[Team]
Team LinkedIn Account
[Ahmed M.Osman] [Link]
[Khaled Abdelrahman] [Link]
[Ibrahim Hafez] [Link]
[Mohamed Adel] [Link]
[fatma yasser] [Link]

The SQL SELECT DISTINCT Statement

The `SELECT DISTINCT` statement is used to return only distinct (different) values.

Inside a table, a column often contains many duplicate values; and sometimes you only want to list the different (distinct) values.

SELECT DISTINCT Syntax

```
SELECT DISTINCT column1, column2, ...
FROM table_name;
```

Demo Database

Below is a selection from the "Customers" table in the Northwind sample database:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
5	Berglunds snabbköp	Christina Berglund	Berguvsväg en 8	Luleå	S-958 22	Sweden

SELECT Example Without DISTINCT

The following SQL statement selects all (including the duplicates) values from the "Country" column in the "Customers" table:

Example

```
SELECT Country FROM Customers;
```

Now, let us use the `SELECT DISTINCT` statement and see the result.

SELECT DISTINCT Examples

The following SQL statement selects only the DISTINCT values from the "Country" column in the "Customers" table:

Example

```
SELECT DISTINCT Country FROM Customers;
```

The following SQL statement lists the number of different (distinct) customer countries:

Example

```
SELECT COUNT(DISTINCT Country) FROM Customers;
```

So, using DISTINCT would just neglect the duplicates and show you all the countries listed just for one time, instead of filling the whole page with duplicates.

Ibrahim Hafez

LinkedIn: https://www.linkedin.com/in/ibrahim-hafez/

Khaled Abdelrahman

LinkedIn: https://www.linkedin.com/in/khaled-abdelrahman-350493210/

PRACTICAL:

functionality to add a student to a session from scratch:

- In src\models\students.ts:

```ts
import client from '../database'
import bcrypt from 'bcrypt'
import config from '../config'

const salt_rounds  = config.salt, pepper = config.pepper


export type student = {
    id?: number
    first_name: string
    last_name: string
    password: string
}

export const hashing = (password: string) => {
    const salt = parseInt(salt_rounds as unknown as string, 10)
    return bcrypt.hashSync(`${password}${pepper}`, salt)
}
```

```
//add student
    async add(
        s: student
    ): Promise<{ id: number; first_name: string; last_name: string }> {
        try {
            const sql = ‘INSERT INTO students (first_name, last_name,
password) VALUES($1, $2, $3) RETURNING id, first_name, last_name ;’
            const conn = await client.connect()

            const hash = hashing(s.password)

            const result = await conn.query(sql, [
                s.first_name,
                s.last_name,
                hash,
            ])

            conn.release()
            return result.rows[0]
        } catch (err) {
            throw new Error(
                `Could not add new student ${s.first_name} ${s.last_name}.
Error: ${err}`
            )
        }

    }
```

· In src\handlers\students_handler.ts :

```
import { NextFunction, Request, Response } from 'express'

import { studentModel } from '../models/students'

import jwt from 'jsonwebtoken'

import config from '../config'

import errorMiddleWare from '../middleware/errorMiddleware'


const student_model = new studentModel()




export const add = async (

    req: Request,

    res: Response,

    next: NextFunction

) => {

    try {

        const newstudent = await student_model.create(req.body)

        const token = jwt.sign(

            { student: newstudent },

            config.token as unknown as string

        )
        res.json({

            status: 'success',

            student: newstudent,

            token: token,

            message: 'Student added Successfully',

        })

    } catch (error) {

        next(errorMiddleWare(error as Error, req, res))

    }

}
```

· In src\routes\api\students_route.ts :

```
import { Router } from 'express'


import * as handler from '../../handlers/students_handler'


const routes = Router()


routes.route('/').post(handler.add)


export default routes
```

THEORETICAL:

Explain how to get a unique list of rows in a SELECT statement

- We can use ``` SELECT DISTINCT ``` query to get only unique values in a column
- Note that ```DISTINCT``` only operates on a single column.

EXAMPLE:

```
SELECT DISTINCT category FROM products
```

PostgreSQL – SELECT DISTINCT clause

This article will be focusing on the use of **SELECT statement with the DISTINCT clause** to remove duplicates rows from a result set of query data.

Removing duplicate rows from a query result set in PostgreSQL can be done using the SELECT statement with the DISTINCT clause. It keeps one row for each group of duplicates. The DISTINCT clause can be used for a single column or for a list of columns.

If you desire to operate on a list of columns the syntax will somewhat be like below:

Syntax:*SELECT DISTINCT column_1, column_2, column_3 FROM table_name;*

Now, let's look into a few examples for better understanding. For the sake of example, we will create a sample database as explained below:

Create a database(say, Favourite_colours) using the commands shown below:

```
CREATE DATABASE Favourite_colours;
```

Now add a table(say, my_table) with columns(say, id, colour_1 and colour_2) to the database using the command below:

```
CREATE TABLE my_table(
    id serial NOT NULL PRIMARY KEY,
    colour_1 VARCHAR,
    colour_2 VARCHAR
);
```

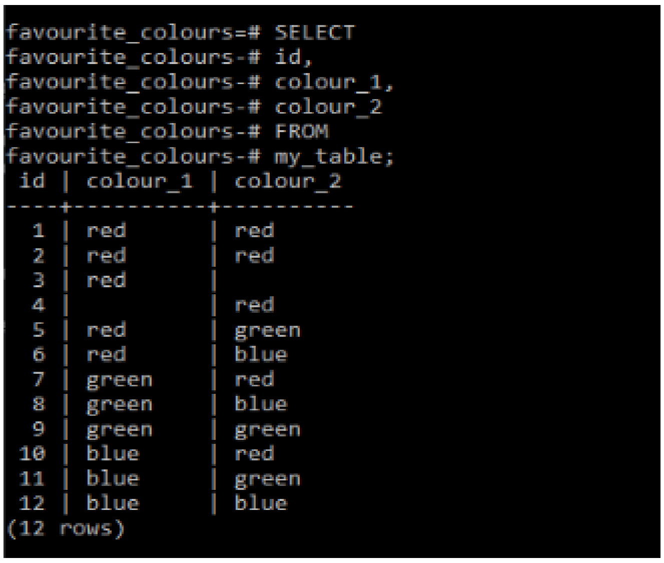
Now insert some data in the table that we just added to our database using the command below:

```
INSERT INTO my_table(colour_1, colour_2)
VALUES
    ('red', 'red'),
    ('red', 'red'),
    ('red', NULL),
    (NULL, 'red'),
    ('red', 'green'),
    ('red', 'blue'),
    ('green', 'red'),
    ('green', 'blue'),
    ('green', 'green'),
    ('blue', 'red'),
    ('blue', 'green'),
    ('blue', 'blue');
```

Now check if everything is as intended by making a query as below:

```
SELECT
    id,
    colour_1,
    colour_2
FROM
    my_table;
```

If everything is as intended, the output will be like as shown below:



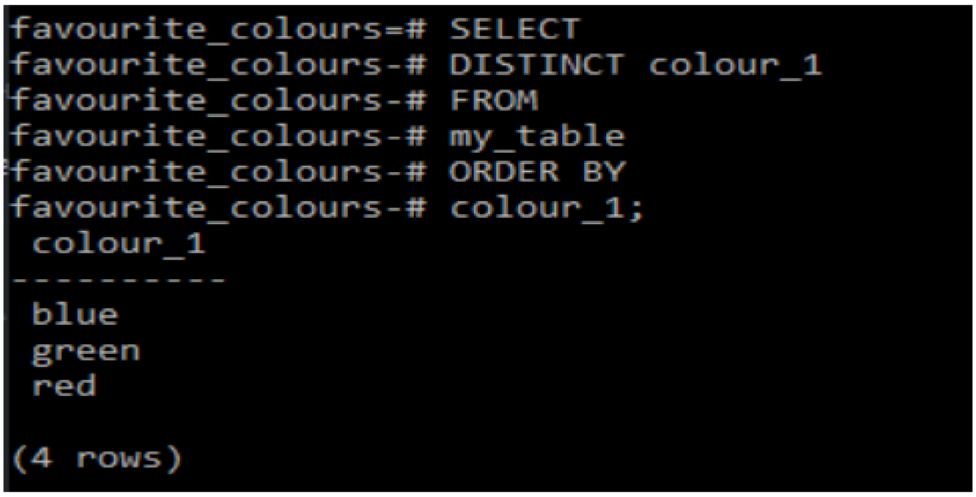
```
favourite_colours=# SELECT
favourite_colours-# id,
favourite_colours-# colour_1,
favourite_colours-# colour_2
favourite_colours-# FROM
favourite_colours-# my_table;
 id | colour_1 | colour_2
----+----------+----------
  1 | red      | red
  2 | red      | red
  3 | red      |
  4 |          | red
  5 | red      | green
  6 | red      | blue
  7 | green    | red
  8 | green    | blue
  9 | green    | green
 10 | blue     | red
 11 | blue     | green
 12 | blue     | blue
(12 rows)
```

Since, our database is good to go, we move onto the implementation of the SELECT DISTINCT clause.

Example 1:
PostgreSQL DISTINCT on one column

```
SELECT
    DISTINCT colour_1
FROM
    my_table
ORDER BY
    colour_1;
```

Output:



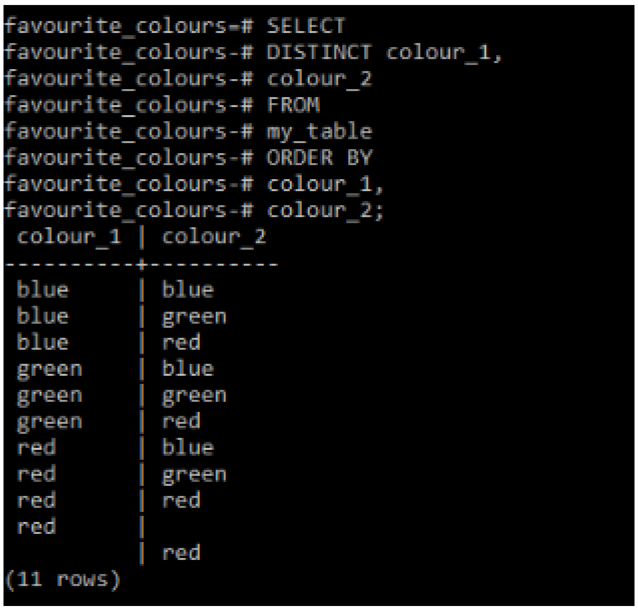
```
favourite_colours=# SELECT
favourite_colours-# DISTINCT colour_1
favourite_colours-# FROM
favourite_colours-# my_table
favourite_colours-# ORDER BY
favourite_colours-# colour_1;
 colour_1
----------
 blue
 green
 red

(4 rows)
```

Example 2:
PostgreSQL DISTINCT on multiple columns

```
SELECT
    DISTINCT colour_1,
    colour_2
FROM
    my_table
ORDER BY
    colour_1,
    colour_2;
```

Output:



```
favourite_colours-# SELECT
favourite_colours-# DISTINCT colour_1,
favourite_colours-# colour_2
favourite_colours-# FROM
favourite_colours-# my_table
favourite_colours-# ORDER BY
favourite_colours-# colour_1,
favourite_colours-# colour_2;
 colour_1 | colour_2
----------+----------
 blue     | blue
 blue     | green
 blue     | red
 green    | blue
 green    | green
 green    | red
 red      | blue
 red      | green
 red      | red
 red      |
          | red
(11 rows)
```

What is the COUNT function in SQL?

This SQL function will return the count for the number of rows for a given group.

Here is the basic syntax:

```
SELECT COUNT(column_name) FROM table_name;
```

The `SELECT` statement in SQL tells the computer to get data from the table.

`COUNT(column_name)` will not include `NULL` values as part of the count.
A `NULL` value in SQL is referring to values that are not present in the table.
Sometimes you can use an `*` inside the parenthesis for the `COUNT` function.

```
SELECT COUNT(*) FROM table_name;
```

The `COUNT(*)` function will return the total number of items in that group including `NULL` values.
The `FROM` clause in SQL specifies which table we want to list.
You can also use the `ALL` keyword in the `COUNT` function.

```
SELECT COUNT(ALL column_name) FROM table_name;
```

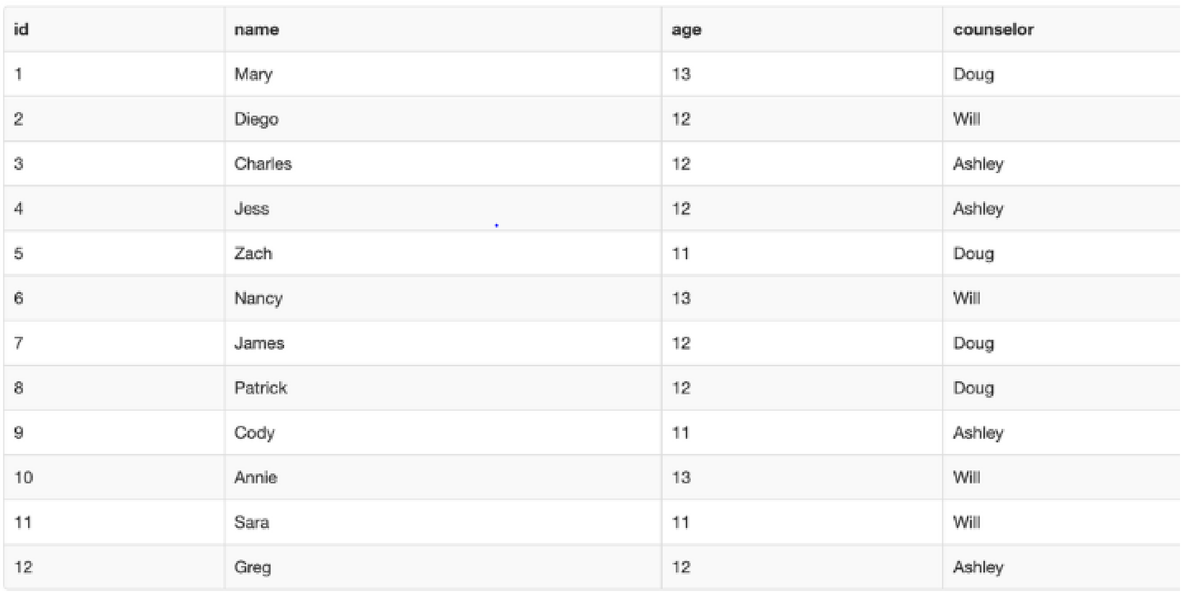
The `ALL` keyword will count all values in the table including duplicates. You can omit this keyword because the `COUNT` function uses the `ALL` keyword as the default whether you write it or not.
Sometimes you will see the `DISTINCT` keyword used with the `COUNT` function.

```
SELECT COUNT(DISTINCT column_name) FROM table_name;
```

The `DISTINCT` keyword will only count unique values that are `NOT NULL`. The computer will ignore any duplicate values.

How to use the COUNT function in SQL

In this example, we have a table for young campers with the columns of `id`, `name`, `age` and `counselor`.



id	name	age	counselor
1	Mary	13	Doug
2	Diego	12	Will
3	Charles	12	Ashley
4	Jess	12	Ashley
5	Zach	11	Doug
6	Nancy	13	Will
7	James	12	Doug
8	Patrick	12	Doug
9	Cody	11	Ashley
10	Annie	13	Will
11	Sara	11	Will
12	Greg	12	Ashley

If we want to select all of the rows in our table, then we can use the following syntax:

```
SELECT COUNT(*) FROM campers;
```

As you can see, the query returned the number 12 which represents the total number of rows in our `campers` table.

Using the **`WHERE`** clause

We can use the `WHERE` clause to specify the number of rows for the name of a particular camp counselor.
In this example, we want to count the number of rows for the camp counselor by the name of Ashley.

In the `WHERE` clause, we need to specify `counselor` with a value of `"Ashley"`.

```
WHERE counselor="Ashley";
```

This is the complete code:

```
SELECT COUNT(*) FROM campers WHERE counselor="Ashley";
```

How to use the **`ORDER BY`** clause

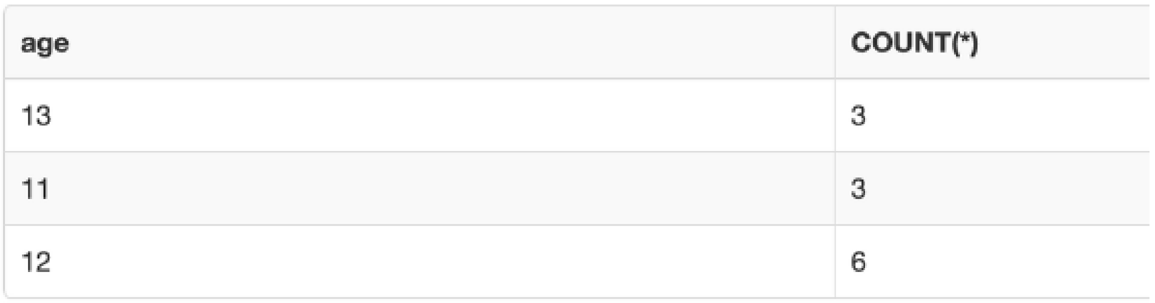
We can modify our example for the list of ages and use the `ORDER BY` clause to list the results from smallest to largest.
This is the code for the `ORDER BY` clause:

```
ORDER BY COUNT(*);
```

We add that clause at the end of the `SELECT` statement like this:

```
SELECT age, COUNT(*) FROM campers GROUP BY age ORDER BY COUNT(*);
```

This is what the modified example looks like:



age	COUNT(*)
13	3
11	3
12	6

If we wanted the count results to be sorted from largest to smallest, then we can use the `DESC` keyword.
This is the code for the `ORDER BY` clause using the `DESC` keyword:

```
ORDER BY COUNT(*) DESC;
```

This is the complete code:

```
SELECT age, COUNT(*) FROM campers GROUP BY age ORDER BY COUNT(*) DESC;
```

This is what the new result would look like:

age	COUNT(*)
12	6
13	3
11	3

Conclusion

In SQL, you can make a database query and use the `COUNT` function to get the number of rows for a particular group in the table.
Here is the basic syntax:

```
SELECT COUNT(column_name) FROM table_name;
```

`COUNT(column_name)` will not include `NULL` values as part of the count.
A `NULL` value in SQL refers to values that are not present in the table.
Sometimes you can use an `*` inside the parenthesis for the `COUNT` function.

```
SELECT COUNT(*) FROM table_name;
```

The `COUNT(*)` function will return the total number of items in that group including `NULL` values.