

# How to Write a SQL Project Report for Your Portfolio

## 1. Report Header

Start with clear identifying information.

- **Project Title:** Activity: Final Exam: AdventureWorks Data Analysis
- **Your Name:** Natascha Martin
- **Date Completed:** June 10, 2025
- **Course/Module:** Introduction to SQL (Postgres SQL in PgAdmin)

## 2. Project Overview / Introduction

This final focused on my ability to apply what I have learned over the last three weeks with another database and apply what I learned to a new database.

- **What was the lab about?** Answering test questions to test my knowledge on the last three weeks of coursework. I chose to write a report even though it was not required.
- **What was the scenario?**

1<sup>st</sup> scenario In this scenario, the AdventureWorks management team has asked you to do some basic data analysis and manipulation of our customer table.

2<sup>nd</sup> scenario In this scenario, you are tasked with analyzing customer addresses. This requires you to use table joins across three tables: customer, customeraddress, and address.

3<sup>rd</sup> Scenario For this next scenario, we're going to start taking a look at the sales data in our AdventureWorks database.

Notice how the sales order data is broken down into two tables: salesorderheader and salesorderdetail.

Salesorderheader is a header table that contains aggregate information about the sales order itself, such as the customer, the shipping address, the billing address, total sales amount and taxes, order date, and so forth.

Salesorderdetail is a detail table that contains line-item details about each sales order. Thus, there is a parent-child relationship that exists from salesorderheader to salesorderdetail. The salesorderdetail table contains links to each product, as well as the price and quantity.

- **What SQL concepts were emphasized?** (e.g., "It specifically explored the use of AND, OR, and NOT operators, alongside LIKE, to retrieve specific records.")

### 3. Tools Used

List the key technologies and environments you worked with.

- **Database System:** Postgres SQL
- **Interface:** Command-line client
- **Platform:** PgAdmin

### 4. Task-by-Task Analysis (The Core of Your Report)

#### Task 1. Objective:

The objective was to identify the first name of a specific customer by filtering the `customer` table based on their `CompanyName`.

#### SQL Query:

```
SELECT FirstName  
FROM customer  
WHERE CompanyName = 'Alpine Ski House';
```

#### Explanation of Query Logic:

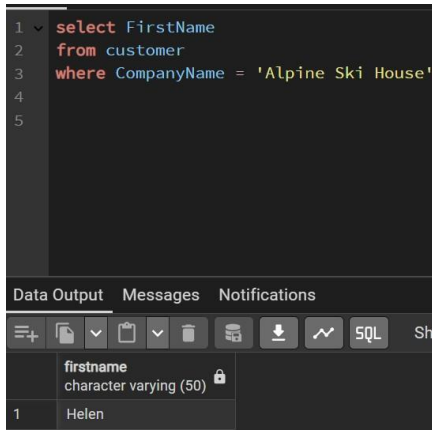
- **SELECT FirstName:** This clause was used to specify that only the `FirstName` column should be retrieved from the dataset. While `SELECT *` could have been used to view all columns, explicitly selecting `FirstName` makes the query more efficient and focuses the output on the required information.
- **FROM customer:** This clause indicated that the data should be sourced from the `customer` table. Initial attempts revealed that the table name was singular (`customer`) rather than plural (`customers`), which is crucial for correct database interaction in PostgreSQL.
- **WHERE CompanyName = 'Alpine Ski House':** This clause acted as a filter, restricting the returned rows to only those where the value in the `CompanyName` column exactly matched the string "Alpine Ski House". The `=` operator was used for this direct comparison.

#### Result/Output

The query successfully returned "Helen" as the first name of the customer.

## Challenges Encountered

Initially, I encountered syntax errors due to incorrect placement of commas and missing comparison operators in the `WHERE` clause. I also faced a `relation 'customers' does not exist` error, which helped me identify that the correct table name was `customer` (singular), emphasizing the importance of precise object naming in SQL. Overcoming these issues reinforced attention to SQL syntax and schema details.



The screenshot shows a SQL IDE with a dark theme. The query editor contains the following SQL code:

```
1 select FirstName
2 from customer
3 where CompanyName = 'Alpine Ski House'
4
5
```

Below the query editor, there is a toolbar with icons for Data Output, Messages, and Notifications. The Data Output tab is active, showing a table with the following data:

1	firstname
	character varying (50)
1	Helen

## Task 2. Objective:

The objective was to find out how many AdventureWorks `customers` go by the title of “Mr.”?

## SQL Query:

```
SELECT COUNT (*)
FROM customer
WHERE title = 'Mr.';
```

## Explanation of Query Logic:

`SELECT COUNT (*)`: This clause was used to count the total number of rows that matched the specified criteria. The `COUNT (*)` aggregate function is ideal for getting a numerical tally of records.

`FROM customer`: This clause designated the `customer` table as the source of the data for the query.

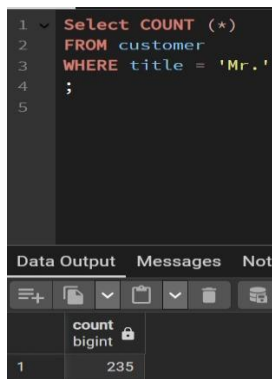
`WHERE title = 'Mr.'`: This clause served as the filter, ensuring that only records where the `Title` column contained the exact string `'Mr.'` were included in the count. The `=` operator was used for this precise string comparison.

## Result/Output

The query successfully returned a count of 235, indicating that there are 235 customers with the title “Mr.”.

### Challenges Encountered

Initially, I included a `LIMIT 10` clause, which restricted the count to only the first 10 matching rows, leading to an incorrect total. Removing the `LIMIT` clause was essential to ensure `COUNT (*)` processed all relevant rows in the table. I also considered using `LIKE 'Mr.%'` but confirmed that an exact match with `=` was appropriate for the specific title `'Mr.'`.



```
1 Select COUNT (*)
2 FROM customer
3 WHERE title = 'Mr.'
4 ;
5
```

The screenshot shows a SQL query editor with a dark theme. The query is: `Select COUNT (*) FROM customer WHERE title = 'Mr.' ;`. Below the editor, there is a 'Data Output' tab showing a single row with the column 'count bigint' and the value '235'.

	count bigint
1	235

### Task 3. Objective:

The objective was to find out which salesperson in the customer table represents the LEAST number of customers?

### SQL Query:

```
Select SalesPerson, Count (*) AS customer_count
FROM customer
GROUP BY SalesPerson
ORDER BY customer_count ASC
Limit 1;
```

### Explanation of Query Logic:

**Select SalesPerson, Count (\*) AS customer\_count:** The clause needed a column to identify the salesperson. I wanted the least number, so I ordered by `COUNT (*)` in ascending order. I gave `COUNT (*)` an **alias** to make ordering easier.

**FROM customer:** This clause designated the `customer` table as the source of the data for the query.

**GROUP BY SalesPerson:** I told SQL to apply `Count (*)` to each unique `SalesPerson`.

**ORDER BY customer\_count ASC:** sorting from least to greatest

**Limit 1;** : for one row.

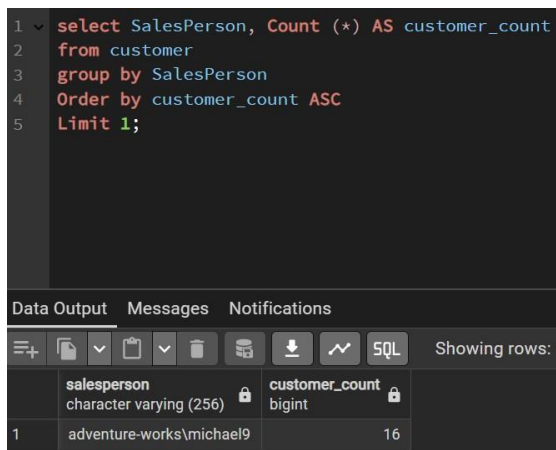
### Result/Output:

The query successfully returned the salesperson adventure-works\michael9 had the LEAST number of customers.

### Challenges Encountered:

I had one significant challenge for this question. Initially, I encountered an ERROR: column "salespersonID" does not exist due to an incorrect column name. PostgreSQL's helpful hint, Perhaps you meant to reference the column "customer.salesperson", guided me to the correct column. After verifying with the ERD, I confirmed the column was indeed SalesPerson, resolving the error.

```
1 select SalesPerson, Count (*) AS customer_count
2 from customer
3 group by SalesPerson
4 Order by customer_count ASC
5 Limit 1;
```



salesperson	customer_count
adventure-works\michael9	16

### Task 4. Objective:

Which of the following company names appears multiple times in our customer table?

### SQL Query:

```
SELECT CompanyName
FROM customer
GROUP By CompanyName
HAVING COUNT (*) > 1
```

### Explanation of Query Logic:

**SELECT CompanyName:** Selects the CompanyName  
**FROM customer:** Retrieves the data from the customer table  
**GROUP By CompanyName:** Groups the rows by CompanyName to count occurrences of each unique company

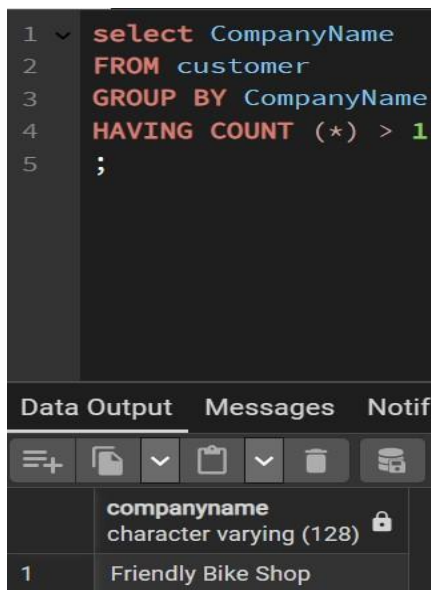
`HAVING COUNT (*) > 1;` : Filters those groups to include only the ones where the count of customers (`COUNT (*)`) is greater than 1, meaning they appear multiple times.

### Result/Output:

Friendly Bike Shop appears multiple times in our customer table.

### Challenges Encountered:

The challenge I encountered was a syntax error at or near “HAVING” Line 4:, `HAVING COUNT (*)`. I included an extra open parenthesis where one didn’t need to be.



The screenshot shows a SQL IDE with a query editor and a results pane. The query editor contains the following SQL code:

```
1 select CompanyName
2 FROM customer
3 GROUP BY CompanyName
4 HAVING COUNT (*) > 1
5 ;
```

The results pane shows the output of the query:

	companyname character varying (128)
1	Friendly Bike Shop

### Task 5. Objective:

Which customer has the MOST RECENT modified date in the customer table?

### SQL Query:

```
SELECT FirstName, LastName, ModifiedDate
FROM customer -- <--- Correct table name
ORDER BY ModifiedDate DESC
LIMIT 1;
```

### Explanation of Query Logic:

`SELECT FirstName, LastName, ModifiedDate`: I needed to find the customer’s name in the most recent modified date in the customer table.  
`FROM customer -- <--- Correct table name`:

**ORDER BY ModifiedDate DESC:** To find the most recent, I needed to ORDER BY the ModificationDatecolumn in the descending order.

**LIMIT 1:** I limited 1 to get just the single most recent one.

### Result/Output:

Jay Adams (company: Valley Bicycle Specialists) was the customer that had the MOST RECENT modified date in the customer table.

### Challenges Encountered:

I incorrectly named the table name as costumertable when the table name was customer.

```
1 select FirstName, LastName, ModifiedDate
2 FROM customer
3 ORDER BY ModifiedDate DESC
4 LIMIT 1
5 ;
```

Data Output Messages Notifications

Showing rows: 1 to 1 Page

	firstname character varying (50)	lastname character varying (50)	modifieddate timestamp without time zone (3)
1	Jay	Adams	2009-05-16 16:33:33.123

### Task 6. Objective:

How many company names in our customer table contain the word "bike"?

### SQL Query:

```
SELECT COUNT(DISTINCT CompanyName)
FROM customer
WHERE CompanyName ILIKE '%bike%';
```

### Explanation of Query Logic:

**SELECT COUNT(DISTINCT CompanyName):** I needed to obtain the unique count of the company names that contained the word 'bike'

**FROM customer:**

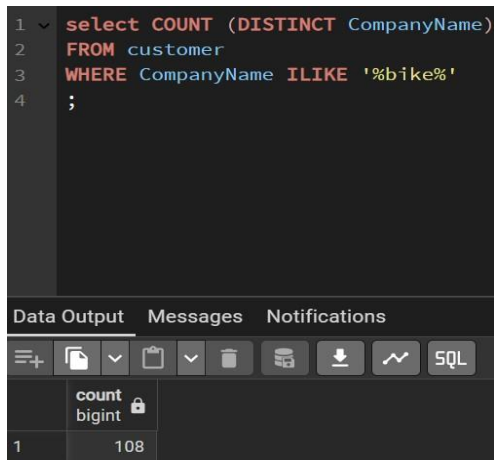
**WHERE CompanyName ILIKE '%bike%':** I needed to find strings that contained a specific word, so I needed to use the LIKE operator and a wildcard character (%). ILIKE converts strings to lower/upper case before comparison. The wildcard matches zero or more characters.

## Result/Output:

There were 108 company names in the customer table that contained the word 'bike'

## Challenges Encountered:

My initial query resulted in the answer as 6 and I got the answer wrong. I realized I only had the % after the word 'bike' ('bike%') when it should have been '%bike%'



The screenshot shows a SQL IDE with a query editor and a results pane. The query in the editor is:

```
1 select COUNT (DISTINCT CompanyName)
2 FROM customer
3 WHERE CompanyName ILIKE '%bike%'
4 ;
```

The results pane shows a table with one row:

count bigint
108

## Task 7. Objective:

How many companies have an address in the US state of Colorado?

## SQL Query:

**select COUNT (DISTINCT customer.CompanyName)**

```
select COUNT (DISTINCT customer.CompanyName)
FROM customer
    INNER JOIN customeraddress on customer.customerid =
customeraddress.customerid
    INNER JOIN address on customeraddress.addressid = address.addressid
WHERE address.StateProvince = 'Colorado';
```

## Explanation of Query Logic:

**select COUNT (DISTINCT customer.CompanyName):** I needed to get the count of the unique companies.

**FROM customer:** I needed data from the customer table (for company info)

**INNER JOIN customeraddress on customer.customerid = customeraddress.customerid:** These are the common columns that linked these tables. The JOIN operator brings these tables together. An INNER JOIN is appropriate when matching records for all tables.



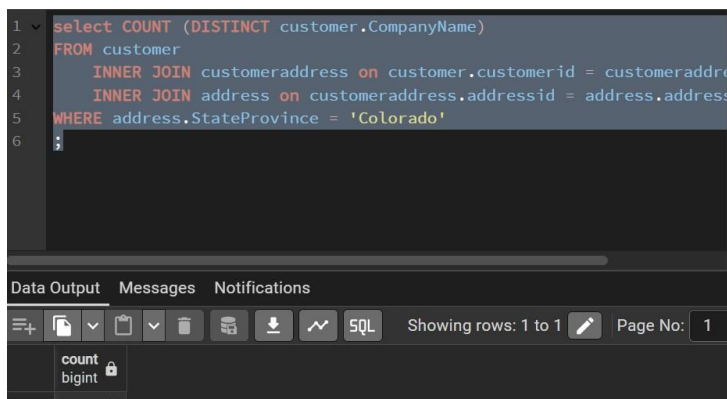
INNER JOIN address on customeraddress.addressid = address.addressid:  
WHERE address.StateProvince = 'Colorado'; The WHERE clause was used to filter for  
State Province that is equal to Colorado.

### Result/Output:

Eight companies have an address in the US state of Colorado.

### Challenges Encountered:

Challenges in this section was attention detail. I initially had Providence instead on  
Province and I had commas before INNER JOIN, thereby receiving syntax errors.

A screenshot of a SQL query editor interface. The query is as follows:

```
1 select COUNT (DISTINCT customer.CompanyName)
2 FROM customer
3     INNER JOIN customeraddress on customer.customerid = customeraddress.customerid
4     INNER JOIN address on customeraddress.addressid = address.addressid
5 WHERE address.StateProvince = 'Colorado'
6 ;
```

The interface includes a 'Data Output' tab at the bottom, which shows a single row with the column 'count' and the value 'bigint'. The status bar at the bottom indicates 'Showing rows: 1 to 1' and 'Page No: 1'.

### Task 8. Objective:

How many companies are based in the United Kingdom?

### SQL Query:

```
select COUNT (DISTINCT customer.CompanyName)
FROM customer
    INNER JOIN customeraddress on customer.customerid =
customeraddress.customerid
    INNER JOIN address on customeraddress.addressid = address.addressid
WHERE address.countryregion = 'United Kingdom';
```

### Explanation of Query Logic:

select COUNT (DISTINCT customer.CompanyName): I needed to get the count of the  
unique companies.

FROM customer: I needed data from the customer table (for company info)

INNER JOIN customeraddress on customer.customerid =  
customeraddress.customerid: These are the common columns that linked these tables.

The JOIN operator brings these tables together. An INNER JOIN is appropriate when matching records for all tables.

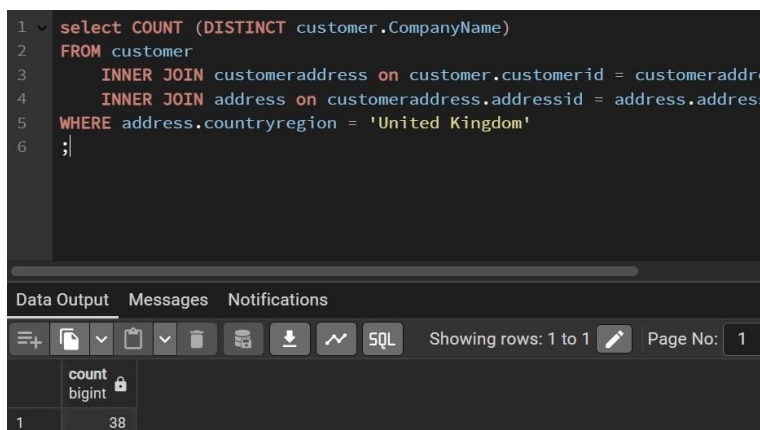
INNER JOIN address on customeraddress.addressid = address.addressid:  
WHERE address.countryregion = 'United Kingdom'; The WHERE clause was used to filter for country region that is equal to United Kingdom.

### Result/Output:

There are 38 companies based in the United Kingdom.

### Challenges Encountered:

No challenges encountered. I replaced address.StateProvince = 'Colorado'; with  
address.countryregion = 'United Kingdom';



```
1 select COUNT (DISTINCT customer.CompanyName)
2 FROM customer
3     INNER JOIN customeraddress on customer.customerid = customeraddress.customerid
4     INNER JOIN address on customeraddress.addressid = address.addressid
5 WHERE address.countryregion = 'United Kingdom'
6 ;
```

count bigint
38

### Task 9. Objective:

How many companies in our database have more than one address?

### SQL Query:

```
select COUNT (*)
FROM customeraddress
GROUP BY CustomerID
HAVING COUNT(*) > 1;
```

### Explanation of Query Logic:

select COUNT (\*) : This clause is used to count the number of address for each company.  
FROM customeraddress : this clause is used to figure out the address from the customers.

GROUP BY CustomerID : this clause is used as the company identifier and is generally more robust for grouping more unique companies.

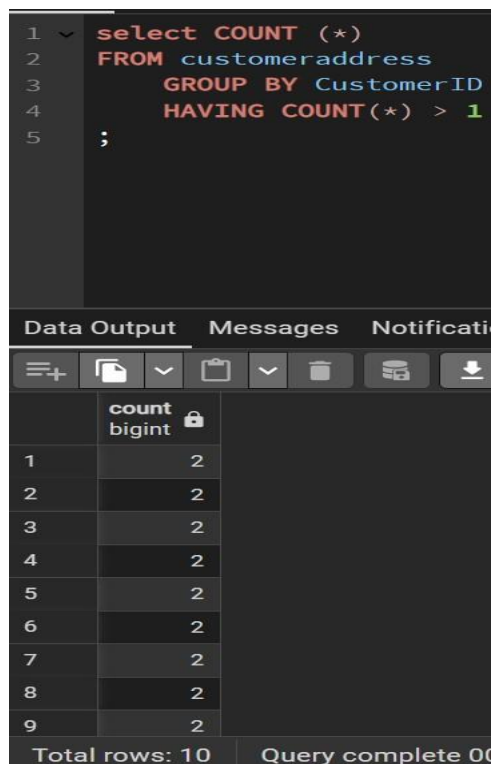
**HAVING COUNT(\*) > 1;** : The HAVING clause is used to filter these groups, keeping only those where the count of addresses is greater than 1. The COUNT clause returns a list of counts for each customer who has more than one address.

### Result/Output:

There are 10 companies in the database that have more than one address.

### Challenges Encountered:

I encountered several challenges. I am still having a difficult time figuring out what words are used where.



The screenshot shows a SQL query editor with a query and its results. The query is:

```
1 select COUNT (*)
2 FROM customeraddress
3 GROUP BY CustomerID
4 HAVING COUNT(*) > 1
5 ;
```

The results are displayed in a table with the following columns: **count** (type: bigint) and **bigint** (type: bigint). The table contains 10 rows, each with a count of 2. The status bar at the bottom indicates "Total rows: 10" and "Query complete 00".

	count	bigint
1	2	
2	2	
3	2	
4	2	
5	2	
6	2	
7	2	
8	2	
9	2	
Total rows: 10		Query complete 00

### Task 10. Objective:

Which of the following cities is NOT included among the list of companies that have more than one address?

### SQL Query:

```
SELECT DISTINCT A.City
FROM address AS A
INNER JOIN customeraddress AS CA ON A.addressid = CA.addressid
WHERE CA.customerid IN (
    SELECT customerid
```

```
FROM customeraddress  
GROUP BY customerid  
HAVING COUNT(*) > 1  
);
```

#### **Explanation of Query Logic:**

**SELECT DISTINCT A.City:** This clause tells SQL to get unique City names. I used A.City because I'll be giving the address table an alias A.

**FROM address AS A:** I start from the address table and give it a short alias A for easier reference.

**INNER JOIN customeraddress AS CA ON A.addressid = CA.addressid:** I joined address (aliased as A) with customer address (aliased as CA). They link on addressID. This brings the customerid column from customeraddress into the picture alongside the address details.

**WHERE CA.customerid IN (:** This is the crucial filter. It says, "only include rows where the customerid from customeraddress (aliased as CA) is present in the list generated by my subquery."

```
SELECT customerid  
FROM customeraddress  
GROUP BY customerid  
HAVING COUNT(*) > 1
```

**);** This is the exact subquery that was developed in task 7 that identifies all the customerid's who have more than one address.

#### **Result/Output:**

London was not included among the list of companies that had more than one address.

#### **Challenges Encountered:**

I encountered every problem because I tried to reference task 7 but none of the queries were alike. I assumed that I needed to use the NOT IN clause as well.

```

1 SELECT DISTINCT A.City
2 FROM address AS A
3 INNER JOIN customeraddress AS CA ON A.addressid = CA.addressid
4 WHERE CA.customerid IN (
5     SELECT customerid
6     FROM customeraddress
7     GROUP BY customerid
8     HAVING COUNT(*) > 1
9 );

```

Data Output Messages Notifications

Showing rows: 1 to 9 Page 1

city
1 Austin
2 Montreal
3 Bothell
4 Minneapolis
5 Phoenix
6 Dallas
7 Denver
8 Renton
9 Bellevue

Total rows: 9 Query complete 00:00:00.055

### Task 11. Objective:

How many unique companies are located in the US state of Oregon?

### SQL Query:

```

select COUNT (DISTINCT customer.CompanyName)
FROM customer
    INNER JOIN customeraddress on customer.customerid =
customeraddress.customerid
    INNER JOIN address on customeraddress.addressid = address.addressid
WHERE address.StateProvince = 'Oregon';

```

### Explanation of Query Logic:

`select COUNT (DISTINCT customer.CompanyName)`: I needed to get the count of the unique companies.

`FROM customer`: I needed data from the customer table (for company info)

`INNER JOIN customeraddress on customer.customerid = customeraddress.customerid`: These are the common columns that linked these tables. The JOIN operator brings these tables together. An INNER JOIN is appropriate when matching records for all tables.

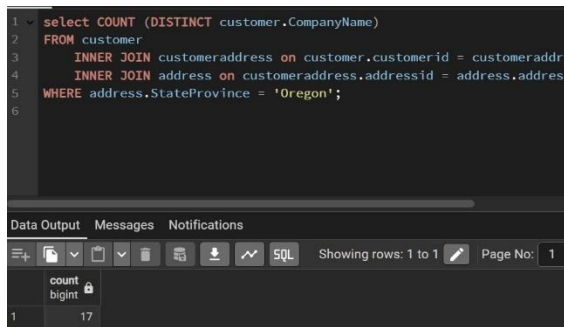
`INNER JOIN address on customeraddress.addressid = address.addressid`: `WHERE address.StateProvince = 'Oregon'`: The WHERE clause was used to filter for State Province that is equal to Oregon.

### Result/Output:

There are 17 unique companies that are located in the US state of Oregon.

### Challenges Encountered:

No challenges encountered. I used the same query as I did with task 7 and replaced Colorado with Oregon.



```
1 select COUNT (DISTINCT customer.CompanyName)
2 FROM customer
3     INNER JOIN customeraddress on customer.customerid = customeraddress.customerid
4     INNER JOIN address on customeraddress.addressid = address.addressid
5 WHERE address.StateProvince = 'Oregon';
6
```

The screenshot shows a SQL query editor with a dark theme. The query is as follows:

```
1 select COUNT (DISTINCT customer.CompanyName)
2 FROM customer
3     INNER JOIN customeraddress on customer.customerid = customeraddress.customerid
4     INNER JOIN address on customeraddress.addressid = address.addressid
5 WHERE address.StateProvince = 'Oregon';
6
```

Below the query editor, there is a toolbar with icons for Data Output, Messages, and Notifications. The Data Output tab is active, showing a table with one row and one column:

count bigint
17

The table has a single row with the value 17. The interface also shows "Showing rows: 1 to 1" and "Page No: 1".

### Task 12. Objective:

If you take all the customers with first name "John" and CROSS JOIN them to all the customers with first name "Margaret," how many rows will your query return?

### SQL Query:

```
SELECT COUNT(*)
FROM customer
WHERE FirstName = 'John';
```

```
SELECT COUNT(*)
FROM customer
WHERE FirstName = 'Margaret';
```

### Explanation of Query Logic:

```
SELECT COUNT(*) :
FROM customer :
WHERE FirstName = 'John';
```

I first needed to find the number of customers in each set.

```
SELECT COUNT(*) :
FROM customer :
WHERE FirstName = 'Margaret';
```

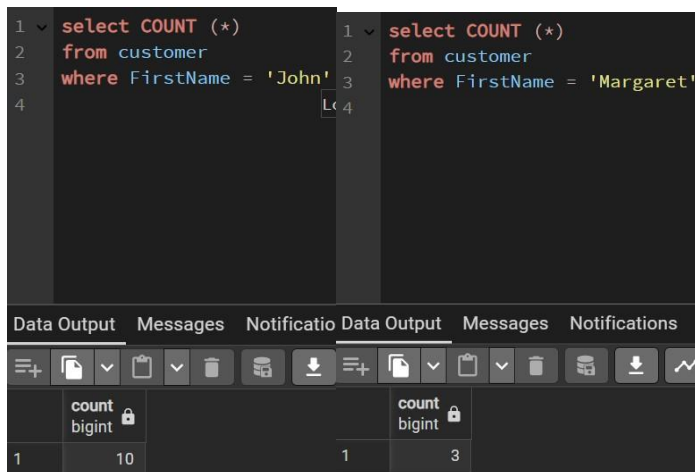
I first needed to find the number of customers in each set.

## Result/Output:

When you take all the customers with the first name “John” and CROSS JOIN them to all the customers with the first name “Margaret” the query will return that John had 10 rows and Margaret had 3 rows. Cross joining means that you multiply  $10 * 3$  to get a total numbers of rows from set A and set B.

## Challenges Encountered:

The primary challenge was ensuring a correct understanding of how CROSS JOIN operates., specifically that it produces a Cartesian product and thus requires multiplication of row counts rather than addition. Once the concept of combining every row from another set was firmly grasped, calculating the expected output became straight forward.



The screenshot shows two SQL queries and their results. The first query counts the number of customers with the first name 'John', resulting in 10. The second query counts the number of customers with the first name 'Margaret', resulting in 3.

Query	SQL Statement	Output
1	<code>select COUNT (*) from customer where FirstName = 'John'</code>	count bigint 10
2	<code>select COUNT (*) from customer where FirstName = 'Margaret'</code>	count bigint 3

## 6. Key Learnings

This lab significantly enhanced my SQL proficiency, particularly in handling complex queries and debugging. The main takeaways include:

- Mastered the application of **AND**, **OR**, and **NOT** for building complex and precise filtering conditions.
- Understood how to use **LIKE** for effective pattern matching with wildcards (**%** and **\_**), crucial for flexible data searches.
- Gained experience filtering based on various data types, including dates, times, and Boolean values, ensuring accurate data retrieval.
- Learned the importance of specific column selection (**SELECT column1, column2**) versus **SELECT \*** for optimizing query performance and improving data clarity.
- Acquired robust skills in joining multiple tables (e.g., **INNER JOIN**) across different schemas (**customer, salesorderheader, salesorderdetail, product, productmodel,**

productcategory, productsubcategory, address) to extract interconnected data, which is fundamental for real-world analysis.

- Reinforced the use of aggregate functions like `SUM()` with `GROUP BY` to summarize data effectively, along with the `HAVING` clause to filter aggregated results.
- Gained practical experience with advanced ranking functions like `DENSE_RANK()` and their application within Common Table Expressions (`WITH` clause) for complex analytical tasks such as identifying top performers or tied ranks.
- Significantly improved debugging skills in the SQL command-line environment. This included:
  - o Proactively inspecting database schema using `information_schema.columns` and `information_schema.tables` to correctly identify table and column names (e.g., `companyname` vs. `name`, United Kingdom vs. England).
  - o Understanding and resolving `FROM`-clause entry errors related to SQL scope in nested queries and CTEs.
  - o Diagnosing "no results" outputs, distinguishing between query syntax errors and limitations stemming from data sparsity or inconsistencies within the specific database instance. This emphasized the critical difference between a syntactically correct query and one that yields expected results based on available data.

## 7. Conclusion / Summary

This lab activity significantly enhanced my ability to precisely extract, filter, aggregate, and rank data using advanced SQL concepts within the PgAdmin 4 environment. Navigating unexpected database schema nuances and debugging "no results" scenarios provided invaluable real-world experience, reinforcing the importance of thorough data exploration and systematic problem-solving. The skills developed here are directly applicable to any data-driven role, enabling more efficient and accurate analysis crucial for informed decision-making.