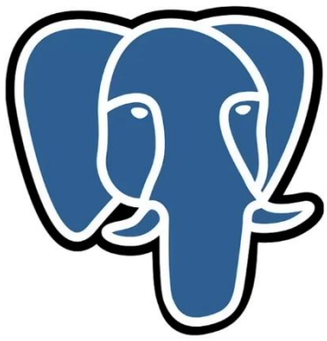


POSTGRESQL TUTORIAL!!!!!

CONTENTS

DOWNLOADS

Time to get set-up for the course!

We will install

PostgreSQL (also known as Postgres) is an open-source relational database management system (RDBMS) that is known for its robustness, reliability, and advanced features. It provides a powerful and scalable platform for managing large volumes of structured data. PostgreSQL supports a wide range of data types, indexing options, and query optimization techniques, making it suitable for various applications. It offers ACID (Atomicity, Consistency, Isolation, Durability) compliance and supports transactions, which ensures data integrity and reliability. With its extensibility and support for various programming languages, PostgreSQL is widely used in enterprise environments and by developers for building high-performance, data-driven applications

Time to get set-up for the course!

We will install

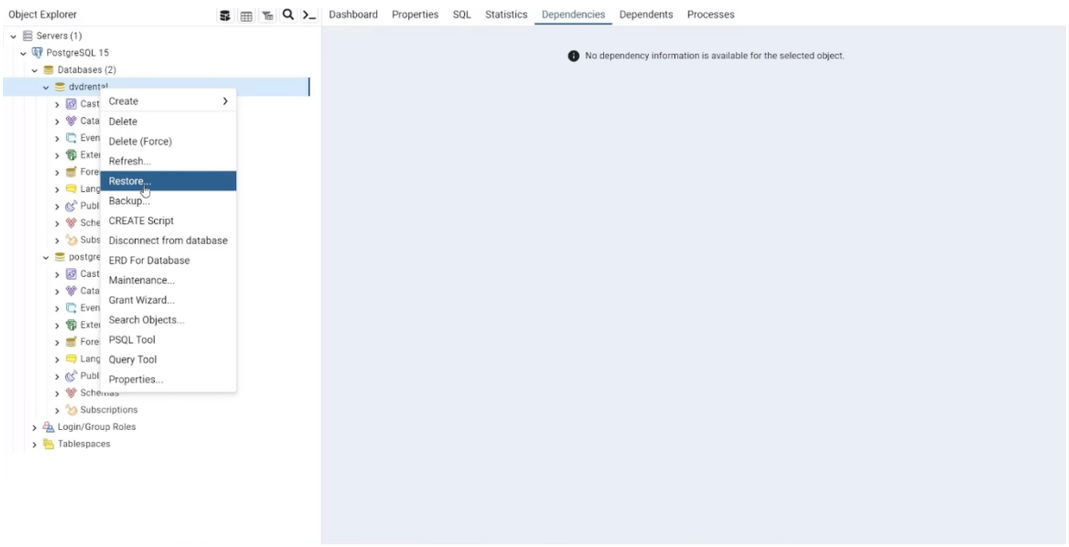
pgAdmin is a free and open-source administration and development platform for managing PostgreSQL databases. It provides a graphical user interface (GUI) that allows users to interact with the database and perform various administrative tasks. pgAdmin offers a range of features, including database object management (such as creating tables, views, and indexes), querying and editing data, monitoring database activity, and managing server settings. It provides a user-friendly environment for database administrators and developers to efficiently work with PostgreSQL databases, allowing them to visually design database schemas, write and execute SQL queries, and monitor database performance. pgAdmin is available for multiple operating systems and is widely used as a primary tool for PostgreSQL database administration.

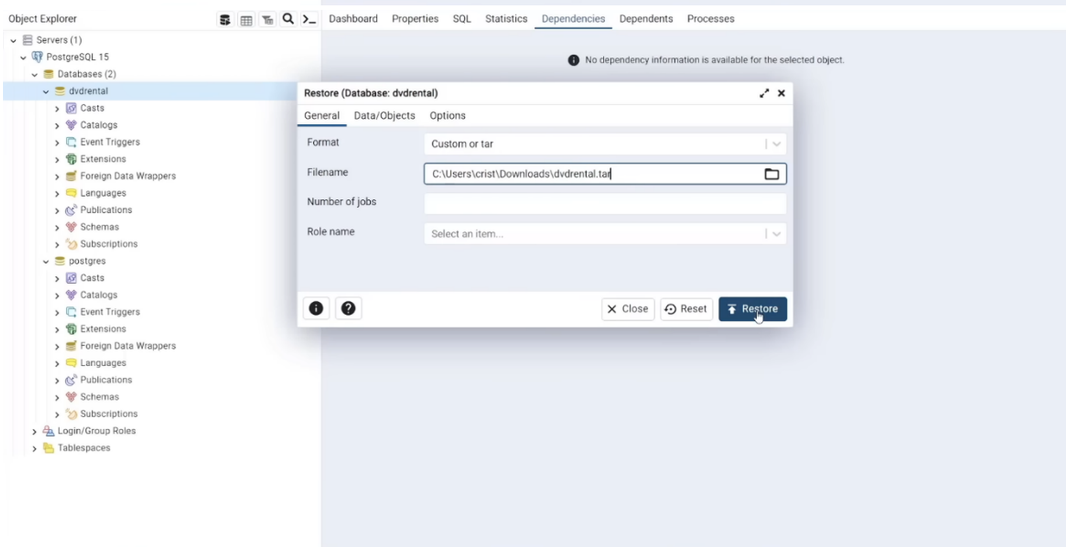
Time to get set-up for the course!

We will install

PostgreSQL - SQL Engine that stores data and read queries and returns information.

PgAdmin - Graphical User Interface for connectiog with PostgreSQL

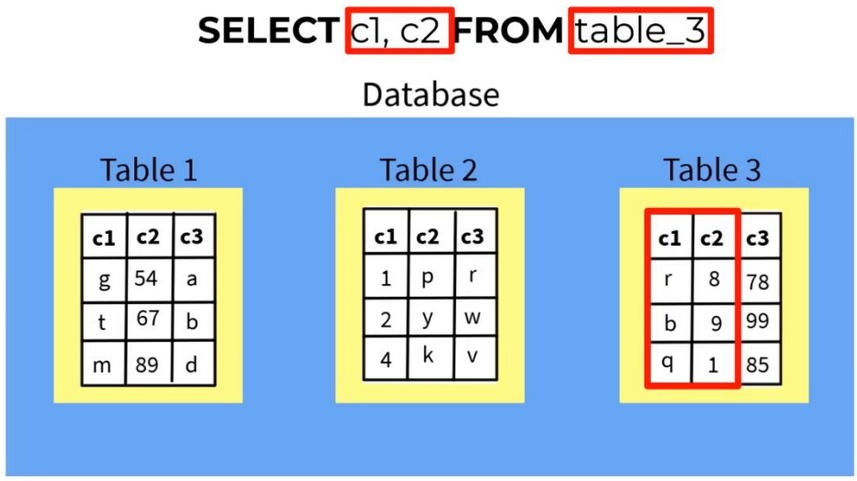

Object Explorer
Dashboard
Properties
SQL
Statistics
Dependencies
Dependents
Processes
Servers (1)
PostgreSQL 15
Databases (2)
dvdrental
Create
Delete
Delete (Force)
Refresh...
Restore...
Backup...
CREATE Script
Disconnect from database
ERD For Database
Maintenance...
Grant Wizard...
Search Objects...
PSQL Tool
Query Tool
Properties...
Subscriptions
Login/Group Roles
Tablespaces
No dependency information is available for the selected object.

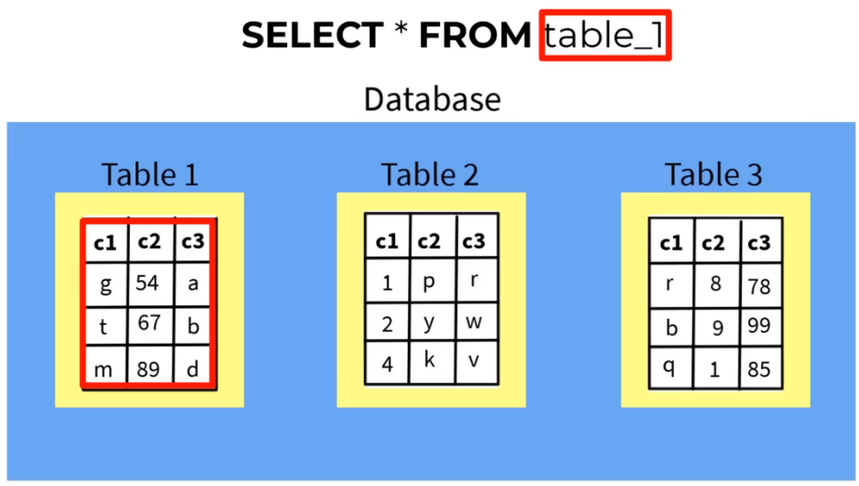

Object Explorer
Dashboard
Properties
SQL
Statistics
Dependencies
Dependents
Processes
Servers (1)
PostgreSQL 15
Databases (2)
dvdrental
Casts
Catalogs
Event Triggers
Extensions
Foreign Data Wrappers
Languages
Publications
Schemas
Subscriptions
postgres
Login/Group Roles
Tablespaces
No dependency information is available for the selected object.
Restore (Database: dvdrental)
General
Data/Objects
Options
Format
Custom or tar
Filename
C:\Users\crist\Downloads\dvdrental.tar
Number of jobs
Role name
Select an item...
Close
Reset
Restore

SELECT

SELECT statement is used to retrieve data from one or more database tables. It allows you to specify the columns you want to retrieve, the table(s) from which you want to retrieve the data, and any conditions that must be met for the data to be included in the result set.

Later we will learn how to combine **SELECT** statement with other SQL statements to perform more complex queries.





SELECT DISCTINCT

In SQL, the **DISTINCT** statement is used to filter the result set of a query and retrieve only unique rows from the specified column or combination of columns. It is particularly useful when you want to eliminate duplicate records from the query results.

Example:
Consider a table called "employees" with the following data:

employee_id	first_name	last_name	department
1	John	Smith	HR
2	Jane	Doe	IT
3	John	Smith	HR
4	Alice	Johnson	HR

If you run the following query:

SELECT DISTINCT first_name, last_name
FROM employees;

The result will be:

first_name	last_name
John	Smith
Jane	Doe
Alice	Johnson

COUNT

COUNT Function in SQL:

The **COUNT** function in SQL is an aggregate function that allows you to count the number of rows that match a specified condition within a table or a result set. It returns a single value representing the count of rows that meet the criteria.

How to Use **COUNT** with the **DISTINCT** Statement:

The **DISTINCT** keyword is used to eliminate duplicate rows from the resul set. When combined with the **COUNT** function, it allows you to count the number of unique values in a specific column. Here's how you can use it:

Let's consider a sample table "employees":

emp_id	emp_name	departament
1	John	HR
2	Alice	IT
3	Bob	Finance
4	Emily	HR
5	Michael	IT

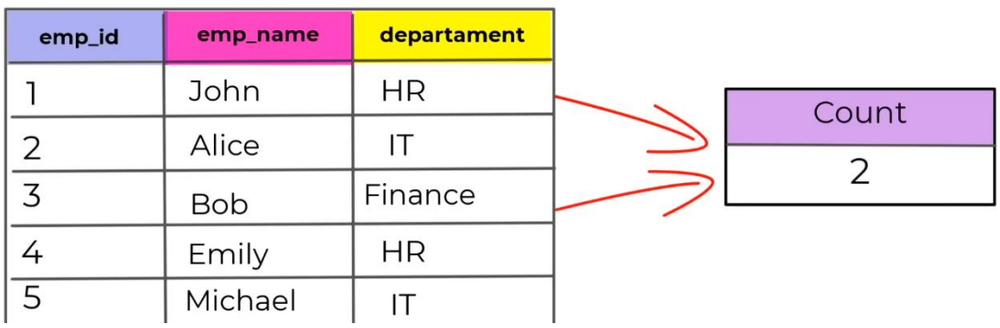
Example 1: Count all employees in the "employees" table.

SELECT COUNT(*) FROM employees;

Count
5

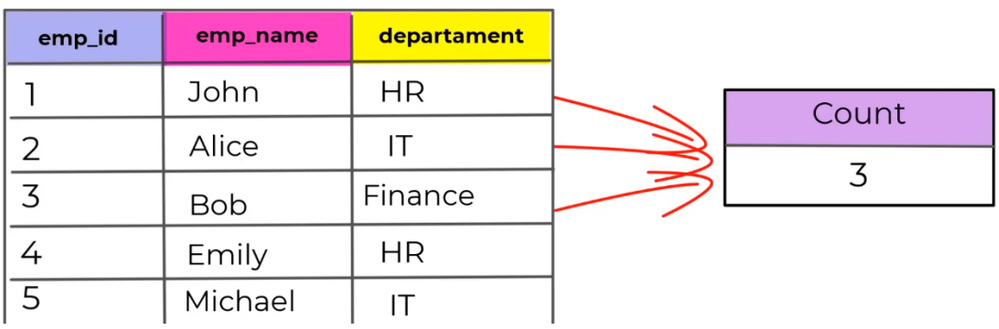
Example 2: Count the number of unique departments in the "employees" table using **DISTINCT.**

SELECT COUNT(**DISTINCT** department)
FROM employees;



Example 2: Count the number of unique departments in the "employees" table using **DISTINCT.**

SELECT COUNT(**DISTINCT** department)
FROM employees;



In this example, we used the **COUNT** function with **DISTINCT** to count the unique departments in the "employees" table. It returned the count as 3 because there are three unique departments: HR, IT, and Finance.

I hope these examples help illustrate the usage and usefulness of the **COUNT** function in SQL, particularly when combined with the **DISTINCT** statement. It allows you to perform various data analysis tasks and retrieve valuable information from your database.

In SQL, the usage of the **COUNT** function with and without asterix depends on what you want to count. Let's explore the differences and when to use each form:

1. **COUNT with Parentheses and asterix:**
When you use **COUNT(*)** with parentheses and asterix, it counts all the rows in the specified table, regardless of the values in the columns. Here's the syntax:

SELECT COUNT(*) FROM table_name;

Usage:
Use **COUNT(*)** when you want to count the total number of rows in a table, irrespective of whether there are NULL values or not. It counts all rows, including those with NULL values in any column.

COUNT(*):
The **COUNT(*)** form of the **COUNT** function counts all rows in the specified table or result set, regardless of whether there are NULL values in the columns. It essentially counts the total number of rows, including those with NULL values in any column.
Here's the syntax:
Example:
Consider a table "students":

student_id	student_name	age
1	Alice	20
2	Bob	21
3	Chris	NULL
4	David	19
5	NULL	22

COUNT(column_name):
The **COUNT(column_name)** form of the **COUNT** function counts the number of non-NULL values in th specified column. It ignores NULL values in that particular column and only counts the rows where the specified column has a non-NULL value.

Here's the syntax:

SELECT COUNT(student_name) FROM students;

Usage:
We use **COUNT(column_name)** when you want to count the occurence of NON-NULL values in a specific column. It ignore NULL values in the column and counts only NON-NULL entries.

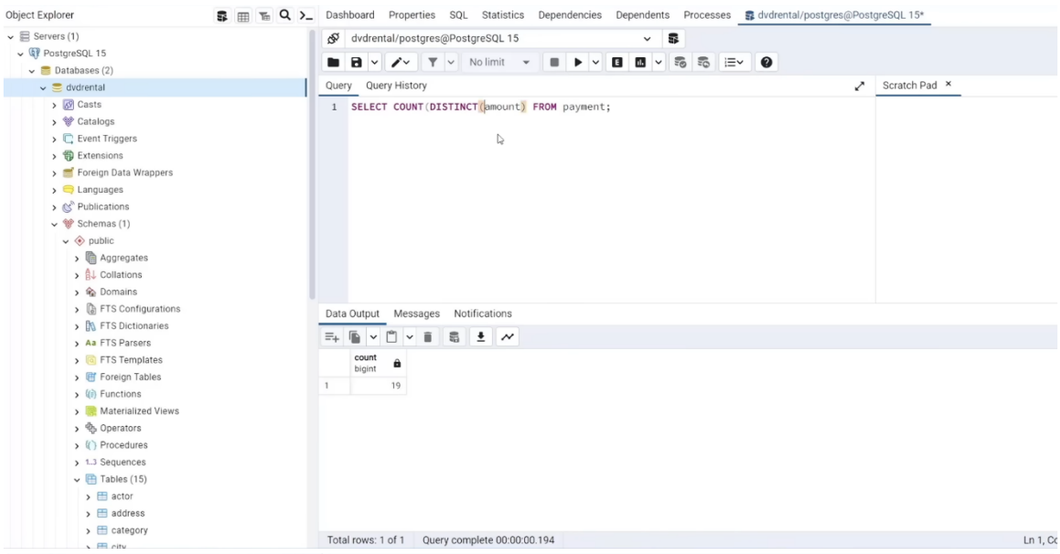
SELECT COUNT(student_name)
FROM table_name;

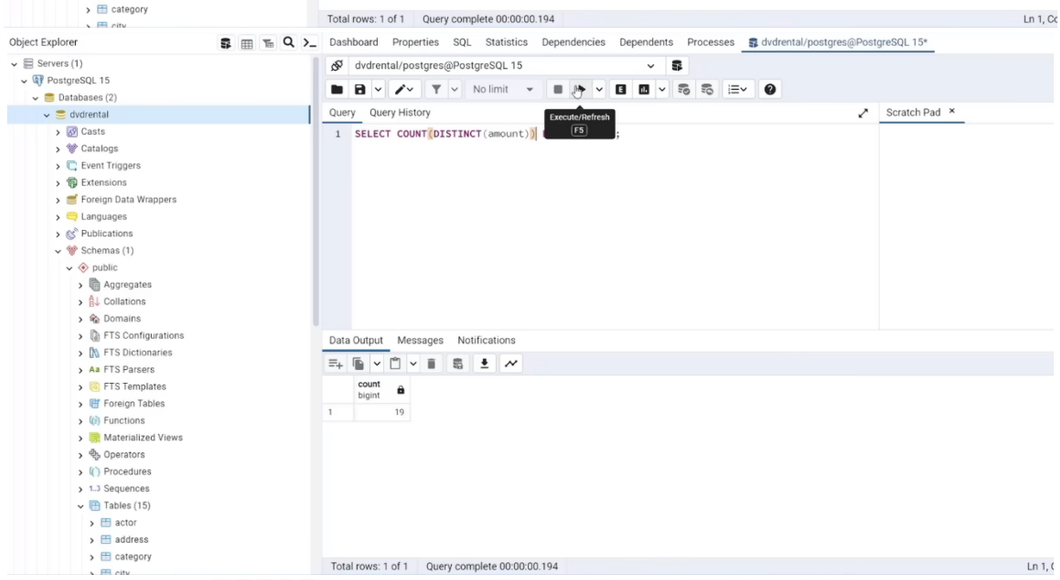
student_id	student_name	age
1	Alice	20
2	Bob	21
3	Chris	NULL
4	David	19
5	NULL	22

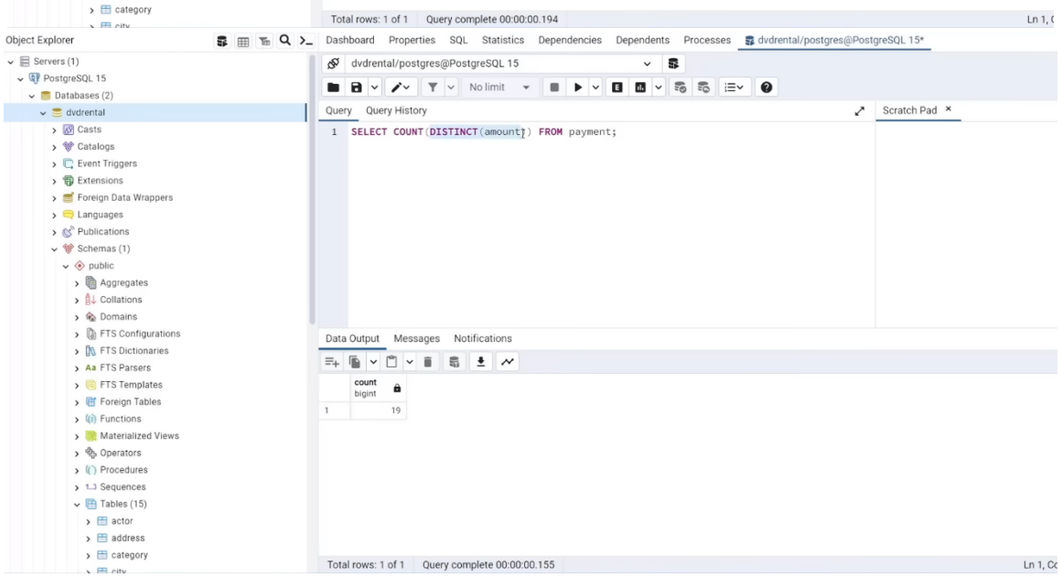
Count
4

In summary:

COUNT(*): Counts all rows in the table, including rows with NULL values in an column.
COUNT(column_name): Counts the number of non-NULL values in the specified column, ignoring rows with NULL values in that column.
Choose the appropriate form of the **COUNT** function based on your specific requirement. If you want to count all rows in the table, regardless of NULL values, use **COUNT(*)**. If you want to count the number of non-NULL values in specific column, use COUNT(column_name).


Object Explorer
Servers (1)
PostgreSQL 15
Databases (2)
dvdrental
Casts
Catalogs
Event Triggers
Extensions
Foreign Data Wrappers
Languages
Publications
Schemas (1)
public
Aggregates
Collations
Domains
FTS Configurations
FTS Dictionaries
FTS Parsers
FTS Templates
Foreign Tables
Functions
Materialized Views
Operators
Procedures
Sequences
Tables (15)
actor
address
category
Dashboard
Properties
SQL
Statistics
Dependencies
Dependents
Processes
dvdrental/postgres@PostgreSQL 15*
dvdrental/postgres@PostgreSQL 15
No limit
Query
Query History
Scratch Pad
1 SELECT COUNT(DISTINCT(amount)) FROM payment;
Data Output
Messages
Notifications
count
bigint
1 19
Total rows: 1 of 1
Query complete 00:00:00.194
Ln 1, Co


Object Explorer
Servers (1)
PostgreSQL 15
Databases (2)
dvdrental
Casts
Catalogs
Event Triggers
Extensions
Foreign Data Wrappers
Languages
Publications
Schemas (1)
public
Aggregates
Collations
Domains
FTS Configurations
FTS Dictionaries
FTS Parsers
FTS Templates
Foreign Tables
Functions
Materialized Views
Operators
Procedures
Sequences
Tables (15)
actor
address
category
Dashboard
Properties
SQL
Statistics
Dependencies
Dependents
Processes
dvdrental/postgres@PostgreSQL 15*
dvdrental/postgres@PostgreSQL 15
No limit
Query
Query History
Execute/Refresh
F5
Scratch Pad
1 SELECT COUNT(DISTINCT(amount))
Data Output
Messages
Notifications
count
bigint
1 19
Total rows: 1 of 1
Query complete 00:00:00.194


Object Explorer
Servers (1)
PostgreSQL 15
Databases (2)
dvdrental
Casts
Catalogs
Event Triggers
Extensions
Foreign Data Wrappers
Languages
Publications
Schemas (1)
public
Aggregates
Collations
Domains
FTS Configurations
FTS Dictionaries
FTS Parsers
FTS Templates
Foreign Tables
Functions
Materialized Views
Operators
Procedures
Sequences
Tables (15)
actor
address
category
Dashboard
Properties
SQL
Statistics
Dependencies
Dependents
Processes
dvdrental/postgres@PostgreSQL 15*
dvdrental/postgres@PostgreSQL 15
No limit
Query
Query History
Scratch Pad
1 SELECT COUNT(DISTINCT(amount)) FROM payment;
Data Output
Messages
Notifications
count
bigint
1 19
Total rows: 1 of 1
Query complete 00:00:00.155

SELECT WHERE

The SQL **WHERE** statement is a crucial part of querying and filtering data in a relational database. It allows you to retrieve specific rows from a table that match certain conditions. In this detailed step-by-step explanation, I'll cover everything you need to know about the SQL **WHERE** statement:

Basic Syntax:
The **WHERE** clause is typically used in combination with the **SELECT** statement to filter rows based on specific conditions. The basic syntax is as follows:

Basic Syntax:
The **WHERE** clause is typically used in combination with the **SELECT** statement to filter rows based on specific conditions. The basic syntax is as follows:

```
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

• **SELECT**: Specifies the columns you want to retrieve from the table.
• **FROM**: Indicates the name of the table you are querying.
• **WHERE**: Introduces the condition that filters the rows. The condition is evaluated for each row in the table, and only the rows that meet the condition will be included in the result set

A condition is a logical expression that evaluates to true or false for each row in the table. The **WHERE** clause uses these conditions to determine which rows to include in the result set. Common operators used in conditions include:

- **=**: Equal to
- **<> or !=**: Not equal to
- **>**: Greater than
- **<**: Less than
- **>=**: Greater than or equal to
- **<=**: Less than or equal to
- **BETWEEN**: Between a range of values (inclusive)
- **LIKE**: Pattern matching using wildcards (often used with % and _)
- **IN**: Matches any value in a list
- **IS NULL**: Checks for NULL values
- **AND, OR, NOT**: Logical operators for combining multiple conditions

Equal To Operator (=):
The equal to operator (=) checks whether two values are exactly equal. It is used to filter rows where the specified column is equal to a given value.

employee_id	first_name	last_name	job_title	age	salary	department	experience	rating	phone_number
1	John	Doe	Manager	35	80000	Sales	7	4.5	555-123-4567
2	Jane	Smith	Supervisor	28	60000	HR	5	4.2	555-987-6543
3	Robert	Johnson	Analyst	42	75000	Finance	10	4.7	NULL
4	Jennifer	Williams	Manager	38	90000	Marketing	12	4.9	555-444-3333
5	Michael	Brown	Supervisor	32	650000	Sales	8	4.3	555-777-8888
6	Laura	Davis	Coordinator	27	550000	Admin	3	4.0	555-222-1111
7	James	Miller	Analyst	29	70000	HR	6	4.6	555-999-0000
8	Emily	Anderson	Supervisor	31	68000	Marketing	7	4.4	555-999-0000

```
SELECT * FROM Employees
WHERE department = 'Sales';
```

employee_id	first_name	last_name	job_title	age	salary	department	experience	rating	phone_number
1	John	Doe	Manager	35	80000	Sales	7	4.5	555-123-4567
2	Jane	Smith	Supervisor	28	60000	HR	5	4.2	555-987-6543
3	Robert	Johnson	Analyst	42	75000	Finance	10	4.7	NULL
4	Jennifer	Williams	Manager	38	90000	Marketing	12	4.9	555-444-3333
5	Michael	Brown	Supervisor	32	650000	Sales	8	4.3	555-777-8888
6	Laura	Davis	Coordinator	27	550000	Admin	3	4.0	555-222-1111
7	James	Miller	Analyst	29	70000	HR	6	4.6	555-999-0000
8	Emily	Anderson	Supervisor	31	68000	Marketing	7	4.4	555-999-0000

SELECT * FROM Employees
WHERE department = 'Sales';

employee_id	first_name	last_name	job_title	age	salary	department	experience	rating	phone_number
1	John	Doe	Manager	35	80000	Sales	7	4.5	555-123-4567
2	Jane	Smith	Supervisor	28	60000	HR	5	4.2	555-987-6543
3	Robert	Johnson	Analyst	42	75000	Finance	10	4.7	NULL
4	Jennifer	Williams	Manager	38	90000	Marketing	12	4.9	555-444-3333
5	Michael	Brown	Supervisor	32	650000	Sales	8	4.3	555-777-8888
6	Laura	Davis	Coordinator	27	550000	Admin	3	4.0	555-222-1111
7	James	Miller	Analyst	29	70000	HR	6	4.6	555-999-0000
8	Emily	Anderson	Supervisor	31	68000	Marketing	7	4.4	555-999-0000

SELECT * FROM Employees
WHERE department <> 'Finance';

employee_id	first_name	last_name	job_title	age	salary	department	experience	rating	phone_number
1	John	Doe	Manager	35	80000	Sales	7	4.5	555-123-4567
2	Jane	Smith	Supervisor	28	60000	HR	5	4.2	555-987-6543
3	Robert	Johnson	Analyst	42	75000	Finance	10	4.7	NULL
4	Jennifer	Williams	Manager	38	90000	Marketing	12	4.9	555-444-3333
5	Michael	Brown	Supervisor	32	650000	Sales	8	4.3	555-777-8888
6	Laura	Davis	Coordinator	27	550000	Admin	3	4.0	555-222-1111
7	James	Miller	Analyst	29	70000	HR	6	4.6	555-999-0000
8	Emily	Anderson	Supervisor	31	68000	Marketing	7	4.4	555-999-0000

SELECT * FROM Employees
WHERE age > 30;

employee_id	first_name	last_name	job_title	age	salary	department	experience	rating	phone_number
1	John	Doe	Manager	35	80000	Sales	7	4.5	555-123-4567
2	Jane	Smith	Supervisor	28	60000	HR	5	4.2	555-987-6543
3	Robert	Johnson	Analyst	42	75000	Finance	10	4.7	NULL
4	Jennifer	Williams	Manager	38	90000	Marketing	12	4.9	555-444-3333
5	Michael	Brown	Supervisor	32	650000	Sales	8	4.3	555-777-8888
6	Laura	Davis	Coordinator	27	550000	Admin	3	4.0	555-222-1111
7	James	Miller	Analyst	29	70000	HR	6	4.6	555-999-0000
8	Emily	Anderson	Supervisor	31	68000	Marketing	7	4.4	555-999-0000

SELECT * FROM Employees
WHERE salary < 50000;

employee_id	first_name	last_name	job_title	age	salary	department	experience	rating	phone_number
1	John	Doe	Manager	35	80000	Sales	7	4.5	555-123-4567
2	Jane	Smith	Supervisor	28	60000	HR	5	4.2	555-987-6543
3	Robert	Johnson	Analyst	42	75000	Finance	10	4.7	NULL
4	Jennifer	Williams	Manager	38	90000	Marketing	12	4.9	555-444-3333
5	Michael	Brown	Supervisor	32	650000	Sales	8	4.3	555-777-8888
6	Laura	Davis	Coordinator	27	550000	Admin	3	4.0	555-222-1111
7	James	Miller	Analyst	29	70000	HR	6	4.6	555-999-0000
8	Emily	Anderson	Supervisor	31	68000	Marketing	7	4.4	555-999-0000

SELECT * FROM Employees
WHERE experience >= 5;

employee_id	first_name	last_name	job_title	age	salary	department	experience	rating	phone_number
1	John	Doe	Manager	35	80000	Sales	7	4.5	555-123-4567
2	Jane	Smith	Supervisor	28	60000	HR	5	4.2	555-987-6543
3	Robert	Johnson	Analyst	42	75000	Finance	10	4.7	NULL
4	Jennifer	Williams	Manager	38	90000	Marketing	12	4.9	555-444-3333
5	Michael	Brown	Supervisor	32	650000	Sales	8	4.3	555-777-8888
6	Laura	Davis	Coordinator	27	550000	Admin	3	4.0	555-222-1111
7	James	Miller	Analyst	29	70000	HR	6	4.6	555-999-0000
8	Emily	Anderson	Supervisor	31	68000	Marketing	7	4.4	555-999-0000

SELECT * FROM Employees
WHERE rating <= 4.5;

employee_id	first_name	last_name	job_title	age	salary	department	experience	rating	phone_number
1	John	Doe	Manager	35	80000	Sales	7	4.5	555-123-4567
2	Jane	Smith	Supervisor	28	60000	HR	5	4.2	555-987-6543
3	Robert	Johnson	Analyst	42	75000	Finance	10	4.7	NULL
4	Jennifer	Williams	Manager	38	90000	Marketing	12	4.9	555-444-3333
5	Michael	Brown	Supervisor	32	650000	Sales	8	4.3	555-777-8888
6	Laura	Davis	Coordinator	27	550000	Admin	3	4.0	555-222-1111
7	James	Miller	Analyst	29	70000	HR	6	4.6	555-999-0000
8	Emily	Anderson	Supervisor	31	68000	Marketing	7	4.4	555-999-0000

SELECT * FROM Employees
WHERE age BETWEEN 25 AND 40;

employee_id	first_name	last_name	job_title	age	salary	department	experience	rating	phone_number
1	John	Doe	Manager	35	80000	Sales	7	4.5	555-123-4567
2	Jane	Smith	Supervisor	28	60000	HR	5	4.2	555-987-6543
3	Robert	Johnson	Analyst	42	75000	Finance	10	4.7	NULL
4	Jennifer	Williams	Manager	38	90000	Marketing	12	4.9	555-444-3333
5	Michael	Brown	Supervisor	32	650000	Sales	8	4.3	555-777-8888
6	Laura	Davis	Coordinator	27	550000	Admin	3	4.0	555-222-1111
7	James	Miller	Analyst	29	70000	HR	6	4.6	555-999-0000
8	Emily	Anderson	Supervisor	31	68000	Marketing	7	4.4	555-999-0000

SELECT * FROM Employees
WHERE first_name LIKE 'J%';

employee_id	first_name	last_name	job_title	age	salary	department	experience	rating	phone_number
1	John	Doe	Manager	35	80000	Sales	7	4.5	555-123-4567
2	Jane	Smith	Supervisor	28	60000	HR	5	4.2	555-987-6543
3	Robert	Johnson	Analyst	42	75000	Finance	10	4.7	NULL
4	Jennifer	Williams	Manager	38	90000	Marketing	12	4.9	555-444-3333
5	Michael	Brown	Supervisor	32	650000	Sales	8	4.3	555-777-8888
6	Laura	Davis	Coordinator	27	550000	Admin	3	4.0	555-222-1111
7	James	Miller	Analyst	29	70000	HR	6	4.6	555-999-0000
8	Emily	Anderson	Supervisor	31	68000	Marketing	7	4.4	555-999-0000

SELECT * FROM Employees
WHERE department IN ('HR', 'Finance', 'Admin');

employee_id	first_name	last_name	job_title	age	salary	department	experience	rating	phone_number
1	John	Doe	Manager	35	80000	Sales	7	4.5	555-123-4567
2	Jane	Smith	Supervisor	28	60000	HR	5	4.2	555-987-6543
3	Robert	Johnson	Analyst	42	75000	Finance	10	4.7	NULL
4	Jennifer	Williams	Manager	38	90000	Marketing	12	4.9	555-444-3333
5	Michael	Brown	Supervisor	32	650000	Sales	8	4.3	555-777-8888
6	Laura	Davis	Coordinator	27	550000	Admin	3	4.0	555-222-1111
7	James	Miller	Analyst	29	70000	HR	6	4.6	555-999-0000
8	Emily	Anderson	Supervisor	31	68000	Marketing	7	4.4	555-999-0000

SELECT * FROM Employees
WHERE phone_number IS NULL;

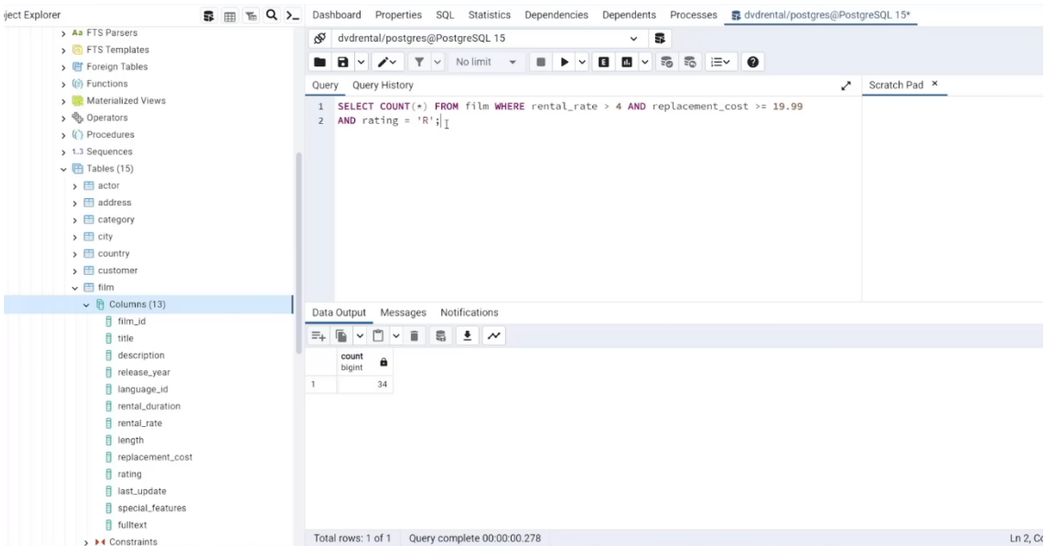
employee_id	first_name	last_name	job_title	age	salary	department	experience	rating	phone_number
1	John	Doe	Manager	35	80000	Sales	7	4.5	555-123-4567
2	Jane	Smith	Supervisor	28	60000	HR	5	4.2	555-987-6543
3	Robert	Johnson	Analyst	42	75000	Finance	10	4.7	NULL
4	Jennifer	Williams	Manager	38	90000	Marketing	12	4.9	555-444-3333
5	Michael	Brown	Supervisor	32	650000	Sales	8	4.3	555-777-8888
6	Laura	Davis	Coordinator	27	550000	Admin	3	4.0	555-222-1111
7	James	Miller	Analyst	29	70000	HR	6	4.6	555-999-0000
8	Emily	Anderson	Supervisor	31	68000	Marketing	7	4.4	555-999-0000

SELECT * FROM Employees
WHERE department = 'HR' AND age >= 30;

employee_id	first_name	last_name	job_title	age	salary	department	experience	rating	phone_number
1	John	Doe	Manager	35	80000	Sales	7	4.5	555-123-4567
2	Jane	Smith	Supervisor	28	60000	HR	5	4.2	555-987-6543
3	Robert	Johnson	Analyst	42	75000	Finance	10	4.7	NULL
4	Jennifer	Williams	Manager	38	90000	Marketing	12	4.9	555-444-3333
5	Michael	Brown	Supervisor	32	650000	Sales	8	4.3	555-777-8888
6	Laura	Davis	Coordinator	27	550000	Admin	3	4.0	555-222-1111
7	James	Miller	Analyst	29	70000	HR	6	4.6	555-999-0000
8	Emily	Anderson	Supervisor	31	68000	Marketing	7	4.4	555-999-0000

SELECT * FROM Employees
WHERE department = 'Finance' OR department = 'Marketing';

employee_id	first_name	last_name	job_title	age	salary	department	experience	rating	phone_number
1	John	Doe	Manager	35	80000	Sales	7	4.5	555-123-4567
2	Jane	Smith	Supervisor	28	60000	HR	5	4.2	555-987-6543
3	Robert	Johnson	Analyst	42	75000	Finance	10	4.7	NULL
4	Jennifer	Williams	Manager	38	90000	Marketing	12	4.9	555-444-3333
5	Michael	Brown	Supervisor	32	650000	Sales	8	4.3	555-777-8888
6	Laura	Davis	Coordinator	27	550000	Admin	3	4.0	555-222-1111
7	James	Miller	Analyst	29	70000	HR	6	4.6	555-999-0000
8	Emily	Anderson	Supervisor	31	68000	Marketing	7	4.4	555-999-0000



ORDER BY

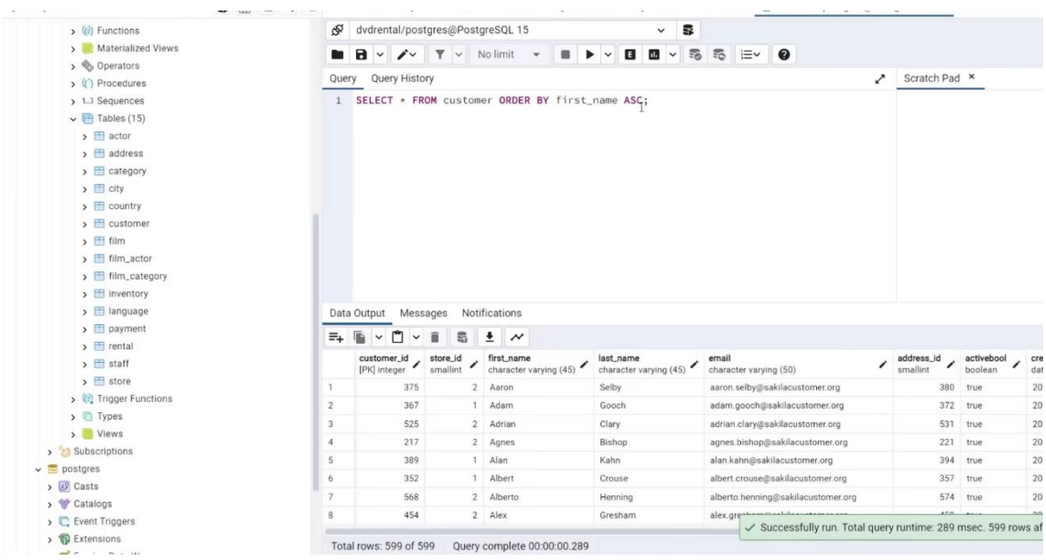
The **ORDER BY** clause in PostgreSQL is used to sort the result set of a query in a specified order. It is used when you want the query's output to be presented in a particular sequence based on one or more columns' values. This can be crucial for making query results more meaningful and easier to interpret.

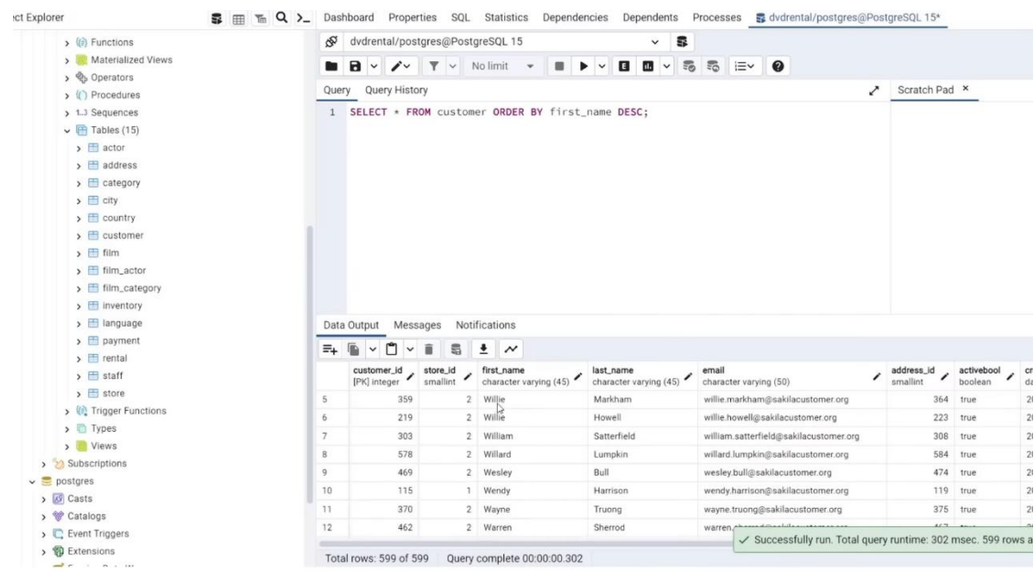
Usage of **ORDER BY:**
The syntax of the **ORDER BY** clause is as follows:

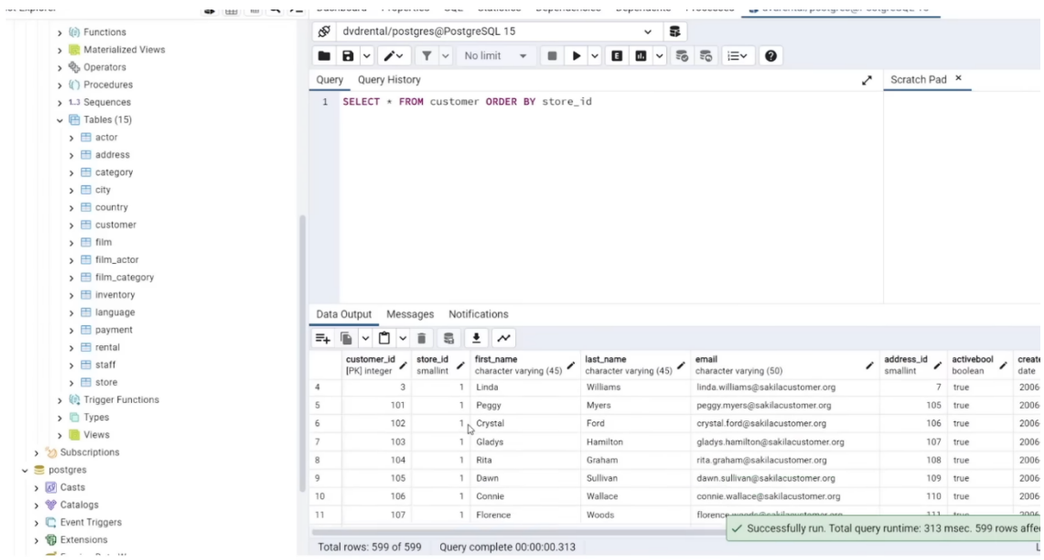
SELECT column1, column2, ...
FROM table_name
WHERE conditions
ORDER BY column1 [**ASC** | **DESC**], column2 [**ASC** | **DESC**], ...;

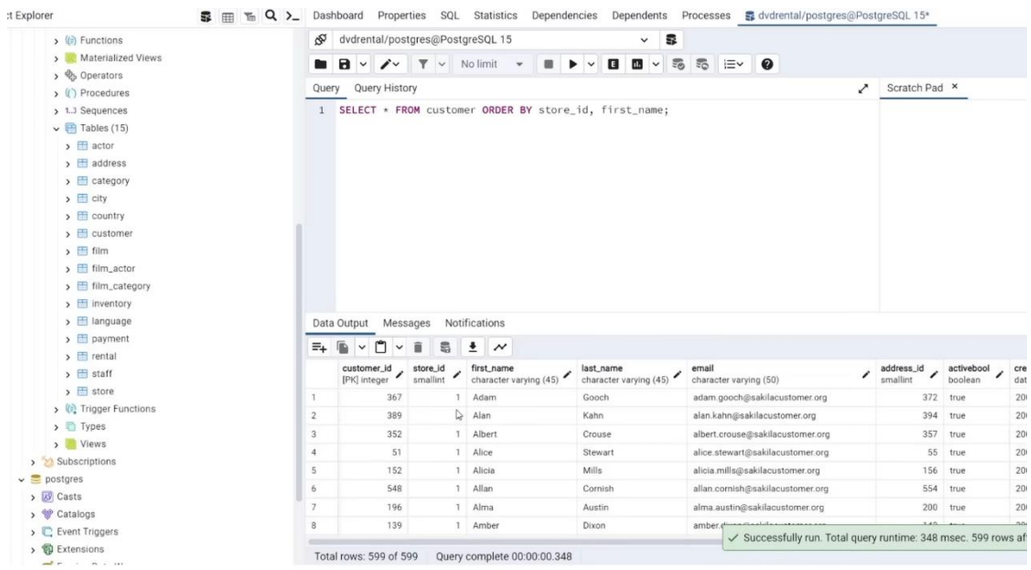
Here's what each component does:

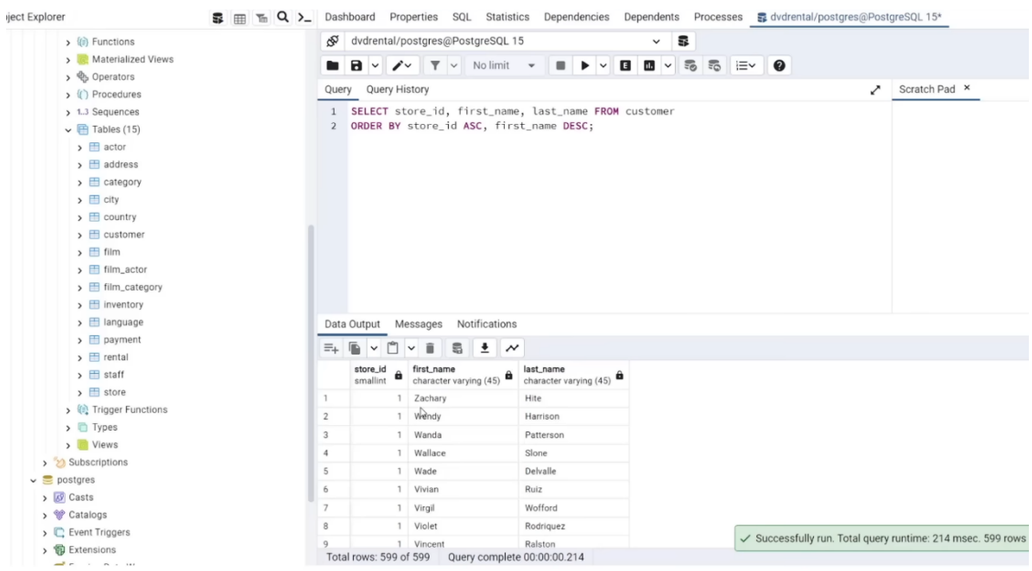
- **SELECT:** Specifies the columns you want to retrieve from the table.
- **FROM:** Specifies the table from which you're retrieving data.
- **WHERE:** Optional condition to filter rows before sorting.
- **ORDER BY:** Specifies the columns by which the result set should be sorted. You can sort by one or more columns, and for each column, you can specify ascending (ASC, which is the default) or descending (DESC) order.




Dashboard
Properties
SQL
Statistics
Dependencies
Dependents
Processes
dvdrental/postgres@PostgreSQL 15*
dvdrental/postgres@PostgreSQL 15
No limit
Query
Query History
Scratch Pad
1 SELECT * FROM customer ORDER BY first_name DESC;
Data Output
Messages
Notifications
customer_id [PK] integer
store_id smallint
first_name character varying (45)
last_name character varying (45)
email character varying (50)
address_id smallint
activebool boolean
5 359 2 Willie Markham willie.markham@sakilacustomer.org 364 true
6 219 2 Willie Howell willie.howell@sakilacustomer.org 223 true
7 303 2 William Satterfield william.satterfield@sakilacustomer.org 308 true
8 578 2 Willard Lumpkin willard.lumpkin@sakilacustomer.org 584 true
9 469 2 Wesley Bull wesley.bull@sakilacustomer.org 474 true
10 115 1 Wendy Harrison wendy.harrison@sakilacustomer.org 119 true
11 370 2 Wayne Truong wayne.truong@sakilacustomer.org 375 true
12 462 2 Warren Sherrod
Total rows: 599 of 599
Query complete 00:00:00.302
Successfully run. Total query runtime: 302 msec. 599 rows a
Functions
Materialized Views
Operators
Procedures
Sequences
Tables (15)
actor
address
category
city
country
customer
film
film_actor
film_category
inventory
language
payment
rental
staff
store
Trigger Functions
Types
Views
Subscriptions
postgres
Casts
Catalogs
Event Triggers
Extensions


dvdrental/postgres@PostgreSQL 15
No limit
Query
Query History
Scratch Pad
1 SELECT * FROM customer ORDER BY store_id
Data Output
Messages
Notifications
customer_id [PK] integer
store_id smallint
first_name character varying (45)
last_name character varying (45)
email character varying (50)
address_id smallint
activebool boolean
4 3 1 Linda Williams linda.williams@sakilacustomer.org 7 true
5 101 1 Peggy Myers peggy.myers@sakilacustomer.org 105 true
6 102 1 Crystal Ford crystal.ford@sakilacustomer.org 106 true
7 103 1 Gladys Hamilton gladys.hamilton@sakilacustomer.org 107 true
8 104 1 Rita Graham rita.graham@sakilacustomer.org 108 true
9 105 1 Dawn Sullivan dawn.sullivan@sakilacustomer.org 109 true
10 106 1 Connie Wallace connie.wallace@sakilacustomer.org 110 true
11 107 1 Florence Woods
Total rows: 599 of 599
Query complete 00:00:00.313
Successfully run. Total query runtime: 313 msec. 599 rows affe
Functions
Materialized Views
Operators
Procedures
Sequences
Tables (15)
actor
address
category
city
country
customer
film
film_actor
film_category
inventory
language
payment
rental
staff
store
Trigger Functions
Types
Views
Subscriptions
postgres
Casts
Catalogs
Event Triggers
Extensions


Dashboard
Properties
SQL
Statistics
Dependencies
Dependents
Processes
dvdrental/postgres@PostgreSQL 15*
dvdrental/postgres@PostgreSQL 15
No limit
Query
Query History
Scratch Pad
SELECT * FROM customer ORDER BY store_id, first_name;
Data Output
Messages
Notifications
customer_id [PK] integer
store_id smallint
first_name character varying (45)
last_name character varying (45)
email character varying (50)
address_id smallint
activebool boolean
Adam Gooch adam.gooch@sakilacustomer.org
Alan Kahn alan.kahn@sakilacustomer.org
Albert Crouse albert.crouse@sakilacustomer.org
Alice Stewart alice.stewart@sakilacustomer.org
Alicia Mills alicia.mills@sakilacustomer.org
Allan Cornish allan.cornish@sakilacustomer.org
Alma Austin alma.austin@sakilacustomer.org
Amber Dixon
Successfully run. Total query runtime: 348 msec. 599 rows af
Total rows: 599 of 599
Query complete 00:00:00.348
Functions
Materialized Views
Operators
Procedures
Sequences
Tables (15)
actor
address
category
city
country
customer
film
film_actor
film_category
inventory
language
payment
rental
staff
store
Trigger Functions
Types
Views
Subscriptions
postgres
Casts
Catalogs
Event Triggers
Extensions


Dashboard
Properties
SQL
Statistics
Dependencies
Dependents
Processes
dvdrental/postgres@PostgreSQL 15*
dvdrental/postgres@PostgreSQL 15
No limit
Query
Query History
Scratch Pad
SELECT store_id, first_name, last_name FROM customer
ORDER BY store_id ASC, first_name DESC;
Data Output
Messages
Notifications
store_id smallint
first_name character varying (45)
last_name character varying (45)
Zachary Hite
Wendy Harrison
Wanda Patterson
Wallace Slone
Wade Delvalle
Vivian Ruiz
Virgil Wofford
Violet Rodriquez
Vincent Ralston
Successfully run. Total query runtime: 214 msec. 599 rows
Total rows: 599 of 599
Query complete 00:00:00.214
Functions
Materialized Views
Operators
Procedures
Sequences
Tables (15)
actor
address
category
city
country
customer
film
film_actor
film_category
inventory
language
payment
rental
staff
store
Trigger Functions
Types
Views
Subscriptions
postgres
Casts
Catalogs
Event Triggers
Extensions

LIMIT

The **LIMIT** command in PostgreSQL is used to restrict the number of rows returned by a query. It allows you to specify a maximum number of rows to be included in the result set. This can be particularly useful when you have a large dataset and only need to retrieve a subset of the data for analysis or display purposes.

Here's how the **LIMIT** command works:

```
SELECT column1, column2, ...
FROM table_name
LIMIT number_of_rows;
```

In this SQL query:

- **column1, column2, etc.:** The columns you want to retrieve data from.
- **table_name:** The name of the table you want to query.
- **number_of_rows:** The maximum number of rows you want to retrieve.

For example, if you have a table named employees and you want to retrieve the first 10 employees, you would use the following query:

```
SELECT first_name, last_name
FROM employees
LIMIT 10;
```

BETWEEN

• The **BETWEEN** command in PostgreSQL is used to filter data within a specified range of values. It is typically used in conjunction with the **WHERE** clause to retrieve rows that fall within a specified range of values for a particular column.

Here's a breakdown of its usage and significance:

1. Syntax: The basic syntax of the **BETWEEN** command is as follows:

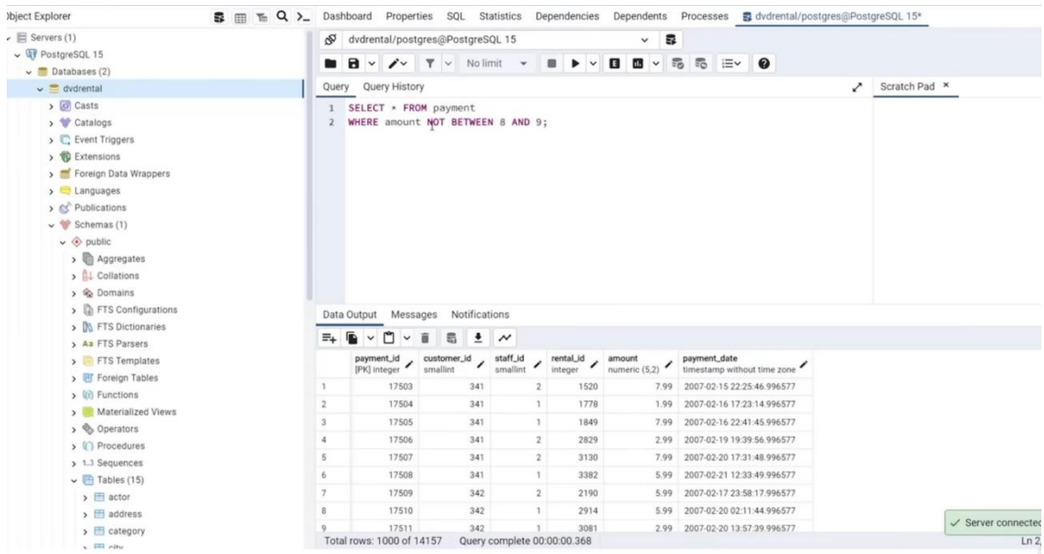
```
SELECT column_name(s)
FROM table_name
WHERE column_name BETWEEN
value1 AND value2;
```

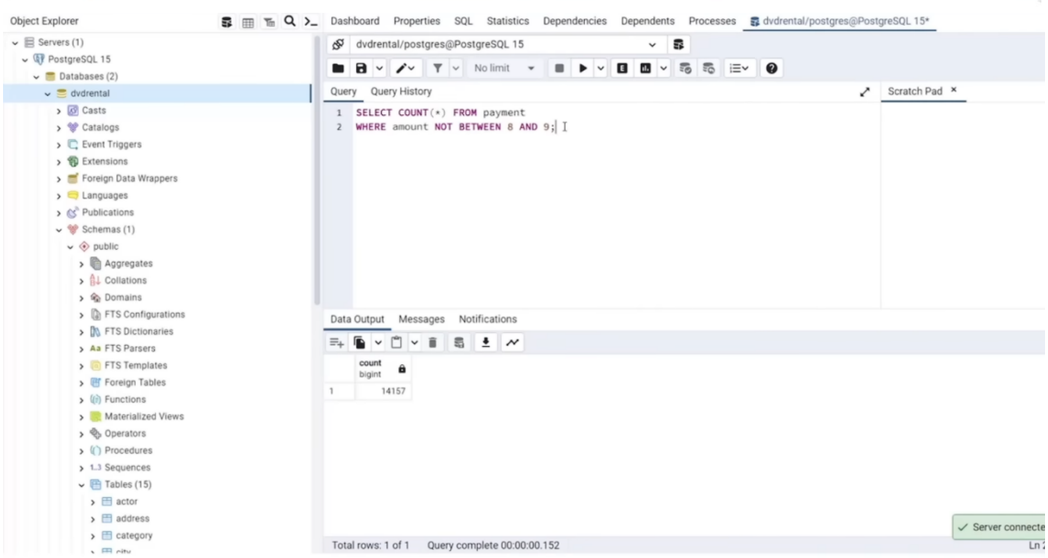
Performance: When using the BETWEEN command, make sure the column you're applying it to has an appropriate index. Indexing the column can significantly improve query performance, as PostgreSQL can quickly narrow down the rows within the specified range.

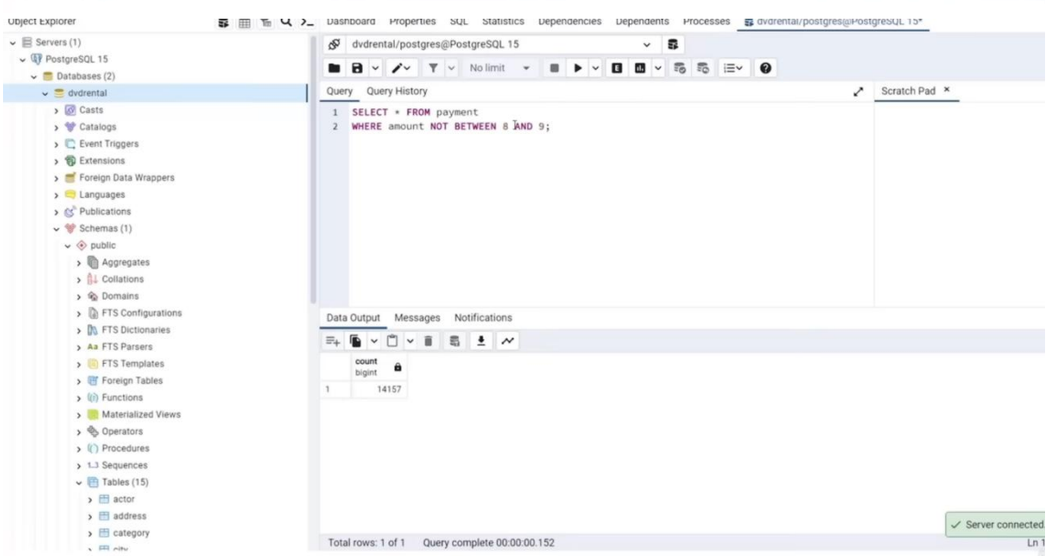
Examples:

• Retrieving orders made within a specific date range:

```
SELECT order_id, order_date
FROM orders
WHERE order_date BETWEEN '2023-01-01' AND '2023-06-30
```


Object Explorer
Servers (1)
PostgreSQL 15
Databases (2)
dvdrental
Casts
Catalogs
Event Triggers
Extensions
Foreign Data Wrappers
Languages
Publications
Schemas (1)
public
Aggregates
Collations
Domains
FTS Configurations
FTS Dictionaries
FTS Parsers
FTS Templates
Foreign Tables
Functions
Materialized Views
Operators
Procedures
Sequences
Tables (15)
actor
address
category
Dashboard
Properties
SQL
Statistics
Dependencies
Dependents
Processes
dvdrental/postgres@PostgreSQL 15*
dvdrental/postgres@PostgreSQL 15
No limit
Query
Query History
Scratch Pad
SELECT * FROM payment
WHERE amount NOT BETWEEN 8 AND 9;
Data Output
Messages
Notifications
payment_id [PK] integer
customer_id smallint
staff_id smallint
rental_id integer
amount numeric (5,2)
payment_date timestamp without time zone
1 17503 341 2 1520 7.99 2007-02-15 22:25:46.996577
2 17504 341 1 1778 1.99 2007-02-16 17:23:14.996577
3 17505 341 1 1849 7.99 2007-02-16 22:41:45.996577
4 17506 341 2 2829 2.99 2007-02-19 19:39:56.996577
5 17507 341 2 3130 7.99 2007-02-20 17:31:48.996577
6 17508 341 1 3382 5.99 2007-02-21 12:33:49.996577
7 17509 342 2 2190 5.99 2007-02-17 23:58:17.996577
8 17510 342 1 2914 5.99 2007-02-20 02:11:44.996577
9 17511 342 1 3081 2.99 2007-02-20 13:57:39.996577
Total rows: 1000 of 14157
Query complete 00:00:00.368
Server connected
Ln 2


Object Explorer
Servers (1)
PostgreSQL 15
Databases (2)
dvdrental
Casts
Catalogs
Event Triggers
Extensions
Foreign Data Wrappers
Languages
Publications
Schemas (1)
public
Aggregates
Collations
Domains
FTS Configurations
FTS Dictionaries
FTS Parsers
FTS Templates
Foreign Tables
Functions
Materialized Views
Operators
Procedures
Sequences
Tables (15)
actor
address
category
Dashboard
Properties
SQL
Statistics
Dependencies
Dependents
Processes
dvdrental/postgres@PostgreSQL 15*
dvdrental/postgres@PostgreSQL 15
No limit
Query
Query History
Scratch Pad
SELECT COUNT(*) FROM payment
WHERE amount NOT BETWEEN 8 AND 9;
Data Output
Messages
Notifications
count bigint
1 14157
Total rows: 1 of 1
Query complete 00:00:00.152
Server connecte
Ln 2


Object Explorer
Servers (1)
PostgreSQL 15
Databases (2)
dvdrental
Casts
Catalogs
Event Triggers
Extensions
Foreign Data Wrappers
Languages
Publications
Schemas (1)
public
Aggregates
Collations
Domains
FTS Configurations
FTS Dictionaries
FTS Parsers
FTS Templates
Foreign Tables
Functions
Materialized Views
Operators
Procedures
Sequences
Tables (15)
actor
address
category
dvdrental/postgres@PostgreSQL 15
No limit
Query
Query History
Scratch Pad
SELECT * FROM payment
WHERE amount NOT BETWEEN 8 AND 9;
Data Output
Messages
Notifications
count bigint
1 14157
Total rows: 1 of 1
Query complete 00:00:00.152
Server connected

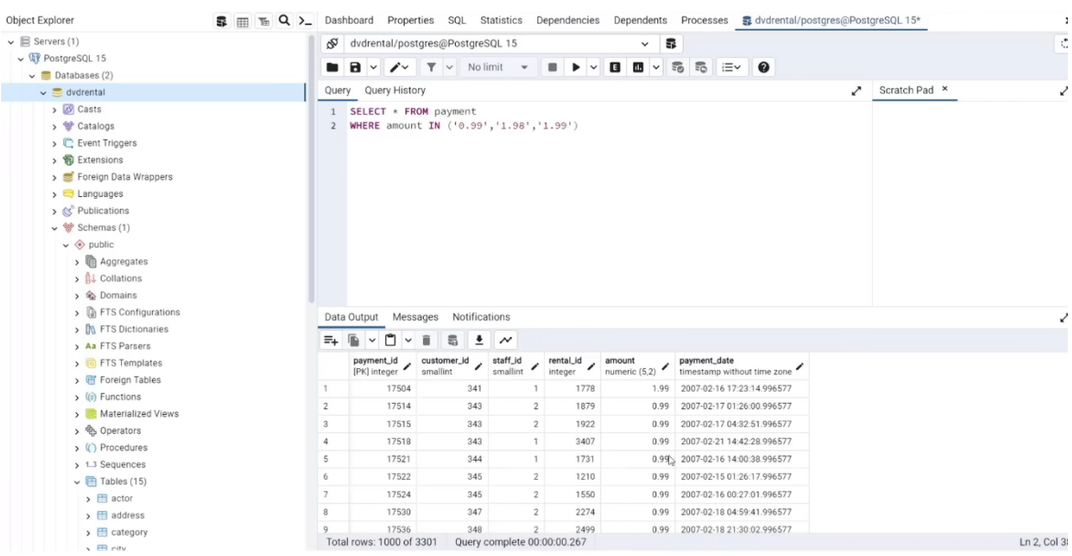
IN

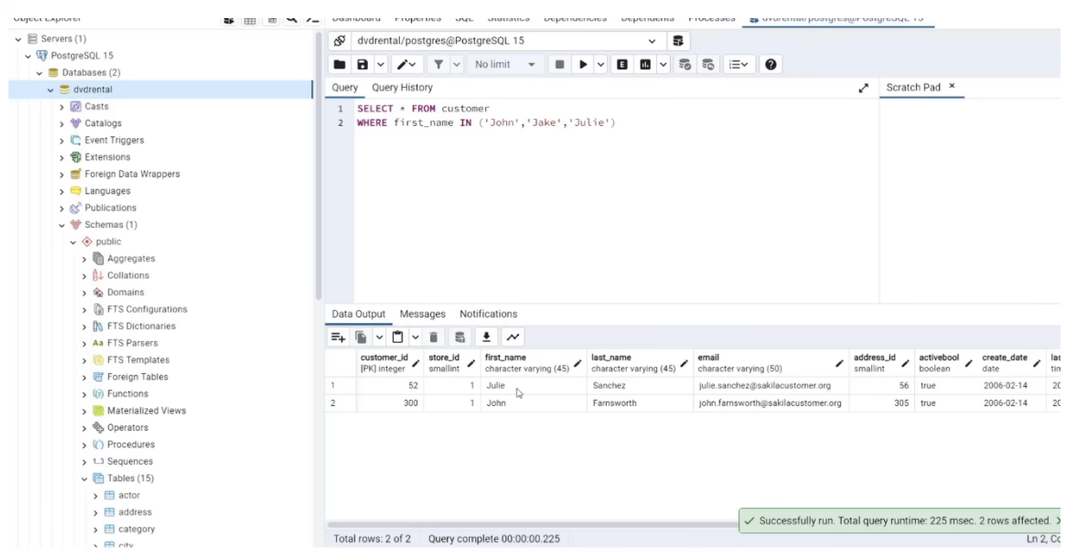
The **IN** command in PostgreSQL is used to filter data based on a specified list of values for a particular column. It allows you to retrieve rows where a specified column's value matches any of the values in the given list.

Here's a detailed explanation of the **IN** command and its significance:

1. Syntax: The basic syntax of the IN command is as follows:

```
SELECT column_name(s)
FROM table_name
WHERE column_name IN (value1, value2, ...);
```





LIKE AND ILIKE

The **LIKE** and **ILIKE** commands in PostgreSQL are used for pattern matching within text columns. They allow you to filter rows based on patterns defined using wildcard characters. The main difference between the two commands is that **LIKE** performs case-sensitive matching, while **ILIKE** performs case-insensitive matching.

Here's a detailed explanation of the **LIKE** and **ILIKE** commands and their significance:

1. Syntax:

- The basic syntax of the LIKE command is:

```
SELECT column_name(s)
FROM table_name
WHERE column_name LIKE pattern;
```

- The basic syntax of the **ILIKE** command is:

```
SELECT column_name(s)
FROM table_name
WHERE column_name ILIKE  pattern;
```

2. **Wildcard Characters:** Wildcards are special characters used to represent one or more characters in a pattern.

- **'%':** Represents any sequence of characters (including zero characters).
- **' _':** Represents a single character.
- **'[]':** Represents a character class, allowing you to match any single character within the specified set.

4. **Case Sensitivity:**

LIKE: Performs case-sensitive pattern matching. For example, if you use LIKE 'apple%', it will match rows with values like "apple", "applesauce", but not "Apple".
ILIKE: Performs case-insensitive pattern matching. It will match rows regardless of the case. For example, ILIKE 'apple%' would match "apple", "Applesauce", and "APPLEJUICE

Examples:

- Retrieving all customers with names starting with "J":

```
SELECT customer_name, contact_email
FROM customers
WHERE customer_name LIKE 'J%';
```

- Finding products with names containing "blue":

```
SELECT product_name, category
FROM products
WHERE product_name LIKE '%blue%'
```

• Searching for employees with last names ending in "son" (case-insensitive):

```
SELECT first_name, last_name
FROM employees
WHERE last_name ILIKE '%son';
```

Wildcard characters are special symbols used in pattern matching within SQL queries. They are extremely useful for performing flexible searches and filtering based on specific patterns within text columns. Here's an explanation of the common wildcard characters, their benefits, and when to use them:

% (Percent Sign):

Represents any sequence of characters (including zero characters).
Use cases:
Searching for values that start with or end with a specific sequence of characters.
Finding records that contain a particular substring within a larger string.
Example:

```
SELECT product_name
FROM products
WHERE product_name LIKE '%apple%';
```

This query retrieves all products with names containing the word "apple".

_ (Underscore):

- Represents a single character.
- Use cases:
  - Searching for values with a specific character at a certain position.
  - Finding records with a particular pattern where you know the character at a specific position.

Example:

```
SELECT customer_name
FROM customers
WHERE customer_name LIKE 'J_n%';
```

This query retrieves customers with names that start with "J" and have any second character followed by "n".

[] (Square Brackets):

• Represents a character class, allowing you to match any single character within the specified set.
• Use cases:
• Searching for values that can have variations in a specific character position (e.g., for alternate spellings or formats).
Example:

```
SELECT product_name
FROM products
WHERE product_name LIKE 'colo[u]r%';
```

Example:

```
SELECT product_name
FROM products
WHERE product_name LIKE 'colo[u]r%';
```

This query matches products with names starting with "color" and accounts for variations in spelling, such as "colour" in British English.

Combining the percent sign (%), underscore (_), and square brackets ([]) allows you to create even more complex and specific pattern-based searches within SQL queries. This combination is particularly useful when you need to search for data with a specific structure or pattern while accounting for variations. Here's an explanation of how to use this combination, along with a use case:

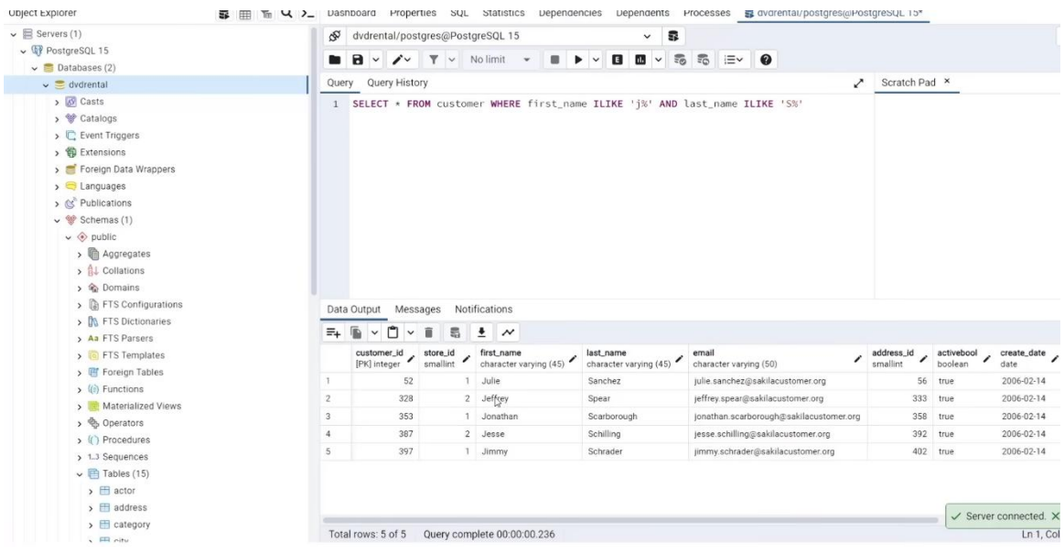
Syntax for Combining Wildcard Characters:
'%': Represents any sequence of characters (including zero characters).
_: Represents a single character.
[]: Represents a character class, allowing you to match any single character within the specified set.

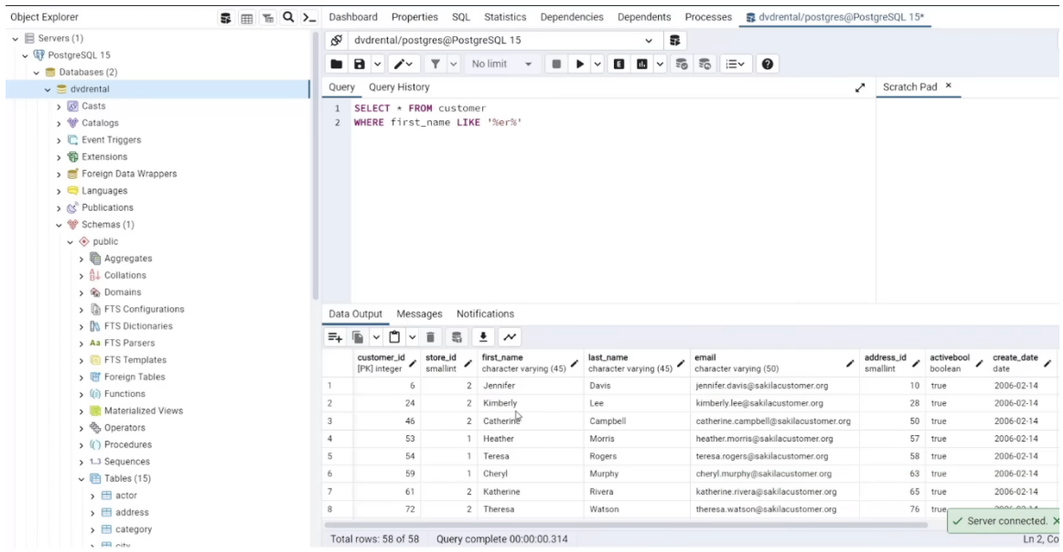
Use Case - Searching for Email Domains:
Suppose you have a database table containing email addresses and you want to retrieve all email addresses from a specific domain, while allowing for variations in the domain name, such as different subdomains or top-level domains (TLDs). Let's say you're interested in retrieving email addresses from both "example.com" and "example.net".
Query:

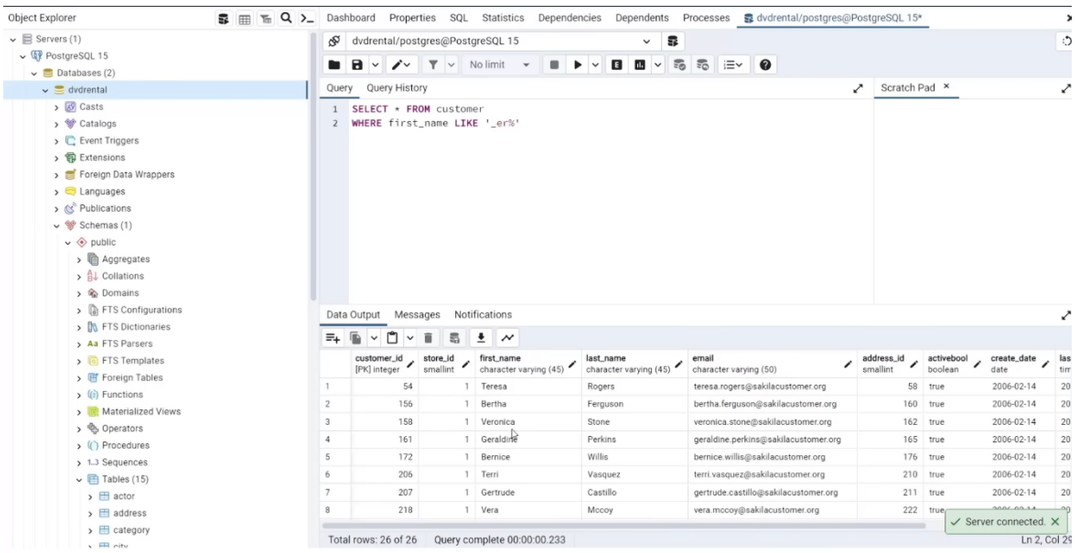
```
SELECT email_address
FROM users
WHERE email_address LIKE '%example[_].[cno]m';
```

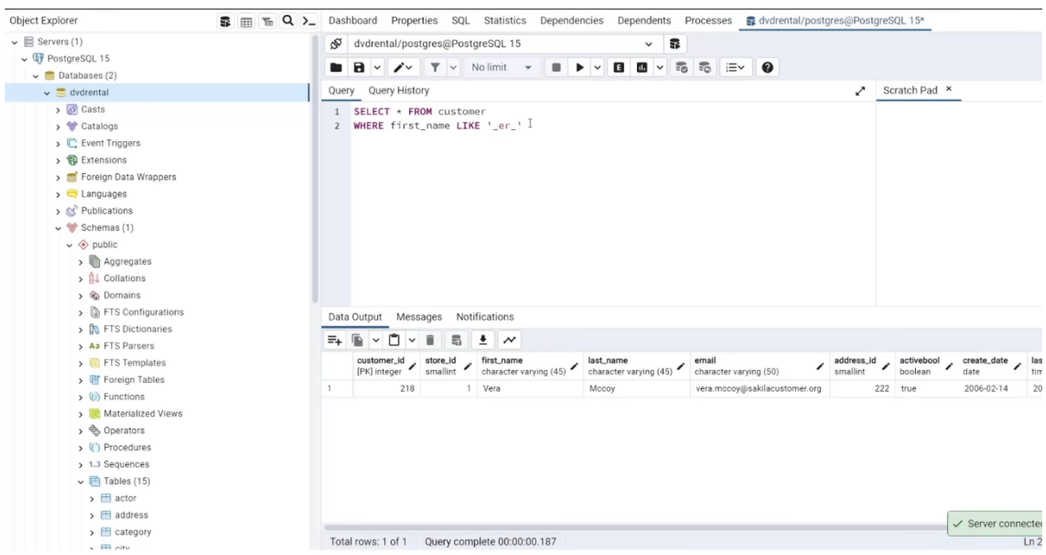
Explanation:

'%'example matches any email address that ends with "example".
'[_]' matches a single character, which can be any character (including a dot) in the email address.
'.[cno]' matches a single character from the character set [cno], which covers variations of the TLD: "c", "n", or "o" (for .com, .net, or .org).
Results:
This query would retrieve email addresses like "user@example.com", "admin@example.net", and any other variations that match the specified pattern.




Object Explorer
Dashboard Properties SQL Statistics Dependencies Dependents Processes dvdrental/postgres@PostgreSQL 15*
dvdrental/postgres@PostgreSQL 15
No limit
Query Query History
Scratch Pad
SELECT * FROM customer
WHERE first_name LIKE '%er%'
Data Output Messages Notifications
customer_id [PK] integer
store_id smallint
first_name character varying (45)
last_name character varying (45)
email character varying (50)
address_id smallint
activebool boolean
create_date date
1 6 2 Jennifer Davis jennifer.davis@sakilacustomer.org 10 true 2006-02-14
2 24 2 Kimberly Lee kimberly.lee@sakilacustomer.org 28 true 2006-02-14
3 46 2 Catherine Campbell catherine.campbell@sakilacustomer.org 50 true 2006-02-14
4 53 1 Heather Morris heather.morris@sakilacustomer.org 57 true 2006-02-14
5 54 1 Teresa Rogers teresa.rogers@sakilacustomer.org 58 true 2006-02-14
6 59 1 Cheryl Murphy cheryl.murphy@sakilacustomer.org 63 true 2006-02-14
7 61 2 Katherine Rivera katherine.rivera@sakilacustomer.org 65 true 2006-02-14
8 72 2 Theresa Watson theresa.watson@sakilacustomer.org 76 true
Server connected.
Total rows: 58 of 58 Query complete 00:00:00.314
Ln 2, Co


Object Explorer
Dashboard Properties SQL Statistics Dependencies Dependents Processes dvdrental/postgres@PostgreSQL 15*
dvdrental/postgres@PostgreSQL 15
No limit
Query Query History
Scratch Pad
SELECT * FROM customer
WHERE first_name LIKE '_er%'
Data Output Messages Notifications
customer_id [PK] integer
store_id smallint
first_name character varying (45)
last_name character varying (45)
email character varying (50)
address_id smallint
activebool boolean
create_date date
1 54 1 Teresa Rogers teresa.rogers@sakilacustomer.org 58 true 2006-02-14
2 156 1 Bertha Ferguson bertha.ferguson@sakilacustomer.org 160 true 2006-02-14
3 158 1 Veronica Stone veronica.stone@sakilacustomer.org 162 true 2006-02-14
4 161 1 Geraldine Perkins geraldine.perkins@sakilacustomer.org 165 true 2006-02-14
5 172 1 Bernice Willis bernice.willis@sakilacustomer.org 176 true 2006-02-14
6 206 1 Terri Vasquez terri.vasquez@sakilacustomer.org 210 true 2006-02-14
7 207 1 Gertrude Castillo gertrude.castillo@sakilacustomer.org 211 true 2006-02-14
8 218 1 Vera Mccoy vera.mccoy@sakilacustomer.org 222 true
Server connected.
Total rows: 26 of 26 Query complete 00:00:00.233
Ln 2, Col 25


Object Explorer
Dashboard Properties SQL Statistics Dependencies Dependents Processes dvdrental/postgres@PostgreSQL 15*
dvdrental/postgres@PostgreSQL 15
No limit
Query Query History
Scratch Pad
SELECT * FROM customer
WHERE first_name LIKE '_er_'
Data Output Messages Notifications
customer_id [PK] integer
store_id smallint
first_name character varying (45)
last_name character varying (45)
email character varying (50)
address_id smallint
activebool boolean
create_date date
1 218 1 Vera Mccoy vera.mccoy@sakilacustomer.org 222 true 2006-02-14
Total rows: 1 of 1 Query complete 00:00:00.187

AGGREGATE FUNCTIONS

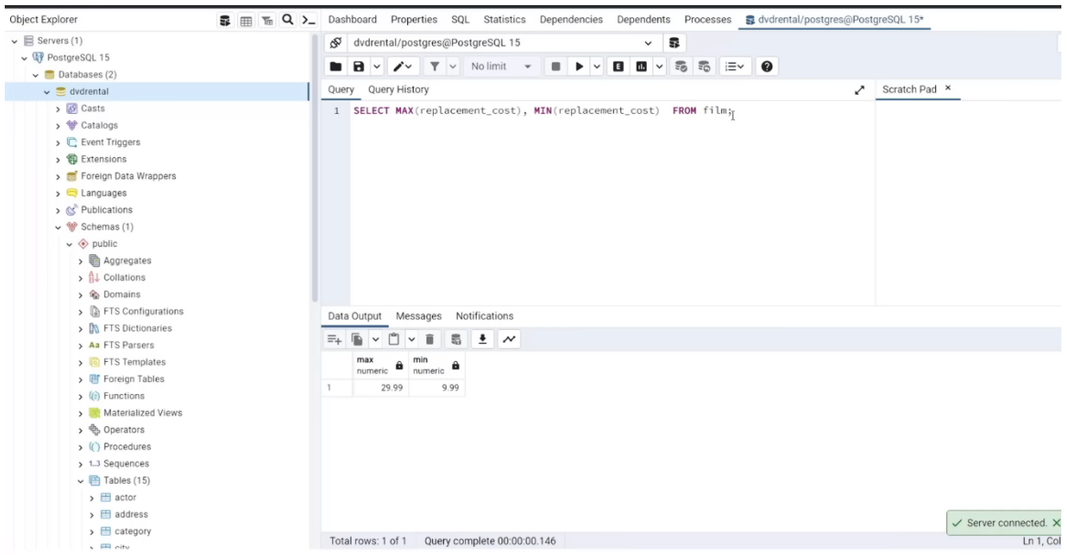
In PostgreSQL, **AGGREGATE** functions are special functions that perform calculations on a set of values and return a single value as a result. These functions are commonly used to summarize or derive meaningful insights from large datasets in a database. Instead of operating on individual rows, aggregate functions work on a group of rows and return a single value for each group. They are crucial for tasks such as calculating averages, sums, counts, minimum and maximum values, and more.

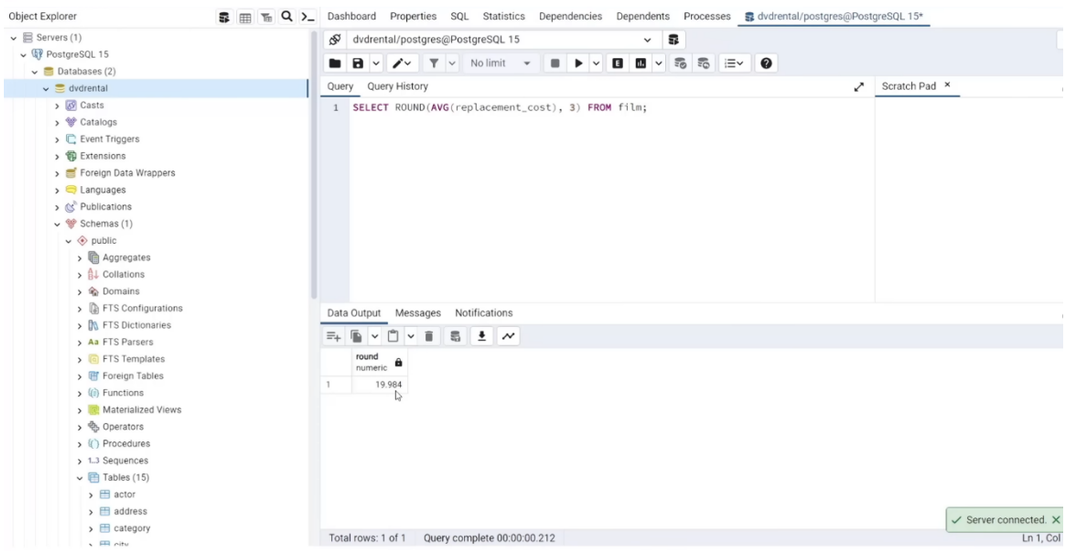
Here are some key aspects of aggregate functions in PostgreSQL:

1. Grouping Rows:
Aggregate functions are often used in conjunction with the **GROUP BY** clause. The **GROUP BY** clause divides the result set into groups based on one or more columns. The aggregate functions then operate on each group separately, producing a single result for each group.

2. Common Aggregate Functions:
PostgreSQL provides a wide range of aggregate functions that cater to various analytical needs. Some of the most common aggregate functions include:

- **SUM:** Calculates the sum of a numeric column within a group.
- **AVG:** Computes the average of a numeric column within a group.
- **COUNT:** Counts the number of rows within a group.
- **MIN:** Returns the minimum value from a column within a group.
- **MAX:** Returns the maximum value from a column within a group.





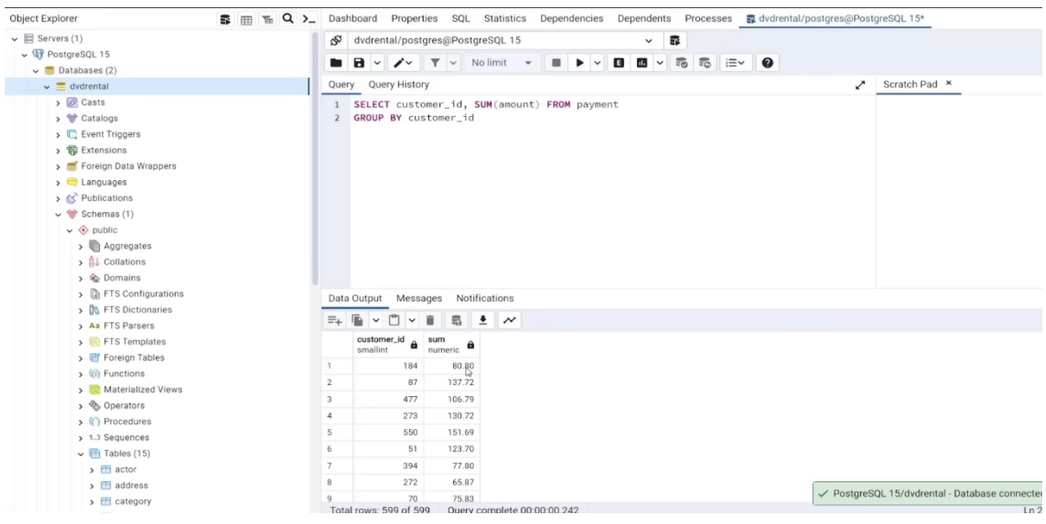
GROUP BY

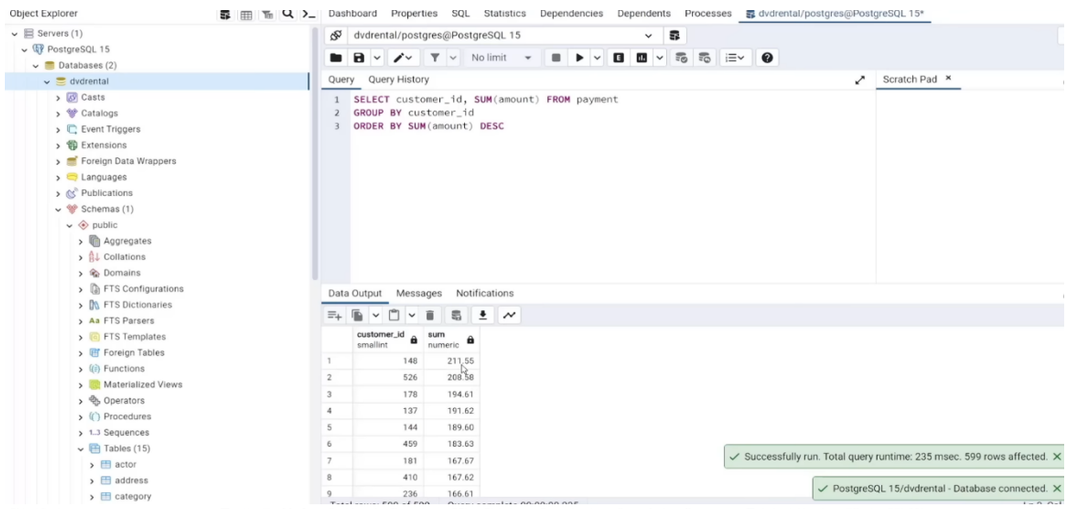
What is **GROUP BY** and Why is it Necessary?

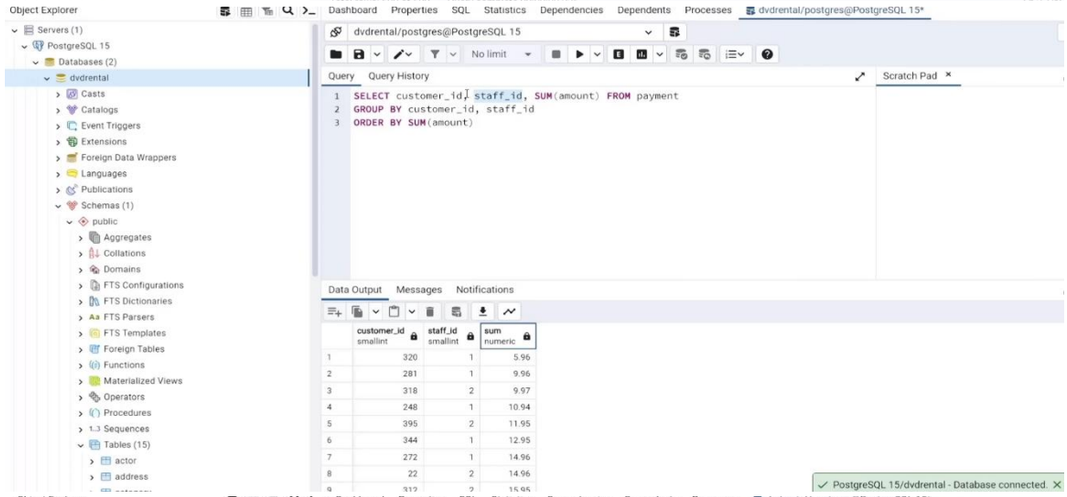
In SQL, the **GROUP BY** clause is used to group rows from a table based on one or more columns. This grouping enables the application of aggregate functions, such as **SUM, COUNT, AVG, MIN,** and **MAX**, on subsets of data sharing common values in the specified columns. Essentially, it transforms individual rows into summarized information, aiding in the analysis of large datasets and revealing trends and insights that might not be immediately apparent.

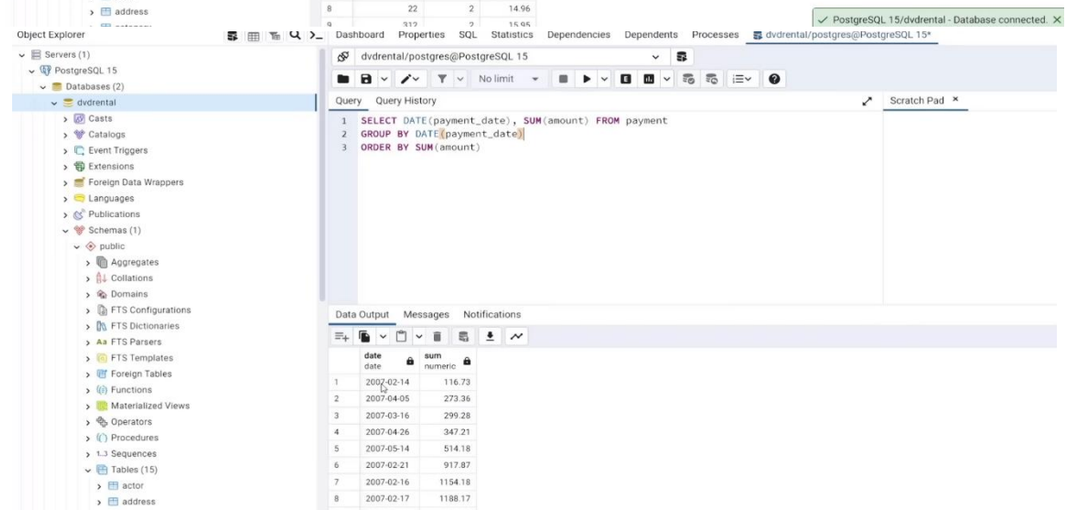
The **GROUP BY** operation follows a **Split, Apply, Combine** process. Let's consider a hypothetical sales table with columns "Region," "Product," and "SalesAmount." To analyze total sales for each region, we apply the following steps:

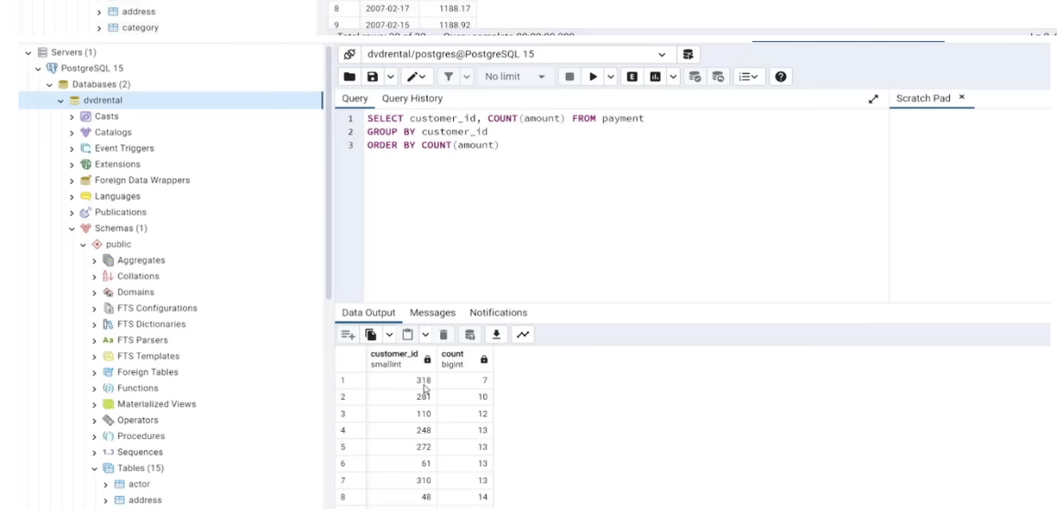
Split: The dataset is divided into groups based on unique values in the "Region" column.
Apply: Aggregate functions, such as SUM(SalesAmount), are applied to the "SalesAmount" column within each region group.
Combine: The aggregated results are combined, creating a summary table that displays the total sales for each region.




Object Explorer
Dashboard Properties SQL Statistics Dependencies Dependents Processes dvdrental/postgres@PostgreSQL 15*
dvdrental/postgres@PostgreSQL 15
No limit
Query Query History
Scratch Pad
1 SELECT customer_id, SUM(amount) FROM payment
2 GROUP BY customer_id
3 ORDER BY SUM(amount) DESC
Data Output Messages Notifications
customer_id smallint
sum numeric
1 148 211.55
2 526 208.58
3 178 194.61
4 137 191.62
5 144 189.60
6 459 183.63
7 181 167.67
8 410 167.62
9 236 166.61
Successfully run. Total query runtime: 235 msec. 599 rows affected.
PostgreSQL 15/dvdrental - Database connected.


Object Explorer
Dashboard Properties SQL Statistics Dependencies Dependents Processes dvdrental/postgres@PostgreSQL 15*
dvdrental/postgres@PostgreSQL 15
No limit
Query Query History
Scratch Pad
1 SELECT customer_id, staff_id, SUM(amount) FROM payment
2 GROUP BY customer_id, staff_id
3 ORDER BY SUM(amount)
Data Output Messages Notifications
customer_id smallint
staff_id smallint
sum numeric
1 320 1 5.96
2 281 1 9.96
3 318 2 9.97
4 248 1 10.94
5 395 2 11.95
6 344 1 12.95
7 272 1 14.96
8 22 2 14.96
PostgreSQL 15/dvdrental - Database connected.


Object Explorer
Dashboard Properties SQL Statistics Dependencies Dependents Processes dvdrental/postgres@PostgreSQL 15*
dvdrental/postgres@PostgreSQL 15
No limit
Query Query History
Scratch Pad
1 SELECT DATE(payment_date), SUM(amount) FROM payment
2 GROUP BY DATE(payment_date)
3 ORDER BY SUM(amount)
Data Output Messages Notifications
date date
sum numeric
1 2007-02-14 116.73
2 2007-04-05 273.36
3 2007-03-16 299.28
4 2007-04-26 347.21
5 2007-05-14 514.18
6 2007-02-21 917.87
7 2007-02-16 1154.18
8 2007-02-17 1188.17
9 2007-02-15 1188.92


dvdrental/postgres@PostgreSQL 15
No limit
Query Query History
Scratch Pad
1 SELECT customer_id, COUNT(amount) FROM payment
2 GROUP BY customer_id
3 ORDER BY COUNT(amount)
Data Output Messages Notifications
customer_id smallint
count bigint
1 318 7
2 281 10
3 110 12
4 248 13
5 272 13
6 61 13
7 310 13
8 48 14

HAVING

The **HAVING** command in PostgreSQL is used in conjunction with the **GROUP BY** clause to filter the results of a query based on the aggregated values of a column or columns. It allows you to apply conditions to the result set after grouping has been performed. In other words, while the **WHERE** clause filters individual rows before they are grouped, the **HAVING** clause filters groups of rows after they have been grouped.

Here's a breakdown of its key aspects:

1. Purpose: The **HAVING** clause is used to filter rows in the result set after grouping and aggregation, specifically when you want to filter groups based on aggregated values.

2. Usage: It follows the **GROUP BY** clause and comes after it in a SQL query. The basic syntax is:

SELECT column1, aggregate_function(column2)
FROM table
GROUP BY column1
HAVING condition;

3. Need for **HAVING**: While the **WHERE** clause filters individual rows before grouping, the **HAVING** clause filters groups of rows based on the aggregated results. This is especially useful when you want to filter out groups that meet certain criteria.

4. Scenario to use: You would use **HAVING** when you need to filter out groups based on aggregate functions like **COUNT, SUM, AVG,** etc. For example, finding the average salary of departments with an average salary greater than a certain value.

5. Comparison with **WHERE**: Use **WHERE** to filter individual rows before grouping and **HAVING** to filter groups after aggregation.

6. Avoid unnecessary complexity: While **HAVING** is powerful, avoid overusing it or making it too complex. Sometimes, it might be more efficient to use subqueries or other methods to achieve the desired result.

Performance Consideration: When using **HAVING**, consider the performance impact, especially when dealing with large datasets. Improper use of **HAVING** can lead to slower query execution times.

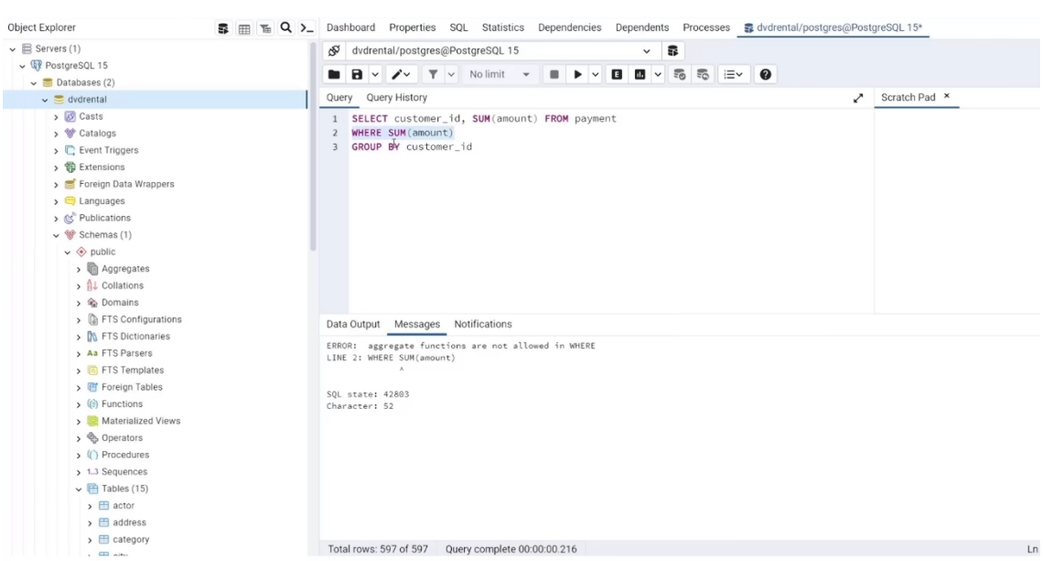
Examples:

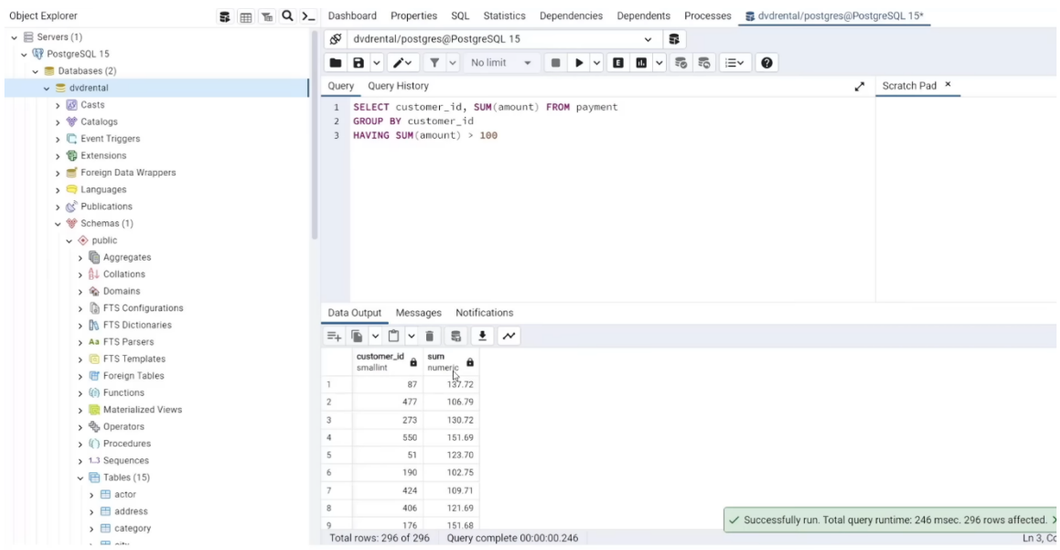
To find departments with more than 10 employees:

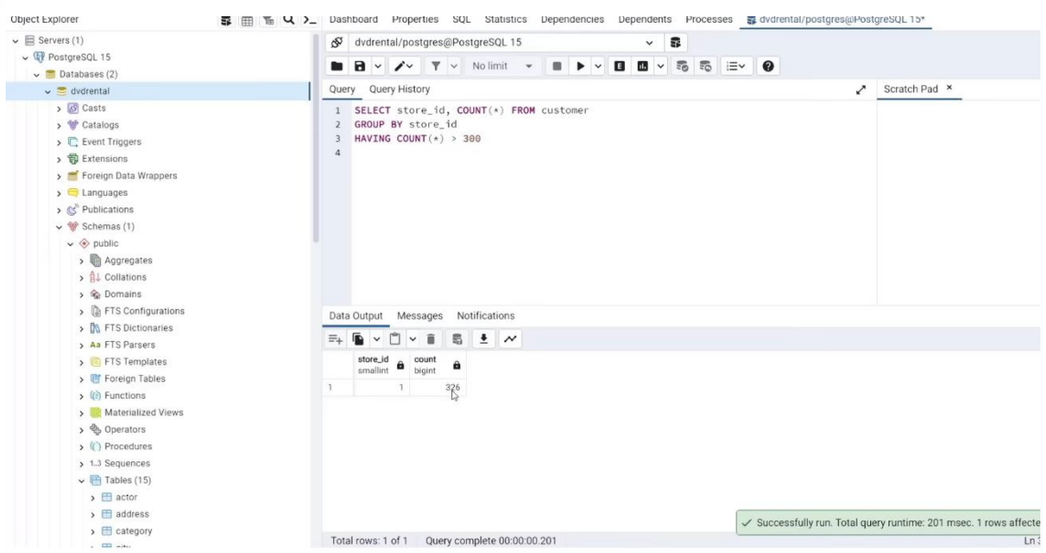
SELECT department, **COUNT(*) FROM** employees
GROUP BY department
HAVING COUNT(*) > 10;

• To find the average salary of departments with an average salary above $50000:

SELECT department, **AVG(salary) FROM** employees
GROUP BY department
HAVING AVG(salary) > 50000;




Object Explorer
Servers (1)
PostgreSQL 15
Databases (2)
dvdrental
Casts
Catalogs
Event Triggers
Extensions
Foreign Data Wrappers
Languages
Publications
Schemas (1)
public
Aggregates
Collations
Domains
FTS Configurations
FTS Dictionaries
FTS Parsers
FTS Templates
Foreign Tables
Functions
Materialized Views
Operators
Procedures
Sequences
Tables (15)
actor
address
category
Dashboard
Properties
SQL
Statistics
Dependencies
Dependents
Processes
dvdrental/postgres@PostgreSQL 15*
dvdrental/postgres@PostgreSQL 15
No limit
Query
Query History
Scratch Pad
SELECT customer_id, SUM(amount) FROM payment
GROUP BY customer_id
HAVING SUM(amount) > 100
Data Output
Messages
Notifications
customer_id
smallint
sum
numeric
1 87 137.72
2 477 106.79
3 273 130.72
4 550 151.69
5 51 123.70
6 190 102.75
7 424 109.71
8 406 121.69
9 176 151.68
Successfully run. Total query runtime: 246 msec. 296 rows affected.
Total rows: 296 of 296
Query complete 00:00:00.246


Object Explorer
Servers (1)
PostgreSQL 15
Databases (2)
dvdrental
Casts
Catalogs
Event Triggers
Extensions
Foreign Data Wrappers
Languages
Publications
Schemas (1)
public
Aggregates
Collations
Domains
FTS Configurations
FTS Dictionaries
FTS Parsers
FTS Templates
Foreign Tables
Functions
Materialized Views
Operators
Procedures
Sequences
Tables (15)
actor
address
category
Dashboard
Properties
SQL
Statistics
Dependencies
Dependents
Processes
dvdrental/postgres@PostgreSQL 15*
dvdrental/postgres@PostgreSQL 15
No limit
Query
Query History
Scratch Pad
SELECT store_id, COUNT(*) FROM customer
GROUP BY store_id
HAVING COUNT(*) > 300
Data Output
Messages
Notifications
store_id
smallint
count
bigint
1 1 326
Successfully run. Total query runtime: 201 msec. 1 rows affecte
Total rows: 1 of 1
Query complete 00:00:00.201

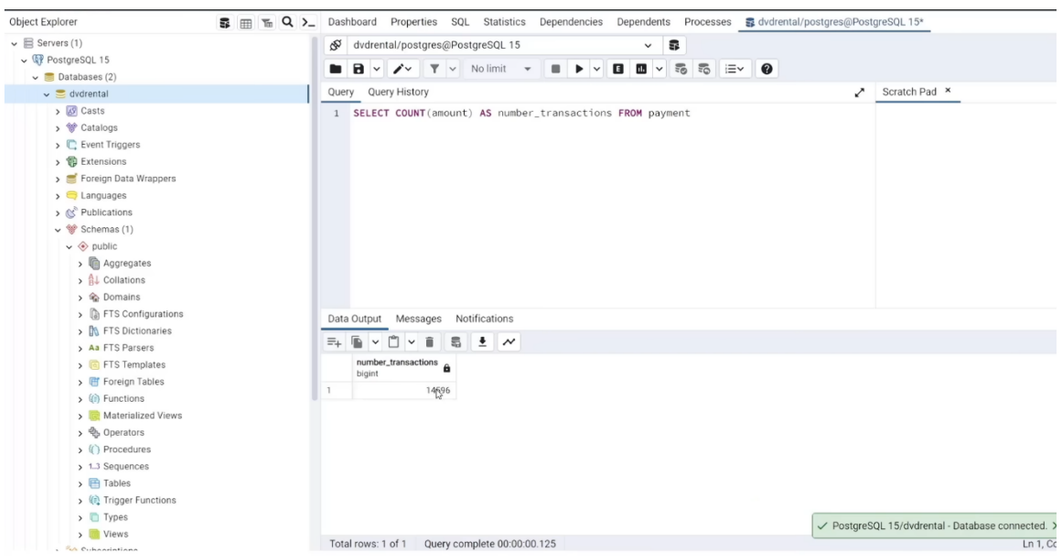
AS

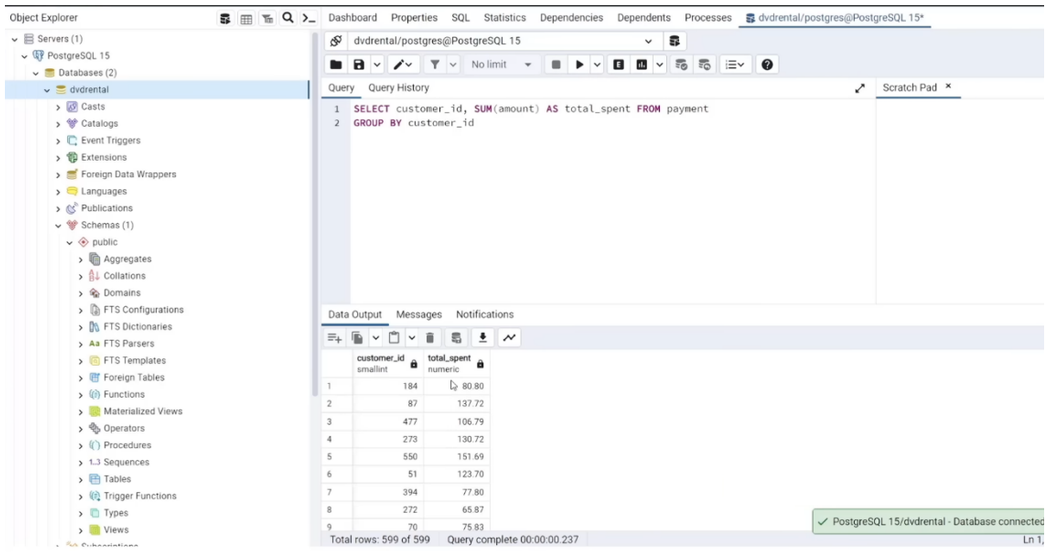
In PostgreSQL, think of the **AS** statement as a tool that lets you give nicknames to tables, columns, or calculations within your queries. Just like how you might give your friends nicknames, you can give these elements easier-to-remember names to make your queries less confusing.

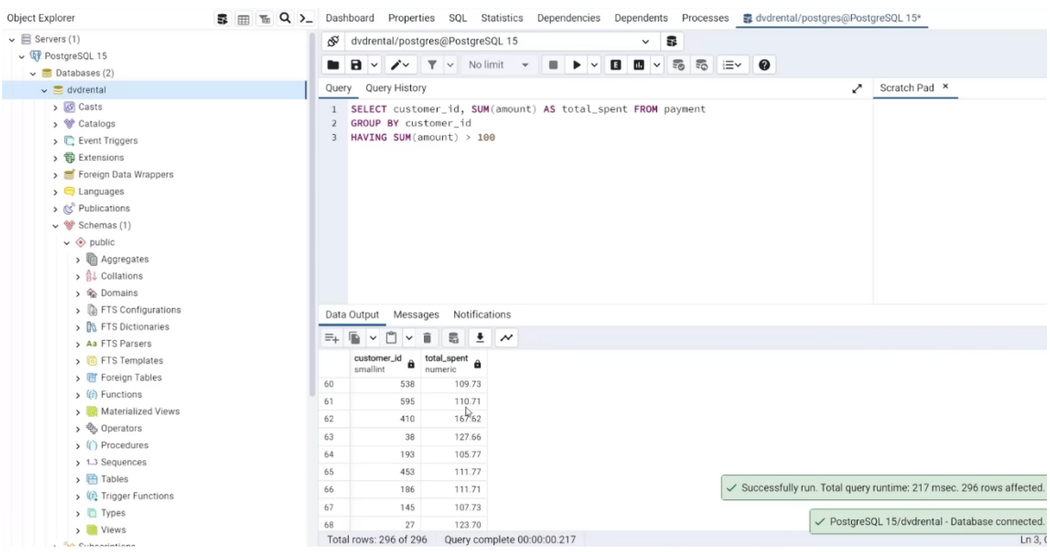
You'd want to use the **"AS"** statement when:

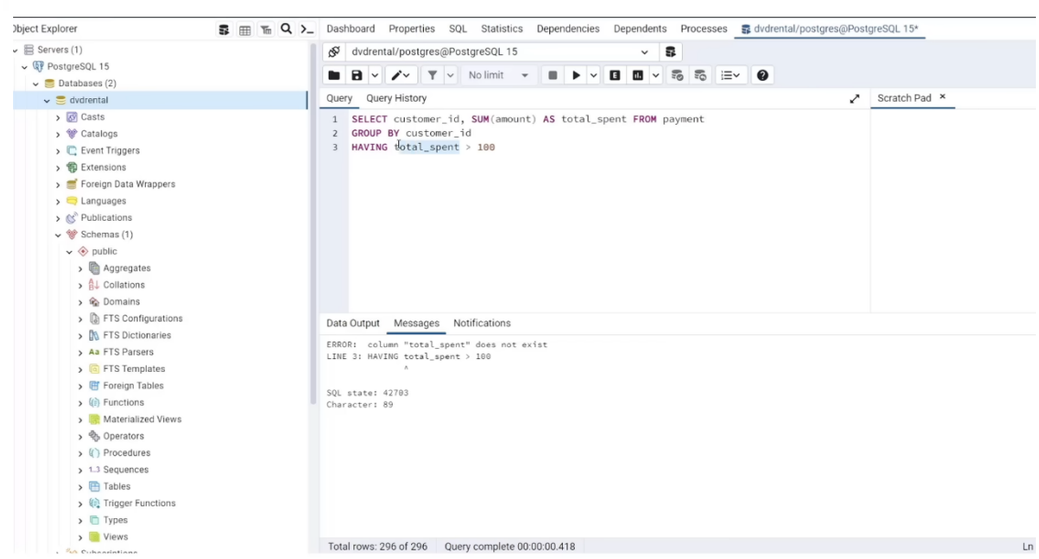
1. You want to give friendlier names to columns in your query results.
2. You're dealing with math or complicated stuff, and you want to simplify things by using a simpler name.
3. You're working with a table and its own copy (kind of like a twin), and you need to tell them apart

Here's how you do it:
SELECT original_name **AS** new_name
FROM table_name;




Object Explorer
Servers (1)
PostgreSQL 15
Databases (2)
dvdrental
Casts
Catalogs
Event Triggers
Extensions
Foreign Data Wrappers
Languages
Publications
Schemas (1)
public
Aggregates
Collations
Domains
FTS Configurations
FTS Dictionaries
FTS Parsers
FTS Templates
Foreign Tables
Functions
Materialized Views
Operators
Procedures
Sequences
Tables
Trigger Functions
Types
Views
Dashboard
Properties
SQL
Statistics
Dependencies
Dependents
Processes
dvdrental/postgres@PostgreSQL 15*
dvdrental/postgres@PostgreSQL 15
No limit
Query
Query History
Scratch Pad
1 SELECT customer_id, SUM(amount) AS total_spent FROM payment
2 GROUP BY customer_id
Data Output
Messages
Notifications
customer_id smallint
total_spent numeric
1 184 80.80
2 87 137.72
3 477 106.79
4 273 130.72
5 550 151.69
6 51 123.70
7 394 77.80
8 272 65.87
9 70 75.83
PostgreSQL 15/dvdrental - Database connected
Total rows: 599 of 599
Query complete 00:00:00.237
Ln 1,


Object Explorer
Servers (1)
PostgreSQL 15
Databases (2)
dvdrental
Casts
Catalogs
Event Triggers
Extensions
Foreign Data Wrappers
Languages
Publications
Schemas (1)
public
Aggregates
Collations
Domains
FTS Configurations
FTS Dictionaries
FTS Parsers
FTS Templates
Foreign Tables
Functions
Materialized Views
Operators
Procedures
Sequences
Tables
Trigger Functions
Types
Views
Dashboard
Properties
SQL
Statistics
Dependencies
Dependents
Processes
dvdrental/postgres@PostgreSQL 15*
dvdrental/postgres@PostgreSQL 15
No limit
Query
Query History
Scratch Pad
1 SELECT customer_id, SUM(amount) AS total_spent FROM payment
2 GROUP BY customer_id
3 HAVING SUM(amount) > 100
Data Output
Messages
Notifications
customer_id smallint
total_spent numeric
60 538 109.73
61 595 110.71
62 410 167.62
63 38 127.66
64 193 105.77
65 453 111.77
66 186 111.71
67 145 107.73
68 27 123.70
Successfully run. Total query runtime: 217 msec. 296 rows affected.
PostgreSQL 15/dvdrental - Database connected.
Total rows: 296 of 296
Query complete 00:00:00.217
Ln 3,


Object Explorer
Servers (1)
PostgreSQL 15
Databases (2)
dvdrental
Casts
Catalogs
Event Triggers
Extensions
Foreign Data Wrappers
Languages
Publications
Schemas (1)
public
Aggregates
Collations
Domains
FTS Configurations
FTS Dictionaries
FTS Parsers
FTS Templates
Foreign Tables
Functions
Materialized Views
Operators
Procedures
Sequences
Tables
Trigger Functions
Types
Views
Dashboard
Properties
SQL
Statistics
Dependencies
Dependents
Processes
dvdrental/postgres@PostgreSQL 15*
dvdrental/postgres@PostgreSQL 15
No limit
Query
Query History
Scratch Pad
1 SELECT customer_id, SUM(amount) AS total_spent FROM payment
2 GROUP BY customer_id
3 HAVING total_spent > 100
Data Output
Messages
Notifications
ERROR: column "total_spent" does not exist
LINE 3: HAVING total_spent > 100
^
SQL state: 42703
Character: 89
Total rows: 296 of 296
Query complete 00:00:00.418
Ln