SQL - Nested Queries

A Nested query or a Subquery or an Inner query is a query within another SQL query and embedded within the WHERE, FROM, SELECT, HAVING clauses. Both queries may be run as separate queries.

You can include a subquery:

In the WHERE clause, to filter data.

In the FROM clause, to specify a new table.

In the SELECT clause, to specify a certain column.

In the HAVING clause, as a group selector.

Subqueries can be used with the SELECT, INSERT, UPDATE, and DELETE statements along with the operators like =, <, >, >=, <=, IN, BETWEEN, etc.

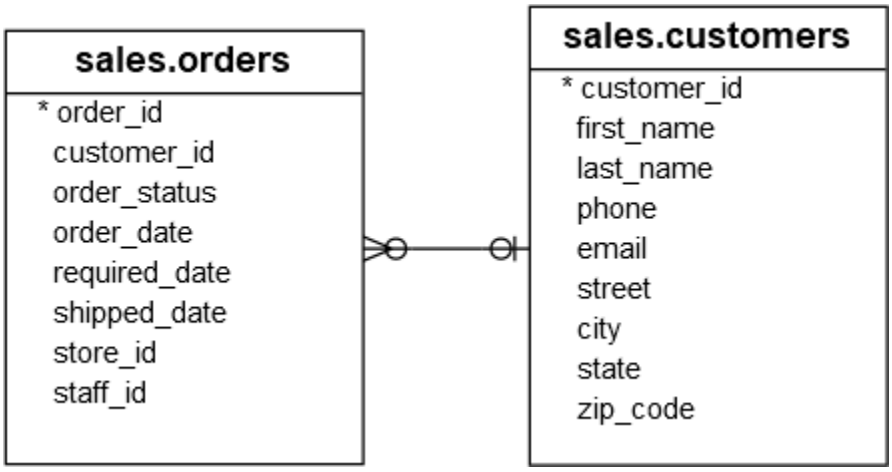
There are a few rules that subqueries must follow:

- Subqueries must be enclosed within parentheses.
- A subquery can have only one column in the SELECT clause, unless multiple columns are in the main query for the subquery to compare its selected columns.
- An ORDER BY command cannot be used in a subquery, although the main query can use an ORDER BY. The GROUP BY command can be used to perform the same function as the ORDER BY in a subquery.
- Subqueries that return more than one row can only be used with multiple value operators such as the IN operator.
- The SELECT list cannot include any references to values that evaluate to a BLOB, ARRAY, CLOB, or NCLOB.
- A subquery cannot be immediately enclosed in a set function.
- The BETWEEN operator cannot be used with a subquery. However, the BETWEEN operator can be used within the subquery.

Subqueries with the SELECT Statement

Subqueries are most frequently used with the SELECT statement.

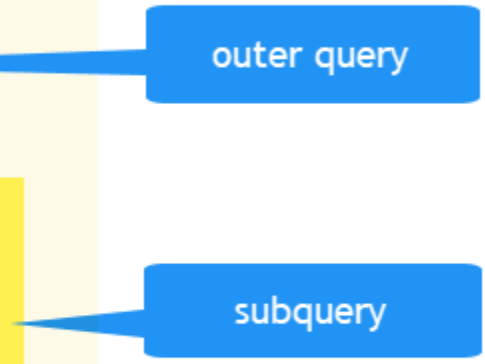
```
SELECT column_name [, column_name ]
FROM   table1 [, table2 ]
WHERE  column_name OPERATOR
   (SELECT column_name [, column_name ]
   FROM table1 [, table2 ]
   [WHERE])
```




```
SELECT
    order_id,
    order_date,
    customer_id
FROM
    sales.orders
WHERE
    customer_id IN (
```

```
        SELECT
            customer_id
        FROM
            sales.customers
        WHERE
            city = 'New York'
    )
ORDER BY
    order_date DESC;
```

```sql
SELECT
    order_id,
    order_date,
    customer_id
FROM
    sales.orders
WHERE
    customer_id IN (
        SELECT
            customer_id
        FROM
            sales.customers
        WHERE
            city = 'New York'
    )
ORDER BY
    order_date DESC;
```



A subquery can be nested within another subquery. SQL Server supports up to 32 levels of nesting.

```
SELECT product_name, list_price
FROM production.products
WHERE
    list_price > (
        SELECT AVG(list_price)
        FROM production.products
        WHERE
            brand_id IN (
                SELECT brand_id
                FROM production.brands
                WHERE
                    brand_name = 'Strider' OR brand_name = 'Trek'
            )
    )
ORDER BY list_price;
```

SQL Server Subquery Types

You can use a subquery in many places:

- In place of an expression
- With IN or NOT IN
- With ANY or ALL
- With EXISTS or NOT EXISTS
- In UPDATE, DELETE, or INSERT statement
- In the FROM clause

SQL Server subquery is used in place of an expression

If a subquery returns a single value, it can be used anywhere an expression is used.

```
SELECT
    order_id,
    order_date,
    (
        SELECT MAX(list_price)
        FROM sales.order_items i
        WHERE i.order_id = o.order_id
    ) AS max_list_price
FROM sales.orders o
order by order_date desc;
```

SQL Server subquery is used with IN operator

A subquery that is used with the IN operator returns a set of zero or more values. After the subquery returns values, the outer query makes use of them.

```
SELECT product_id, product_name
FROM production.products
WHERE
    category_id IN (
        SELECT category_id
        FROM production.categories
        WHERE category_name = 'Mountain Bikes' OR category_name = 'Road Bikes'
    );
```

SQL Server subquery is used with ANY operator

The subquery is introduced with the ANY operator has the following syntax:

scalar_expression comparison_operator ANY (subquery)

Assuming that the subquery returns a list of value v1, v2, … vn. The ANY operator returns TRUE if one of a comparison pair (scalar_expression, vi) evaluates to TRUE; otherwise, it returns FALSE.

```
SELECT product_name, list_price
FROM  production.products
WHERE
    list_price >= ANY (
        SELECT AVG(list_price) FROM production.products GROUP BY brand_id
    )
```

SQL Server subquery is used with ALL operator

The ALL operator has the same syntax as the ANY operator:

scalar_expression comparison_operator ALL (subquery)

The ALL operator returns TRUE if all comparison pairs (scalar_expression, vi) evaluate to TRUE; otherwise, it returns FALSE.

```
SELECT product_name, list_price
FROM production.products
WHERE
   list_price >= ALL (
      SELECT AVG(list_price)
      FROM production.products
      GROUP BY brand_id
   )
```

SQL Server subquery is used with EXISTS or NOT EXISTS

The following illustrates the syntax of a subquery introduced with EXISTS operator:

WHERE [NOT] EXISTS (subquery)

The EXISTS operator returns TRUE if the subquery return results; otherwise it returns FALSE.

On the other hand, the NOT EXISTS is opposite to the EXISTS operator.

```
SELECT customer_id, first_name, last_name, city
FROM sales.customers c
```

```
WHERE
    EXISTS (
        SELECT customer_id
        FROM sales.orders o
        WHERE o.customer_id = c.customer_id AND YEAR (order_date) = 2017
    )
ORDER BY first_name, last_name;
```

If you use the NOT EXISTS instead of EXISTS, you can find the customers who did not buy any products in 2017.

SQL Server subquery in the FROM clause

Suppose that you want to find the average of the sum of orders of all sales staff. To do this, you can first find the number of orders by staffs:

```
SELECT staff_id, COUNT(order_id) order_count
FROM sales.orders
GROUP BY staff_id;
```

Then, you can apply the AVG() function to this result set. Since a query returns a result set that looks like a virtual table, you can place the whole query in the FROM clause of another query like this:

```
SELECT AVG(order_count) average_order_count_by_staff
FROM
(
    SELECT staff_id, COUNT(order_id) order_count
    FROM sales.orders
    GROUP BY staff_id
```

) t;

The query that you place in the FROM clause must have a table alias. In this example, we used the t as the table alias for the subquery.

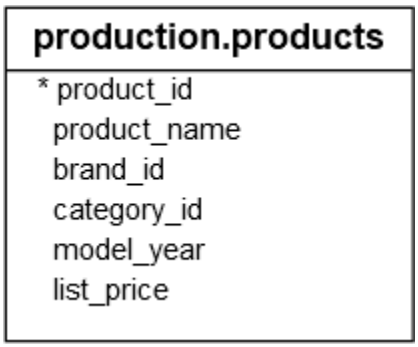
Correlated Subqueries

Subqueries are correlated when the inner and outer queries are interdependent.

A correlated subquery is a subquery that uses the values of the outer query. In other words, it depends on the outer query for its values. Because of this dependency, a correlated subquery cannot be executed independently as a simple subquery.

Moreover, a correlated subquery is executed repeatedly, once for each row evaluated by the outer query. The correlated subquery is also known as a repeating subquery.

Consider the following products table from the sample database:



The following example finds the products whose list price is equal to the highest list price of the products within the same category:

```
SELECT product_name, list_price, category_id
FROM production.products p1
WHERE
```

```
    list_price IN (
        SELECT MAX (p2.list_price)
        FROM production.products p2
        WHERE p2.category_id = p1.category_id
        GROUP BY p2.category_id
    )
ORDER BY category_id, product_name;
```

In this example, for each product evaluated by the outer query, the subquery finds the highest price of all products in its category. If the price of the current product is equal to the highest price of all products in its category, the product is included in the result set. This process continues for the next product and so on.

As you can see, the correlated subquery is executed once for each product evaluated by the outer query.

SQL Server EXISTS operator overview

The EXISTS operator is a logical operator that allows you to check whether a subquery returns any row. The EXISTS operator returns TRUE if the subquery returns one or more rows.

The following shows the syntax of the SQL Server EXISTS operator:

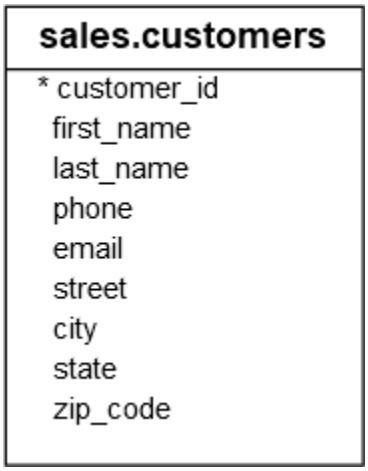
```
EXISTS ( subquery)
```

In this syntax, the subquery is a SELECT statement only. As soon as the subquery returns rows, the EXISTS operator returns TRUE and stop processing immediately.

Note that even though the subquery returns a NULL value, the EXISTS operator is still evaluated to TRUE.

A) Using EXISTS with a subquery returns NULL example

See the following customers table from the sample database.



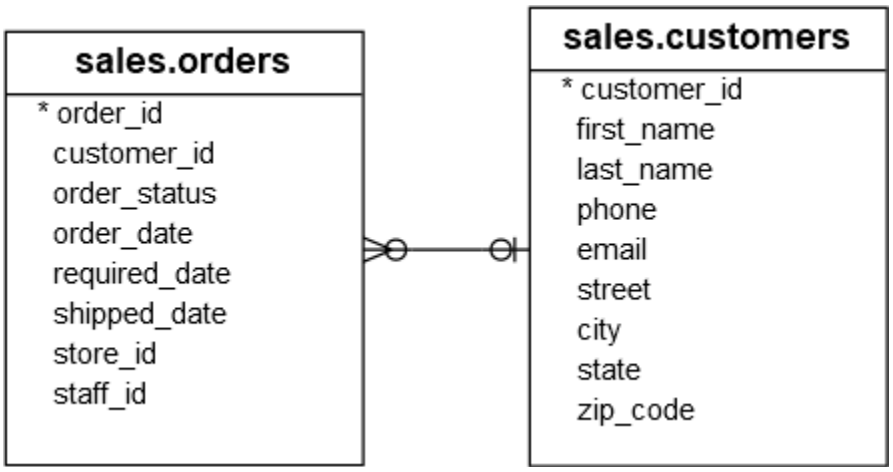
The following example returns all rows from the customers table:

```
SELECT customer_id, first_name, last_name
FROM sales.customers
WHERE
    EXISTS (SELECT NULL)
ORDER BY first_name, last_name;
```

In this example, the subquery returned a result set that contains NULL which causes the EXISTS operator to evaluate to TRUE. Therefore, the whole query returns all rows from the customers table.

B) Using EXISTS with a correlated subquery example

Consider the following customers and orders tables:



The following example finds all customers who have placed more than two orders:

```
SELECT customer_id, first_name, last_name
FROM sales.customers c
WHERE
   EXISTS (
      SELECT COUNT (*)
      FROM sales.orders o
      WHERE customer_id = c.customer_id
      GROUP BY customer_id
      HAVING COUNT (*) > 2
   )
ORDER BY first_name, last_name;
```

In this example, we had a correlated subquery that returns customers who place more than two orders.

CTE in SQL Server

CTE stands for common table expression. A CTE allows you to define a temporary named result set that available temporarily in the execution scope of a statement such as SELECT, INSERT, UPDATE, DELETE, or MERGE.

The following shows the common syntax of a CTE in SQL Server:

```
WITH expression_name[(column_name [,...])]
AS
    (CTE_definition)
SQL_statement;
```

We prefer to use common table expressions rather than to use subqueries because common table expressions are more readable. We also use CTE in the queries that contain analytic functions (or window functions)

A) Simple SQL Server CTE example

```
WITH cte_sales_amounts (staff, sales, year) AS (
    SELECT first_name + ' ' + last_name, SUM(quantity * list_price * (1 - discount)), YEAR(order_date)
    FROM sales.orders o
    INNER JOIN sales.order_items i ON i.order_id = o.order_id
    INNER JOIN sales.staffs s ON s.staff_id = o.staff_id
    GROUP BY first_name + ' ' + last_name, year(order_date)
)
SELECT staff, sales
FROM cte_sales_amounts
WHERE year = 2018;
```

B) Using a common table expression to make report averages based on counts

This example uses the CTE to return the average number of sales orders in 2018 for all sales staffs.

```
WITH cte_sales AS (
    SELECT staff_id, COUNT(*) order_count
    FROM sales.orders
    WHERE YEAR(order_date) = 2018
    GROUP BY staff_id
)
SELECT AVG(order_count) average_orders_by_staff
FROM cte_sales;
```