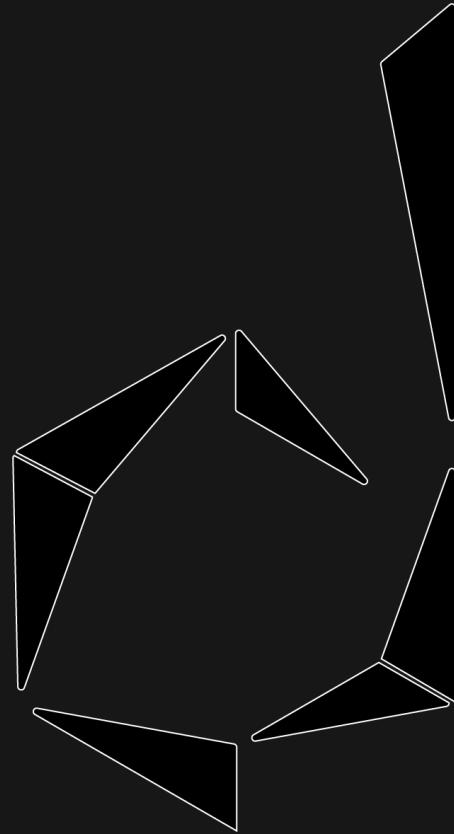


# INGENIERÍA MECATRÓNICA



## DI\_CERO

DIEGO CERVANTES RODRÍGUEZ

PROGRAMACIÓN: DESARROLLO BACKEND

SQL

Ejercicios SQL, Queries y  
Diseño Diagrama - PostgreSQL 

# Contenido

<b>Representación de las Bases de Datos: Nomenclatura de Chen .....</b>	3
<b>Lenguaje de Programación SQL - PostgreSQL .....</b>	3
Interfaz de usuario pgAdmin o Consola SQL Shell de PostgreSQL .....	3
Comandos Comunes de la Consola de Postgres: SQL Shell .....	4
Funcionalidades Comunes de la Interfaz Gráfica de PostgreSQL: pgAdmin .....	8
<b>Diseño del Diagrama ER De una Base de Datos Relacional - PostgreSQL .....</b>	16
Tipos de Datos Especiales de PostgreSQL .....	17
<b>Sub-lenguajes de SQL: DDL (Data Definition Language).....</b>	18
<b>CREATE:</b> .....	19
Crear una Base de Datos con SQL .....	19
Crear una Tabla con SQL.....	20
Crear una Vista con SQL.....	23
<b>ALTER:</b> .....	23
Alterar una Tabla con SQL.....	23
Llaves Principales y Foráneas (PRIMARY & FOREIGN KEYS).....	23
<b>DROP:</b> .....	26
Borrar una Tabla, Columna o Base de Datos con SQL .....	26
<b>Sub-lenguajes de SQL: DML (Data Manipulation Language) .....</b>	27
<b>INSERT:</b> .....	28
Insertar Datos Nuevos a la Tabla de una Base de Datos con SQL .....	28
<b>UPDATE:</b> .....	30
Editar Datos en la Tabla de una Base de Datos con SQL .....	30
Funciones Especiales de Postgres para los Comandos INSERT Y UPDATE .....	31
<b>DELETE:</b> .....	32
Borrar Todos los Datos de una Fila Perteneciente a la Tabla de una Base de Datos con SQL .....	32
<b>SELECT:</b> .....	33
Extraer Todos los Datos de una Columna Perteneciente a la Tabla de una Base de Datos .....	33
Consultas o Queries: Extracción de información de una base de datos.....	33
Funciones Especiales de Postgres para el Comando SELECT .....	39
Crear una Vista: Almacenar un Query en una “Variable” con SQL .....	41
<b>Nested Queries:</b> Consultas Anidadas (Agujero de Conejo) .....	41

Vistas Volátiles y Materializadas: Query que extrae datos actuales o históricos .....	43
Obtención del Diagrama ER De una Base de Datos Relacional .....	45
Ejercicios Query de SQL.....	47
Código SQL - Creación y/o Modificación de la Base de Datos (DDL y DML).....	52
Referencias.....	52



## Representación de las Bases de Datos: Nomenclatura de Chen

- **Entidad:** Se refiere a una **tabla** que almacena datos sobre un tipo de objeto o elemento del mundo real.
  - Cada **fila** en la **tabla** representa una **instancia individual** de esa **entidad**.
  - Cada **columna** en la **tabla** representa un **atributo o característica** de esa **entidad**.
- **Atributo:** Son las **columnas de una tabla** que representan las **características o propiedades** de la **entidad** que está siendo modelada, todas ellas tienen un **nombre y tipo de dato** asociado.
- **Registro:** Representa una **fila perteneciente a una tabla**. También es conocido como "**tupla**" y **contiene los valores** de los **atributos** correspondientes a una **instancia** específica de una **entidad**.

## Lenguaje de Programación SQL - PostgreSQL

Las siglas de SQL significan Structured Query Language, la función principal de este lenguaje de programación es **realizar consultas** a una **base de datos (DB)** de una forma estandarizada no importando que base de datos se esté utilizando y fue creado por la empresa IBM en los años 70.

Además del lenguaje SQL existen los lenguajes NoSQL, cuyas siglas significan “Not Only SQL”, estos se utilizan más que nada en bases de datos no relacionales, donde, aunque se basan principalmente en el lenguaje SQL, pueden variar considerablemente en términos de sintaxis y funcionalidad, dependiendo del tipo de base de datos NoSQL que se esté utilizando.

Pero hablando de **PostgreSQL**, este es un motor de base de datos, **no un tipo de database**, por lo cual a continuación describiremos los elementos de los que se conforman las **DB**:

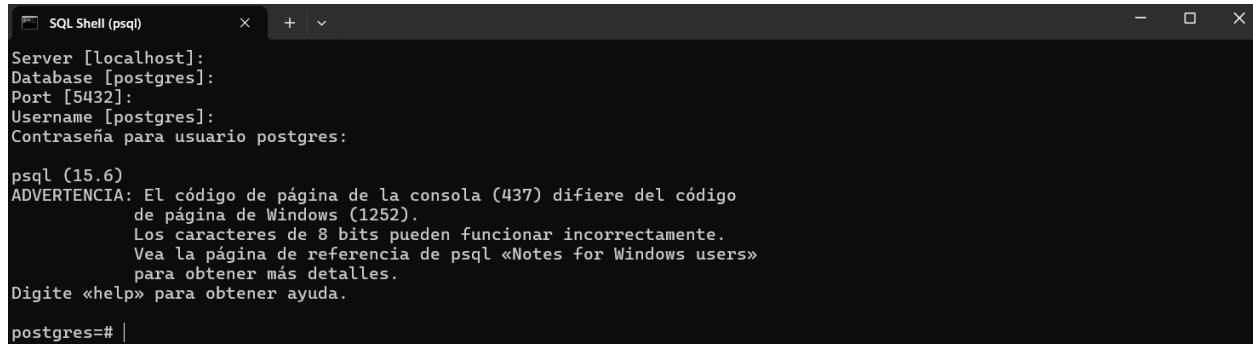
- **Servidor de base de datos:** Es un computador o servicio de nube con una URL o dominio asignado que tiene un **motor** de **base de datos** instalado y en ejecución.
- **Motor de base de datos:** Se refiere a un software que provee un conjunto de funcionalidades encargadas de administrar una **DB**, como lo es **PostgreSQL**.
- **Base de datos:** Grupo de datos que pertenecen a un mismo contexto.
- **Tabla de base de datos:** Estructura que organiza los datos en **filas** y **columnas** formando una matriz, llamada también **entidad**.
- **Esquemas de base de datos en PostgreSQL:** Grupo de objetos ORM (Object Related Mapping) que definen cómo se organiza y almacena la información en una **database**, guardando la relación de las entidades (**tablas, relaciones, funciones, etc**).

## Interfaz de usuario pgAdmin o Consola SQL Shell de PostgreSQL

Cuando se utilicen **bases de datos** en el motor de **PostgreSQL** podremos acceder a ellas a través de una interfaz llamada **pgAdmin** o a través de la consola **SQL Shell**. Si utilizamos **SQL Shell**, lo primero que veremos en pantalla es la consola esperando que se ingresen las características de conexión, al dar clic en la tecla de Enter irán apareciendo uno a uno los siguientes parámetros de ingreso:

- **Server [IP]:** En este parámetro se indica la dirección IP donde se encuentra localizada la base de datos, por defecto se encuentra en **localhost**, que corresponde a la dirección IP **127.0.0.1** (representan lo mismo), pero si queremos conectarnos a otra dirección, se debe indicar antes de dar clic en la tecla Enter.
- **Database [nombreDataBase]:** Aquí se indica el nombre de la base de datos a la que nos queremos conectar, en un inicio esta corresponde por default a la base de datos postgres, la cual fue declarada durante la instalación de **PostgreSQL**.
- **Port [numeroPuerto]:** En este parámetro se indica el número de puerto de conexión al servidor, el cual se declara durante la instalación de **PostgreSQL** y por defecto es el 5432, aunque si queremos, aquí podemos declarar otro puerto de conexión, este se debe indicar antes de dar clic en la tecla Enter.
- **Username [nombreUsuario]:** Aquí se indica el nombre de usuario utilizado para realizar la conexión con el servidor de la base de datos, este por defecto es **postgres**, el cual fue declarado durante la instalación de **PostgreSQL**, aunque podemos elegir otro, pero este antes debe haber sido configurado en Postgres.
- **Contraseña para el usuario de postgres:** Si el usuario elegido es **postgres**, la contraseña será la declarada durante la instalación de **PostgreSQL**, sino será la declarada para el usuario elegido.

Podemos dejar todos los valores predeterminados presionando Enter sin ingresar ningún valor nuevo a los parámetros hasta que la consola pregunte por la clave del usuario maestro, luego ya podremos ingresar comandos SQL para crear o modificar las **tablas** de la **base de datos PostgreSQL**.



```

SQL Shell (psql)      X + v
Server [localhost]:
Database [postgres]:
Port [5432]:
Username [postgres]:
Contraseña para usuario postgres:

psql (15.6)
ADVERTENCIA: El código de página de la consola (437) difiere del código
de página de Windows (1252).
Los caracteres de 8 bits pueden funcionar incorrectamente.
Vea la página de referencia de psql «Notes for Windows users»
para obtener más detalles.

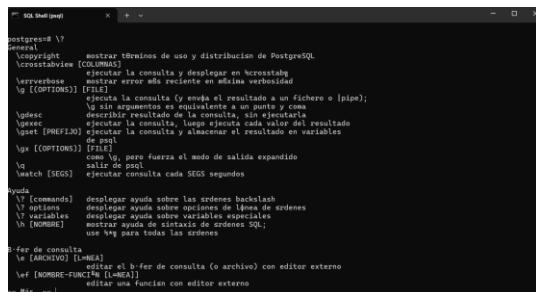
Digite «help» para obtener ayuda.

postgres=# |

```

## Comandos Comunes de la Consola de Postgres: SQL Shell

- **\?:** Comando que sirve para ver una lista con la descripción de todos los comandos disponibles de **Postgres**. Para ver más y más comandos debemos presionar la tecla Enter en la parte inferior que dice -- Más -- y al terminar de recorrer toda la lista, podremos introducir un nuevo comando debajo de este o simplemente podemos presionar las teclas **Ctrl + C**.



```

postgres# \?
General
\c [nombre]           mostrar términos de uso y distribución de PostgreSQL
\crosscheck            mostrar errores en tablas
\dd [esquema.]tbl     mostrar error más reciente en máxima verosimilitud
\g [OPTIONS]           [FILE]
                      ejecuta la consulta y envía el resultado a un fichero o |pipe|;
                      \g sin argumentos es equivalente a un punto y coma
\help [comando]        despliega ayuda sobre el comando, tabla, función, etc.
\pexec                ejecutar la consulta, luego ejecutar cada valor del resultado
\pset [NAMEVAL]         ejecutar la consulta y almacenar el resultado en variables
\q [OPTIONS]           [FILE]
                      para fuerza el modo de salida expandido
\q                     salir de psql
\water [SEGS]          ejecutar consulta cada SEGS segundos

Ayuda
\? [comando]           despliega ayuda sobre las ordenes backslash
\b [opciones]           despliega ayuda sobre opciones de líneas de órdenes
\c [variables]          despliega ayuda sobre variables especiales
\l [NOMBRE]             listar las bases de datos y sus usuarios de PostgreSQL SQL;
                      use \? para todas las ordenes

Operaciones
\f [ARCHIVO] [L=modo]   editar el -fer de consulta (o archivo) con editor externo
\ef [NOMBRE-FUNCION] [L=modo] editar una función con editor externo
-- Más -- |

```

- **\h**: Comando que sirve para enlistar todos los **comandos del lenguaje SQL** que se pueden ejecutar de forma directa dentro de la consola **SQL Shell**. Para ver más y más comandos debemos presionar la tecla Enter en la parte inferior que dice -- Más -- y al terminar de recorrer toda la lista, podremos introducir un nuevo comando debajo de este o simplemente podemos presionar las teclas **Ctrl + C**. Cabe mencionar que todas las funciones de SQL ejecutadas siempre deben terminar con un punto y coma (;).
- **\h nombre\_comando\_SQL**: Comando que sirve para describir la función y forma de ejecución de un comando SQL en específico.



```

SQL Shell (psql)      x  +  v
de página de Windows (1252).
Los caracteres de 8 bits pueden funcionar incorrectamente.
Vea la página de referencia de psql «Notes for Windows users»
para obtener más detalles.

Digite «help» para obtener ayuda.

postgres=# \h
Ayuda disponible:
    ABORT          CREATE USER MAPPING
    ALTER AGGREGATE CREATE VIEW
    ALTER COLLATION DEALLOCATE
    ALTER CONVERSION DECLARE
    ALTER DATABASE   DELETE
    ALTER DEFAULT PRIVILEGES DISCARD
    ALTER DOMAIN     DO
    ALTER EVENT TRIGGER DROP ACCESS METHOD
    ALTER EXTENSION  DROP AGGREGATE
    ALTER FOREIGN DATA WRAPPER DROP CAST
    ALTER FOREIGN TABLE DROP COLLATION
    ALTER FUNCTION   DROP CONVERSION
    ALTER GROUP      DROP DATABASE
    ALTER INDEX      DROP DOMAIN
    ALTER LANGUAGE   DROP EVENT TRIGGER
    ALTER LARGE OBJECT DROP EXTENSION
    ALTER MATERIALIZED VIEW DROP FOREIGN DATA WRAPPER
    ALTER OPERATOR   DROP FOREIGN TABLE
    ALTER OPERATOR CLASS DROP FUNCTION
    ALTER OPERATOR FAMILY DROP GROUP
    ALTER POLICY    DROP INDEX
    ALTER PROCEDURE  DROP LANGUAGE
    ALTER PUBLICATION DROP MATERIALIZED VIEW
    ALTER ROLE       DROP OPERATOR
    ALTER ROUTINE   DROP OPERATOR CLASS
    ALTER RULE      DROP OPERATOR FAMILY
    ALTER SCHEMA    DROP OWNED
    ALTER SEQUENCE  DROP POLICY
-- Más --

```

- **\l**: Comando que sirve para enlistar todas las **bases de datos** creadas en el motor **PostgreSQL**. Por defecto existe la base de datos `postgres` creada durante la instalación y dos más llamadas `template0` y `template1`.



```

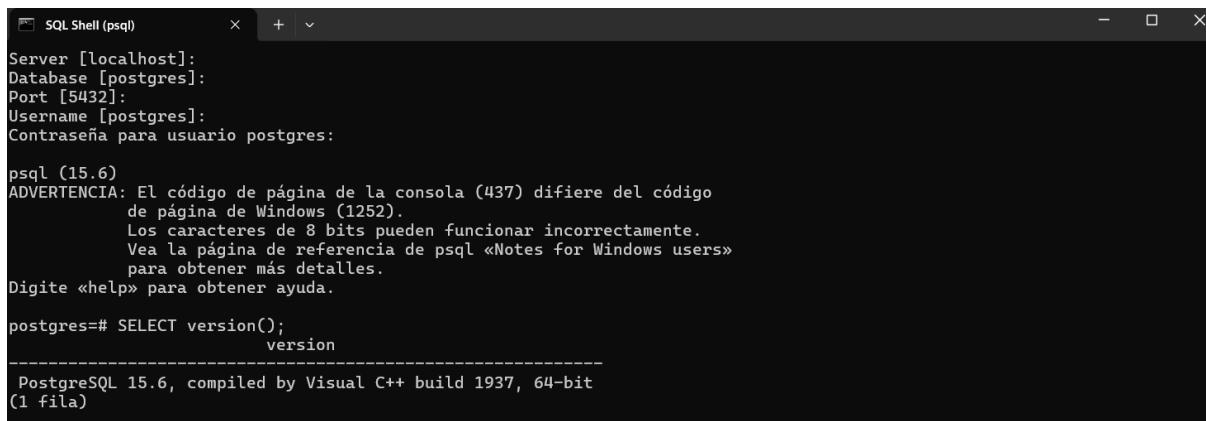
SQL Shell (psql)      x  +  v
postgres=# \l
                                         Listado de base de datos
   Nombre    | Dueño    | Codificación | Collate           | Ctype            | configuración ICU | Proveedor de loca
   | Privilegios
---+-----+-----+-----+-----+-----+-----+
postgres | postgres | UTF8        | Spanish_Mexico.1252 | Spanish_Mexico.1252 |                | libc
|         |
template0 | postgres | UTF8        | Spanish_Mexico.1252 | Spanish_Mexico.1252 |                | libc
|         |
|         |
|         |
template1 | postgres | UTF8        | Spanish_Mexico.1252 | Spanish_Mexico.1252 |                | libc
|         |
|         |
|         |
(3 filas)

postgres=#

```

- **\c nombre\_db**: Comando que sirve para cambiar a alguna otra **base de datos** dentro de la consola **SQL Shell**, para ello simplemente debemos indicar su nombre y al hacerlo veremos que cambia el nombre del puntero de la consola mencionado antes de los símbolos `=#`.

- **\dt**: Comando que sirve para enlistar todas las **tablas (entidades)** pertenecientes a la **base de datos** seleccionada. Este comando no nos permitirá ver las tablas de las bases de datos creadas por default (**postgres**, **template0**, **template1**), pero si nos permitirá ver las tablas de las demás.
  - **\d nombre\_tabla**: Comando que sirve para describir ciertas características (**columnas**, **sus tipos de datos**, **restricciones**, **relaciones**, **etc.**) de una **tabla (entidad)** que pertenece a la **base de datos** seleccionada actualmente.
  - **\dn**: Comando para enlistar los esquemas de la **base de datos** actual.
  - **\df**: Comando para enlistar las funciones disponibles de la **base de datos** actual.
  - **\dv**: Comando para enlistar las vistas de la **base de datos** actual.
  - **\du o \dg**: Comando para enlistar los usuarios y sus roles en la **base de datos** actual.
- **SELECT current\_date**;: Comando SQL para obtener la fecha actual.
- **SELECT version()**;: Comando SQL que sirve para ver la versión de **PostgreSQL** utilizada.



```

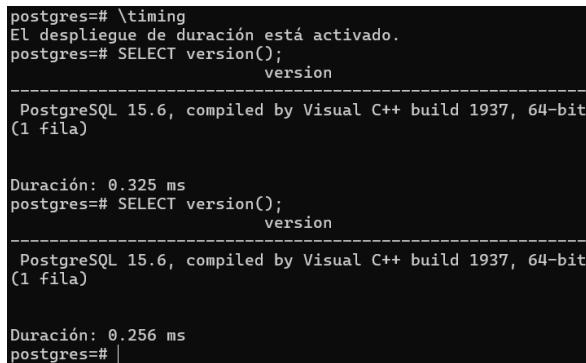
SQL Shell (psql)  X + 
Server [localhost]: Database [postgres]:
Port [5432]: Username [postgres]:
Contraseña para usuario postgres:

psql (15.6)
ADVERTENCIA: El código de página de la consola (437) difiere del código
de página de Windows (1252).
Los caracteres de 8 bits pueden funcionar incorrectamente.
Vea la página de referencia de psql «Notes for Windows users»
para obtener más detalles.

Digite «help» para obtener ayuda.

postgres=# SELECT version();
              version
-----
PostgreSQL 15.6, compiled by Visual C++ build 1937, 64-bit
(1 fila)
  
```

- **\g**: Comando que permite volver a ejecutar la última instrucción realizada, ya sea por nosotros o por algún otro usuario que se encuentre conectado a la misma base de datos.
- **\s**: Comando para ver el historial de comandos ejecutados
- **\timing**: Comando que inicializa un temporizador para indicarnos cuánto tiempo demorarán en ejecutarse los siguientes comandos.



```

postgres=# \timing
El despliegue de duración está activado.
postgres=# SELECT version();
              version
-----
PostgreSQL 15.6, compiled by Visual C++ build 1937, 64-bit
(1 fila)

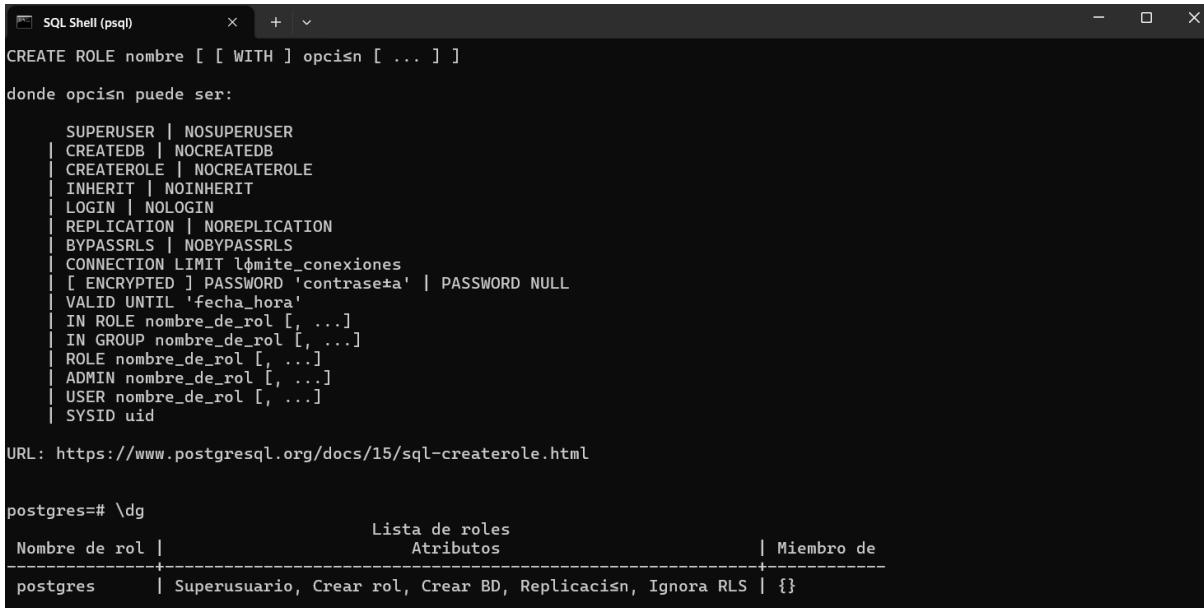
Duración: 0.325 ms
postgres=# SELECT version();
              version
-----
PostgreSQL 15.6, compiled by Visual C++ build 1937, 64-bit
(1 fila)

Duración: 0.256 ms
postgres=#
  
```

- **\! cls**: Comando para limpiar los comandos mostrados en la pantalla de la consola **SQL Shell**.
- **\q**: Comando para cerrar la consola
- **CREATE ROLE nombre\_rol WITH permisos\_asignados**;: Comando para la creación de **usuarios**.
- **ALTER ROLE nombre\_usuario WITH permisos\_asignados**;: Comando para la edición de **roles**.
- **DROP ROLE nombre\_rol WITH permisos\_asignados**;: Comando para la eliminación de **usuarios**.

- Los roles y usuarios se refieren a lo mismo y a estos se les pueden asignar ciertos permisos para que puedan o no realizar ciertas acciones como acceder a la base de datos (login), eliminar o editar tablas, heredar los permisos de otro usuario, etc. El rol predeterminado de PostgreSQL es postgres, el cual se declara durante la instalación del programa y es de tipo super usuario, por lo que posee todos los permisos existentes, pero se pueden crear usuarios que tengan los siguientes permisos específicos:
  - **SUPERUSER | NOSUPERUSER**: Si se asigna el permiso SUPERUSER, el usuario tendrá permisos sin restricciones que le permitirán realizar cualquier acción dentro del sistema de gestión de bases de datos, como la creación, edición o eliminación de nuevos usuarios, crear, editar o eliminar bases de datos, acceder a todas las entidades de todos los databases (osea acceder a todos los datos), ejecutar comandos que afecten la configuración y el funcionamiento del sistema de bases de datos y ejecutar comandos que interactúen con el sistema operativo y otros componentes de bajo nivel. Pero si se asigna NOSUPERUSER el rol será normal y sus accesos a la base de datos estarán limitados, por lo que se tendrán que declarar individualmente con los siguientes comandos.
    - **CREATEDB | NOCREATEDB**: Si se asigna el permiso CREATEDB, el rol podrá crear bases de datos, si se asigna NOCREATEDB el usuario no lo podrá hacer.
    - **CREATEROLE | NOCREATEROLE**: Si se asigna el permiso CREATEROLE, el rol podrá crear nuevos usuarios, si se asigna NOCREATEROLE, no lo podrá hacer.
    - **LOGIN | NOLOGIN**: Si se asigna el permiso LOGIN, el rol podrá acceder a las bases de datos, si se asigna NOLOGIN el usuario no lo podrá hacer.
    - **REPLICATION | NOREPLICATION**: Si se utiliza el permiso REPLICATION, se le otorga al rol la capacidad de iniciar sesiones de replicación en la base de datos. Esto es necesario para configurar y mantener la replicación entre servidores, permitiendo al rol acceder a funciones críticas para la sincronización de datos entre nodos. Si se asigna NOREPLICATION, se niega al usuario esta capacidad.
    - **BYPASSRLS | NOBYPASSRLS**: Si se asigna el permiso BYPASSRLS, se le otorga al rol la capacidad de ignorar las políticas de seguridad de nivel de fila (Row-Level Security, RLS) aplicadas a las tablas. Esto permite al usuario acceder a todas las filas de una tabla sin restricciones. Si se asigna NOBYPASSRLS, el rol debe cumplir con las RLS, limitando su acceso según las políticas definidas en la base de datos.
    - **CONNECTION LIMIT límite\_de\_conexiones**: Este comando especifica el número máximo de conexiones que el usuario puede tener abiertas de forma simultánea hacia la base de datos. Si se establece en -1, no hay límite en el número de conexiones que el rol puede tener.
    - **PASSWORD 'contraseña' | PASSWORD NULL**: Si se utiliza el permiso PASSWORD 'contraseña', se estará asignando la contraseña indicada entre comillas simples al usuario del comando, pero si se asigna PASSWORD NULL el usuario no poseerá ninguna contraseña y por lo tanto no podrá acceder a la base de datos.

- **UNTIL 'fecha\_hora'**: Este comando define una **fecha y hora de expiración para el usuario**, después de ella, el **rol** no podrá iniciar sesión en la **database**. La fecha y hora se especifican en un formato de **año-mes-día hora24hrs:mins:seg**s, ya que este es reconocible para **PostgreSQL**.
- **IN ROLE nombre\_de\_rol**: Este comando asigna el nuevo **rol** como miembro de uno o más roles existentes, lo cual hace que herede los permisos del **usuario** al que fue asignado. El rol padre por default del que se puede heredar es **postgres**.
  - **INHERIT | NOINHERIT**: Si se utiliza el permiso **INHERIT**, el **rol** heredará automáticamente todos los **privilegios** de los **usuarios** de los cuales es miembro. Pero si se asigna **NONINHERIT**, este solo poseerá los **privilegios** que se le hayan otorgado directamente a través de comandos y no los de sus **roles padres**.
- **ADMIN nombre\_de\_rol**: Define **uno o más roles** que serán miembros de este usuario y que, por lo tanto, heredarán sus **permisos**.
- **SYSID id**: Define un **identificador específico para el rol** que sea único en el sistema de **base de datos PostgreSQL**.
  - **\du o \dg**: Comando para enlistar los usuarios y sus roles en la **base de datos** actual.



```

SQL Shell (psql)
CREATE ROLE nombre [ [ WITH ] opcion [ ... ] ]
donde opcion puede ser:
  SUPERUSER | NOSUPERUSER
  | CREATEDB | NOCREATEDB
  | CREATEROLE | NOCREATEROLE
  | INHERIT | NOINHERIT
  | LOGIN | NOLOGIN
  | REPLICATION | NOREPLICATION
  | BYPASSRLS | NOBYPASSRLS
  | CONNECTION LIMIT límite_conexiones
  | ENCRYPTED ] PASSWORD 'contraseña' | PASSWORD NULL
  | VALID UNTIL 'fecha_hora'
  | IN ROLE nombre_de_rol [ , ... ]
  | IN GROUP nombre_de_rol [ , ... ]
  | ROLE nombre_de_rol [ , ... ]
  | ADMIN nombre_de_rol [ , ... ]
  | USER nombre_de_rol [ , ... ]
  | SYSID uid

URL: https://www.postgresql.org/docs/15/sql-createrole.html

postgres=# \dg
          Lista de roles
  Nombre de rol | Atributos           | Miembro de
  postgres      | Superusuario, Crear rol, Crear BD, Replicaci n, Ignora RLS | {}

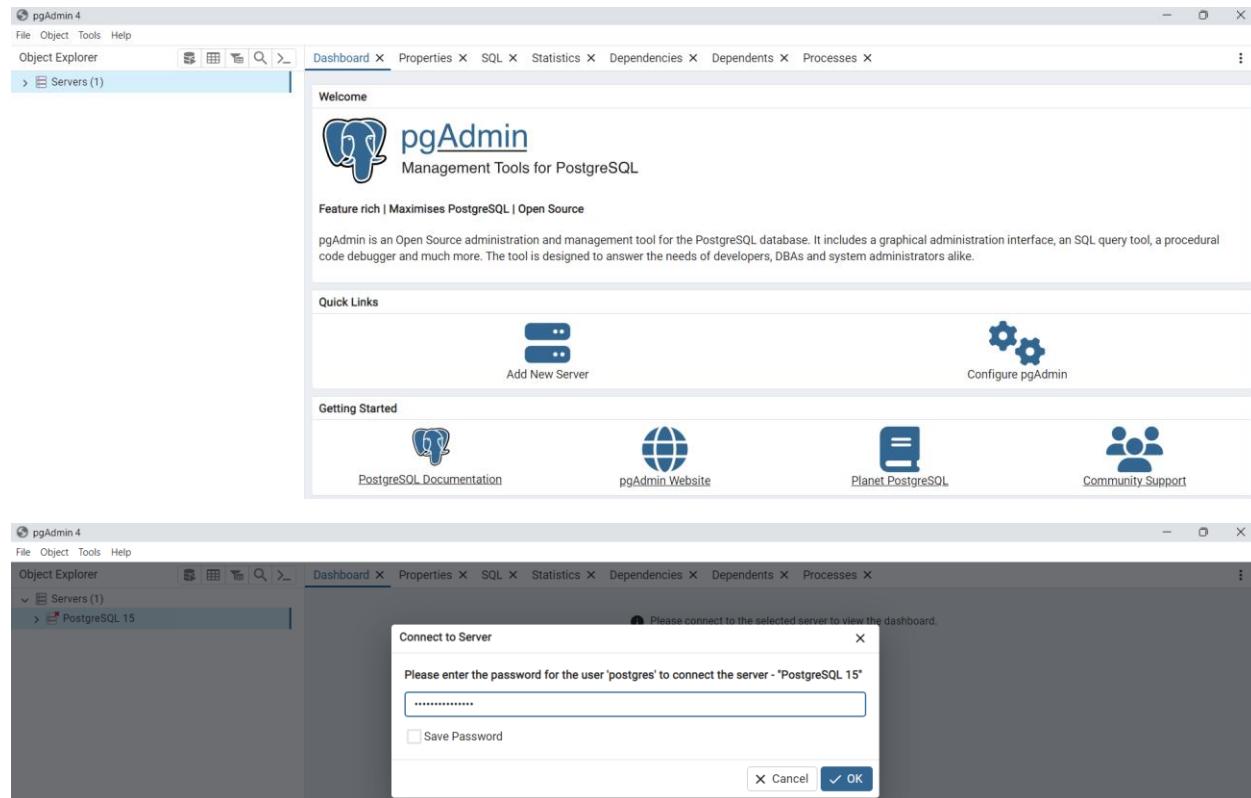
```

## Funcionalidades Comunes de la Interfaz Gráfica de PostgreSQL: pgAdmin

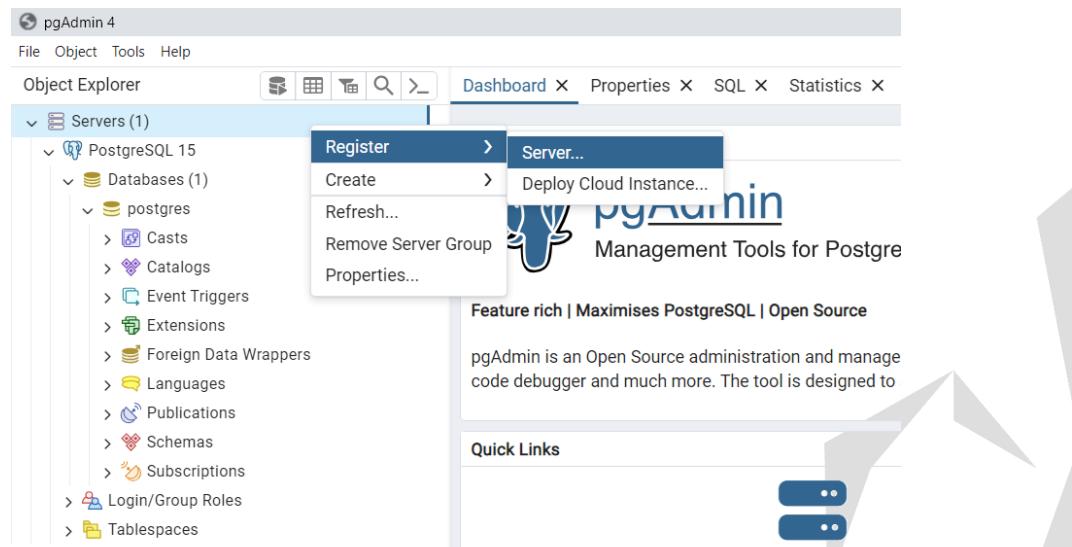
Para acceder a la interfaz simplemente debemos buscar y ejecutar la aplicación de escritorio llamada **pgAdmin**, ésta a veces se puede abrir en una pantalla emergente o abrirse a través de algún navegador web, cualquiera de las dos opciones funciona y se ejecuta en una u otra dependiendo de la versión que hayamos instalado.

En la esquina superior izquierda podemos ver un desplegable que dice **Servers**, al dar clic sobre este podremos acceder a cada servidor que esté conectado a **pgAdmin** a través de una dirección IP, que se

representa por una cara de elefantino, el cual luego despliega todas las **bases de datos** creadas dentro de cada servidor, tanto las predeterminadas (**postgres**, **template0**, **template1**), como las que creamos manualmente. Para poder acceder al contenido de **Servers**, se nos pedirá que introduzcamos la contraseña asignada a la base de datos **postgres**, la cual se indicó durante la instalación del programa.



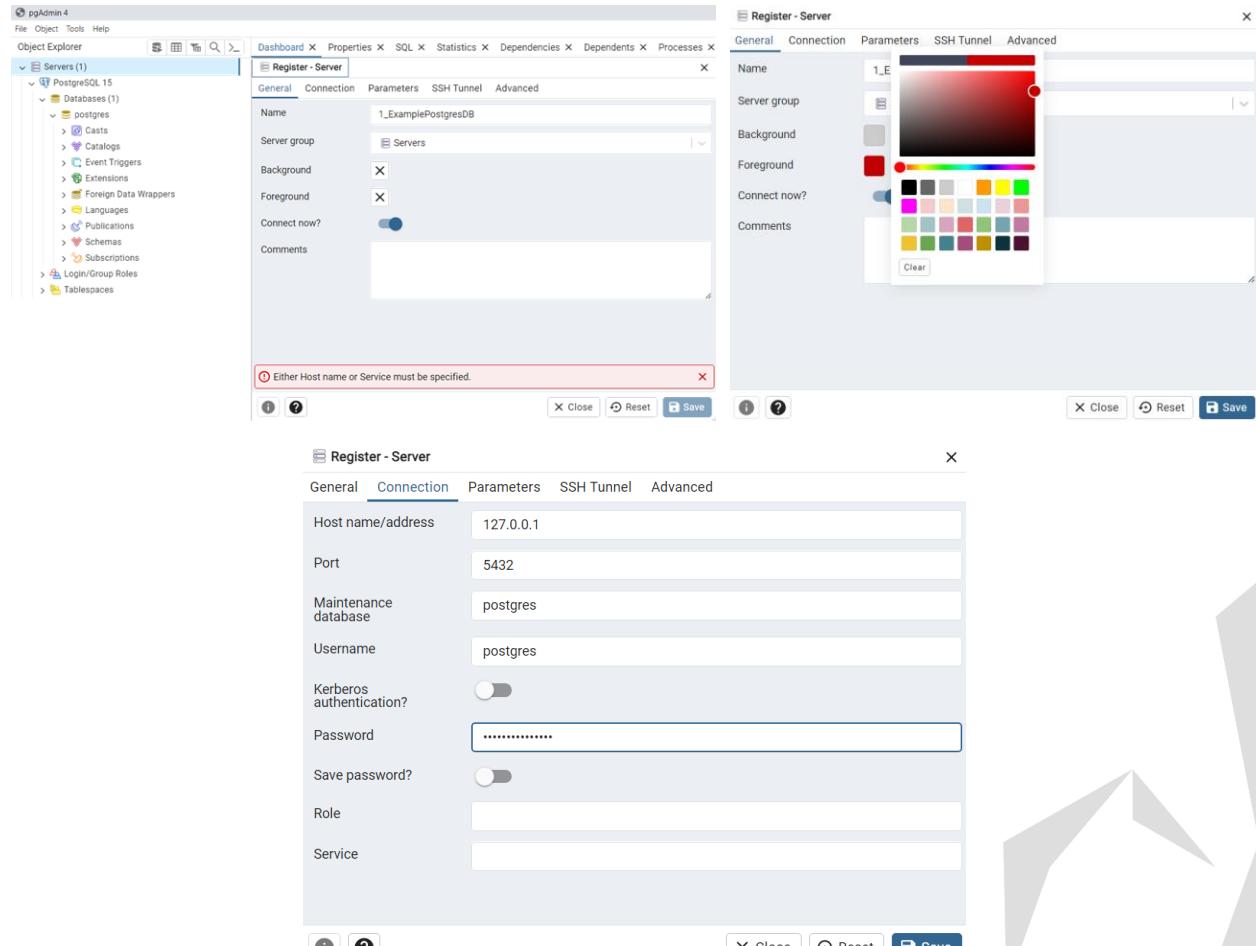
Una vez que nos encontremos dentro de la interfaz, podemos conectar nuevos computadores o servicios de nube que tengan el **motor de base de datos PostgreSQL** instalado y en ejecución al RDBMS (Relational DataBase Management System) de **pgAdmin** dando clic derecho sobre el ícono de **Servers** y seleccionando la opción de **Servers → Clic Derecho → Register → Server**.



Para posteriormente indicar los siguientes parámetros de conexión de la nueva **conexión** que queremos crear:

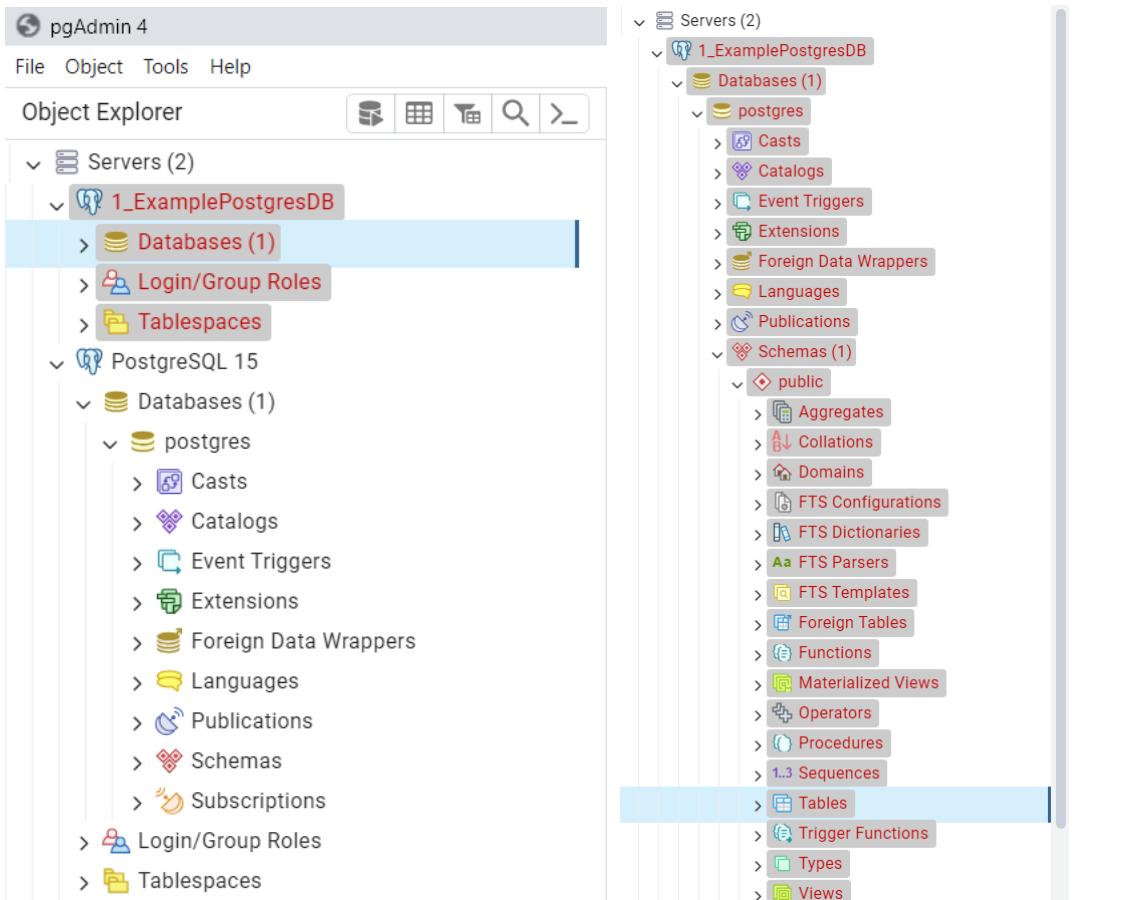
- **Host name/address:** En este parámetro se indica la dirección IP donde se encuentra localizada la **base de datos**, se puede utilizar **localhost**, que corresponde a la dirección IP **127.0.0.1** (representan lo mismo).
- **Port:** En este parámetro se indica el número de puerto de conexión al servidor, el cual se declara durante la instalación de **PostgreSQL** y por defecto es el 5432, aunque si queremos, aquí podemos declarar otro puerto.
- **Maintenance database:** Aquí se indica el nombre de la **database** predeterminada que posee el servidor, en un inicio esta corresponde por default a la base de datos **postgres**, la cual fue declarada durante la instalación de **PostgreSQL**.
- **Username:** Aquí se indica el **nombre de usuario** utilizado para realizar la conexión con el servidor de la **DB**, este por defecto es **postgres**, el cual fue declarado durante la instalación de **PostgreSQL**, aunque podemos declarar uno nuevo, pero se deberá indicar su contraseña debajo.
- **Contraseña para el usuario de postgres:** Si el usuario elegido es **postgres**, la contraseña será la declarada durante la instalación de **PostgreSQL**, sino se declarará una nueva.

Además, por fines estéticos podemos asignar algún color de fondo y demás aspectos.

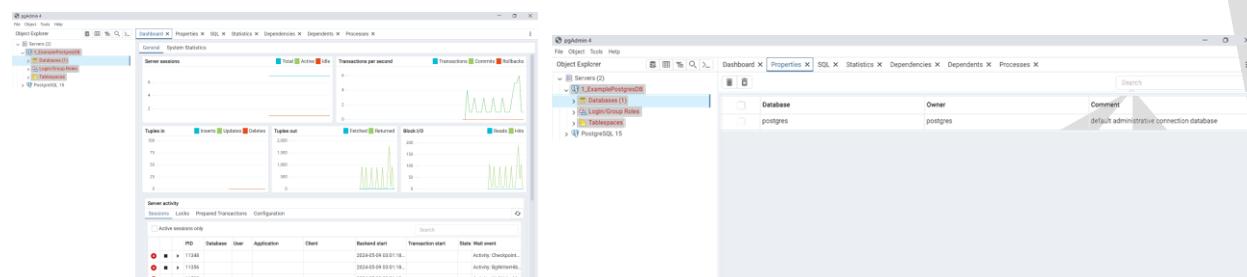


Y ahora tendremos dos accesos a diferentes **bases de datos** con distintas conexiones a servidores **PostgreSQL**, el **servidor** predeterminado se llama **PostgreSQL 15** y nuestro **propio servidor local** fue llamado **1\_ExamplePostgresDB**, estos accesos nos permiten ver las siguientes características:

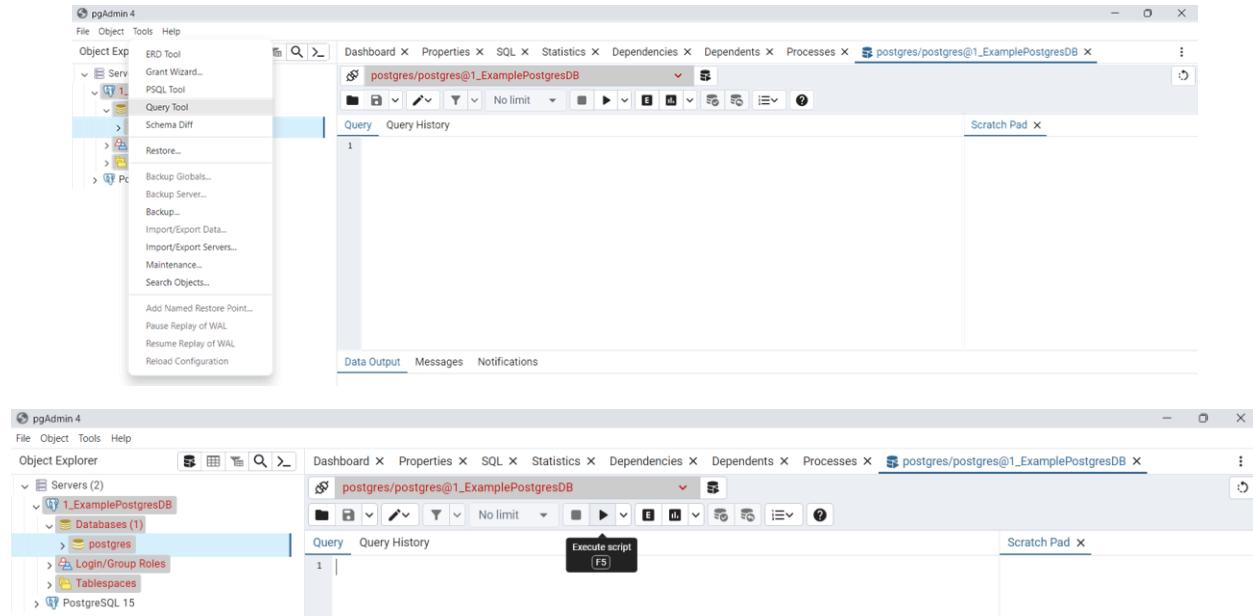
- **Databases**: Nos deja ver información de las **databases**, varias opciones internas son propias de **PostgreSQL**, pero la que nos interesa para poder ver las **tablas** de la **base de datos** es **Schemas** → **Tables** → **Nombre de la Tabla a la que quiero acceder**.
- **Logins/group roles**: Representa a todos los **usuarios** o **grupos de usuarios** con diferentes **permisos** que pueden acceder a la **database**, incluido el declarado por default llamado **postgres**.
- **Tablespaces**: Es simplemente un espacio de memoria físico para guardar la información de la **base de datos**, así como en nuestro ordenador tenemos discos duros C:, D:, etc.



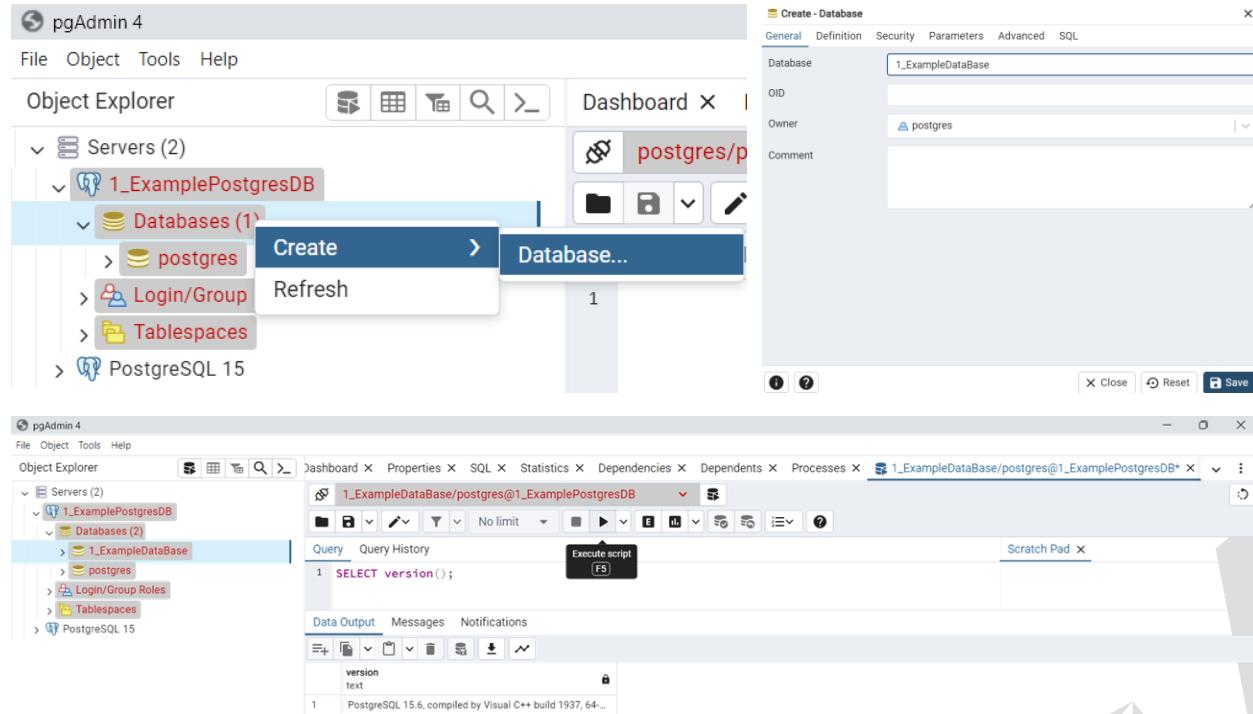
Podemos además observar varias otras características de la base de datos como sus transacciones por segundo, propiedades, código SQL, estadísticas, etc.



Y para introducir de forma directa código SQL seleccionamos la **base de datos** → Tools → Query Tool. Y si queremos ejecutar una consulta damos clic en el botón de Execute script.



Si queremos crear una nueva base de datos debemos dar clic en la opción **Databases** → Create → Database... → Database → Save. Y aquí podemos ejecutar cualquier código de SQL.



La creación de tipos de **usuarios (o roles, que es lo mismo)** por medio de **pgAdmin** se realiza al dar clic en la opción de: **Servers** → **Motor PostgreSQL** → **Login/Group Roles** → Clic Derecho → Create → **Login/Group Role...** → **Pestaña General** → Name: **nombre\_rol** → **Pestaña Definition** → Password: Contraseña.

The screenshot shows the pgAdmin 4 interface. On the left, the Object Explorer pane displays a tree structure with servers, databases, and roles. A context menu is open over the 'Login/Group Roles' node under '1.ExamplePostgresDB', with options like 'Create', 'Refresh', and 'Login/Group Role...'. The main window shows two tabs: 'General' and 'Definition'. The 'General' tab has a 'Name' field set to 'di\_cero\_consulta' and a 'Comments' field with the text: 'Este usuario sirve para realizar solo consultas, sin poder cambiar nada en la base de datos de PostgreSQL.'. The 'Definition' tab shows a password field with '.....', an account expiry field set to 'No Expiry', and a connection limit field set to '-1'.

Después de haber asignado las **características básicas** del **usuario**, podemos indicar sus **permisos** individuales al seleccionar la opción de: Pestaña **Privileges** → Asignar **permisos** → Pestaña **Membership** → Asignar **grupos de usuarios**, donde los **roles hijos** heredan los **privilegios** de los **usuarios padres** → Pestaña **SQL** → Observar el código SQL equivalente a las propiedades configuradas del **usuario** → **Save**:

- **Can login?**: Si se asigna el permiso **LOGIN**, el **rol** podrá **acceder a las bases de datos**, si no, no lo podrá hacer.
- **Superuser?**: Si se asigna el permiso **Superuser**, el **usuario** tendrá permisos sin restricciones que le permitirán realizar cualquier acción dentro del sistema de gestión de bases de datos, como la **creación, edición o eliminación de nuevos usuarios**, **crear, editar o eliminar bases de datos**, **acceder a todas las entidades de todos los databases** (osea **acceder a todos los datos**), ejecutar comandos que afecten la configuración y el funcionamiento del **sistema de bases de datos** y

ejecutar comandos que interactúen con el **sistema operativo y otros componentes de bajo nivel**. Si no, el rol será normal y sus accesos a la base de datos estarán limitados, por lo que se tendrán que declarar individualmente.

- **Create roles?**: Si se asigna el permiso **Create roles**, el **rol** podrá **crear nuevos usuarios**, si no, no lo podrá hacer.
- **Create databases?**: Si se asigna el permiso **Create databases**, el **rol** podrá **crear bases de datos**, si no, el **usuario** no lo podrá hacer.
- **Inherit rights from the parent roles?**: Si se utiliza el permiso **Inherit**, el **rol** heredará automáticamente todos los **privilegios** de los **usuarios** de los cuales es miembro. Pero si no, este solo poseerá los **privilegios** que se le hayan otorgado directamente a través de comandos y no los de sus **roles padres**. El padre por default es el **usuario postgres**.
- **Can initiate streaming replication and backups?**: Si se utiliza el permiso **Replication**, se le otorga al **rol** la capacidad de **iniciar sesiones de replicación** en la **base de datos**. Esto es necesario para configurar y mantener la replicación entre servidores, permitiendo al rol acceder a funciones críticas para la sincronización de datos entre nodos. Si no, se niega al **usuario** esta capacidad.

**Group Role - Login/Group Roles (Privileges Tab)**

Setting	Value
Can login?	Off
Superuser?	Off
Create roles?	Off
Create databases?	Off
Inherit rights from the parent roles?	On
Can initiate streaming replication and backups?	Off

**Group Role - Login/Group Roles (Membership Tab)**

Role	Type	Privilege
User/Role	WITH ADMIN	
Members		
User/Role	WITH ADMIN	

**Group Role - Login/Group Roles (SQL Tab)**

```

1 CREATE ROLE di_cero_consulta WITH
2   LOGIN
3   NOSUPERUSER
4   NOCREATEDB
5   NOCREATEROLE
6   INHERIT
7   NOREPLICATION
8   CONNECTION LIMIT -1
9   PASSWORD 'xxxxxxxx';
10 COMMENT ON ROLE di_cero_consulta IS 'Este usuario sirve para realizar consultas';
  
```

Y finalmente, para poder asignar el usuario creado a una tabla de la base de datos con el fin de darle acceso a ella se debe seleccionar la siguiente opción: Databases → **nombre de la base de datos** → **Schemas** → public → **Tables** → Clic derecho → Grant Wizard... → Seleccionamos todas las **tablas** en el checkbox de Object Type → Next → + (Add row) → **Grantee**: Indicar el **usuario o rol** al que se le quiere dar acceso a las **tablas** de la **base de datos** → Privileges: Indicar los **permisos** que el **rol Grantor** (usualmente es **postgres**) le asigna al nuevo **usuario** → Next → Checar Código SQL equivalente → Finish.

The screenshot shows the PostgreSQL pgAdmin interface with the 'Grant Wizard' open. The process consists of three steps:

- Object Selection:** Shows a list of tables in the 'public' schema selected for granting. The tables listed are: estacion, pasajero, trayecto, tren, and viaje.
- Privilege Selection:** Shows the 'di\_cero\_consulta' user selected as the grantee. A dropdown menu indicates the grantor is 'postgres'. A note at the bottom states: "'Grantee in Privileges' cannot be empty."
- Review:** Displays the generated SQL commands for granting privileges. The SQL code is as follows:

```

1 GRANT INSERT, UPDATE, SELECT ON TABLE public.estacion TO di_cero_consulta;
2
3 GRANT INSERT, UPDATE, SELECT ON TABLE public.pasajero TO di_cero_consulta;
4
5 GRANT INSERT, UPDATE, SELECT ON TABLE public.trayecto TO di_cero_consulta;
6
7 GRANT INSERT, UPDATE, SELECT ON TABLE public.tren TO di_cero_consulta;
8
9 GRANT INSERT, UPDATE, SELECT ON TABLE public.viaje TO di_cero_consulta;
10
11

```

Y así el **usuario** creado ya tiene **permisos** para acceder a las **tablas** del **database**, esto se ve reflejado al ver el código SQL de las tablas.

```

1 -- Table: public.estacion
2
3 -- DROP TABLE IF EXISTS public.estacion;
4
5 CREATE TABLE IF NOT EXISTS public.estacion
6 (
7     id_estacion integer NOT NULL DEFAULT nextval('estacion_id_estacion_seq'::regclass),
8     nombre_estacion character varying COLLATE pg_catalog."default",
9     direccion_estacion character varying COLLATE pg_catalog."default",
10    CONSTRAINT estacion_pkey PRIMARY KEY (id_estacion)
11 )
12
13 TABLESPACE pg_default;
14
15 ALTER TABLE IF EXISTS public.estacion
16     OWNER to postgres;
17
18 REVOKE ALL ON TABLE public.estacion FROM di_cer0_consulta;
19
20 GRANT UPDATE, INSERT, SELECT ON TABLE public.estacion TO di_cer0_consulta;
21
22 GRANT ALL ON TABLE public.estacion TO postgres;

```

## Diseño del Diagrama ER De una Base de Datos Relacional - PostgreSQL

En el ejercicio que se llevará a cabo se realiza el diseño de la **base de datos** de una **estación de transporte masiva** como lo puede ser un tren, metro o algo así. Al finalizar el diseño se llegará al **diagrama ER (entidad-relación)** y **diagrama físico** de la **database relacional**. En ambos diagramas se podrán observar todas las **entidades (tablas)** y **relaciones (conexiones)** que la conforman y en el **diagrama físico** hasta se describirá el **tipo de dato** de cada uno de sus **atributos (columnas de la tabla)**.



Para llevar a cabo el diseño de los diagramas de la **DB (DataBase)** podemos empezar a pensar en los **objetos** que poseen todos los **sistemas de transportes masivos**, los cuales pueden ser modelados como las **entidades (tablas)** de la **base de datos**, después debemos pensar en las **características individuales** importantes de dichos objetos, las cuales se modelarán como sus **columnas (atributos)** y luego estos tendrán que ser asociados entre sí a través de **verbos de relación**, indicando su **cardinalidad (1:N, N:N, 1:1, etc.)**, osea cuántas **instancias o filas** de una **tabla** le corresponden a las **tuplas** de la otra **entidad**.

## *Tipos de Datos Especiales de PostgreSQL*

Además, antes de empezar a modelar cabe mencionar que los **tipos de datos** que pueden adoptar los datos de las **columnas** en las **DB** de **PostgreSQL** son los siguientes:

- **PRINCIPALES:**
  - **Texto:**
    - **Char(n):** Almacena cadenas de caracteres de longitud fija, ocupando siempre n caracteres de espacio en disco.
    - **VarChar(n):** Almacena cadenas de caracteres de longitud variable, reservando como mínimo un espacio de memoria y extendiéndolo si es necesario hasta n caracteres.
      - **character varying(n) y character(n):** Alternativas de **PostgreSQL** equivalentes a **Char(n)** y **VarChar(n)** respectivamente, donde la longitud máxima es descrita por n.
    - **Text:** Almacena cadenas de caracteres (palabras u oraciones) de longitud arbitraria.
  - **Numéricos:**
    - **Enteros:** Integer (4 bytes), BigInt (8 bytes) y SmallInt (2 bytes).
    - **Decimales:** Se declara Decimal(n, s) o Numeric(n, s), ambos son esencialmente lo mismo, donde n es el número total de dígitos y s es el número de dígitos a la derecha del punto decimal.
    - **Serial:** Valor entero auto incremental que usualmente se asigna a los id.
  - **Monetarios:**
    - **Money:** Almacena valores monetarios.
  - **Binarios:**
    - **Binary:** Almacena datos binarios de longitud fija.
  - **Fecha/Hora:**
    - **Date:** Contiene año, fecha y día.
    - **Time:** Contiene solo la hora.
    - **Datetime y Timestamp:** Contienen la fecha y la hora, pero pueden diferir en la precisión y el rango de valores admitidos.
  - **Lógicos:**
    - **Boolean:** Puede adoptar valores true (1) o false (0).
- **ESPECIALES DE PostgreSQL:**
  - **Geométricos:**
    - **Geometric:** Permiten usando coordenadas (x, y) calcular distancias y áreas.
  - **Dirección de Red:**
    - **Inet/Cidr:** Almacenan direcciones IP para realizar cálculos de máscaras de red.
  - **Texto tipo Bit:**
    - **Bit:** Este tipo de dato permite hacer cálculos en otros sistemas numéricos que no sea el decimal, como el binario o hexadecimal.
  - **XML o JSON:**
    - **XML/JSON:** Sirve para recibir datos transmitidos por APIs.

- **Arreglos (Arrays):**

- **Arrays:** Son vectores o matrices que permiten realizar cálculos vectoriales más complejos.

El siguiente enlace contiene más información: <https://www.postgresql.org/docs/11/datatype.html>

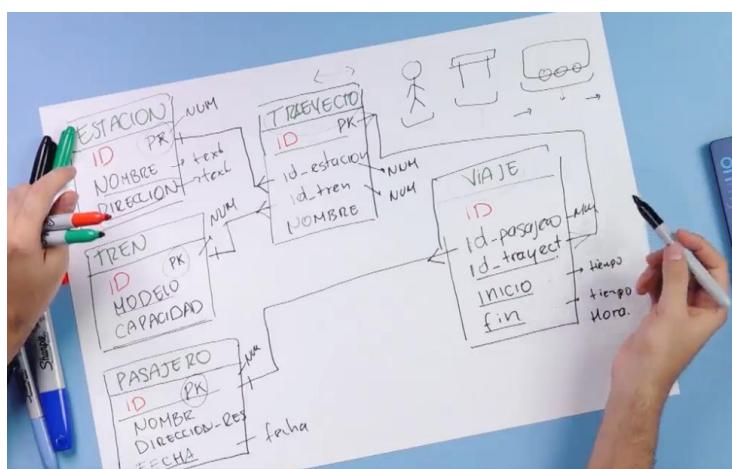
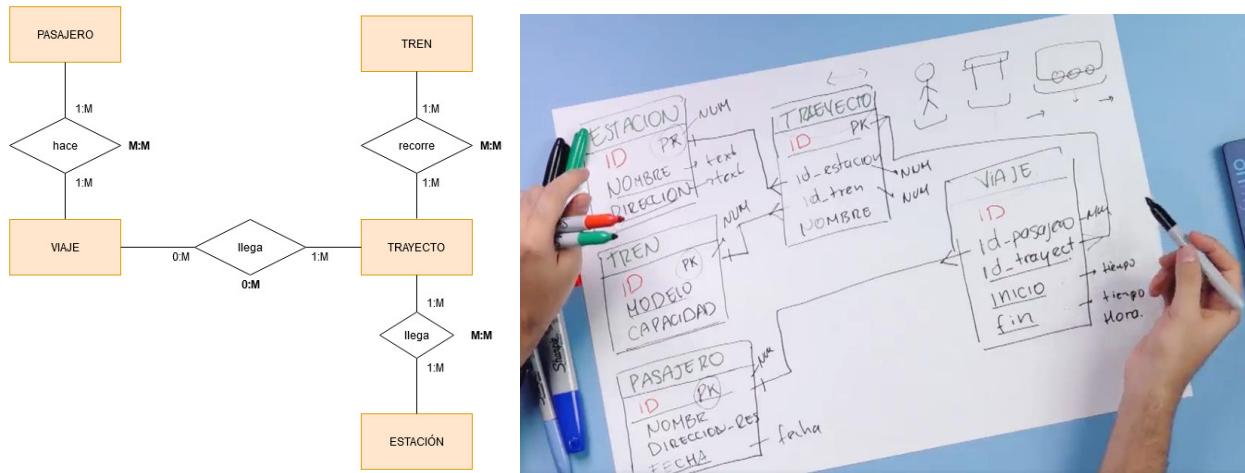
Y de igual forma se deben declarar los **constraints**, los cuales indican **características de las columnas**:

## Constraints (Restricciones)

Constraint	Descripción
NOT NULL	Se asegura que la columna no tenga valores nulos
UNIQUE	Se asegura que cada valor en la columna no se repita
PRIMARY KEY	Es una combinación de NOT NULL y UNIQUE
FOREIGN KEY	Identifica de manera única una tupla en otra tabla
CHECK	Se asegura que el valor en la columna cumpla una condición dada
DEFAULT	Coloca un valor por defecto cuando no hay un valor especificado
INDEX	Se crea por columna para permitir búsquedas más rápidas

Ya habiendo considerado los pasos de diseño descritos anteriormente, llegamos al siguiente resultado, tomando en cuenta los objetos principales y luego objetos que los relacionen, para ello siempre es bueno hacer este proceso inicial en forma de papel para ver de forma gráfica sus conexiones:

- **Pasajeros:** **Id\_Pasajero (Primary Key)**, **Nombre**, **Dirección de Residencia**, **Fecha de Nacimiento**.
- **Estaciones:** **Id\_Estación (Primary Key)**, **Nombre de Estación**, **Dirección de la Estación**.
- **Trenes:** **Id\_Tren (Primary Key)**, **Modelo del Tren**, **Capacidad de Pasajeros**.
  - **Trayectos:** **Id\_Trayecto (Primary Key)**, **Id\_Estación**, **Id\_Tren**, **Nombre de Ruta**.
    - **Viajes:** **Id\_Viaje (Primary Key)**, **Id\_Pasajero**, **Id\_Trayecto**, **Tiempo de Inicio del Viaje**, **Tiempo Final del Viaje**.



## Sub-lenguajes de SQL: DDL (Data Definition Language)

Los dos sub-lenguajes más importantes del lenguaje SQL son llamados **DDL** y **DML**, primero se explicará el **DDL** que sirve para estructurar la base de datos:

- **DDL (Data Definition Language):** La función principal de este sub-lenguaje de SQL es la de crear la estructura de una **database**. Esto se refiere a establecer las **entidades, atributos, relaciones, etc.** que son descritos en un **diagrama ER** o en un **diagrama físico**. Los **3 comandos** con los que se cuenta para llevar a cabo la estructuración de datos con el lenguaje **DDL** son los siguientes:
  - **CREATE:** Comando utilizado para crear una **base de datos, tabla, vista, índice, etc.**
  - **ALTER:** Comando que sirve para modificar una **entidad, tabla, tipo de dato, etc.**
  - **DROP:** Comando para **eliminar elementos** de una **base de datos**. Esta se debe utilizar con mucho cuidado, ya que accidentalmente podríamos borrar nuestra **DB** completa.

**Algunos de los elementos u objetos** que se van a manipular con los **comandos DDL** son los siguientes:

- **DATABASE:** Se refiere a la **base de datos**, la cual también se puede llamar **SCHEMA**.
- **TABLE:** Las **tablas** se refieren a la visualización de los datos después de haberlos modelado a través de un **diagrama ER** o **diagrama físico**.
- **VIEW:** Las vistas se refieren a la forma en la que se pueden interpretar y ordenar los datos extraídos de una **database** al realizar un **Query**. De tal forma que puedan ser utilizados por el usuario, ya que, en las **tablas** de las **DB**, muchas veces la información está segmentada y con las **vistas** la podemos filtrar y organizar en un objeto diferente.

## CREATE:

### *Crear una Base de Datos con SQL*

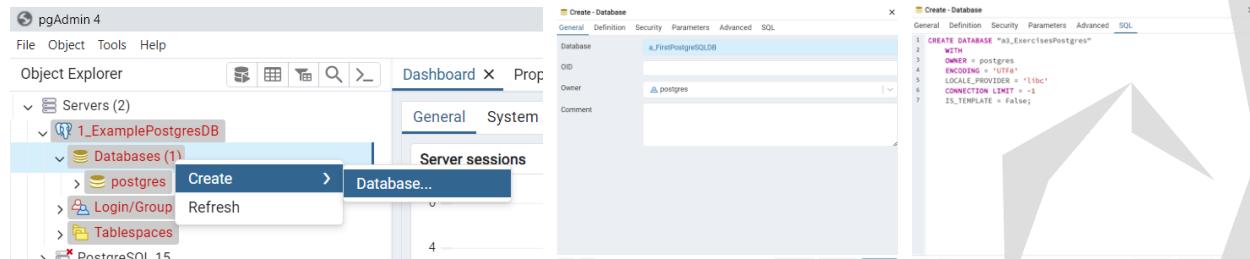
Siempre que se quiera crear una **base de datos nueva** se deberán ejecutar los siguientes comandos, ya sea en el manejador de bases de datos **pgAdmin** de **PostgreSQL** o directamente en su consola **SQL Shell**, muchas veces es mejor utilizar la consola en vez de la interfaz gráfica, ya que esta última presenta limitaciones o bugs que no se pueden resolver desde la interfaz, solamente desde consola:

**CREATE DATABASE**      “Nombre\_Base\_de\_Datos”;

--La siguiente instrucción no siempre se incluye, porque se da por entendida.

**USE      DATABASE**      “Nombre\_Base\_de\_Datos”;

Para crear una **database** en **Postgres**, seleccionaremos la opción de: Databases → Clic derecho → Create → Database... → Database (Nombre) → Save. Además, si queremos ver el código SQL que crea la base de datos podemos seleccionar la pestaña **SQL** para verlo y este mismo podría ser ejecutado en la consola **SQL Shell** para obtener el mismo resultado.

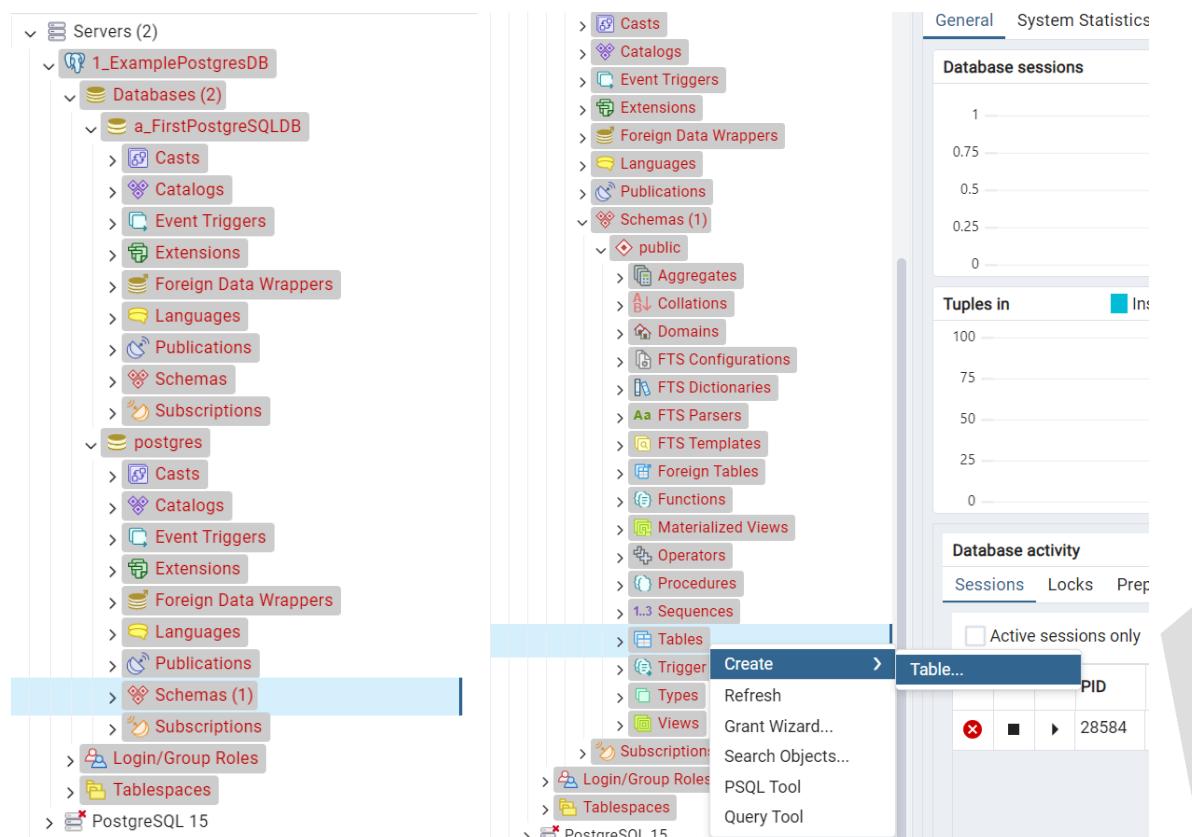


## Crear una Tabla con SQL

Cuando se quiera crear una nueva tabla en una base de datos se deberá ejecutar el siguiente comando, ya sea en un Manejador de Bases de Datos Relacionales (RDBMS) o en una Base de Datos con Servicios Administrados (Nube o Cloud), aunque vale la pena mencionar que es importante previamente tener ya listo el **diagrama ER** y/o **diagrama físico de la base de datos**, para seguir dicha estructura:

```
CREATE TABLE Nombre_Entidad_o_Tabla(  
    Nombre_Atributo      Tipo_de_Dato_y_Constraints ,  
    Nombre_Atributo      Tipo_de_Dato_y_Constraints  
);
```

Ahora para crear una nueva **entidad** dentro de la **base de datos** creada, debemos dar clic en las siguientes opciones del desplegable: **Servers** → **Motor de base de datos PostgreSQL** → **Databases** → **Nombre\_Database** → **Schemas** → **Public** → **Tables** → Clic Derecho → **Create** → **Table...** → **Pestaña General** → **Name** (nombre **tabla**) → **Pestaña Columns** → **+ (Add row)** → Crear una a una los **atributos** de la **entidad** indicando su **tipo de dato** y **constraints básico** → **Pestaña Constraints** → **+ (Add row)** → Indicar **constraints especiales**, como por ejemplo cuál **columna** es un **Primary Key** con el formato **nombreColumna\_pkey** → **Pestaña SQL** (Ver código) → **Save**.



Recordemos que el **tipo de dato serial** de **PostgreSQL** indica que este es un **indicador numérico auto incremental** (que subirá su valor de forma automática) para identificar individualmente todas las **filas** de la **columna** de la **tabla** sin necesidad de modificarlo.

```

CREATE TABLE public.pasajero
(
    id_pasajero serial,
    nombre character varying(100),
    direccion_residencia character varying,
    fecha_nacimiento date
)
CONSTRAINT pasajero_pkey PRIMARY KEY (id_pasajero);

ALTER TABLE IF EXISTS public.pasajero
OWNER to postgres;

```

La ventaja de utilizar un cliente gráfico es que se puede visualizar el resultado del código.

Aunque si se cuenta ya con un código SQL que cree una tabla e inserte ciertos datos, conviene utilizar la opción de: **Servers** → **Motor de base de datos PostgreSQL** → **Databases** → **Nombre\_Database** → **Schemas** → Clic Derecho → **Create** → **Schema...** → **Pestaña General** → **Name (nombre tabla)** → **Save** → **Tools** → **Query Tool** → Pegar código SQL de creación de tablas e inserción de datos → **Execute script**.

```

CREATE TABLE ejercicios.carreras (
    id INT,
    carrera VARCHAR(100),
    fecha_alta TIMESTAMP,
    vigente BOOLEAN
);

INSERT INTO ejercicios.carreras (id, carrera, fecha_alta, vigente) VALUES (1, 'Negocios y administración', 1990-01-01, true);
INSERT INTO ejercicios.carreras (id, carrera, fecha_alta, vigente) VALUES (2, 'Administración', 1990-01-01, true);
INSERT INTO ejercicios.carreras (id, carrera, fecha_alta, vigente) VALUES (3, 'Contabilidad y finanzas', 1990-01-01, true);
INSERT INTO ejercicios.carreras (id, carrera, fecha_alta, vigente) VALUES (4, 'Finanzas, banca y seguros', 1990-01-01, true);
INSERT INTO ejercicios.carreras (id, carrera, fecha_alta, vigente) VALUES (5, 'Mercadotecnia y marketing', 1990-01-01, true);
INSERT INTO ejercicios.carreras (id, carrera, fecha_alta, vigente) VALUES (6, 'Negocios y administración', 1990-01-01, true);
INSERT INTO ejercicios.carreras (id, carrera, fecha_alta, vigente) VALUES (7, 'Negocios y administración', 1990-01-01, true);
INSERT INTO ejercicios.carreras (id, carrera, fecha_alta, vigente) VALUES (8, 'Ingeniería civil', 1990-01-01, true);
INSERT INTO ejercicios.carreras (id, carrera, fecha_alta, vigente) VALUES (9, 'Ciencias políticas', 1990-01-01, true);
INSERT INTO ejercicios.carreras (id, carrera, fecha_alta, vigente) VALUES (10, 'Economía', 1990-01-01, true);
INSERT INTO ejercicios.carreras (id, carrera, fecha_alta, vigente) VALUES (11, 'Psicología', 1990-01-01, true);
INSERT INTO ejercicios.carreras (id, carrera, fecha_alta, vigente) VALUES (12, 'Sociología y antropología', 1990-01-01, true);
INSERT INTO ejercicios.carreras (id, carrera, fecha_alta, vigente) VALUES (13, 'Trabajo y atención social', 1990-01-01, true);
INSERT INTO ejercicios.carreras (id, carrera, fecha_alta, vigente) VALUES (14, 'Ciencias de la salud', 1990-01-01, true);
INSERT INTO ejercicios.carreras (id, carrera, fecha_alta, vigente) VALUES (15, 'Comunicación social', 1990-01-01, true);

```

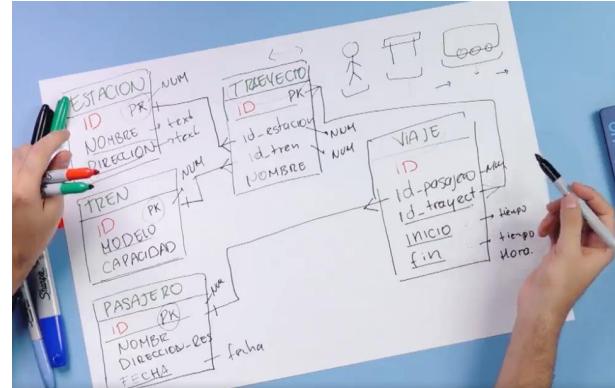
A través de estos mismos pasos, se deberán crear las demás **tablas** de la **base de datos** indicadas en el ejercicio.

Query    Query History

```
1 SELECT * FROM public.pasajero
2 ORDER BY id_pasajero ASC
```

Data Output    Messages    Notifications

	id_pasajero [PK] integer	nombre character varying(100)	direccion_residencia character varying	fecha_nacimiento date
1	1	Primer pasajero Juanito	Dirección x	1995-04-06
2	2	Primer pasajero Juanito	Dirección x	1995-04-06
3	4	Primer pasajero Juanito	Dirección x	1995-04-06



- **Pasajeros:** **Id\_Pasajero (Primary Key)**, **Nombre**, **Dirección de Residencia**, **Fecha de Nacimiento**.
- **Estaciones:** **Id\_Estación (Primary Key)**, **Nombre de Estación**, **Dirección de la Estación**.
- **Trenes:** **Id\_Tren (Primary Key)**, **Modelo del Tren**, **Capacidad de Pasajeros**.
  - **Trayectos:** **Id\_Trayecto (Primary Key)**, **Id\_Estación**, **Id\_Tren**, **Nombre de Ruta**.
    - **Viajes:** **Id\_Viaje (Primary Key)**, **Id\_Pasajero**, **Id\_Trayecto**, **Tiempo de Inicio del Viaje**, **Tiempo Final del Viaje**.

**Create - Table**

General    Columns    Advanced    Constraints    Partitions    Parameters    Security    SQL

```
1 CREATE TABLE public.pasajero
2 (
3     id_pasajero serial,
4     nombre character varying(100),
5     direccion_residencia character varying,
6     fecha_nacimiento date,
7     CONSTRAINT pasajero_pkey PRIMARY KEY (id_pasajero)
8 );
9
10 ALTER TABLE IF EXISTS public.pasajero
11     OWNER to postgres;
```

**Create - Table**

General    Columns    Advanced    Constraints    Partitions    Parameters    Security    SQL

```
1 CREATE TABLE public.estacion
2 (
3     id_estacion serial,
4     nombre_estacion character varying,
5     direccion_estacion character varying,
6     CONSTRAINT estacion_pkey PRIMARY KEY (id_estacion)
7 );
8
9 ALTER TABLE IF EXISTS public.estacion
10     OWNER to postgres;
```

**Create - Table**

General    Columns    Advanced    Constraints    Partitions    Parameters    Security    SQL

```
1 CREATE TABLE public.tren
2 (
3     id_tren serial,
4     modelo_tren character varying,
5     capacidad_pasajeros integer,
6     CONSTRAINT tren_pkey PRIMARY KEY (id_tren)
7 );
8
9 ALTER TABLE IF EXISTS public.tren
10     OWNER to postgres;
```

**Create - Table**

General    Columns    Advanced    Constraints    Partitions    Parameters    Security    SQL

```
1 CREATE TABLE public.trayecto
2 (
3     id_trayecto serial,
4     id_estacion integer,
5     id_tren integer,
6     nombre_ruta character varying,
7     CONSTRAINT trayecto_pkey PRIMARY KEY (id_trayecto)
8 );
9
10 ALTER TABLE IF EXISTS public.trayecto
11     OWNER to postgres;
```

**Create - Table**

General    Columns    Advanced    Constraints    Partitions    Parameters    Security    SQL

```
1 CREATE TABLE public.viaje
2 (
3     id_viaje serial,
4     id_pasajero integer,
5     id_trayecto integer,
6     tiempo_inicio time with time zone,
7     tiempo_fin time with time zone,
8     CONSTRAINT viaje_pkey PRIMARY KEY (id_viaje)
9 );
10
11 ALTER TABLE IF EXISTS public.viaje
12     OWNER to postgres;
```

### *Crear una Vista con SQL*

Como los datos de un **database** se irán actualizando automáticamente y una misma consulta (**Query o extracción específica de datos**) se deberá ejecutar varias veces, siendo esta la misma, pero obteniendo resultados con datos diferentes. Se puede crear una **VISTA**, para no ejecutar todo el código de un Query varias veces, sino guardándolo en un tipo de **variable** que se puede llamar cuando se necesite.

Cuando se quiera crear una nueva **VISTA** en una **base de datos**, la cual representa una **extracción de datos que se encuentren filtrados y organizados**, se deberá ejecutar el siguiente comando:

```
USE      DATABASE      "Nombre_Base_de_Datos";  
CREATE OR REPLACE  VIEW    "v_Nombre_Vista" AS  
--Query SQL aplicado a la tabla de la que se quieran obtener los datos de la VISTA.
```

**ALTER:**

### *Alterar una Tabla con SQL*

Cuando se quiera modificar la **tabla** de una **base de datos** se deberá ejecutar el siguiente comando, ya sea en un Manejador de Bases de Datos Relacionales (RDBMS) o en una Base de Datos con Servicios Administrados (Nube o Cloud):

```
ALTER  TABLE      "Nombre_Base_de_Datos". "Nombre_Entidad_o_Tabla";  
ADD    COLUMN     "Nombre_Atributo"      Tipo_de_Dato_y_Constraints;  
CHANGE COLUMN    "Columna_a_Modificar"  Tipo_de_Dato_y_Constraints;  
DROP    COLUMN     "Columna_a_Borrar";
```

### *Llaves Principales y Foráneas (PRIMARY & FOREIGN KEYS)*

Un uso muy común del comando **ALTER** es cuando, después de haber creado las **tablas** de la **base de datos**, se deben indicar sus **llaves foráneas**, las cuales indican la **dirección de conexión entre los datos de una tabla con otra** a través de sus **columnas id**, denotadas por las **restricciones PRIMARY KEY (tabla de origen)** y **FOREIGN KEY (tabla final)**.

Podemos saber el **sentido de la relación** entre dos **tablas** al analizar su **cardinalidad**, donde se indica cuántas **instancias (filas)** de una **entidad** están relacionadas con cuántas **filas** de la otra **tabla**. Para exemplificar esto, se puede pensar en una **conexión de tablas** donde la **entidad usuario** **puede tener varios comentarios**, pero la **tabla comentario**, **no puede venir de varios usuarios**, por lo que esa **relación** tiene una **cardinalidad de 1:N**.

En términos de la **dirección de una relación** en una **cardinalidad 1:N**:

- La **llave principal (punto inicial en la dirección)** reside en la **tabla** que representa el lado de "**1**".
- Y la **llave foránea (punto final en la dirección)** se encuentra en la **entidad** del lado "**N**".

Mientras que en una **relación N:N no existe sentido de conexión**, ya que:

- Se debe crear una **tabla intermedia** que contenga las **llaves foráneas** de las dos **tablas conectadas** y las **relacione** entre sí, pero como en la **tabla media de conexión** **ninguna conexión** se declara como **llave principal**, en este caso no existe un **punto final o inicial en la dirección**.

## Constraints (Restricciones)

Constraint	Descripción
NOT NULL	Se asegura que la columna no tenga valores nulos
UNIQUE	Se asegura que cada valor en la columna no se repita
PRIMARY KEY	Es una combinación de NOT NULL y UNIQUE
FOREIGN KEY	Identifica de manera única una tupla en otra tabla
CHECK	Se asegura que el valor en la columna cumpla una condición dada
DEFAULT	Coloca un valor por defecto cuando no hay un valor especificado
INDEX	Se crea por columna para permitir búsquedas más rápidas

Además de pensar en la **dirección de conexión**, se deben considerar las acciones que se ejecutarán en la **tabla con llave foránea (entidad final)** cuando se **actualicen (On update)** o **borren (On delete)** los datos de la **tabla que tiene la llave principal (entidad inicial)**, las posibles acciones que ocurrirán son las siguientes:

- **NO ACTION**: Cuando se **actualicen o borren** los datos de la **tabla que tiene la PRIMARY KEY**, en la **entidad que posee la FOREIGN KEY** **no se verán reflejados** estos cambios.
- **RESTRICT**: Cuando se **actualicen o borren** los datos de la **tabla que tiene la PRIMARY KEY**, en la **entidad que posee la FOREIGN KEY** se **restringirán** los cambios realizados y se mostrará un mensaje de error.
- **CASCADE**: Cuando se **actualicen o borren** los datos de la **tabla que tiene la PRIMARY KEY**, en la **entidad que posee la FOREIGN KEY** se **aplicarán** los cambios realizados.
- **SET NULL**: Cuando se **actualicen o borren** los datos de la **tabla que tiene la PRIMARY KEY**, en la **entidad que posee la FOREIGN KEY** se **asignará siempre el valor Null**.
- **SET DEFAULT**: Cuando se **actualicen o borren** los datos de la **tabla que tiene la PRIMARY KEY**, en la **entidad que posee la FOREIGN KEY** se **asignará el valor indicado por defecto durante la creación de la columna**.

*Nota: Casi siempre se asigna la acción CASCADE.*

Para agregar las llaves foráneas de una **entidad o tabla** perteneciente a una **base de datos** de **PostgreSQL** se debe seleccionar la opción de: **Servers** → **Motor de PostgreSQL** → **Databases** → **nombre\_base\_de\_datos** → **Schemas** → **public** → **Tables** → **nombre\_tabla** → Clic derecho → **Properties...** → **Pestaña Constraints** → **Pestaña Foreign Key**: Antes de esto ya se debe haber analizado la **cardinalidad de la conexión** para saber cuál columna es la **PRIMARY** y la **FOREIGN KEY** en la **tabla** → **+ (Add row)** → **Name**: El nombre debe ser **tablaOrigen\_tablaDestino\_fkey** → **✓ (Edit row)** → **Pestaña Columns** → **Local column**: **Columna de esta tabla**, **References**: **Tabla** que contiene la **columna** de la **relación**, **Referencing**: **Columna de la tabla de la relación** → **Add** → **Pestaña Action** → **On update**: Acción a ejecutar cuando los datos se **actualicen**, **On delete**: Acción a ejecutar cuando los datos se **borren**, usualmente se asigna el valor **CASCADE** → Esto se repite para cada **FOREIGN KEY** → **SQL**.

Object Explorer

General Columns Advanced Constraints Parameters Security SQL

Primary Key Foreign Key Check Unique Exclude

Name	Columns	Referenced Table
trayecto_estacion_fkey		

**Columns**

Local column: id\_estacion

References: public.estacion

Referencing: id\_estacion

**General** **Definition** **Columns** **Action**

Please specify columns for Foreign key.

**trayecto**

General Columns Advanced Constraints Parameters Security SQL

Primary Key Foreign Key Check Unique Exclude

Name	Columns	Referenced Table
trayecto_estacion_fkey	(id_estacion) -> (id_estacion)	public.estacion
trayecto_tren_fkey	(id_tren) -> (id_tren)	public.tren

**trayecto**

General Columns Advanced Constraints Parameters Security SQL

Primary Key Foreign Key Check Unique Exclude

Name	Columns	Referenced Table
trayecto_estacion_fkey	(id_estacion) -> (id_estacion)	public.estacion
trayecto_tren_fkey	(id_tren) -> (id_tren)	public.tren

Esto se repetirá para crear las demás **relaciones** entre las **tablas** indicadas en el ejercicio de la DB.

trayecto

General Columns Advanced Constraints Parameters Security SQL

```

1 ALTER TABLE IF EXISTS public.trayecto
2 ADD CONSTRAINT trayecto_estacion_fkey FOREIGN KEY (id_estacion)
3 REFERENCES public.estacion (id_estacion) MATCH SIMPLE
4 ON UPDATE CASCADE
5 ON DELETE CASCADE
6 NOT VALID;
7
8 ALTER TABLE IF EXISTS public.trayecto
9 ADD CONSTRAINT trayecto_tren_fkey FOREIGN KEY (id_tren)
10 REFERENCES public.tren (id_tren) MATCH SIMPLE
11 ON UPDATE CASCADE
12 ON DELETE CASCADE
13 NOT VALID;

```

**traj**

General Columns Advanced Constraints Parameters Security SQL

```

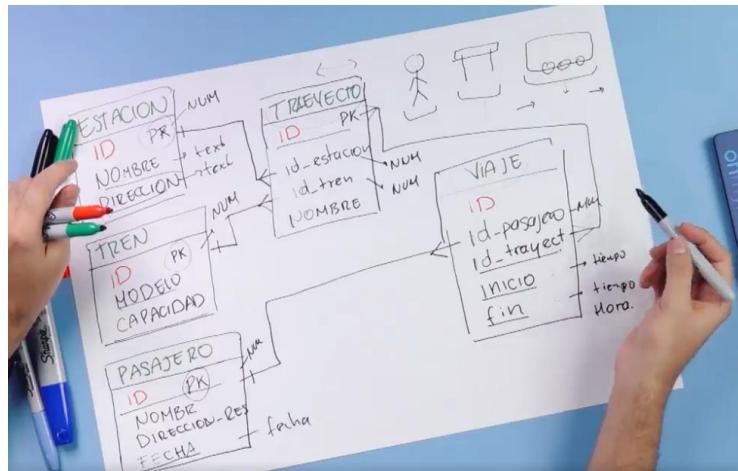
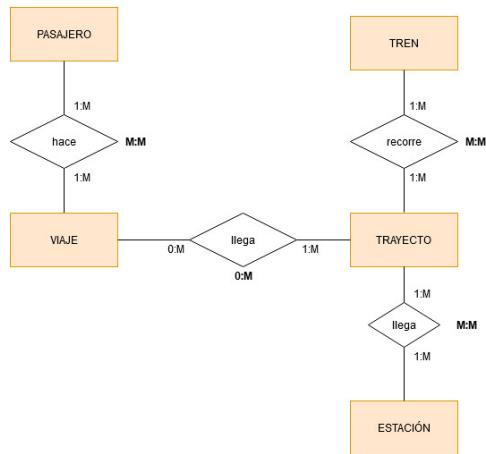
1 ALTER TABLE IF EXISTS public.viaje
2 ADD CONSTRAINT viaje_pasajero_fkey FOREIGN KEY (id_pasajero)
3 REFERENCES public.pasajero (id_pasajero) MATCH SIMPLE
4 ON UPDATE CASCADE
5 ON DELETE CASCADE
6 NOT VALID;
7
8 ALTER TABLE IF EXISTS public.viaje
9 ADD CONSTRAINT viaje_trayecto_fkey FOREIGN KEY (id_trayecto)
10 REFERENCES public.trayecto (id_trayecto) MATCH SIMPLE
11 ON UPDATE CASCADE
12 ON DELETE CASCADE
13 NOT VALID;

```

**traj**

General Columns Advanced Constraints Parameters Security SQL

El código SQL completo de las **tablas** se puede observar al seleccionar su nombre y dar clic en la pestaña de SQL y para saber dónde se deben poner las conexiones se debe observar los **diagramas ER** y **físico**.



```

-- Table: public.estacion
1 -- Table: public.estacion
2
3 -- DROP TABLE IF EXISTS public.estacion;
4
5 CREATE TABLE IF NOT EXISTS public.estacion
6 (
7     id_estacion integer NOT NULL DEFAULT nextval('estacion_id_estacion_seq'::regclass),
8     nombre_estacion character varying COLLATE pg_catalog."default",
9     direccion_estacion character varying COLLATE pg_catalog."default",
10    CONSTRAINT estacion_pkey PRIMARY KEY (id_estacion)
11 )
12
13 TABLESPACE pg_default;
14
15 ALTER TABLE IF EXISTS public.estacion
16     OWNER TO postgres;
17
18 REVOKE ALL ON TABLE public.estacion FROM di_cero_consulta;
19
20 GRANT UPDATE, INSERT, SELECT ON TABLE public.estacion TO di_cero_consulta;
21
22 GRANT ALL ON TABLE public.estacion TO postgres;

```

```

-- Table: public.viaje
1 -- Table: public.viaje
2
3 -- DROP TABLE IF EXISTS public.viaje;
4
5 CREATE TABLE IF NOT EXISTS public.viaje
6 (
7     id_viaje integer NOT NULL DEFAULT nextval('viaje_id_viaje_seq'::regclass),
8     id_trayecto integer,
9     id_pasajero integer,
10    tiempo_inicio time with time zone,
11    tiempo_fin time with time zone,
12    CONSTRAINT viaje_pk PRIMARY KEY (id_viaje),
13    CONSTRAINT viaje_pkkey FOREIGN KEY (id_viaje)
14        REFERENCES public.viaje_pkkey (id_viaje) MATCH SIMPLE
15        ON UPDATE CASCADE
16        ON DELETE CASCADE
17    NOT VALID,
18    CONSTRAINT viaje_trayecto_fkkey FOREIGN KEY (id_trayecto)
19        REFERENCES public.trayecto (id_trayecto)
20        MATCH SIMPLE
21        ON UPDATE CASCADE
22        NOT VALID
23 )
24
25 TABLESPACE pg_default;
26
27 ALTER TABLE IF EXISTS public.viaje
28     OWNER TO postgres;
29
30 REVOKE ALL ON TABLE public.viaje FROM di_cero_consulta;
31
32 GRANT UPDATE, INSERT, SELECT ON TABLE public.viaje TO di_cero_consulta;
33
34 GRANT ALL ON TABLE public.viaje TO postgres;

```

DROP:

### Borrar una Tabla, Columna o Base de Datos con SQL

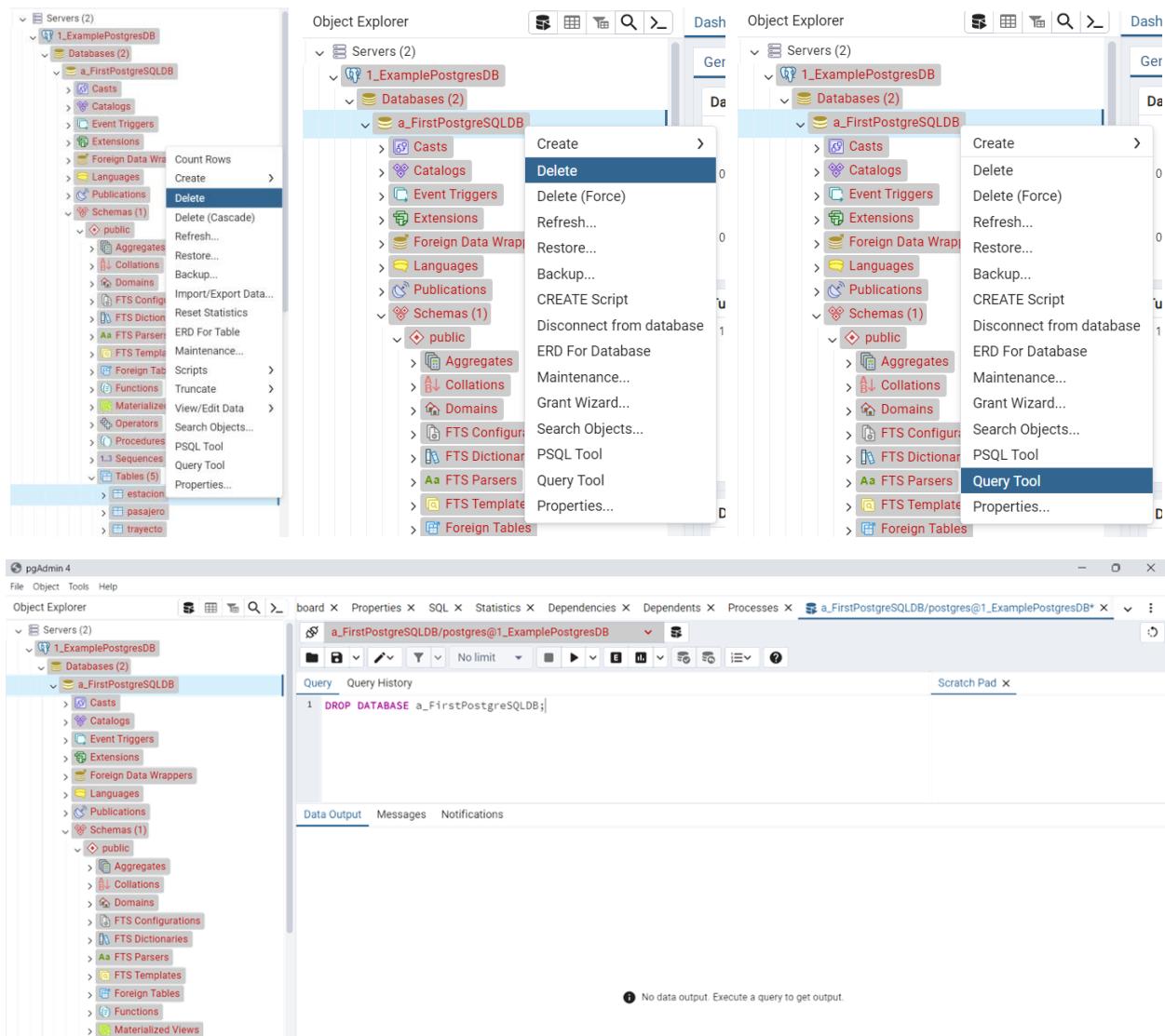
Cuando se quiera modificar una tabla en una base de datos se deberá ejecutar el siguiente comando, ya sea en un Manejador de Bases de Datos Relacionales (RDBMS) o en una Base de Datos con Servicios Administrados (Nube o Cloud):

```

DROP DATABASE Nombre_Base_de_Datos;
DROP TABLE Nombre_Entidad_o_Tabla;
DROP COLUMN Columna_a_Borrar;

```

La misma secuencia aplicada con el fin de **alterar tablas**, se aplica para borrar **columnas, tablas o db**.



## Sub-lenguajes de SQL: DML (Data Manipulation Language)

Los dos sub-lenguajes más importantes del lenguaje SQL son llamados **DDL** y **DML**, ya se abordó el lenguaje **DDL** que sirve para estructurar la base de datos y ahora se explicará el **DML** que **sirve para manipular los datos dentro de ella**:

- **DML (Data Manipulation Language):** La función principal de este sub-lenguaje de SQL es la de manipular los datos de una **base de datos**, ya sea para crear **vistas** o **para meter, actualizar, borrar o extraer datos**. Los **4 comandos** con los que se cuenta para llevar a cabo la manipulación de datos con el lenguaje **DML** son los siguientes:
  - **INSERT:** Comando que permite introducir **nuevas filas de datos** (también llamados **registros** o **tuplas**) a una **tabla (entidad)** de una **base de datos**.
  - **UPDATE:** Comando que actualiza o modifica datos ya existentes en una **tabla** perteneciente a una **base de datos**.

- **DELETE**: Instrucción que sirve para borrar toda la **fila** de datos de una **tabla** perteneciente a una **base de datos**.
- **SELECT**: Este comando es muy importante, ya que permite traer información de la **base de datos**, ya sea para crear una **vista**, extraerla para ponerla en otro lado, etc.

Con los comandos descritos se pueden realizar todas las acciones del **CRUD** (acrónimo de **Create, Read, Update, Delete**), que es un conjunto de operaciones básicas realizadas en cualquier sistema de gestión de **bases de datos**.

## INSERT:

### *Insertar Datos Nuevos a la Tabla de una Base de Datos con SQL*

Cuando se quiera añadir nuevos datos a la fila de la tabla de una base de datos se ejecutará el comando **INSERT** de la siguiente manera:

```
INSERT INTO Nombre_Entidad_o_Tabla(
    Nombre_Atributo_o_Columna_1,
    Nombre_Atributo_o_Columna_2,
    Nombre_Atributo_o_Columna_n
)
VALUES(
    "Valor_Atributo_o_Columna_1",
    "Valor_Atributo_o_Columna_2",
    "Valor_Atributo_o_Columna_n"
);
;
```

Para insertar datos dentro de una **tabla**, seleccionaremos la opción de: **Tables** → **nombreTabla** → Clic derecho → Scripts → INSERT Script.

- Aquí hay que tomar en cuenta que si el **tipo de dato** del id es **serial**, **no se debe insertar ningún valor manualmente en él**, este se asigna por sí solo.
- Para los datos tipo date se puede utilizar el comando **SELECT current\_date**; para saber el formato de **fecha** y solo ejecutar esta parte del código al seleccionarla y dar clic sobre el botón de **Execute script**, donde podremos ver que **el formato del tipo de dato date es año-mes-día**.



The screenshot shows the pgAdmin 4 interface. In the Object Explorer, under the 'Tables (1)' section, the 'pasajero' table is selected. A context menu is open over the table, with the 'INSERT Script' option highlighted. Below the table, a preview window shows the current state of the 'pasajero' table with one row: id\_pasajero: 1, nombre: 'Diego Cervantes Rodriguez', direccion\_residencia: 'Vive en tu corazón', fecha\_nacimiento: '1995-04-06'. The main query editor window contains the following SQL script:

```

1 INSERT INTO public.pasajero(
2     nombre, direccion_residencia, fecha_nacimiento)
3     VALUES (?, ?, ?);
4
5 SELECT current_date;

```

The Data Output tab shows the result of the last query: current\_date: 2024-05-15.

Si se quiere observar el contenido de la tabla se debe seleccionar la opción de **Tables → nombreTabla** → Clic derecho → View/Edit Data → All rows:

Y veremos la **entidad** con los **atributos** que le agregamos en forma de tabla.

The screenshot shows the pgAdmin 4 interface. In the Object Explorer, under the 'Tables (1)' section, the 'pasajero' table is selected. A context menu is open over the table, with the 'View/Edit Data' option highlighted. Below the table, a preview window shows the current state of the 'pasajero' table with two rows:

	id_pasajero [Pk] integer	nombre character varying (100)	direccion_residencia character varying	fecha_nacimiento date
1	1	Diego Cervantes Rodriguez	Vive en tu corazón	1995-04-06

Al asignar valores a las **tablas**, se toman en cuenta las **relaciones** que existen entre ellas. Por ejemplo, si se quiere asignar un valor a la **tabla trayecto**, la cual **conecta los id de la tabla estación con la tabla tren**, se deben haber **creado valores previamente** en dichas **tablas** y utilizar **ids válidos**, ya que, si se quiere **insertar un valor** en la **tabla trayecto** de un **id que no existe**, se lanzará una excepción.

Data Output				Data Output			
	<b>id_estacion</b> [PK] integer	<b>nombre_estacion</b> character varying	<b>direccion_estacion</b> character varying		<b>id_tren</b> [PK] integer	<b>modelo_tren</b> character varying	<b>capacidad_pasajeros</b> integer
1	1	Estación centro	St 1 #12	1	1	Tren 8000	100
2	2	Estación norte	Calle 100 #12				

## UPDATE:

### Editar Datos en la Tabla de una Base de Datos con SQL

Cuando se busque editar los datos de la tabla de una base de datos se deberá indicar exactamente a que posición o posiciones de la tabla nos estamos refiriendo, ya que el cambio se puede realizar en una o varias filas de la **entidad**. Para ello se utiliza el comando **SET** seguido del nombre del **atributo** que se quiere modificar y el nuevo **valor** que adoptará, luego se señala la **fila** por medio de la instrucción **WHERE** (también llamada **Operador Unario de Selección o o**) acompañada de algún valor que la identifique, para así indicar si se busca modificar solo una fila en específico o todas las filas donde se cumpla esta condición:

```

UPDATE "Nombre_Entidad_o_Tabla"
SET      "Columna_1" = "Valor_Columna_1", "Atributo_2" = "Valor_Atributo_2"
WHERE  "Nombre_Atributo_o_Columna" = "Valor_Fila_o_Identificador";

```

Cabe mencionar que, si no se indica nada en la instrucción **WHERE**, todos los datos de la **columna** indicada serán editados al nuevo valor indicado. Pero al intentar esto es cuando se vuelve evidente el uso de un manejador de **base de datos**, ya que cuando se intente realizar una edición masiva de datos al dar clic en el rayo que se encuentra en la parte superior, esto será evitado por el programa, evitando así que se realicen cambios indeseables.

```

1 UPDATE public.estacion
2   SET nombre_estacion='Estación central', direccion_estacion='Calle 13 #13'
3   WHERE id_estacion=1;
4
5 SELECT * FROM public.estacion
6 ORDER BY id_estacion ASC;

```

	id_estacion [PK] integer	nombre_estacion character varying	direccion_estacion character varying
1	1	Estación central	Calle 13 #13
2	2	Estación norte	Calle 100 #12

## Funciones Especiales de Postgres para los Comandos INSERT Y UPDATE

Estas son las funciones estándar de SQL para insertar o actualizar datos, pero se cuenta con algunas propias del motor **PostgreSQL**, las cuales realizan las siguientes funciones especiales:

- **ON CONFLICT DO**: El comando se utiliza junto la instrucción **INSERT** para el **manejo de excepciones** que ocurren cuando una inserción violaría la restricción indicada por una **PRIMARY** o **FOREIGN KEY**, pudiendo especificar que hacer en caso de conflicto, ya sea actualizar los registros existentes (**DO UPDATE**) o no hacer nada (**DO NOTHING**).

```

INSERT INTO Nombre_Entidad_o_Tabla(
    Nombre_Atributo_o_Columna_1,
    Nombre_Atributo_o_Columna_2,
    Nombre_Atributo_o_Columna_n
)
VALUES(
    "Valor_Atributo_o_Columna_1",
    "Valor_Atributo_o_Columna_2",
    "Valor_Atributo_o_Columna_n"
)
ON CONFLICT (Nombre_Atributo_Conflicto) DO NOTHING/UPDATE...;

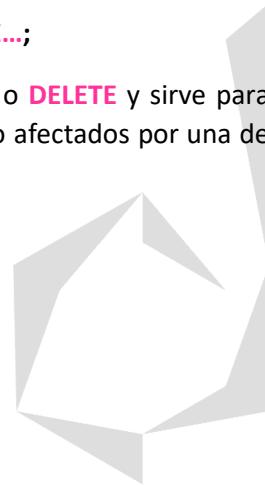
```

- **RETURNING**: El comando se utiliza junto las funciones **INSERT**, **UPDATE** o **DELETE** y sirve para devolver los valores de ciertos **atributos** después de que éstos hayan sido afectados por una de las **operaciones de modificación** antes mencionadas.

```

INSERT INTO Nombre_Entidad_o_Tabla(
    Nombre_Atributo_o_Columna_1,
    Nombre_Atributo_o_Columna_2,
    Nombre_Atributo_o_Columna_n
)

```



```

        Nombre_Atributo_o_Columna_n
    )
VALUES(
    "Valor_Atributo_o_Columna_1",
    "Valor_Atributo_o_Columna_2",
    "Valor_Atributo_o_Columna_n"
)
RETURNING Nombre_Atributo_Modificado_1, ..., Nombre_Atributo_Modificado_n;

```

**DELETE:**

### Borrar Todos los Datos de una Fila Perteneciente a la Tabla de una Base de Datos con SQL

Cuando se busque borrar datos de una **tabla**, esto se hará en toda la **fila** de ese **registro o tupla**. Para ello se utiliza el comando **WHERE** acompañado de algún valor que identifique la fila o filas que se busca eliminar:

```

DELETE FROM "Nombre_Entidad_o_Tabla"
WHERE "Nombre_Atributo_o_Columna" = "Valor_Fila_o_Identificador";

```

Cabe mencionar que, si no se indica nada en la instrucción **WHERE**, todos los datos de la **tabla** indicada serán eliminados. Por lo que hay que tener mucho cuidado cuando se utilice este comando, ya que podríamos borrar accidentalmente todos los datos de nuestra **base de datos**.

	id_trayecto [PK] integer	id_estacion integer	id_tren integer	nombre_ruta character varying
1	1	1	1	Ruta 1
2	2	2	1	Ruta 2
3	3	1	1	Ruta 3

Query    Query History

```

1 DELETE FROM public.trayecto
2   WHERE id_trayecto=3;
3
4 SELECT * FROM public.trayecto
5 ORDER BY id_trayecto ASC;

```

Data Output    Messages    Notifications

	id_trayecto [PK] integer	id_estacion integer	id_tren integer	nombre_ruta character varying
1		1	1	Ruta 1
2		2	2	Ruta 2

**SELECT:**

### Extraer Todos los Datos de una Columna Perteneciente a la Tabla de una Base de Datos

Cuando se busque obtener todos los datos pertenecientes a la **columna (atributo)** de una **tabla**, ya sea para crear una **vista** o para simplemente moverlos a otro lado, se utiliza el comando **SELECT** (también llamado **Operador Unario de Proyección o  $\pi$** ), donde se debe indicar el nombre del **atributo** o **columna** de datos que se quiere extraer y a qué **tabla** o **entidad** pertenecen:

```
SELECT Nombre_Atributo_o_Columna_1, Nombre_Atributo_o_Columna_n
```

```
FROM Nombre_Entidad_o_Tabla;
```

Cabe mencionar que, si no se indica un **atributo** en específico y en su lugar se utiliza un asterisco, se extraerán todas las **columnas** de la **tabla** indicada. Para ejecutar el código SQL en la **base de datos PostgreSQL** se debe dar clic dentro de pgAdmin en la opción de: **Servers** → **Motor PostgreSQL** → **Nombre\_Database** → **Schemas** → **public** → **Tables** → **nombreTabla** → Clic derecho → **Scripts** → **SELECT Script** → **Execute script**.

The screenshot shows the pgAdmin interface with the following details:

- Servers:** 1.ExamplePostgresDB
- Databases:** a.FirstPostgreSQL
- Tables:** estacion (selected)
- Script Execution:** A context menu is open over the table, with the "SELECT Script" option highlighted.
- Query History:** Shows the query: `SELECT id_estacion, nombre_estacion FROM public.estacion;`
- Data Output:** Shows the results of the query:
 

	id_estacion [PK] integer	nombre_estacion character varying
1	2	Estación norte
2	1	Estación central

### Consultas o Queries: Extracción de información de una base de datos

Las **consultas o Queries** son una parte fundamental de las **bases de datos**, ya que de esta forma es como se pueden **extraer datos para realizar un análisis**, responder una pregunta o simplemente utilizar la

información almacenada. Algunas aplicaciones de ello son: Business intelligence, Machine learning, Data science, etc.

La estructura de un Query se conforma de los comandos **SELECT**, **FROM** y opcionalmente **WHERE** para indicar la posición y el elemento de donde se busca obtener cierta información.

- La **tabla (entidad)** de la cual se busca extraer los datos se indica con el comando **FROM**.
- La **columna (atributo)** se indica con el comando **SELECT** (también llamado **Operador Unario de Proyección o  $\pi$** ).
- La **fila** se señala con el comando **WHERE** (también llamado **Operador Unario de Selección o  $\sigma$** ), para ello en el código no se especifican directamente las **filas**, sino las condiciones que deben cumplir las **columnas** para obtener ciertas **instancias**.
  - En las consultas simples el orden en el que se utilizan los comandos es, primero **SELECT** junto con el nombre del **atributo** que se quiere extraer y luego **FROM** indicando la **tabla** a la que pertenecen. Si después de la instrucción **SELECT** se utiliza un asterisco \* en vez del nombre de una **columna**, es porque se busca extraer todos los datos de dicha **entidad**.
    - **AS**: Es una instrucción adicional que se puede utilizar en conjunto con el comando **SELECT**, **FROM** o **JOIN**, el cual sirve para cambiar el nombre de la **columna de datos** extraída y **asignarle un alias o nombre de variable**, cambiando solo la forma en la que se representan los datos extraídos, no su nombre en la **base de datos**.
    - **COUNT()**: Método que cuando se utiliza, siempre se debe poner después del método **SELECT**; este recibe como parámetro un **atributo** de los datos pertenecientes a la **tabla** y retorna el número de **filas** de datos que pertenecen a dicha **columna**.
    - **SUM()**: Método para sumar todos los valores numéricos de una **columna**.
    - **AVG()**: Función para obtener el promedio de los valores numéricos de una **columna**.
    - **MIN()**: Método para obtener el **valor mínimo** encontrado en las **filas** de una **columna**.
    - **MAX()**: Encuentra el **valor máximo** en las **filas** de una **columna**.
    - **GROUP\_CONCAT()**: Es una función que sirve para obtener las **filas** de una consulta y retornar sus valores en forma de tupla (separados por comas). Esta se aplica cuando se busca **agrupar varios valores** en función de un **atributo** en específico.
    - **IF/ CASE**: Los condicionales en SQL se crean con las instrucciones **IF** o **CASE** cuando se analice más de una condición.

```
SELECT      *      IF  (Condición,  Valor_si_True,  Valor_si_False);  
  
SELECT  Atributo_1, ..., Atributo_n, CASE  
        WHEN  Condición_1  THEN  Valor_si_True  
        WHEN  Condición_2  THEN  Valor_si_True  
        ...  
        WHEN  Condición_n  THEN  Valor_si_True  
END  AS  Nuevo_Alias_Resultado;
```

- **JOIN**: Se había mencionado previamente que a través de la sentencia **FROM** se indica de qué **tabla** se extraerán los datos, aunque solo se estableció el caso donde esto se

realizaba para una sola **entidad**, pero cuando se quiera extraer **filas** de datos de **dos o más tablas distintas**, se añade la instrucción **JOIN**. Es muy importante mencionar que esto solo se podrá realizar en aquellas **entidades** que se encuentren enlazadas a través de una **relación**, osea cuando una contenga una **PRIMARY KEY** y la otra posea una **FOREIGN KEY** (**o las dos posean FOREIGN KEYS si tienen cardinalidad N:N**).

- Se puede representar de forma gráfica el funcionamiento de una instrucción **JOIN** a través de los **operadores binarios** (**unión, diferencia, multiplicación, etc.**) usualmente utilizados en un **Diagrama de Venn**.
  - Los pasos para **relacionar** los datos de ambas **tablas** son los siguientes:
    - Primero se indica a través del método **FROM** la primera **entidad** que de la cual se quieren extraer datos, la de **cardinalidad 1**, aunque si se tiene una **cardinalidad N:N**, se puede elegir cualquiera de las **tablas** (esta adoptará la **posición izquierda en el Diagrama de Venn**).
    - Luego a través de alguna variante de **Diferencia, Intersección, Unión o Diferencia Simétrica** del método **JOIN** se denota la **entidad** con **cardinalidad de N** (**que tomará la posición derecha**).
    - Finalmente, ambas se **conectan** a través de la instrucción **ON** que se acompaña tanto del **atributo** que representa el **PRIMARY\_KEY** en la **tabla izquierda** como del **atributo** que represente el **FOREIGN\_KEY** de la **entidad derecha** y ambos se igualan.
      - **JOIN = INNER JOIN:** Intersección:  $A \cap B$ .

--Intersección entre 2 tablas diferentes =  $A \cap B$

```
SELECT * FROM Nombre_Tabla_Izq
JOIN Nombre_Tabla_Der ON Tabla_Izq.PRIMARY_KEY = Tabla_Der.FOREIGN_KEY;
      • FULL OUTER JOIN: Unión:  $A \cup B$ .
```

--Intersección entre 2 tablas diferentes =  $A \cup B$

```
SELECT Nombre_Columna FROM Nombre_Tabla_Izq
FULL OUTER JOIN Nombre_Tabla_Der ON Tabla_Izq.PRIMARY_KEY = Tabla_Der.FOREIGN_KEY;
      • LEFT OUTER JOIN: Diferencia e intersección izquierda:  $A - B + (A \cap B)$ .
```

-- Diferencia e intersección izquierda entre 2 tablas diferentes =  $A - B + (A \cap B)$

```
SELECT * FROM Nombre_Tabla_Izq
LEFT JOIN Nombre_Tabla_Der ON Tabla_Izq.PRIMARY_KEY = Tabla_Der.FOREIGN_KEY;
```

- **LEFT JOIN:** Diferencia izquierda:  $A - B$ .

-- Diferencia izquierda entre 2 tablas diferentes =  $A - B$

```
SELECT Nombre_Atributo FROM Nombre_Tabla_Izq
LEFT JOIN Nombre_Tabla_Der ON Tabla_Izq.PRIMARY_KEY = Tabla_Der.FOREIGN_KEY
WHERE Tabla_Der.FOREIGN_KEY IS NULL;
```

- **RIGHT OUTER JOIN:** Diferencia e intersección derecha:  $B - A + (A \cap B)$ .

-- Diferencia e intersección derecha entre 2 tablas diferentes =  $B - A + (A \cap B)$

```
SELECT * FROM Nombre_Tabla_Izq
RIGHT JOIN Nombre_Tabla_Der ON Tabla_Izq.PRIMARY_KEY = Tabla_Der.FOREIGN_KEY;
• RIGHT JOIN: Diferencia derecha:  $B - A$ .
```

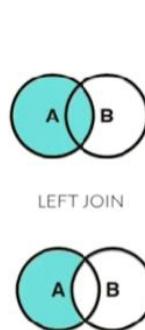
-- Diferencia derecha entre 2 tablas diferentes =  $B - A$

```
SELECT Nombre_Columna FROM Nombre_Tabla_Izq
RIGHT JOIN Nombre_Tabla_Der ON Tabla_Izq.PRIMARY_KEY = Tabla_Der.FOREIGN_KEY
WHERE Tabla_Izq.PRIMARY_KEY IS NULL;
• OUTER JOIN: Diferencia simétrica:  $A \cup B - (A \cap B)$ .
```

-- Diferencia simétrica entre 2 tablas diferentes =  $A \cup B - (A \cap B)$ .

```
SELECT * FROM Nombre_Tabla_Izq
FULL OUTER JOIN Nombre_Tabla_Der ON Tabla_Izq.PRIMARY_KEY = Tabla_Der.FOREIGN_KEY
WHERE Tabla_Izq.PRIMARY_KEY IS NULL OR Tabla_Der.FOREIGN_KEY IS NULL;
```

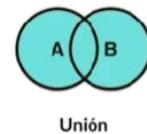
## JOIN



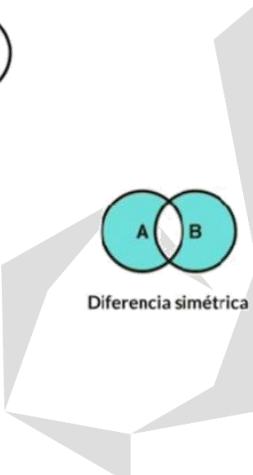
Diferencia



## JOIN



OUTER JOIN





**LEFT JOIN**

**RIGHT JOIN**

**INNER JOIN**

**FULL OUTER JOIN**

- **WHERE:** De forma opcional se podrá indicar exactamente a cuáles **filas** de la **tabla** nos estamos refiriendo, filtrándola a través de cierta **condición matemática** ( $=$ ,  $>$ ,  $<$ , etc.), ya que la extracción se puede realizar en una o varias **filas**, para ello se utiliza el comando **WHERE** acompañado del **valor** de algún **atributo**.
  - **AND:** Si se quiere agregar más de un filtro en una búsqueda, lo que se hace es agregar después del primer filtro la sentencia **AND** y con eso se podrán sumar filtros adicionales.
  - **OR:** Dentro de la instrucción **WHERE**, se puede utilizar la compuerta **OR** para analizar más de una opción como filtro, aunque el resultado dependerá de si se usan paréntesis en la operación lógica o no.
    - En el siguiente primer ejemplo, no se utilizará un paréntesis para indicar que el operador **AND** se aplique tanto a las **condiciones adicionales 1 y 2**, por lo que el **Query** traerá todos los elementos que tengan un **name = "Diego"** y **last\_name = "Cervantes"**, pero también traerá todos los elementos que tengan un **last\_name = "Rodríguez"**, no importando si el **name = "Diego"**, lo cual es incorrecto.

```
SELECT * FROM Nombre_Tabla
```

```
WHERE name = "Diego"
```

```
AND last_name = "Cervantes"
```

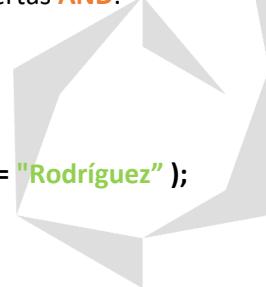
```
OR last_name = "Rodríguez";
```

- En el segundo ejemplo, si se utiliza un paréntesis para indicar que el operador **AND** se aplique tanto a las **condiciones adicionales 1 y 2**, la **Consulta** traerá todos los datos que tengan **name = "Diego"**, pero que tengan además **last\_name = "Cervantes"** o **last\_name = "Rodríguez"**, logrando añadir así una condición adicional doble, que use correctamente la compuerta **OR**. Esto igual se puede aplicar con compuertas **AND**.

```
SELECT * FROM Nombre_Tabla
```

```
WHERE name = "Diego"
```

```
AND (last_name = "Cervantes" OR last_name = "Rodríguez");
```



- **BETWEEN**: Si se quiere filtrar el resultado a través de un **rango de valores**, se utiliza una combinación de los comandos **BETWEEN** y **AND** de la siguiente manera:

```
SELECT * FROM Nombre_Tabla
WHERE Nombre_Columna BETWEEN Límite_Min_Rango AND Límite_Max_Rango;
```

- **IN**: Cuando se quiera crear una condición de filtrado a una **columna**, **igualándola no a un valor, sino a varios**, se utiliza el comando **IN**.

```
SELECT * FROM Nombre_Tabla
WHERE Nombre_Columna IN ('Valor_de_Filtrado_1', ..., 'Valor_de_Filtrado_n');
```

- **GROUP BY**: Esta sentencia agrupa las **filas** resultantes de una consulta según **uno o más atributos** especificados. Se utiliza junto los métodos **COUNT()**, **SUM()**, **AVG()**, **MIN()**, **MAX()**, etc. para calcular ciertos valores numéricos de cada agrupación, en lugar de calcular dichos **valores** sobre **todas las filas** de una **tabla** de forma innecesaria.
  - La forma en la que se utiliza el método **GROUP BY** depende mucho de la información que contenga la **base de datos**, ya que a través de ella se podrán hacer informes agrupados por cierta clasificación indicada por **una o más columnas** de datos.
  - **HAVING**: Este comando igualmente se usa de forma opcional y lo que hace es **filtrar** a través de cierta **condición lógica** las **filas de información** extraídas de una **tabla**, de la misma forma cómo funciona el método **WHERE**, pero si hacemos pruebas con este, podremos ver que no funciona después de haber agrupado los datos obtenidos con el método **GROUP BY**, por lo que se debe reemplazar con la sentencia **HAVING** cuando se cumpla esta condición, pero realiza la misma función.
- **ORDER BY**: Comando opcional cuya función es la de ordenar una agrupación de datos para observar de mejor manera el resultado, cuando se busca que este orden se ejecute de forma **ascendente (de menos a más en el valor de cierto atributo)** se incluye la sentencia **ASC** y cuando se quiere que se ordenen de forma **descendente (de más a menos)** se añade la sentencia **DESC**.
  - **LIMIT**: El comando **ORDER BY** se puede acompañar de la instrucción **LIMIT**, la cual después de haber organizado los datos, limita el número de filas que se van a mostrar. Aunque esta sentencia se suele utilizar después del comando **ORDER BY**, se puede utilizar cuando sea.
  - **OFFSET**: Este comando se utiliza en conjunto con la instrucción **LIMIT**, ya que indica cuántos datos nos tenemos que saltar o ignorar de arriba hacia abajo, para que desde ahí se empiecen a recabar.

```
SELECT Nombre_Columna_1 AS Nuevo_Nombre_Atributo_1, COUNT(Columna_n)
FROM Nombre_Tabla_Izq
```

--Unión opcional de Diferencia, Intersección, etc. entre dos tablas diferentes.

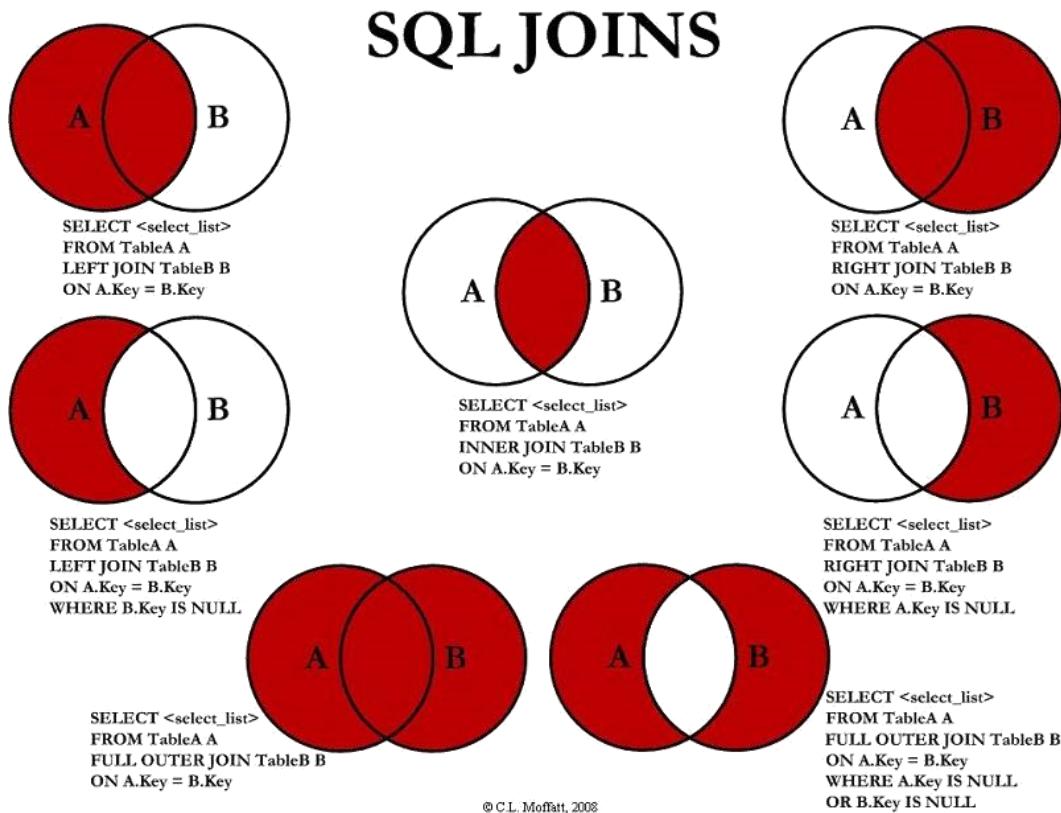
```
JOIN Entidad_Der ON Tabla_Izq.PRIMARY_KEY = Tabla_Der.FOREIGN_KEY
```

```
WHERE Nombre_Atributo_o_Columna Operación Math "Valor_Fila_Para_Filtro"
```

```

AND      Nombre_Atributo Operación Mat "Valor_Fila_Para_Filtro_Adicional"
GROUP BY  Nombre_Columna_1
HAVING  Nombre_Atributo_o_Columna Operación Mat "Valor_Fila_Para_Filtro"
ORDER BY  Nombre_Atributo_o_Columna ASC_o_DESC
LIMIT    Número_de_Filas_A_Ignorar_Antes_de_Recolectar
LIMIT    Número_de_Filas_Ordenadas_a_Mostrar;

```



### Funciones Especiales de Postgres para el Comando SELECT

Estas son las funciones estándar de SQL para consultas, pero se cuenta con algunas propias del motor **PostgreSQL**, las cuales realizan las siguientes funciones especiales:

- **LIKE/ILIKE:** El comando se utiliza junto la función **SELECT/WHERE** para buscar un patrón específico en una **columna** de tipo texto, respetando mayúsculas y minúsculas. Cuando no sepamos si algún texto se va a encontrar antes o después del string indicado, se usa el símbolo de porcentaje %, además podemos añadir un comodín que valga cualquier letra en una posición específica con un guión bajo \_ . **ILIKE** funciona de la misma forma, pero realiza una búsqueda insensible a mayúsculas y minúsculas. Si se coloca antes la compuerta **NOT**, traera lo contrario.

```

SELECT  Nombre_Atributo      FROM      Nombre_Tabla
WHERE   Nombre_Atributo      LIKE/ILIKE  %Texto_A_Buscar%;
```

- **IS/IS NOT:** El comando se utiliza junto la función **SELECT/WHERE** y se utiliza para buscar las filas con valores nulos (**NULL**) o no nulos (**NOT NULL**).

```
SELECT Nombre_Atributo FROM Nombre_Tabla  
WHERE Nombre_Atributo IS/IS NOT NULL/NOT NULL;
```

- **COALESCE:** El método se aplica individualmente a cada **atributo** de una **consulta o Query** y se usa para comparar dos valores y retornar aquel que sea no nulo (**NOT NULL**), para ello este recibe como parámetros **un atributo** y un **valor estático** para que, si la columna tiene asignado un valor nulo (**NULL**), retorne el **valor estático** en vez de devolver **NULL** y lo asigne al nuevo **Nuevo\_Nombre\_Atributo** indicado después de la instrucción **AS**.

```
SELECT COALESCE(Atributo_1, "Valor_Estático") AS Nuevo_Nombre_Atributos  
FROM Nombre_Tabla;
```

- **NULLIF:** El método se aplica individualmente a cada **atributo** de una **consulta o Query** y se usa para comprarrar los valores de dos **atributos numéricos** que reciba como parámetros y retornar un valor nulo (**NULL**) cuando estos tengan el mismo valor; de lo contrario, devuelve el valor de **Atributo\_Num\_1**.

```
SELECT NULLIF(Atributo_Num_1, Atributo_Num_2) AS Nuevo_Nombre_Atributos  
FROM Nombre_Tabla;
```

- **GREATEST:** El método se usa para comprarrar el valor de todos los **atributos numéricos** que reciba como parámetros y **retornar el valor más grande que encuentre**.

```
SELECT GREATEST (Atributo_1, ..., Atributo_n) AS Nuevo_Nombre_Atributos  
FROM Nombre_Tabla;
```

- **LEAST:** El comando sirve para comprarrar el valor de todos los **atributos numéricos** que reciba como parámetros y **retornar el valor más pequeño que encuentre**.

```
SELECT LEAST (Atributo_1, ..., Atributo_n) AS Nuevo_Nombre_Atributos  
FROM Nombre_Tabla;
```

- **BLOQUES ANÓNIMOS:** Estos nos permiten ingresar condicionales con SQL dentro de una consulta hecha a la base de datos.

```
SELECT (Atributo_1, ..., Atributo_n),  
CASE  
WHEN (Condicional con Operadores Lógicos ==, !=, >, <, etc. )  
Valor_Aisgnado_Cuando_el_Condicional_sea_True  
ELSE
```



```

    Valor_Aisgnado_Cuando_el_Condicional_sea_False
END      AS Nuevo_Nombre_Columna_Agregada_Con_Condicional
FROM    Nombre_Tabla;

```

### *Crear una Vista: Almacenar un Query en una “Variable” con SQL*

Como los datos de un **database** se irán actualizando automáticamente y una misma consulta (**Query o extracción específica de datos**) se deberá ejecutar varias veces, siendo ésta la misma, pero obteniendo resultados con datos diferentes. Se puede crear una **VISTA**, para no ejecutar todo el código de un Query varias veces, sino guardándolo en un tipo de **variable** que se puede llamar cuando se necesite.

Cuando se quiera crear una nueva **VISTA** en una **base de datos**, la cual representa una **extracción de datos que se encuentren filtrados y organizados**, se deberá ejecutar el siguiente comando:

```

CREATE OR REPLACE   VIEW   v_Nombre_Vista      AS
--Query SQL aplicado a la tabla de la que se quieran obtener los datos de la VISTA.

SELECT      (Atributo_1, ..., Atributo_n)          AS Nuevo_Nombre_Atributo
FROM        Nombre_Tabla_Izq
--Unión opcional de Diferencia, Intersección, etc. entre dos tablas diferentes.

JOIN       Entidad_Der ON Tabla_Izq.PRIMARY_KEY = Tabla_Der.FOREIGN_KEY
WHERE      Nombre_Atributo_o_Columna Operación Lógica "Valor_Fila_Para_Filtro"
AND        Nombre_Atributo Operación Lógica "Valor_Fila_Para_Filtro_Adicional"
GROUP BY   Nombre_Columna_1
HAVING     Nombre_Atributo_o_Columna Operación Lógica "Valor_Fila_Para_Filtro"
ORDER BY   Nombre_Atributo_o_Columna ASC_o_DESC
LIMIT      Número_de_Filas_Ordenadas_a_Mostrar;

```

### *Nested Queries: Consultas Anidadas (Agujero de Conejo)*

Una Query anidada se da cuando dentro de una consulta se introduce otra, esto es muy utilizado cuando dentro de alguna condición se quiere utilizar algún un **valor máximo o mínimo** perteneciente a la **columna (atributo)** de una **tabla (entidad)**, por lo que muchas veces se utiliza en conjunto con los métodos **MIN()** o **MAX()**, pero el gran problema que tiene es cuando esta búsqueda se va a realizar varias veces en una base de datos, ya que el tiempo de ejecución se incrementa exponencialmente, por esa razón es que hay que analizar detenidamente sus casos de uso para evitar así que se creen agujeros de conejo interminables. La sintaxis que se puede utilizar para ejecutar es la siguiente:

```

SELECT      Query_Anidado_1.Atributo_Anidado_1, COUNT(Columna)

```

```

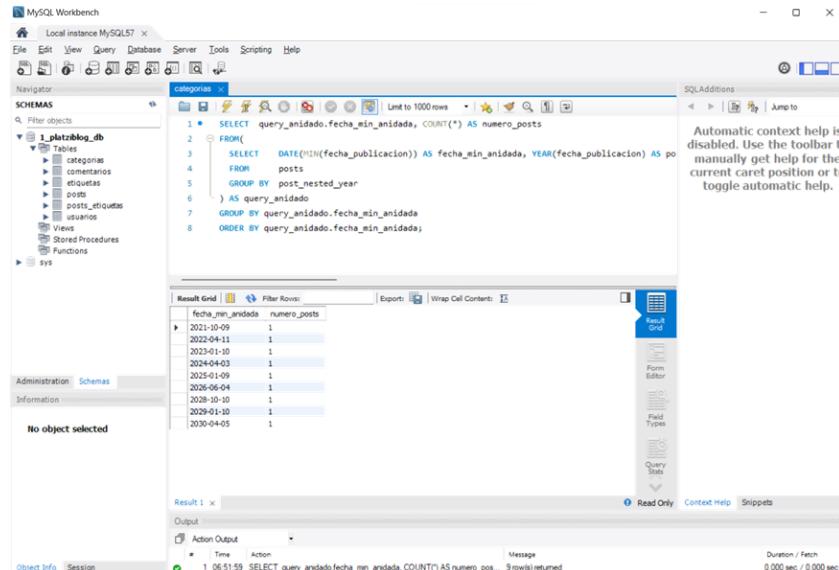
FROM      (
--Consulta (Query) anidado.

SELECT      MIN(Atributo_1) AS Atributo_Anulado_1, COUNT(Columna_n)
FROM        Nombre_Tabla_o_Entidad
...
) AS Query_Anulado_1

GROUP BY    Query_Anulado_1.Atributo_Anulado_1
HAVING     Query_Anulado_1.Columna_n Operación Lógica "Valor_Fila_Para_Filtro";
ORDER BY    Query_Anulado_1.Atributo_Anulado_1 ASC_o_DESC
LIMIT      Número_de_Filas_Ordenadas_a_Mostrar

```

Hay que tener mucho cuidado con las consultas anidadas, pero no se puede negar su utilidad, ya que permiten primero hacer un análisis de la base de datos y luego hacer un análisis posterior con dicho resultado:



The screenshot shows the MySQL Workbench interface with a query editor window open. The query is a nested SELECT statement. The outer query selects the minimum date from a subquery and counts the rows. The subquery itself is a SELECT statement that retrieves the minimum date from the 'fecha\_publicacion' column and the year from the same column. The results are grouped by the year and ordered by the minimum date. The result grid shows the minimum date and the count of posts for each year from 2021-10-09 to 2020-04-05.

fecha_min_anulado	numero_posts
2021-10-09	1
2022-04-11	1
2023-01-10	1
2024-04-03	1
2025-01-09	1
2026-04-04	1
2028-10-10	1
2029-01-10	1
2030-04-05	1

Otra aplicación de las consultas aplicadas es la siguiente, donde ahora el query interior fue hecho para obtener la condición que extrae solo cierta fila de la tabla:

```

SELECT      Nombre_Columna_1 AS Nuevo_Nombre_Atributo_1, COUNT(Columna_n)
FROM        Nombre_Tabla_o_Entidad
WHERE       Nombre_Atributo_o_Columna Operación Lógica (
--Consulta (Query) anidado.

SELECT      MAX(Atributo_1)

```

```

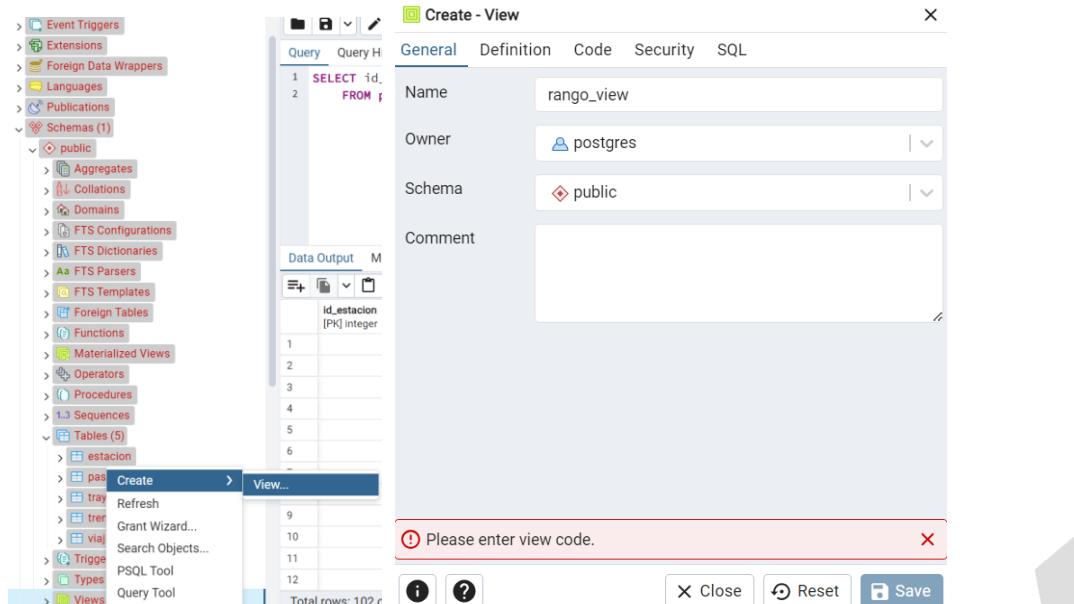
    FROM      Nombre_Tabla_o_Entidad
    ...
)

```

### Vistas Volátiles y Materializadas: Query que extrae datos actuales o históricos

- **VISTA Volátil:** También conocida simplemente como "vista", es una **tabla virtual** que almacena una consulta **SELECT** que se vaya a usar varias veces para extraer datos diferentes. Esta no almacena los datos físicamente en memoria; en cambio, cada vez que se accede a la **VISTA**, se ejecuta la consulta para mostrar los datos actuales de la **base de datos**.
- **VISTA Materializada:** Es una **vista** que almacena físicamente los resultados de una consulta **SELECT** en la memoria de la **base de datos**. Esta se actualiza periódicamente a una cierta hora o mediante un mecanismo de actualización definido, lo que significa que los datos pueden no estar siempre actualizados, osea que guarda un historial.

Si se quiere crear una **VISTA Volátil** dentro de **pgAdmin** se debe seleccionar la opción de: **Servers** → **Motor PostgreSQL** → **Nombre\_Database** → Schemas → public → **Views** → Clic derecho → Create → View... → **Pestaña General** → Name: **Nombre de la VISTA Volátil** → **Pestaña Code** → Introducir consulta SQL hecha con el comando **SELECT** → **Pestaña SQL** → Ver código SQL → Save.



```

1 SELECT *,  
2 CASE  
3 WHEN (fecha_nacimiento > '2015-01-01') THEN  
4 'Mayor'  
5 ELSE  
6 'Niño'  
7 END AS clasif_rango  
8 FROM pasajero  
9 ORDER BY clasif_rango;  
  

1 CREATE VIEW public.rango_edad_view  
2 AS  
3 SELECT *,  
4 CASE  
5 WHEN (fecha_nacimiento > '2015-01-01') THEN  
6 'Mayor'  
7 ELSE  
8 'Niño'  
9 END AS clasif_rango  
10 FROM pasajero  
11 ORDER BY clasif_rango;  
12  
13 ALTER TABLE public.rango_edad_view  
14 OWNER TO postgres;

```

Finalmente, para ejecutar una vista se debe correr el siguiente código SQL:

**SELECT \* FROM v\_Nombre\_Vista;**

Si se quiere crear una **VISTA Materializada** dentro de **pgAdmin** se debe seleccionar la opción de: **Servers** → **Motor PostgreSQL** → **Nombre\_Database** → **Schemas** → **public** → **Materialized Views** → Clic derecho → Create → Materialized View... → **Pestaña General** → **Name: Nombre de la VISTA Materializada** → **Pestaña Code** → Introducir consulta SQL hecha con el comando **SELECT** → **Pestaña SQL** → Ver código SQL → Save.

```

1 SELECT *  
2 FROM viaje  
3 WHERE tiempo_inicio > '22:00:00';  
  

1 CREATE MATERIALIZED VIEW public."rango_tiempo_Mview"  
2 AS  
3 SELECT *  
4 FROM viaje  
5 WHERE tiempo_inicio > '22:00:00'  
6 WITH NO DATA;  
7  
8 ALTER TABLE IF EXISTS public."rango_tiempo_Mview"  
9 OWNER TO postgres;

```

Finalmente, para ejecutar una vista se debe correr el siguiente código SQL dando clic en la opción de: **Servers** → **Motor PostgreSQL** → **Nombre\_Database** → **Schemas** → **public** → **Materialized Views** → **Nombre de la VISTA Materializada** → Clic derecho → Scripts → **SELECT** Script → Execute script. Primero se debe ejecutar una línea que actualice los datos de la memoria dentro de la **base de datos** donde se almacena la **vista** y luego ya podrá ver su contenido.

**REFRESH MATERIALIZED VIEW**

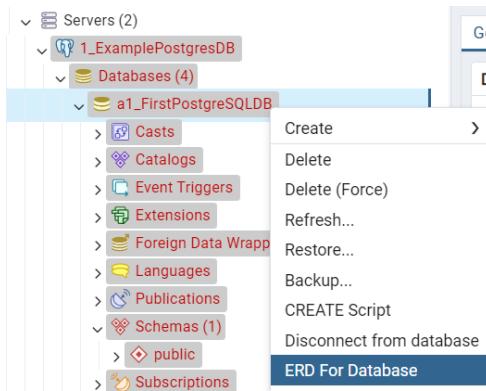
**SELECT \* FROM**

**public."v\_Nombre\_Vista\_Materializada";**

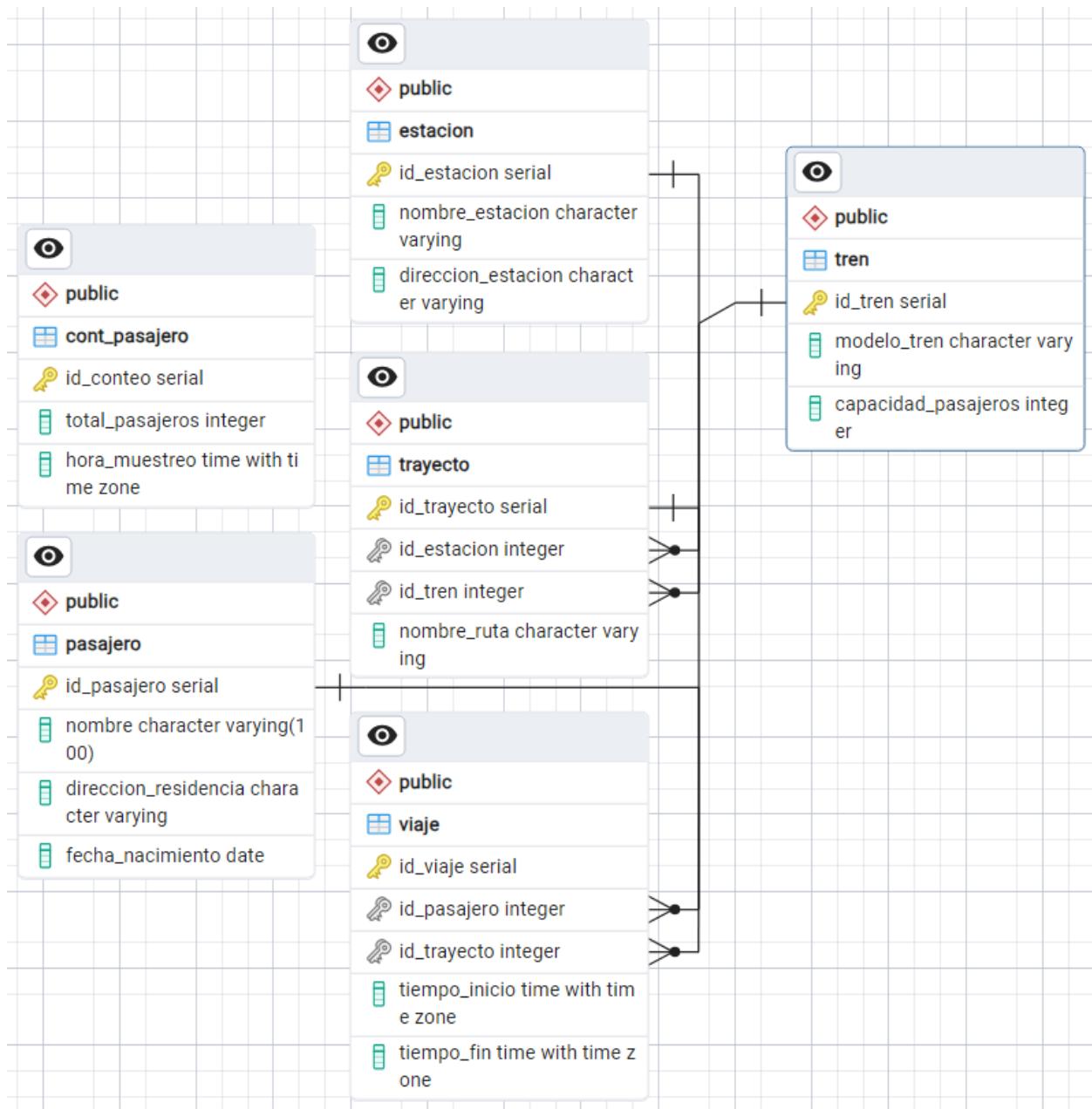
**public."v\_Nombre\_Vista\_Materializada";**

## Obtención del Diagrama ER De una Base de Datos Relacional

Para observar el **diagrama físico** de la **database relacional** debemos seleccionar la opción de: **Servers → Motor PostgreSQL → Nombre\_Database** → Clic derecho → EDR For Database.



Podremos ver el diagrama que incluye todas las **entidades** de la **DB** con sus **relaciones** y **tipos de dato**.



# Ejercicios Query de SQL

## De pregunta a Query

Lo que quieres mostrar = SELECT

De donde voy a tomar los datos = FROM

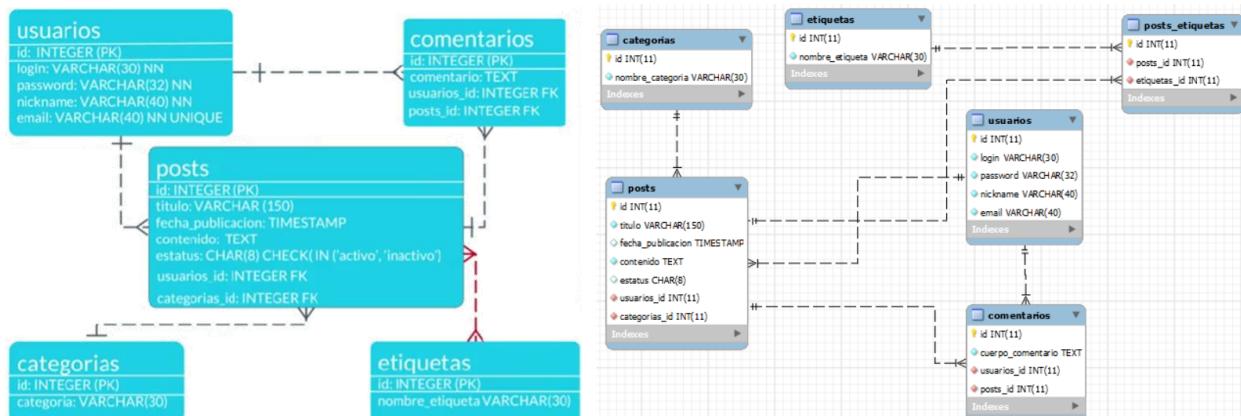
Los filtros de los datos que quieres mostrar = WHERE

Los rubros por los que me interesa agrupar la información = GROUP BY

El orden en que quiero presentar mi información ORDER BY

Los filtros que quiero que mis datos agrupados tengan HAVING

A continuación, responderemos algunas preguntas de prueba acerca de la base de datos relacional del blog:



1. ¿Cuántas etiquetas tiene cada post del blog?
  - a. Para identificar cada post podemos utilizar su título.
  - b. La información proviene de 3 tablas distintas: posts, etiquetas y posts\_etiquetas (tabla intermedia por la cardinalidad N:N).

- i. Debido a esta situación se deberá ejecutar un INNER JOIN doble que considere la intersección de las 3 tablas.
- c. La información se agrupa a través del id del post, ya que esa es la relación que hay entre la tabla de posts y la tabla de etiquetas y la información que quiero saber son las etiquetas contra el título del post.
- d. Podría colocar un orden numérico descendente para observar de más a menos el número de etiquetas de cada post.

The screenshot shows the MySQL Workbench interface. On the left, the Navigator pane displays the database schema, including the '1\_platziblog\_db' schema which contains tables like 'categorias', 'comentarios', 'etiquetas', 'posts', and 'posts\_etiquetas'. The main area shows a query editor with the following SQL code:

```

1  SELECT posts.titulo, COUNT(*) num_etiquetas
2  FROM posts
3  INNER JOIN posts_etiquetas ON posts.id = posts_etiquetas.posts_id
4  INNER JOIN etiquetas ON etiquetas.id = posts_etiquetas.etiquetas_id
5  GROUP BY posts.id
6  ORDER BY num_etiquetas DESC;

```

The results grid displays the following data:

título	num_etiquetas
Los mejores vestidos en la alfombra roja	4
La bolsa cae estrepitosamente	4
Se mejora la conducción autónoma de vehículos	4
Bienes raíces más baratos que nunca	3
Fuca OS sacude al mundo	3
Se descubre la unión entre astrofísica y física cu...	3
Químicos descubren nanomaterial	3
Equipo veterano da un gran espectáculo	3
Se descubre nueva partícula del modelo estandar	3
Escándalo en el mundo de la moda	3
Tenemos ganador de la formula e	2
Cierra campeonato mundial de football de mane...	2
Se fortalece el peso frente al dólar	2
U.S. Robotics presenta hallazgo	2
Ganador del premio Nobel por trabajo en genética	2
Tenemos un nuevo auto inteligente	2
Se presenta el nuevo teléfono móvil en evento	2
Tenemos campeona del mundial de voleibol	2
Ganan partido frente a visitantes	2
Los paparazzi captan escándalo en cámara	1

The right sidebar contains various tools: SQLAdditions, Result Grid, Form Editor, Field Types, Query Stats, and Execution Plan. The status bar at the bottom shows 'Object Info Session' and performance metrics: Duration / Fetch 0.000 sec / 0.000 sec.

2. Ahora que ya sé el número de etiquetas, ¿Cuáles etiquetas pertenecen a cada post del blog?
- a. Para saber el nombre de las etiquetas utilizaremos el método **GROUP\_CONCAT()** aplicado al **atributo** que indica el nombre de las etiquetas, manteniendo la misma estructura del código anterior.

Schemas: 1\_platziblog\_db

```

1 • SELECT posts.titulo, GROUP_CONCAT(nombre_etiqueta) AS nombre_tag
2   FROM posts
3     INNER JOIN posts_etiquetas ON posts.id = posts_etiquetas.posts_id
4     INNER JOIN etiquetas ON etiquetas.id = posts_etiquetas.etiquetas_id
5   GROUP BY posts.id
6   ORDER BY nombre_tag ASC

```

Result Grid:

titulo	nombre_tag
Tenemos ganador de la formula e	Automovilismo,Campeonatos
Fuccia OS sacude al mundo	Avances,Computación,Teléfonos Móviles
Se descubre nueva partícula del modelo estandar	Avances,Nobel,Física
Químicos descubren nanomaterial	Avances,Nobel,Química
Se fortalece el peso frente al dólar	Bolsa de valores,Inversiones
La bolsa cae estrepitosamente	Bolsa de valores,Inversiones,Brokers,Largo plazo
Tenemos campeona del mundial de voleibol	Campeonatos,Celebridades
Ganan partido frente a visitantes	Campeonatos,Equipos
Cierra campeonato mundial de football de mane...	Campeonatos,Equipos
Equipo veterano da un gran espectáculo	Campeonatos,Equipos,Celebridades
Los paparazzi captan escándalo en cámara	Celebridades
Escándalo con el boxeador del momento	Celebridades
Los mejores vestidos en la alfombra roja	Celebridades,Eventos,Moda,Estilo
Escándalo en el mundo de la moda	Celebridades,Moda,Estilo
Tenemos un nuevo auto inteligente	Computación,Automovilismo
Se mejora la conducción autónoma de vehículos	Computación,Automovilismo,Avances,Robótica
Bienes raíces más baratos que nunca	Inversiones,Largo plazo,Bienes Raíces
Ganador del premio Nobel por trabajo en genética	Nobel,Avances
Se descubre la unión entre astrofísica y física cu...	Nobel,Física,Avances
U.S. Robotics presenta hallazgo	Robótica,Avances

Output:

#	Time	Action	Message	Duration / Fetch
9	07:29:18	SELECT posts.titulo, GROUP_CONCAT(nombre_etiqueta) AS nombre... 21 row(s) returned		0.000 sec / 0.000 sec

- ¿Existe alguna etiqueta que no corresponda a ningún post?
- Quiero mostrar todas las etiquetas que no estén ligadas a ningún post.
- Los datos los voy a tomar de la tabla de las etiquetas, pero como quiero saber su conexión con post, no es necesario que analice post, solo la tabla de etiquetas y su tabla de transición intermedia.
- Debido a esta situación se deberá ejecutar un LEFT JOIN (de la tabla etiquetas), doble se considere solo las etiquetas que no tengan conexión, osea A – B, siendo A = etiquetas y B = tabla\_intermedia\_con\_conexión\_a\_posts.
- No es necesario agrupar la información.

Schemas: 1\_platziblog\_db

```

1 • SELECT *
2   FROM etiquetas
3     LEFT JOIN posts_etiquetas ON etiquetas.id = posts_etiquetas.etiquetas_id
4   WHERE posts_etiquetas.etiquetas_id IS NULL;

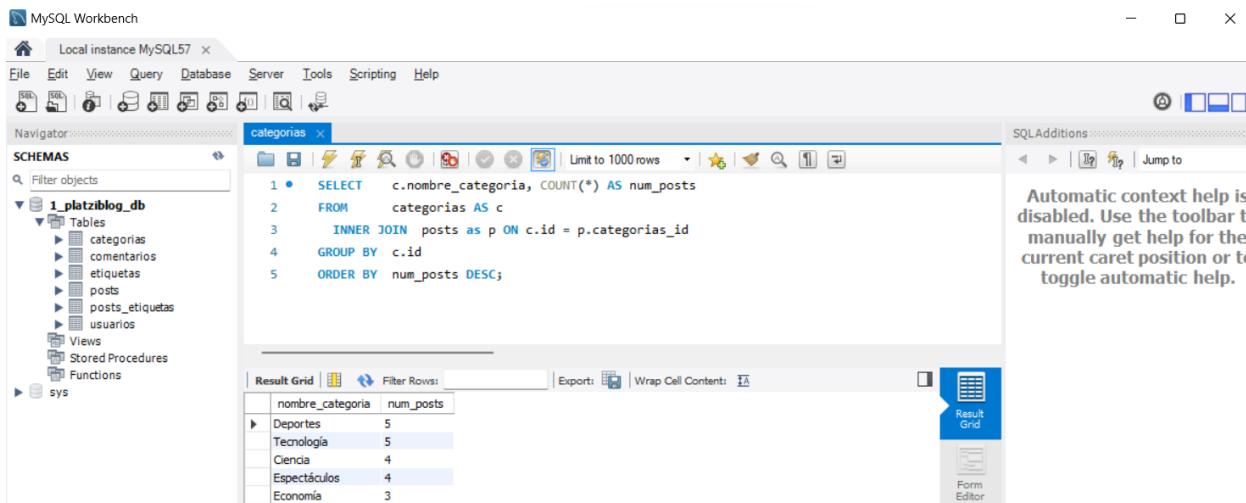
```

Result Grid:

id	nombre_etiqueta	id	posts_id	etiquetas_id
15	Matemáticas	HULL	HULL	HULL

- ¿Cuál categoría posee el mayor número de posts?
- Los datos que se quieren mostrar son el nombre de la categoría y el número de posts que corresponden a cada una.
- La información proviene de 2 tablas distintas: posts y etiquetas.

- i. Como se busca encontrar los datos relacionados se deberá ejecutar un INNER JOIN que considere la intersección de las 2 tablas, osea  $A \cap B$ , recordemos que esto se logra al utilizar el índice que relaciona ambas tablas.
- c. La información se agrupa a través del id de la categoría porque de esa manera se podrá mostrar cada tipo de categoría distinta.
- d. Se colocará un orden numérico descendente para observar de mayor a menor el número de posts de cada etiqueta.



The screenshot shows the MySQL Workbench interface. The left sidebar displays the database schema with tables like categorias, comentarios, etiquetas, posts, posts\_etiquetas, and usuarios. The central pane shows a query editor with the following SQL code:

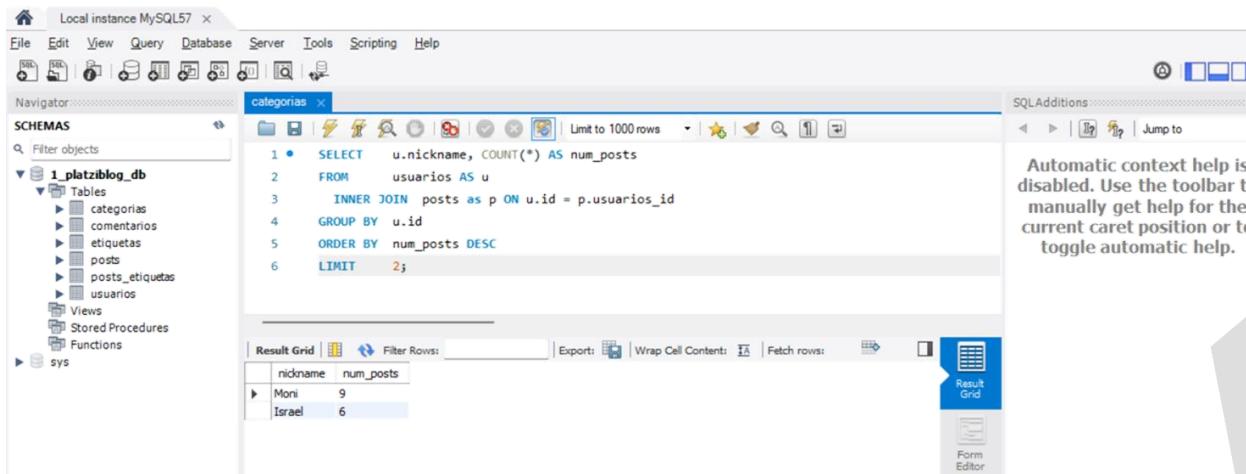
```

1 • SELECT c.nombre_categoria, COUNT(*) AS num_posts
2   FROM categorias AS c
3   INNER JOIN posts AS p ON c.id = p.categorias_id
4   GROUP BY c.id
5   ORDER BY num_posts DESC;
    
```

The results grid shows the following data:

nombre_categoria	num_posts
Deportes	5
Tecnología	5
Ciencia	4
Especiales	4
Economía	3

- 5. ¿Qué usuario ha creado el mayor número de posts en el sistema?
- a. El procedimiento es el mismo al ejercicio anterior, pero cambiando la tabla de donde provienen los datos y el dato mismo que se quiere mostrar.
- b. Opcionalmente nos podemos limitar a mostrar 1 solo valor, de esta manera mostrando solo el mayor o los primeros dos, para comprobar si en el segundo se repite el número.



The screenshot shows the MySQL Workbench interface. The left sidebar displays the database schema with tables like categorias, comentarios, etiquetas, posts, posts\_etiquetas, and usuarios. The central pane shows a query editor with the following SQL code:

```

1 • SELECT u.nickname, COUNT(*) AS num_posts
2   FROM usuarios AS u
3   INNER JOIN posts AS p ON u.id = p.usuarios_id
4   GROUP BY u.id
5   ORDER BY num_posts DESC
6   LIMIT 2;
    
```

The results grid shows the following data:

nickname	num_posts
Moni	9
Israel	6

- 6. ¿De qué categorías (temas) están escribiendo los 3 usuarios que han creado el mayor número de posts en el sistema?
- a. El código resultante del ejercicio anterior se repite, pero se debe añadir el dato adicional que se está solicitando, que en este caso son las categorías de las que está escribiendo el usuario.

- Para obtener y mostrar una lista de las categorías de temas de los que escribe cada usuario se utiliza el método **GROUP\_CONCAT()** aplicado al nombre de las categorías de temas.
- Como la información proviene de 3 tablas distintas: usuarios, categorías y posts, se debe realizar una interconexión de todas ellas.
- Debido a que se están buscando los datos que pertenezcan a las 3 tablas a la vez, se deberá ejecutar un INNER JOIN doble que considere la intersección de las 3 tablas osea  $A \cap B \cap C$ , recordemos que esto se logra al utilizar el índice que relaciona cada una de las tablas por separado.

The screenshot shows the MySQL Workbench interface. The left sidebar displays the 'schemas' tree, with '1\_platziblog\_db' selected, showing tables like 'categorias', 'comentarios', 'etiquetas', 'posts', 'posts\_etiquetas', and 'usuarios'. The central pane is titled 'categorias' and contains a SQL editor with the following query:

```

1 • SELECT u.nickname, COUNT(*) AS num_posts, GROUP_CONCAT(c.nombre_categoria)
2   FROM usuarios AS u
3     INNER JOIN posts AS p ON u.id = p.usuarios_id
4     INNER JOIN categorias AS c ON c.id = p.categorias_id
5   GROUP BY u.id
6   ORDER BY num_posts DESC
7   LIMIT 3;
  
```

The 'Result Grid' below the query shows three rows of data:

nickname	num_posts	GROUP_CONCAT(c.nombre_categoria)
Moni	9	Deportes,Deportes,Deportes,Tecnología,Ciencia,Ciencia,Economía,Economía,Deportes
Israel	6	Tecnología,Tecnología,Tecnología,Tecnología,Economía,Deportes
Ed	4	Especiales,Especiales,Especiales,Especiales

The right sidebar contains 'SQLAdditions' and a note: 'Automatic context help is disabled. Use the toolbar to manually get help for the current caret position or to toggle automatic help.'

#### 7. ¿Qué usuarios no han escrito ningún post?

- Se busca mostrar todos los nombres de los usuarios que no estén ligados a ningún post.
- Los datos los voy a tomar de la tabla de los usuarios y de los posts.
- Debido que quiero saber todos los usuarios que no tengan ningún post se ejecutará una operación de LEFT JOIN, doble se considere solo los usuarios que no tengan conexión, osea  $A - B$ , siendo  $A =$ usuarios y  $B =$ posts.
- El filtro que se aplicará es encontrar las filas de datos donde el post sea nulo para lograr la operación  $A - B$ .

The screenshot shows the MySQL Workbench interface. The left sidebar displays the 'schemas' tree, with '1\_platziblog\_db' selected, showing tables like 'categorias', 'comentarios', 'etiquetas', 'posts', 'posts\_etiquetas', and 'usuarios'. The central pane is titled 'categorias' and contains a SQL editor with the following query:

```

1 • SELECT *
2   FROM usuarios AS u
3     LEFT JOIN posts AS p ON u.id = p.usuarios_id
4   WHERE p.usuarios_id IS NULL;
  
```

The 'Result Grid' below the query shows one row of data:

id	login	password	nickname	email	id	título	fecha_publicación	conte...
5	perezoso	&N_>S)_Y*&TG...	Oso Pérez	perezoso@platziblog.com	HULL	HULL	HULL	HULL

The right sidebar contains 'SQLAdditions' and a note: 'Automatic context help is disabled. Use the toolbar to manually get help for the current caret position or to toggle automatic help.'

## Basic Queries

```
-- filter your columns  
SELECT col1, col2, col3, ... FROM table1  
-- filter the rows  
WHERE col4 = 1 AND col5 = 2  
-- aggregate the data  
GROUP by ...  
-- limit aggregated data  
HAVING count(*) > 1  
-- order of the results  
ORDER BY col2
```

### Useful keywords for SELECTS:

**DISTINCT** - return unique results  
**BETWEEN** a AND b - limit the range, the values can be numbers, text, or dates  
**LIKE** - pattern search within the column text  
**IN** (a, b, c) - check if the value is contained among given.

## Data Modification

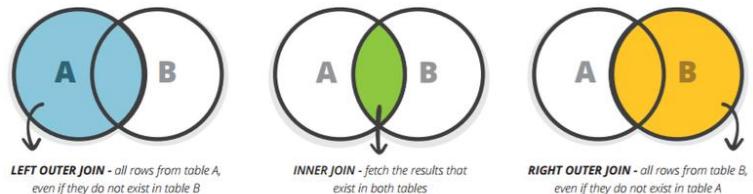
```
-- update specific data with the WHERE clause  
UPDATE table1 SET col1 = 1 WHERE col2 = 2  
-- insert values manually  
INSERT INTO table1 (ID, FIRST_NAME, LAST_NAME)  
VALUES (1, 'Rebel', 'Labs');  
-- or by using the results of a query  
INSERT INTO table1 (ID, FIRST_NAME, LAST_NAME)  
SELECT id, last_name, first_name FROM table2
```

## Views

A **VIEW** is a virtual table, which is a result of a query. They can be used to create virtual tables of complex queries.

```
CREATE VIEW view1 AS  
SELECT col1, col2  
FROM table1  
WHERE ...
```

## The Joy of JOINS



**LEFT OUTER JOIN** - all rows from table A, even if they do not exist in table B

**INNER JOIN** - fetch the results that exist in both tables

**RIGHT OUTER JOIN** - all rows from table B, even if they do not exist in table A

## Updates on JOINed Queries

You can use **JOINS** in your **UPDATES**  
UPDATE t1 SET a = 1  
FROM table1 t1 JOIN table2 t2 ON t1.id = t2.t1\_id  
WHERE t1.col1 = 0 AND t2.col2 IS NULL;

NB! Use database specific syntax, it might be faster!

## Semi JOINS

You can use subqueries instead of **JOINS**:

```
SELECT col1, col2 FROM table1 WHERE id IN  
(SELECT t1_id FROM table2 WHERE date >  
CURRENT_TIMESTAMP)
```

## Indexes

If you query by a column, index it!

```
CREATE INDEX index1 ON table1 (col1)
```

### Don't forget:

Avoid overlapping indexes  
Avoid indexing on too many columns  
Indexes can speed up **DELETE** and **UPDATE** operations

## Useful Utility Functions

-- convert strings to dates:  
**TO\_DATE** (Oracle, PostgreSQL), **STR\_TO\_DATE** (MySQL)

-- return the first non-NULL argument:

**COALESCE** (col1, col2, "default value")

-- return current time:

**CURRENT\_TIMESTAMP**

-- compute set operations on two result sets

**SELECT** col1, col2 **FROM** table1

**UNION** / **EXCEPT** / **INTERSECT**

**SELECT** col3, col4 **FROM** table2;

**Union** - returns data from both queries

**Except** - rows from the first query that are not present in the second query

**Intersect** - rows that are returned from both queries

## Reporting

Use aggregation functions

**COUNT** - return the number of rows

**SUM** - cumulate the values

**AVG** - return the average for the group

**MIN** / **MAX** - smallest / largest value

BROUGHT TO YOU BY  
**XRebel**

## Código SQL - Creación y/o Modificación de la Base de Datos (DDL y DML)

# Referencias

Platzi, Israel Vázquez, “Curso de Fundamentos de Bases de Datos”, 2018 [Online], Available: <https://platzi.com/new-home/clases/1566-bd/19781-bienvenida-conceptos-basicos-y-contexto-historico-/>

Platzi, Oswaldo Rodríguez, “Curso de PostgreSQL”, 2019 [Online], Available: <https://platzi.com/cursos/postgresql/>

Platzi, Israel Vázquez Morales, “Curso Práctico de SQL”, 2020 [Online], Available: <https://platzi.com/cursos/practico-sql/>