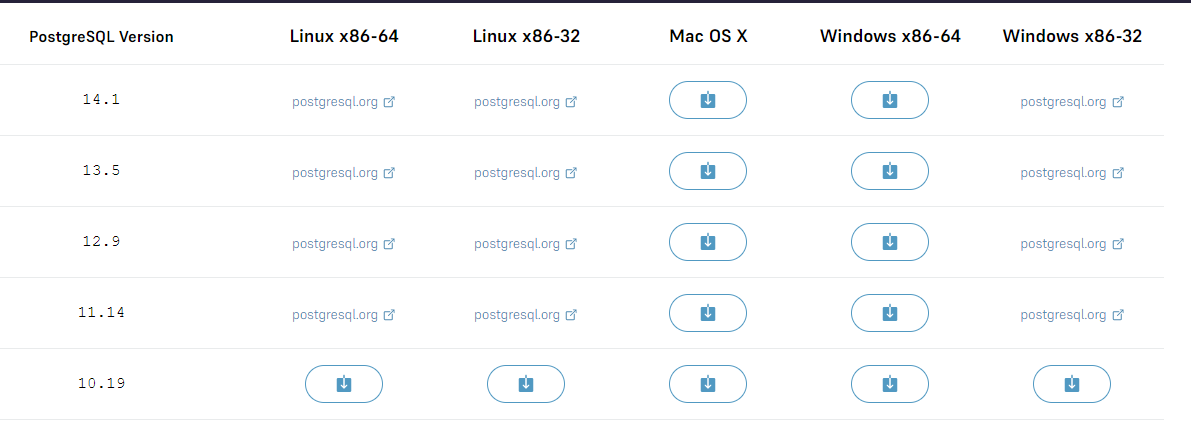
PostgreSQL Documentation

**STEP 1: Download the installer of pSQL**

First, let’s visit this website for a select download.

<https://www.enterprisedb.com/downloads/postgres-postgresql-downloads>

Select the version that you like … I refer that the more stable version is down one of the latest versions.

So we are going to use the **Windows x86-64 version 13.5** 

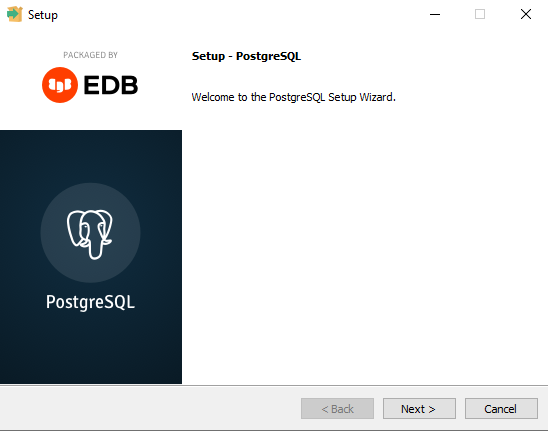
And just wait until the download is done. And let us proceed to install.



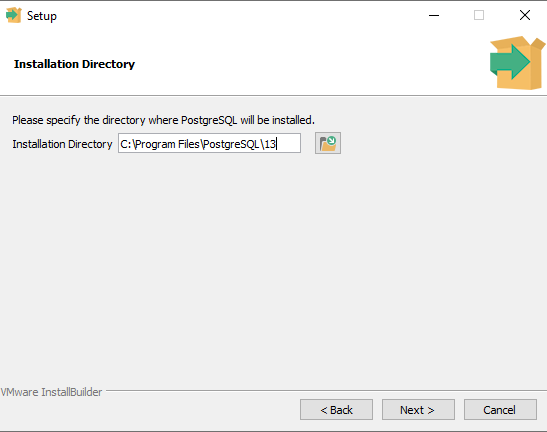
**STEP 2: Install the PostgreSQL**

To install the PostgreSQL open your file explorer and go to download.

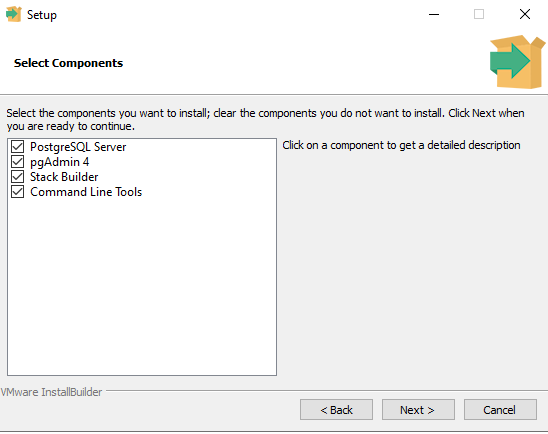
Then open your PostgreSQL installer. Select and press ENTER or Double click it.

Then the installation will show up

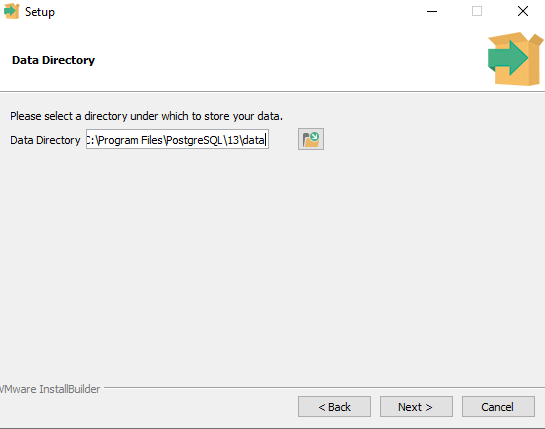
This is the directory where your postgre location. Set it to default.



This section is you going to choose what you want to install. I recommend to install everything we can use all of this sooner or later.

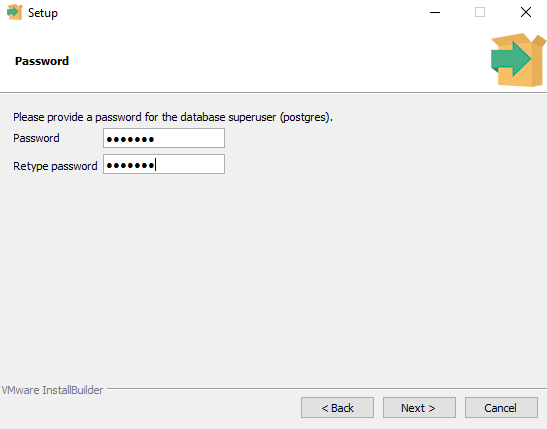


This is your data directory where your all data location. Just set to default and click next.

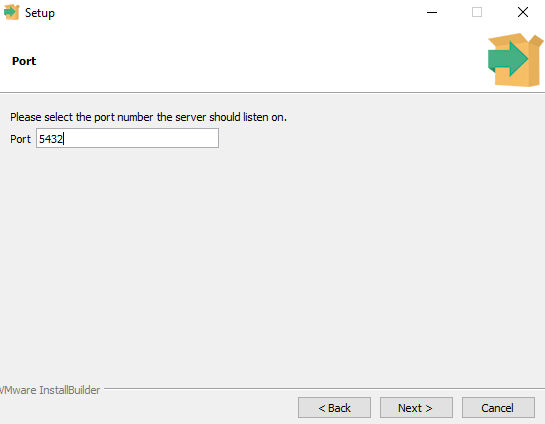


Now, we are going to set a password for you PostgreSQL. Just simply type your password 2times.

And click next.



From here all you need is to set all section to default and simply click next.





Wait until the installation is done. Just simply click finish. And just close the stock builder after that.

**STEP 3:** **Basic PostgreSQL**

To begin lets open our SQL Shell, and simply press ENTER.

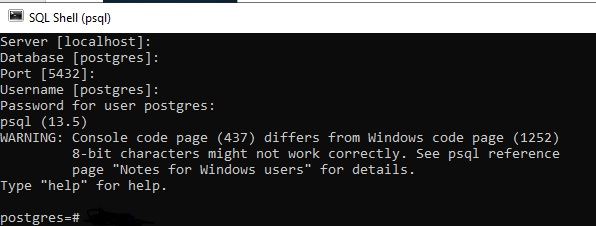
**Server** press ENTER.

**Database** press ENTER.

**Port [5432]** press ENTER.

**Username** press ENTER.

**Password** type your password that you set in postgre installation.



And we are good to go.

**Managing Tables**

In PostgreSQL we have a Data Type and there are. **PostgreSQL data types** including Boolean, character, numeric, temporal, array, json, uuid, and special types.

**<https://www.postgresqltutorial.com/postgresql-data-types/>**

**Boolean**

Character types such as **char, varchar, and text.**

Numeric types such as integer and floating-point number.

Temporal types such as **date, time, timestamp, and interval**

**UUID** for storing Universally Unique Identifiers

**Array** for storing array strings, numbers, etc.

**JSON** stores JSON data

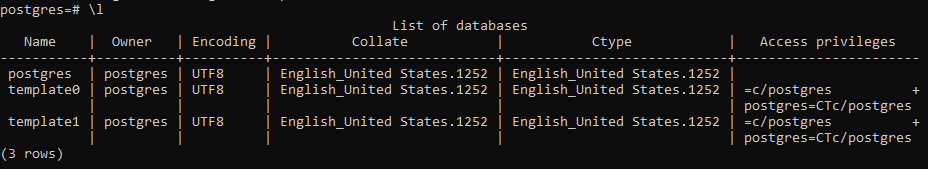
**hstore** stores key-value pair

Special types such as network address and geometric data**.**

**CREATE DATABASE**

First, let’s check our databases by using this command \l

As you see we have only 3 database so we are going to create new database to work on.



We have successfully created our database. check your database list again by using \l

To create a database we need to follow this query.

**CREATE DATABASE tutor;**



Let’s connect to our database so we can create our table inside or database.

Use this command **\c name\_of\_database;**



Now, we are going to create our **TABLE.** To do that just follow the simple query below.

**CREATE TABLE accounts (**

**user\_id serial PRIMARY KEY,**

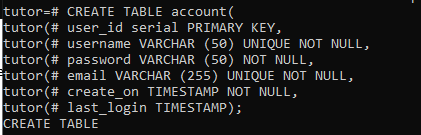
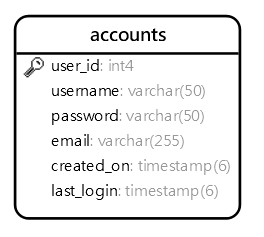
**username VARCHAR ( 50 ) UNIQUE NOT NULL,**

**password VARCHAR ( 50 ) NOT NULL,**

**email VARCHAR ( 255 ) UNIQUE NOT NULL,**

**created\_on TIMESTAMP NOT NULL,**

**last\_login TIMESTAMP );**

We have successfully created our 1st table. Let’s add more tables to work with.

The following statement creates the **roles** table that consists of two columns: role\_id and role\_name.

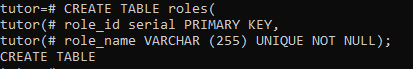
Follow the query below.

**CREATE TABLE roles(**

**role\_id serial PRIMARY KEY,**

**role\_name VARCHAR (255) UNIQUE NOT NULL**

**);**

We have successfully created our 2nd table.

The next statement is to create the **account\_roles** 3rd table that has three columns: user\_id, role\_id, and grant\_date.

**CREATE TABLE account\_roles (**

**user\_id INT NOT NULL,**

**role\_id INT NOT NULL,**

**grant\_date TIMESTAMP,**

**PRIMARY KEY (user\_id, role\_id),**

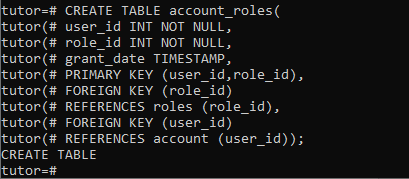
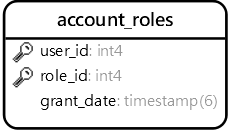
**FOREIGN KEY (role\_id)**

**REFERENCES roles (role\_id),**

**FOREIGN KEY (user\_id)**

**REFERENCES account (user\_id)**

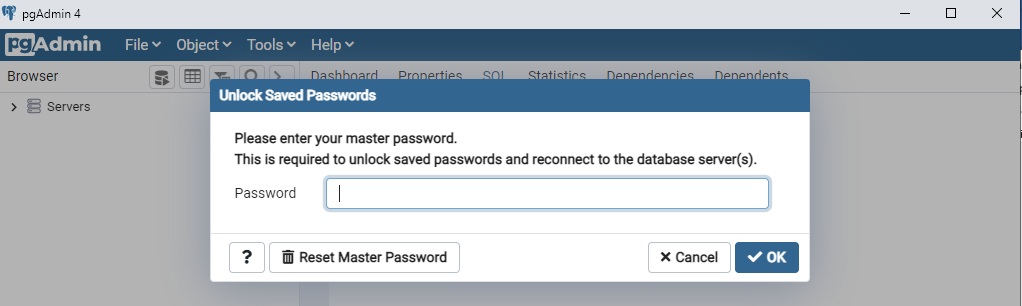
**);**

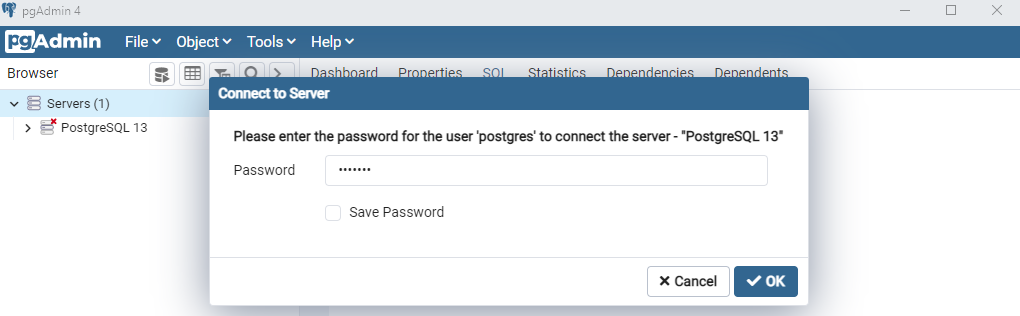
 



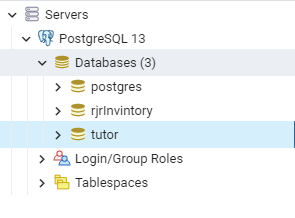
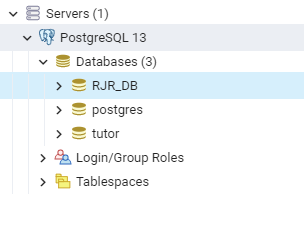
Here we are going to use pg admin 4 to practice the data and the interface of this software.

Input your password like in postgre shell ..you enter the database server and another pass for the PostgreSQL 13 and press ok.

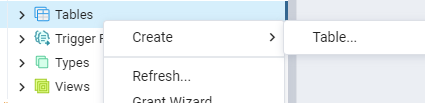




This is database that we create in Shell tutor database. This database I create inside the pg admin.

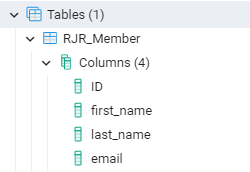
 

Now, select rjrinvitory/schemas/public/table, right-click the tables, and create or select **query tool**. You can create a table manually or you can create using a query. Now we are in creating tables.

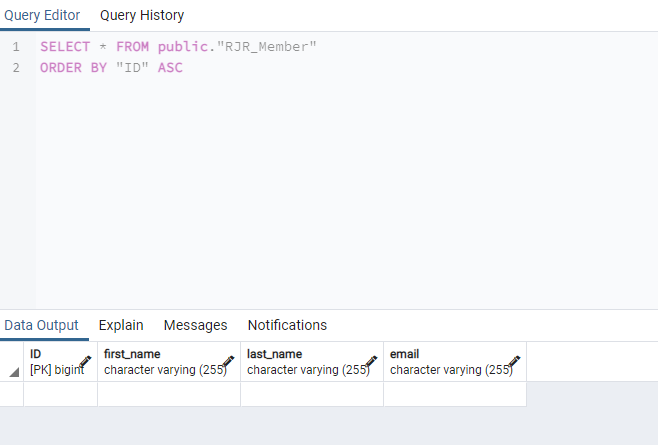


After we created out table we are now proceeding to SELECT INTO statement. Lets go and add some data inside the table.

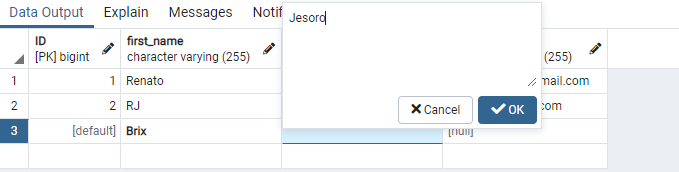
We successfully created our tables and now let add some data to it.



Right-click your table name and select View/Edit data and select all rows. So this will show up. you can edit here manually for the data or you can use query.



Double click the blank and edit and press ok ,



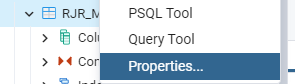
This icon can save you change 

Lets go to section 1.

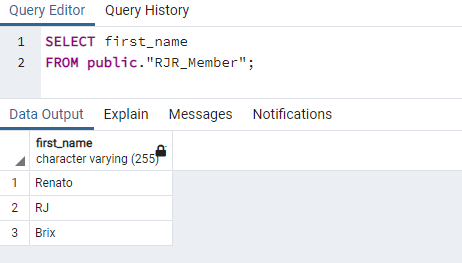
**Section 1**

To show you what is the use of this query is to show the roll of the table. **SELECT select\_list FROM table\_name;**

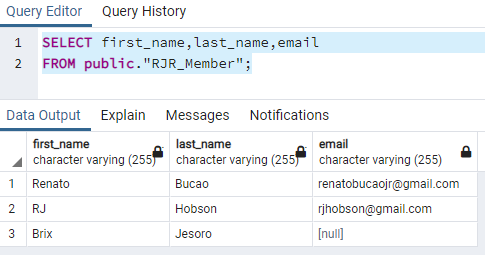
Go to query tool. Right-click your database name and select query tool.



So in Pg admin, as you see I use “ “ in table names you can also don’t use that.



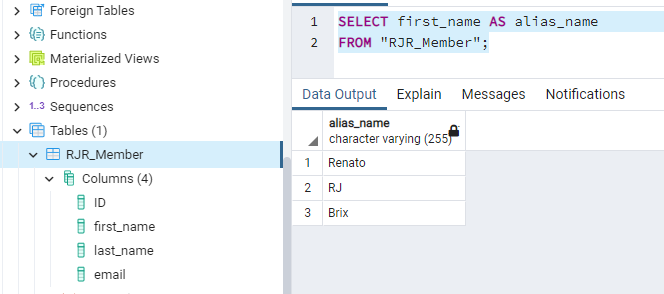
You can also try this query.

SELECT first\_name,last\_name,email FROM “table\_name”; 

Next, what is the use of this **SELECT first\_name AS alias\_name**

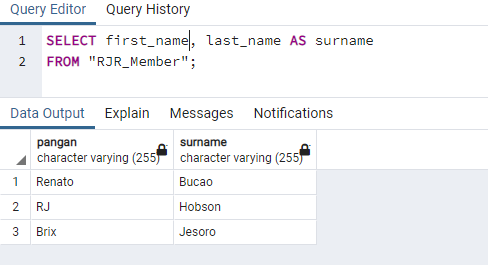
**FROM table\_name;**

As you see we change our first\_name to alias\_name. but it can’t change your real roll name.



Next, what is the use of this **SELECT first\_name AS alias\_name**

This query assigned the **surname** as the alias of the **last\_name** column:



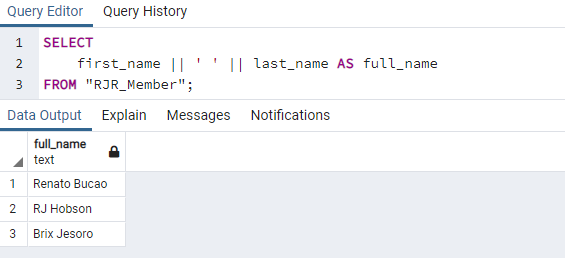
Assigning a column alias to an expression example

The following query returns the full names of all customers. It constructs the full name by concatenating the first name, space, and the last name.

**SELECT**

**first\_name || ' ' || last\_name**

**FROM customer;**



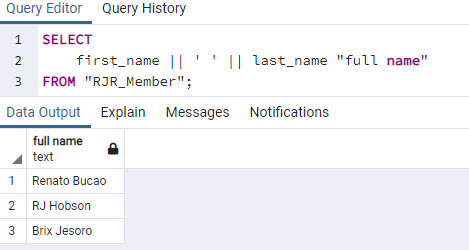
you use the || as the concatenating operator that concatenates one or more strings into a single string.

Column aliases that contain spaces

If a column alias contains one or more spaces, you need to surround it with double quotes like this.

**SELECT**

**first\_name || ' ' || last\_name "full name FROM table\_name;**



Next, we are in ORDER BY statement When you query data from a table, the **SELECT** statement returns rows in an unspecified order. To sort the rows of the result set, you use the **ORDER BY** clause in the **SELECT** statement.

**SELECT select\_list**

**FROM table\_name**

**ORDER BY**

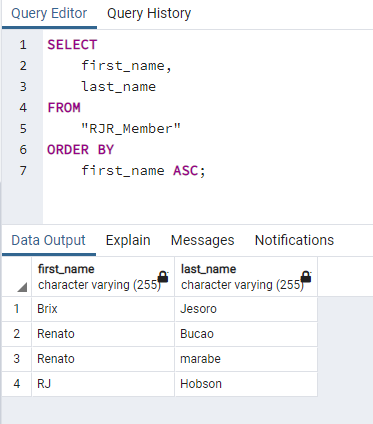
**sort\_expression1 [ASC | DESC],**

**...**

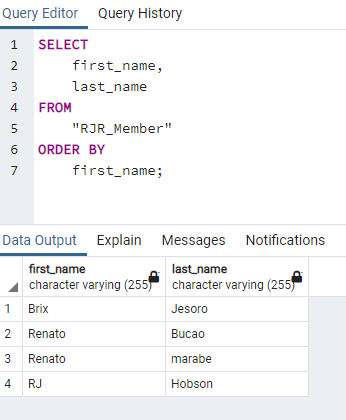
**sort\_expressionN [ASC | DESC**];

The following query uses the ORDER BY clause to sort customers by their first names in ascending order.

**SELECT first\_name, last\_name FROM customer ORDER BY first\_name ASC;**

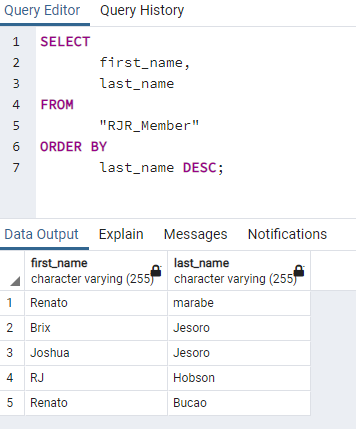


Since the ASC option is the default, you can omit it in the ORDER BY clause like this.



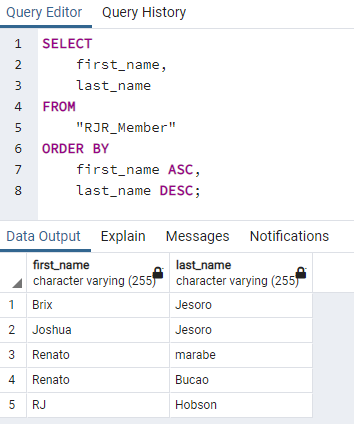
Next, is using PostgreSQL ORDER BY clause to sort rows by one column in descending order

The following statement selects the first name and last name from the customer table and sorts the rows by values in the last name column in descending order.



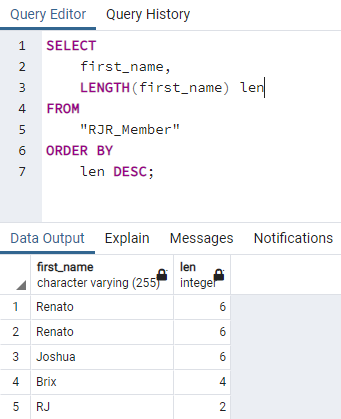
Next, is Using PostgreSQL ORDER BY clause to sort rows by multiple columns.

The following statement selects the first name and last name from the customer table and sorts the rows by the first name in ascending order and last name in descending order.



Next, we have Using PostgreSQL ORDER BY clause to sort rows by expressions.

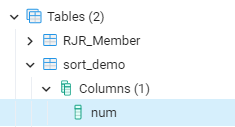
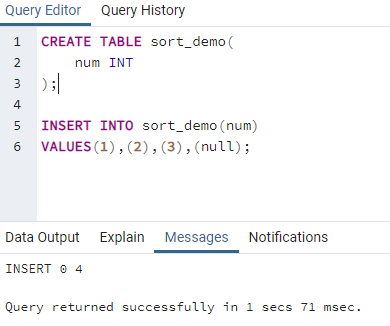
The LENGTH() function accepts a string and returns the length of that string. The following statement selects the first names and their lengths. It sorts the rows by the lengths of the first names.



Next, PostgreSQL ORDER BY clause and NULL.

The NULLS FIRST option places NULL before other non-null values and the NULL LAST option places NULL after other non-null values.

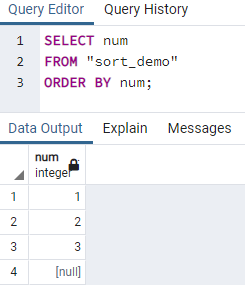
Let’s **[create a table](https://www.postgresqltutorial.com/postgresql-create-table/)** for the demonstration



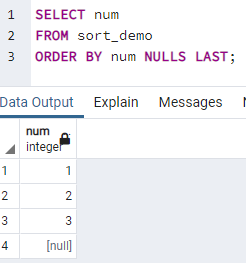
Note that you don’t need to understand the CREATE TABLE and INSERT statements. You just need to execute it from pgAdmin or psql to create the sort\_demo table and insert data into it.

The following query returns data from the **sort\_demo** table.

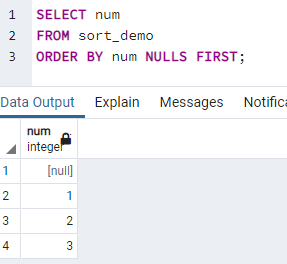
 It places NULL after other values



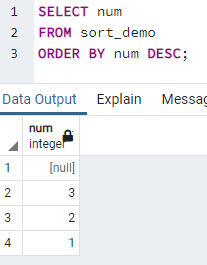
So if you use the ASC option, the ORDER BY clause uses the NULLS LAST option by default. Therefore, the following query returns the same result.



To place NULL before other non-null values, you use the NULLS FIRST option

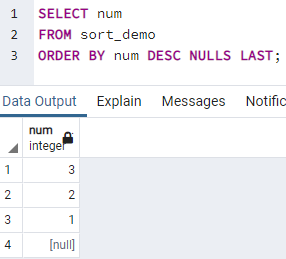


The following statement sorts values in the num column of the sort\_demo table in descending order.

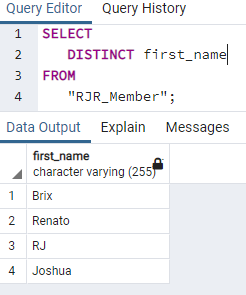


As you can see clearly from the output, the ORDER BY clause with the DESC option uses the NULLS FIRST by default.

To reverse the order, you can use the NULLS LAST option.



Now let’s go to PostgreSQL SELECT DISTINCT clause.  
The DISTINCT clause is used in the SELECT statement to remove duplicate rows from a result set. The DISTINCT clause keeps one row for each group of duplicates. The DISTINCT clause can be applied to one or more columns in the select list of the SELECT statement

 we have only one renato left.

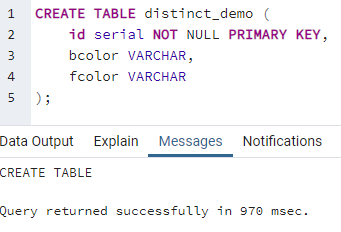
PostgreSQL also provides the DISTINCT ON (expression) to keep the “first” row of each group of duplicates using the following syntax.

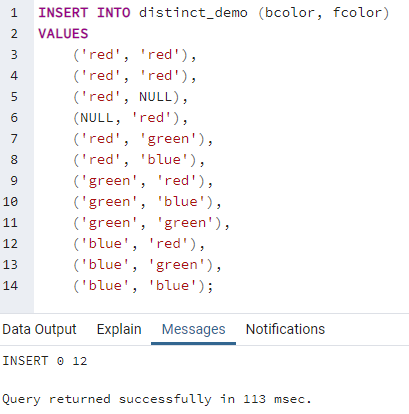
**SELECT DISTINCT ON (column1) column\_alias, column2 FROM table\_name ORDER BY column1, column2;**

Let’s [create a new table](https://www.postgresqltutorial.com/postgresql-create-table/) called distinct\_demo and [insert data](https://www.postgresqltutorial.com/postgresql-insert/) into it for practicing the DISTINCT clause

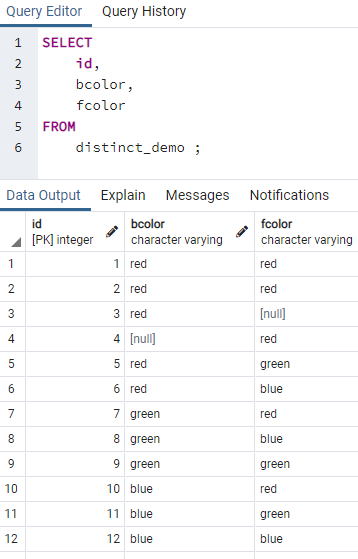
First, use the following [CREATE TABLE](https://www.postgresqltutorial.com/postgresql-create-table/) statement to create the **distinct\_demo** table that consists of three columns: id, bcolor and fcolor.

We successfully created our new table. Next, we are going to add some data.



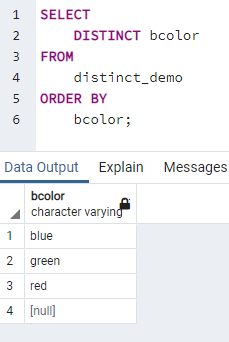


Third, query the data from the **distinct\_demo** table using the [SELECT](https://www.postgresqltutorial.com/postgresql-select/) statement



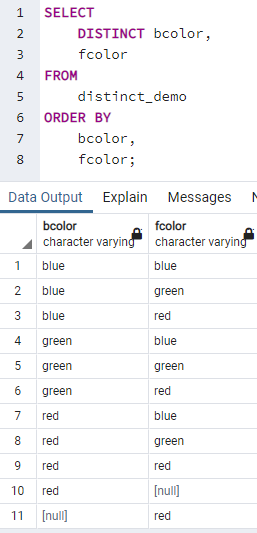
Next, PostgreSQL DISTINCT one column example.

The following statement selects unique values in the **bcolor** column from the **t1** table and sorts the result set in alphabetical order by using the **ORDER** BY clause.



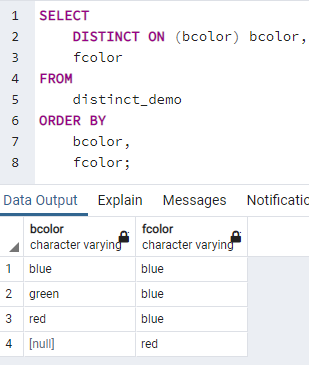
Next, PostgreSQL DISTINCT multiple columns

The following statement demonstrates how to use the DISTINCT clause on multiple columns.



Now, let’s do the PostgreSQL DISTINCT ON example

The following statement sorts the result set by the **bcolor** and **fcolor**, and then for each group of duplicates, it keeps the first row in the returned result set.



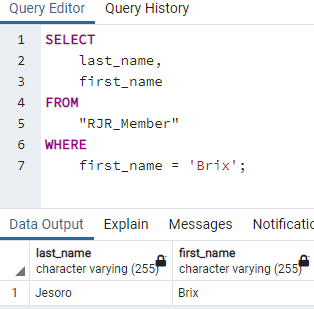
**Filtering Data**

PostgreSQL WHERE clause overview. WHERE you can add condition

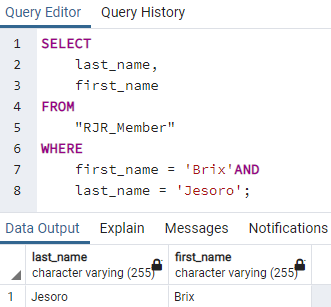
The syntax of the PostgreSQL WHERE clause is as follows:

**SELECT select\_list FROM table\_name WHERE condition ORDER BY sort\_expression**

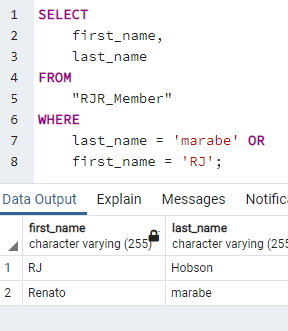
Using WHERE clause with the equal (=) operator example.



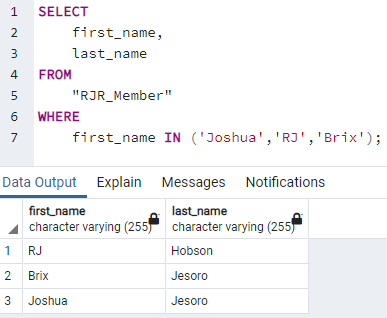
Next, Using WHERE clause with the AND operator example



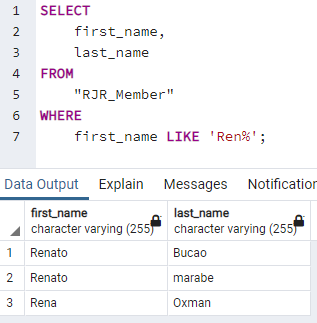
Next, we are using the WHERE clause with the OR operator example.



Next, We are using WHERE clause with the IN operator example.

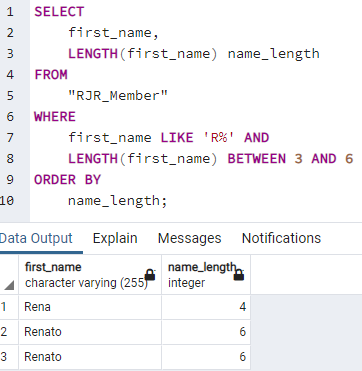


Next, we use the WHERE clause with the LIKE operator example.

The % is called a wildcard that matches any string. The 'Ren%' pattern matches any string that starts with 'Ren'.

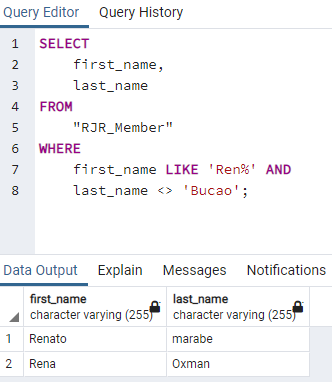
Next, use the WHERE clause with the BETWEEN operator example.

The following example finds customers whose first names start with the letter R and contains 3 to 6 characters by using the BETWEEN operator.



In this example, we used the LENGTH() function gets the number of characters of an input string.

Last we do is using the WHERE clause with the not equal operator (<>) example.



Note that you can use the != operator and <> operator interchangeably because they are equivalent.

We have learned how to use PostgreSQL WHERE clause in the SELECT statement to filter rows based on a specified condition.

Let’s know how to use the PostgreSQL LIMIT clause to get a subset of rows generated by a query.

The following illustrates the syntax of the LIMIT clause:

**SELECT select\_list**

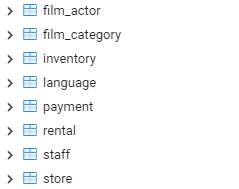
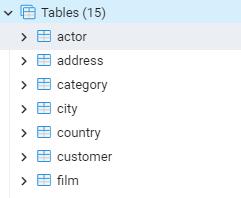
**FROM table\_name**

**ORDER BY sort\_expression**

**LIMIT row\_count;**

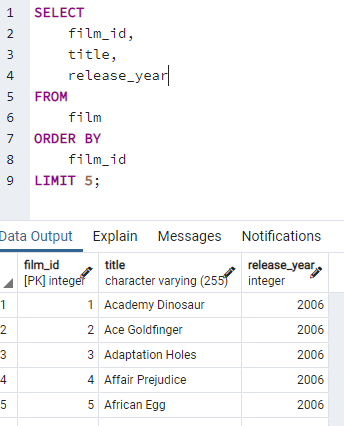
Now, lets try to import the database to our pg admin. Just simply follow this link <https://www.postgresqltutorial.com/load-postgresql-sample-database/>

We successfully imported the data inside out pg admin.

 this is name of the database and we have  15 tables and this are all the name of tables .  So lets start.

Now, go going to use PostgreSQL LIMIT to constrain the number of returned rows example.

This example uses the LIMIT clause to get the first five films sorted by film\_id:

 only 5 data are showing.

Next, let’s use PostgreSQL LIMIT OFFSSET to get top / bottom N rows

Typically, you often use the LIMIT clause to select rows with the highest or lowest values from a table

For example, to get the top 10 most expensive films in terms of rental, you sort films by the rental rate in descending order and use the LIMIT clause to get the first 10 films. The following query illustrates the idea

the PostgreSQL LIMIT OFFSET clause to retrieve a subset of rows returned by a query.

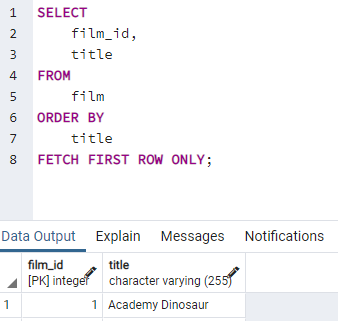
Let’s move forward to constrain the number of rows returned by a query, you often use the LIMIT clause. The LIMIT clause is widely used by many relational database management systems such as MySQL, H2, and HSQLDB. However, the LIMIT clause is not a SQL-standard.

To conform with the SQL standard, PostgreSQL supports the FETCH clause to retrieve a number of rows returned by a query. Note that the FETCH clause was introduced in SQL:2008.

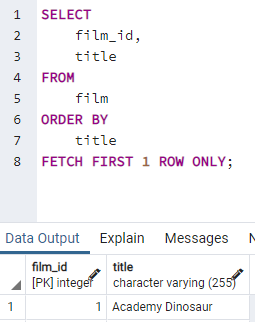
The following illustrates the syntax of the PostgreSQL FETCH clause:

**OFFSET start { ROW | ROWS }**

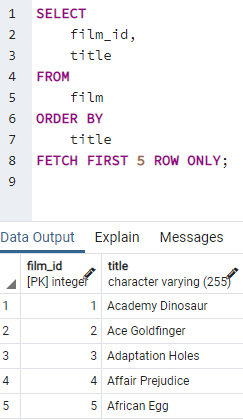
**FETCH { FIRST | NEXT } [ row\_count ] { ROW | ROWS } ONLY**



It is equivalent to the following query.



The following query use the FETCH clause to select the first five films sorted by titles:

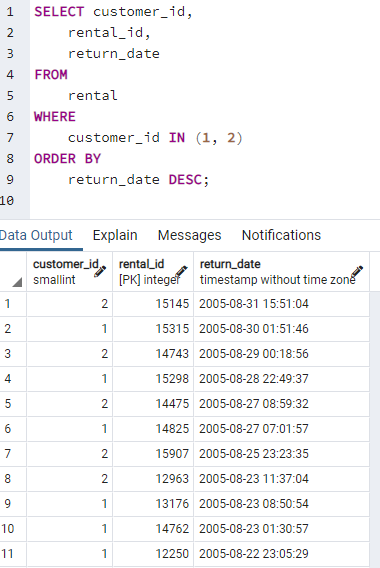


Next, we are going to learn PostgreSQL IN operator syntax

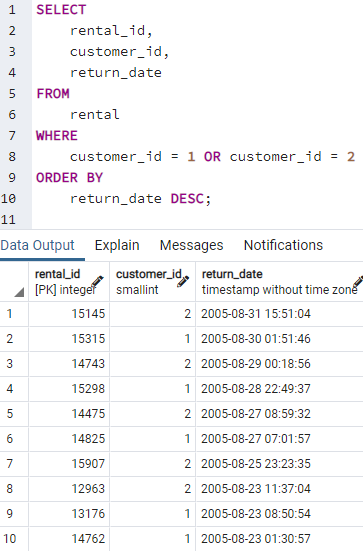
You use IN operator in the WHERE clause to check if a value matches any value in a list of values.

The syntax of the IN operator is as follows: **value IN (value1,value2,...)**

The IN operator returns true if the value matches any value in the list i.e., value1 , value2 The IN operator examples

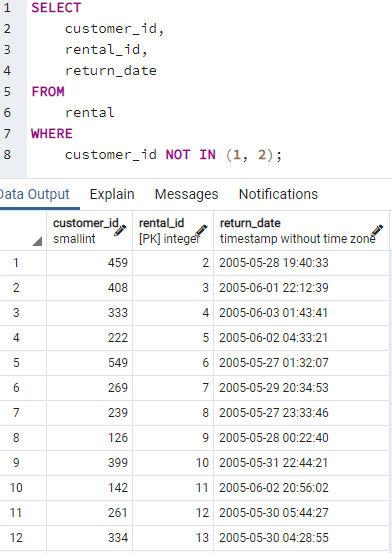


The next following query uses the equal (=) and OR operators instead of the IN operator.

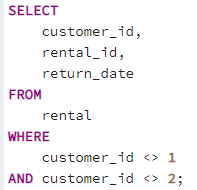


You can combine the IN operator with the NOT operator to select rows whose values do not match the values in the list.

following statement finds all rentals with the customer id is not 1 or 2.

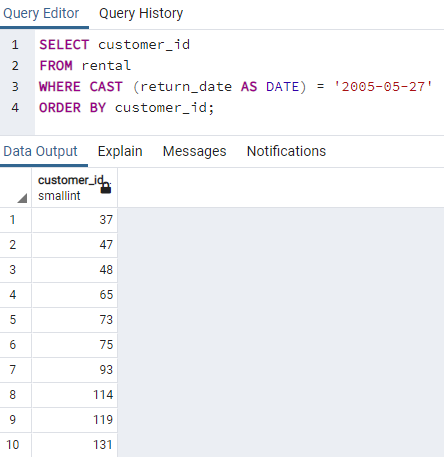
 As you see there is no 1 and 2 in customer\_id.

Similar to the IN operator, you can use the not equal (<>) and AND operators to write the NOT IN operator.

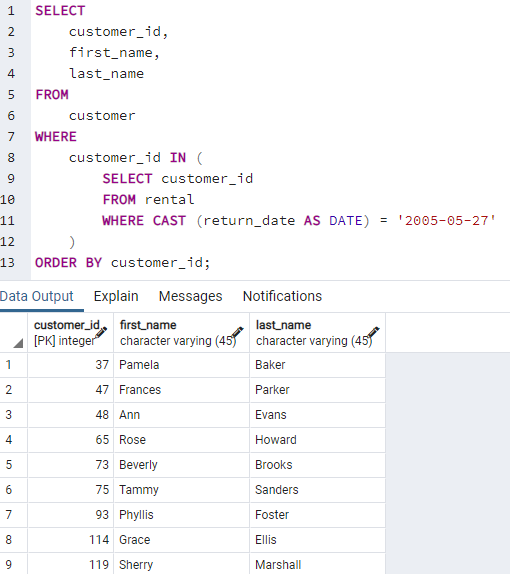


Let’s moving for ward to PostgreSQL IN with a subquery.

The following query returns a list of customer ids from the rental table with the return date is 2005-05-27



Because this query returns a list of values, you can use it as the input of the IN operator like this:



Now let’s learn how to use the PostgreSQL **BETWEEN** operator to match a value against a range of values.

You use the BETWEEN operator to match a value against a range of values. The following illustrates the syntax of the BETWEEN operator

value BETWEEN low AND high;

Code language: SQL (Structured Query Language) (sql)

If the value is greater than or equal to the low value and less than or equal to the high value, the expression returns true, otherwise, it returns false.

You can rewrite the BETWEEN operator by using the greater than or equal ( >=) or less than or equal ( <=) operators like this:

value >= low and value <= high

Code language: SQL (Structured Query Language) (sql)

If you want to check if a value is out of a range, you combine the NOT operator with the BETWEEN operator as follows:

value NOT BETWEEN low AND high;

Code language: SQL (Structured Query Language) (sql)

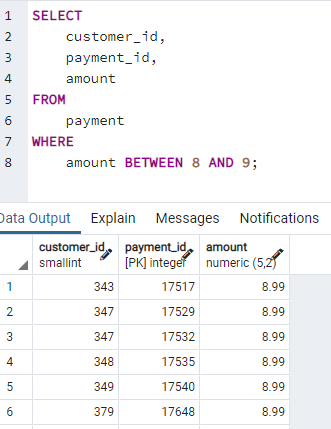
The following expression is equivalent to the expression that uses the NOT and BETWEEN operators:

value < low OR value > high

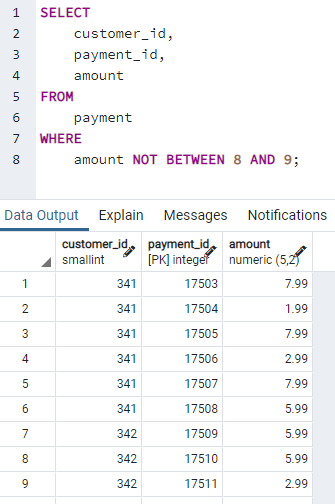
Code language: SQL (Structured Query Language) (sql)

You often use the BETWEEN operator in the [WHERE](https://www.postgresqltutorial.com/postgresql-where/" \o "PostgreSQL WHERE)clause of a [SELECT](https://www.postgresqltutorial.com/postgresql-select/" \o "PostgreSQL SELECT), [INSERT](https://www.postgresqltutorial.com/postgresql-insert/), [UPDATE](https://www.postgresqltutorial.com/postgresql-update/)or [DELETE](https://www.postgresqltutorial.com/postgresql-delete/) statement.

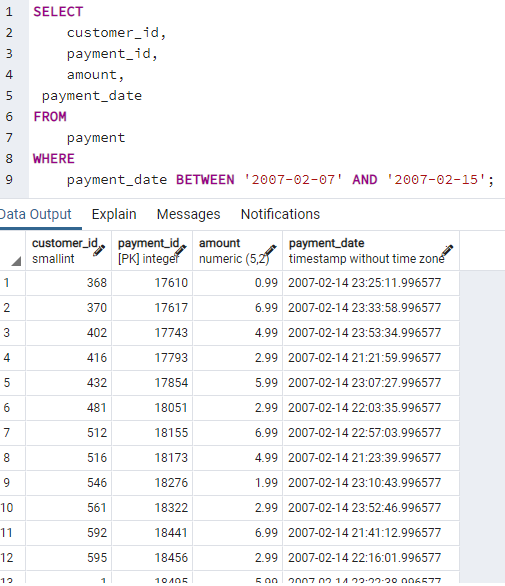
BETWEEN operator examples



To get payments whose amount is not in the range of 8 and 9, you use the following query.

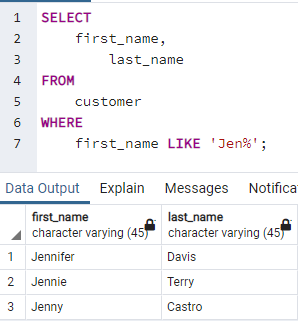


If you want to check a value against of date ranges, you should use the literal date in ISO 8601 format i.e., YYYY-MM-DD. For example, to get the payment whose payment date is between 2007-02-07 and 2007-02-15, you use the following query.



Now, let’s procced to PostgreSQL LIKE operator.

Fortunately, you can use the PostgreSQL LIKE operator to match the first name of the customer with a string like this query.



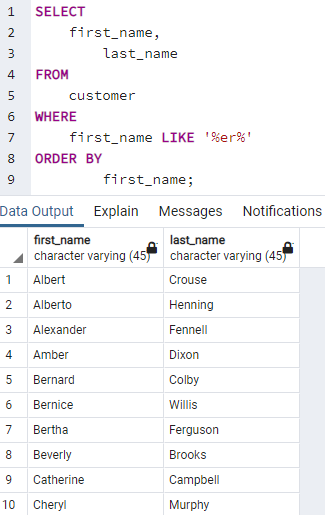
You construct a pattern by combining literal values with wildcard characters and use the LIKE or NOT LIKE operator to find the matches. PostgreSQL provides you with two wildcards:

Percent sign ( %) matches any sequence of zero or more characters.

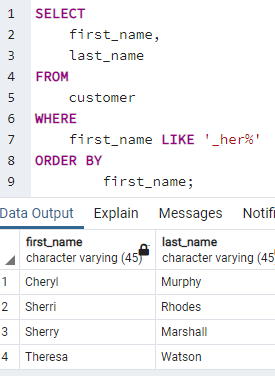
Underscore sign ( \_) matches any single character.

The syntax of PostgreSQL LIKE operator is as follows:  and 

For example, the following query returns customers whose first name contains er string like Jenifer, Kimberly, etc.

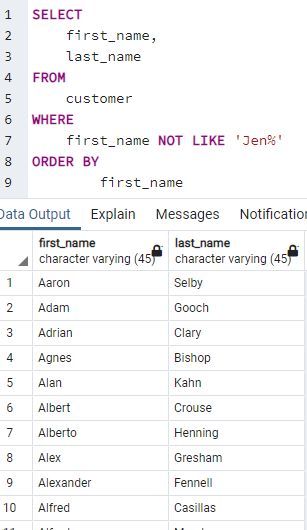


You can combine the percent ( % ) with underscore ( \_ ) to construct a pattern as the following example.

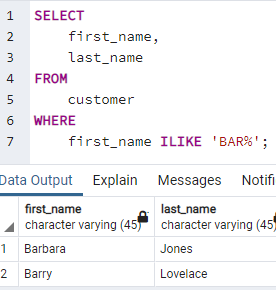


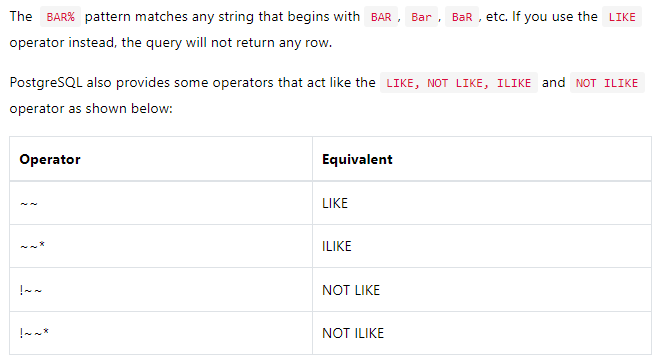
* Begin with any single character ( \_ )
* And is followed by the literal string her.
* And is ended with any number of characters.

The following query uses the NOT LIKE operator to find customers whose first names do not begin with Jen.

no jen available.

PostgreSQL supports the ILIKE operator that works like the LIKE operator. In addition, the ILIKE operator matches value case-insensitively. For example





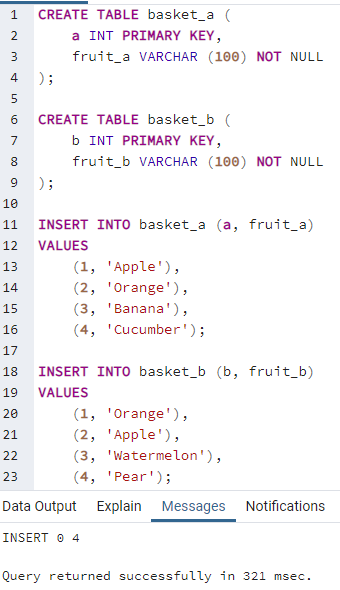
**Section 3. Joining Multiple Tables**

PostgreSQL join is used to combine columns from one ([self-join](https://www.postgresqltutorial.com/postgresql-self-join/)) or more tables based on the values of the common columns between related tables. The common columns are typically the [primary key](https://www.postgresqltutorial.com/postgresql-primary-key/) columns of the first table and [foreign key](https://www.postgresqltutorial.com/postgresql-foreign-key/) columns of the second table.

PostgreSQL supports [inner join](https://www.postgresqltutorial.com/postgresql-inner-join/),[left join](https://www.postgresqltutorial.com/postgresql-left-join/), [right join](https://www.postgresqltutorial.com/postgresql-right-join/), [full outer join](https://www.postgresqltutorial.com/postgresql-full-outer-join/), [cross join](https://www.postgresqltutorial.com/postgresql-cross-join/), [natural join](https://www.postgresqltutorial.com/postgresql-natural-join/), and a special kind of join called [self-join](https://www.postgresqltutorial.com/postgresql-self-join/).

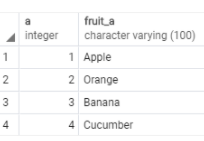
Setting up sample tables

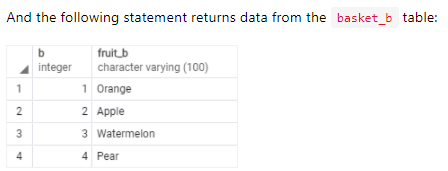
Suppose you have two tables called basket\_a and basket\_b that store fruits



The tables have some common fruits such as apple and orange.

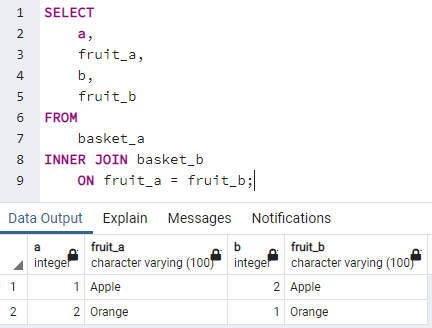
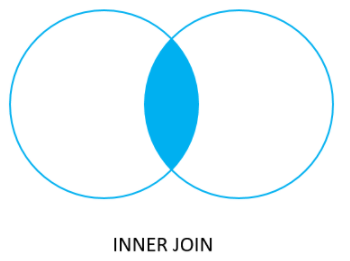
The following statement returns data from the basket\_a table:





**PostgreSQL inner join**

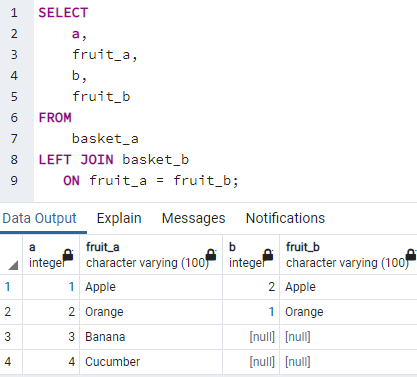
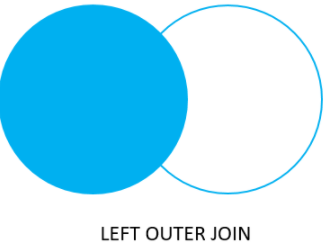
The following statement joins the first table (basket\_a) with the second table (basket\_b) by matching the values in the fruit\_a and fruit\_b columns

The inner join examines each row in the first table (basket\_a). It compares the value in the fruit\_a column with the value in the fruit\_b column of each row in the second table (basket\_b). If these values are equal, the inner join creates a new row that contains columns from both tables and adds this new row the result set

**PostgreSQL left join**

The following statement uses the left join clause to join the basket\_a table with the basket\_b table. In the left join context, the first table is called the left table and the second table is called the right table.

The left join starts selecting data from the left table. It compares values in the fruit\_a column with the values in the fruit\_b column in the basket\_b table.

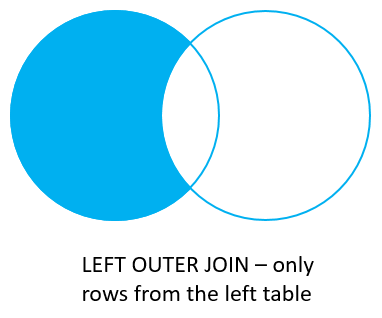
If these values are equal, the left join creates a new row that contains columns of both tables and adds this new row to the result set. (see the row #1 and #2 in the result set).

In case the values do not equal, the left join also creates a new row that contains columns from both tables and adds it to the result set. However, it fills the columns of the right table (basket\_b) with null. (see the row #3 and #4 in the result set).

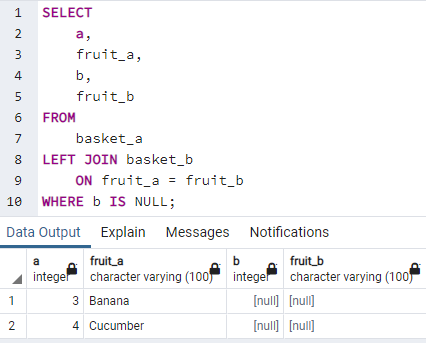
The following Venn diagram illustrates the left join.

Note that the LEFT JOIN is the same as the LEFT OUTER JOIN so you can use them interchangeably.

The following Venn diagram illustrates the left join that returns rows from the left table that do not have matching rows from the right table:



To select rows from the left table that do not have matching rows in the right table, you use the left join with a [WHERE](https://www.postgresqltutorial.com/postgresql-where/) clause. For example



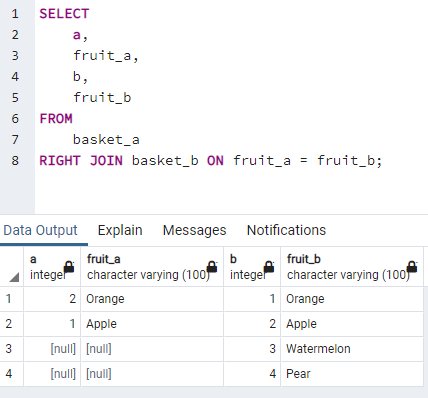
**PostgreSQL right join**

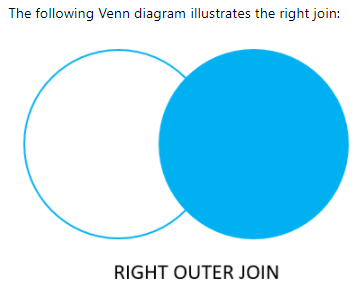
The [right join](https://www.postgresqltutorial.com/postgresql-right-join/) is a reversed version of the left join. The right join starts selecting data from the right table. It compares each value in the fruit\_b column of every row in the right table with each value in the fruit\_a column of every row in the fruit\_a table.

If these values are equal, the right join creates a new row that contains columns from both tables.

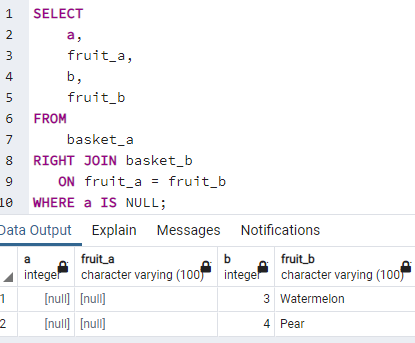
In case these values are not equal, the right join also creates a new row that contains columns from both tables. However, it fills the columns in the left table with NULL.

The following statement uses the right join to join the basket\_a table with the basket\_b table.



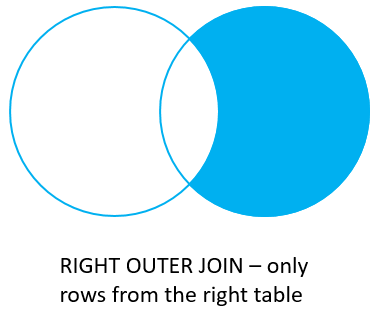


Similarly, you can get rows from the right table that do not have matching rows from the left table by adding a WHERE clause as follows.



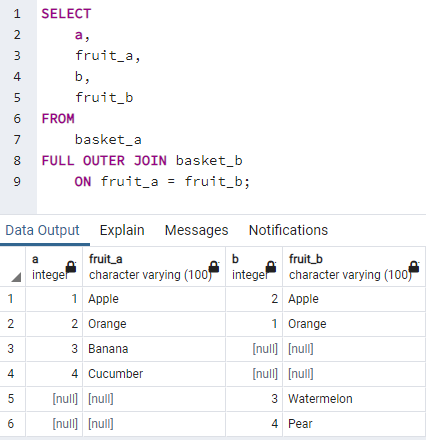
The RIGHT JOIN and RIGHT OUTER JOIN are the same therefore you can use them interchangeably.

The following Venn diagram illustrates the right join that returns rows from the right table that do not have matching rows in the left table:

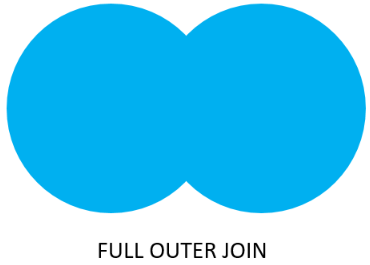


**PostgreSQL full outer join**

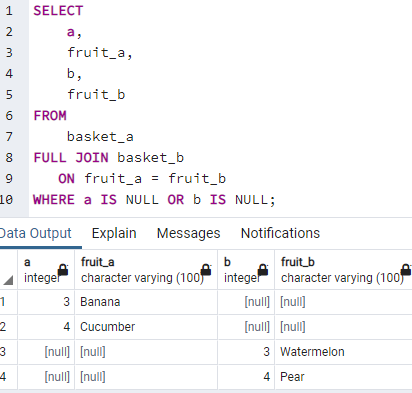
The [full outer join](https://www.postgresqltutorial.com/postgresql-full-outer-join/) or full join returns a result set that contains all rows from both left and right tables, with the matching rows from both sides if available. In case there is no match, the columns of the table will be filled with NULL.



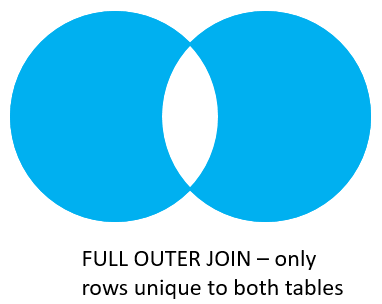
The following Venn diagram illustrates the full outer join:



To return rows in a table that do not have matching rows in the other, you use the full join with a WHERE clause like this:



The following Venn diagram illustrates the full outer join that returns rows from a table that do not have the corresponding rows in the other table:



The following picture shows all the PostgreSQL joins that we discussed so far with the detailed syntax:

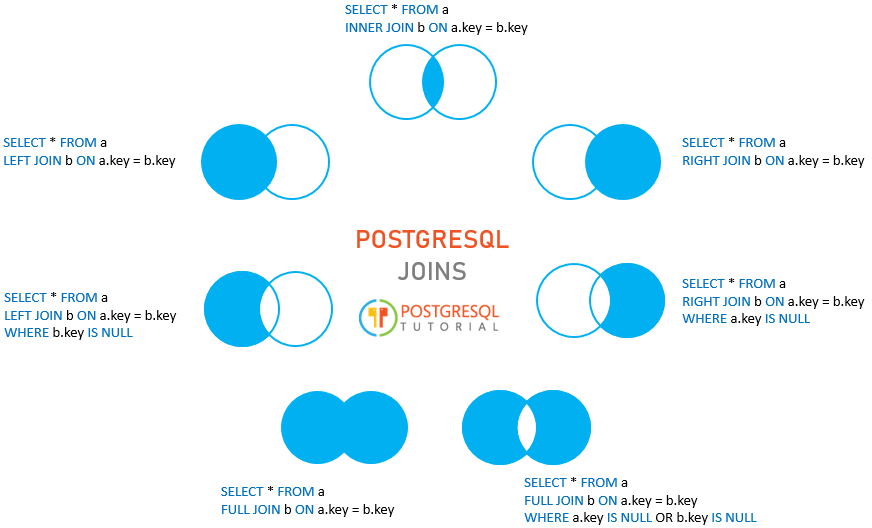
**PostgreSQL table aliases**

Table aliases temporarily assign tables new names during the execution of a query.

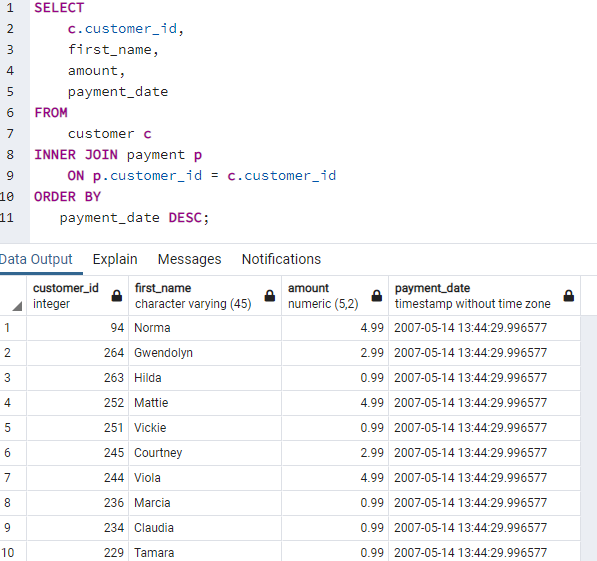
The following illustrates the syntax of a table alias

table\_name AS alias\_name;

Code language: SQL (Structured Query Language) (sql)

### **Using table aliases in join clauses**

To make the query shorter, you can use the table aliases for the table names listed on FROM and [INNER JOIN](https://www.postgresqltutorial.com/postgresql-inner-join/) clauses. For example



**PostgreSQL CROSS JOIN clause**

A CROSS JOIN clause allows you to produce a Cartesian Product of rows in two or more tables.

Different from other [join](https://www.postgresqltutorial.com/postgresql-joins/) clauses such as [LEFT JOIN](https://www.postgresqltutorial.com/postgresql-left-join/)  or [INNER JOIN](https://www.postgresqltutorial.com/postgresql-inner-join/), the CROSS JOIN clause does not have a join predicate.

Suppose you have to perform a CROSS JOIN of two tables T1 and T2.

If T1 has n rows and T2 has m rows, the result set will have nxm rows. For example, the T1 has 1,000 rows and T2 has 1,000 rows, the result set will have 1,000 x 1,000 = 1,000,000 rows.

The following illustrates the syntax of the CROSS JOIN syntax.

SELECT select\_list

FROM T1

CROSS JOIN T2;

The following statement is equivalent to the above statement:

SELECT select\_list

FROM T1, T2;

Also, you can use an INNER JOIN clause with a condition that always evaluates to true to simulate the cross join:

SELECT \*

FROM T1

INNER JOIN T2 ON true;

**PostgreSQL CROSS JOIN example**

The following [CREATE TABLE](https://www.postgresqltutorial.com/postgresql-create-table/) statements create T1 and T2 tables and [insert some sample data](https://www.postgresqltutorial.com/postgresql-insert/) for the cross demonstration.

DROP TABLE IF EXISTS T1;

CREATE TABLE T1 (label CHAR(1) PRIMARY KEY);

DROP TABLE IF EXISTS T2;

CREATE TABLE T2 (score INT PRIMARY KEY);

INSERT INTO T1 (label)

VALUES

('A'),

('B');

INSERT INTO T2 (score)

VALUES

(1),

(2),

(3);

The following statement uses the CROSS JOIN operator to join the table T1 with the table T2.

SELECT \*

FROM T1

CROSS JOIN T2;

label | score

-------+-------

A | 1

B | 1

A | 2

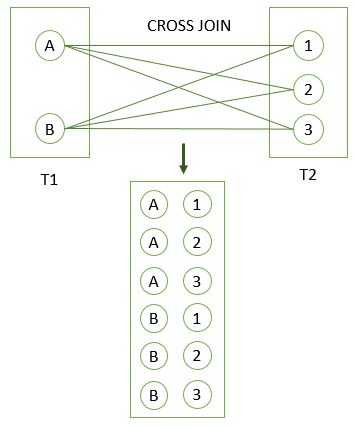
B | 2

A | 3

B | 3

(6 rows)

The following picture illustrates the result of the CROSS JOIN when joining the table T1 to the table T2:



**NATURAL JOIN Explained**

PostgreSQL NATURAL JOIN to query data from two or more tables.

A natural join is a join that creates an implicit join based on the same column names in the joined tables.

The following shows the syntax of the PostgreSQL natural join:

**SELECT select\_list**

**FROM T1 NATURAL [INNER, LEFT, RIGHT] JOIN T2;**

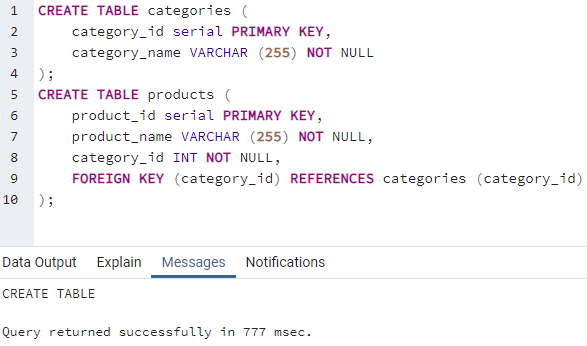
A natural join can be an [inner join](https://www.postgresqltutorial.com/postgresql-inner-join/), [left join](https://www.postgresqltutorial.com/postgresql-left-join/), or right join. If you do not specify a join explicitly e.g., INNER JOIN, LEFT JOIN, RIGHT JOIN, PostgreSQL will use the INNER JOIN by default.

If you use the asterisk (\*) in the select list, the result will contain the following columns:

**NATURAL JOIN examples**

To demonstrate the PostgreSQL natural join, we will create two tables: categories and products.

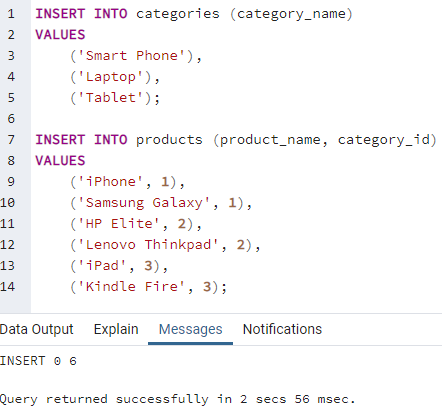
The following CREATE TABLE statements create the categories and products tables.



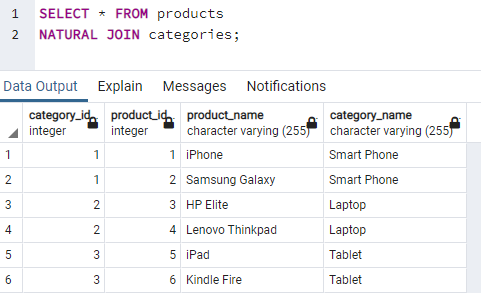
Each category has zero or many products and each product belongs to one and only one category.

The category\_id column in the products table is the [foreign key](https://www.postgresqltutorial.com/postgresql-foreign-key/) that references to the [primary key](https://www.postgresqltutorial.com/postgresql-primary-key/) of the categories table. The category\_id is the common column that we will use to perform the natural join.

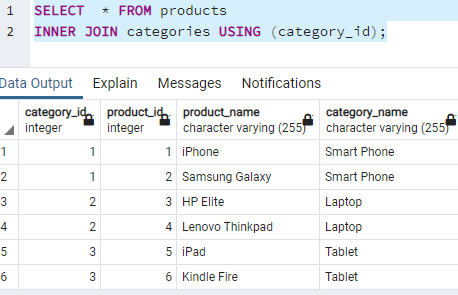
The following [INSERT](https://www.postgresqltutorial.com/postgresql-insert/) statements insert some data into the categories and products tables.



The following statement uses the NATURAL JOIN clause to join the products table with the categories table.



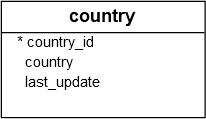
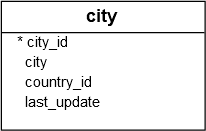
The above statement is equivalent to the following statement that uses the INNER JOIN clause.



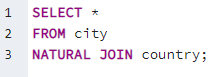
The convenience of the NATURAL JOIN is that it does not require you to specify the join clause because it uses an implicit join clause based on the common column.

However, you should avoid using the NATURAL JOIN whenever possible because sometimes it may cause an unexpected result.

For example, See the following city and country tables from the [sample database](https://www.postgresqltutorial.com/postgresql-sample-database/):



Both tables have the same country\_id column so you can use the NATURAL JOIN to join these tables as follows:



The query returns an empty result set.

The reason is that…

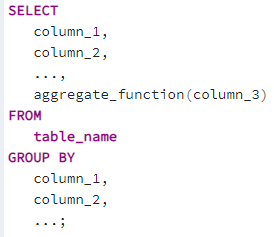
Both tables also have another common column called last\_update, which cannot be used for the join. However, the NATURAL JOIN clause just uses the last\_update column.

### **Grouping Data**

### **PostgreSQL GROUP BY clause**

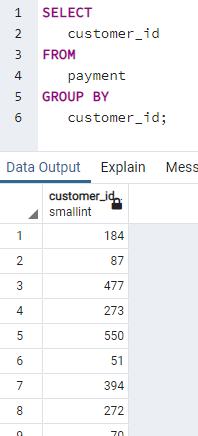
The GROUP BY clause divides the rows returned from the [SELECT](https://www.postgresqltutorial.com/postgresql-select/) statement into groups. For each group, you can apply an aggregate function e.g.,  [SUM()](https://www.postgresqltutorial.com/postgresql-sum-function/) to calculate the sum of items or [COUNT()](https://www.postgresqltutorial.com/postgresql-count-function/) to get the number of items in the groups.

The following statement illustrates the basic syntax of the GROUP BY clause:



### Using PostgreSQL GROUP BY without an aggregate function example

You can use the GROUP BY clause without applying an aggregate function. The following query gets data from the payment table and groups the result by customer id.



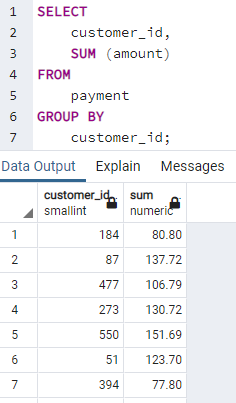
In this case, the GROUP BY works like the [DISTINCT](https://www.postgresqltutorial.com/postgresql-select-distinct/) clause that removes duplicate rows from the result set.

### **2) Using PostgreSQL GROUP BY with SUM() function example**

The GROUP BY clause is useful when it is used in conjunction with an [aggregate function](https://www.postgresqltutorial.com/postgresql-aggregate-functions/).

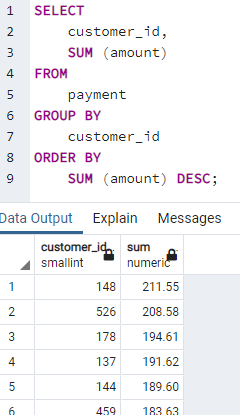
For example, to select the total amount that each customer has been paid, you use the GROUP BY clause to divide the rows in the payment table into groups grouped by customer id. For each group, you calculate the total amounts using the [SUM()](https://www.postgresqltutorial.com/postgresql-sum-function/) function.

The following query uses the GROUP BY clause to get total amount that each customer has been paid:



The GROUP BY clause sorts the result set by customer id and adds up the amount that belongs to the same customer. Whenever the customer\_id changes, it adds the row to the returned result set.

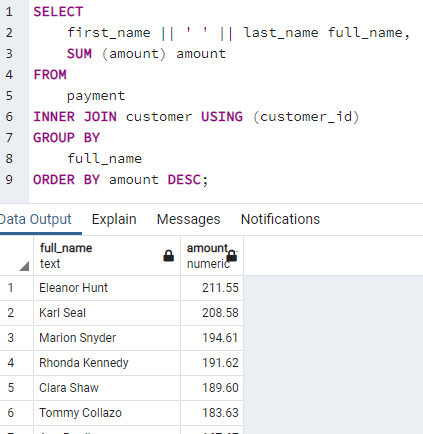
The following statement uses the [ORDER BY](https://www.postgresqltutorial.com/postgresql-order-by/) clause with GROUP BY clause to sort the groups.



### **Using PostgreSQL GROUP BY clause with the JOIN clause**

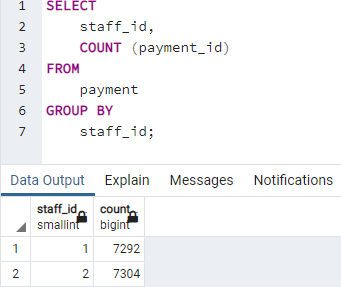
The following statement uses the GROUP BY clause with the [INNER JOIN](https://www.postgresqltutorial.com/postgresql-inner-join/) clause the get the total amount paid by each customer.

Unlike the previous example, this query joins the payment table with the customer table and group customers by their names.



### **Using PostgreSQL GROUP BY with COUNT() function example**

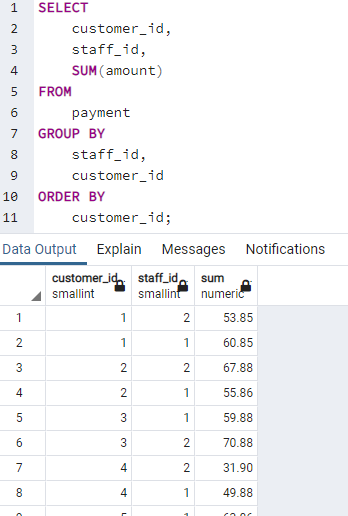
To find the number of payment transactions that each staff has been processed, you group the rows in the payment table by the values in the staff\_id column and use the [COUNT()](https://www.postgresqltutorial.com/postgresql-count-function/) function to get the number of transactions.



The GROUP BY clause divides the rows in the payment into groups and groups them by value in the staff\_id column. For each group, it returns the number of rows by using the [COUNT()](https://www.postgresqltutorial.com/postgresql-count-function/) function.

### **Using PostgreSQL GROUP BY with multiple columns**

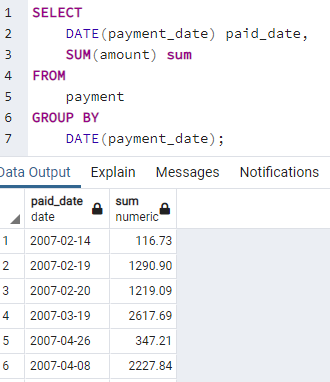
The following example uses multiple columns in the GROUP BY clause:



In this example, the GROUP BY clause divides the rows in the payment table by the values in the customer\_id and staff\_id columns. For each group of (customer\_id, staff\_id), the SUM() calculates the total amount.

### **Using PostgreSQL GROUP BY clause with date column**

The payment\_date is a timestamp column. To group payments by dates, you use the DATE() function to convert timestamps to dates first and then group payments by the result date.



## PostgreSQL HAVING clause

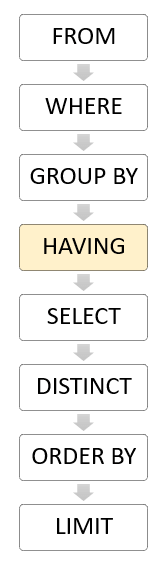
The HAVING clause specifies a search condition for a group or an aggregate. The HAVING clause is often used with the [GROUP BY](https://www.postgresqltutorial.com/postgresql-group-by/) clause to filter groups or aggregates based on a specified condition.

The following statement illustrates the basic syntax of the HAVING clause:

In this syntax, the group by clause returns rows grouped by the column1. The HAVING clause specifies a condition to filter the groups.

It’s possible to add other clauses of the SELECT statement such as [JOIN](https://www.postgresqltutorial.com/postgresql-joins/), [LIMIT](https://www.postgresqltutorial.com/postgresql-limit/), [FETCH](https://www.postgresqltutorial.com/postgresql-fetch/) etc.

PostgreSQL evaluates the HAVING clause after the FROM, [WHERE](https://www.postgresqltutorial.com/postgresql-where/), [GROUP BY](https://www.postgresqltutorial.com/postgresql-group-by/), and before the [SELECT](https://www.postgresqltutorial.com/postgresql-select/), [DISTINCT](https://www.postgresqltutorial.com/postgresql-select-distinct/), [ORDER BY](https://www.postgresqltutorial.com/postgresql-order-by/) and [LIMIT](https://www.postgresqltutorial.com/postgresql-limit/) clauses.



Since the HAVING clause is evaluated before the SELECT clause, you cannot use column aliases in the HAVING clause. Because at the time of evaluating the HAVING clause, the column aliases specified in the SELECT clause are not available.

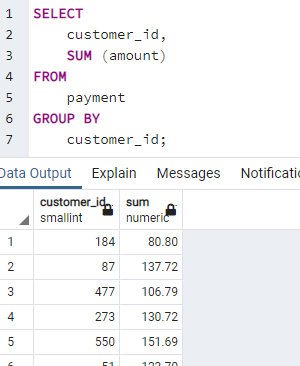
### **HAVING vs. WHERE**

The [WHERE](https://www.postgresqltutorial.com/postgresql-where/) clause allows you to filter rows based on a specified condition. However, the HAVING clause allows you to filter groups of rows according to a specified condition.

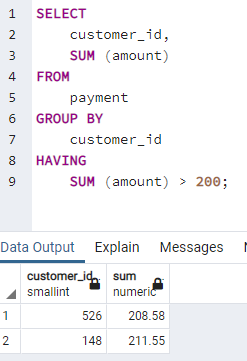
In other words, the WHERE clause is applied to rows while the HAVING clause is applied to groups of rows.

### **Using PostgreSQL HAVING clause with SUM function example**

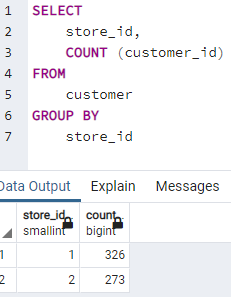
The following query uses the [GROUP BY](https://www.postgresqltutorial.com/postgresql-group-by/) clause with the [SUM()](https://www.postgresqltutorial.com/postgresql-sum-function/) function to find the total amount of each customer:



The following statement adds the HAVING clause to select the only customers who have been spending more than 200.



The following query uses the GROUP BY clause to find the number of customers per store.



The following statement adds the HAVING clause to select the store that has more than 300 customers.

