Modelamiento y gestión de bases de datos

Competencias

- Reconocer y diferenciar entidades de atributos.
- Generar modelos lógicos y físicos diagramando relaciones.
- Instalar y configurar postgresQL
- Utilizar la consola de postgres para facilitar la gestión.
- Construir bases de datos, tablas, índices y relaciones.
- Cargar datos en formato SQL y dump
- Exportar datos a formato csv
- Implementar instrucciones de creación, inserción, actualización y eliminación de datos.
- Implementar instrucciones selección, agrupación y ordenamiento de datos.
- Implementar instrucciones de unión entre tablas.
- Implementar subqueries.

¿Qué es SQL?

Structured Query Language (Lenguaje estructurado de consultas) es un lenguaje creado para la manipulación de bases de datos relacionales, el cual permite la creación y modificación de estas. El beneficio de esto es que facilita la administración de los datos almacenados.

Analicemos un caso cotidiano

Imaginemos un directorio telefónico con los datos de todas las personas de una ciudad, pero queremos obtener el número de sólo una de ellas.

- ¿Qué es lo que hacemos para encontrarlo?

Naturalmente, buscar en orden alfabético hasta encontrar el nombre de la persona, lo cual puede tomar bastante tiempo. Sin embargo, SQL nos da la facilidad de obtener rápidamente el número deseado.

- ¿Y qué pasaría si quisiéramos los números de todas las mujeres de la ciudad? ¿O de toda la gente que tenga edad entre los 20 y 30 años? ¿O incluso ambas condiciones juntas?

SQL nos da la posibilidad de hacer este tipo de consultas, reduciendo el tiempo de ejecución en comparación a un humano realizando la misma tarea.

¿Por qué PostgresSQL?

A lo largo de este módulo aprenderemos sobre la implementación de SQL mediante PostgreSQL. Esta es una herramienta utilizada para el manejo de las bases de datos. Lo que diferencia a este de otros sistemas es que permite implementar funciones de distintos lenguajes como C, C++, Java, entre otros. Al ser código abierto, puede ser modificado a gusto del usuario, acomodándose el uso a cualquier persona.

Instalación de PostgresSQL

Un aspecto a considerar es que la instalación de PostgreSQL dependerá del sistema operativo que estén ocupando.

Para efectos prácticos, explicaremos los procesos de instalación para Windows, Linux y MacOS.

Descargar PostgreSQL

1. Entrar a postgresql.org.
2. Elegir la versión para su sistema operativo.
3. Seleccionar *Download the installer*.
4. Hacer click en Download en la versión que desea descargar.

Windows

1. Abrir el archivo instalador
2. Seguir las instrucciones dadas.

El puerto que se indica por defecto es el 5432, este será el que utilizaremos.

Linux

En el caso de Ubuntu se requieren más pasos intermedios:

1. Agregar el repositorio donde se encuentran los archivos del lenguaje a nuestro sistema operativo. Debemos incorporar este repositorio en la carpeta `/etc/apt/sources.list.d/pgdg.list`. Esto facilitará que cuando utilicemos el comando de instalación `apt`, éste sepa dónde buscar.

```
sudo sh -c "echo 'deb http://apt.postgresql.org/pub/repos/apt/
xenial-pgdg main' > /etc/apt/sources.list.d/pgdg.list"
```

2. Posterior a la inclusión del repositorio de PostgreSQL a nuestro `sources.list`, debemos agregar la llave pública de PostgreSQL. Este paso permite que la transacción de información y paquetes esté validada como segura por el gestor de paquetes apt.

```
wget --quiet -O - http://apt.postgresql.org/pub/repos/apt/ACCC4CF8.asc |
sudo apt-key add -
```

3. El último paso es actualizar todos los paquetes que residen en la lista de repositorios mediante la siguiente línea de código:

```
sudo apt-get update
```

Cabe destacar que los pasos anteriores son **procedimentales** para la correcta instalación de PostgreSQL. Las siguientes líneas de código son las que instalan el lenguaje en sí.

```
sudo apt-get install postgresql-common
sudo apt-get install postgresql-9.6 libpq-dev
```

MACOSX

Existen múltiples instaladores que puedes encontrar en el siguiente link, sin embargo, el proceso es similar al de Windows, donde basta con seguir las instrucciones que brinda este mismo.

Configuración de PostgreSQL

- Para poder utilizar la consola de PostgresSQL, debemos abrir al SQL Shell.
- Al abrirla, nos preguntará por Server, Database, Port y Username, estos tendrán un valor por defecto entre corchetes, bastará con apretar Enter para continuar.
- Luego, nos pedirá la contraseña de Postgres, esta será la misma que indicamos en el instalador. La consola quedará lista para trabajar y usar comandos SQL.

Digresión: Algunas características de PostgreSQL

En primer lugar, debemos dejar claro algunos puntos con respecto a la sintáxis de PostgreSQL:

- Las instrucciones deben ser terminadas con el símbolo ;
- Existen nombres restringidos para variables, ya que estos suelen ser comandos y pasa a ser fácil de confundir. Algunos comandos son:
    - ALIAS
    - AND
    - AS
    - CREATE
    - CREATEDB
    - CREATEUSER
    - DATABASE
    - FROM
    - INNER
    - JOIN
    - LARGE
    - PASSWORD
    - WHERE

- Existen varios más aparte de estos, puede encontrarlos en la documentación de PostgreSQL.
- No es sensible a las letras minúsculas o mayúsculas, puedes escribir los comandos de cualquiera de estas formas y PostgreSQL lo reconocerá de igual forma.
  Para crear la base de datos, debemos usar la siguiente sintaxis:

```
CREATE DATABASE nombre_baseDeDatos;
```

Esta nos permitirá alojar todas las tablas que generaremos posteriormente.

Administración de usuarios en PostgreSQL

A la hora de trabajar, más de una persona ocupa la información de una misma base de datos. PostgreSQL da la posibilidad de crear usuarios mediante la definición del rol *administrador de base de datos*. Este usuario con este rol será el encargado de asignar roles, los cuales van a definir el rango de acciones posibles para usuarios ordinarios; así como la creación/modificación/eliminación de tablas, revocación de roles/usuarios por nombrar algunas de las funcionalidades.

Solamente los usuarios registrados van a poder acceder a la base de datos. Para poder crear un usuario, se utiliza la siguiente sintaxis:

```
CREATE USER nombre_usuario;
```

Para ponerle algún límite a ese usuario, se debe hacer de a la siguiente forma:

```
CREATE USER nombre_usuario WITH comando_opcional;
```

Donde algunos de los comandos posibles para *comando_opcional* son:

- **ENCRYPTED PASSWORD:** Le asigna una contraseña encriptada al usuario creado.
- **UNENCRYPTED PASSWORD:** Le asigna una contraseña no encriptada al usuario creado.
- **VALID UNTIL:** La cuenta expirará en la fecha indicada.
- **CREATEDB:** Permite al usuario a crear bases de datos.
- **NOCREATEDB:** No permite al usuario crear bases de datos.
- **SUPERUSER:** Puede crear otros usuarios (volveremos a ver esto más adelante).
- **NOSUPERUSER:** No puede crear otros usuarios.

Estos comandos pueden ser usados simultáneamente. Veamos un ejemplo de esto:

```
CREATE USER Bastian WITH PASSWORD 'contraseña_secreta' VALID UNTIL
2019-12-31 CREATEDB;
```

En este caso, nuestro usuario Bastián tendrá de contraseña contraseña_Secreta, su cuenta expirará el 31 de Diciembre del 2019 y tiene permiso para crear bases de datos.

Para eliminar usuarios, sin tener que esperar la fecha de expiración, podemos usar el comando DROP USER:

```
DROP USER nombre_usuario;
```

Habrán casos en que necesitemos saber cuales son todos los usuarios que tienen acceso a nuestra base de datos. Para esto, haremos lo siguiente:

```
SELECT nombre_usuario FROM pg_user;
```

Donde pg_user es el catálogo que nos ayudará en nuestra búsqueda.

A veces es necesario crear otro usuario que tenga todos los permisos además del administrador de la base de datos, lo que buscamos es crear un superuser. Esto lo hacemos de la siguiente forma:

```
CREATE USER nombre_usuario CREATEUSER;
```

Operaciones comunes a nivel de consola

En la consola de PostgreSQL existen una serie de comandos que nos van a permitir conectarnos a bases de datos, listar tablas y usuarios.

A continuación se presenta una tabla con los comandos más utilizados.

Comando	Acción
`\c nombre_base`	Conectarse a una base de datos específica.
`\l`	Listar todas las bases de datos existentes.
`\du`	Listar todos los usuarios existentes en el motor.
`\dt`	Listar todas las relaciones (o tablas) existentes en una base de datos específica.
`\q`	Salir de la consola de PostgreSQL.

Tabla 1. Operaciones a nivel de consola.

Elementos de una base de datos SQL

Una base de datos se compone de múltiples **tablas**. Cada una de éstas presentarán dos dimensiones: **filas**, que representan a los registros en la tabla; y **columnas**, que van a representar los atributos ingresados en cada registro, definiendo el tipo de dato a ingresar.

Supongamos que deseamos crear un registro telefónico donde alojaremos el nombre, apellido, número telefónico, dirección y edad de una serie de individuos. Resulta que el registro en sí será la tabla `directorio_telefonico`, donde tendremos los campos `nombre`, `apellido`, `numero_telefonico`, `direccion`, y `edad`.

Otro aspecto a considerar es el hecho que existen dos tipos de bases de datos: las **Relacionales** y las **No Relacionales**
:

- Las bases de datos **Relacionales** son aquellas compuestas por varias tablas donde se almacena la información, y posteriormente se relacionan entre sí.
- Las bases de datos **No Relacionales** son aquellas que siguen esquemas más flexibles de organización, donde no necesariamente todas las entradas tienen la misma estructura.

Un aspecto relevante a tomar en cuenta es que para implementar bases de datos relacionales, es necesario tener conocimiento previo sobre qué es lo que vamos a almacenar. Ante la falta de información sobre qué es lo que deseamos almacenar, el mejor enfoque es implementar una base de datos no relacional. Por ejemplo, si trabajamos con una serie de diccionarios anidados, será difícil almacenarlos en tablas.

De manera adicional a las **filas** y **columnas**, las tablas también cuentan con **claves primarias y foráneas**. Estas buscan generar identificadores para cada registro mediante algún valor específico de una columna. Cuando hacemos referencia a esta columna dentro de su tabla de origen, hablaremos de una **clave primaria**. Cuando hacemos referencia a una columna identificadora en otra tabla a la cual hacemos referencia, hablamos de una clave foránea.

Supongamos que en base a nuestra tabla `directorio_telefonico`, deseamos incorporar información de otra tabla llamada agenda que tiene las columnas `nick` y `numero_telefonico`. Ante esta situación, vamos a identificar que la columna `numero_telefonico` en la tabla `directorio_telefonico` será la **clave primaria** y la columna `numero_telefonico` en la tabla `agenda` corresponderá a la **clave foránea**. Este comportamiento es posible dado que el número registrado será congruente en ambas tablas.

Una de las principales características del trabajo con bases de datos, es la necesidad de declarar los distintos tipos de datos existentes en cada campo a completar. Estos aplican restricciones sobre lo que puede ingresar a los registros. No podemos ingresar caracteres cuando piden un número, ni sobrepasar el límite de caracteres posibles. Los tipos de datos más comunes son:

- **INT:** Números enteros de 4 bytes que pueden tomar valor desde -2147483648 hasta +2147483647.
- **SMALLINT:** Números enteros de 2 bytes que pueden tomar valor desde -32768 hasta +32767.
- **BIGINT:** Números enteros de 8 bytes que pueden tomar valor desde -9223372036854775808 hasta 9223372036854775807.
- **FLOAT:** Números decimales de 32 bit de precisión.
- **DOUBLE:** Números decimales de 64 bit de precisión con hasta 15 dígitos decimales.
- **CHAR:** Cadena de hasta 255 caracteres de longitud fija.
- **VARCHAR:** Cadena de hasta 65.535 caracteres de longitud variable. A diferencia de CHAR, si no se ocupa toda la memoria, esta queda libre. CHAR ocupará toda la memoria solicitada.
- **DATE:** Almacena fecha en formato aaaa-mm-dd. <año>-<mes>-<dia>.
- **TIME:** Almacena en tiempo horario desde 00:00:00 hasta 24:00:00.
- **TIMESTAMP:** Almacena fecha y hora juntos: yyyy-mm-dd hh:mm:ss.
- **BOOLEAN:** Tiene 3 valores posibles: Verdadero, Falso o NULL. Estos pueden representarse para el caso Verdadero como 1, yes, t o true, y para el caso Falso como 0, no, false o f.

Instrucciones de creación, inserción, actualización y eliminación de datos

Hasta el momento hemos trabajado con la definición de nuestras tablas a utilizar, y los campos a asignar.

El proceso de vida de una tabla en una base de datos parte con el proceso de Crear, Insertar, Actualizar y Eliminar datos, que responde a las operaciones elementales que podemos realizar en una tabla.

Creación de tablas

La primera tarea que debemos realizar es la creación de la tabla que permitirá almacenar la información en filas y columnas.

Para crear una tabla dentro de nuestro motor, debemos utilizar el comando `CREATE TABLE` acompañado del nombre de la tabla y declarar los tipos de datos a ingresar entre paréntesis. La forma canónica de creación es la siguiente:

```
CREATE TABLE nombre_tabla(
columna1 tipo_de_dato1,
columna2 tipo_de_dato2,
columna3 tipo_de_dato3
);
```

Volviendo a nuestro ejemplo, vamos a crear la tabla `directorio_telefonico` con los campos `nombre`, `apellido`, `numeroTelefonico` y `edad`. Las líneas precedidas por `--` representan comentarios específicos sobre cada línea del código.

```
-- Creamos una tabla con el nombre directorio_telefonico
CREATE TABLE Directorio_telefonico(
   -- Definimos el campo nombre con el tipo de dato cadena con un largo
de 25 caracteres.
  nombre VARCHAR(25),
   -- Definimos el campo apellido con el tipo de dato cadena con un
largo de 25 caracteres.
  apellido VARCHAR(25),
   -- Definimos el campo numeroTelefonico con el tipo de dato cadena con
un largo de 25 caracteres.
  numeroTelefonico VARCHAR(8),
   -- Definimos el campo dirección con el tipo de dato cadena con un
largo de 25 caracteres.
  direccion VARCHAR(255),
   -- Definimos el campo edad con el tipo de dato entero
  edad INT,
   -- Definimos que el campo numeroTelefonico representará la clave
primaria de la tabla.
  PRIMARY KEY (numeroTelefonico)
);
```

nombre	apellido	numeroTelefonico	direccion

Tabla 2. Tabla `Directorio_telefonico` vacía.

Definición de una clave foránea

Por ahora, nuestra tabla tiene la siguiente estructura, la cual carece de registros específicos.

Posterior a la creación de ésta, definiremos los componentes de la segunda tabla, agenda con el campo `numeroTelefonico` y asignaremos una clave foránea proveniente de la tabla `directorio_telefonico`.

```
-- Creamos una tabla con el nombre agenda
CREATE TABLE Agenda(
   -- Definimos el campo numeroTelefonico con el tipo de dato cadena con
un largo de 8 caracteres.
  numeroTelefonico VARCHAR(8),
   -- Vinculamos una clave foránea entre nuestra columna
numeroTelefonico y su simil en la tabla directorio_telefonico
  FOREIGN KEY (numeroTelefonico) REFERENCES
Directorio_telefonico(numeroTelefonico)
);
```

Profundizemos sobre la última instrucción. La forma canónica de implementar una clave foránea responde a los siguientes componentes:

```
-- Explicitamos el tipo de relación a crear en esta tabla
FOREIGN KEY
-- Definimos la columna a referenciar
(columna a referenciar en la tabla)
-- Ahora con REFERENCES vamos a apuntar a la otra tabla donde se
generará la dependencia.
REFERENCES
-- Posterior a la instrucción REFERENCES, indicamos la tabla y la
columna específica
Directorio_telefonico(numeroTelefonico)
```

De manera similar, nuestra tabla agenda no contiene registros por el momento.

nick	numeroTelefonico

Tabla 3. Tabla `agenda` vacía.

Inserción de datos en una tabla

Para que una base de datos sea útil, como dice su nombre, debe contener datos.

Es por esto que existe el comando **INSERT**, en el cual se indica la tabla a la que se agregarán datos, los tipos de datos que esta contiene, y los valores que queremos ingresar en el registro. Esto es un proceso ordenado, y debemos especificar a qué fila pertenecen los datos que estamos ingresando.

Si no usáramos un orden claro, la tabla perdería su estructura, se nos haría imposible obtener la información buscada y la base de datos perdería su utilidad. La forma canónica de la instrucción `INSERT` es la siguiente:

```
INSERT INTO nombre_tabla (columna1, columna2, columna3) VALUES (valor1, valor2, valor3);
Por ejemplo, ingresaremos un registro a las tablas Directorio_telefónico y Agenda:
-- Definimos qué tabla vamos a insertar datos
INSERT INTO Directorio_telefonico
-- Explicitamos cuáles son las columnas a insertar
(nombre, apellido, numeroTelefonico, direccion, edad)
-- Con la instrucción VALUES logramos asociada cada columna con un valor específico
VALUES ('Juan', 'Perez', '123456789' , 'Villa Pajaritos', 21);

-- Realicemos el mismo procedimiento en la tabla Agenda
INSERT INTO Agenda (nick, numeroTelefonico) VALUES ('Juanito', '123456789');
```

Suponiendo que la tabla ya tiene los datos del ejemplo de la sección Elementos de una base de datos SQL, esto se ve de esta manera:

nombre	apellido	numeroTelefonico	direccion	edad
'Pedro'	'Arriagada'	'38472940'	'Valdivia'	23
'Matias'	'Valenzuela'	'38473623'	'Nogales'	22
'Cristobal'	'Missana'	'43423244'	'Con Con'	20
'Javiera'	'Arce'	'94367238'	'Quilpue'	20
'Farid'	'Zalaquett'	'32876523'	'La Florida'	20
'Daniel'	'Hebel'	'43683283'	'San Bernardo'	20
'Bastian'	'Quezada'	'32344242'	'Los Angeles'	22
'Fabian'	'Salas'	'32846352'	'Playa Ancha'	21
'John'	'Rodriguez'	'23764362'	'Constitucion'	21
'Braulio'	'Fuentes'	'23781363'	'Rancagua'	19
'Juan'	'Perez'	'12345678'	'Villa Pajaritos'	21

Tabla 4. Muestra tabla `Directorio_telefonico`.

nick	numeroTelefonico
'Peter'	'38472940'
'Mati'	'38473623'
'Cris'	'43423244'
'Javi'	'94367238'
'Farid'	'32876523'
'Dani'	'43683283'
'Basti'	'32344242'
'Fabi'	'32846352'
'John'	'23764362'
'Braulio'	'23781363'
'Juanito'	'12345678'

Tabla 5. Muestra tabla `agenda`.

Como podemos observar, no podemos llegar e ingresar cualquier dato, sino los que les corresponden a Juan, ya que de lo contrario, los resultados de las búsquedas no serían correctas.

Actualización

A veces nos vemos en la necesidad de cambiar los registros ya existentes, sin embargo, es riesgoso borrarlos e ingresarlos de nuevo (más adelante explicaremos esto). Si queremos actualizar los datos de un registro, debemos usar el comando UPDATE de la siguiente forma:

```
UPDATE Tabla SET columna1=valor_nuevo WHERE condicion;
```

Juan se cambió de casa a Villa Los Leones, por lo que tendremos que actualizar la tabla Directorio_telefónico:

```
-- definimos la instrucción UPDATE en la tabla directorio_telefonico
UPDATE Directorio_telefonico
-- Declaramos que modificaremos el valor de dirección
SET direccion='Villa Los Leones'
-- Definimos a qué registro específico afectará la instrucción.
WHERE nombre='Juan';
```

nombre	apellido	numeroTelefonico	direccion	edad
'Pedro'	'Arriagada'	'38472940'	'Valdivia'	23
'Matias'	'Valenzuela'	'38473623'	'Nogales'	22
'Cristobal'	'Missana'	'43423244'	'Con Con'	20
'Javiera'	'Arce'	'94367238'	'Quilpue'	20
'Farid'	'Zalaquett'	'32876523'	'La Florida'	20
'Daniel'	'Hebel'	'43683283'	'San Bernardo'	20
'Bastian'	'Quezada'	'32344242'	'Los Angeles'	22
'Fabian'	'Salas'	'32846352'	'Playa Ancha'	21
'John'	'Rodriguez'	'23764362'	'Constitucion'	21
'Braulio'	'Fuentes'	'23781363'	'Rancagua'	19
'Juan'	'Perez'	'12345678'	'Villa Los Leones'	21

Tabla 6. Muestra de tabla `Directorio_telefonico` actualizado.

Eliminación

Cuando estamos completamente seguros de que queremos eliminar un registro de una tabla, o incluso la tabla completa, y que de verdad estamos seguro de que la condición que usamos es la correcta, podemos utilizar el comando **DELETE**. Este lo que hace es eliminar uno, varios o todos los registros de la tabla según la condición dada. La sintaxis para eliminar toda la tabla es:

```
DELETE FROM tabla;
```

Para poder seleccionar que registros queremos borrar debemos hacerlo de la siguiente forma:

```
DELETE FROM tabla WHERE condicion;
```

En nuestro ejemplo, eliminaremos a Juan de la tabla de la siguiente forma:

```
DELETE FROM Directorio_telefonico WHERE nombre='Juan';
```

nombre	apellido	numeroTelefonico	direccion	edad	genero
'Pedro'	'Arriagada'	'38472940'	'Valdivia'	23	h
'Matias'	'Valenzuela'	'38473623'	'Nogales'	22	h
'Cristobal'	'Missana'	'43423244'	'Con Con'	20	h
'Javiera'	'Arce'	'94367238'	'Quilpue'	20	m
'Farid'	'Zalaquett'	'32876523'	'La Florida'	20	h
'Daniel'	'Hebel'	'43683283'	'San Bernardo'	20	h
'Bastian'	'Quezada'	'32344242'	'Los Angeles'	22	h
'Fabian'	'Salas'	'32846352'	'Playa Ancha'	21	h
'John'	'Rodriguez'	'23764362'	'Constitucion'	21	h
'Braulio'	'Fuentes'	'23781363'	'Rancagua'	19	h

Tabla 7. Datos de “Juan” eliminados.

Y para borrar todos los datos de la tabla:

```
DELETE FROM Directorio_telefonico;
```

nombre	apellido	numeroTelefonico	direccion

Tabla 8. Datos eliminados.

Lo pernicioso del método DELETE

Si bien el comando **DELETE** existe para ser usado, debe usarse con mucha precaución, es más, sólo el administrador de la base de datos debería ser capaz de eliminar datos de ella.

Este comando es susceptible a errores, ya que si no implementamos correctamente la condición en el comando **WHERE**, podemos eliminar datos que no teníamos pensado en borrar y no hay posibilidad de recuperarlos.

PostgreSQL les da a todos permisos por defecto, sin embargo, utilizar el comando **DELETE** no es parte de ellas, el administrador de la base de datos es el encargado de otorgar ese poder.

Añadiendo o eliminando columnas

Resulta que en la definición anterior de nuestra tabla, olvidamos agregar la columna `nick`.

Ante la eventualidad que deseemos añadir/remover una columna específica de una tabla, podemos implementar la siguiente sintaxis:

```
ALTER TABLE nombre_tabla
ADD nueva_columna tipo_de_dato;
```

```
-- Declaramos que alteraremos la tabla Agenda
ALTER TABLE Agenda
-- Definimos el campo nick con el tipo de dato cadena con un largo de 25 caracteres.
ADD nick VARCHAR(25);
```

Si deseamos eliminar alguna columna en específico, podemos implementar la instrucción `DROP` de manera análoga a como lo hicimos con `ADD`.

Restricciones

Puede que existan casos en los que nuestras columnas necesiten reglas que cumplir, como que no hayan valores repetidos o que no existan campos vacíos. Es por eso que existen las restricciones. Algunas de estas son:

- **NOTNULL:** La columna no puede tener valores NULL.
- **UNIQUE:** Todos los valores de la columna deben ser diferentes unos a otros.
- **PRIMARY KEY:** Aplica la clave primaria.
- **FOREIGN KEY:** Aplica la clave foránea.
- **CHECK:** Todos los valores de una columna deben satisfacer una condición en específico.
- **DEFAULT:** Le da un valor por defecto a aquellos registros que no tengan un valor asignado.
- **INDEX:** Sirve para crear y recuperar data de forma rápida.

Estos se aplican de la siguiente forma:

```
-- Creamos una tabla
CREATE TABLE nombre_tabla(
-- Declaramos una serie de restricciones a cada campo de dato creado
columna1 tipo_de_dato1 restriccion,
columna2 tipo_de_dato2 restriccion,
columna3 tipo_de_dato3 restriccion
);
```

PRIMERY KEY y FOREIGN KEY se manejan de forma distinta, los explicaremos en la siguiente sección. Usaremos de ejemplo la misma tabla que creamos antes. Le asignaremos restricciones a las tablas que creamos anteriormente:

Volveremos a crear las tablas Directorio_telefónico y Agenda:

```
CREATE TABLE Directorio_telefonico(
nombre VARCHAR(25), NOTNULL
apellido VARCHAR(25),
numeroTelefonico VARCHAR(8) UNIQUE,
direccion VARCHAR(255),
edad INT
);
```

```
CREATE TABLE Agenda(
nick VARCHAR(25), NOTNULL
numeroTelefonico VARCHAR(8) UNIQUE
);
```

En este caso, hicimos que numeroTelefonico sea un valor único que no pueda repetirse, y que nombre y nick no puedan tener valores NULL.

Restricciones a Nivel de PRIMARY y FOREIGN KEY

Como mencionamos antes, las claves primarias sirven para identificar un registro único en una tabla, y la clave foránea sirve para referenciar esa columna desde otra tabla. La forma de aplicarlas es la siguiente:

```
CREATE TABLE tabla1(
columna1 tipo_de_dato1,
columna2 tipo_de_dato2,
columna3 tipo_de_dato3,
PRIMARY KEY (columna1)
);
```

```
CREATE TABLE columna2(
columna4 tipo_de_dato4,
columna5 tipo_de_dato5,
FOREIGN KEY (columna4) REFERENCES tabla1(columna1)
);
```

Volvamos al ejemplo de Directorio_telefonico y Agenda, ambos usan los mismos números telefónicos para referirse a las mismas personas. La diferencia aquí, es que obtenemos los números desde Directorio_telefonico para llevarlos a Agenda.

Es por esto, que numeroTelefonico será clave primaria **(PRIMARY KEY)** en Directorio_telefonico y clave foránea **(FOREIGN KEY)** en Agenda. Esto se puede ver reflejado así:

```
CREATE TABLE Directorio_telefonico(
nombre VARCHAR(25),
apellido VARCHAR(25),
numeroTelefonico VARCHAR(8),
direccion VARCHAR(255),
edad INT,
PRIMARY KEY (numeroTelefonico)
);
```

```
CREATE TABLE Agenda(
nick VARCHAR(25),
numeroTelefonico VARCHAR(8),
FOREIGN KEY (numeroTelefonico) REFERENCES
Directorio_telefonico(numeroTelefonico)
);
```

Instrucciones de selección, agrupación y ordenamiento de datos

Selección

Tenemos todos los comandos para armar una base de datos, pero ¿Qué hacemos con ella? ¿De qué nos sirve tener todos esos datos ordenados? Con el comando SELECT podemos obtener información de las columnas que cumplan las condiciones indicadas.

Si solamente queremos ver una columna en específico, la sintaxis es la siguiente:

```
SELECT columna FROM tabla;
```

Si queremos seleccionar toda la tabla, en el espacio de la columna debemos usar `*`, conocido como un comodín. La sintaxis de uso es la siguiente:

```
SELECT * FROM tabla;
```

En estos casos, estamos mostrando las columnas. Si queremos ver filas en específico, tendremos que usar condiciones con comandos como el ya conocido **WHERE**, y otros como **LIMIT**, que mostrará las primeras filas hasta la cantidad indicada.

```
-- Seleccionamos todas las filas de una columna que cumplan con la
condición
SELECT columna FROM tabla WHERE condicion;
```

```
-- Seleccionamos una cantidad limitada de una columna específica
SELECT columna FROM tabla LIMIT cantidad;
```

Volviendo al directorio telefónico, vamos a consultar por aquellos que su apellido sea Quezada:

```
SELECT * FROM Directorio_telefonico WHERE apellido='Quezada';
```

nombre	apellido	numeroTelefonico	direccion	edad
'Bastian'	'Quezada'	'32344242'	'Los Angeles'	22

Tabla 9. Muestra de `apellido='Quezada'`.

SELECT nos permite crear nuevas columnas, aplicando operaciones a las columnas ya existentes. Algunas operaciones normalmente utilizadas son:

- **MIN:** Entrega el mínimo de los datos de una columna.
- **MAX:** Entrega el máximo de los datos de una columna.
- **LENGTH:** Calcula el largo de los datos en una columna.
- **COUNT:** Cuenta la cantidad de filas.

En base a estas operaciones, podemos extraer resultados intermedios que puedan ser asignados a tablas o impresos en la consola.

La sintaxis para extraer una operación específica de una tabla sería la siguiente:

```
SELECT operacion(columna) AS nombre_nueva_columna FROM tabla
```

Generaremos una consulta sobre el largo de cada nombre en nuestra tabla de directorio telefónico:

```
-- seleccionaremos el largo de los nombres
SELECT LENGTH(nombre)
-- y le asignaremos el nombre largo_de_nombre
AS largo_del_nombre
-- de la tabla Directorio_telefonico
FROM Directorio_telefonico
```

largo_del_nombre
5
6
8
7
5
6
7
6
4
7

Tabla 10. Largo de nombres.

El comando **AS** también puede ser usado en columnas que no necesariamente se les haya aplicado una operación, a este nuevo nombre lo conoceremos como Alias, esto se hace cuando se quiere dar un nombre más pequeño o más fácil de usar cuando este se utilizará varias veces.

```
-- seleccionaremos
SELECT
-- una columna específica como alias_columna
alias_tabla.columna as alias_columna
-- de una tabla
FROM tabla alias_tabla;
```

Podemos notar que también se le dio un alias a la tabla. Veamos este ejemplo, Directorio_telefonico es muy largo, así que lo renombraremos a dt, también le daremos alias a algunas de sus columnas:

```
SELECT
dT.nombre as n,
dT.numeroTelefonico as nT
FROM Directorio_telefonico dT;
```

Es importante destacar que una vez renombrada la columna, **solo puedes llamarla por su Alias**.

Agrupación

A veces nos vemos en la necesidad de agrupar filas que tengan datos iguales para poder trabajar con ellos de manera más sencilla, para eso tenemos el comando GROUP BY, dejará las filas juntas según los valores de la columna indicada.

```
SELECT columna1, columna2, columna3
FROM tabla
GROUP BY columna1;
```

Por ejemplo, agruparemos las filas de dT por apellido:

```
SELECT n, edad
FROM dT
GROUP BY edad;
```

n	edad
'Pedro'	23
'Matias'	22
'Bastian'	22
'Cristobal'	20
'Javiera'	20
'Farid'	20
'Daniel'	20
'Fabian'	21
'John'	21
'Braulio'	19

Tabla 11. Agrupación.

Ordenamiento

Similar a lo que es agrupar, puede que necesitamos que nuestras filas estén ordenadas según los valores que tengan en alguna columna.

Un ejemplo común sería ordenar los índices de menor a mayor. Podemos ordenar las filas utilizando el comando **ORDER BY**:

```
SELECT columna1, columna2, columna3
FROM tabla
ORDER BY columna1;
```

Podemos ordenar de forma decreciente **DESC** o ascendente **ASC**, ubicando el comando después de la columna seleccionada para ordenar las filas. Por ejemplo, si queremos ver las filas ordenadas con nombres desde la Z hasta la A, debemos hacer lo siguiente:

```
SELECT n, apellido, nT
FROM dT
GROUP BY n DESC;
```

n	apellido	nT
'Farid'	'Zalaquett'	'32876523'
'Matias'	'Valenzuela'	'38473623'
'Fabian'	'Salas'	'32846352'
'John'	'Rodriguez'	'23764362'
'Bastian'	'Quezada'	'32344242'
'Cristobal'	'Missana'	'43423244'
'Daniel'	'Hebel'	'43683283'
'Braulio'	'Fuentes'	'23781363'
'Pedro'	'Arriagada'	'38472940'
'Javiera'	'Arce'	'94367238'

Tabla 12. Ordenamiento.

Operaciones de Unión entre tablas

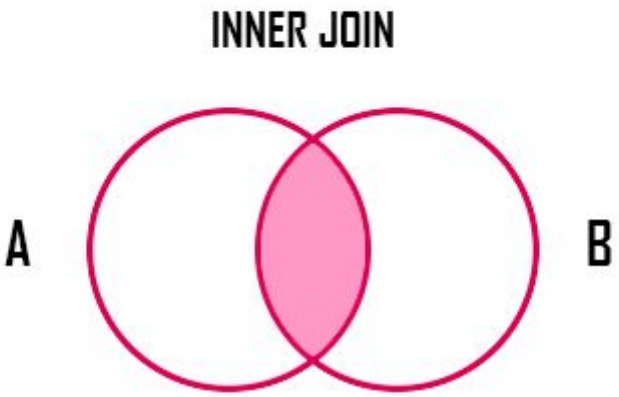
Anteriormente mencionamos cómo las tablas se relacionan a través de las claves primaria y foraneas. Resulta que una de las virtudes de implementar claves primarias y foráneas en nuestras tablas es la posibilidad de generar operaciones entre tablas para incorporar información de distintas fuentes.

Tomemos el siguiente caso hipotético: queremos unir las filas de tablas que cumplan con algún dato común, lo cual podemos expresar así:

```
-- Seleccionamos las columnas desde la tabla1
SELECT columnas FROM tabla1
-- Posterior a la selección de la columna, indicamos que vamos a generar
la unión
-- con la columna de la tabla2
JOIN tabla2 ON tabla1.columna=tabla2.columa;
```

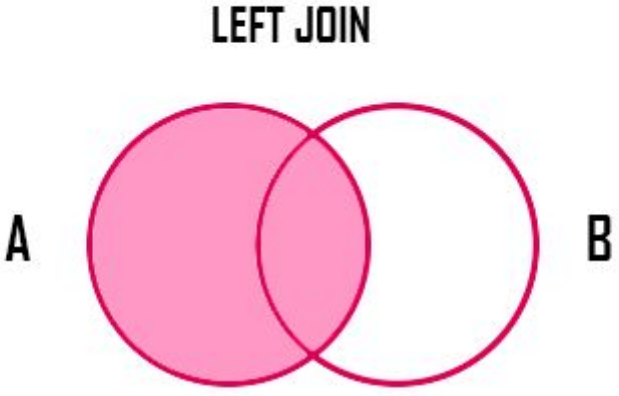
Esto unirá las columnas indicadas, pero sólo se mostrarán las filas que cumplan la condición dada. Existen distintas formas de combinar filas, esto se ve traducido en distintos tipos de **JOIN**:

- **INNER JOIN:** Une sólo las columnas comunes entre ambas tablas.
- **LEFT JOIN:** Une todas las columnas de la primera tabla con las columnas en común de la segunda tabla.
- **FULL OUTER JOIN:** Une todas las columnas de ambas las tablas.



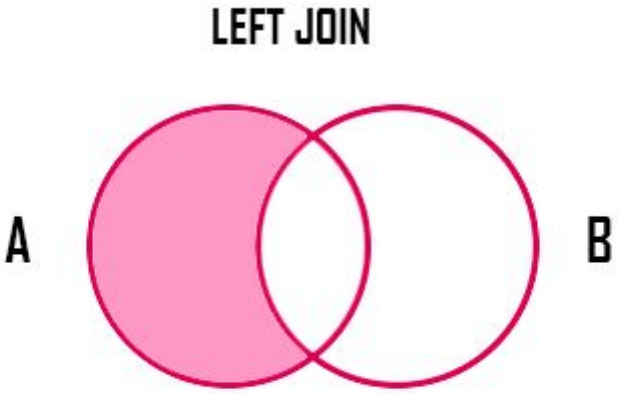

```
SELECT columnas
FROM A
INNER JOIN B
ON A.columna=B.columna
```

Imagen 1. Inner Join.



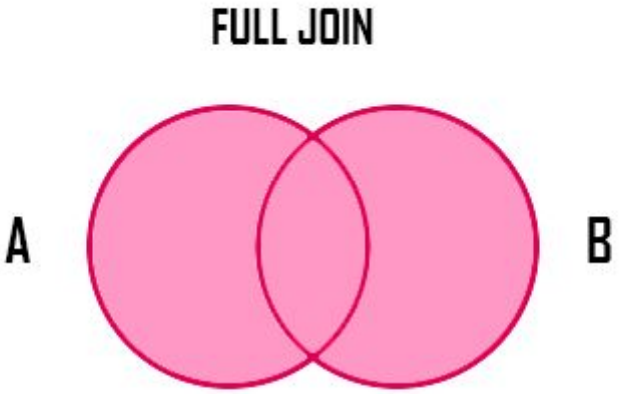

```
SELECT columnas
FROM A
LEFT JOIN B
ON A.columna=B.columna
```

Imagen 2. Left Join.




```
SELECT columnas
FROM A
LEFT JOIN B
ON A.columna=B.columna
WHERE B.columna IS NULL
```

Imagen 3. Left Join Null.




```
SELECT columnas
FROM A
FULL OUTER JOIN B
ON A.columna=B.columna
```

Imagen 4. Full Join.

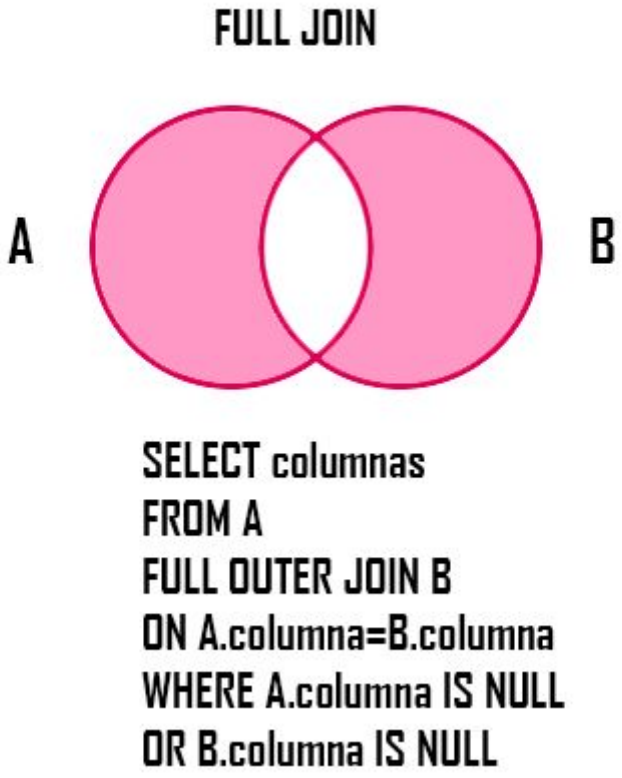


Imagen 5. Full Join Null.

Regresando al ejemplo del directorio, haremos un JOIN con Agenda que muestre solo las filas en que el número sea igual en el directorio y la agenda:

```
SELECT a.nick, dT.direccion
FROM dT
JOIN Agenda a ON dT.numeroTelefonico=a.numeroTelefonico;
```

nick	direccion
'Peter'	'Valdivia'
'Mati'	'Nogales'
'Cris'	'Con Con'
'Javi'	'Quilpue'
'Farid'	'La Florida'
'Dani'	'San Bernardo'
'Basti'	'Los Angeles'
'Fabi'	'Playa Ancha'
'John'	'Constitucion'
'Braulio'	'Rancagua'

Tabla 13. Ejemplo de JOIN.

Implementación de subqueries

Una subquery (o consulta interna) es una query implementada **dentro** de otra query principal de SQL, la cual debe enmarcarse dentro de la cláusula `WHERE`. Una de las principales aplicaciones de las subqueries es para retornar datos que serán utilizados posteriormente en una consulta principal. De esta manera funcionan como una condición para restringir los datos.

En algunas ocasiones, para poder hacer una consulta a una tabla es necesaria la consulta de otra tabla, la cual no es nuestro objetivo final. Para eso, existen las **subqueries** (sub-consultas), las cuales son consultas temporales que sólo estarán para poder obtener el resultado de nuestra consulta. Si está familiarizado con el concepto de recursión, esto resultará familiar para usted.

La forma canónica de la sintaxis es la siguiente:

```
-- Esta es nuestra query principal: seleccionamos todas las columnas
especificadas
SELECT columnas
-- A partir de la siguiente subquery:
-- Seleccionaremos otras columnas dentro de la misma tabla, que
satisfagan
-- la condición especificada
FROM (SELECT otras_columnas FROM tabla2 WHERE condicion)
-- Renombramos con el siguiente alias
AS nuevo_nombre
JOIN tabla2 ON condicion2;
```

Retomemos el ejemplo anterior donde agrupamos la tabla por edad. Un problema es que no están ordenados. Para lograr este objetivo implementaremos una subquery donde primero agruparemos los datos y posteriormente ordenaremos.

```
-- Seleccionando n y edad de la tabla dt
SELECT n, edad FROM dT
-- Ordenaremos por el resultado de la siguiente subquery
ORDER BY(
-- Agrupamos n y edad por cada edad.
  SELECT n, edad
  FROM dT
  GROUP BY edad);
```

n	edad
'Pedro'	23
'Matias'	22
'Bastian'	22
'Fabian'	21
'John'	21
'Cristobal'	20
'Javiera'	20
'Farid'	20
'Daniel'	20
'Braulio'	19

Tabla 14. Aplicando agrupación y ordenamiento.

Algunas reglas que deben seguir las subqueries:

- Las consultas internas **deben** estar encapsuladas entre paréntesis.
- Un subquery puede tener **sólo una columna** especificada en `SELECT`, con la excepción de múltiples columnas definidas en la consulta principal.
- El comando `ORDER BY` **no se puede utilizar** en una consulta interna. La excepción es que esta instrucción si puede ser incluida en la consulta principal.
- Para obtener un resultado similar a `ORDER BY` dentro de una consulta interna, se puede implementar el comando `GROUP BY`.
- Aquellas consultas internas que retornen **más de una fila** sólo pueden ser utilizadas con operadores de múltiples valores como `IN`.

Para trabajar con archivos .csv

Por lo general nuestra primera ingesta de datos a una base de datos se realizará mediante un `.csv`. A continuación se detalla el flujo con el cual podemos ingresar una serie de registros alojados en un `.csv`.

- Abra el archivo en un editor de texto, de recomendación **Sublime Text** o **Atom**.
- Si las filas están encerradas entre comillas, eliminalas.
- Cree la tabla a la cual desea agregar el archivo csv. Este paso lo encontrará explicado en la sección.

Utilizar en la consola de `postgresql` el siguiente comando:

```
COPY nombre_tabla FROM 'directorio/archivo.csv' delimiter ',' csv
[header];
```

Notas:

- El símbolo `,` representa el límite entre columnas en cada fila del archivo csv. Si está usando otro símbolo, remplace `,` por lo que esté ocupando.

- [header] es opcional, si su archivo csv tiene los nombres de las columnas en la primera fila, [header] reemplácelo por header. De ser el caso contrario, deje en blanco ese espacio.

Si lo que desea es almacenar una tabla en un archivo de texto, utilice el siguiente comando:

```
COPY nombre_tabla TO 'directorio/archivo.csv' WITH CSV DELIMITER ',';
```

Transacciones

Las transacciones son secuencias de instrucciones ordenadas, las cuales pueden ser indicadas de forma manual o pueden ser aplicadas automáticamente. Estas realizan cambios en las bases de datos, a la hora de aplicar comandos de manipulación de columnas y registros.

Estas transacciones tienen las siguientes propiedades:

- **Atomicidad:** Todas las operaciones realizadas en la transacción deben ser completadas. En el caso que ocurra un fallo, está la transacción es abortada y devuelve todo al estado previo a la transacción.
- **Consistencia:** La base de datos cambiará solamente cuando la transacción se haya realizado.
- **Aislamiento:** Las transacciones pueden ocurrir independientes unas de otras.
- **Durability:** El resultado de la transacción persiste a pesar de que el sistema falle.

Es posible tener control de las transacciones. Para eso, existen los siguientes comandos:

- **COMMIT:** Guarda los cambios de la transacción.
- **ROLLBACK:** Retrocede los cambios realizados.
- **SAVEPOINT:** Guarda el punto de partida al cual volver a la hora de aplicar **ROLLBACK.**
- **SET TRANSACTION:** Le asigna nombre a la transacción.

Estos comandos sólo pueden ser usados con las operaciones INSERT, UPDATE y DELETE, ya que aquellos que manipulan toda la tabla hacen este proceso automáticamente. La sintaxis de estos comandos es la siguiente:

```
COMMIT;
```

```
SAVEPOINT nombre_savepoint;
```

```
ROLLBACK [TO nombre_savepoint];
```

Lo que está entre corchetes es de carácter opcional, por lo que podemos decirle a ROLLBACK a que punto volver, o este volverá al último punto por defecto.

```
SET TRANSACTION [READ ONLY|WRITE][NAME nombre_transaccion];
```

En este caso, podemos usar **READ ONLY** para solamente leer la base de datos, **READ WRITE** para leer y escribir sobre ella, y poder nombrar la transacción con el comando **NAME**.

Veamos el siguiente ejemplo:

```
SET TRANSACTION READ WRITE NAME primera_transaccion;
INSERT INTO nombre_tabla (columna1, columna2, columna3) VALUES (valor1,
valor2, valor3);
SAVEPOINT registro_ingresado;
```

En este caso creamos una transacción en la que se puede leer y escribir sobre la base de datos que tiene de nombre 'primera_transaccion', luego insertamos un valor e hicimos **SAVEPOINT** para poder volver a este punto en cualquier caso.

Veamos el siguiente ejemplo:

```
UPDATE nombre_tabla SET columna1=valor_nuevo WHERE condicion;
ROLLBACK TO registro_ingresado;
COMMIT;
```

Ahora, realizamos un **UPDATE**, sin embargo, nos devolvimos al punto guardado en **SAVEPOINT**, para así finalmente guardarlo haciendo **COMMIT**.

Plan de consulta

Un plan de consulta es una muestra de cuanto demoraría realizar una consulta y los pasos ordenados que esta sigue para poder realizarla sin la necesidad de hacer la consulta como tal. Cuando utilizamos comandos tales como **SELECT**, **UPDATE** o **DELETE**, el motor de la base de datos debe elegir cuál es el mejor realizar una consulta, sin embargo, no podemos saber a simple viste que es lo que este elige.

Para poder saber eso, utilizamos el comando **EXPLAIN**. Este lo que haces es mostrar un árbol con distintos niveles, el cual está ordenado según las acciones que esta toma. La sintaxis es la siguiente:

```
EXPLAIN [ANALYZE] operacion_a_ejecutar;
```

El comando opcional **ANALYZE** permite que no sólo pregunte por el plan, sino que también realice la operación indicada. Si utilizamos **EXPLAIN ANALYZE** pero no queremos que se aplica la operación debemos hacer lo siguiente:

```
BEGIN;
EXPLAIN ANALYZE operacion_a_ejecutar;
ROLLBACK;
```

Veamos el siguiente ejemplo que hace una consulta con WHERE:

```
EXPLAIN ANALYZE SELECT * FROM Directorio_telefonico WHERE
apellido='Quezada';
```

Nos da como respuesta:

```
Seq Scan on directorio_telefonico (cost=0.00..11.38 rows=1 width=690)
(actual time=2.905..2.908 rows=1 loops=1)
  Filter: ((apellido)::text = 'Quezada'::text)
  Rows Removed by Filter: 9
Planning Time: 0.189 ms
Execution Time: 2.954 ms
```

Donde nos indica que se hizo un escaneo por secuencia, donde el primer paréntesis indica lo que espera obtener en cuanto al costo de iniciación (antes de los 2 puntos) y el costo de ejecución (después de los puntos), la cantidad de columnas seleccionadas y el ancho de estas, y en el segundo paréntesis indica lo que realmente se obtuvo, más la cantidad de ciclos realizados. También indica el filtro que se utilizó, cuantas columnas removió, el tiempo del plan y el tiempo de ejecución de este.

Hacer uso del plan de consulta es útil para saber qué es lo que está pasando durante la ejecución de nuestra consulta y detectar si es que existe algún problema que esté haciendo tomar más tiempo de lo que es necesario.

Ahora, veamos este ejemplo utilizando **JOIN**:

```
EXPLAIN ANALYZE SELECT a.nick, dT.direccion
FROM Directorio_telefonico dT
JOIN Agenda a ON dT.numeroTelefonico=a.numeroTelefonico;
```

Nos da como respuesta:

```
Hash Join (cost=12.47..30.34 rows=620 width=584) (actual
time=0.250..0.259 rows=10 loops=1)
  Hash Cond: ((a.numerotelefonico)::text = (dt.numerotelefonico)::text)
  -> Seq Scan on agenda a (cost=0.00..16.20 rows=620 width=102) (actual
time=0.056..0.058 rows=10 loops=1)
  -> Hash (cost=11.10..11.10 rows=110 width=550) (actual
time=0.106..0.106 rows=10 loops=1)
        Buckets: 1024 Batches: 1 Memory Usage: 9kB
        -> Seq Scan on directorio_telefonico dt (cost=0.00..11.10
rows=110 width=550) (actual time=0.067..0.074 rows=10 loops=1)
Planning Time: 39.777 ms
Execution Time: 0.337 ms
```

En este caso, podemos ver que el árbol es más grande, dado la complejidad de la operación, sin embargo, es el mismo formato anterior, donde lo único que cambia es la acción ejecutada en el plan.

Cada símbolo `->` representa un nivel del árbol, donde las acciones se ven en orden desde arriba hacia abajo.