commands that you need to know.

Felipe Florencio Garcia Follow
Apr 13 · 15 min read

Here we will go through most important and basic commands that you should know in order to create, manage and delete your data on your database, beside the commands here i'm using the PostgreSQL syntax, most relational database share the same concepts.

Learning this mean that you will be able to complete manage and create your database.

This are the commands that you can expect learn from here:

- **Create Table**
- **Insert**
- **Alter Table**
- **Update**
- **Upsert**
- **Select**
- **Where**
- **Join**
- **Union**
- **Order**
- **Alias**
- **Delete**
- **Drop Table**

In order to make “live samples” here we are using PostgreSQL database, so you may want to look on how to configure postgresql on Mac.

We will try to follow all the “path” that we would maybe do on a real situation, the first one for sure is the “*create*” command.

Create Table

Let’s create our first customer:

```
CREATE TABLE customer (
        name TEXT,
        age INTEGER,
        email CHARACTER(255),
        address CHARACTER(400),
        zip_code CHARACTER(20)
);
```

Pretty simples right, let’s dig into by step:

> Create the table with a name `CREATE TABLE <NAME>(` that for us is "*customer*";
>
> Add the *columns* and the appropriate *data type* like `email CHARACTER(255),`

It’s simple, we will not dig into the *data type* that we could use, as is long and can change from database to another database, the idea is learn the principle.

Insert

Now moving on, we already have our first table, and now you want to add some information, otherwise what’s the purpose, for this you will run this command:

```
INSERT INTO customer(name, age) VALUES('Felipe', 31);
```

The code speak by himself, we want to *add* `<INSERT INTO>` our table called "*customer*" for the column with key name "*name*" and "*age*" the `<VALUES>` following the order, name " `Felipe` " and the age `31` .

Alter Table

It's pretty normal that you are creating your database and you start with the basic, and start improve, and with command you can alter modify the table structure. But of course is not unlimited if you try to alter some data that already exist and could cause conflict like change from *integer* to *string*.

```
ALTER TABLE customer ADD COLUMN height INTEGER;
```

As you can see we are adding a new `COLUMN` with the name `height` and the type `INTEGER` .

This command have a lot of other properties, so you can take a look at the documentation in order to see all other variations.

Update

We added some data inside, but let's consider that you did a mistake, or, actually that you want to update just, because this is the idea, be able to hold data and manipulate.

```
UPDATE customer SET height = 0;
```

This will be useful right? As we just altered our table, and we do not want to have empty value by any reason, so with this command we `UPDATE` our customer table and `SET` the height for everyone to be "*0*", like a default value.

But I do not want to do this to everyone always of course, so next is more specific.

```
UPDATE customer SET height = 60, age = 101 WHERE name =
'Felipe';
```

We did the same as before, the difference here that we use a new clause `WHERE` to specify who / what we want to change, I will talk latter more about this, for now it's enough to you understand that using this clause I validate if "*name*" match the name, and only if is true I update the values.

And this kind of validation it's really important, and because this here another way of doing with even more "*validations*".

```
UPDATE customer SET height = 60, age = 101 WHERE (name =
'Felipe' AND zip_code = '2323LL') OR email =
'felipe@gmail.com';
```

We have two important differences where, let's see:

> Our `WHERE` clause now have another clause called `AND`, and this allow us to "*concatenate*" two values that we want to check, we area saying that both need to fit in order to pass our *validation*. It's important to notice that they are inside "*( )*", this is to kind of "*hold*" as one value or *constraint*, basically we check what is inside that will give the "*output*" that will be *true* or *false*.
>
> The second one is `OR` and is exactly what the "*name*" suggest, if our `WHERE` clause does not fit we check for this one.

Upsert

Beside the name looks weird, it's very useful this command, as we are talking about update, one of most common scenarios is you want to update something that already exist, but how do you know in upfront that already exist? For this usually you can think, first check if exist, and if so you use the update command right? But actually you already know what you want, you want that if that value is already there update.

PostgreSQL implement this not long ago, as other database's already had this for quite long time, let's see how to implement for using Postgres.

First you need need to know that we have 2 constraint's that we can use in order to "say" what we want to do in case that value is already there.

ON CONFLICT DO NOTHING

ON CONFLICT DO UPDATE

By the name you already can imagine what happen, on update if find a "conflict" other one just don't do nothing.

Let's look how to use `ON CONFLICT DO NOTHING` :

```
INSERT INTO customer (name, height) VALUES ('Felipe', 60) ON
CONFLICT DO NOTHING;
```

Basically what's gonna happen is nothing, if the database find's that already have the value just don't do anything.

On other hand for `ON CONFLICT DO UPDATE` :

```
INSERT INTO customer (name, height) VALUES ('Felipe', 60) ON
CONFLICT DO UPDATE;
```

Here is exactly what the name say, if find any conflict, just update.

It's simple use this scenario when you do know that you want to update when have the same value, just remember that the implementation change a little for each database, but, important to know the concept, just look at internet the specific constraint name for your database.

Select

Now that we basically did the beginning following a logical way like create your database, update, delete, most of the bases in order to have your database setup, we need to get the data.

Most probably this will be one of the most used commands by you, so let's see it.

```
SELECT * FROM customer;
```

The basic is very simple, let's go step by step:

`SELECT *` - What is doing is saying "I want" (`SELECT`) everything (`*`);

`FROM` - From where I want (`FROM`) that is from table name (`customer`).

Here i'm getting everything that is inside my table customer, let's be more specific.

```
SELECT height, name FROM customer;
```

Now that you already understand how is created you probably understood that we want specific datas, that is the data from columns "*height*" and "*name*" from my table *customer*.

This constraint have many other possibilities you definitely need to look into the documentation in order to be able to see all other variations.

One more, is for the situation that you actually don't want to SELECT by the column name but if match the value, for this try this:

```
SELECT height, 'Felipe' FROM customer;
```

Like I said, we will look for data inside column "*height*" and all values that match the string "*Felipe*" beside match column name.

Where

This constraint it's another one that you will use a lot even more together with the *SELECT* constraint, let's see how do we use, you will use this for filter what do you want when selecting your data.

```
SELECT height, name FROM customer WHERE height > 50;
```

As you can see, we are looking for *customer height* and *name* `WHERE` the "*height*" is bigger than *50*.

But you can add even more constraints (*filters*).

```
SELECT height, name FROM customer WHERE height > 50 AND
height < 80;
```

We added here a new constraint `AND` that do exactly that the name suggest, that our height need to be higher than 50 *AND* lower **than 80.

Another constraint that we can use is OR.

```
SELECT height, name FROM customer WHERE height > 50 OR name
= 'Felipe';
```

Here we added another constraint `OR` that means if our "*previous*" check do not fit as the name suggest "*OR*" the name is *equal* we return as a match.

Join

This is definitely a very important topic, not only because in a job interview they will ask you for sure about, but because there's a lot of possibilities. Even though the first time that you try to understand looks weird.

For this we have 5 types:

CROSS JOIN;

INNER JOIN;

LEFT JOIN;

RIGHT JOIN;

FULL JOIN;

I will try explain the easiest way, but I do recommend you try this live, when you do this you will understand better and see how easy it's.

For this we do need to have some data in order to be able to visualise, for this I will put a data to be inserted, I accept that you already know how to create a table. We will have our *customer* and the *contact* table.

Our customer data:

```
INSERT INTO customer(name,age,email,address,zip_code)
VALUES
('Paul',23,'paul@gmail.com','address from paul','2321LL'),
('Felipe',32,'felipegarcia@gmail.com','address from
felipe','3413MS'),
('Teddy',90,'teddy@gmail.com','address from
teddy','3423PO'),
('Mark',17,'mark@gmail.com','address from mark','9423MA'),
('David',35,'david@gmail.com','address from
david','2341DA'),
('Allen',56,'allen@gmail.com','address from
allen','3423PO'),
('James',56,'james@gmail.com','address from
james','3423PO');
```

And for our contact data:

```
INSERT INTO contact(email,zip_code)
VALUES
('teddy@gmail.com','3423PO'),
('david@gmail.com','2341DA'),
('james@gmail.com','3234PO'),
('felipe@gmail.com','');
```

You need just to run this on your already created table and will insert.

Cross Join

I would say that you will not use too much this one, as can generate a lot of data, basically what he does here is join both tables and generate an output with all data.

The way that this constraint works is that he will return a *cartesian product* data between the two table, this command will *multiply* the first table with the *amount* of data on the second table, will return ever possible combination. I believe you can imagine how much data this can create right?!

It's easy to see the representation of how this query will be handle from this image, but it's not only match:

Cross Join

Let's go to the command:

```
SELECT * FROM customer CROSS JOIN contact;
```

I will not add a print of the full output here as is too big, but running this you will generate a total of 36 items / rows, that is the multiplication between the first table *customer* that has 9 items multiplied by the *contact* table that has 4.

But one valuable information here is that with this we will generate a output that has all the rows from the first table and all rows from the second:

Cross Join select everything

And why repeat table? Well, we `SELECT *` , we chose to get all data from both basically, but let's show them how to not repeat, or better, how to chose which output row we want.

Use this query:

```
SELECT customer."name", customer.age, customer.address,
contact.email, contact.zip_code FROM customer CROSS JOIN
contact;
```

Part of the output:

If you pay attention, what we did was say how will be output, by specify which row we want and from where like `customer."name"` and `contact.zip_code` so the output will follow this sequence order.

I did not add the “end” of this query, and even the previous one, but let talk about the “*join*” concept, basically we are looking for a match, but for this constraint, basically “*get all*” from one table and “*all from*” second table, but in the type *CROSS* if some item does not match will still return but where there’s no match will be as empty value.

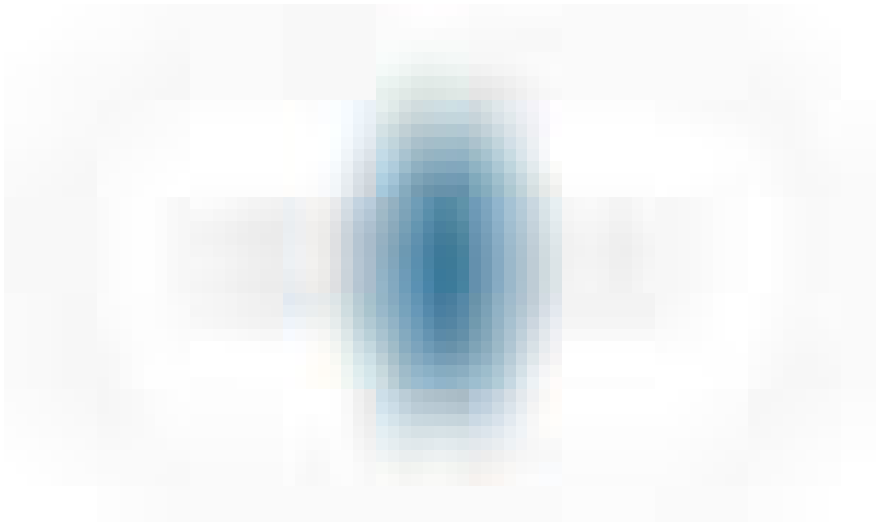
Because if you pay attention our *contact* table for the user “*felipe@gmail.com”* there’s no match with any one at *customer* table, but still will be returned.

Inner Join

Now that we already know some basic thing regarding what we can do regarding *join*, let’s move to the next one, that looks very similar to the

cross join but with one simple difference that here will only return matches.

This is the representation of the kind of join that we are doing here between tables, as you see only the "*centre*" that is what match that we will have as data.



Inner Join

Here the syntax is a little difference we have to add one more *constraint*, that is the one that will look for the "*validator*" row.

```
SELECT * FROM customer INNER JOIN contact ON
customer.zip_code=contact.zip_code;
```

The main difference that we have here is, we will look for a match using between table *customer* and table *contact*, for this we will use the row "*zip_code*".

This is what you will have as retrieve data.

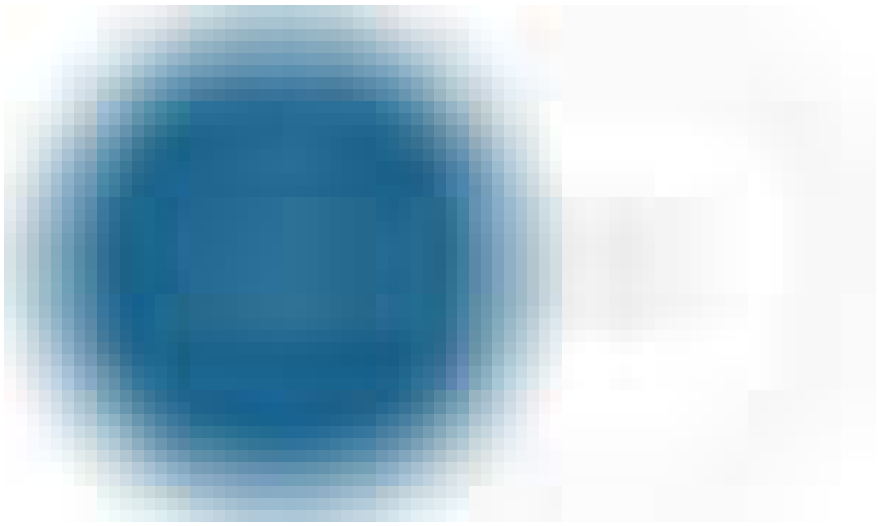
It's easy to see as we return all cells we have the cell *zip_code* from *customer* and from *contact*, all them match.

This join is very useful in order to retrieve data that you want to match by some specific value, it's different from have relationship.

Left Join

As you may have notice, the main goal is match data, and if the data match we return that row.

You can see here that we have in the centre the *join* part, that is the validator, and the left part that mean that if the join does not match that data will be returned but with the row for the data that did not match "*empty*".



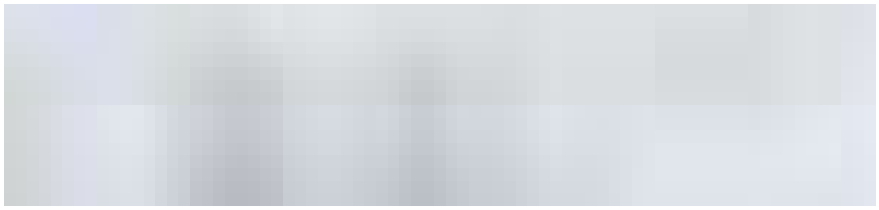
Left Join

But here now we start doing different, let's see how it's this query:

```
SELECT * FROM customer LEFT JOIN contact ON
customer.zip_code=contact.zip_code;
```

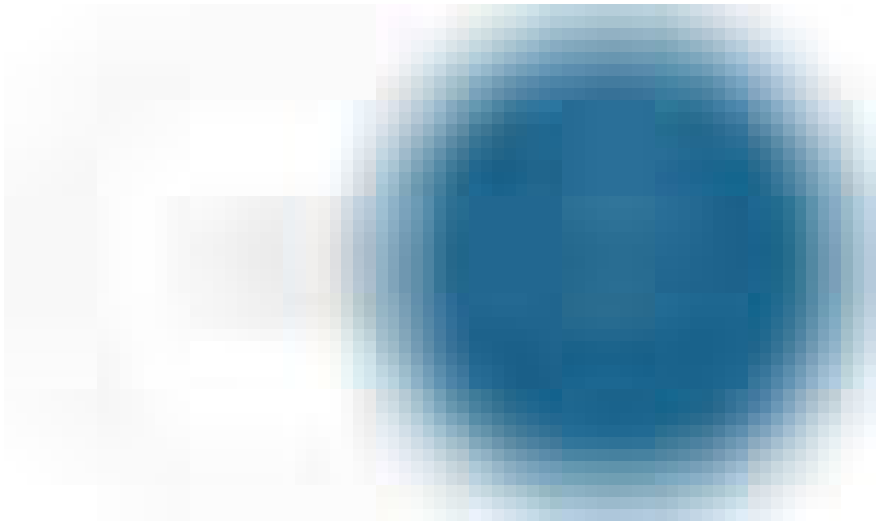
The “transcription” of what we are requesting here is, we want all the data from the `LEFT` that in our scenario is *customer*, and we want from the table *contact* only those that match our validation.

Doing this you should have this data as result from this query:



Right Join

Now that we already saw how it’s *left join* this is not different, the only difference here is that we now want the “*right*” side as the main table, this means that we will return all data from right table even if does not match, of course what does not have a match is empty.



Right Join

Query:

```
SELECT * FROM customer RIGHT JOIN contact ON
```

```
customer.zip_code=contact.zip_code;
```

But it's important to not misunderstand get all data from the right or left side with only have the exactly amount of data that we have at that specific table.

What I mean with this is, for this join we want "everything" from the right side, that is our contact table that has 4 items that we added, and this is the result:

We have 6 items and not 4, but remember, it's not a query to "*return all from one side if match*" only, it's a query that return all from "*one side*" that **MATCH**, we have 2 matches for *zip_code 3423PO*, it's a match so return.

Full Join

Now the last one, that looks like the cross join, the difference here is that we are looking for ***match*** and not a *cartesian product* like the cross join.

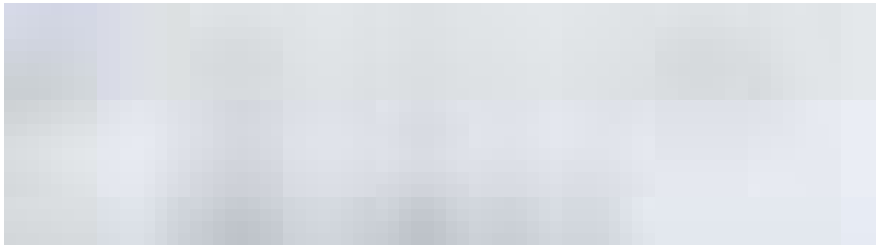
All the blue area is what we will look for the join, basically validating both side according to our "validation" constraint, that in our scenario is zip_code.

Full Join

The full join will *combine* both tables, left and right data and return all *matches* and *unmatched*, let's look the query

```
SELECT * FROM customer FULL JOIN contact ON
customer.zip_code=contact.zip_code;
```

This is what we will have as return:



Here we follow the same principle as the "*joins*" before, that is what does not have a match we still have the data but "*empty*"

Union

This is very useful command, as we are talking about getting data from

2 tables, but of course do not confuse both.

What Union does is that we will combine data from both side, but will return only “unique” data between both, if “*same*” then will not return duplicated, for example that is what happen using join.

This is the query that we use:

```
SELECT customer.email, customer.zip_code FROM customer UNION
SELECT contact.email, contact.zip_code FROM contact;
```

This what we will have as return:

As you can notice, we specify which columns we want, and it’s mandatory, as we need to say which one are “*equal*” so he can compare and return unique items between both.

One important thing regarding this command is that he operate in a very efficient way, this means that order can change.

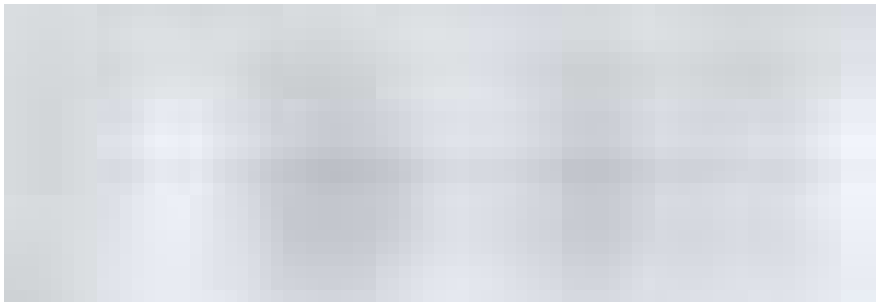
Order

This is a very helpful constraint in order to organise the data that you are requesting, with this we do not "*filter*" the data actually, what he does is organise your response by some criteria.

Just make simple, let's order our query by the age of our customers:

```
SELECT * FROM customer ORDER BY customer.age desc;
```

And this is the response that you should expect:



Alias

This is not intended to query something or anything like, it's just a way to help you organise the name for a table, it's simple a *nickname*.

The basic syntax is like this:

```
SELECT A.name, A.zip_code FROM customer A;
```

And you should get something like this:

Let's split for each part, first what we did, we `SELECT` what we want, but if you pay attention we add a character "**A**" before the *column name*, in our scenario *name* and *zip_code*. But we need to tell to our SQL what that character or "*nickname*" means right?! And we do this after, when we say `FROM customer A` , here we are say how SQL should interpret that letter, that should be the ***alias*** for *customer*.

You can also can do this when have multiple tables, for example:

```
SELECT A.zip_code, B.zip_code FROM customer A, contact B;
```

You can see that both are "the same" but we rename to have another ***alias***.

One more thing that you can do is "change" the name of the table that you are returning, for example:

```
SELECT name as "Person Age", A.zip_code FROM customer A;
```

If you see here I used two formats of alias, for the first i'm not using the alias in order to know which column is, but what I want is that the return from this column be called "*Person Age*" and the second is the alias way of call that we already learned, this is the result.

Delete

As the name shows this is for delete, and I don't need to warn you that is a very "*dangerous*" command, so take care, I will show 2 basic syntax:

```
DELETE FROM customer;
```

You may have notice that we do not specify any particular data, so will delete everything, easy like this, but what we want is to delete a specific data, for this do this:

2.

```
DELETE FROM customer WHERE name = 'Felipe';
```

Here I would say is the safer way of doing, use a specific person.

Drop Table

The subject is about deletion, let's continue with this, the previous command is to delete the data from a table / column, now we will continue but with another kind of deletion, the one that delete the table, or *drop*.

```
DROP TABLE customer;
```

Simple like this, but you do need to be even more careful than the *delete* command, here you will drop the hole table and there's no way to recover.

Another major difference is, this should no be used from your "program" this command should only be used by the terminal command when you do know what need to be done or dropped.

. . .

This is it, I hope you have enjoyed this tutorial, I tried to make easier as possible in order for you easy get the concept, each of this constraints can have more format's or other combinations.

But if you learn all this, i'm 100% confident that you will be able to make your queries and fit your needs, it's a universe, that's why we have DBA's, you really have a lot to learn only from this.

Another important step that I believe that worth see after know this commands is understand, create a unique identifier for your tables, for this take a look on my post about this: Primary Key and Foreign Key

Please share as most as possible as this help me reach more people and continue writing, found any mistake? Get in touch let's help together!

Twitter: @dr_nerd