree-structured data. This is particularly useful for operations like traversing organizational charts or managing recursive relationships.
- **Temporary Scope:** CTEs exist only for the duration of the query in which they are defined. This means they do not create temporary tables or persist beyond the execution of the main query, ensuring efficient use of resources.

Syntax of CTE:

```
WITH CTE_NAME AS
```

```
(
  SELECT column_name1, column_name2,..., column_nameN
  FROM table_name
  WHERE condition
)
SELECT column_name1, column_name2,..., column_nameN
FROM CTE_NAME;
```

NOTE:

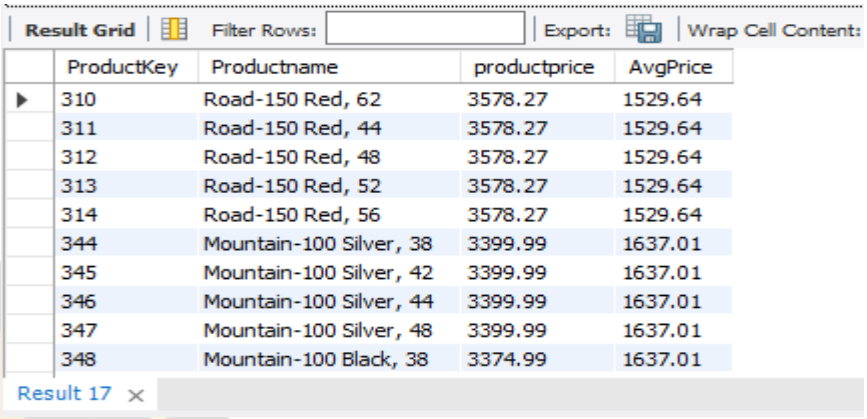
- *WITH Clause: Starts the definition of one or more CTEs.*
- *CTE Name: An identifier for the CTE that will be used in subsequent queries.*
- *Optional Column List: A list of column names for the CTE result set. This is optional and usually used when the CTE query does not specify column aliases.*
- *CTE Query: The query that defines the CTE. This can be any valid SQL query that returns a result set.*
- *Main Query: The query that uses the CTE. This can reference the CTE as if it were a regular table or view.*

Different ways how CTE can be useful:

Example 1:

```
WITH AverageCategoryPrice AS (
    SELECT psc.productsubcategorykey,
           ROUND(AVG(p.productprice), 2) AS AvgPrice
    FROM products p
    JOIN product_subcategories psc ON p.productsubcategorykey =
psc.productsubcategorykey
    GROUP BY psc.productsubcategorykey
)
SELECT p.ProductKey, p.Productname, p.productprice, acp.AvgPrice
FROM products p
JOIN AverageCategoryPrice acp ON p.productsubcategorykey =
acp.productsubcategorykey
WHERE p.productprice > acp.AvgPrice
ORDER BY p.productprice DESC;
```

Output:



Result Grid | Filter Rows: | Export: | Wrap Cell Content:

ProductKey	Productname	productprice	AvgPrice
310	Road-150 Red, 62	3578.27	1529.64
311	Road-150 Red, 44	3578.27	1529.64
312	Road-150 Red, 48	3578.27	1529.64
313	Road-150 Red, 52	3578.27	1529.64
314	Road-150 Red, 56	3578.27	1529.64
344	Mountain-100 Silver, 38	3399.99	1637.01
345	Mountain-100 Silver, 42	3399.99	1637.01
346	Mountain-100 Silver, 44	3399.99	1637.01
347	Mountain-100 Silver, 48	3399.99	1637.01
348	Mountain-100 Black, 38	3374.99	1637.01

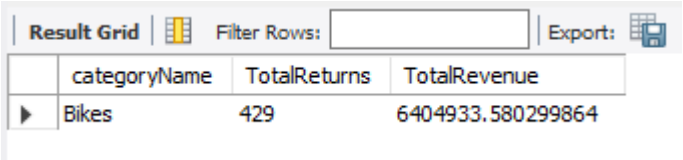
Result 17

Example 2:

Note: Example 2 and 3 imply the way of using multiple CTEs in SQL which helps us optimize our query.

```
WITH ReturnsByCategory AS (
    SELECT pc.categoryName,
           SUM(r.returnQuantity) AS TotalReturns
    FROM returns r
    JOIN products p ON r.productkey = p.productkey
    JOIN product_subcategories psc ON p.productsubcategorykey =
psc.productsubcategorykey
    JOIN product_categories pc ON psc.productcategorykey =
pc.productcategorykey
    GROUP BY pc.categoryName
),
RevenueByCategory AS (
    SELECT pc.categoryName,
           SUM(p.productprice * s.orderquantity) AS TotalRevenue
    FROM sales_2015 s
    JOIN products p ON s.productkey = p.productkey
    JOIN product_subcategories psc ON p.productsubcategorykey =
psc.productsubcategorykey
    JOIN product_categories pc ON psc.productcategorykey =
pc.productcategorykey
    GROUP BY pc.categoryName
)
SELECT rbc.categoryName, rbc.TotalReturns, rvc.TotalRevenue
FROM ReturnsByCategory rbc
JOIN RevenueByCategory rvc ON rbc.categoryName = rvc.categoryName
ORDER BY rbc.TotalReturns DESC;
```

Output:



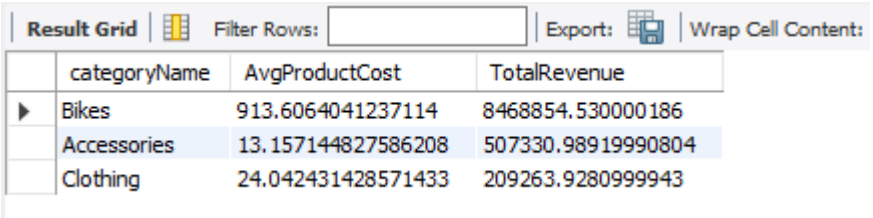
Result Grid | Filter Rows: | Export:

	categoryName	TotalReturns	TotalRevenue
▶	Bikes	429	6404933.580299864

Example 3:

```
WITH AvgCost AS (
    SELECT
        pc.categoryName,
        AVG(p.productcost) AS AvgProductCost
    FROM products p
    JOIN product_subcategories psc ON p.productsubcategorykey =
psc.productsubcategorykey
    JOIN product_categories pc ON psc.productcategorykey =
pc.productcategorykey
    GROUP BY pc.categoryName
),
RevenueByCategory AS (
    SELECT
        pc.categoryName,
        SUM(p.productprice * s_2017.orderquantity) AS TotalRevenue
    FROM sales_2017 s_2017
    JOIN products p ON s_2017.productkey = p.productkey
    JOIN product_subcategories psc ON p.productsubcategorykey =
psc.productsubcategorykey
    JOIN product_categories pc ON psc.productcategorykey =
pc.productcategorykey
    GROUP BY pc.categoryName
)
SELECT ac.categoryName, ac.AvgProductCost, rbc.TotalRevenue
FROM AvgCost ac
JOIN RevenueByCategory rbc ON ac.categoryName = rbc.categoryName
ORDER BY rbc.TotalRevenue DESC;
```

Output:



Result Grid | Filter Rows: | Export: | Wrap Cell Content:

	categoryName	AvgProductCost	TotalRevenue
▶	Bikes	913.6064041237114	8468854.530000186
	Accessories	13.157144827586208	507330.98919990804
	Clothing	24.042431428571433	209263.9280999943

Understanding the use of sub-queries in SQL:

1. **What is sub-queries/nested queries:**

Subqueries and nested queries are terms that are often used interchangeably in SQL, but they refer to the same concept: a query embedded within another query.
A subquery is a SQL query that is nested inside another SQL query. It is executed independently, and its result is used in the outer query for filtering, joining, or other purposes. Subqueries can be used in different parts of the outer query, such as the SELECT, FROM, WHERE, HAVING, or JOIN clauses.

2. **Why do we need sub-queries in SQL:**

It is very important to understand what are the use cases or the requirements of using sub-queries in SQL. Here are some of the reasons which make this a very important topic for a data analyst:

- **Data Retrieval from Multiple Tables**: Subqueries allow you to combine and retrieve data from multiple tables in a single query. This is particularly useful when you need to fetch data based on conditions involving data from other tables.
- **Complex Calculations**: Subqueries can be used to perform complex calculations that would be difficult or impossible to express in a single query. This includes calculations involving aggregations, window functions, or correlated data.
- **Filtering with Values from Other Tables**: Subqueries enable you to filter data based on values from other tables. For example, you can retrieve products that have a price greater than the average price of all products by using a subquery to calculate the average price.
- **Dynamic Values in Conditions**: Subqueries can provide dynamic values for conditions in the `WHERE`, `HAVING`, or `JOIN` clauses. This allows you to create more flexible and adaptable queries.
- **Reusability and Modularity:** By encapsulating complex logic or data retrieval within a subquery, you can improve the reusability and modularity of your SQL code, making it easier to maintain and understand.

Rules to be followed:

- *Subqueries must be enclosed within parentheses.*
- *Subqueries can be nested within another subquery.*
- *A subquery must contain the SELECT query and the FROM clause always.*
- *A subquery consists of all the clauses an ordinary SELECT clause can contain: GROUP BY, WHERE, HAVING, DISTINCT, TOP/LIMIT, etc. However, an ORDER BY clause is only used when a TOP clause is specified.*

3. **Examples of subquery in SELECT statement:**

Syntax:

```
SELECT column1, column2, ...,
```

```
       (subquery)
FROM table1
[WHERE condition];
```

NOTE:

- *The subquery is enclosed in parentheses () and is treated as a scalar expression, meaning it returns a single value.*
- *The subquery can be used in the SELECT list, along with other columns or expressions.*
- *The subquery can reference columns from the outer query or other tables.*
- *The result of the subquery is substituted for each row in the outer query.*

Example 1:

```
SELECT ps.SubcategoryName, (
    SELECT ROUND(AVG(p.ProductCost), 2)
    FROM Products p
    WHERE p.ProductSubcategoryKey = ps.ProductSubcategoryKey
) AS AvgProductCost
FROM Product_Subcategories ps;
```

NOTE:

Subquery Operation: *Inside the main query, there's a smaller query that calculates the average cost of products. This smaller query finds the average cost of all items within a specific subcategory, rounding the result to two decimal places. It uses the product details to filter items that match the current subcategory being processed.*

Main Query Operation: *The main query retrieves the name of each subcategory. For each subcategory, it also retrieves the result of the smaller query (the average product cost). The subcategory details are the source of the data for this process.*

Output:

Result Grid | Filter Rows:

SubcategoryName	AvgProductCost
Mountain Bikes	906.21
Road Bikes	933.27
Touring Bikes	885.93
Handlebars	30.52
Bottom Brackets	40.95
Brakes	47.29
Chains	8.99
Cranksets	123.87

Result 26

4. **Examples of subquery in FROM statement:**

Syntax:

```
SELECT column1, column2, ...
```

```
FROM (
  subquery
) AS alias
[WHERE condition];
```

NOTE:

- *The subquery is enclosed in parentheses ().*
- *An alias (temporary table name) is assigned to the subquery using the AS keyword. This alias is then used to reference the columns returned by the subquery.*
- *The subquery can be a simple SELECT statement or a more complex query involving joins, aggregations, or other operations.*
- *The outer query can then reference the columns from the subquery's result set using the alias, just like referencing columns from a regular table.*

Example 1:

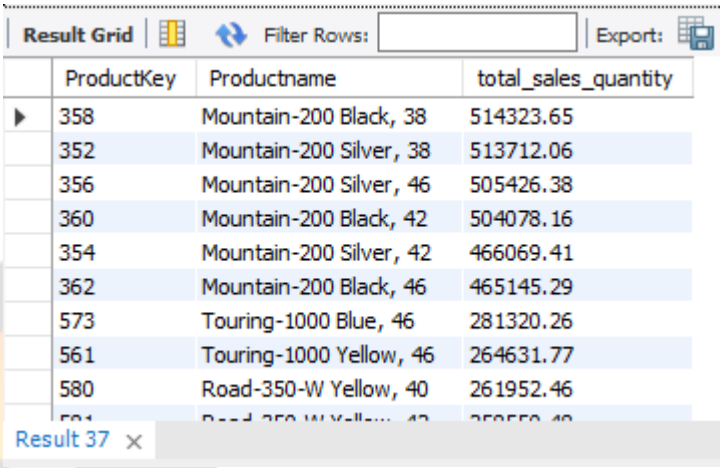
```
SELECT p.ProductKey, p.Productname, sub.total_sales_quantity
FROM products p,
    (SELECT
        s.productkey,
        ROUND(SUM(s.orderquantity * p.productPrice), 2) AS
total_sales_quantity
    FROM sales_2017 s
    JOIN products p on p.ProductKey = s.productkey
    GROUP BY s.productkey) sub
WHERE p.ProductKey = sub.productkey
ORDER BY sub.total_sales_quantity DESC
LIMIT 10;
```

NOTE:

- ***Subquery Operation:*** *The subquery calculates the total sales quantity for each product. It multiplies the order quantity by the product price and sums this value for each product. The result is rounded to two decimal places and grouped by each product.*

- ***Main Query Operation:*** *The main query selects the product key, product name, and total sales quantity from the subquery. It matches products from the main product list with the results of the subquery based on the product key. The results are sorted by total sales quantity in descending order, and only the top 10 products are returned.*

Output:



Result Grid | Filter Rows: | Export:

ProductKey	Productname	total_sales_quantity
358	Mountain-200 Black, 38	514323.65
352	Mountain-200 Silver, 38	513712.06
356	Mountain-200 Silver, 46	505426.38
360	Mountain-200 Black, 42	504078.16
354	Mountain-200 Silver, 42	466069.41
362	Mountain-200 Black, 46	465145.29
573	Touring-1000 Blue, 46	281320.26
561	Touring-1000 Yellow, 46	264631.77
580	Road-350-W Yellow, 40	261952.46

Result 37

5. **Examples of subquery in WHERE statement:**

Syntax:

```
SELECT column1, column2, ...
FROM table1
WHERE column_expression [NOT] operator (
  subquery
);
```

NOTE:

- ***The subquery is enclosed in parentheses () after a comparison operator (=, >, <, >=, <=, <>, !=, IN, NOT IN, EXISTS, or NOT EXISTS).***
- ***The subquery can be a simple SELECT statement or a more complex query involving joins, aggregations, or other operations.***
- ***The result of the subquery is compared with the column expression from the outer query using the specified operator.***
- ***The NOT keyword can be used to negate the condition if needed.***
- ***The outer query retrieves rows where the condition in the WHERE clause is satisfied.***

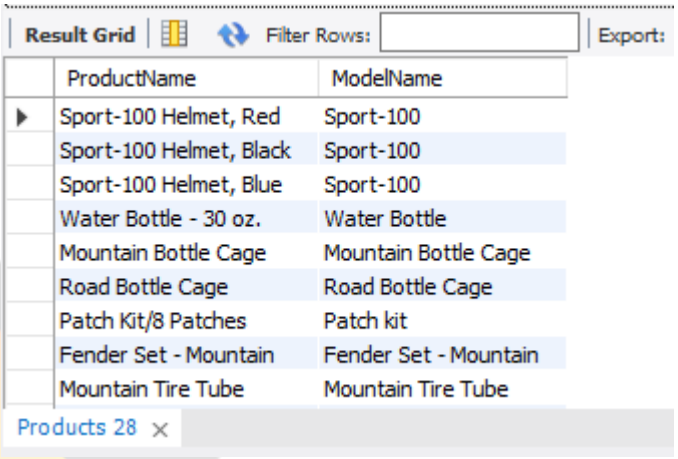
Example 1:

```
SELECT p.ProductName, p.ModelName
FROM Products p
WHERE (
    SELECT SUM(r.ReturnQuantity)
    FROM Returns r
    WHERE r.ProductKey = p.ProductKey
) > 50;
```

NOTE:

This query is looking for products that have been returned more than 50 times. It does this by checking each product's return quantity in the Returns table. If the sum of return quantities for a product is greater than 50, that product is included in the results.

Output:



Result Grid | Filter Rows: | Export:

ProductName	ModelName
Sport-100 Helmet, Red	Sport-100
Sport-100 Helmet, Black	Sport-100
Sport-100 Helmet, Blue	Sport-100
Water Bottle - 30 oz.	Water Bottle
Mountain Bottle Cage	Mountain Bottle Cage
Road Bottle Cage	Road Bottle Cage
Patch Kit/8 Patches	Patch kit
Fender Set - Mountain	Fender Set - Mountain
Mountain Tire Tube	Mountain Tire Tube

Products 28

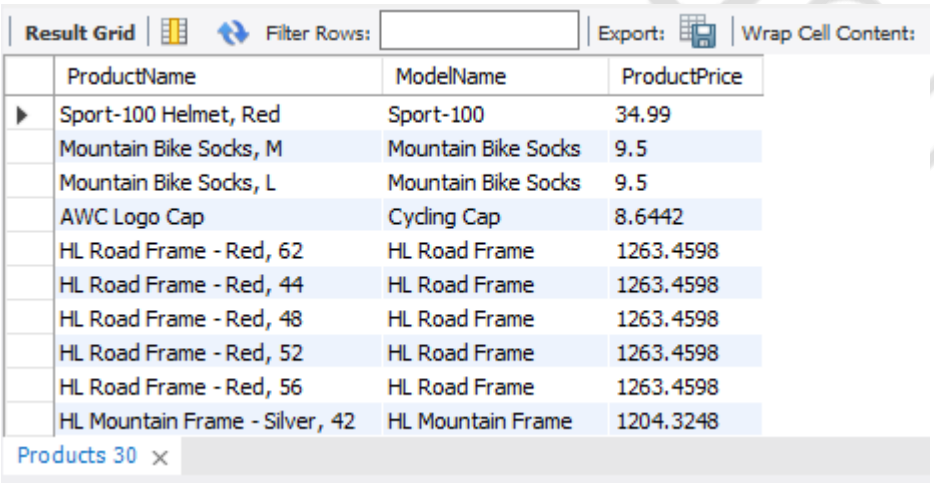
Example 2:

```
SELECT p.ProductName, p.ModelName, p.ProductPrice
FROM Products p
WHERE p.ProductPrice > (
    SELECT ROUND(AVG(p2.ProductPrice), 2)
    FROM Products p2
    WHERE p2.ProductSubcategoryKey = p.ProductSubcategoryKey
);
```

NOTE:

This query retrieves the product name, model name, and price for products that have a price higher than the average price of all products in the same subcategory. It does this by comparing each product's price to the average price of products in its subcategory, ensuring that only products with a higher price are included in the results.

Output:



Result Grid | Filter Rows: | Export: | Wrap Cell Content:

ProductName	ModelName	ProductPrice
Sport-100 Helmet, Red	Sport-100	34.99
Mountain Bike Socks, M	Mountain Bike Socks	9.5
Mountain Bike Socks, L	Mountain Bike Socks	9.5
AWC Logo Cap	Cycling Cap	8.6442
HL Road Frame - Red, 62	HL Road Frame	1263.4598
HL Road Frame - Red, 44	HL Road Frame	1263.4598
HL Road Frame - Red, 48	HL Road Frame	1263.4598
HL Road Frame - Red, 52	HL Road Frame	1263.4598
HL Road Frame - Red, 56	HL Road Frame	1263.4598
HL Mountain Frame - Silver, 42	HL Mountain Frame	1204.3248

Products 30

6. **Example of subquery in HAVING clause:**

Syntax:

```
SELECT column1, column2
FROM table1
GROUP BY column1, column2
HAVING aggregate_condition [NOT] operator (
  subquery
);
```

NOTE:

- *SELECT column1, column2, ..., aggregate_function(column_or_expression): This is the main query that will retrieve the desired columns and aggregated values from a table or tables.*
- *FROM table1: This specifies the table(s) from which the data will be retrieved in the main query.*
- *GROUP BY column1, column2, ...: This clause groups the rows based on the specified columns before the aggregate functions are applied.*
- *HAVING aggregate_condition [NOT] operator: This is the beginning of the HAVING clause, where you specify an aggregate condition (involving an aggregate function like SUM, AVG, COUNT, etc.) to be compared against the result of the subquery. The optional NOT keyword can be used to negate the condition.*
- *operator: This is a comparison operator such as =, >, <, >=, <=, <>, !=, IN, NOT IN.*
- *(subquery): This is the subquery itself, enclosed in parentheses. The subquery can be any valid SQL query that returns a single value or a set of values, depending on the operator used.*

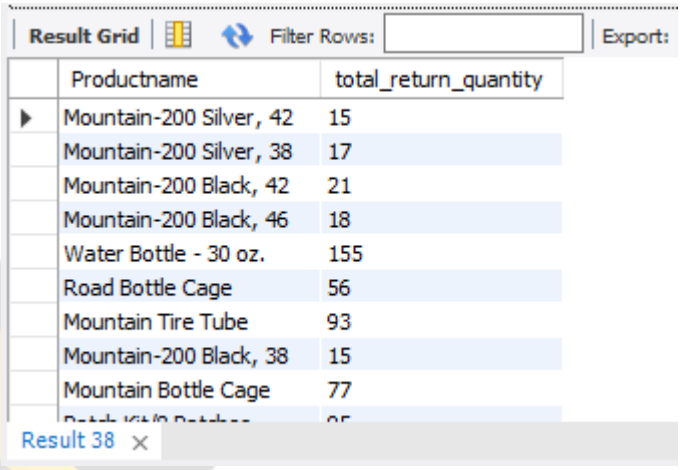
Example 1:

```
SELECT
    p.Productname,
    SUM(r.returnQuantity) AS total_return_quantity
FROM products p
JOIN returns r ON p.ProductKey = r.productkey
GROUP BY p.Productname
HAVING SUM(r.returnQuantity) >
      (SELECT AVG(total_return_quantity)
    FROM
            (SELECT SUM(returnQuantity) AS total_return_quantity
FROM returns GROUP BY productkey) subquery);
```

Note:

This query retrieves the product name and the total return quantity for each product. It does this by joining the products and returns tables on the ProductKey. The results are grouped by Productname, and the HAVING clause filters the results to include only products whose total return quantity is greater than the average return quantity across all products.

Output:



Result Grid | Filter Rows: | Export:

Productname	total_return_quantity
Mountain-200 Silver, 42	15
Mountain-200 Silver, 38	17
Mountain-200 Black, 42	21
Mountain-200 Black, 46	18
Water Bottle - 30 oz.	155
Road Bottle Cage	56
Mountain Tire Tube	93
Mountain-200 Black, 38	15
Mountain Bottle Cage	77

Result 38

7. **Example of subquery in JOINs:**

Syntax:

```
SELECT column1, column2, ...
FROM table1
[INNER | LEFT | RIGHT | FULL] JOIN table2
  ON table1.column = (
    subquery
  )
[WHERE condition];
```

NOTE:

- *SELECT column1, column2, ...: This is the main query that will retrieve the desired columns from the joined tables.*
- *FROM table1: This specifies the left table for the join operation.*
- *[INNER | LEFT | RIGHT | FULL] JOIN table2: This specifies the type of join (INNER, LEFT, RIGHT, or FULL) and the right table for the join operation.*
- *ON table1.column = (subquery): This is the join condition, where a column from the left table (table1) is compared to the result of a subquery. The subquery is enclosed in parentheses ().*
- *subquery: This is the subquery itself, which can be any valid SQL query that returns a set of values to be compared with the column from the left table (table1).*
- *[WHERE condition]: This is an optional WHERE clause to filter the rows after the join operation.*

Example 1:

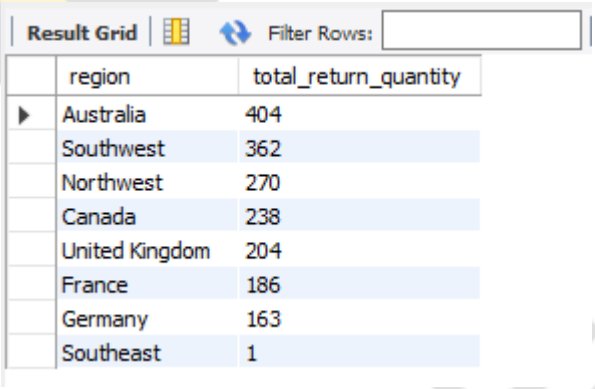
```
SELECT t.region, sub.total_return_quantity
FROM territories t
JOIN (
    SELECT r.territorykey,
        SUM(r.returnQuantity) AS total_return_quantity
    FROM returns r
```

```
    GROUP BY r.territorykey
) sub ON t.salesterritorykey = sub.territorykey
ORDER BY sub.total_return_quantity DESC;
```

NOTE:

This query combines data from the territories table and a subquery. The subquery calculates the total return quantity for each territory by summing the returnQuantity column from the returns table, grouping the results by territorykey. The main query then joins the territories table with the subquery on the salesterritorykey and territorykey columns respectively. Finally, the results are ordered in descending order based on the total_return_quantity.

Output:



region	total_return_quantity
Australia	404
Southwest	362
Northwest	270
Canada	238
United Kingdom	204
France	186
Germany	163
Southeast	1

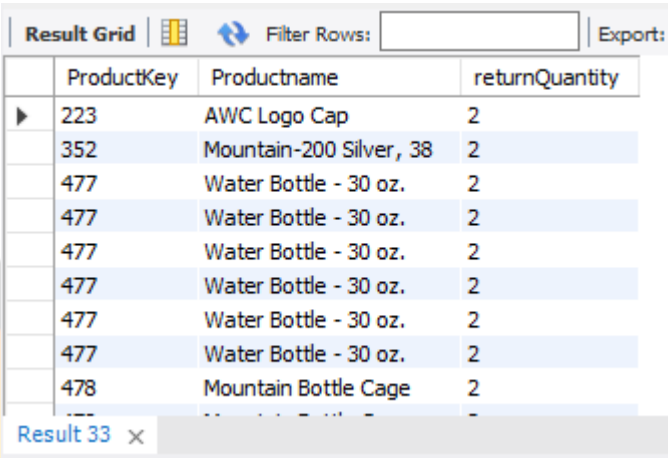
8. **Example of correlated subquery:**

```
SELECT p.ProductKey, p.Productname, r.returnQuantity
FROM products p
JOIN returns r ON p.ProductKey = r.productkey
WHERE r.returnQuantity > (
    SELECT AVG(r2.returnQuantity)
    FROM returns r2
    JOIN products p2 ON r2.productkey = p2.ProductKey
    WHERE p2.productsubcategorykey = p.productsubcategorykey
)
ORDER BY r.returnQuantity DESC;
```

NOTE:

This query retrieves the ProductKey, Productname, and returnQuantity from the products and returns tables. It joins these tables on the ProductKey to match products with their return quantities. The WHERE clause filters the results to include only rows where the returnQuantity is greater than the average returnQuantity for products in the same product subcategory. Finally, the results are sorted in descending order of returnQuantity.

Output:



Result Grid | Filter Rows: | Export:

ProductKey	Productname	returnQuantity
223	AWC Logo Cap	2
352	Mountain-200 Silver, 38	2
477	Water Bottle - 30 oz.	2
477	Water Bottle - 30 oz.	2
477	Water Bottle - 30 oz.	2
477	Water Bottle - 30 oz.	2
477	Water Bottle - 30 oz.	2
477	Water Bottle - 30 oz.	2
478	Mountain Bottle Cage	2

Result 33

9. **Examples of sub-query inside another sub-query:**

Syntax:

```
SELECT column1, column2, ...
FROM table1
WHERE column_expression operator (
  SELECT column1, column2, ...
  FROM table2
  WHERE column_expression operator (
    subquery
  )
);
```

NOTE:

- *SELECT column1, column2, ...: This is the main query that will retrieve the desired columns from a table or tables.*
- *FROM table1: This specifies the table(s) from which the data will be retrieved in the main query.*
- *WHERE column_expression operator: This is the beginning of the WHERE clause, where you specify a column or expression to be compared against the result of the outer subquery.*
- *(SELECT column1, column2, ... FROM table2 WHERE column_expression operator (subquery)): This is the outer subquery, which itself contains an inner subquery. The outer subquery retrieves data from table2 based on a condition involving the inner subquery.*
- *(subquery): This is the innermost subquery, enclosed in parentheses. This subquery can be any valid SQL query that returns a single value or a set of values, depending on the operator used in the outer subquery.*

Example 1:

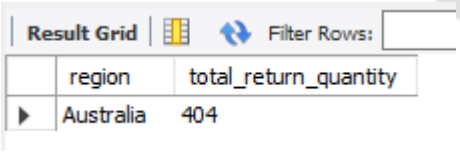
```
SELECT t.region, sub.total_return_quantity
FROM territories t
JOIN (
    SELECT  r.territorykey,
                SUM(r.returnQuantity) AS total_return_quantity
```

```
    FROM returns r
    GROUP BY r.territorykey
) sub ON t.salesterritorykey = sub.territorykey
WHERE sub.total_return_quantity = (
    SELECT MAX(total_return_quantity)
    FROM (
        SELECT SUM(r2.returnQuantity) AS total_return_quantity
        FROM returns r2
        GROUP BY r2.territorykey
    ) sub2
);
```

NOTE:

This query retrieves the region from the territories table along with the total return quantity for each territory. It does this by joining the territories table with a subquery that calculates the total return quantity for each territory. The subquery groups the return quantities by territorykey and calculates the sum. The main query then joins this result with the territories table on the salesterritorykey to get the region. Finally, it filters the results to only include the rows where the total_return_quantity is equal to the maximum total_return_quantity calculated across all territories.

Output:



EXAMPLE 1: (Query optimization using sub-query)

This is one of the examples that we used in session 9 where we explained how full join can be correlated with the union function and we have also mentioned that this is not the optimized query for this question.

Unoptimized query:

```
SELECT c.customerkey,
CONCAT(c.firstname, ' ', c.lastname) AS customer_name,
SUM(s_2015.orderquantity) AS total_sales_quantity
FROM sales_2015 s_2015
JOIN customers c ON s_2015.customerkey = c.customerkey
GROUP BY c.customerkey, customer_name

UNION ALL

SELECT c.customerkey,
CONCAT(c.firstname, ' ', c.lastname) AS customer_name,
```

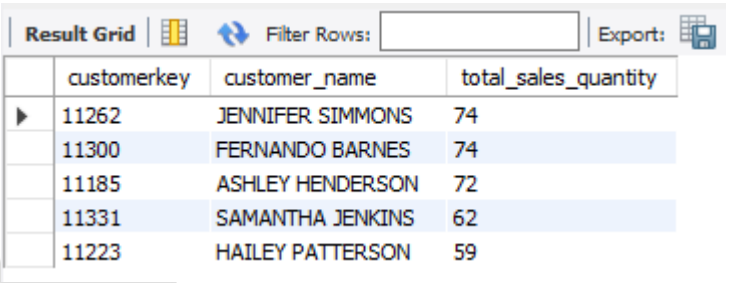
```
SUM(s_2016.orderquantity) AS total_sales_quantity
FROM sales_2016 s_2016
JOIN customers c ON s_2016.customerkey = c.customerkey
GROUP BY c.customerkey, customer_name

UNION ALL

SELECT c.customerkey,
CONCAT(c.firstname, ' ', c.lastname) AS customer_name,
SUM(s_2017.orderquantity) AS total_sales_quantity
FROM sales_2017 s_2017
JOIN customers c ON s_2017.customerkey = c.customerkey
GROUP BY c.customerkey, customer_name
ORDER BY total_sales_quantity DESC
LIMIT 5;
```

Here is how we can optimize the above query using sub-query and union all conditions in SQL. This will help us remove all the repetitive functions we have used in the above query.

Optimized Query:

```
SELECT c.customerkey,
       CONCAT(c.firstname, ' ', c.lastname) AS customer_name,
       total_sales_quantity
FROM (
    SELECT customerkey, SUM(orderquantity) as total_sales_quantity
    FROM sales_2015
    GROUP BY customerkey
    UNION ALL
    SELECT customerkey, SUM(orderquantity) as total_sales_quantity
    FROM sales_2016
    GROUP BY customerkey
    UNION ALL
    SELECT customerkey, SUM(orderquantity) as total_sales_quantity
    FROM sales_2017
    GROUP BY customerkey
) AS s
JOIN customers c ON s.customerkey = c.customerkey
ORDER BY total_sales_quantity DESC
LIMIT 5;
```

Output:



Result Grid | Filter Rows: | Export:

customerkey	customer_name	total_sales_quantity
11262	JENNIFER SIMMONS	74
11300	FERNANDO BARNES	74
11185	ASHLEY HENDERSON	72
11331	SAMANTHA JENKINS	62
11223	HAILEY PATTERSON	59

NOTE: The outputs for both queries will be the same as here we haven't changed the logic of the query, rather we have just optimized the query.