 arielZarate / **henryAll** Public

Code

Issues

Pull requests

Actions

Projects

Wiki

Security







Insights

Settings

 main ▾




henryAll / FT-M4-master / 02-sql /

 arielZarate ...	3 days ago 
..	
 demo	3 days ago
 homework	3 days ago
 README.json	3 days ago
 README.md	3 days ago




 README.md





Hacé click acá para dejar tu feedback sobre esta clase.



Hacé click acá completar el quiz teórico de esta lecture.

# SQL - Postgres

Como en todo en este mundo, hay muchas opciones de bases de datos SQL. De hecho las hay pagas y gratis. Podríamos usar: MySQL, ORACLE, IBM DB2, SQL server, access, etc. Todas utilizan el lenguaje SQL, así que son muy parecidas, el 80% de las cosas se puede hacer con cualquier motor. Nosotros vamos a ver en nuestro ejemplo PostgreSQL, que es un motor gratis de código abierto que tiene una comunidad muy activa. De hecho, postgres logra sacar funcionalidades antes que los motores pagos!



Para instalar Postgre sigan las instrucciones para su sistema operativo [acá](#).

## SQL

---

Como dijimos, vamos a interactuar con la base de datos a través de SQL. Este es un lenguaje especialmente diseñado para hacer consultas a las bases de datos relacionales. SQL es el acrónimo de Structured Query Language y es un standard mantenido por el ANSI (American National Standards Institute). Usando SQL vamos a poder crear tablas, buscar datos, insertar filas, borrarlas, etc..

### Creando una BD y una tabla

Lo primero que tenemos que hacer es crear una base de dato (Un motor de SQL puede manejar muchas Bases de Datos). Para eso vamos a usar el Statement `Create database`.

```
CREATE DATABASE prueba;
```

Cada Statement SQL termina en ; (punto y coma). En algunas interfaces es obligatorio.

Una vez ejecutado el comando vamos a ver listada la nueva base de datos, yo estoy usando la interfaz CLI de postgres, también pueden usar alguna interfaz gráfica.

Ahora vamos a crear una tabla. Usamos el statement `CREATE TABLE` que tiene la siguiente forma:

```
CREATE TABLE table_name
(
  column_name1 data_type(size),
  column_name2 data_type(size),
  column_name3 data_type(size),
  ....
);
```

Básicamente ponemos el nombre de la columna y luego el tipo de datos de esa columna. Podemos ver algunos tipos de datos comunes [aquí](#). También vamos a poder agregar **CONSTRAINTS** o restricciones por ejemplo:

```
CREATE TABLE ciudades
(
  id serial PRIMARY KEY,
  nombre varchar(255) UNIQUE
);

CREATE TABLE personas
(
  id serial PRIMARY KEY,
  apellido varchar(255) NOT NULL,
  nombre varchar(255) UNIQUE,
  ciudad integer references ciudades (id)
);
```

Con este statement hemos creado dos tablas. La primera es ciudad, que cuenta con ID que tiene la constraint que es PRIMARY KEY, es decir que tiene que ser única y no puede ser nula, y además tiene una columna nombre que es de tipo texto (varchar) y tiene la constraint UNIQUE, por lo tanto no puede haber dos iguales en la misma tabla.

La segunda es la tabla personas, vemos que también tenemos un id que es PRIMARY KEY (esta práctica es muy común y muy recomendable), tenemos nombre y apellido, con la condición que nombre no se repita (es sólo para el ejemplo) y pusimos una columna que se llama ciudad, que hace referencia a un ID de la tabla *ciudades*. Esto último denota una *relación* entre las tablas, y lo hicimos de esta forma para respetar la normalización y de esa forma reducir el tamaño final de la base de datos. Más adelante veremos cómo hacer queries y obtener los datos de una relación.

Ahora agregamos algunos datos en las tablas. Tenemos que empezar por ciudades, ya que para cargar una persona luego (y mantener la **integridad referencial**) vamos a tener que tener algunas ciudades y sus ids. Para insertar datos vamos a usar el statement INSERT INTO :

```
INSERT INTO table_name (column1,column2,column3,...)
VALUES (value1,value2,value3,...);
```

Insertemos tres ciudades:

```
INSERT INTO ciudades (nombre)
VALUES ('Tucuman');

INSERT INTO ciudades (nombre)
VALUES ('Buenos Aires');

INSERT INTO ciudades (nombre)
VALUES ('New York');
```

El tipo de datos SERIAL (el id) es un entero AUTOINCREMENTAL, es decir que no tengo que especificar el ID, si no que se va generando solo. El primero es 1, el segundo 2 y así sucesivamente.

Perfecto ahora tenemos ciudades! Insertemos algunas personas que sean de esas ciudades!

```
INSERT INTO personas (nombre, apellido, ciudad)
VALUES ('Toni', 'Tralice', 1);
```

```
INSERT INTO personas (nombre, apellido, ciudad)
VALUES ('Emi', 'Chequer', 3);
```

```
INSERT INTO personas (nombre, apellido, ciudad)
VALUES ('Fran', 'Etcheverri', 2);
```



Ahora tenemos tablas con datos, pero cómo los consultamos?

## SELECT, WHERE, ORDER BY

Para recuperar datos usamos el statement `SELECT` de esta forma:

```
SELECT column_name, column_name
FROM table_name;
```

Para recuperar todas las personas:

```
SELECT * FROM personas;
```

y todas las ciudades:

```
SELECT * FROM ciudades;
```

El asterisco quiere decir 'todas las columnas'.

Genial, esto sería equivalente al `db.prueba.find()` de mongo! Las filas en SQL no tiene un orden dado, ni siquiera por el orden en el que fueron creadas (muchas veces coincide pero no es necesariamente así), así que si queremos tener los resultados ordenados vamos a usar la cláusula `ORDER BY`, esta va al final del query y le especificamos en qué columna se tiene que fijar para ordenar:

```
SELECT * FROM ciudades
ORDER BY nombre;
```

El motor se da cuenta el tipo de datos de la columna y los ordena en base a eso. también podemos especificar que ordene por más de un campo, y si queremos que sea en orden ascendente o descendiente:

```
SELECT * FROM ciudades
ORDER BY nombre, id DESC;
```

Ahora veamos como buscar o filtrar filas: para eso vamos a usar la cláusula `WHERE` :

```
SELECT column_name,column_name
FROM table_name
WHERE column_name operator value;
```

Por ejemplo, busquemos todas las personas que se llaman 'Toni':

```
SELECT * FROM personas
WHERE nombre = 'Toni';
```

Casi que podemos leerlo en lenguaje natural: 'Seleccioná todas las columnas de la tabla personas donde el nombre sea Toni'. 😊

Podemos agregar más de una condicion:

```
SELECT * FROM personas
WHERE nombre = 'Toni'
AND apellido = 'Tralice';
```

Ahora, si vemos el output de consultar la tabla 'personas', vemos que tenemos el *código* de ciudad, pero no el nombre. Y probablemente en nuestra aplicación querramos mostrar el nombre y no el código, no? Bien, entonces para hacerlo podríamos consultar ambas tablas, y luego buscar el código de cada fila de *personas* en la tabla *ciudades*. De esa forma tendríamos el nombre asociado... por suerte SQL ya viene preparado para eso! con el statement `JOIN` .

## JOINS

La cláusula `JOIN` nos sirve para combinar filas de una tabla con otras de otra tabla, basándonos en un campo que tengan en común. Es el caso de ciudad y personas, vamos a unir las filas de cada tabla basados en el ID de la ciudad.

Para definir un `JOIN` tenemos que decidir qué tablas queremos unir y en base a qué campos, para lo primero ponemos el nombre de la tabla a unir después del `JOIN` y lo segundo lo hacemos con el parámetro `ON` :

```
SELECT * FROM personas
JOIN ciudades
ON ciudades.id = personas.ciudad;
```

Básicamente estamos diciendo: 'Seleccioná todas las columnas de la tabla personas y uní todas las filas con la tabla ciudades donde el id de ciudades sea igual al campo ciudad de personas.'

Podemos reescribir la consulta de esta forma:

```
SELECT p.nombre, p.apellido, c.nombre FROM personas p
JOIN ciudades c
ON c.id = p.ciudad;
```

Ahora sólo pedimos el nombre, apellido de las personas y el nombre de la ciudad. Como `nombre` está en las dos tablas, tenemos que especificar de qué tabla es la columna. Para no escribir todo el nombre completo, podemos definir un ALIAS en la consulta, en esta caso a **personas** le dimos el alias `p` y a **ciudades** el alias `c`.

Según el tipo de union que queremos hacer vamos a usar alguno de estos tipos de JOINS :



Los joins pueden ser operaciones muy costosas, de hecho, las bases de datos no relacionales suelen ser tan performantes porque esquivan los JOINS, logran ser más rápidas, pero ocupando más espacio.

## Homework

---

Completa la tarea descrita en el archivo [README](#)

---