

Udemy Course Notes

Complete SQL Bootcamp

Table of Contents

[SELECT](#)

[SELECT DISTINCT](#)

[WHERE](#)

[PostgreSQL WHERE examples](#)

[LIMIT](#)

[IN Operator](#)

[PostgreSQL IN operator examples](#)

[NOT IN Operator](#)

[ORDER BY](#)

[PostgreSQL ORDER BY examples](#)

[BETWEEN](#)

[PostgreSQL BETWEEN operator examples](#)

[LIKE](#)

[GROUP BY](#)

[PostgreSQL GROUP BY with SUM function example](#)

[HAVING](#)

[Example](#)

[JOINS](#)

[SUBQUERY](#)

[CREATE TABLE and Constraints](#)

[PostgreSQL column constraints](#)

[PostgreSQL table constraints](#)




[PostgreSQL CREATE TABLE example](#)

SELECT

One of the most common tasks, when you work with PostgreSQL, is to query data from tables by using the `SELECT` statement. The `SELECT` statement is one of the most complex statements in PostgreSQL. It has many clauses that you can combine to form a powerful query.

Because of its complexity, we divide the PostgreSQL `SELECT` statement tutorial into many short tutorials so that you can learn each clause of the `SELECT` statement easier. The following are the clauses that appear in the `SELECT` statement:

- Select distinct rows by using `DISTINCT` operator.
- Filter rows by using `WHERE` clause.
- Sort rows by using the `ORDER BY` clause.
- Select rows based on various operators such as `BETWEEN`, `IN` and `LIKE`.
- Group rows into groups by using `GROUP BY` clause
- Apply condition for groups by using `HAVING` clause.
- Join to another table by using `INNER JOIN`, `LEFT JOIN`, `RIGHT JOIN` clauses.



Let's start with a basic form of the SELECT statement to query data from a table. The following illustrates the syntax of the SELECT statement:

```
1  SELECT column_1, column_2,  
  
2  FROM table_name
```

Let's examine the SELECT statement in more detail:

- First, you specify a list of columns in the table from which you want to query data in the SELECT clause. You use a comma between each column in case you want to query data from multiple columns. If you want to query data from all columns, you can use an asterisk (*) as the shorthand for all columns.
- Second, you indicate the table name after the FROM keyword

Notice that SQL language is case insensitive. It means if you use SELECT or select the effect is the same. By convention, we will use SQL keywords in uppercase to make the code easier to read and stand out clearly.

SELECT DISTINCT

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. You can use the DISTINCT clause on one or more columns of a table.

The syntax of DISTINCT clause is as follows:

```
1  SELECT DISTINCT column_1
2  FROM table_name;
```

If you specify multiple columns, the DISTINCT clause will evaluate the duplicate based on the combination of values of those columns.

```
1  SELECT DISTINCT column_1, column_2
2  FROM tbl_name;
```

PostgreSQL also provides the DISTINCT ON (expression) to keep the “first” row of each group of duplicates where the expression is equal. See the following syntax:

```
1  SELECT DISTINCT ON (column_1), column_2
2  FROM tbl_name
3  ORDER BY column_1, column_2;
```

The order of rows returned from the SELECT statement is unpredictable therefore the “first” row of each group of the duplicate is also unpredictable. It is good practice to

always use the `ORDER BY` clause with the `DISTINCT ON(expression)` to make the result obvious.

Notice that the `DISTINCT ON` expression must match the leftmost expression in the `ORDER BY` clause.

WHERE

The syntax of the PostgreSQL WHERE clause is as follows:

```
1  SELECT column_1, column_2 ... column_n
2  FROM table_name
3  WHERE conditions;
```

The WHERE clause appears right after the FROM clause of the SELECT statement. The conditions are used to filter the rows returned from the SELECT statement. PostgreSQL provides you with various standard operators to construct the conditions.

The following table illustrates the standard comparison operators.

OPERATOR	DESCRIPTION
=	Equal
>	Greater than
<	Less than
>=	Greater than or equal

<=	Less than or equal
<> or !=	Not equal
AND	Logical operator AND
OR	Logical operator OR

Let's practice with some examples of using the WHERE clause with conditions.


PostgreSQL WHERE examples

If you want to get all customers whose first names are Jamie, you can use the WHERE clause with the equal (=) operator as follows:

```
1  SELECT last_name, first_name
2
3  FROM customer
4
5  WHERE first_name = 'Jamie';
```

If you want to select the customer whose first name is Jamie and last names is rice, you can use the AND logical operator that combines two conditions as the following query:

```
1  SELECT last_name, first_name
2
3  FROM customer
4
5  WHERE first_name = 'Jamie' AND last_name = 'rice';
```



```
3 WHERE first_name = 'Jamie' AND
```

```
4 last_name = 'Rice';
```

If you want to know who paid the rental with amount is either less than 1USD or greater than 8USD, you can use the following query with OR operator:

```
1 SELECT customer_id,amount,payment_date
```

```
2 FROM payment
```

```
3 WHERE amount <= 1 OR amount >= 8;
```

LIMIT

PostgreSQL LIMIT is used in the SELECT statement to get a subset of rows returned by the query. The following is the common syntax of the LIMIT clause:

```
1  SELECT *  
  
2  FROM TABLE  
  
3  LIMIT n;
```

PostgreSQL returns n number of rows generated by the query. If n is zero or NULL, it produces the result that is same as omitting the LIMIT clause.

In case you want to skip a number of rows before returning n rows, you use OFFSET clause followed by the LIMIT clause as follows:

```
1  SELECT * FROM table  
  
2  LIMIT n OFFSET m;
```

PostgreSQL first skips m rows before returning n rows generated by the query. If m is zero, PostgreSQL will behave like without the OFFSET clause.

Because the order of the rows in the database table is unknown and unpredictable, when you use the LIMIT clause, you should always use the ORDER BY clause to control the order of rows. If you don't do so, you will get an unpredictable result.

IN Operator

You use the 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:

```
1  value IN (value1,value2,...)
```

The expression returns true if the value matches any value in the list i.e., value1, value2, etc. The list of values is not limited to a list of numbers or strings but also a result set of a SELECT statement as shown in the following query:

```
1  value IN (SELECT value FROM tbl_name);
```

The statement inside the parentheses is called a subquery, which is a query nested inside another query.

PostgreSQL IN operator examples

Suppose you want to know the rental information of customer id 1 and 2, you can use the IN operator in the WHERE clause as follows:

```
1  SELECT customer_id, rental_id, return_date
2
3  FROM rental
4
5  WHERE customer_id IN (1, 2)
```

```
4 ORDER BY return_date DESC;
```

NOT 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. The following statement selects rentals of customers whose customer id is not 1 or 2.

```
1 SELECT customer_id, rental_id, return_date
2 FROM rental
3 WHERE customer_id NOT IN (1, 2);
```

ORDER BY

When you query data from a table, PostgreSQL returns the rows in the order that they were inserted into the table. In order to sort the result set, you use the ORDER BY clause in the SELECT statement.


The ORDER BY clause allows you to sort the rows returned from the SELECT statement in ascending or descending order based on criteria specified by different criteria.

The following illustrates the syntax of the ORDER BY clause:

```
1  SELECT column_1,column_2
2  FROM tbl_name
3  ORDER BY column_1 ASC, column_2 DESC;
```

Let's examine the syntax of the ORDER BY clause in more detail:

- Specify the column that you want to sort in the ORDER BY clause. If you sort the result set by multiple columns, use a comma to separate between two columns.
- Use ASC to sort the result set in ascending order and DESC to sort the result set in descending order. If you leave it blank, the ORDER BY clause will use ASC by default.



Let's take some examples of using the PostgreSQL ORDER BY clause.

PostgreSQL ORDER BY examples

The following query sorts customers by the first name in ascending order:

```
1  SELECT first_name,last_name
2  FROM customer
3  ORDER BY first_name ASC;
```

BETWEEN

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

```
1  value BETWEEN low AND high;
```

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, or vice versa.

We can rewrite the BETWEEN operator by using the greater than or equal (\geq) or less than or equal (\leq) operators as the following statement:

```
1  value  $\geq$  low and value  $\leq$  high
```

If we want to check if a value is out of a range, we use the NOT BETWEEN operator as follows:

```
1  value NOT BETWEEN low AND high;
```

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

```
1  value < low OR value > high
```

We often use the BETWEEN operator in the WHERE clause of a SELECT, INSERT, UPDATE or DELETE statement.



PostgreSQL BETWEEN operator examples

Let's take a look at the payment table in the sample database.

The following query selects any payment whose amount is between 8 and 9:

```
1  SELECT customer_id, payment_id, amount
2  FROM payment
3  WHERE amount BETWEEN 8 AND 9;
```

LIKE


Suppose the store manager asks you find a customer that he does not remember the name exactly. He just remembers that customer's first name begins with something like Jen. How do you find the exact customer that the store manager is asking? You may find the customer in the customer table by looking at the first name column to see if there is any value that begins with Jen. It is kind of tedious because there many rows in the customer table.

Fortunately, you can use the PostgreSQL LIKE operator to as the following query:

```
1  SELECT first_name,last_name
2  FROM customer
3  WHERE first_name LIKE 'Jen%';
```

Notice that the WHERE clause contains a special expression: the first_name, the LIKE operator and a string that contains a percent (%) character, which is referred as a *pattern*.

The query returns rows whose values in the first name column begin with Jen and may be followed by any sequence of characters. This technique is called pattern matching.



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

- Percent (%) for matching any sequence of characters.
- Underscore (_) for matching any single character.

GROUP BY

The GROUP BY clause divides the rows returned from the SELECT statement into groups. For each group, you can apply an aggregate function e.g., to calculate the sum of items or count the number of items in the groups.


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

```
1  SELECT column_1, aggregate_function(column_2)
2  FROM tbl_name
3  GROUP BY column_1;
```

The GROUP BY clause must appear right after the FROM or WHERE clause. Followed by the GROUP BY clause is one column or a list of comma-separated columns. You can also put an expression in the GROUP BY clause.

PostgreSQL GROUP BY with SUM function example

The GROUP BY clause is useful when it is used in conjunction with an aggregate function. For example, to get how much a customer has been paid, you use the GROUP BY clause to divide the payments table into groups; for each group, you calculate the total amounts of money by using the SUM function as the following query:



```
1  SELECT customer_id,  
2  SUM (amount)  
3  FROM payment  
4  GROUP BY customer_id;
```

HAVING

We often use the HAVING clause in conjunction with the GROUP BY clause to filter group rows that do not satisfy a specified condition.

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

```
1  SELECT column_1, aggregate_function (column_2)
2  FROM tbl_name
3  GROUP BY column_1
4  HAVING condition;
```

The HAVING clause sets the condition for group rows created by the GROUP BY clause after the GROUP BY clause applies while the WHERE clause sets the condition for individual rows before GROUP BY clause applies. This is the main difference between the HAVING and WHERE clauses.

In PostgreSQL, you can use the HAVING clause without the GROUP BY clause. In this case, the HAVING clause will turn the query into a single group. In addition, the SELECT list and HAVING clause can only refer to columns from within aggregate functions. This kind of query returns a single row if the condition in the HAVING clause is true or zero row if it is false.

Example

You can apply the HAVING clause to select the only customer who has been spending more than 200 as the following query:

```
1  SELECT customer_id,  
2  SUM (amount)  
3  FROM payment  
4  GROUP BY customer_id  
5  HAVING SUM (amount) > 200;
```

JOINS

A full review of SQL JOINS is available online here:

<https://medium.com/@josemarcialportilla/review-of-sql-joins-ac5463dc71c9#.ayjcuatvj>



SUBQUERY

A subquery is a query nested inside another query such as SELECT, INSERT, DELETE and UPDATE. In this tutorial, we are focusing on the SELECT statement only.

To construct a subquery, we put the second query in brackets and use it in the WHERE clause as an expression:

```
1  SELECT film_id, title, rental_rate
2  FROM film
3  WHERE rental_rate > (
4    SELECT AVG (rental_rate)
5    FROM film );
```

The query inside the brackets is called a **subquery or an inner query**. The query that contains the **subquery is known as an outer query**.

PostgreSQL executes the query that contains a subquery in the following sequence:

- First, executes the subquery.
- Second, gets the result and passes it to the outer query.
- Third, executes the outer query.

CREATE TABLE and Constraints

To create a new table in PostgreSQL, you use the CREATE TABLE statement. The following illustrates the syntax of the CREATE TABLE statement:

```
1  CREATE TABLE table_name (  
2  column_name TYPE column_constraint,  
3  table_constraint table_constraint  
4  ) INHERITS existing_table_name;
```

Let's examine the syntax of the CREATE TABLE statement in more detail.

- First, you specify the name of the new table after the CREATE TABLE clause. The TEMPORARY keyword is for creating a temporary table, which we will discuss in the temporary table tutorial.
- Next, you list the column name, its data type, and column constraint. You can have multiple columns in a table, each column is separated by a comma (,). The column constraint defines the rules for the column e.g., NOT NULL.
- Then, after the column list, you define a table-level constraint that defines rules for the data in the table.
- After that, you specify an existing table from which the new table inherits. It means the new table contains all columns of the existing table and the

columns defined in the CREATE TABLE statement. This is a PostgreSQL's extension to SQL.

PostgreSQL column constraints

The following are the commonly used column constraints in PostgreSQL:

- **NOT NULL** – the value of the column cannot be NULL.
- **UNIQUE** – the value of the column must be unique across the whole table.
However, the column can have many NULL values because PostgreSQL treats each NULL value to be unique. Notice that SQL standard only allows one NULL value in the column that has the UNIQUE constraint.
- **PRIMARY KEY** – this constraint is the combination of NOT NULL and UNIQUE constraints. You can define one column as PRIMARY KEY by using column-level constraint. In case the primary key contains multiple columns, you must use the table-level constraint.
- **CHECK** – enables to check a condition when you insert or update data. For example, the values in the price column of the product table must be positive values.
- **REFERENCES** – constrains the value of the column that exists in a column in another table. You use REFERENCES to define the foreign key constraint.



PostgreSQL table constraints

The table constraints are similar to column constraints except that they are applied to the entire table rather than to an individual column.

The following are the table constraints:

- **UNIQUE (column_list)**– to force the value stored in the columns listed inside the parentheses to be unique.
- **PRIMARY KEY(column_list)** – to define the primary key that consists of multiple columns.
- **CHECK (condition)** – to check a condition when inserting or updating data.
- **REFERENCES**– to constrain the value stored in the column that must exist in a column in another table.

PostgreSQL CREATE TABLE example

We will create a new table named `account` that has the following columns with the corresponding constraints:

- `user_id` – primary key
- `username` – unique and not null
- `password` – not null
- `email` – unique and not null
- `created_on` – not null

- last_login – null

The following statement creates the account table:

```
1  CREATE TABLE account(  
2  user_id serial PRIMARY KEY,  
3  username VARCHAR (50) UNIQUE NOT NULL,  
4  password VARCHAR (50) NOT NULL,  
5  email VARCHAR (355) UNIQUE NOT NULL,  
6  created_on TIMESTAMP NOT NULL,  
7  last_login TIMESTAMP);
```

