

Data Analytics

Subqueries in SQL

Our Learning Goals

- Construct subqueries for multi-step operations.
- Identify subquery use cases for various business scenarios.



Quick Review

SELECT *picks the columns (including using **CASE**)*

FROM *points to the table or JOINed tables*

WHERE *puts filters on the rows*

GROUP BY *aggregates across values of a variable*

HAVING *filters aggregated values **after** grouping*

ORDER BY *sorts the results*

LIMIT n *limits results to the first **n** rows*



Discussion:

Cost of Returns

The VP of Operations at Superstore needs to determine the operational cost of shipping wrong items to customers. To help her make this critical decision, you've been asked to find out the following:

What is the number of orders (with wrong items) that have been returned for each shipping mode?

Writing queries is all about asking the right questions.

What are two questions we need to answer in order to write this query?

Subqueries in SQL

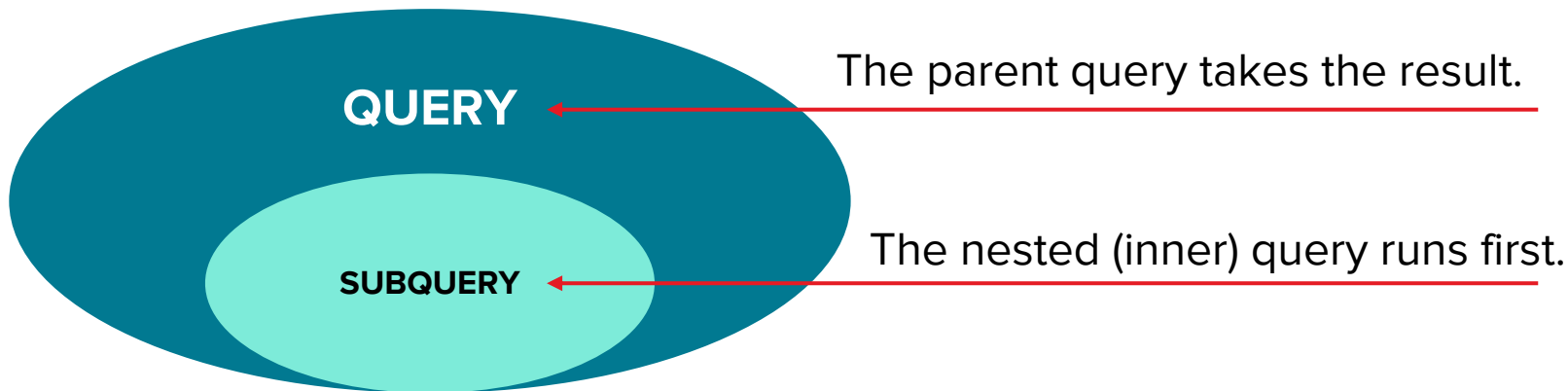
Introduction to Subqueries



Meet Subqueries

Also known as a **nested query**, a subquery is an SQL statement that combines multiple questions into one query.

How this works: When you run one query and get a result, you can feed it to another query.



Pseudo-Coding Subqueries

Referring back to the request from the VP of Operations, we need to:

- Identify the order numbers that have been returned as “wrong item.”
- Count the number of returned orders by shipping mode.

Table: Returns

	order_id	return_date	return_quantity	reason_returned
0	AG-2016-3024683	2016-12-14	1	Wrong Item
1	AG-2016-4817505	2016-12-06	1	Wrong Item
2	AG-2017-1275260	2017-06-04	1	Wrong Item
3	AG-2017-1506124	2017-09-10	1	Wrong Item
4	AG-2017-1508402	2017-08-19	1	Wrong Item

Table: Orders (select columns)

	order_id	ship_mode	sales
0	AE-2016-1308551	Second Class	82.67
1	AE-2016-1522857	Standard Class	78.41
2	AE-2016-184765	Second Class	82.67
3	AE-2016-1878215	Standard Class	78.41
4	AE-2016-218276	Standard Class	78.41





Guided Walk-Through: Subquery Syntax

```
SELECT ship_mode, COUNT( * )  
FROM Orders  
WHERE order_id IN (  
  
    SELECT order_id  
    FROM returns  
    WHERE reason_returned =  
           "Wrong Item"  
)  
GROUP BY ship_mode;
```

A breakdown of the subquery (in red):

- **SELECT** all of the order_ids.
- FROM the Returns table, because we want to identify orders that match our criteria for return reason.
- Filter the results for only order_ids with **reason_returned** equal to "Wrong Item."



Guided Walk-Through: Subquery Syntax

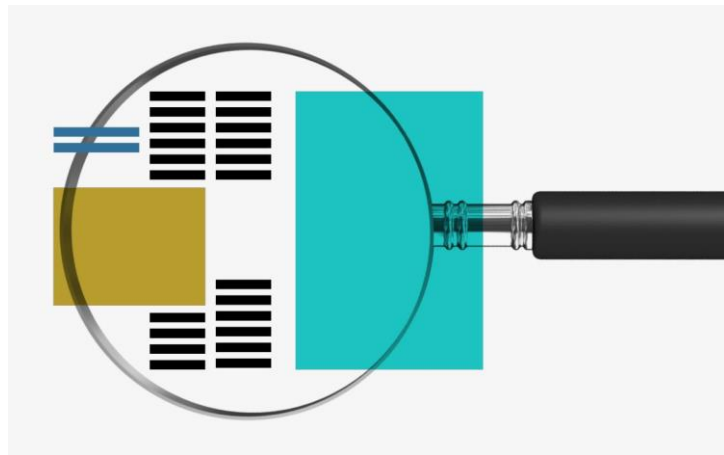
```
SELECT ship_mode, COUNT( * )  
FROM Orders  
WHERE order_id IN (  
  
    SELECT order_id  
    FROM returns  
    WHERE reason_returned =  
        "Wrong Item"  
    )  
GROUP BY ship_mode;
```

A breakdown of the outer query (in red) that reads the results from the inner query:

- **SELECT** takes ship_mode and a count of all rows.
- **FROM** tells us where the information is coming from.
- **WHERE** filters the selected rows from the Orders table using the results of our subquery.
- **GROUP BY** ship_mode because of our aggregation function COUNT.

Subquery Syntax | A Checklist

- ❑ Subqueries are enclosed in parentheses and will execute first.
- ❑ In order to execute, subquery structures **must** have complete query components: **SELECT**, **FROM**, and a specified criteria.
- ❑ Assigning aliases is important for syntax and readability.



Why Do You Need Subqueries...

...when you already have JOINS?

Subqueries and JOINS are both used to combine data from different tables.

Subqueries:

- Can be used to return either a single value or a row set.
- Can run on their own (as queries).
- Results are used immediately.
- Used in SELECT, WHERE, FROM, and HAVING.

JOINS:

- Used to return rows.
- Cannot run on their own.
- The “JOINED” results are available in SELECT statements.
- Used as FROM clauses of WHERE statements.

Subqueries in SQL

Subquery Types



Types of Subqueries

The three types of subqueries differ by how the results are used by the outer query: FROM, IN, or comparison.

Type 1: FROM statement uses the results of one query as a table for the *FROM statement* of the outer query.

Type 2: IN operator filters the results of one query by the results of another query using the *IN operator*.

Type 3: Comparison operator filters the results of one query by the results of another query using a *comparison operator*.

Type 1 Subquery Syntax

- Also known as “a query *within* a query,” it allows us to supply an entire query between parentheses as a subquery instead of calling a normal table.
- The outer query then treats result of the inner query as if it were a table.

```
SELECT a_column  
FROM (SELECT another_column  
      FROM a_different_table) AS alias;
```



Type 1: A Table Within a Query

The Head of Sales asked for insights on whether we've seen an increase in **average monthly sales** over time.

	order_date timestamp without time zone	order_id text	sales numeric
1	2015-12-18 00:00:00	CA-2015-366239	31.12
2	2015-12-21 00:00:00	ES-2015-992348	108.24
3	2015-12-21 00:00:00	MX-2015-1384450	38.40
4	2015-12-22 00:00:00	CA-2015-5270762	572.58
5	2015-12-26 00:00:00	CA-2015-1688572	10.43
6	2015-12-26 00:00:00	IN-2015-2451468	59.25
7	2015-12-27 00:00:00	NI-2015-220400	3.09

We need to
calculate the total
sales in each
month



Month	Total Sales
2015, Dec	2214.06
2016, Jan	109175.47
2016, Feb	143608.35
...	...



Month	Sales
2015	2214.06
2016	798018.39
2017	2535652.52
...	...

.. then obtain the average
monthly sales for each year



Type 1: A Table Within a Query

The Head of Sales asked for insights on whether we've seen an increase in average monthly sales over time. First, let's write a subquery to *calculate* the sales for every month:

```
SELECT DATE_TRUNC('month',order_date) as order_month,  
SUM(sales) AS monthly_sales  
FROM orders  
GROUP BY 1;
```




Type 1: A Table Within a Query (Cont.)

Then, average the results in the outer query:

```
SELECT DATE_PART('year',order_month) AS year, AVG(monthly_sales) AS  
avg_monthly_sales  
FROM (SELECT DATE_TRUNC('month',order_date) as order_month, SUM(sales)  
AS monthly_sales  
FROM orders  
GROUP BY 1) AS temp  
GROUP BY 1  
ORDER BY 1 DESC;
```



Solo Exercise:

Categorizing Customers

5 minutes



Most customers make several hundred purchases from us as a business supplier. Let's look at how much of our revenue these customers make up.

Inner query: Create a query that categorizes customers by purchase frequency and returns total sales per customer:

- 1000+ = Supplier
- 100+ = Multiple
- All others

Outer query: How much in sales did each segment make?



Solo Exercise:

Categorizing Customers | Solution

```
SELECT customer_type, SUM(total_sales) AS total_sales
FROM
    (SELECT customer_id,
        CASE WHEN COUNT(DISTINCT order_id)>=1000 THEN 'Supplier'
        WHEN COUNT(DISTINCT order_id)>=100 THEN 'Frequent'
        ELSE 'All others' END as customer_type,
        SUM(sales) as total_sales
    FROM orders
    GROUP BY 1) AS temp
GROUP BY 1
ORDER BY 2 DESC;
```

Type 2 Subquery Syntax

- The *inner query* retrieves a list of items in the form of **a single column**.
- The *outer query* then tests one of its columns against this list of items.

```
SELECT a_column
FROM a_table
WHERE column_id IN
    (SELECT column_id
     FROM a_different_table
     WHERE a_different_column meets some_condition);
```



Type 2: IN Operator

Looking at orders in more detail, the VP of Product wants you to focus on the consumer segments and find out:

How many orders had products with a cost to consumer of more than \$500?

First, write a query that first selects the appropriate products:

```
SELECT product_id  
      FROM products  
WHERE CAST(product_cost_to_consumer AS int) > 500;
```

Then, insert it into the outer query's WHERE clause (see next slide).



Type 2: IN Operator (Cont.)

```
SELECT COUNT(*)  
FROM orders  
WHERE product_id IN (  
    SELECT product_id  
    FROM products  
    WHERE CAST(product_cost_to_consumer AS int) > 500);
```

Here, the inner query must *only* return **one column** for an **IN** filter. Consider that the inner query is *building an array of values* that are then substituted into the **WHERE** clause in the outer loop.



Solo Exercise:

How About **NOT IN**?

5 minutes



Conversely, you can use **NOT IN** in the subquery to exclude data.

Going back to the previous query, we can find the number of **orders that had products with a cost to consumer of more than \$500** in either of two ways with the options below:

Give Option 2 a try, and we'll discuss as a class.

Option 1: Use the **IN** statement and the logical operator **greater than**.

Option 2: Use the **NOT IN** statement and the logical operator **less than**.





Query 1 Use IN

```
SELECT COUNT(*)  
FROM orders  
WHERE product_id IN (  
  SELECT product_id  
  FROM products  
  WHERE  
  CAST(product_cost_to_consumer  
  AS int) > 500)
```

Query 2 Use NOT IN

```
SELECT COUNT(*)  
FROM orders  
WHERE product_id  
NOT IN (  
  SELECT product_id  
  FROM products  
  WHERE  
  CAST(product_cost_to_consumer  
  AS int) < 500)
```


Type 3 Subquery Syntax

- The *inner query* retrieves **the average of another column** from a different table.
- The *outer query* then takes the inner query and uses it with a comparison operator to filter a list of values.

```
SELECT *  
FROM a_table  
WHERE a_column <  
      (SELECT AVG(another_column)  
       FROM a_different_table);
```



Type 3: Comparison Operator

Now that you have a gauge on the number of orders for products that had a cost of more than \$500 to the consumer, let's shift the focus to profit:

How many orders have more profit than the average product cost to consumer?

Use a subquery to *first* calculate the most expensive item to consumers:

```
SELECT AVG(CAST(product_cost_to_consumer AS int))  
FROM products;
```



Type 3: Comparison Operator (Cont.)

Then, apply the subquery in the outer query:

```
SELECT COUNT(*)  
FROM orders  
WHERE CAST(profit AS int) >  
      (SELECT AVG(CAST(product_cost_to_consumer AS int))  
       FROM products);
```

CAST() is used in case the profit and product_cost_to_consumer was stored as text



Solo Exercise:

Use SELECT to Find %

5 minutes



Let's take our query from earlier and find the share of sales that each frequency of customer makes up. Add a SELECT subquery in our outer query to find the total sum of all sales.

```
SELECT customer_type, SUM(total_sales) AS total_sales
FROM
    (SELECT customer_id,
     CASE WHEN COUNT(DISTINCT order_id)>=1000 THEN 'Supplier'
     WHEN COUNT(DISTINCT order_id)>=100 THEN 'Frequent'
     ELSE 'All others' END as customer_type,
          SUM(sales) as total_sales
     FROM orders
     GROUP BY 1) AS temp
GROUP BY 1
ORDER BY 2 DESC;
```





Solo Exercise:

Use SELECT to Find % | Solution

```
SELECT customer_type, SUM(total_sales) AS total_sales,  
       SUM(total_sales)::numeric/(SELECT SUM(sales) FROM orders)::numeric  
       AS share_of_sales  
FROM  
    (SELECT customer_id,  
     CASE WHEN COUNT(DISTINCT order_id)>=1000 THEN 'Supplier'  
     WHEN COUNT(DISTINCT order_id)>=100 THEN 'Frequent'  
     ELSE 'All others' END as customer_type,  
          SUM(sales) as total_sales  
     FROM orders  
     GROUP BY 1) AS temp  
GROUP BY 1  
ORDER BY 2 DESC;
```



Discussion:

Key Differences of Subquery Types

Now that we've seen all three types of subqueries, let's review their uses by focusing on the key differences among the three:

- **Type 1** subqueries can provide a whole table.
- **Type 2** IN subqueries must provide a single column.
- **Type 3** comparison subqueries must provide a single value.

Any other differences that come to mind?

Subqueries in SQL

Additional Subquery Use Cases



Running Multiple Queries at Once

Using the subqueries methodology, you can run multiple SELECT statements at once and get the results on the same screen.

The benefit: If you need fields from multiple data sources or from multiple parts of a single data set (e.g., for comparison).

What this looks like in practice:

Instead of separately running two SELECT statements, we can **wrap each SELECT statement in its own parentheses** to be evaluated separately in a common results pane.



Discussion:

Running Multiple Queries

The queries will be run at once on the same screen. Knowing that each SELECT statement is wrapped in its own parentheses, what else do you notice about these queries? And what can you anticipate seeing in the results? Share your answer with the class.

```
(SELECT COUNT(*)  
FROM orders  
WHERE product_id IN (  
SELECT product_id FROM products  
WHERE  
    CAST(product_cost_to_consumer  
    AS int) > 500) )  
AS query_1_result,
```

```
(SELECT COUNT(*)  
FROM orders  
WHERE product_id NOT IN (  
SELECT product_id FROM products  
WHERE  
    CAST(product_cost_to_consumer  
    AS int) < 500) )  
AS query_2_result
```



Running Multiple Queries

```
SELECT * FROM
```

```
(SELECT COUNT(*)
```

```
FROM orders
```

```
WHERE product_id IN (
```

```
SELECT product_id FROM products
```

```
WHERE
```

```
CAST(product_cost_to_consumer
```

```
AS int) > 500)) AS query_1_result,
```

```
(SELECT COUNT(*)
```

```
FROM orders
```

```
WHERE product_id NOT IN (
```

```
SELECT product_id FROM products
```

```
WHERE
```

```
CAST(product_cost_to_consumer
```

```
AS int) < 500)) AS query_2_result
```

On your screen, the queries will be stacked like this, and...

1. We have to start the query with a **SELECT** statement.
2. Each query is wrapped in its own parentheses.
3. Each subquery is separated by a single comma.
4. **Alias** each query to keep track of the results.

Using CASE Statements in Subqueries

CASE statements can be used in *both* the inner query and outer queries.

What makes it cool? One useful and common construction is to:

- Use the inner (nested) query to create a binary value, sometimes called a “flag.”
- Then, use the outer query to average that value.

Let's build an example using a **CASE** classification and a binary value!



CASE in Subqueries

Let's say we classified profit per order as small, medium, large. Now we want to use those categories to count how many orders fit into each group. Let's look at the first classification query:

```
SELECT order_id, profit,  
       CASE  
         WHEN CAST(profit AS int) >= 1000 THEN 'large'  
         WHEN CAST(profit AS int) >= 50 THEN 'medium'  
         ELSE 'small'  
       END AS profit_size  
FROM orders
```



CASE in Subqueries (Cont.)

Here is the first query (in red) serving as our inner query, with the outer query counting occurrences:

```
SELECT profit_size, COUNT(*) AS number_orders
FROM (SELECT order_id, profit,
CASE
    WHEN CAST(profit AS int) >= 1000 THEN 'large'
    WHEN CAST(profit AS int) >= 50 THEN 'medium'
    ELSE 'small'
END AS profit_size
FROM orders) AS temp
GROUP BY profit_size;
```



CASE in Subqueries With a Binary Value

Imagine you want to get even more granular and find out, ***by product_id***, what **percentage of discounts were more than 25%**. First, create the inner query that classifies each discount as either more or less than 25%:

```
SELECT ship_mode, profit,  
       CASE  
         WHEN CAST(discount AS float) > 0.25 THEN 1  
         ELSE 0  
       END AS over_25  
FROM orders;
```



CASE in Subqueries With a Binary Value (Cont.)

Now, wrap query in an outer query that averages the *flag (binary value)* per store:

```
SELECT product_id, ROUND(AVG(over_25), 4) AS pct_over_25
FROM (SELECT product_id, profit,
             CASE WHEN CAST(discount AS float) > 0.25 THEN 1
                  ELSE 0
             END AS over_25
      FROM orders) AS temp
GROUP BY product_id
ORDER BY product_id
LIMIT 100;
```

Subqueries in SQL

Wrapping Up



Recap

Today, we...

- Practiced writing subqueries for multi-step operations.
- Compared types of subqueries and use cases for various business scenarios.

Looking Ahead

Homework:

- Homework 5: Subqueries in SQL

Next:

- Data Cleaning and Formulas in Excel



Additional Resources

- “Using Subqueries in the Select Statement” by essentialSQL.com:
<https://goo.gl/GsfwAb>
- “Writing Subqueries in SQL” by Mode Analytics:
<https://community.modeanalytics.com/sql/tutorial/sql-subqueries/>
- Beginner SQL Tutorial: <http://beginner-sql-tutorial.com/sql-subquery.htm>

Before You Go...

- Check to make sure you have your **Tableau license key**.
- Follow the instructions using our [Tableau Installation Guide](#).
- [Install Tableau on your laptop](#) and ensure that it works correctly.
- If you have any problems with your installation, let us know!

