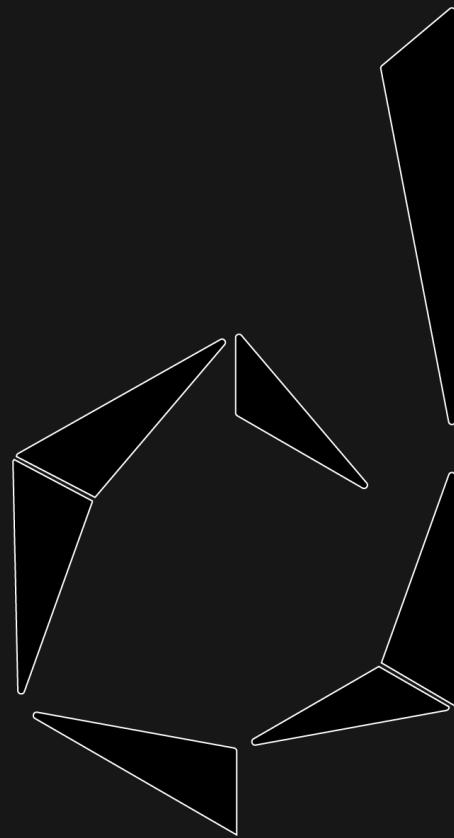


INGENIERÍA MECATRÓNICA



DI_CERO

DIEGO CERVANTES RODRÍGUEZ

PROGRAMACIÓN: DESARROLLO BACKEND

SQL

Ejercicios Avanzados SQL, DB
Distribuidas y Window Functions:
Queries PostgreSQL 🐘

Contenido

Representación de las Bases de Datos: Nomenclatura de Chen	3
Lenguaje de Programación SQL - PostgreSQL	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
Sub-lenguajes de SQL: DDL (Data Definition Language)	16
CREATE:	16
Crear una Base de Datos con SQL	16
Crear una Tabla con SQL.....	17
Crear una Vista con SQL.....	18
ALTER:	18
Alterar una Tabla con SQL.....	18
Llaves Principales y Foráneas (PRIMARY & FOREIGN KEYS).....	18
DROP:	20
Borrar una Tabla, Columna o Base de Datos con SQL	20
Sub-lenguajes de SQL: DML (Data Manipulation Language)	20
INSERT:	21
Insertar Datos Nuevos a la Tabla de una Base de Datos con SQL	21
UPDATE:	21
Editar Datos en la Tabla de una Base de Datos con SQL	21
Funciones Especiales de Postgres para los Comandos INSERT Y UPDATE.....	22
DELETE:	23
Borrar Todos los Datos de una Fila Perteneciente a la Tabla de una Base de Datos	23
SELECT:	23
Extraer Todos los Datos de una Columna Perteneciente a la Tabla de una Base de Datos	23
Consultas o Queries: Extracción de información de una base de datos.....	24
Funciones Especiales de Postgres para el Comando SELECT	30
Crear una Vista: Almacenar un Query en una “Variable” con SQL	31
Nested Queries: Consultas Anidadas (Agujero de Conejo)	32
Vistas Volátiles y Materializadas: Query que extrae datos actuales o históricos	33
Obtención del Diagrama ER De una Base de Datos Relacional	34

Ejercicios Avanzados de SQL Queries	35
1.- Obtención de primera fila: FETCH, NESTED QUERY y ROW NUMBER()	36
2.- Segundo valor más alto: DISTINCT, COUNT(), LIMIT, OFFSET, etc.	36
3.- Segunda mitad de una tabla: ROW_NUMBER() & COUNT()	37
4.- Consulta con Arrays: WHERE/IN	38
5.- Consulta de Tiempos y Fechas: EXTRACT y DATE_PART()	39
6.- Consulta de Duplicados: CAST (::), WHERE/IN y DELETE	40
7.- Selectores de Intervalos: Tipos de dato Range	41
8.- Subconsultas con Intervalos: Tipos de dato Range, MIN() y MAX()	45
9.- Consulta de 2 o más Máximos y Mínimos: GROUP BY, MAX() y MIN()	45
10.- Datos en Orden Alfabético y Otra Columna: GROUP BY, MAX() y MIN()	47
11.- Self JOINs: CONCAT(), JOIN, AS & ON	48
12.- Self JOINs - Top 10: JOIN, COUNT(), GROUP/ORDER BY & LIMIT	49
13.- Self JOINs - Promedio SubQuery: JOIN, AVG() & GROUP BY	50
14.- JOINs - Diferencias con dos tablas (A-B): CONCAT(), LEFT JOIN, etc.	51
15.- Tipos de JOINs con 2 tablas: Unión, Diferencia, Intersección, etc.	53
16.- Autocompletando Strings/Figuras Geométricas: lpad() & CAST()	57
17.- Generación de Rangos: generate_series()	58
18.- Expresiones Regulares - Validación de Patrones:	61
Bases de Datos Distribuidas	62
Ejercicio de Query Distribuido	63
Pasos de resolución para el Ejercicio de Consultas en una Base de Datos Distribuida	64
Sharding (Particiones)	66
Particiones en PgAdmin de PostgreSQL.....	67
Window Functions.....	70
Referencias.....	71



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**:

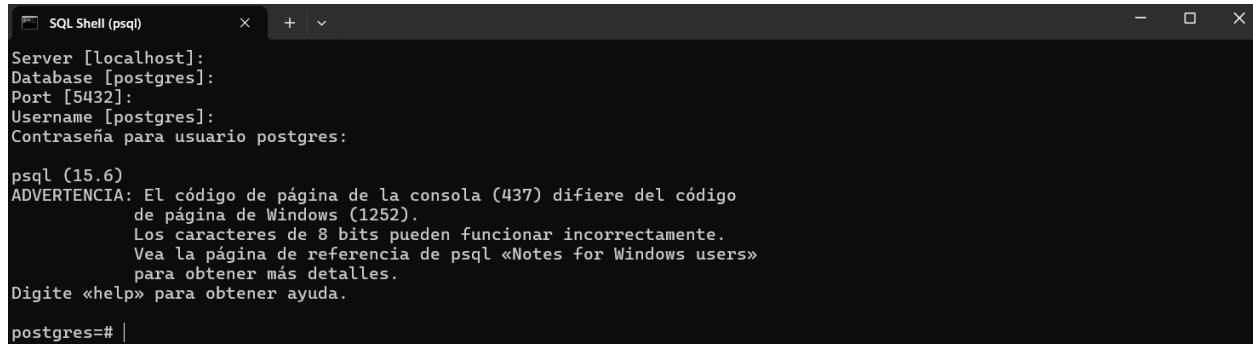
- **Servidor de base de datos:** Es un computador o servicio de nube con una URL o dominio asignado que tiene un **motor de base de datos** instalado y en ejecución.
- **Motor de base de datos:** Se refiere a un software que provee un conjunto de funcionalidades encargadas de administrar una **DB**, 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.

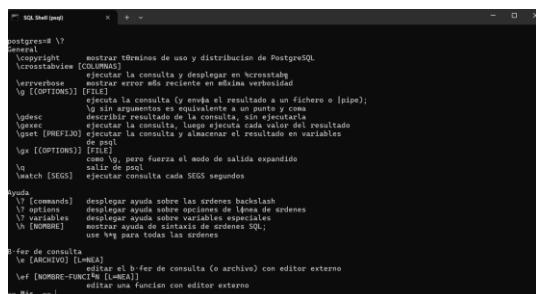
Digite «help» para obtener ayuda.

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
\copyright   mostrar términos de uso y distribución de PostgreSQL
\c          cambiar nombre de conexión actual
\crosscheck    mostrar errores más reciente en máxima verosimilitud
\g           mostrar error más reciente en máxima verosimilitud
\goptions [FILE]   ejecuta la consulta y envía el resultado a un fichero o |pipe|;
                   \g sin argumentos es equivalente a un punto y coma
\help        desplegar ayuda sobre la consulta, función, tipo, 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
\qg          salir de psql
\set         ejecutar consulta cada SESS segundos
\k [COMMANDS]  despliega ayuda sobre las ordenes backslash
\l [OPTIONS]  despliega ayuda sobre opciones de líneas de órdenes
\variables   despliega ayuda sobre variables especiales
\w [NOMBRE]   usar \w para guardar el resultado de una consulta de SQL;
                   use \w* para todas las órdenes
Ayuda
\h [COMMAND] despliega ayuda sobre las órdenes backslash
\l [OPTIONS] despliega ayuda sobre opciones de líneas de órdenes
\variables despliega ayuda sobre variables especiales
\w [NOMBRE]  usar \w para guardar el resultado de una consulta de SQL;
                   use \w* para todas las órdenes
\bf [CONSULTA] editar la -\bf de consulta (o archivo) con editor externo
\ef [NOMBRE-FUNCION] [-l[NEA]] 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`.



```

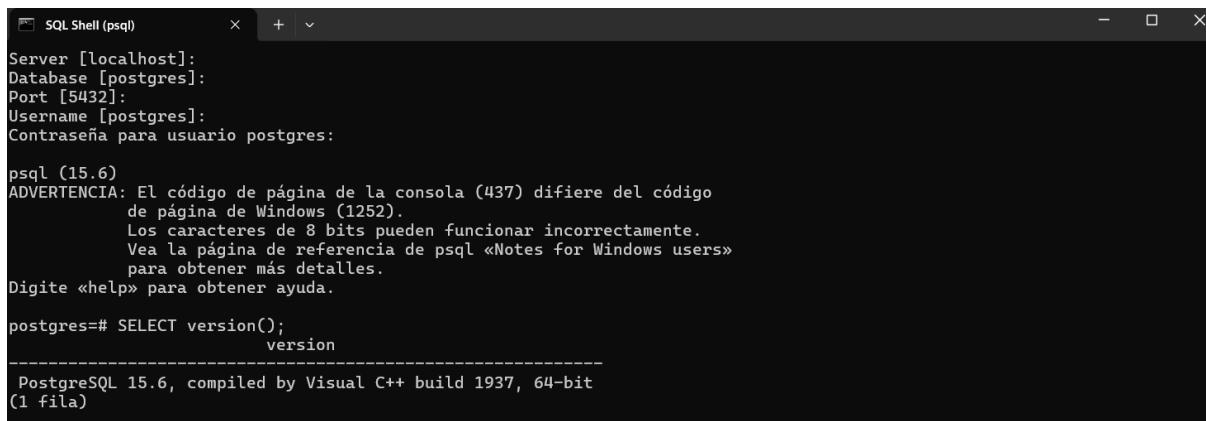
SQL Shell (psql)      x  +  v
postgres=# \l
                                         Listado de base de datos
   Nombre    | Dueño    | Codificación | Collate           | Ctype            | configuración ICU | Proveedor de loca
   | Privilegios
---+-----+-----+-----+-----+-----+-----+
postgres | postgres | 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.



```

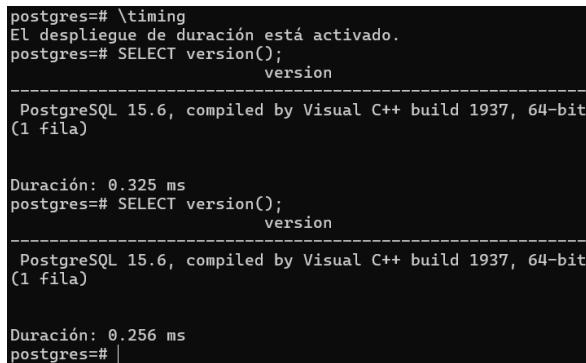
SQL Shell (psql)  X + 
Server [localhost]: Database [postgres]:
Port [5432]: Username [postgres]:
Contraseña para usuario postgres:

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)

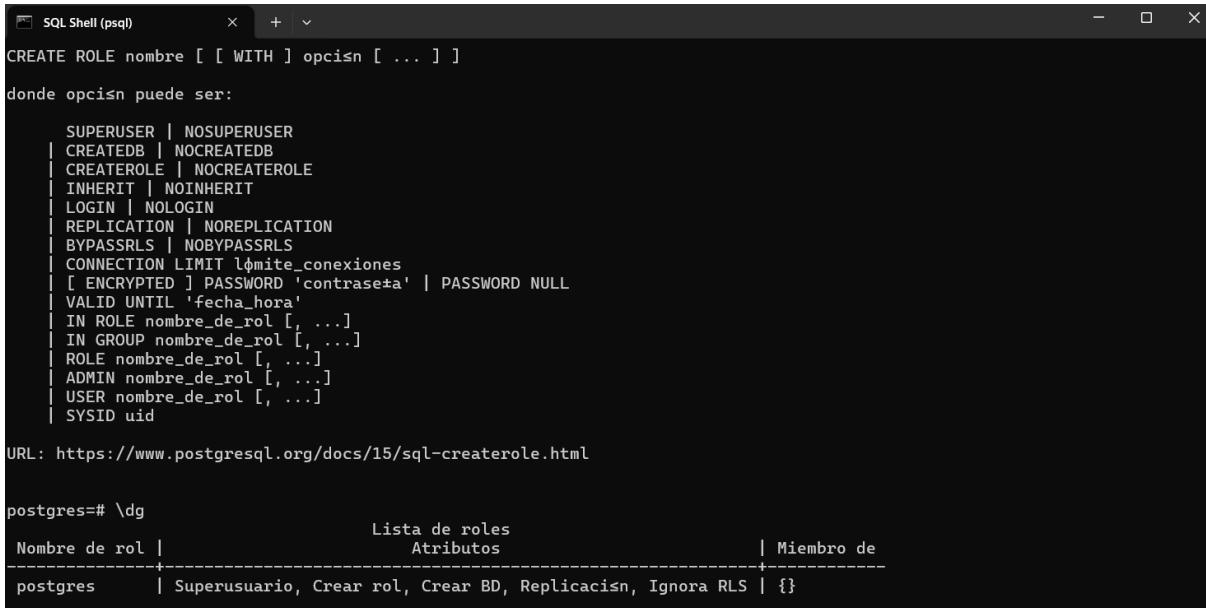
Duración: 0.325 ms
postgres=# SELECT version();
              version
-----
PostgreSQL 15.6, compiled by Visual C++ build 1937, 64-bit
(1 fila)

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 | {}

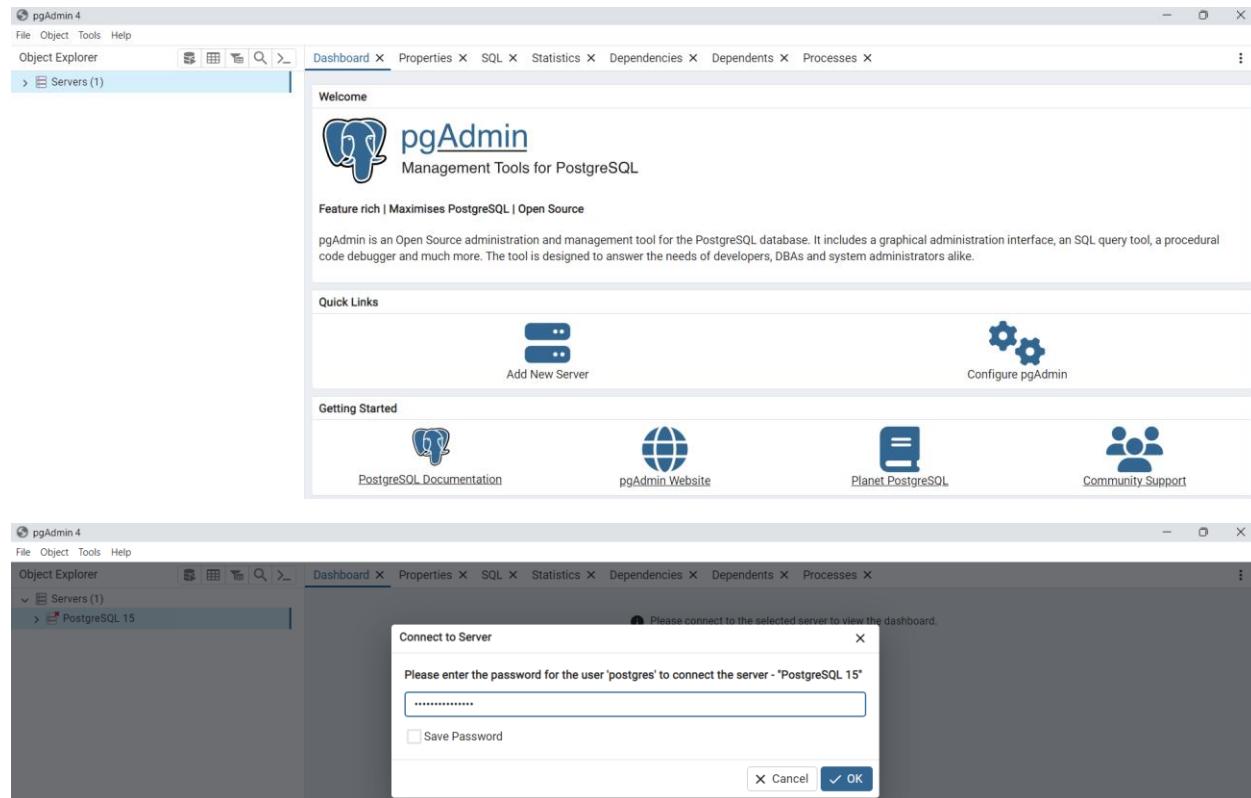
```

Funcionalidades Comunes de la Interfaz Gráfica de PostgreSQL: pgAdmin

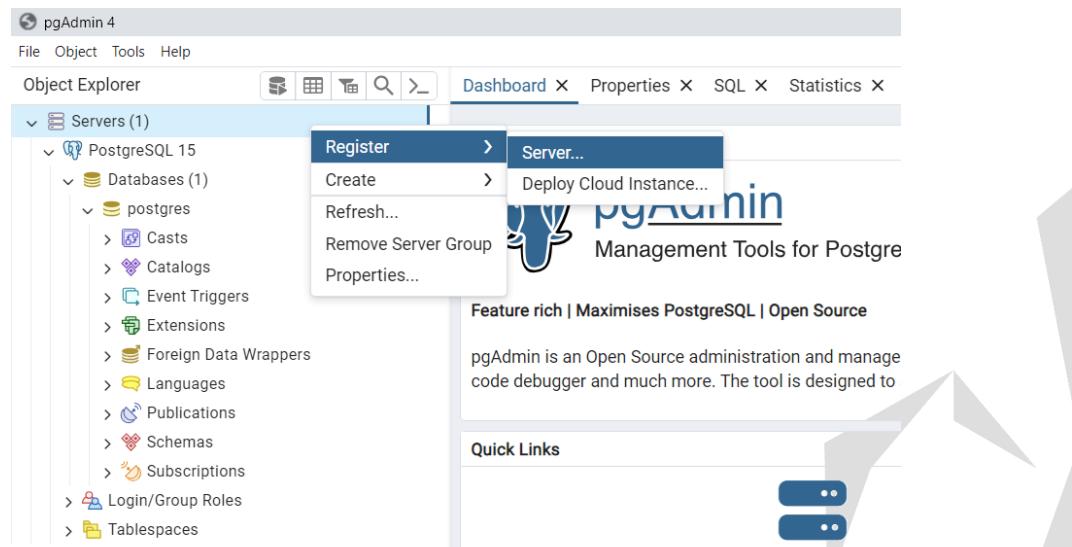
Para acceder a la interfaz simplemente debemos buscar y ejecutar la aplicación de escritorio llamada **pgAdmin**, ésta a veces se puede abrir en una pantalla emergente o abrirse a través de algún navegador web, cualquiera de las dos opciones funciona y se ejecuta en una u otra dependiendo de la versión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



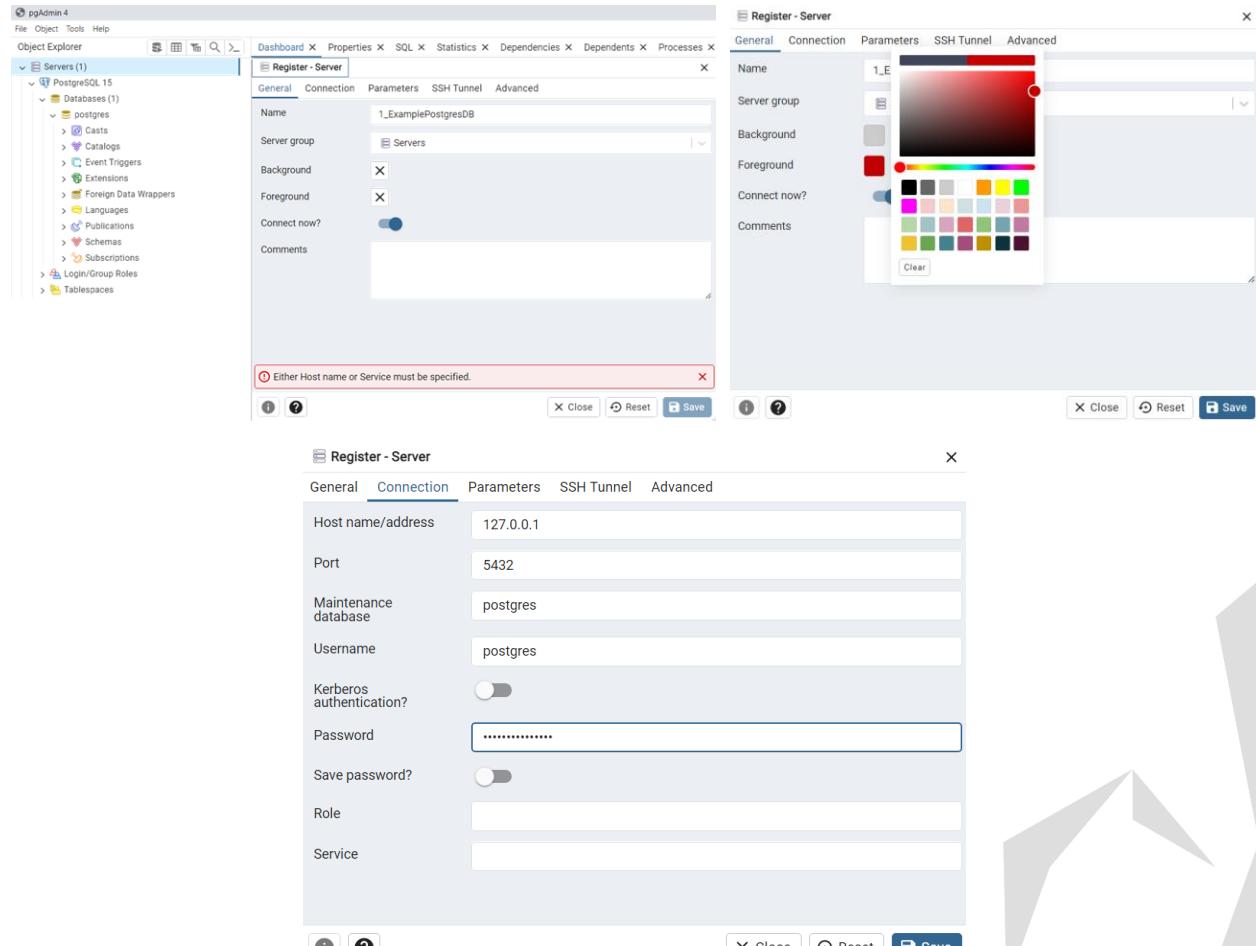
Una vez que nos encontremos dentro de la interfaz, podemos conectar nuevos computadores o servicios de nube que tengan el **motor de base de datos PostgreSQL** instalado y en ejecución al RDBMS (Relational DataBase Management System) de **pgAdmin** dando clic derecho sobre el ícono de **Servers** y seleccionando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



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

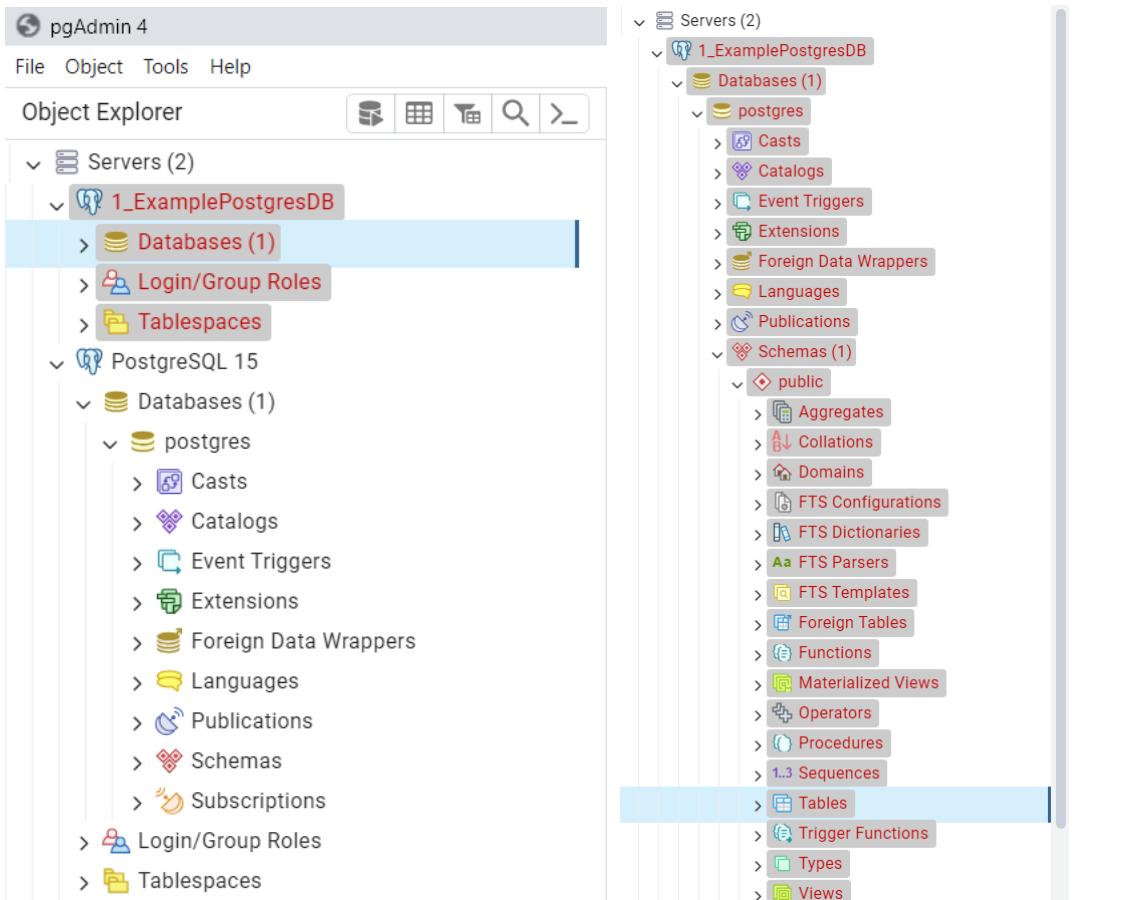
- **Host name/address:** En este parámetro se indica la dirección IP donde se encuentra localizada la **base de datos**, se puede utilizar **localhost**, que corresponde a la dirección IP **127.0.0.1** (representan lo mismo).
- **Port:** En este parámetro se indica el número de puerto de conexión al 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.

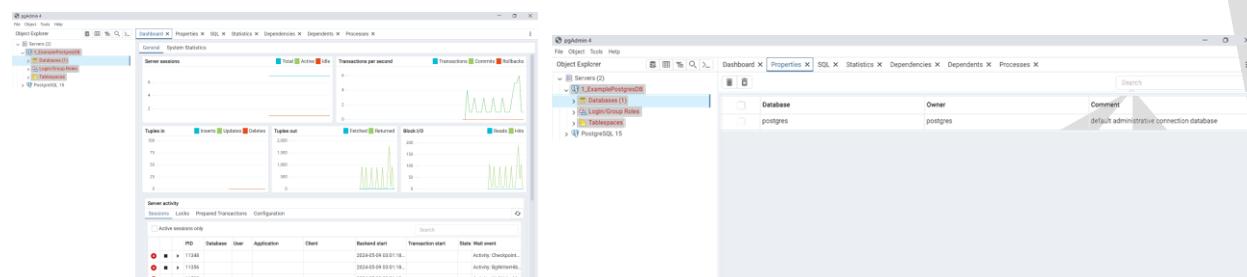


Y ahora tendremos dos accesos a diferentes **bases de datos** con distintas conexiones a servidores **PostgreSQL**, el **servidor** predeterminado se llama **PostgreSQL 15** y nuestro **propio servidor local** fue llamado **1_ExamplePostgresDB**, estos accesos nos permiten ver las siguientes características:

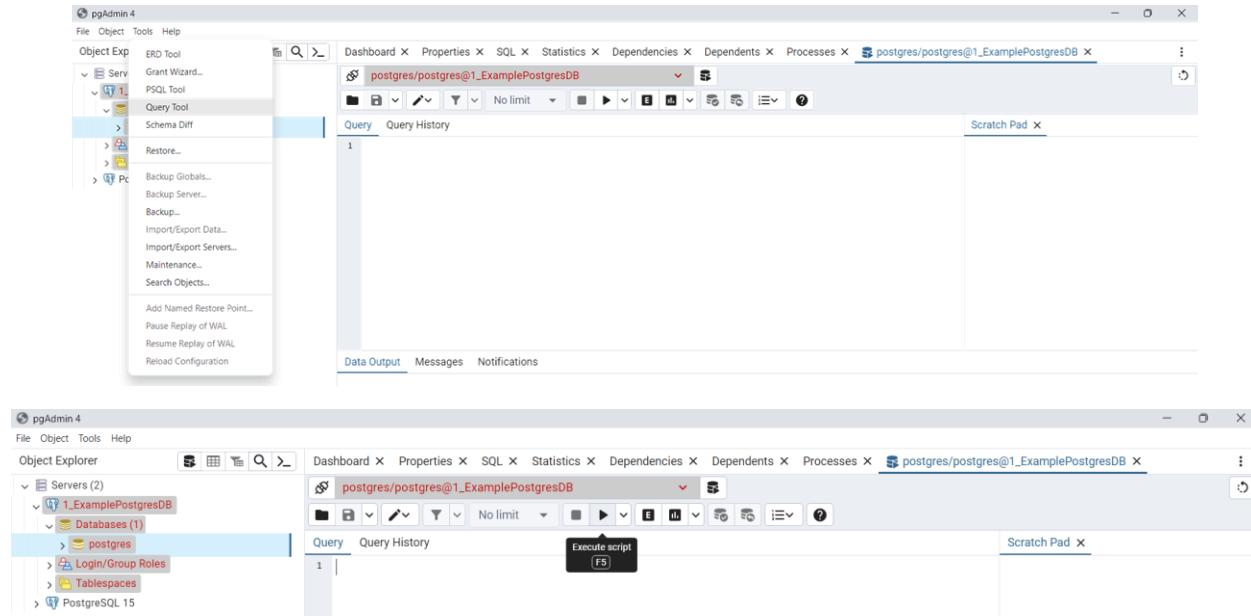
- **Databases**: Nos deja ver información de las **databases**, varias opciones internas son propias de **PostgreSQL**, pero la que nos interesa para poder ver las **tablas** de la **base de datos** es **Schemas** → **Tables** → **Nombre de la Tabla a la que quiero acceder**.
- **Logins/group roles**: 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.



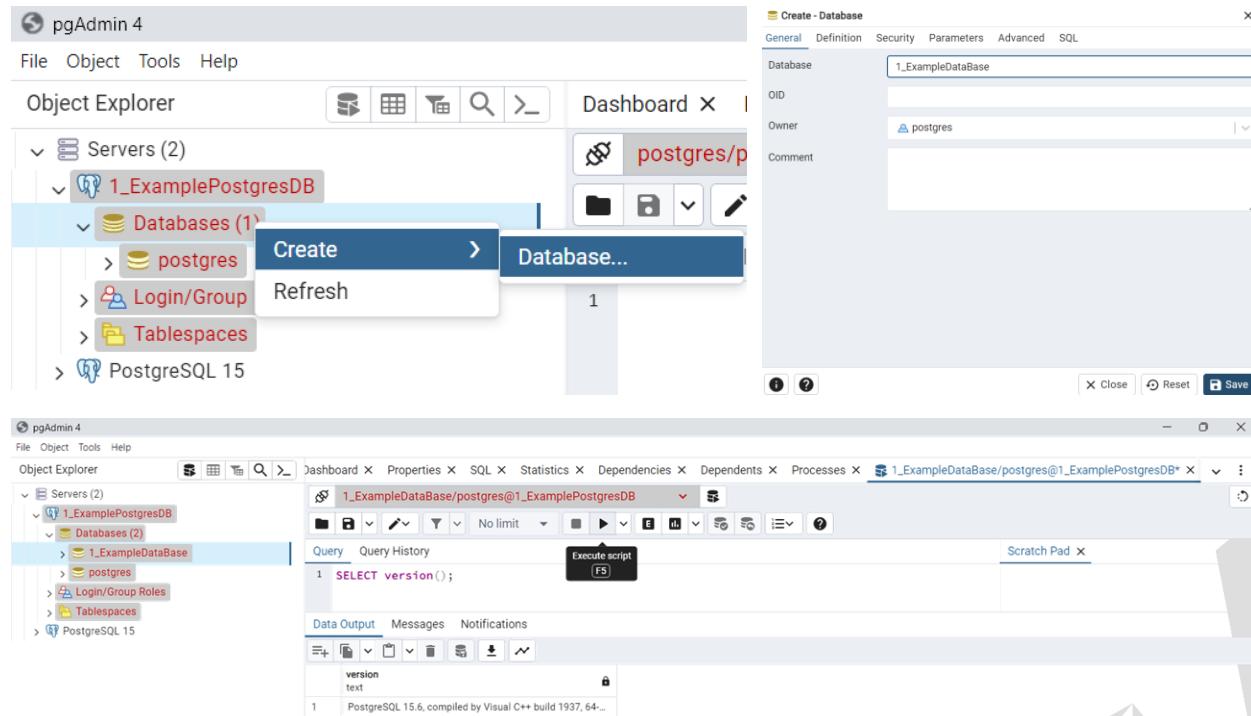
Podemos además observar varias otras características de la base de datos como sus transacciones por segundo, propiedades, código SQL, estadísticas, etc.



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.

The image displays three separate windows of the PostgreSQL Group Role configuration interface, each showing a different tab: **Privileges**, **Membership**, and **SQL**.

Privileges Tab:

Setting	Status
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

Membership Tab:

Role	Privilege
User/Role	WITH ADMIN
Members	WITH ADMIN

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 message 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 tendrá **permisos** para acceder a las **tablas** del **database**, esto se ve reflejado al ver el código SQL de las **entidades**.

```

pgAdmin 4
File Object Tools Help
Object Explorer Dashboard Properties SQL Statistics Dependencies Dependents Processes postgres/postgre... X
Schemas (1)
  public
    Aggregates
    Collations
    Domains
    FTS Configurations
    FTS Dictionaries
    FTS Parsers
    FTS Templates
    Foreign Tables
    Functions
    Materialized Views
    Operators
    Procedures
    Sequences
    Tables (4)
      estacion
      pasajero
      trayecto
      tren
  -- Table: public.estacion
  --
  1 -- DROP TABLE IF EXISTS public.estacion;
  2
  3 -- CREATE TABLE IF NOT EXISTS public.estacion
  4 (
  5   id_estacion integer NOT NULL DEFAULT nextval('estacion_id_estacion_seq'::regclass),
  6   nombre_estacion character varying COLLATE pg_catalog."default",
  7   direccion_estacion character varying COLLATE pg_catalog."default",
  8   CONSTRAINT estacion_pkey PRIMARY KEY (id_estacion)
  9 )
  10
  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;

```

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 haberlas 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.

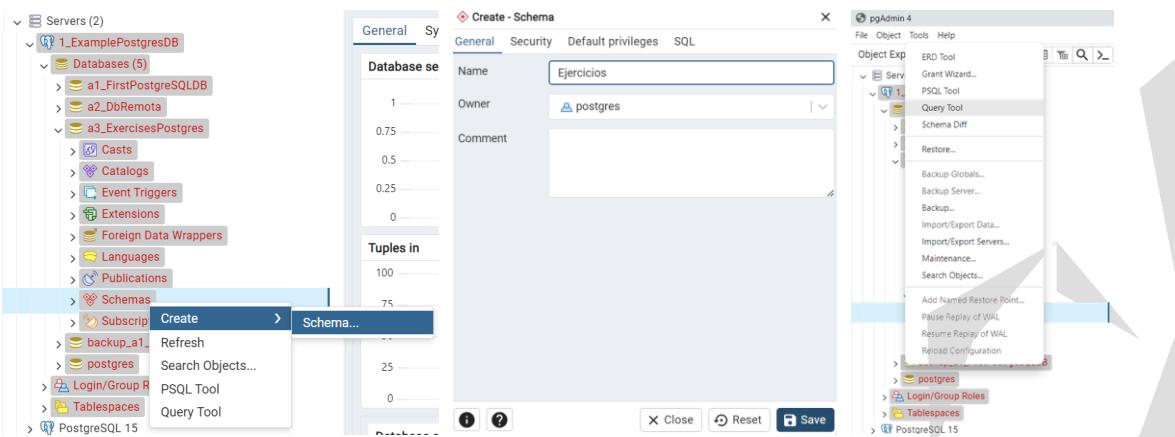
Crear una Tabla con SQL

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 su **base de datos**, 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 uno a uno los **atributos** de la **entidad** indicando su **tipo de dato** y **constraints básicos** → **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.

Aunque si ya contamos 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.



```

a3_ExercisesPostgres/postgres@1_ExamplePostgresDB
Query History No limit Execute script (FS)
1 CREATE TABLE ejercicios.carreras
2   id INT,
3   carrera VARCHAR(100),
4   fecha_alta TIMESTAMP,
5   vigente BOOLEAN
6 );
7 INSERT INTO ejercicios.carreras (id, carrera, fecha_alta, vigente) VALUES (1, 'Negocios y adm')
8 INSERT INTO ejercicios.carreras (id, carrera, fecha_alta, vigente) VALUES (2, 'Administración')
9 INSERT INTO ejercicios.carreras (id, carrera, fecha_alta, vigente) VALUES (3, 'Contabilidad y
10 INSERT INTO ejercicios.carreras (id, carrera, fecha_alta, vigente) VALUES (4, 'Finanzas, banci
11 INSERT INTO ejercicios.carreras (id, carrera, fecha_alta, vigente) VALUES (5, 'Mercadotecnia')
12 INSERT INTO ejercicios.carreras (id, carrera, fecha_alta, vigente) VALUES (6, 'Negocios y adm')
13 INSERT INTO ejercicios.carreras (id, carrera, fecha_alta, vigente) VALUES (7, 'Negocios y com
14 INSERT INTO ejercicios.carreras (id, carrera, fecha_alta, vigente) VALUES (8, 'Ciencias social
15 INSERT INTO ejercicios.carreras (id, carrera, fecha_alta, vigente) VALUES (9, 'Ciencias polític
16 INSERT INTO ejercicios.carreras (id, carrera, fecha_alta, vigente) VALUES (10, 'Economía', '20
17 INSERT INTO ejercicios.carreras (id, carrera, fecha_alta, vigente) VALUES (11, 'Psicología', '20
18 INSERT INTO ejercicios.carreras (id, carrera, fecha_alta, vigente) VALUES (12, 'Sociología y
19 INSERT INTO ejercicios.carreras (id, carrera, fecha_alta, vigente) VALUES (13, 'Trabajo y aten
20 INSERT INTO ejercicios.carreras (id, carrera, fecha_alta, vigente) VALUES (14, 'Ciencias de la
21 INSERT INTO ejercicios.carreras (id, carrera, fecha_alta, vigente) VALUES (15, 'Comunicación
Data Output Messages Notifications
INSERT 0 1
Query returned successfully in 66 msec.

a3_ExercisesPostgres/postgres@1_ExamplePostgresDB
Query History No limit Execute script (FS)
1 create table ejercicios.alumnos (
2   id INT,
3   nombre VARCHAR(50),
4   apellido VARCHAR(50),
5   email VARCHAR(50),
6   colegiatura FLOAT,
7   fecha_incorporacion TIMESTAMP,
8   carrera_id INT,
9   tutor_id INT
10 );
11 insert into ejercicios.alumnos (id, nombre, apellido, email, colegiatura, fecha_incorporacion
12 insert into ejercicios.alumnos (id, nombre, apellido, email, colegiatura, fecha_incorporacion
13 insert into ejercicios.alumnos (id, nombre, apellido, email, colegiatura, fecha_incorporacion
14 insert into ejercicios.alumnos (id, nombre, apellido, email, colegiatura, fecha_incorporacion
15 insert into ejercicios.alumnos (id, nombre, apellido, email, colegiatura, fecha_incorporacion
16 insert into ejercicios.alumnos (id, nombre, apellido, email, colegiatura, fecha_incorporacion
17 insert into ejercicios.alumnos (id, nombre, apellido, email, colegiatura, fecha_incorporacion
18 insert into ejercicios.alumnos (id, nombre, apellido, email, colegiatura, fecha_incorporacion
19 insert into ejercicios.alumnos (id, nombre, apellido, email, colegiatura, fecha_incorporacion
20 insert into ejercicios.alumnos (id, nombre, apellido, email, colegiatura, fecha_incorporacion
21 insert into ejercicios.alumnos (id, nombre, apellido, email, colegiatura, fecha_incorporacion
Data Output Messages Notifications
INSERT 0 1
Query returned successfully in 73 msec.

```

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.

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**.

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**.

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

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 qué 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 σ**) 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.

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
)
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**.

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 π**), 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.

	id_estacion	nombre_estacion
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 π**).
- La **fila** se señala con el comando **WHERE** (también llamado **Operador Unario de Selección o σ**), 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 opcional que se puede utilizar en conjunto con el comando **SELECT**, **FROM** o **JOIN**, la 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 ∩ 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 = $A \cup B - (A \cap B)$.

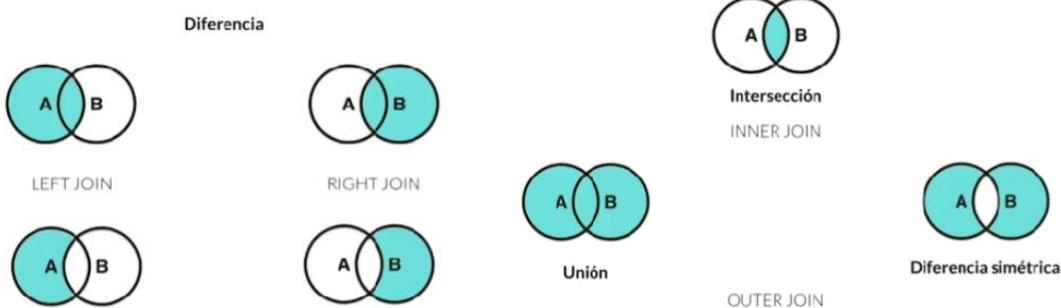
```

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

JOIN



- **WHERE:** Comando para indicar exactamente a cuáles **filas** de la **tabla** nos estamos refiriendo, filtrando así la consulta a través de cierta **condición matemática** ($=$, $>$, $<$, etc.), ya que la extracción se puede realizar de 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.
 - **IN:** Si se quiere agregar un **filtro en forma de Array estático**, se utiliza la instrucción **IN** seguida de sus valores entre paréntesis (**valorArray_1, ..., valorArray_n**) y con eso se podrán indicar **varios valores de filtro específicos**. Si se quiere obtener sus valores opuestos, se usa la instrucción **NOT IN**.
 - **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 Limite_Min_Rango AND Limite_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');

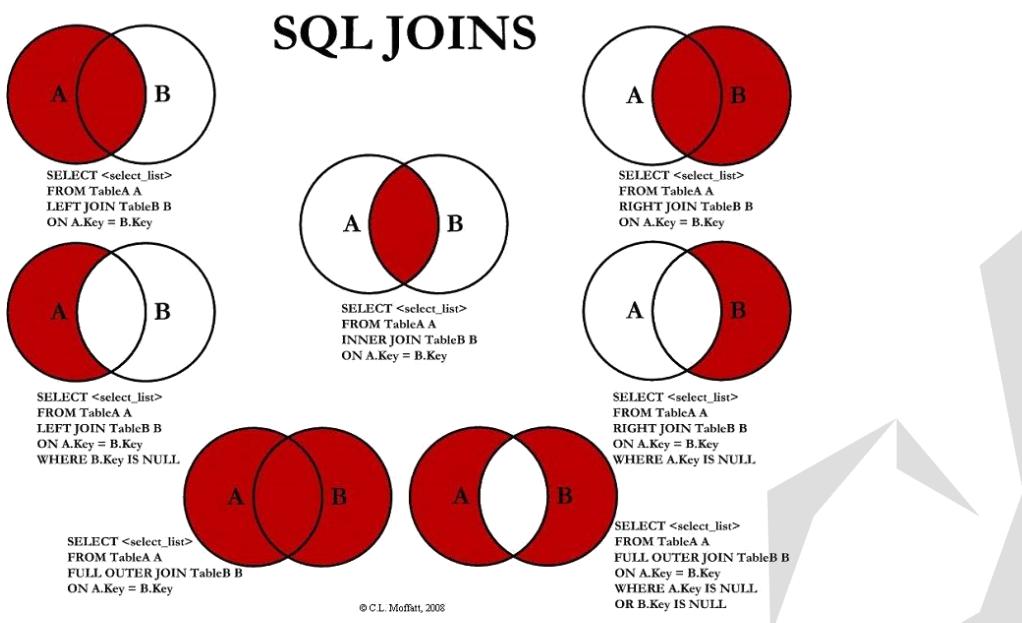
```

- 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 asociados a cada agrupación**, mostrando así en la tabla del resultado, **mínimo dos columnas**, en la primera se indica **el valor de la columna o columnas indicadas en el comando GROUP BY** y en la segunda se coloca el **resultado del cálculo realizado para cada valor de la primera columna**.
 - 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 su resultado, cuando se busca que este orden se ejecute de forma **ascendente (de menos a más viéndolos de arriba hacia abajo en función del 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:** Este comando sirve para limitar el número de filas que se van a obtener a través de una consulta. Aunque se suele utilizar después del comando **ORDER BY**, se puede usar 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 empiece a recabar la información del Query.

```

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 Math "Valor_Fila_Para_Filtro_Adicional"
GROUP BY    Nombre_Columna_1
HAVING     Nombre_Atributo_o_Columna Operación Math "Valor_Fila_Para_Filtro"
ORDER BY    Nombre_Atributo_o_Columna ASC_o_DESC
OFFSET     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;
```

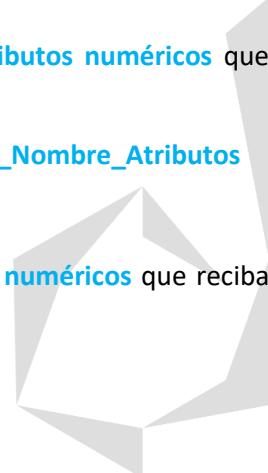
- **NULLIF:** El método se aplica individualmente a cada **atributo** de una **consulta o Query** y se usa para comprarrar los valores de dos **atributos numéricos** que reciba como parámetros y retornar un valor nulo (**NULL**) cuando estos tengan el mismo valor; de lo contrario, devuelve el valor de **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.) THEN
    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 a través de un paréntesis, esto es muy utilizado cuando por ejemplo, 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()**. 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. Su sintaxis puede ser la siguiente:

```
SELECT      Query_Anidado_1.Atributo_Anidado_1, COUNT(Columna)
FROM        (
    --Consulta (Query) anidado.

    SELECT      MIN(Atributo_1) AS Atributo_Anidado_1, COUNT(Columna_n)
    FROM        Nombre_Tabla_o_Entidad
    ...
) AS Query_Anidado_1
GROUP BY    Query_Anidado_1.Atributo_Anidado_1
HAVING     Query_Anidado_1.Columna_n Operación Lógica "Valor_Fila_Para_Filtro";
ORDER BY    Query_Anidado_1.Atributo_Anidado_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.

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:

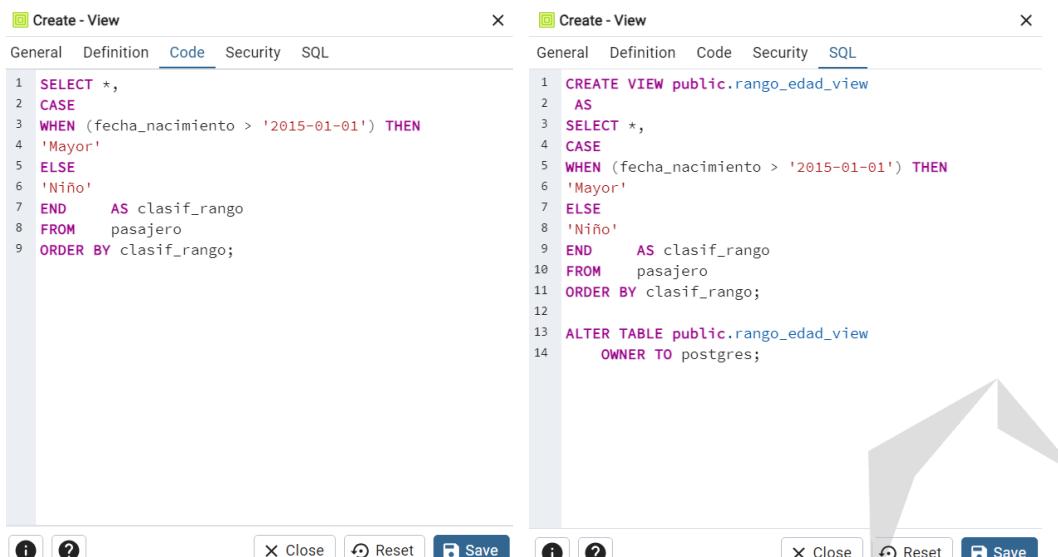
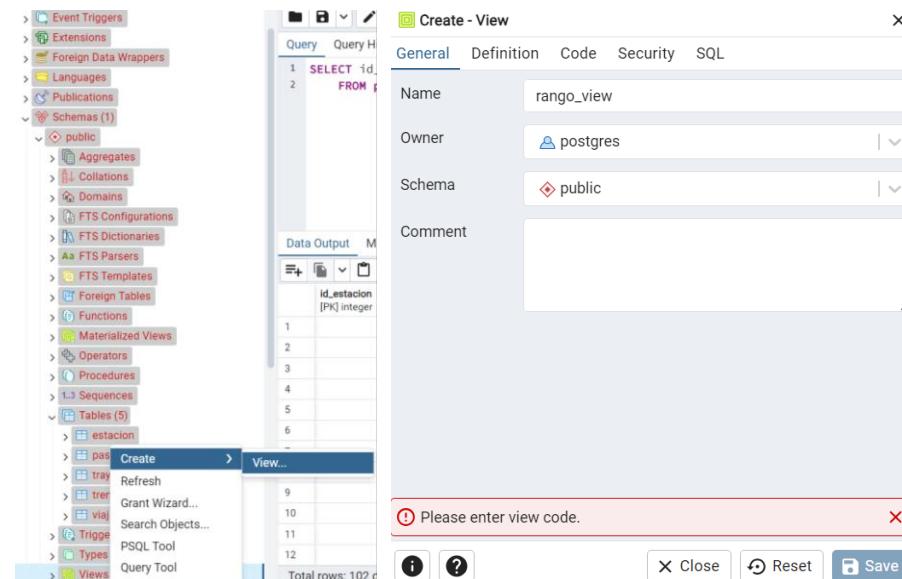
```
SELECT      Nombre_Columna_1 AS Nuevo_Nombre_Atributo_1, COUNT(Columna_n)
FROM        Nombre_Tabla_o_Entidad
WHERE       Nombre_Atributo_o_Columna Operación Lógica (
    --Consulta (Query) anidado.

    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.

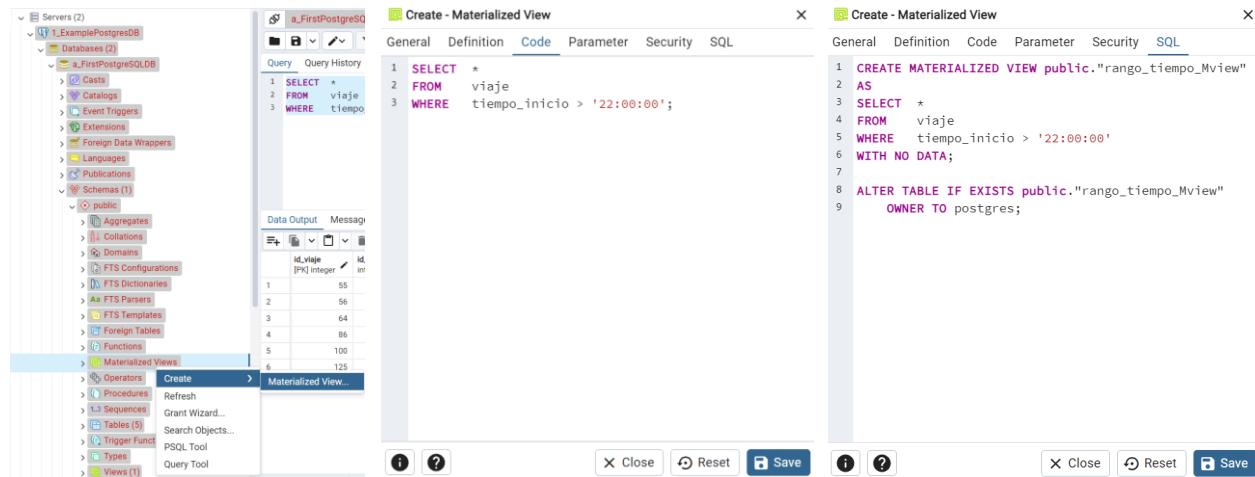
Si se quiere crear una **VISTA Volátil** dentro de **pgAdmin** se debe seleccionar la opción de: **Servers** → **Motor PostgreSQL** → **Nombre_Database** → **Schemas** → **public** → **Views** → Clic derecho → **Create** → **View...** → **Pestaña General** → **Name: Nombre de la VISTA Volátil** → **Pestaña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



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.



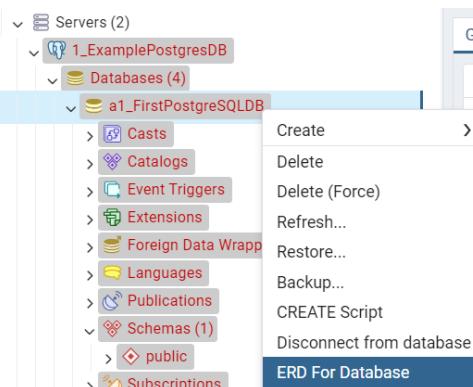
Finalmente, para ejecutar una vista se debe correr el siguiente código SQL dando clic en la opción de: **Servers** → **Motor PostgreSQL** → **Nombre_Database** → Schemas → public → **Materialized Views** → **Nombre de la VISTA Materializada** → Clic derecho → Scripts → **SELECT** Script → Execute script. 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 podré ver su contenido.

```
REFRESH MATERIALIZED VIEW public."v_Nombre_Vista_Materializada";
```

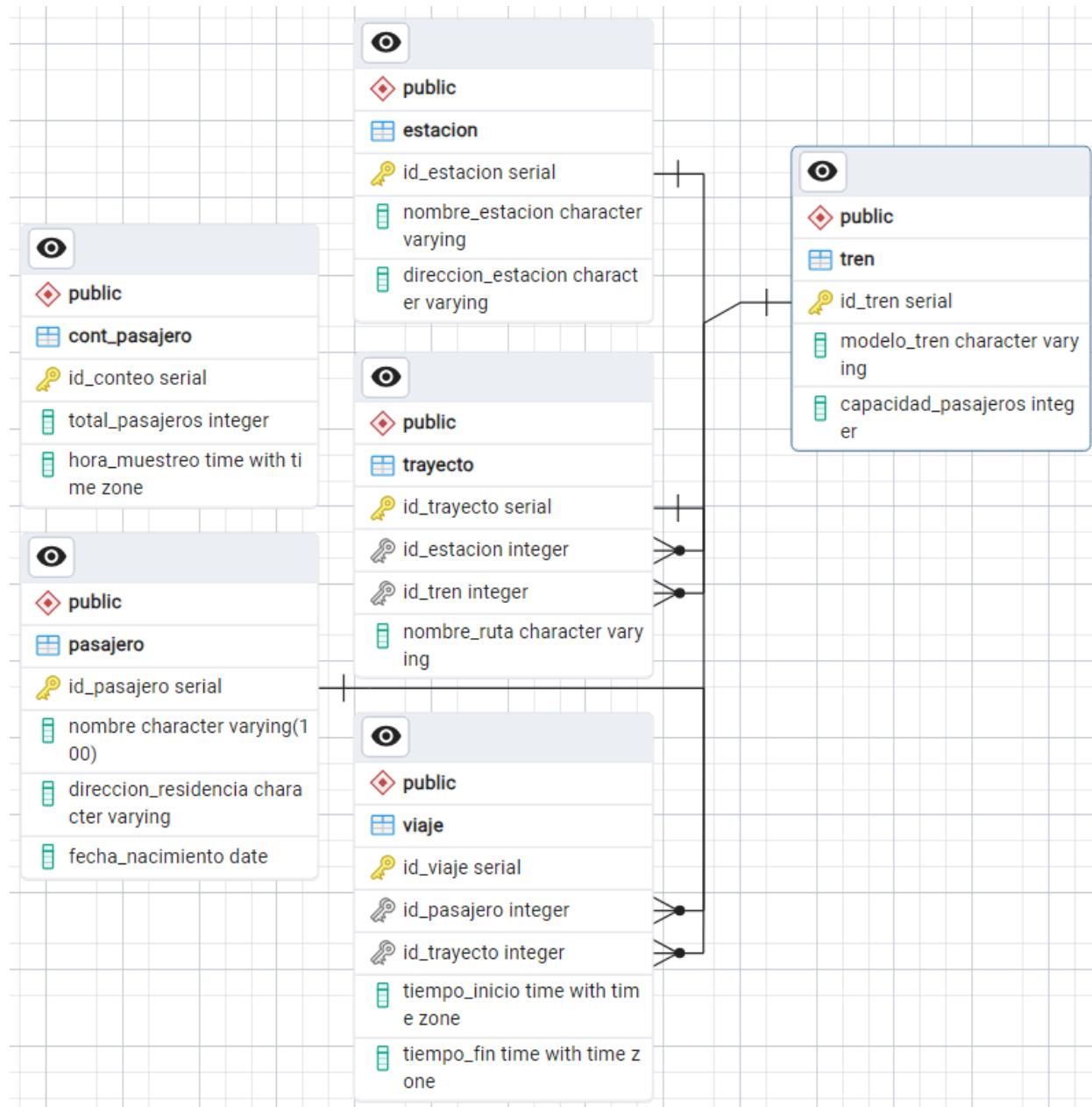
```
SELECT * FROM public."v_Nombre_Vista_Materializada";
```

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 Avanzados de SQL Queries

Cuando se tenga una **pregunta que se quiera responder a través de los datos almacenados en un database**, las siguientes partes del código SQL corresponden a cada parte de la pregunta:

- **SELECT**: Comando que indica las **columnas** de datos que se quiere extraer y mostrar.
- **FROM**: Comando que indica la **tabla** de donde se tomarán los datos.
- **WHERE**: Comando que indica los **filtros** de los datos que se quiere mostrar.
- **GROUP BY**: Indica las **clasificaciones** de las agrupaciones de datos que se quiere mostrar.

- **ORDER BY:** Indica el **orden** de las agrupaciones de datos que se quiere mostrar.
- **HAVING:** Indica los **filtros** que quiero que mis datos agrupados tengan.

A continuación, responderemos algunas preguntas acerca de la **base de datos relacional** del ejercicio a través de consultas SQL, la cual contiene las tablas de **alumnos** y **carreras**, se utilizarán varios códigos para resolver un mismo problema, pero vale la pena observar como cada código se tarda cierto tiempo en ser ejecutado, por lo que convendrá utilizar uno frente a otro. El tiempo de ejecución de las consultas SQL se puede observar en la esquina inferior derecha al correrlas y toda la documentación del código se encuentra en el archivo **a7.2.-Ejercicios Avanzados SQL Queries - Consultas.sql**.

1.- Obtención de primera fila: FETCH, NESTED QUERY y ROW NUMBER()

1. Realiza una consulta SQL que **obtenga siempre el primer registro o fila** de cualquier **tabla**, sin utilizar para ello su **columna id**. Para ello todos los siguientes códigos son equivalentes y en **pgAdmin** se ejecutan al seleccionar la opción de: Tools → Query Tool.

```

Query   Query History
1 SELECT *
2 FROM ejercicios.alumnos
3 FETCH FIRST 1 ROWS ONLY;
4
5 SELECT *
6 FROM ejercicios.alumnos
7 LIMIT 1;
8
9 SELECT *
10 FROM (
11     SELECT ROW_NUMBER() OVER() AS row_id, *
12     FROM ejercicios.alumnos
13 ) AS query_alumnos_anidado
14 WHERE row_id = 1;

Scratch Pad X

Data Output  Messages  Notifications
row_id | id | nombre | apellido | email | colegiatura | fecha_incorporacion | carrera_id | tutor_id
1      | 1  | Wanda  | Billington | wbillington0@nsw.gov.au | 5000 | 2020-05-17 22:49:29 | 23        | 9

```

2.- Segundo valor más alto: DISTINCT, COUNT(), LIMIT, OFFSET, etc.

2. Realiza una consulta SQL que **obtenga el registro o fila del atributo colegiatura perteneciente a la tabla alumnos que tenga el segundo valor más alto**. Para ello todos los siguientes códigos son equivalentes y en **pgAdmin** se ejecutan al seleccionar la opción de: Tools → Query Tool.
 - a. Ahora de esta misma tabla se presentará solo la segunda mitad de la tabla obtenida como resultado del Query.

```

Query   Query History
1 SELECT DISTINCT colegiatura
2 FROM ejercicios.alumnos AS a1
3 WHERE (
4     SELECT COUNT(DISTINCT colegiatura)
5     FROM ejercicios.alumnos AS a2
6     WHERE a1.colegiatura <= a2.colegiatura
7 ) = 2;
8
9 SELECT DISTINCT colegiatura
10 FROM ejercicios.alumnos
11 ORDER BY colegiatura DESC
12 LIMIT 1 OFFSET 1;

Scratch Pad X

Data Output  Messages  Notifications
colegiatura
double precision
1      | 4800

```

Query History

```

14 SELECT *
15 FROM ejercicios.alumnos AS datos_alumnos
16 INNER JOIN (
17     SELECT DISTINCT colegiatura
18     FROM ejercicios.alumnos
19     ORDER BY colegiatura DESC
20     LIMIT 1 OFFSET 1
21 ) AS segunda_mayor_colegiatura
22 ON datos_alumnos.colegiatura = segunda_mayor_colegiatura.colegiatura;
23
24 SELECT *
25 FROM ejercicios.alumnos AS datos_alumnos
26 WHERE colegiatura = (
27     SELECT DISTINCT colegiatura
28     FROM ejercicios.alumnos
29     ORDER BY colegiatura DESC
30     LIMIT 1 OFFSET 1
31 );

```

Data Output Messages Notifications

	id integer	nombre character varying (50)	apellido character varying (50)	email character varying (50)	colegiatura double precision	fecha_incorporacion timestamp without time zone	carrera_id integer	tutor_id integer	colegiatura double precision
1	5	Maire	Garnall	mgarnall4@artisteer.com	4800	2020-08-04 03:43:57	14	7	4800
2	16	Susy	Weedall	sweedallf@google.ru	4800	2019-09-03 11:22:10	15	2	4800
3	34	Denyse	Baldam	dbaldamx@oocn.ne.jp	4800	2019-03-10 15:08:58	32	2	4800
4	42	Jenelle	Askwith	jaskwith15@ebay.com	4800	2018-07-13 16:40:35	18	13	4800
5	55	Linette	Loeber	lloeber11@jathis.com	4800	2018-05-27 16:56:21	39	14	4800
6	90	Karina	Koschke	kkoschke2h@purevolume.com	4800	2019-04-05 02:13:27	22	21	4800
7	94	Hanni	Ewence	hewence2l@nhs.uk	4800	2018-03-09 20:02:23	12	6	4800
8	97	Sharleen	Sawdon	ssawdon2o@goo.ne.jp	4800	2019-03-15 03:10:43	46	1	4800
9	100	Tibold	Kumar	tkumar2r@pbs.org	4800	2018-10-05 10:58:33	13	24	4800
10	121	Selle	Stoffler	sstoffler3c@go.com	4800	2019-01-17 17:04:37	21	7	4800
11	127	Jori	Kopelman	jkopelman3i@cargocollective.com	4800	2018-04-10 01:17:43	5	3	4800
12	135	Julie	Medgwick	jmedgwick3q@state.tx.us	4800	2018-10-29 20	Successfully run. Total query runtime: 90 msec. 129 rows affected.		

Total rows: 129 of 129 Query complete 00:00:00.090

Scratch Pad

Query History

```

14 SELECT *
15 FROM ejercicios.alumnos AS datos_alumnos
16 INNER JOIN (
17     SELECT DISTINCT colegiatura
18     FROM ejercicios.alumnos
19     ORDER BY colegiatura DESC
20     LIMIT 1 OFFSET 1
21 ) AS segunda_mayor_colegiatura
22 ON datos_alumnos.colegiatura = segunda_mayor_colegiatura.colegiatura;
23
24 SELECT *
25 FROM ejercicios.alumnos AS datos_alumnos
26 WHERE colegiatura = (
27     SELECT DISTINCT colegiatura
28     FROM ejercicios.alumnos
29     ORDER BY colegiatura DESC
30     LIMIT 1 OFFSET 1
31 );

```

Data Output Messages Notifications

	id integer	nombre character varying (50)	apellido character varying (50)	email character varying (50)	colegiatura double precision	fecha_incorporacion timestamp without time zone	carrera_id integer	tutor_id integer	colegiatura double precision
1	5	Maire	Garnall	mgarnall4@artisteer.com	4800	2020-08-04 03:43:57	14	7	4800
2	16	Susy	Weedall	sweedallf@google.ru	4800	2019-09-03 11:22:10	15	2	4800
3	34	Denyse	Baldam	dbaldamx@oocn.ne.jp	4800	2019-03-10 15:08:58	32	2	4800
4	42	Jenelle	Askwith	jaskwith15@ebay.com	4800	2018-07-13 16:40:35	18	13	4800
5	55	Linette	Loeber	lloeber11@jathis.com	4800	2018-05-27 16:56:21	39	14	4800
6	90	Karina	Koschke	kkoschke2h@purevolume.com	4800	2019-04-05 02:13:27	22	21	4800
7	94	Hanni	Ewence	hewence2l@nhs.uk	4800	2018-03-09 20:02:23	12	6	4800
8	97	Sharleen	Sawdon	ssawdon2o@goo.ne.jp	4800	2019-03-15 03:10:43	46	1	4800
9	100	Tibold	Kumar	tkumar2r@pbs.org	4800	2018-10-05 10:58:33	13	24	4800
10	121	Selle	Stoffler	sstoffler3c@go.com	4800	2019-01-17 17:04:37	21	7	4800
11	127	Jori	Kopelman	jkopelman3i@cargocollective.com	4800	2018-04-10 01:17:43	5	3	4800
12	135	Julie	Medgwick	jmedgwick3q@state.tx.us	4800	2018-10-29 20	Successfully run. Total query runtime: 107 msec. 129 rows affected.		

Total rows: 129 of 129 Query complete 00:00:10.7

Scratch Pad

3.- Segunda mitad de una tabla: ROW_NUMBER() & COUNT()

3. Realiza una consulta SQL que **retorne solo la última mitad de las filas pertenecientes a la tabla alumnos**. Para ello en pgAdmin se selecciona la opción de: Tools → Query Tool.

Query Query History Execute script [FS]

```

1 SELECT ROW_NUMBER() OVER() AS row_id
2 FROM ejercicios.alumnos
3 OFFSET (
4   SELECT COUNT(*)/2
5   FROM ejercicios.alumnos
6 );

```

Data Output Messages Notifications

	row_id	id	nombre	apellido	email	colegiatura	fecha_incorporacion	carrera_id	tutor_id
	bigint	integer	character varying (50)	character varying (50)	character varying (50)	double precision	timestamp without time zone	integer	integer
1	501	501	Yance	[null]	yodamh@yellowbook.com	3000	2018-03-07 17:59:38	23	15
2	502	502	Claudius	Draye	cdrayed@youtube.com	3500	2018-08-25 21:38:15	13	14
3	503	503	Corinne	Marler	cmarler@census.gov	2500	2017-09-16 23:56:22	39	2
4	504	504	Geri	[null]	gffetreplace@wuffoo.com	2300	2018-11-02 09:16:18	1	8
5	505	505	Kelbee	Llewellyn	llewellyn@fastcompany.com	3000	2018-10-22 05:51:16	32	22
6	506	506	Carma	Kirwin	dkirwin@marshable.com	5000	2020-06-01 11:05:48	24	29
7	507	507	Tri	Demann	tdemann2@ftc.gov	4500	2019-12-13 05:21:39	42	26
8	508	508	Gweni	Durbridge	gturbridge@spiegel.de	4500	2020-03-05 15:53:54	30	26
9	509	509	Genia	Brugmann	gbnugmanne@oiaic.gov.au	2000	2020-02-24 05:15:37	41	11
10	510	510	Shoshanna	Collingdon	scollingdon5@merriam-webster.com	2500	2019-08-15 12:20:07	15	20
11	511	511	Fields	Baxter	fbaxter@swiley.com	4500	2018-07-20 06:19:38	40	2
12	512	512	Ourt	Kuhrt	ckuhrt@odnoklassniki.ru	2300	2019-04-24 20:05:39	13	6
13	513	513	Bally	Body	bbodye@technorati.com	2300	2018-08-01 22:19:52	20	14
14	514	514	Hall	Scallion	hscalione@google.de	3500	2020-06-26 11:50:44	50	25
15	515	515	Franciskus	Entres	fentres@drupal.org	2500	2020-01-19 06:14:44	10	9
16	516	516	Tabb	Chrispin	tchrispineb@discovery.com	2300	2017-12-27 21:56:40	43	12
17	517	517	Jasmin	[null]	jmarchesi@house.gov	2500	2019-07-01 00:02:35	8	8
18	518	518	Ambrose	Rozier	arozere@qq.com	2500	2019-08-10 04:48:49	36	30
19	519	519	Debbie	Mallindine	dmallindenee@1688.com	2300	2018-08-29 15:53:24	32	30
20	520	520	Edin	[null]	ecoaleef@yahoo.ru	5000	2020-07-31 20:58:10	28	24
21	521	521	Abram	Aronow	aaronweg@alexa.com	5000	✓ Successfully run. Total query runtime: 108 msec. 500 rows affected.		

Total rows: 500 of 500 Query complete 00:00:00.108 Ln 6, Col 3

4.- Consulta con Arrays: WHERE/IN

4. Realiza una consulta SQL que **filtre las filas de información de la tabla alumnos obtenidas a través de varios valores estáticos incluidos en un array**. Para ello en pgAdmin se selecciona la opción de: Tools → Query Tool.

Query Query History Execute script [FS]

```

1 SELECT *
2 FROM (
3   SELECT ROW_NUMBER() OVER() AS row_id, *
4   FROM ejercicios.alumnos
5 ) AS query_anidado_alumnos
6 WHERE row_id IN (1, 5, 10, 12, 15);

```

Data Output Messages Notifications

	row_id	id	nombre	apellido	email	colegiatura	fecha_incorporacion	carrera_id	tutor_id
	bigint	integer	character varying (50)	character varying (50)	character varying (50)	double precision	timestamp without time zone	integer	integer
1	1	1	Wanda	Billington	wbillington0@nsw.gov.au	5000	2020-05-17 22:49:29	23	9
2	5	5	Maire	Garnall	mgarnall4@artisteer.com	4800	2020-08-04 03:43:57	14	7
3	10	10	Garnet	Endrighi	gendrigh9@spiegel.de	2300	2019-03-28 06:13:15	41	29
4	12	12	Billi	Mattinson	bmattinsonh@abc.net.au	3000	2019-10-16 02:26:17	42	14
5	15	15	Hana	Vaughn	hvaughne@ted.com	2000	2017-10-14 17:30:19	7	6

Query Query History Execute script [FS]

```

1 SELECT *
2 FROM ejercicios.alumnos
3 WHERE id IN (
4   SELECT id
5   FROM ejercicios.alumnos
6   WHERE tutor_id = 30
7 );

```

Data Output Messages Notifications

	id	nombre	apellido	email	colegiatura	fecha_incorporacion	carrera_id	tutor_id
	integer	character varying (50)	character varying (50)	character varying (50)	double precision	timestamp without time zone	integer	integer
1	22	Ashen	[null]	aromei@bluehost.com	5000	2017-09-15 15:46:56	31	30
2	38	Neale	Mattiuzzi	rmatthiaz11@163.com	2300	2019-12-17 19:13:52	24	30
3	118	Tabby	Heinreich	theinreich39@hc360.com	3000	2018-05-24 14:32:13	3	30
4	146	Belva	Cundy	bcundy4@mlb.com	2000	2019-11-29 21:19:59	41	30
5	190	Brigid	Fripps	bfripps9@theforest.net	3500	2019-03-13 09:31:06	43	30
6	282	Cathiee	Prandi	oprandi7@yahoo.com	2000	2019-11-28 09:47:19	37	30
7	329	Yetie	Noad	ymoad4@samsung.com	4500	2017-11-15 02:04:31	47	30
8	343	Valentine	Fernao	vfernao@bing.com	5000	2018-10-16 14:17:27	2	30
9	397	Natalya	Bartoloma	nbartolomab@marketwatch.com	4500	2020-04-10 22:15:56	29	30
10	405	Dido	Veanscomb	dveanscomb6@example.com	2000	2019-04-05 18:40:50	33	30
11	413	Diane-marie	[null]	dstoboebsg@tiny.cc	4800	2019-04-06 22:38:14	43	30
12	427	Zach	Seally	zseallybu@github.io	2500	2019-10-17 04:59:27	1	30
13	443	Vittoria	Klett	vklett@ttmesonline.co.uk	3500	2019-12-27 14:53:22	2	30
14	516	Ambrose	Rozier	arozere@qq.com	2500	2019-08-10 02:48:49	36	30
15	519	Debbie	Mallindine	dmallindenee@1688.com	2300	2018-08-29 15:53:24	32	30
16	531	Stanwood	Urling	surlinge@odnoklassniki.ru	2300	2019-02-20 06:23:11	5	30
17	542	Ruby	[null]	rchiplin1@ustream.tv	2300	2017-11-15 14:46:09	3	30
18	621	Jess	Bawme	jbawmet@utoronto.com	2300	2017-12-02 23:17:05	34	30
19	626	Fleur	Casina	fcasina9@google.pl	2000	2018-07-11 07:47:16	9	30
20	690	Dre	Munn	dmunns5@paypal.com	2000	2017-09-14 21:25:54	✓ Successfully run. Total query runtime: 98 msec. 30 rows affected.	

Total rows: 30 of 30 Query complete 00:00:00.098 Ln 1, Col 1

5.- Consulta de Tiempos y Fechas: EXTRACT y DATE_PART()

5. Realiza una consulta SQL que utilice los comandos **EXTRACT** y **DATE_PART()** para extraer partes de una **columna** perteneciente a la tabla **alumnos** que contenga **datos tipo fecha (date)**, **hora (timestamp)** o **intervalo (interval)**. Para ello en pgAdmin se selecciona la opción de: Tools → Query Tool.

- a. Muchas veces dentro de las **bases de datos** las fechas y horas no se manejan con **tipos de dato date, datetime, time, timestamp, Interval, etc.** sino que se declaran como **strings**, pero esto es de malas prácticas, ya que **existen métodos específicos de SQL que permiten realizar operaciones de fechas y horas en Queries llamados EXTRACT, DATE_PART()**. Conviene utilizar una sobre otra al fijarnos en cuanto es su tiempo de ejecución en cada consulta específica.

Query History

Execute script (F5)

Scratch Pad

```
1 SELECT EXTRACT (YEAR FROM fecha_incorporacion) AS anio_incorporacion,
2        EXTRACT (HOUR FROM fecha_incorporacion) AS hora_incorporacion,
3        EXTRACT (MINUTE FROM fecha_incorporacion) AS minuto_incorporacion
4 FROM ejercicios.alumnos;
5
6 SELECT DATE_PART ('year', fecha_incorporacion) AS anio_incorporacion,
7        DATE_PART ('month', fecha_incorporacion) AS mes_incorporacion,
8        DATE_PART ('day', fecha_incorporacion) AS dia_incorporacion
9 FROM ejercicios.alumnos;
```

Data Output

Messages Notifications

	anio_incorporacion numeric	hora_incorporacion numeric	minuto_incorporacion numeric
1	2020	22	49
2	2018	17	51
3	2020	0	3
4	2018	17	28
5	2020	3	43
6	2019	7	45
7	2020	19	21
8	2020	14	46
9	2020	4	46
10	2019	6	13
11	2019	3	36
12	2019	2	26
13	2020	19	9
14	2019	18	26
15	2017	17	30
16	2019	11	22
17	2017	11	37
18	2018	0	58
19	2019	4	49

Total rows: 1000 of 1000. Query complete 00:00:00.094 ✓ Successfully run. Total query runtime: 94 msec. 1000 rows affected.

Query Query History Execute script (F5) Scratch Pad X

```
1 SELECT *  
2 FROM ejercicios.alumnos  
3 WHERE (  
4 EXTRACT (YEAR FROM fecha_incorporacion)  
5 ) = 2019;  
6  
7 SELECT *  
8 FROM ejercicios.alumnos  
9 WHERE (  
10 DATE_PART ('year', fecha_incorporacion)  
11 ) = 2019;  
12  
13 SELECT *  
14 FROM (  
15 SELECT *, DATE_PART ('year', fecha_incorporacion) AS anio_incorporacion  
16 FROM ejercicios.alumnos  
17 ) AS query_alumnos_conAnio_anulado  
18 WHERE anio_incorporacion = 2019;
```

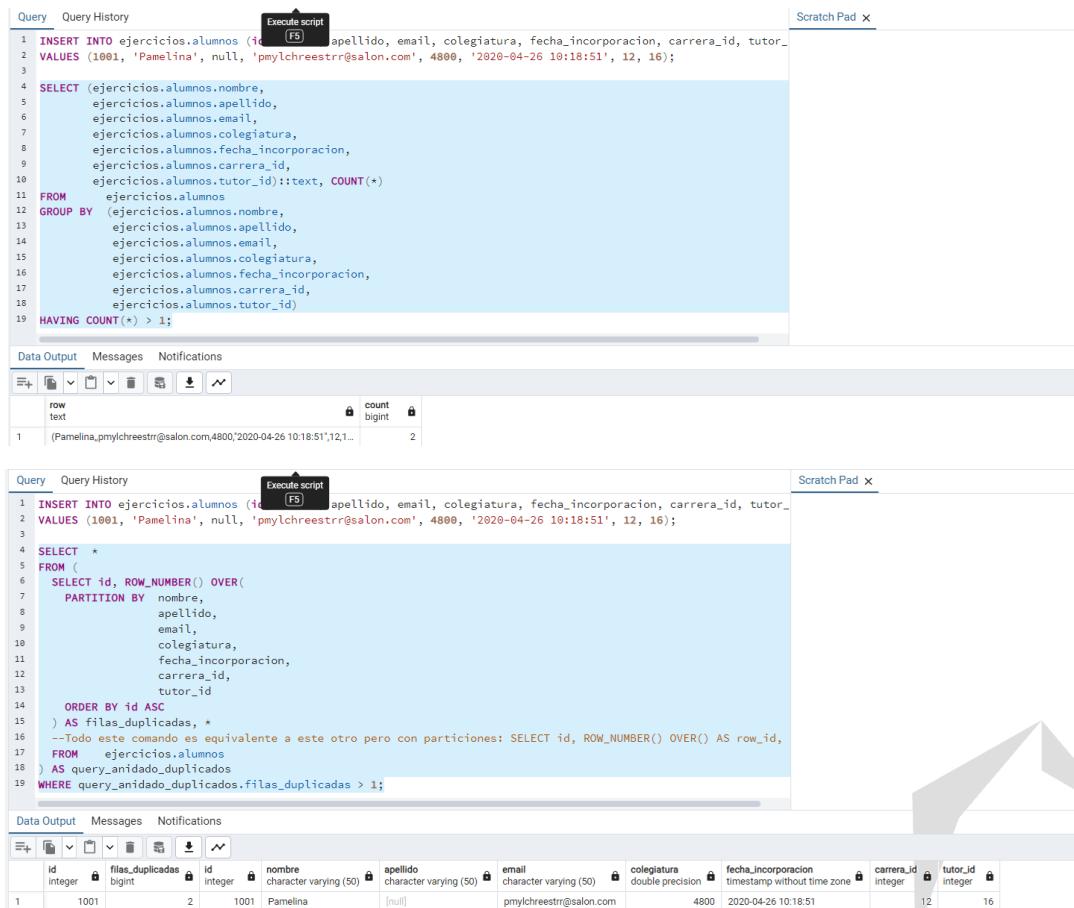
Data Output Messages Notifications

	id	nombre	apellido	email	colegiatura	fecha_incorporacion	carrera_id	tutor_id	anio_incorporacion
	integer	character varying (50)	character varying (50)	character varying (50)	double precision	timestamp without time zone	integer	integer	double precision
1	6	Nikolaus	Freeborn	nfreeborn5@yandex.ru	2000	2019-06-05 07:45:19	28	13	2019
2	10	Garnet	Endrighi	gendifrighi@spiegel.de	2300	2019-03-28 06:13:15	41	29	2019
3	11	Terri	Piercy	tpiercy@a8.net	5000	2019-05-31 03:36:39	31	17	2019
4	12	Billi	Mattinson	bmattinson@abc.net.au	3000	2019-10-16 02:26:17	42	14	2019
5	14	Saree	Trousdale	stroussdale@yale.edu	2500	2019-12-23 18:26:56	45	25	2019
6	16	Susy	Weedall	sweedall@google.ru	4800	2019-09-03 11:22:10	15	2	2019
7	19	Shir	Dionisotto	sdionisotto@google.com	5000	2019-05-31 04:49:29	35	24	2019
8	24	Veriee	Sealey	vsealey@dalymotion.com	3000	2019-03-29 03:06:25	2	3	2019
9	31	Sherlock	Manuely	smanuely@biglobe.ne.jp	4500	2019-07-13 04:06:05	18	2	2019
10	33	Barde	Turford	bturford@yale.edu	5000	2019-09-23 12:54:06	38	27	2019
11	34	Denyse	Baldam	dbaldamx@ccn.ne.jp	4800	2019-03-10 15	✓ Successfully run. Total query runtime: 95 msec. 336 rows affected.		

Total rows: 336 of 336 Query complete 00:00:00.095 Ln 13 Col 1

6.- Consulta de Duplicados: CAST (:) , WHERE/IN y DELETE

6. Realiza una consulta SQL que encuentre las filas duplicadas (que se repitan) en la tabla **alumnos** que tenga el segundo valor más alto. Para ello primero tendremos que insertar un nuevo dato para forzar que exista un registro duplicado. Los siguientes códigos son equivalentes y luego en pgAdmin se ejecutará la consulta del ejercicio al seleccionar la opción de: Tools → Query Tool.
- Cast:** Se refiere a la transformación de un tipo de dato en otro distinto y su sintaxis es la siguiente: **tabla.columna::nuevo_tipo_de_dato**
 - Aunque cabe mencionar que si se busca realizar una conversión de **varias columnas**, se pueden colocar todas dentro de un paréntesis y luego utilizar el símbolo de **Cast (:)** para transformarlas:
(tabla.columna_1,..., tabla.columna_n)::nuevo_tipo_de_dato
 - Si se quiere encontrar duplicados en las filas de una tabla, se debe evitar incluir en los campos de análisis la columna **id**, ya que esta siempre será diferente, por lo que conviene transformar a través de un Cast todos los valores de las columnas en un string para poderlo comparar, para ello se deben añadir las columnas una por una al comando **SELECT**, sin incluir el atributo **id**.
 - Window Functions:** También se pueden utilizar este tipo de comandos para encontrar duplicados y es hasta más conveniente porque se pueden observar todos sus datos.



```
Query 1: SELECT ejercicios.alumnos.nombre, ejercicios.alumnos.apellido, ejercicios.alumnos.email, ejercicios.alumnos.colegiatura, ejercicios.alumnos.fecha_incorporacion, ejercicios.alumnos.carrera_id, ejercicios.alumnos.tutor_id::text, COUNT(*) FROM ejercicios.alumnos GROUP BY (ejercicios.alumnos.nombre, ejercicios.alumnos.apellido, ejercicios.alumnos.email, ejercicios.alumnos.colegiatura, ejercicios.alumnos.fecha_incorporacion, ejercicios.alumnos.carrera_id, ejercicios.alumnos.tutor_id) HAVING COUNT(*) > 1;
```

```
Query 2: INSERT INTO ejercicios.alumnos (id, apellido, email, colegiatura, fecha_incorporacion, carrera_id, tutor_id) VALUES (1001, 'Pamelina', null, 'pmylchreestrr@salon.com', 4800, '2020-04-26 10:18:51', 12, 16);
```

```
Query 2: SELECT * FROM ( SELECT id, ROW_NUMBER() OVER( PARTITION BY nombre, apellido, email, colegiatura, fecha_incorporacion, carrera_id, tutor_id ORDER BY id ASC ) AS filas_duplicadas, * --Todo este comando es equivalente a este otro pero con particiones: SELECT id, ROW_NUMBER() OVER() AS row_id, FROM ejercicios.alumnos ) AS query_anidado_duplicados WHERE query_anidado_duplicados.filas_duplicadas > 1;
```

row_text	count bigint
1 (Pamelina,pmylchreestrr@salon.com,4800,'2020-04-26 10:18:51',12,1...	2

Id integer	filas_duplicadas bigint	id integer	nombre character varying(50)	apellido character varying(50)	email character varying(50)	colegiatura double precision	fecha_incorporacion timestamp without time zone	carrera_id integer	tutor_id integer
1001	2	1001	Pamelina	[null]	pmylchreestrr@salon.com	4800	2020-04-26 10:18:51	12	16

The screenshot shows two separate sessions in pgAdmin's Query History tab.

```

Query   Query History
1 INSERT INTO ejercicios.alumnos (id, nombre, apellido, email, colegiatura, fecha_incorporacion, carrera_id, tutor_
2 VALUES (1001, 'Pamelina', null, 'pmylchreestrr@salon.com', 4800, '2020-04-26 10:18:51', 12, 16);
3
4 DELETE FROM ejercicios.alumnos
5 WHERE id IN (
6   SELECT id
7   FROM (
8     SELECT ROW_NUMBER() OVER(
9       PARTITION BY nombre,
10      apellido,
11      email,
12      colegiatura,
13      fecha_incorporacion,
14      carrera_id,
15      tutor_id
16     ORDER BY id ASC
17   ) AS filas_duplicadas, *
18   --Todo este comando es equivalente a este otro pero con particiones: SELECT ROW_NUMBER() OVER() AS row_id, *
19   FROM ejercicios.alumnos
20 ) AS query_anidado_duplicados
21 WHERE query_anidado_duplicados.filas_duplicadas > 1
22 );

```

Data Output

DELETE 1

Query returned successfully in 58 msec.


```

Query   Query History
1 INSERT INTO ejercicios.alumnos (id, nombre, apellido, email, colegiatura, fecha_incorporacion, carrera_id, tutor_
2 VALUES (1001, 'Pamelina', null, 'pmylchreestrr@salon.com', 4800, '2020-04-26 10:18:51', 12, 16);
3
4 SELECT id
5 FROM (
6   SELECT ROW_NUMBER() OVER(
7     PARTITION BY nombre,
8     apellido,
9     email,
10    colegiatura,
11    fecha_incorporacion,
12    carrera_id,
13    tutor_id
14   ORDER BY id ASC
15 ) AS filas_duplicadas, *
16   --Todo este comando es equivalente a este otro pero con particiones: SELECT id, ROW_NUMBER() OVER() AS row_id,
17   FROM ejercicios.alumnos
18 ) AS query_anidado_duplicados
19 WHERE query_anidado_duplicados.filas_duplicadas > 1;

```

Data Output

7.- Selectores de Intervalos: Tipos de dato Range

7. Realiza una consulta SQL que **filtre las filas de información de la tabla alumnos obtenidas a través de varios valores estáticos**, pero estos **no deben estar incluidos en un array**, sino en un **valor de tipo range**. Para ello en pgAdmin se selecciona la opción de: Tools → Query Tool.
 - a. Recordemos que las consultas que obtienen **filas de información de una tabla a través de varios valores estáticos incluidos en un array** se ejecutan a través de los comandos **WHERE e IN**, pero esto se puede manejar también con **tipos de dato Range**, los cuales representan los siguientes intervalos de valores continuos:
 - i. **int4range**: Representa un rango de enteros de 4 bytes (0 a 65,000).
 - ii. **int8range**: Representa un rango de enteros de 8 bytes (0 a 4 millones).
 - iii. **numrange**: Representa un rango de números decimales (1.5 a 10.5).
 - iv. **tsrange**: Representa un rango de fecha/hora **sin zona horaria (timestamp)**.
 - v. **tstzrange**: Rango de fecha/hora **con zona horaria (timestamp with time zone)**.
 - vi. **daterange**: Representa un rango de fechas (**date**).

- b. Cuando se utilicen **tipos de dato range** en consultas SQL, usualmente se acompañan de los siguientes operadores para realizar acciones con ellos:
- @> (Contiene)**: Operación que verifica si un **rango** contiene otro **conjunto de valores estáticos** o un **valor singular** en específico, devolviendo un booleano True si existe el valor o False si no existe.
 - * (Intersección)**: Usualmente este operador **multiplica dos valores numéricos**, pero cuando se aplica a **tipos de dato range**, **obtiene su intersección** y devuelve los datos donde esto ocurre.
 - && (Intersección)**: Verifica si dos **rangos, arrays, o conjuntos** se superponen o **interseccionan**, devolviendo un booleano True si existe superposición o False si no existe.
 - UPPER()**: Convierte una cadena de texto a mayúsculas o **devuelve el valor máximo** de un **tipo de dato range**.
 - LOWER()**: Convierte una cadena de texto a minúsculas o **devuelve el valor mínimo** de un **tipo de dato range**.
 - ISEMPTY()**: Verifica si un **tipo de dato range** está vacío, devolviendo un booleano True si no existe ningún valor o False si existe alguno.

Primero se obtendrán los rangos utilizando el comando **WHERE/IN (Arrays)**, el comando **AND** y el comando **BETWEEN/AND** y luego se analizará la alternativa de realizar la misma operación con **tipos de dato Range**. Aunque antes de aplicarlos en consultas, se realizarán ejemplos sencillos del uso de sus operadores.

```

Query   Query History          Execute script (F5)   Scratch Pad x
1 --Consultas con obtención de Intervalos
2 SELECT *
3 FROM ejercicios.alumnos
4 WHERE tutor_id IN (1,2,3,4,5,6,7,8,9,10);
5
6 SELECT *
7 FROM ejercicios.alumnos
8 WHERE tutor_id >= 1
9     AND tutor_id <= 10;
10
11 SELECT *
12 FROM ejercicios.alumnos
13 WHERE tutor_id BETWEEN 1 AND 10;

```

Data Output Messages Notifications

	id integer	nombre character varying (50)	apellido character varying (50)	email character varying (50)	colegiatura double precision	fecha_incorporacion timestamp without time zone	carrera_id integer	tutor_id integer
1	1	Wanda	Billington	wbillington0@nsw.gov.au	5000	2020-05-17 22:49:29	23	9
2	5	Maire	Garnall	mgarnall4@artisteer.com	4800	2020-08-04 03:43:57	14	7
3	15	Hana	Vaughn	hvaughne@ted.com	2000	2017-10-14 17:30:19	7	6
4	16	Susy	Weedall	sweedallf@google.ru	4800	2019-09-03 11:22:10	15	2
5	20	Adrienne	Thompson	athompsonj@comcast.net	3500	2020-07-15 02:53:20	36	8
6	21	Axel	Shade	ashadek@google.fr	2000	2018-09-05 09:05:09	28	4
7	24	Veriee	Sealey	vsealeyn@animotion.com	3000	2019-03-29 03:06:25	2	3
8	26	Sheryl	Sherill	ssherillp@cisco.com	5000	2018-08-07 23:00:58	43	8
9	28	Madeleine	Dunphie	mdunphier@cloudflare.com	2500	2020-02-22 22:24:48	45	2
10	30	Graham	Caustic	gcaustict@eventbrite.com	2000	2018-04-21 23:02:04	18	4
11	31	Sherlock	Manuely	smanuelyu@biglobe.ne.jp	4500	2019-07-13 04:06:05	18	2
12	34	Denyse	Baldam	dbaldamx@ocn.ne.jp	4800	2019-03-10 15:08:58	32	2
13	36	Wren	Gable	wgablez@sogou.com	5000	2018-07-10 19:16:32	9	3
14	49	Griffin	Tubble	gtubble1c@google.com	2500	2018-08-12 02:22:51	40	6
15	50	Vincent	Blizard	vblizard1@phpb.com	3000	2018-05-12 03:28:38	3	1
16	56	Dew	Matyas	dmatyas1@latimes.com	2300	2020-02-23 11:18:32	1	1

Total rows: 316 of 316 Query complete 00:00:00.118 Successfully run. Total query runtime: 118 msec. 316 rows affected. Ln 2, Col 1

Query Query History Execute script [F5] Scratch Pad x

```

1 --Consultas con obtención de Intel
2 SELECT *
3 FROM ejercicios.alumnos
4 WHERE tutor_id IN (1,2,3,4,5,6,7,8,9,10);
5
6 SELECT *
7 FROM ejercicios.alumnos
8 WHERE tutor_id >= 1
9 AND tutor_id <= 10;
10
11 SELECT *
12 FROM ejercicios.alumnos
13 WHERE tutor_id BETWEEN 1 AND 10;

```

Data Output Messages Notifications

	id integer	nombre character varying (50)	apellido character varying (50)	email character varying (50)	colegiatura double precision	fecha_incorporacion timestamp without time zone	carrera_id integer	tutor_id integer
1	1	Wanda	Billington	wbillington@nsw.gov.au	5000	2020-05-17 22:49:29	23	9
2	5	Maire	Garnall	mgarnall4@artisteer.com	4800	2020-08-04 03:43:57	14	7
3	15	Hana	Vaughn	hvaughe@ted.com	2000	2017-10-14 17:30:19	7	6
4	16	Susy	Weedall	sweedallf@google.ru	4800	2019-09-03 11:22:10	15	2
5	20	Adrienne	Thompson	athompsonj@comcast.net	3500	2020-07-15 02:53:20	36	8
6	21	Axel	Shade	ashadek@google.fr	2000	2018-09-05 09:05:09	28	4
7	24	Verilee	Sealey	vsealeyn@dailymotion.com	3000	2019-03-29 03:06:25	2	3
8	26	Sheryl	Shergill	sshergillp@cisco.com	5000	2018-08-07 23:00:58	43	8
9	28	Madeleine	Dunphie	mdunphier@cloudflare.com	2500	2020-02-22 22:24:48	45	2
10	30	Graham	Caustic	gcaustict@eventbrite.com	2000	2018-04-21 23:02:04	18	4
11	31	Sherlock	Manuely	smanuelyu@biglobe.ne.jp	4500	2019-07-13 04:06:05	18	2
12	34	Denyse	Baldam	dbaldamx@ocn.ne.jp	4800	2019-03-10 15:08:58	32	2
13	36	Wren	Gable	wgablez@sogou.com	5000	2018-07-10 19:16:32	9	3
14	49	Griffin	Tubble	gtubble1c@go.com	2500	2018-08-12 02:22:51	40	6
15	50	Vincent	Blizard	vblizard1d@phpb.com	3000	2018-05-12 03:28:38	3	1
16	56	Dew	Matyas	dmatyas1j@latimes.com	2300	2020-02-23 19:		

Total rows: 316 of 316 Query complete 00:00:00.084 L 1 n 6, Col 1

Query Query History Execute script [F5] Scratch Pad x

```

1 --Consultas con obtención de Intel
2 SELECT *
3 FROM ejercicios.alumnos
4 WHERE tutor_id IN (1,2,3,4,5,6,7,8,9,10);
5
6 SELECT *
7 FROM ejercicios.alumnos
8 WHERE tutor_id >= 1
9 AND tutor_id <= 10;
10
11 SELECT *
12 FROM ejercicios.alumnos
13 WHERE tutor_id BETWEEN 1 AND 10;

```

Data Output Messages Notifications

	id integer	nombre character varying (50)	apellido character varying (50)	email character varying (50)	colegiatura double precision	fecha_incorporacion timestamp without time zone	carrera_id integer	tutor_id integer
1	1	Wanda	Billington	wbillington@nsw.gov.au	5000	2020-05-17 22:49:29	23	9
2	5	Maire	Garnall	mgarnall4@artisteer.com	4800	2020-08-04 03:43:57	14	7
3	15	Hana	Vaughn	hvaughe@ted.com	2000	2017-10-14 17:30:19	7	6
4	16	Susy	Weedall	sweedallf@google.ru	4800	2019-09-03 11:22:10	15	2
5	20	Adrienne	Thompson	athompsonj@comcast.net	3500	2020-07-15 02:53:20	36	8
6	21	Axel	Shade	ashadek@google.fr	2000	2018-09-05 09:05:09	28	4
7	24	Verilee	Sealey	vsealeyn@dailymotion.com	3000	2019-03-29 03:06:25	2	3
8	26	Sheryl	Shergill	sshergillp@cisco.com	5000	2018-08-07 23:00:58	43	8
9	28	Madeleine	Dunphie	mdunphier@cloudflare.com	2500	2020-02-22 22:24:48	45	2
10	30	Graham	Caustic	gcaustict@eventbrite.com	2000	2018-04-21 23:02:04	18	4
11	31	Sherlock	Manuely	smanuelyu@biglobe.ne.jp	4500	2019-07-13 04:06:05	18	2
12	34	Denyse	Baldam	dbaldamx@ocn.ne.jp	4800	2019-03-10 15:08:58	32	2
13	36	Wren	Gable	wgablez@sogou.com	5000	2018-07-10 19:16:32	9	3
14	49	Griffin	Tubble	gtubble1c@go.com	2500	2018-08-12 02:22:51	40	6
15	50	Vincent	Blizard	vblizard1d@phpb.com	3000	2018-05-12 03:28:38	3	1
16	56	Dew	Matyas	dmatyas1j@latimes.com	2300	2020-02-23 19:		

Total rows: 316 of 316 Query complete 00:00:00.085 L 11, Col 1

Query Query History Execute script [F5] Scratch Pad x

```

14 --Tipos de dato range:
15 SELECT int4range(10,20) @> 3;
16

```

Data Output Messages Notifications

	?column?	boolean
1	false	

Query Query History Execute script [F5] Scratch Pad x

```

14 --Tipos de dato range:
15 SELECT int4range(10,20) @> 3;
16
17 SELECT int8range(10,20) * int8range(15,20);
18
19 SELECT numrange(11.1,22.2) && numrange(20.0,30.0);
20
21 SELECT UPPER(numrange(12.8,30.5));
22
23 SELECT ISEMPTY(numrange(12.8,30.5));

```

Data Output Messages Notifications

?column?	intRange
1	[15,20]

Query Query History Execute script [F5] Scratch Pad x

```

14 --Tipos de dato range:
15 SELECT int4range(10,20) @> 3;
16
17 SELECT int8range(10,20) * int8range(15,20);
18
19 SELECT numrange(11.1,22.2) && numrange(20.0,30.0);
20
21 SELECT UPPER(numrange(12.8,30.5));
22
23 SELECT ISEMPTY(numrange(12.8,30.5));

```

Data Output Messages Notifications

?column?	boolean
1	true

Query Query History Execute script [F5] Scratch Pad x

```

14 --Tipos de dato range:
15 SELECT int4range(10,20) @> 3;
16
17 SELECT int8range(10,20) * int8range(15,20);
18
19 SELECT numrange(11.1,22.2) && numrange(20.0,30.0);
20
21 SELECT UPPER(numrange(12.8,30.5));
22
23 SELECT ISEMPTY(numrange(12.8,30.5));

```

Data Output Messages Notifications

upper	numeric
1	30.5

Query Query History Execute script [F5] Scratch Pad x

```

14 --Tipos de dato range:
15 SELECT int4range(10,20) @> 3;
16
17 SELECT int8range(10,20) * int8range(15,20);
18
19 SELECT numrange(11.1,22.2) && numrange(20.0,30.0);
20
21 SELECT UPPER(numrange(12.8,30.5));
22
23 SELECT ISEMPTY(numrange(12.8,30.5));

```

Data Output Messages Notifications

isempty	boolean
1	false

Query History

```
25 SELECT * FROM ejercicios.alumnos
26 WHERE int4range(0,10) @> tutor_id;
```

Data Output

	id integer	nombre character varying (50)	apellido character varying (50)	email character varying (50)	colegiatura double precision	fecha_incorporacion timestamp without time zone	carrera_id integer	tutor_id integer
1	1	Wanda	Billington	wbillington@nsw.gov.au	5000	2020-05-17 22:49:29	23	9
2	5	Maire	Garnall	mgarnall4@artisteer.com	4800	2020-08-04 03:43:57	14	7
3	15	Hana	Vaughn	hvaughne@ted.com	2000	2017-10-14 17:30:19	7	6
4	16	Susy	Weedall	sweedall@google.ru	4800	2019-09-03 11:22:10	15	2
5	20	Adrienne	Thompson	athomponj@comcast.net	3500	2020-07-15 02:53:20	36	8
6	21	Axel	Shade	ashadek@google.fr	2000	2018-09-05 09:05:09	28	4
7	24	Verree	Sealey	vsealeyn@dailymotion.com	3000	2019-03-29 03:06:25	2	3
8	26	Sheryl	Shergill	sshergillp@cisco.com	5000	2018-08-07 23:00:58	43	8
9	28	Madeleine	Dunphie	mdunphier@cloudflare.com	2500	2020-02-22 22:24:48	45	2
10	30	Graham	Caustic	gcaustict@eventbrite.com	2000	2018-04-21 23:02:04	18	4
11	31	Sherlock	Manuely	smanuelyu@biglobe.ne.jp	4500	2019-07-13 04:06:05	18	2
12	34	Denyse	Baldam	dbaldamx@ocn.ne.jp	4800	2019-03-10 15:08:58	32	2
13	36	Wren	Gable	wgablez@sogou.com	5000	2018-07-10 19:16:32	9	3
14	49	Griffin	Tubble	gtubble1c@go.com	2500	2018-08-12 02:22:51	40	6
15	50	Vincent	Blizard	vblizard1d@phpbb.com	3000	2018-05-12 03:28:38	3	1
16	56	Dew	Matyas	dmatyas1j@latimes.com	2300	2020-02-23 19:32:39	39	8
17	57	Lane	Antley	lantley1k@tamu.edu	5000	2018-02-24 22:37:18	9	7
18	58	Rasia	Rambley	rrambley1l@sbwire.com	4500	2018-04-20 02:27:10	46	6
19	61	Cacilie	[null]	cmenicomb1o@earthlink.net	3500	2019-05-20 09:38:22	8	6
20	63	Sydney	Dener	sdener1q@godaddy.com	2300	2019-09-21 04:29:02	34	6
21	66	Catarina	[null]	cblaase1t@blogs.com	2000	2019-09-22 14:48:00	5	9
22	76	Rinaldo	Planque	rplanque23@upenn.edu	3000	2017-12-09 04:17:53	50	3
23	78	Bonnie	Karpf	bkarpf25@usatoday.com	3000	2019-08-14 03:01:38	18	2
24	80	Brocky	Roels	broels27@huffingtonpost.com	4500	2019-10-07 03:		

Total rows: 289 of 289 Query complete 00:00:00.062 Successfully run. Total query runtime: 62 msec. 289 rows affected. Ln 25, Col 1

8.- Subconsultas con Intervalos: Tipos de dato Range, MIN() y MAX()

8. Realiza una consulta SQL que encuentre el rango numérico de filas donde se intersectan las columnas **tutor_id** y **carrera_id** pertenecientes a la tabla **alumnos**, obtenidas a través de valores tipo range. Para ello en pgAdmin se selecciona la opción de: Tools → Query Tool.

Query History

```
1 SELECT numrange(
2   (SELECT MIN(tutor_id) FROM ejercicios.alumnos),
3   (SELECT MAX(tutor_id) FROM ejercicios.alumnos)
4 ) * numrange(
5   (SELECT MIN(carrera_id) FROM ejercicios.alumnos),
6   (SELECT MAX(carrera_id) FROM ejercicios.alumnos)
7 );
```

Data Output

	?column?	numrange
1		[1,30)

9.- Consulta de 2 o más Máximos y Mínimos: GROUP BY, MAX() y MIN()

9. Realiza una consulta SQL que obtenga y acomode sus filas a través de los valores máximos de 2 columnas con valores de tipos diferentes **carrera_id** (tipo de dato numérico) y **fecha_incorporacion** (tipo de dato date) pertenecientes a la tabla **alumnos**. Para ello en pgAdmin se selecciona la opción de: Tools → Query Tool.

- Para acomodar los resultados de una consulta en función de los valores mínimos o máximos de alguna de sus columnas, usualmente se utilizan los comandos **GROUP BY** y **ORDER BY**. Pero cuando no solamente se quiere agrupar sus resultados a través de los valores mínimos o máximos de una sola una columna, sino de varias, se debe

además hacer uso de los comandos **MAX()** y **MIN()** donde se esté declarando el comando **SELECT** en el código SQL.

Query Query History Execute script (F5) Scratch Pad X

```

1 --No se puede hacer esto: Porque se obtendrá el máximo de toda la tabla y se asignará a cada columna.
2 SELECT carrera_id, fecha_incorporacion
3 FROM ejercicios.alumnos
4 GROUP BY carrera_id, fecha_incorporacion
5 ORDER BY carrera_id DESC;
6 --Se debe hacer esto: Para que se obtenga el máximo de cada agrupación (cada fila).
7 SELECT carrera_id, MAX(fecha_incorporacion)
8 FROM ejercicios.alumnos
9 GROUP BY carrera_id
10 ORDER BY carrera_id DESC;
```

Data Output Messages Notifications

carrera_id	fecha_incorporacion
1	50 2017-09-14 21:25:54
2	50 2017-11-01 22:00:06
3	50 2017-11-29 22:57:41
4	50 2017-12-09 04:17:53
5	50 2017-12-26 03:30:45
6	50 2018-05-27 14:42:28
7	50 2018-07-15 21:26:05
8	50 2018-12-09 16:22:16
9	50 2019-03-18 23:01:53
10	50 2019-04-23 11:04:13
11	50 2019-05-17 19:44:49
12	50 2019-06-01 01:14:13
13	50 2019-07-19 00:47:51
14	50 2019-08-17 23:25:41
15	50 2019-08-29 04:49:50
16	50 2019-08-31 21:23:55
17	50 2020-02-25 00:34:21
18	50 2020-05-24 17:45:49

✓ Successfully run. Total query runtime: 90 msec. 1000 rows affected. X

Total rows: 1000 of 1000 Query complete 00:00:00.090 Ln 2, Col 1

Query Query History Scratch Pad X

```

1 --No se puede hacer esto: Porque se obtendrá el máximo de toda la tabla y se asignará a cada columna.
2 SELECT carrera_id, fecha_incorporacion
3 FROM ejercicios.alumnos
4 GROUP BY carrera_id, fecha_incorporacion
5 ORDER BY carrera_id DESC;
6 --Se debe hacer esto: Para que se obtenga el máximo de cada agrupación (cada fila).
7 SELECT carrera_id, MAX(fecha_incorporacion)
8 FROM ejercicios.alumnos
9 GROUP BY carrera_id
10 ORDER BY carrera_id DESC;
```

Data Output Messages Notifications

carrera_id	max
1	50 2020-06-26 11:50:44
2	49 2020-07-19 13:39:49
3	48 2020-08-05 22:32:45
4	47 2020-08-29 07:55:43
5	46 2020-06-29 23:21:58
6	45 2020-02-22 22:24:48
7	44 2020-08-09 17:53:03
8	43 2020-08-26 05:07:30
9	42 2020-03-21 22:56:07
10	41 2020-08-13 22:10:59
11	40 2020-08-27 05:24:18
12	39 2020-08-19 06:56:04
13	38 2020-05-09 01:48:48
14	37 2020-08-01 07:33:12
15	36 2020-08-22 05:44:12
16	35 2020-08-26 03:21:09
17	34 2020-06-27 11:07:18
18	33 2020-07-29 02:48:59

✓ Successfully run. Total query runtime: 75 msec. 50 rows affected. X

Total rows: 50 of 50 Query complete 00:00:00.075 Ln 7, Col 1

10.- Datos en Orden Alfabético y Otra Columna: GROUP BY, MAX() y MIN()

10. Realiza una consulta SQL que **acomode sus filas** en orden alfabético y numéricamente a través de las 2 columnas con valores de tipos diferentes **tutor_id** (tipo de dato numérico) y **nombre** (tipo de dato string) pertenecientes a la tabla **alumnos**. Para ello en pgAdmin se selecciona la opción de: Tools → Query Tool.

a. De igual forma que en el ejercicio anterior, para acomodar los resultados de una consulta en función de los **valores mínimos o máximos** de alguna de sus **columnas**, usualmente se utilizan los comandos **GROUP BY** y **ORDER BY**. Pero cuando **no solamente se quiere agrupar sus resultados a través de los valores mínimos o máximos de una sola una columna, sino de varias**, se debe además hacer uso de los comandos **MAX()** y **MIN()** donde se esté declarando el comando **SELECT** en el código SQL.

i. Cuando esto se aplique a letras, el comando **MAX()** obtendrá los **registros** en orden alfabético empezando **desde la letra “z”** y **MIN()** los extraerá **desde la letra “a”**.

Query History

```
1 --No se puede hacer esto: Porque se obtendrá el máximo de toda la tabla y se asignará a cada columna.
2 SELECT tutor_id, nombre
3 FROM ejercicios.alumnos
4 GROUP BY tutor_id, nombre
5 ORDER BY tutor_id DESC;
6 --Se debe hacer esto: Para que se obtenga el máximo de cada agrupación (cada fila).
7 SELECT tutor_id, MIN(nombre)
8 FROM ejercicios.alumnos
9 GROUP BY tutor_id
10 ORDER BY tutor_id DESC;
```

Data Output

tutor_id	nombre
1	Ambrose
2	Ashien
3	Barrie
4	Belva
5	Bendick
6	Brigid
7	Candace
8	Cathee
9	Christal

Query History

```
1 --No se puede hacer esto: Porque se obtendrá el mínimo de toda la tabla y se asignará a cada columna.
2 SELECT tutor_id, nombre
3 FROM ejercicios.alumnos
4 GROUP BY tutor_id, nombre
5 ORDER BY tutor_id DESC;
6 --Se debe hacer esto: Para que se obtenga el mínimo de cada agrupación (cada fila).
7 SELECT tutor_id, MIN(nombre)
8 FROM ejercicios.alumnos
9 GROUP BY tutor_id
10 ORDER BY tutor_id DESC;
```

Data Output

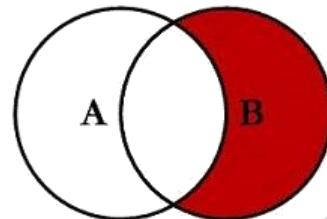
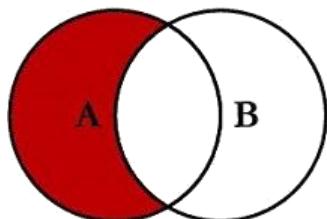
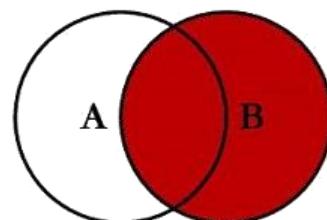
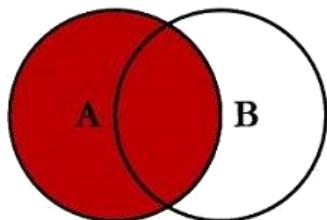
tutor_id	min
1	Ambrose
2	Aldridge
3	Alexandr
4	Adolpho
5	Aile
6	Angelo
7	Adina
8	Abbie
9	Adelaide

11.- Self JOINS: CONCAT(), JOIN, AS & ON

11. Realiza una consulta SQL que analice el caso donde un **alumno** se encuentre dando **tutorías** a otros **alumnos** dentro de una misma clase, obteniendo así el **nombre** y **apellido** del **alumno que tome cada tutoría** y el **nombre** y **apellido** de su **alumno tutor**. Para ello en **pgAdmin** se selecciona la opción de: Tools → Query Tool.

- a. **JOIN**: A través de la sentencia **FROM** se indica de qué **tabla** se extraerán los datos, de una consulta, pero este comando solo sirve cuando esto se realiza para una sola **entidad**, por lo que 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**).
- i. Los **self JOINs** se utilizan para poder realizar operaciones **JOIN** a través de diferentes **instancias o copias** de una misma **tabla**, en vez de hacerlo con más de una **entidad**, logrando así obtener la **intersección**, **unión**, etc. de sus **columnas**. Para ello se debe asignar un alias a cada **instancia** con el comando **AS** y a través de dichos alias se realizan las operaciones lógicas de los Diagramas de Venn, según el resultado que se quiera obtener con el comando **ON** y las **columnas** que se busque **relacionar** entre las **copias de la tabla**.

SQL JOINS



© C.L. Moffatt, 2008

Query Query History Execute script (F5) Scratch Pad x

```
1 --Consulta sin concatenación de variables
2 SELECT alumno.nombre,
3        alumno.apellido,
4        tutor.nombre,
5        tutor.apellido
6 FROM ejercicios.alumnos AS alumno
7      INNER JOIN ejercicios.alumnos AS tutor ON alumno.tutor_id = tutor.id;
```

Data Output Messages Notifications

	nombre	apellido	nombre	apellido
1	Wanda	Billington	Terencio	[null]
2	Blayne	Wakeley	Sheryl	Shergill
3	Hilde	Matiewe	Jerrome	Oganian
4	Leanna	[null]	Saree	Trousdale
5	Maire	Garnall	Charline	[null]
6	Nikolaus	Freeborn	Lou	Peck
7	Charline	[null]	Jerrome	Oganian
8	Tobe	Galia	Sheryl	Shergill
9	Terencio	[null]	Ashien	[null]
10	Garnet	Endrighi	Hendrika	[null]
11	Terri	Piercy	Damian	Scamerden
12	Billi	Mattinson	Saree	Trousdale
13	Lou	Peck	Damian	Scamerden
14	Saree	Trousdale	Jerrome	Oganian
15	Hana	Vaughn	Nikolaus	Freeborn
16	Susy	Weedall	Blayne	Wakeley
17	Damian	Scamerden	Franklyn	China
18	Franklyn	China	Jerrome	Oganian
19	Shir	Dionisotto	Veriee	Sealey
20	Adrienne	Thompson	Tobe	Galia

Total rows: 1000 of 1000 Query complete 00:00:00.104 Ln 7, Col 71

Query Query History Execute script (F5) Scratch Pad x

```
8 --Consulta con concatenación de variables
9 SELECT CONCAT(alumno.nombre, ' ', alumno.apellido) AS nombre_alumno,
10    CONCAT(tutor.nombre, ' ', tutor.apellido) AS nombre_tutor
11   FROM ejercicios.alumnos AS alumno
12     INNER JOIN ejercicios.alumnos AS tutor ON alumno.tutor_id = tutor.id;
```

Data Output Messages Notifications

	nombre_alumno	nombre_tutor
1	Wanda Billington	Terencio
2	Blayne Wakeley	Sheryl Shergill
3	Hilde Matiewe	Jerrome Oganian
4	Leanna	Saree Trousdale
5	Maire Garnall	Charline
6	Nikolaus Freeborn	Lou Peck
7	Charline	Jerrome Oganian
8	Tobe Galia	Sheryl Shergill
9	Terencio	Ashien
10	Garnet Endrighi	Hendrika
11	Terri Piercy	Damian Scamerden
12	Billi Mattinson	Saree Trousdale
13	Lou Peck	Damian Scamerden
14	Saree Trousdale	Jerrome Oganian
15	Hana Vaughn	Nikolaus Freeborn
16	Susy Weedall	Blayne Wakeley
17	Damian Scamerden	Franklyn China
18	Franklyn China	Jerrome Oganian
19	Shir Dionisotto	Veriee Sealey
20	Adrienne Thompson	Tobe Galia
21	Axel Shadé	Leanna
22	Ashien	Graham Caustic

Total rows: 1000 of 1000 Query complete 00:00:00.061 Ln 12, Col 71

12.- Self JOINS - Top 10: JOIN, COUNT(), GROUP/ORDER BY & LIMIT

12. Realiza una consulta SQL que analice el caso donde un **alumno** se encuentre dando **tutorías** a otros **alumnos** dentro de una misma clase, obteniendo primero el **nombre** y **apellido** del **alumno tutor** y luego el **número de alumnos a los que enseña**, ordenando su resultado de

mayor a menor y mostrando el top 10 de **alumnos tutores**. Para ello en **pgAdmin** se selecciona la opción de: Tools → Query Tool.

- JOIN:** A través de la sentencia **FROM** se indica de qué **tabla** se extraerán los datos, de una consulta, pero este comando solo sirve cuando esto se realiza para una sola **entidad**, por lo que 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**).
 - Los **self JOINs** se utilizan para poder realizar operaciones **JOIN** a través de **diferentes instancias o copias de una misma tabla**, en vez de hacerlo con más de una **entidad**, logrando así obtener la **intersección, unión, etc.** de sus **columnas**. Para ello se debe asignar un alias a cada **instancia** con el comando **AS** y a través de dichos alias se realizan las operaciones lógicas de los Diagramas de Venn, según el resultado que se quiera obtener con el comando **ON** y las **columnas** que se busque **relacionar** entre las **copias de la tabla**.
- Para acomodar los resultados de una consulta en función de los **valores mínimos o máximos** de **alguna de sus columnas**, usualmente se utilizan los comandos **GROUP BY** y **ORDER BY**.

The screenshot shows the pgAdmin Query Tool interface. The query window contains the following SQL code:

```

1 --Consulta self JOIN con concatenación de los valores de sus columnas, ordenamiento de mayor a menor y mostrando solo 10 datos:
2 SELECT CONCAT(tutor.nombre, ' ', tutor.apellido) AS nombre_tutor,
3        COUNT(*) AS alumnos_por_tutor
4 FROM ejercicios.alumnos AS alumno
5 INNER JOIN ejercicios.alumnos AS tutor ON alumno.tutor_id = tutor.id
6 GROUP BY nombre_tutor
7 ORDER BY alumnos_por_tutor DESC
8 LIMIT 10;

```

The Data Output window displays the results of the query:

nombre_tutor	alumnos_por_tutor
Saree Trousdale	49
Jerome Oganian	43
Blayne Wakeley	42
Lou Peck	42
Damian Scamerden	38
Hilde Matiewe	37
Terencio	37
Axel Shade	37
Franklyn China	36
Veriee Sealey	36

13.- Self JOINs - Promedio SubQuery: JOIN, AVG() & GROUP BY

- Realiza una consulta SQL que analice el caso donde un **alumno** se encuentre dando **tutorías** a otros **alumnos** dentro de una misma clase, obteniendo el **promedio de alumnos** que tienen todas las **filas de alumnos tutores**. Para ello en **pgAdmin** se selecciona la opción de: Tools → Query Tool.

- JOIN:** A través de la sentencia **FROM** se indica de qué **tabla** se extraerán los datos, de una consulta, pero este comando solo sirve cuando esto se realiza para una sola **entidad**, por lo que 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).

- i. Los **self JOINs** se utilizan para poder realizar operaciones **JOIN** a través de diferentes **instancias o copias** de una misma **tabla**, en vez de hacerlo con más de una **entidad**, logrando así obtener la **intersección, unión, etc.** de sus **columnas**. Para ello se debe asignar un alias a cada **instancia** con el comando **AS** y a través de dichos alias se realizan las operaciones lógicas de los Diagramas de Venn, según el resultado que se quiera obtener con el comando **ON** y las **columnas** que se busque **relacionar** entre las **copias de la tabla**.

```

Query   Query History   Execute script (F5)   Scratch Pad X
1 --Consulta self JOIN, agrupamiento por medio del nombre del alumno tutor, mostrando el promedio de alumnos:
2 SELECT AVG(alumnos_por_tutor) AS promedio_alumnos_por_tutor
3 FROM (
4     SELECT tutor.nombre AS nombre_tutor,
5            COUNT(*) AS alumnos_por_tutor
6     FROM ejercicios.alumnos AS alumno
7     INNER JOIN ejercicios.alumnos AS tutor ON alumno.tutor_id = tutor.id
8     GROUP BY nombre_tutor
9 ) AS alumnos_tutor;

```

Data Output Messages Notifications

	promedio_alumnos_por_tutor
1	33.333333333333333

14.- JOINs - Diferencias con dos tablas (A-B): CONCAT(), LEFT JOIN, etc.

14. Realiza una consulta SQL donde, **después de haber eliminado ciertas profesiones** en una escuela, averigüemos **cuáles alumnos se han quedado sin ninguna carrera asignada**. Para ello se analiza el caso donde **se tengan dos tablas distintas**, una de **alumnos** y otra de **carreras**, estas no se encuentran previamente unidas, pero se pueden **relacionar** a través de la **columna carrera_id** de la entidad **alumnos** y el **atributo id** de la entidad **carreras**. Para resolver el ejercicio se debe realizar una operación **JOIN** entre ellas. Por lo cual, en **pgAdmin** se debe seleccionar la opción de: Tools → Query Tool.

- a. Para resolver el ejercicio primero debemos saber en la tabla **alumnos** cuáles son las **carrera_id** que existen y cuántos **alumnos** le corresponden a cada uno.
- b. Luego debemos saber cuál es la **carrera_id** que posee el mayor número de **alumnos** y cuál es la menor.
- c. Después debemos asegurarnos de que en la tabla **carreras** no existan los **id** de carrera del 30 al 40.
- d. Finalmente, debemos realizar una operación de **Diferencia izquierda: A – B**, donde **A=alumnos** y **B=carreras**, para saber así los alumnos que no tienen asignada ninguna carrera.

Esto se puede resolver con una operación **JOIN** de diferencia, para la cual ya existe una forma estandarizada de aplicación:

- **LEFT JOIN:** Diferencia izquierda: **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;

```



Query History

```

1 --Conteo de alumnos por carrera, agrupando las filas que pertenecen a cada id y ordenandolas por # de carrera de mayor a menor.
2 SELECT carrera_id, COUNT(*) AS alumnos_por_carrera
3 FROM ejercicios.alumnos
4 GROUP BY carrera_id
5 ORDER BY carrera_id DESC;

```

Data Output

carrera_id	alumnos_por_carrera
1	50
2	49
3	48
4	47
5	46
6	45
7	44
8	43
9	42
10	41
11	40
12	39
13	38
14	37
15	36
16	35
17	34
18	33
19	32
20	31
21	30
22	29

Total rows: 50 of 50 Query complete 00:00:00.074

Scratch Pad

✓ Successfully run. Total query runtime: 74 msec. 50 rows affected. X

Ln 1, Col 1

Query History

```

1 --Conteo de alumnos por carrera, agrupando las filas que pertenecen a cada id y ordenandolas por # de carrera de mayor a menor.
2 SELECT carrera_id, COUNT(*) AS alumnos_por_carrera
3 FROM ejercicios.alumnos
4 GROUP BY carrera_id
5 ORDER BY carrera_id DESC;
6 --Conteo de alumnos por carrera, agrupando las filas que pertenecen a cada id y ordenandolas por # de alumnos de mayor a menor.
7 SELECT carrera_id, COUNT(*) AS alumnos_por_carrera
8 FROM ejercicios.alumnos
9 GROUP BY carrera_id
10 ORDER BY alumnos_por_carrera DESC;

```

Data Output

carrera_id	alumnos_por_carrera
1	22
2	9
3	28
4	20
5	35
6	47
7	39
8	42
9	13
10	6
11	19
12	24
13	43
14	31
15	25
16	45
17	23
18	33

Total rows: 50 of 50 Query complete 00:00:00.080

Scratch Pad

✓ Successfully run. Total query runtime: 80 msec. 50 rows affected. X

Ln 6, Col 1

```

11 --Eliminación de las carreras con id de 30 a 40 de la tabla carreras, para posteriormente poder hacer una diferencia A-B.
12 DELETE FROM ejercicios.carreras
13 WHERE id BETWEEN 30 AND 40;

```

Data Output

DELETE 0

Query returned successfully in 43 msec.

Query History

```

1 --Conteo de alumnos por carrera, agrupando las filas que pertenecen a cada id y ordenandolas por # de carrera de mayor a menor.
2 SELECT carrera_id, COUNT(*) AS alumnos_por_carrera
3 FROM ejercicios.alumnos
4 GROUP BY carrera_id
5 ORDER BY carrera_id DESC;
6 --Conteo de alumnos por carrera, agrupando las filas que pertenecen a cada id y ordenandolas por # de alumnos de mayor a menor.
7 SELECT carrera_id, COUNT(*) AS alumnos_por_carrera
8 FROM ejercicios.alumnos
9 GROUP BY carrera_id
10 ORDER BY alumnos_por_carrera DESC;
11 --Eliminación de las carreras con id de 30 a 40 de la tabla carreras, para posteriormente poder hacer una diferencia A-B.
12 DELETE FROM ejercicios.carreras
13 WHERE id BETWEEN 30 AND 40;
14 --Diagrama de Venn de Diferencia A-B = alumnos-carreras
15 SELECT CONCAT(A.nombre, ' ', A.apellido),
16        A.carrera_id,
17        B.id,
18        B.carrera
19 FROM ejercicios.alumnos AS A
20 LEFT JOIN ejercicios.carreras AS B
21 ON A.carrera_id = B.id
22 WHERE B.id IS NULL
23 ORDER BY A.carrera_id;

```

Data Output Messages Notifications

concat	carrera_id	id	carrera
Gill Calveley	30	[null]	[null]
Gwenni Durbridge	30	[null]	[null]
Cornelia Youdell	30	[null]	[null]
Mural Rubica	30	[null]	[null]
Morgan Crackett	30	[null]	[null]
Claire Boyat	30	[null]	[null]
Krishna Barbrick	30	[null]	[null]
Shurlock Kearns	30	[null]	[null]

Total rows: 219 of 219 Query complete 00:00:00.050

Scratch Pad

Successfully run. Total query runtime: 50 msec. 219 rows affected.

Ln 14, Col 1

15.- Tipos de JOINs con 2 tablas: Unión, Diferencia, Intersección, etc.

15. En el ejercicio anterior se realizó la operación de Diferencia izquierda: $A - B$, con una variante del comando **JOIN**, donde $A = \text{alumnos}$ y $B = \text{carreras}$, ahora realiza las operaciones faltantes de Intersección: $A \cap B$, Unión: $A \cup B$, Diferencia e intersección izquierda: $A - B + (A \cap B)$, Diferencia e intersección derecha: $B - A + (A \cap B)$, Diferencia derecha: $B - A$ y Diferencia simétrica: $A \cup B - (A \cap B)$. Para ello en pgAdmin se selecciona la opción de: Tools → Query Tool.

a. Diferencia izquierda: $A - B$.

```

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;

```

Query History

```

14 --Diagrama de Venn de Diferencia A-B = alumnos-carreras
15 SELECT CONCAT(A.nombre, ' ', A.apellido),
16        A.carrera_id,
17        B.id,
18        B.carrera
19 FROM ejercicios.alumnos AS A
20 LEFT JOIN ejercicios.carreras AS B
21 ON A.carrera_id = B.id
22 WHERE B.id IS NULL
23 ORDER BY A.carrera_id;

```

Data Output Messages Notifications

concat	carrera_id	id	carrera
Gill Calveley	30	[null]	[null]
Gwenni Durbridge	30	[null]	[null]
Cornelia Youdell	30	[null]	[null]
Mural Rubica	30	[null]	[null]
Morgan Crackett	30	[null]	[null]
Claire Boyat	30	[null]	[null]
Krishna Barbrick	30	[null]	[null]
Shurlock Kearns	30	[null]	[null]
Giustina Pole	30	[null]	[null]
Zed Curnning	30	[null]	[null]
Ursula Singeck	30	[null]	[null]
Addison Mar	30	[null]	[null]
Troy Hartner	30	[null]	[null]
Matilda Saffell	30	[null]	[null]
Harper Kulle	30	[null]	[null]
Rafaello	30	[null]	[null]
Gilberte Abela	30	[null]	[null]
Kyriha Mattasseri	31	[null]	[null]

Total rows: 219 of 219 Query complete 00:00:00.057

Scratch Pad

Successfully run. Total query runtime: 57 msec. 219 rows affected.

Ln 14, Col 1

b. Diferencia e intersección izquierda: $A - B + (A \cap B)$.

SELECT * FROM Nombre_Tabla_Izq

LEFT JOIN Nombre_Tabla_Der ON Tabla_Izq.PRIMARY_KEY = Tabla_Der.FOREIGN_KEY;

```

24 --Diagrama de Venn de Diferencia izquierda: A - B + (A ∩ B)
25 SELECT CONCAT(A.nombre, ' ', A.apellido),
26       A.carrera_id,
27       B.id,
28       B.carrera
29 FROM ejercicios.alumnos AS A
30 LEFT JOIN ejercicios.carreras AS B
31 ON A.carrera_id = B.id
32 ORDER BY A.carrera_id DESC;

```

concat	carrera_id	id	carrera
Penny Margetson	50	50	Ciencias naturales, exactas y de la computación
Aldon Comettoi	50	50	Ciencias naturales, exactas y de la computación
Kirstin Ingafil	50	50	Ciencias naturales, exactas y de la computación
Rosaleen Yosselevitch	50	50	Ciencias naturales, exactas y de la computación
Hall Scallon	50	50	Ciencias naturales, exactas y de la computación
Rinaldo Planque	50	50	Ciencias naturales, exactas y de la computación
Jackie	50	50	Ciencias naturales, exactas y de la computación
Kiersten Evelyn	50	50	Ciencias naturales, exactas y de la computación
Beatrisa	50	50	Ciencias naturales, exactas y de la computación
Dre Mumm	50	50	Ciencias naturales, exactas y de la computación
Odetta Tidman	50	50	Ciencias naturales, exactas y de la computación
Leticia	50	50	Ciencias naturales, exactas y de la computación
Lillian Hagarth	50	50	Ciencias naturales, exactas y de la computación
Jackie Mogford	50	50	Ciencias naturales, exactas y de la computación
Honey Skally	50	50	Ciencias naturales, exactas y de la computación
Perla Baggallay	50	50	Ciencias naturales, exactas y de la computación
Leonardo Oglevie	50	50	Ciencias naturales, exactas y de la computación
Iolanthe Manlow	50	50	Ciencias naturales, exactas y de la computación
Brady Clace	50	50	Ciencias naturales, exactas y de la computación

Successfully run. Total query runtime: 60 msec. 1000 rows affected.

C. Unión: $A \cup B$

SELECT Nombre_Columna FROM Nombre_Tabla_Izq

FULL OUTER JOIN Nombre_Tabla_Der ON Tabla_Izq.PRIMARY_KEY = Tabla_Der.FOREIGN_KEY;

```

33 --Diagrama de Venn de Unión AUB
34 SELECT CONCAT(A.nombre, ' ', A.apellido),
35       A.carrera_id,
36       B.id,
37       B.carrera
38 FROM ejercicios.alumnos AS A
39 FULL OUTER JOIN ejercicios.carreras AS B
40 ON A.carrera_id = B.id
41 ORDER BY A.carrera_id DESC;

```

concat	carrera_id	id	carrera
[null]		53	Ciencias ambientales
[null]		58	Física
[null]		60	Matemáticas y estadística
[null]		55	Ciencias de la computación
[null]		52	Biología y bioquímica
[null]		56	Ciencias físicas, químicas y de la tierra
[null]		51	Ciencias naturales
[null]		54	Ciencias de la computación
[null]		59	Química
[null]		57	Ciencias de la tierra y la atmósfera
Hall Scallon	50	50	Ciencias naturales, exactas y de la computación
Aldon Comettoi	50	50	Ciencias naturales, exactas y de la computación
Kirsten Evelyn	50	50	Ciencias naturales, exactas y de la computación
Penny Margetson	50	50	Ciencias naturales, exactas y de la computación
Rosaleen Yosselevitch	50	50	Ciencias naturales, exactas y de la computación
Dre Mumm	50	50	Ciencias naturales, exactas y de la computación
Perla Baggallay	50	50	Ciencias naturales, exactas y de la computación
Kirstin Ingafil	50	50	Ciencias naturales, exactas y de la computación
Honey Skally	50	50	Ciencias naturales, exactas y de la computación

Successfully run. Total query runtime: 77 msec. 1010 rows affected.

d. Diferencia derecha: $B - A$

```
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;
```

The screenshot shows a MySQL Workbench interface with a query editor and a results grid. The query is:

```
--Diagrama de Venn de Diferencia Derecha: B - A = carreras-alumnos
SELECT CONCAT(A.nombre, ' ', A.apellido),
       A.carrera_id,
       B.id,
       B.carrera
FROM ejercicios.alumnos AS A
RIGHT JOIN ejercicios.carreras AS B
ON A.carrera_id = B.id
WHERE A.carrera_id IS NULL
ORDER BY B.id DESC;
```

The results grid has columns: concat, carrera_id, id, and carrera. The data shows 10 rows of results:

concat	carrera_id	id	carrera
	[null]	60	Matemáticas y estadística
	[null]	59	Química
	[null]	58	Física
	[null]	57	Ciencias de la tierra y la atmósfera
	[null]	56	Ciencias físicas, químicas y de la tierra
	[null]	55	Ciencias de la computación
	[null]	54	Ciencias de la computación
	[null]	53	Ciencias ambientales
	[null]	52	Biología y bioquímica
	[null]	51	Ciencias naturales

Total rows: 10 of 10 Query complete 00:00:00.045 Ln 42, Col 1

e. Diferencia e intersección derecha: $B - A + (A \cap B)$

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

The screenshot shows a MySQL Workbench interface with a query editor and a results grid. The query is:

```
--Diagrama de Venn de Diferencia e Intersección derecha: B - A + (A ∩ B)
SELECT CONCAT(A.nombre, ' ', A.apellido),
       A.carrera_id,
       B.id,
       B.carrera
FROM ejercicios.alumnos AS A
RIGHT JOIN ejercicios.carreras AS B
ON A.carrera_id = B.id
ORDER BY A.carrera_id DESC;
```

The results grid has columns: concat, carrera_id, id, and carrera. The data shows 791 rows of results, including many rows from the previous screenshot and additional rows from the intersection of the two tables.

concat	carrera_id	id	carrera
	[null]	57	Ciencias de la tierra y la atmósfera
	[null]	58	Física
	[null]	56	Ciencias físicas, químicas y de la tierra
	[null]	52	Biología y bioquímica
	[null]	55	Ciencias de la computación
	[null]	54	Ciencias de la computación
	[null]	60	Matemáticas y estadística
	[null]	53	Ciencias ambientales
	[null]	51	Ciencias naturales
	[null]	59	Química
Leonardo Oglevie	50	50	Ciencias naturales, exactas y de la computación
Iolanthe Manlow	50	50	Ciencias naturales, exactas y de la computación
Dre Mumm	50	50	Ciencias naturales, exactas y de la computación
Bradly Clace	50	50	Ciencias naturales, exactas y de la computación
Honey Skally	50	50	Ciencias naturales, exactas y de la computación
Odetta Tidman	50	50	Ciencias naturales, exactas y de la computación
Kirstin Ingarfill	50	50	Ciencias naturales, exactas y de la computación
Jackie Mogford	50	50	Ciencias naturales, exactas y de la computación
Peria Baggallay	50	50	Ciencias naturales, exactas y de la computación

Total rows: 791 of 791 Query complete 00:00:00.096 Ln 60, Col 28

f. Intersección: $A \cap B$

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

concat	carrera_id	id	carrera
1 Penny Margetson	50	50	Ciencias naturales, exactas y de la computación
2 Aldon Comettoi	50	50	Ciencias naturales, exactas y de la computación
3 Kirstin Ingaffill	50	50	Ciencias naturales, exactas y de la computación
4 Rosaleen Yosselevitch	50	50	Ciencias naturales, exactas y de la computación
5 Hall Scallion	50	50	Ciencias naturales, exactas y de la computación
6 Rinaldo Planque	50	50	Ciencias naturales, exactas y de la computación
7 Jackie	50	50	Ciencias naturales, exactas y de la computación
8 Kiersten Evelyn	50	50	Ciencias naturales, exactas y de la computación
9 Beatrisa	50	50	Ciencias naturales, exactas y de la computación
10 Dre Mumm	50	50	Ciencias naturales, exactas y de la computación
11 Odetta Tidman	50	50	Ciencias naturales, exactas y de la computación
12 Leticia	50	50	Ciencias naturales, exactas y de la computación
13 Lillian Haggarth	50	50	Ciencias naturales, exactas y de la computación
14 Jackie Mogford	50	50	Ciencias naturales, exactas y de la computación
15 Honey Skally	50	50	Ciencias naturales, exactas y de la computación
16 Perla Bagallay	50	50	Ciencias naturales, exactas y de la computación
17 Leonardo Ogilvie	50	50	Ciencias naturales, exactas y de la computación
18 Iolanthe Manlow	50	50	Ciencias naturales, exactas y de la computación
19 Brandy Olace	50	50	Ciencias naturales, exactas y de la computación

Total rows: 781 of 781 Query complete 00:00:00.090 Ln 61, Col 1

Successfully run. Total query runtime: 90 msec. 781 rows affected.

g. Diferencia simétrica: $A \Delta B = (A \cup B) - (A \cap B)$

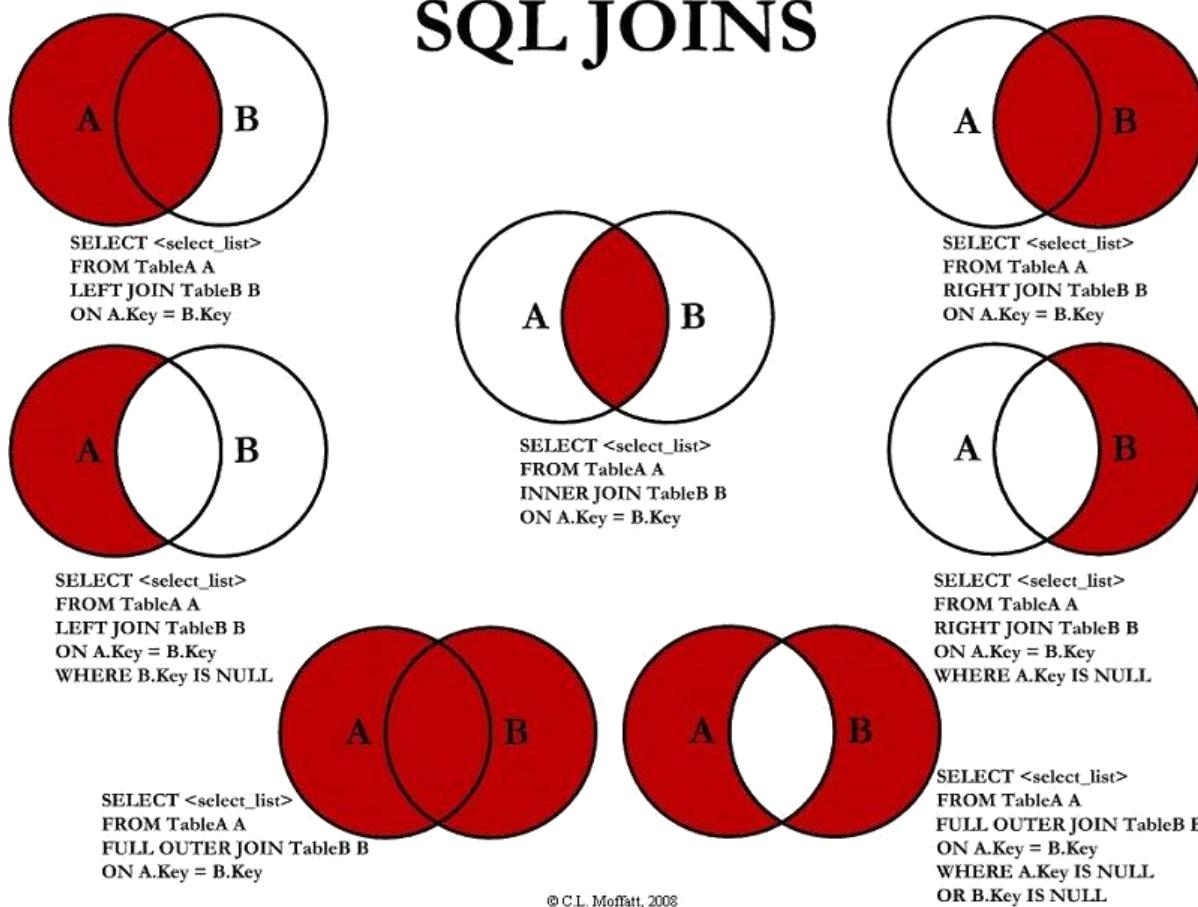
```
SELECT * FROM Nombre_Tabla_Izq
FULL OUTER JOIN Nombre_Tabla_Der ON Tabla_Izq.PRIMARY_KEY = Tabla_Der.FOREIGN_KEY
WHERE Tabla_Izq.PRIMARY_KEY IS NULL OR Tabla_Der.FOREIGN_KEY IS NULL;
```

concat	carrera_id	id	carrera
1 Giustina Pole	30	[null]	[null]
2 Leanna	36	[null]	[null]
3 Tobe Gallia	32	[null]	[null]
4 Terri Piercy	31	[null]	[null]
5 Lou Peck	40	[null]	[null]
6 Shir Dionisotto	35	[null]	[null]
7 Adrienne Thompson	36	[null]	[null]
8 Ashien	31	[null]	[null]
9 Barde Turford	38	[null]	[null]
10 Denyse Baldam	32	[null]	[null]
11 Elizabeth Aspel	34	[null]	[null]
12 Mersey Geeraert	35	[null]	[null]
13 Griffin Tubble	40	[null]	[null]
14 Linette Loebner	39	[null]	[null]
15 Dew Matyas	39	[null]	[null]
16 Sydney Dener	34	[null]	[null]
17 Kynthia	39	[null]	[null]

Total rows: 229 of 229 Query complete 00:00:00.053 Ln 80, Col 20

Successfully run. Total query runtime: 53 msec. 229 rows affected.

SQL JOINS



16.- Autocompletando Strings/Figuras Geométricas: `lpad()` & `CAST()`

16. Realiza una consulta SQL que **genere triángulos a través de varios métodos recursivo (que se ejecuten varias veces)**. Para ello en **pgAdmin** se selecciona la opción de: Tools → Query Tool.

- El 2do parámetro del método `lpad()` indica el **número de caracteres** que conforman al **String**; este recibe solamente **datos de tipo int**, por lo cual se debe utilizar el método `CAST()` para asegurarnos de que la **columna** que reciba sea de ese tipo. Además, este **atributo** se puede asignar de forma directa o provenir de una **subconsulta**.

```

Query   Query History
--Método lpad() estático:
1 SELECT lpad('di_cer0', 20, 'x');
2
3
4 --Método lpad() recursivo con columna numérica y limitado por el comando WHERE:
5 SELECT lpad('di_cer0', id, '/')
6 FROM ejercicios.alumnos
7 WHERE id <= 12;
8
9 --Método lpad() recursivo con columna numérica generada por una subconsulta y reformatteada con el comando CAST():
10 SELECT lpad('*', CAST(num_subconsulta AS int), '*')
11 FROM (
12     SELECT ROW_NUMBER() OVER() AS num_subconsulta,
13     FROM ejercicios.alumnos
14 ) AS columna_numerica_subquery
15 WHERE id <= 7;

Data Output  Messages  Notifications
lpad
text
1 xxxxxxxxxxxxxdi_cer0

```

The screenshot shows the pgAdmin interface with two main windows. The top window is the 'Query' tool, displaying SQL code for generating ranges using the `lpad()` function. The bottom window is the 'Data Output' tool, showing the results of the query. The results are a series of strings where each row contains a number from 1 to 12 followed by a varying number of 'd' characters, demonstrating the static padding behavior of `lpad()`.

```

1 --Método lpad() estático:
2 SELECT lpad('di_cer0', 20, 'x');
3
4 --Método lpad() recursivo con columna numérica y limitado por el comando WHERE:
5 SELECT lpad('di_cer0', id, '/');
6 FROM ejercicios.alumnos
7 WHERE id <= 12;
8
9 --Método lpad() recursivo con columna numérica generada por una subconsulta y reformateada con el comando CAST():
10 SELECT lpad('*', CAST(num_subconsulta AS int), '*')
11 FROM (
12     SELECT ROW_NUMBER() OVER() AS num_subconsulta, *
13     FROM ejercicios.alumnos
14 ) AS columna_numerica_subquery
15 WHERE id <= 7;

```

	lpad text
1	d
2	di
3	di_
4	di_c
5	di_ce
6	di_cer
7	di_cer0
8	/di_cer0
9	///di_cer0
10	///di_cer0
11	////di_cer0
12	/////di_cer0

Total rows: 12 of 12 Query complete 00:00:00.064 Successfully run. Total query runtime: 64 msec. 12 rows affected.

This screenshot shows the same pgAdmin interface, but the results in the 'Data Output' tool are different. Instead of padding with 'd', the results show a series of asterisks ('*') where each row contains a number from 1 to 7 followed by a varying number of '*' characters, demonstrating the recursive padding behavior of `lpad()` with a subquery.

	lpad text
1	*
2	**
3	***
4	****
5	*****
6	*****
7	*****

17.- Generación de Rangos: `generate_series()`

17. Realiza una consulta SQL que **genere una serie de números consecutivos (un rango)** a través del método `generate_series()`. Para ello en pgAdmin se selecciona la opción de: Tools → Query Tool.

- En los parámetros del método `generate_series()` se indica el número de inicio del rango, su número final y su paso (el número con el cuál aumenta o decrece el valor de los números consecutivos dentro del rango). Se puede crear un rango decreciente

solamente cuando se indique explícitamente en el tercer parámetro un paso negativo, sino no se nos permitirá crear dicho rango.

```

Query Query History Execute script Scratch Pad X
1 --Método generate_series() para generar rango numérico creciente sin paso indicado explícitamente:
2 SELECT *
3 FROM generate_series(1,1, 4);
Data Output Messages Notifications
generate_series numeric
1 1.1
2 2.1
3 3.1

Query Query History Execute script Scratch Pad X
1 --Método generate_series() para generar rango numérico creciente sin paso indicado explícitamente:
2 SELECT *
3 FROM generate_series(1.1, 4);
4 /*Método generate_series() para generar rango de tipo timestamp creciente con paso indicado explícitamente:
   - timestamp: En este tipo de dato se pueden especificar partes de la fecha u hora a través de los strings: years, months,
   days, hours, minutes, seconds, quarters, dow (día de la semana), doy (día del año), etc.*/
7 SELECT *
8 FROM generate_series('2024-04-06 00:00:00'::timestamp, '2024-04-09 00:00:00', '10 hours');

Data Output Messages Notifications
generate_series timestamp without time zone
1 2024-04-06 00:00:00
2 2024-04-06 10:00:00
3 2024-04-06 20:00:00
4 2024-04-07 06:00:00
5 2024-04-07 16:00:00
6 2024-04-08 02:00:00
7 2024-04-08 12:00:00
8 2024-04-08 22:00:00

Query Query History Execute script Scratch Pad X
1 --Método generate_series() para generar rango numérico creciente sin paso indicado explícitamente:
2 SELECT *
3 FROM generate_series(1,1, 4);
4 /*Método generate_series() para generar rango de tipo timestamp creciente con paso indicado explícitamente:
   - timestamp: En este tipo de dato se pueden especificar partes de la fecha u hora a través de los strings: years, months,
   days, hours, minutes, seconds, quarters, dow (día de la semana), doy (día del año), etc.*/
7 SELECT *
8 FROM generate_series('2024-04-06 00:00:00'::timestamp, '2024-04-09 00:00:00', '10 hours');
9 --Método generate_series() para generar rango numérico decreciente sin paso indicado explícitamente: Generará un arreglo vacío.
10 SELECT *
11 FROM generate_series(4, 3);

Data Output Messages Notifications
generate_series integer

Query Query History Execute script Scratch Pad X
1 --Método generate_series() para generar rango numérico creciente sin paso indicado explícitamente:
2 SELECT *
3 FROM generate_series(1,1, 4);
4 /*Método generate_series() para generar rango de tipo timestamp creciente con paso indicado explícitamente:
   - timestamp: En este tipo de dato se pueden especificar partes de la fecha u hora a través de los strings: years, months,
   days, hours, minutes, seconds, quarters, dow (día de la semana), doy (día del año), etc.*/
7 SELECT *
8 FROM generate_series('2024-04-06 00:00:00'::timestamp, '2024-04-09 00:00:00', '10 hours');
9 --Método generate_series() para generar rango numérico decreciente sin paso indicado explícitamente: Generará un arreglo vacío.
10 SELECT *
11 FROM generate_series(4, 3);
12 --Método generate_series() para generar rango numérico decreciente con paso indicado explícitamente:
13 SELECT *
14 FROM generate_series(5, 1, -2);

Data Output Messages Notifications
generate_series integer
1 5
2 3
3 1

```

```

15 --Operación con un rango creado por el método generate_series() y la fecha actual del sistema current_date;
16 SELECT current_date + tablaRango.numeros AS fechas_rango
17 FROM generate_series(0, 14, 7) AS tablaRango(numeros);

```

Data Output Messages Notifications

	fechas_rango	date
1	2024-06-14	
2	2024-06-21	
3	2024-06-28	

```

18 --Operación JOIN de una tabla con un rango creado por el método generate_series():
19 SELECT alumno.id,
20       CONCAT(alumno.nombre, ' ', alumno.apellido),
21       alumno.carrera_id,
22       tablaRango.numeros
23 FROM ejercicios.alumnos AS alumno
24 INNER JOIN generate_series(0, 10) AS tablaRango(numeros)
25 ON tablaRango.numeros = alumno.carrera_id
26 ORDER BY alumno.carrera_id;

```

Data Output Messages Notifications

	id	concat	carrera_id	numeros
1	427	Zach Seally	1	1
2	718	Catina Yuryev	1	1
3	922	Elizabet	1	1
4	724	Monroe	1	1
5	544	Niles Cars	1	1
6	197	Ranique Botwood	1	1
7	749	Aleksandr Millhill	1	1
8	494	Tova Midlar	1	1
9	504	Geri	1	1
10	564	Ursula Vispo	1	1
11	625	Nickolaus	1	1
12	634	Merrick Dunnet	1	1
13	37	Shaw Harrill	1	1
14	236	Michal Thrapp	1	1
15	532	Willard Bollis	1	1
16	83	Chloris Markovic	1	1
17	699	Bay Rodson	1	1
18	884	Benedicta Ferryn	2	2
19	489	Cymbre Nannizzi	2	2

Total rows: 190 of 190 Query complete 00:00:00.073

Successfully run. Total query runtime: 73 msec. 190 rows affected. X

Ln 26, Col 29

```

27 --Generación de un rango numérico
28 -- el método generate_series() y su índice incluido con el comando WITH:
29 SELECT *
30 FROM generate_series(10, 2, -2) WITH ordinality;
31 --Creación de triángulos con un rango numérico creado por el método generate_series() con índice incluido y el método lpad():
32 SELECT lpad('^\'', CAST(ordinality AS int), '+')
33 FROM generate_series(10, 2, -2) WITH ordinality;

```

Data Output Messages Notifications

	generate_series	ordinality
1	10	1
2	8	2
3	6	3
4	4	4
5	2	5

```

30 --Creación de triángulos con un rango numérico creado por el método generate_series() con índice incluido y el método lpad():
31 SELECT lpad('^\'', CAST(ordinality AS int), '+')
32 FROM generate_series(10, 2, -2) WITH ordinality;

```

Data Output Messages Notifications

	lpad
1	^
2	^^
3	^^^
4	^^^^
5	^^^^^

18.- Expresiones Regulares - Validación de Patrones:

18. Realiza una consulta SQL que a través de expresiones regulares analice si el contenido de una columna es o no un email. Para ello en pgAdmin se selecciona la opción de: Tools → Query Tool.
- Las expresiones regulares no son particulares de SQL, existen en todos los lenguajes de programación para detectar patrones en strings y así identificar comandos, palabras, etc. Esta tecnología se utiliza en bots como Alexa para detectar palabras de activación que ejecuten cierta acción.

- En Python las expresiones regulares se utilizan a través de la librería **re**.

Query Query History Execute script [F5] Scratch Pad X

```
1 --Consulta con expresiones regulares para encontrar cualquier email:  
2 SELECT email  
3 FROM ejercicios.alumnos  
4 WHERE email ~*'^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}';  
5 --Consulta con expresiones regulares para encontrar una terminación de email específica:  
6 SELECT email  
7 FROM ejercicios.alumnos  
8 WHERE email ~*'^[A-Z0-9._%+-]+@google[A-Z0-9.-]+\.[A-Z]{2,4}';
```

Data Output Messages Notifications

email	character varying (50)
1	wbillington0@nsw.gov.au
2	bwakely1@google.cn
3	hmatiewe2@hibu.com
4	ldusey3@ebay.co.uk
5	mgarnall4@artisteer.com
6	nfreeborn5@yandex.ru
7	cvatts6@google.it
8	tgalia7@album.net
9	twadham8@dot.gov
10	gendrich9@spiegel.de
11	tpiercy9@a8.net
12	bmattinsonb@abc.net.au
13	lpeckc@washingtongpost.com
14	stroudaled@yale.edu
15	hvaughn@ted.com
16	sweedalf@google.ru
17	dscamerdeng@pcworld.com
18	fchinah@netvibes.com
19	sdioniseitto@google.com

✓ Successfully run. Total query runtime: 75 msec. 1000 rows affected. X

Total rows: 1000 of 1000 Query complete 00:00:00.075

Ln 1, Col 1

Query Query History Execute script [F5] Scratch Pad X

```
1 --Consulta con expresiones regulares para encontrar cualquier email:  
2 SELECT email  
3 FROM ejercicios.alumnos  
4 WHERE email ~*'^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}';  
5 --Consulta con expresiones regulares para encontrar una terminación de email específica:  
6 SELECT email  
7 FROM ejercicios.alumnos  
8 WHERE email ~*'^[A-Z0-9._%+-]+@google[A-Z0-9.-]+\.[A-Z]{2,4}';
```

Data Output Messages Notifications

email	character varying (50)
1	rpidy3n@google.com.br
2	kloveredge84@google.com.au
3	kexrolcj@google.com.au
4	cweightmanev@google.co.uk
5	brodsonje@google.co.jp
6	fgudginjq@google.com.br
7	lwigfallkg@google.co.uk
8	adobiels@google.co.jp
9	dhuskeo1@google.com.hk

Bases de Datos Distribuidas

Las bases de datos distribuidas son una colección de bases de datos o una misma base de datos que está partida en diferentes pedazos que se encuentran geográficamente distribuidos, osea que sus servidores se encuentran en distintos lugares, pero se encuentran conectados a través de una red informática.

- *Ventajas:*
 - **Desarrollo modular:** Esto es útil porque al estar separadas las **bases de datos**, cada una puede tener distintos usos o usuarios o simplemente divida la información para reducir los tiempos de respuesta al hacerles consultas.
 - **Incremento de confiabilidad:** La confiabilidad es una medida de la disponibilidad, confiabilidad, velocidad, capacidad de mantenimiento, durabilidad, seguridad o protección de una **base de datos** o cualquier sistema informático. Esta se mejora al tener bases de datos divididas por sectores, por ejemplo, una donde su servidor se encuentra ubicado geográficamente en México, podría dar un mejor servicio a los usuarios de la región de Latinoamérica, mientras que las demás partes de las **databases** pueden proporcionar servicio de datos a los demás usuarios del mundo, dependiendo de su región.
- *Desventajas:*
 - **Manejo de seguridad:** Al tener divididos los datos en varios servidores, cada uno debe manejar protocolos de seguridad y esto puede ser difícil y costoso de mantener.
 - **Complejidad del procesamiento:** La lógica de consultas cuando las **bases de datos** se encuentran divididas por sectores o usuarios se puede complejizar mucho, ya que se tendrá que considerar que la información devuelta estará en cierta parte incompleta, porque solo está accediendo a una parte de la **database** total.
- *Tipos:*
 - **Homogéneas:** Son las **bases de datos distribuidas** donde todas ellas son del mismo tipo (relacional o no relacional), sistema operativo, DBMS (manejador de base de datos) y motor o modelo de base de datos (PostgreSQL, MongoDB, MySQL, etc.).
 - **Heterogéneas:** Son las **databases distribuidas** donde todas ellas no son del mismo tipo, SO, DBMS y/o motor o modelo de bases de datos.
- *Arquitecturas:*
 - **Cliente-Servidor:** Es la más clásica donde se tiene una **base de datos principal** y tiene varias que se comunican con ella (**SLAVES**) que van a tratar de obtener datos de la database principal, a la cual normalmente se le realizan las escrituras de datos.
 - **Par a Par (Peer 2 Peer):** En esta arquitectura **no hay jerarquía de bases de datos**, todas ellas son iguales y se comunican como iguales.
 - **Multi manejador:** Es un tipo de **base de datos distribuidas heterogéneas**, donde de igual manera todas ellas se comunican como iguales.
- *Estrategias de diseño:*
 - **Top down:** Es un sistema de **bases de datos distribuidas con arquitectura cliente-servidor**, donde se tiene bien clara la jerarquía, los tipos de motores de bases de datos que se quiere utilizar y como se comunican unas con las otras.

- **Bottom up:** Es un sistema de **bases de datos distribuidas** donde se debe diseñar una arquitectura con **databases** existentes, tratando de darle el mayor orden posible, pero trabajando con algo previamente hecho.
- *Almacenamiento distribuido:*
 - **Fragmentación:** Cuando se dividen los datos en **databases distribuidas**, se debe tener muy en cuenta qué datos van en dónde.
 - **Horizontal o Sharding:** Tiene que ver con partir la información de la **tabla** de una **base de datos** en diferentes **pedazos horizontales** (osea dividir sus **filas** en partes divididas geográficamente, por ejemplo).
 - **Vertical:** Es cuando la información de la **tabla** de una **base de datos** se divide en **pedazos verticales por columnas** (osea dividir sus **atributos** en función de su ubicación geográfica, por ejemplo).
 - **Mixta:** Es una combinación de una **fragmentación vertical y horizontal**.
 - **Replicación:** **Réplica** se refiere al proceso de **copiar y mantener actualizada una copia de la base de datos en un servidor diferente**. Esta técnica es utilizada para evitar problemas de **entrada y salida de datos** en los sistemas operativos, ya que, si nuestra aplicación realiza **peticiones de lectura y escritura de datos** de forma exponencial; como **no se puede leer y escribir datos** en una **tabla** al mismo tiempo, esta se bloquea durante dicho proceso, pero cuando se tienen manejo de datos múltiples, existen límites electrónicos o físicos, que restringen la capacidad de procesamiento del CPU, por lo que una **petición podría tardar minutos en ejecutarse** y esto es catastrófico. Por eso es tan importante el uso de **réplicas**, donde al menos se cuenta con **dos servidores distintos (aunque pueden ser más de 2)**, uno como **master** y el otro es la **réplica**.
 - Se tiene un **servidor** con una **database principal**, donde solo se realizan las **entradas o modificaciones de datos**.
 - Y otro **servidor** con una **base de datos secundaria** (que es la **réplica**), donde solo se realiza la **lectura de datos**.
 - **Distribución:** También se debe tener en cuenta cómo va a ser compartida la información entre una base de datos y otra.
 - Centralizada, particionada o replicada.

Ejercicio de Query Distribuido

Cuando una **base de datos** se encuentra **distribuida**, las consultas se realizan de otra forma:

- Se tiene una base de datos distribuida donde contamos con **3 tablas**, una de **proveedores (P)**, una de **repuestos (R)** y una **tabla que une las dos anteriores (PR)**, donde se tienen las siguientes características:

Tabla	Columnas		Tuplas	Localización
Proveedores(P)	P#	Ciudad	10,000	Región A
Repuestos (R)	R#	Color	100,000	Región B
Prov-Rep(PR)	P#	R#	1,000,000	Región A



- Cada **tupla** (**fila de información**) pesa **25 bytes ≈ 200 bits**.
- Las **bases de datos** se encuentran **distribuidas** en las regiones A y B.
- La consulta o Query que se le realizará a la **base de datos distribuida** obtendrá el **id o #** (**número**) del **proveedor** que se encuentre en **Bogotá** y que a su vez nos proporcione repuestos de **color rojo**.
 - Los **repuestos rojos** existentes son **10**.
 - El número de pedidos hechos por **proveedores** desde **Bogotá** son **100,000**, estos existen en la **tabla PR**.
 - La **velocidad de comunicación entre las bases de datos** tiene una **tasa de transferencia** de 50,000 bits por segundo.
 - **Tasa de transferencia:** Se refiere a la velocidad en la que se pueden transmitir datos entre diferentes **nodos (servidores o bases de datos individuales)** en una **base de datos distribuida**.
 - **Latencia de acceso:** Es el **tiempo que tarda una consulta** en viajar desde el **cliente o RDBMS (Relational Database Management System)** a la **base de datos** y volver con una respuesta.
 - **Número de mensajes:** Se refiere a la **cantidad de operaciones (envío o recepción de datos)** que ocurren entre las distintas **bases de datos** conectadas (**nodos**) durante el procesamiento de una consulta realizada por un **cliente o RDBMS**.
 - Cada mensaje intercambiado introduce un pequeño retraso debido a la **latencia de acceso de la red** y el tiempo de procesamiento requerido para manejar **cada mensaje** en los **nodos de la base de datos distribuida**.

$$\text{Retraso de comunicación} = \text{Retraso de acceso} + \text{Tiempo de transferencia de datos}$$

$$\text{Retraso de comunicación} = \# \text{mensajes} * \text{Latencia de acceso} + \frac{\text{Volumen total de datos}}{\text{Tasa de transferencia}}$$

$$\text{Retraso de comunicación} = \# \text{mensajes} * \text{Latencia de acceso} + \frac{\# \text{Tuplas} * \text{Bits de peso por fila}}{\text{Tasa de transferencia}}$$

Pasos de resolución para el Ejercicio de Consultas en una Base de Datos Distribuida

Para resolver el Query descrito anteriormente aplicado a **bases de datos distribuidas** se pueden adoptar distintos caminos, pero **la diferencia entre uno y otro es el tiempo de retraso de comunicación que genera cada opción**, los cuales serán calculados a continuación para ver cuál opción es la mejor:

19. Mover los **datos** de la **tabla repuestos** de la “**región B**” a la “**región A**”, ya que son los únicos que se encuentran en una región distinta. Esta operación se ejecutará en el siguiente tiempo:

$$\text{Retraso de comunicación} = \# \text{mensajes} * \text{Latencia de acceso} + \frac{\# \text{Tuplas} * \text{Bits de peso por fila}}{\text{Tasa de transferencia}}$$

$$\text{Retraso de comunicación} = 1 * 0.1 + \frac{100,000 * 200 \text{ bits}}{50,000 \frac{\text{bits}}{\text{seg}}} = 400 \text{ seg} = 6.67 \text{ mins}$$

20. Mover los **datos** de las **tablas proveedores y PR** (**tabla que une las entidades de proveedores y repuestos**) de la “**región A**” a la “**región B**”, para que todos los datos se encuentren en una misma región y se puedan realizar consultas de forma más sencilla. Esta operación se ejecutará en el siguiente tiempo:

$$\text{Retraso de comunicación} = \# \text{mensajes} * \text{Latencia de acceso} + \frac{\# \text{Tuplas} * \text{Bits de peso por fila}}{\text{Tasa de transferencia}}$$

$$\text{Retraso de comunicación} = 2 * 0.1 + \frac{(10,000 + 1,000,000) * 200 \text{ bits}}{50,000 \frac{\text{bits}}{\text{seg}}} = 4,040 \text{ seg} = 1.12 \text{ hrs}$$

21. Realizar una consulta directa que revise los **repuestos rojos** para cada **proveedor de Bogotá**. Para ello se ejecutaría una consulta **JOIN** entre las **tablas P y PR** que se encuentran en la **región A** que además será **filtrada con el comando WHERE** para obtener solo las **filas** de **proveedores** que tengan el valor **Bogotá** en su columna **Ciudades** para después aplicar a este resultado otro comando **JOIN** que junte los datos extraídos con el contenido de la **tabla R** que se encuentra en la **región B** y posea el valor **Rojo** en su columna **Colores**. Esta operación se ejecutará en el siguiente tiempo:

$$\text{Retraso de comunicación} = \# \text{mensajes} * \text{Latencia de acceso} + \frac{\# \text{Tuplas} * \text{Bits de peso por fila}}{\text{Tasa de transferencia}} = 5.56 \text{ hrs}$$

22. Realizar una consulta directa que por cada **repuesto rojos si hay un proveedor que le corresponde de Bogotá**. Para ello se ejecutaría una consulta en la **tabla R** que se encuentra en la **región B** y se filtrará con el comando **WHERE** las **filas** que posean el valor **Rojo** en su columna **Colores**, luego se realizará una operación **JOIN** entre las **tablas P y PR** que se encuentran en la **región A** que además será **filtrada con el comando WHERE** para obtener solo las **filas** de **proveedores** que tengan el valor **Bogotá** en su columna **Ciudades** para finalmente aplicar entre ambos resultados un comando **JOIN** que combine los **datos** extraídos. Esta operación se ejecutará en el siguiente tiempo:

$$\text{Retraso de comunicación} = \# \text{mensajes} * \text{Latencia de acceso} + \frac{\# \text{Tuplas} * \text{Bits de peso por fila}}{\text{Tasa de transferencia}} = 2 \text{ segundos}$$

23. Realizar una consulta que primero filtre **con el comando WHERE** los **proveedores** que tengan el valor **Bogotá** en la columna **Ciudades** de la **tabla PR** que se encuentra en la **región A** para luego moverlos a la **región B**. Esta operación se ejecutará en el siguiente tiempo:

$$\text{Retraso de comunicación} = \# \text{mensajes} * \text{Latencia de acceso} + \frac{\# \text{Tuplas} * \text{Bits de peso por fila}}{\text{Tasa de transferencia}}$$

$$\text{Retraso de comunicación} = 1 * 0.1 + \frac{100,000 * 200 \text{ bits}}{50,000 \frac{\text{bits}}{\text{seg}}} = 400 \text{ seg} = 6.67 \text{ mins}$$

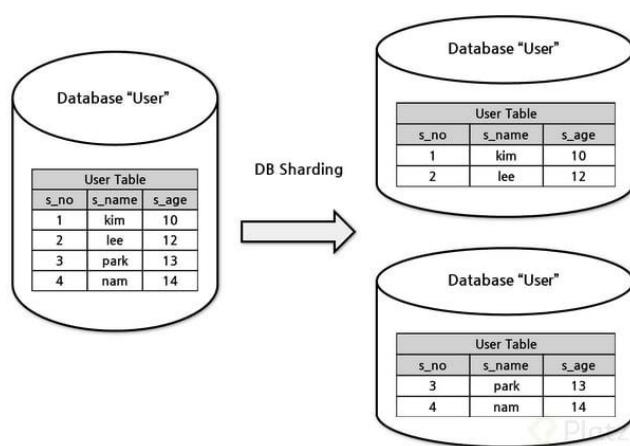
24. Realizar una consulta que primero filtre **con el comando WHERE** los **repuestos** que tengan el valor **Rojo** en la columna **Colores** de la **tabla R** que se encuentra en la **región B** para luego moverlos a la **región A**. Esta operación se ejecutará en el siguiente tiempo:

$$\text{Retraso de comunicación} = \# \text{mensajes} * \text{Latencia de acceso} + \frac{\# \text{Tuplas} * \text{Bits de peso por fila}}{\text{Tasa de transferencia}}$$

$$\text{Retraso de comunicación} = 1 * \mathbf{0.1} + \frac{\mathbf{10 * 200 \text{ bits}}}{\mathbf{50,000 \frac{\text{bits}}{\text{seg}}}} = 0.1 \text{ seg}$$

Sharding (Particiones)

Sharding es una técnica que se utiliza para dividir los **registros** de una base de datos en diferentes **nodos (bases de datos individuales)** pertenecientes a una **base de datos distribuida**. En específico el sharding divide las **filas** de nuestras **tablas** en diferentes **ubicaciones geográficas** de acuerdo con ciertos criterios que nos convengan, ya sea por la inmediatez con la que se necesitan los datos o alguna otra razón. De modo que, al realizar **consultas a nuestra base de datos distribuida**, se tendrán que dirigir al shard o parte que corresponda.



Los principales problemas que ocasiona el abuso del sharding o las particiones de datos es que se complica la consulta de datos de las siguientes formas:

Problemas

- Joins entre shards.
- Baja elasticidad
- Reemplaza PK



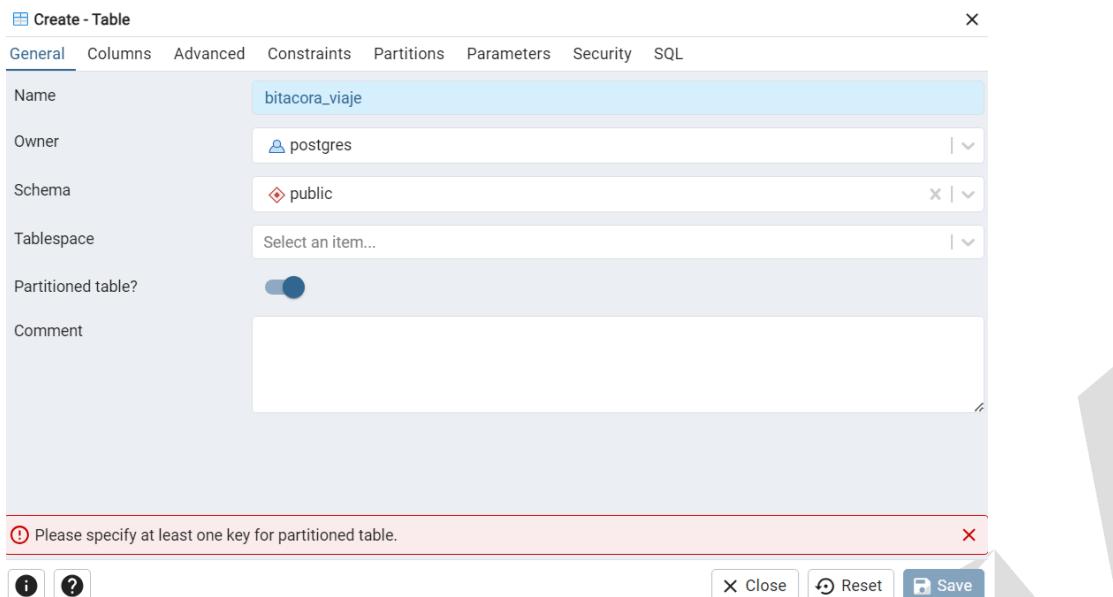
Particiones en PgAdmin de PostgreSQL

En ocasiones, se llega a un punto donde **se tiene mucha información en una sola tabla** y al momento de realizarle consultas a través de SQL, la **base de datos** tendrá que hacer el recorrido de todas sus **filas** de información para encontrar el dato que estamos buscando.

Las **tablas** en los sistemas operativos corresponden a **espacios de memoria** y las **particiones** lo que hacen es **realizar una separación física de los datos** almacenados para **guardar varias partes de la misma tabla** en diferentes espacios de un disco duro (**C:**) o hasta en discos duros distintos (**C:\ o D:**), pero manteniendo la misma estructura lógica que se tenía dentro del **database**, para que estos datos se puedan obtener al ejecutar un mismo SQL Query, aunque se encuentren **particionados**. Un ejemplo de cómo se utiliza esto, es pensar en una **base de datos** que **almacena fechas**; en ella se creará una **partición para los datos de cada mes**, así **cuando se realice una consulta para el mes de junio**, en vez de que se recorran todos los datos del **DB**, la **consulta solamente recorrerá los datos de la partición correspondiente (los datos del mes de junio)**, lo cual acelera enormemente la velocidad de retorno de las consultas y evita que la tabla se pueda tratar cuando contiene muchos datos.

Para exemplificar el uso de las **particiones** vamos a crear una **tabla con particiones** realizadas a través de **ciertos rangos**, le insertaremos datos y luego le haremos una consulta con SQL. Es muy importante mencionar que en las **tablas particionadas** no se permite crear **constraints** de **PRIMARY KEYS**, ya que la información se encuentra dividida.

La **tabla particionada** del ejemplo **contendrá la información histórica de todos los viajes que se han realizado en el sistema** y se llamará **bitacora_viaje**, donde por medio de fechas nos dará toda la información de los viajes. Para realizar las particiones **pgAdmin** cuenta con una pestaña cuando se crean las tablas, pero para ello primero debo seleccionar la opción de: **Tables** → Clic Derecho → Create → **Table...** → **Pestaña General** → **Partitioned Table? = Yes (Radio Button)**.



Al indicar en la pestaña **General** que la **tabla** estará particionada, se nos pedirá que indiquemos una llave para hacer la partición, esto se refiere a la columna o expresión que genera el rango de particiones. Para ello seleccionamos la opción de: **Pestaña Columns** → **+ (Add row)** → Crear una a una los **atributos**

de la **entidad** indicando su **tipo de dato** y **constraints básico** → Pestaña **Partitions** → Partition Type: Range (Así se indica que la partición de realizará a través de un rango) → **Partition Keys** → + (Add row) → Key type: **Column** (De esta manera se declara que el **rango de la partición** se generará a través del valor de un **atributo**) → Pestaña **SQL** (Ver código) → Save (Al guardar una **tabla particionada**, esta aparecerá con un **símbolo distinto a las demás**).

Create - Table

General Columns Advanced Constraints Partitions Parameters Security SQL

Create - Table

General Columns Advanced Constraints Partitions Partitions Parameters Security SQL

Create - Table

General Columns Advanced Constraints Partitions Parameters Security SQL

```

1 CREATE TABLE public.bitacora_viaje
2 (
3     id_bitacora serial,
4     id_viaje integer,
5     fecha_viaje date
6 ) PARTITION BY RANGE (fecha_viaje);
7
8 ALTER TABLE IF EXISTS public.bitacora_viaje
9     OWNER to postgres;

```

Tables (5)

- bitacora_viaje
- estacion
- pasajero
- trayecto
- tren

Si en este momento intentamos insertar datos dentro de la tabla particionada a través un código SQL al seleccionar la opción: **Tables** → **nombreTablaParticionada** → Clic derecho → Scripts → **INSERT Script**. Podremos observar que esto no es posible, porque **todavía no se ha creado la partición (pedazo de memoria) que corresponde a cada rango de datos**.

Para crear las **particiones** de una **tabla particionada**, se debe ejecutar el siguiente código SQL tomando en cuenta que las particiones son en sí una tabla contenida dentro de otra tabla:

```

CREATE TABLE Nombre_Particion      PARTITION OF Nombre_Tabla_Particionada
FOR      VALUES      FROM
(Valor_Inicial_del_Rango_con_Atributo)
TO
(Valor_Final_del_Rango_con_Atributo);

```

Al ejecutar esta línea de código SQL, ya se podrán agregar datos que se encuentren en el rango de valores indicados en la partición creada. Pero si se intenta insertar datos que no se encuentren en el rango indicado, se nos lanzará una excepción y el dato no se podrá añadir.

The screenshot shows two PostgreSQL query editors side-by-side. The left editor contains the following SQL code:

```

1 INSERT INTO public.bitacora_viaje(
2   id_viaje, fecha_viaje)
3   VALUES (1, '2010-01-01');
4
5 CREATE TABLE bitacora_viaje_2010_2019 PARTITION OF bitacora_viaje
6 FOR VALUES FROM ('2010-01-01') TO ('2019-01-01');
7
8 SELECT * FROM bitacora_viaje;

```

The right editor contains the following SQL code:

```

1 INSERT INTO public.bitacora_viaje(
2   id_viaje, fecha_viaje)
3   VALUES (1, '2010-01-01');
4
5 CREATE TABLE bitacora_viaje_2010_2019 PARTITION OF bitacora_viaje
6 FOR VALUES FROM ('2010-01-01') TO ('2019-01-01');
7
8 INSERT INTO public.bitacora_viaje(
9   id_viaje, fecha_viaje)
10  VALUES (1, '2024-01-01');
11
12 SELECT * FROM bitacora_viaje;

```

The output of the right editor shows an error message:

ERROR: La llave de particionamiento de la fila que falla contiene (fecha_viaje) = (2024-01-01), no se encontró una partición de <bitacora_viaje> para el registro
ERROR: no se encontró una partición de <bitacora_viaje> para el registro
SQL state: 23514
Detail: La llave de particionamiento de la fila que falla contiene (fecha_viaje) = (2024-01-01).

Vale la pena mencionar que, para ver las **particiones** creadas en una **tabla**, debemos dar clic en la opción de: **Tables → nombreTablaParticionada → Partitions** → Clic derecho → Refresh.

Y además para ejecutar un código SQL que cree una partición debemos seleccionar la opción: **Tables → nombreTablaParticionada → Partitions** → Clic derecho → Query Tool.

The screenshot shows the pgAdmin interface with a tree view on the left and a query tool on the right.

Tree View (Left):

- Foreign Tables
- Functions
- Materialized Views
- Operators
- Procedures
- Sequences
- Tables (5)
 - bitacora_viaje
 - Columns
 - Constraints
 - Indexes
 - Partitions (2)
 - bitacora_viaje_2010_2019
 - bitacora_viaje_2019_2024
 - RLS Policies
 - Rules
 - Triggers
 - estacion
 - pasajero
 - trayecto
 - tren
 - Trigger Functions
 - Types

Query Tool (Right):

The query tool contains the following SQL code:

```

1 INSERT INTO public.bitacora_viaje(
2   id_viaje, fecha_viaje)
3   VALUES (1, '2010-01-01');
4
5 CREATE TABLE bitacora_viaje_2010_2019 PARTITION OF bitacora_viaje
6 FOR VALUES FROM ('2010-01-01') TO ('2019-01-01');
7
8 CREATE TABLE bitacora_viaje_2019_2024 PARTITION OF bitacora_viaje
9 FOR VALUES FROM ('2019-01-01') TO ('2024-01-01');
10
11 INSERT INTO public.bitacora_viaje(
12   id_viaje, fecha_viaje)
13   VALUES (1, '2023-01-01');
14
15 SELECT * FROM bitacora_viaje;

```

The data output shows the following table:

	id_bitacora	id_viaje	fecha_viaje
1		2	2010-01-01
2		10	2023-01-01

Window Functions

Las ventanas o Windows son un conjunto de **filas** pertenecientes a una **tabla** que se encuentran **agrupadas (particionadas)** con el comando **PARTITION BY** y **ordenadas por el método ORDER BY** dentro de la función **OVER()**. Las Window Functions o funciones de ventana nos permiten realizar cálculos como **sumas acumulativas, promedios, etc.** sobre dichas ventanas, a través de esta sintaxis:

Función_de_ventana() OVER(PARTITION BY columna_de_agrupacion ORDER BY columna_de_ordenamiento)

- **Funciones de Ventana:** Son las operaciones que se pueden realizar sobre un conjunto de filas agrupadas (particionadas) y/o ordenadas por medio del comando **OVER()**.
 - **ROW_NUMBER():** Window function que asigna un **número secuencial único** a cada **fila** dentro de la **partición** de una ventana, comenzando con 1.
 - **RANK():** Window function que asigna un **número de rango** a cada **tupla** dentro de las **particiones** de una ventana. Este rango indica la posición de cada **fila** en relación con las demás en función de su valor (**las de mayor valor, tendrán menor rango y viceversa**) y si dos o más de ellas tienen el mismo valor, reciben el mismo rango. **RANK()** asigna rangos con posibles huecos, ya que respeta el orden del **id**, por lo que puede brincar de un rango 1 a 5.
 - **DENSE_RANK():** Window function que asigna un **número de rango** a cada **tupla** dentro de las **particiones** de una ventana. Este rango indica la posición de cada **fila** en relación con las demás en función de su valor (**las de mayor valor, tendrán menor rango y viceversa**) y si dos o más de ellas tienen el mismo valor, reciben el mismo rango. **DENSE_RANK()** asigna rangos sin huecos, sin respetar el orden del **id**, por lo que los rangos serán consecutivos de 1, 2, 3, etc.
 - **NTILE(n):** Window function que divide las **filas de datos** en n grupos.
 - **SUM(), AVG(), MIN(), MAX(), COUNT():** Window functions que realizan operaciones matemáticas a las particiones de **datos**.
 - **FIRST_VALUE(), LAST_VALUE():** Window functions que devuelven la primera o última **fila** en la ventana.
 - **LAG(), LEAD():** Window functions que devuelven el valor de una **columna** desde una **fila** anterior o posterior a la **tupla** actual en la ventana.
 - **CUME_DIST(), PERCENT_RANK():** Window functions que calculan las distribuciones y rangos porcentuales en las **tuplas** de una ventana.
- **OVER():** Comando que define la agrupación y ordenamiento del conjunto de filas que conforman la ventana a la cual se aplicará una Window Function.
 - **PARTITION BY:** Comando que **crea una partición (agrupación)** de **filas** pertenecientes a una **tabla** a través de cierta **columna de agrupación**.
 - **ORDER BY:** Comando de agregación que **ordena las filas** pertenecientes a una **tabla** a través de cierta **columna de ordenamiento** de forma **ascendente (de menos a más viéndolos de arriba hacia abajo)** en función del valor de cierto **atributo** con el comando **ASC** o de forma descendente **(de más a menos)** a través del comando **DESC**.

Las **Window Functions (Funciones de Ventana)** nos permiten ordenar la forma en la que se muestran nuestros **registros** de datos o realizar cálculos de **posición, promedio, suma, etc.** sobre un **grupo de tuplas agrupadas y ordenadas**.

Su mayor utilidad se denota cuando se quiere evitar el uso de operadores **SELF JOIN**, los cuales se utilizan para poder realizar operaciones **JOIN** a través de **diferentes instancias o copias de una misma tabla**, en vez de hacerlo con más de una **entidad**, logrando así obtener la **intersección, unión, etc.** de sus **columnas**. Para ello se debe asignar un alias a cada **instancia** con el comando **AS** y a través de dichos alias se realizan las operaciones lógicas de los Diagramas de Venn, según el resultado que se quiera obtener con el comando **ON** y las **columnas** que se busque **relacionar** entre las **copias de la tabla**.

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/>

