

INGENIERÍA MECATRÓNICA



DI_CERO

DIEGO CERVANTES RODRÍGUEZ

PROGRAMACIÓN: DESARROLLO BACKEND

SQL

PgSQL, Triggers, Extensiones,
Transacciones, Backups,
Mantenimiento y Réplicas

Contenido

Representación de las Bases de Datos: Nomenclatura de Chen	2
Lenguaje de Programación SQL - PostgreSQL	2
PL/PgSQL - Procedural Language/Postgres SQL 🐘	2
Triggers: Código PL/PgSQL ejecutado automáticamente en tablas	7
Librerías Externas de PostgreSQL 🐘	12
Conexión a bases de datos remotas e importación de datos: DBLink	12
Importación y uso de librerías externas de PostgreSQL 🐘: DBLink	12
Uso de la API de extensiones o librerías de PostgreSQL 🐘	15
Importación y uso de librerías externas de PostgreSQL 🐘: Levenstein y Difference	15
Transacciones	16
Manejo de Excepciones	19
Backups y Restauraciones	20
Mantenimiento	23
Réplicas	25
Ejemplo Réplicas: Jelastic & Cloudjiffy	25
Código SQL - Creación y/o Modificación de la Base de Datos (DDL y DML)	25
Referencias	25



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 a las **DB**:

- **Servidor de base de datos:** Es un computador 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 servicios encargados 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 de base de datos ORM (Object Related Mapping) que guarda la relación de las entidades (**tablas**, **funciones**, **relaciones**, **secuencias**, etc).

PL/PgSQL - Procedural Language/Postgres SQL

Las **PL (Procedural Language)**, también llamados **Procedimientos Almacenados** o **PgSQL (PostgreSQL Structured Query Language)** también son parte del motor de **PostgreSQL** y es una de las razones más importantes por las cuales conviene utilizar **Postgres** antes que otro motor de **bases de datos**, ya que

este nos permite a **desarrollar código directamente sobre la database** a través de una variante de SQL, que se comporta de forma muy similar a una **función de código**, donde se declara el **nombre de la función**, se hace uso de **variables**, se describe un código de ejecución y se finaliza con un **retorno de valores o no**. Los códigos **PL** se conforman de las siguientes partes:

- **DO:** Esta instrucción se utiliza para funciones anónimas (sin nombre) e indica la ejecución de un bloque de **código PL**, dicha ejecución se abre y cierra por medio de dos símbolos de peso **\$\$**.
 - **CREATE FUNCTION nombre_funcion():** Si se quiere declarar una **función con nombre** para que pueda ser utilizada después, se debe utilizar el comando **CREATE** o **CREATE OR REPLACE** de tipo **DDL (Data Definition Language)** SQL seguido de la palabra reservada **FUNCTION** y el **nombre de la función** acompañado de dos símbolos de paréntesis, después se debe dar un salto de línea y tabulador para colocar la instrucción **RETURNS** seguida del **tipo de dato que retorna la función**, si no devuelve nada, se indica como **void** y finalmente se declara la palabra reservada seguida del símbolo **AS** con doble símbolo de pesos **\$\$** para indicar el inicio de la función.
 - Si indicamos dentro de la instrucción **RETURNS** que se devuelva un valor, este aparecerá en la pestaña inferior llamada **Data Output** de **pgAdmin**.
 - **DROP FUNCTION nombre_funcion():** Si ocurre un problema al haber primero creado la función y luego modificado su código cuando se intente ejecutar, esta se debe borrar con el comando **DROP** y luego volver a ejecutar su código de creación.
- **DECLARE:** Este comando se debe declarar después de la instrucción **DO \$\$**, pero antes de la instrucción **BEGIN** e indica la declaración de una variable que se puede utilizar dentro de un **código PL**. Las variables no se declaran justo después del comando, sino abajo y después de añadir un tabulador, se debe indicar el **nombre de la variable**, el **tipo de dato** que almacena (**si guarda el valor de las filas de una consulta debe ser de tipo record**), si se le quiere dar un valor inicial a la variable se usa el símbolo **(:=)** y terminar con un punto y coma **(;)**.
- **BEGIN/END:** La instrucción **BEGIN** indica el inicio de la declaración de un **código PL** y el comando **END** indica su final, pero para que el código se pueda ejecutar todo debe estar contenido dentro de las instrucciones **DO \$\$** y **\$\$**. La declaración de variables se realiza con el comando **DECLARE**, el cual se coloca entre **DO \$\$** y **BEGIN**.
 - **RETURN:** Este comando se debe utilizar antes del comando **END** junto con el **nombre de la variable** que se quiere devolver al finalizar la función, siempre y cuando al declararse, se haya indicado el **tipo de valor** que se quiera retornar y la **variable indicada** tenga ese mismo valor.
- **Bucle For:** En el **código PL** se utiliza para recorrer y obtener el valor de las **columnas** de una **fila** obtenida a través de una consulta SQL que se guarda en una **variable** previamente inicializada con el método **DECLARE** y tiene la siguiente sintaxis, empezando con la palabra reservada **FOR IN LOOP** y terminando con la instrucción **END LOOP;**:
 - **FOR nombre_variable IN Consulta_SQL LOOP**

 --Código PL que se quiere ejecutar con los valores obtenidos de la consulta SQL.

 END LOOP;

- **RAISE NOTICE:** Es un comando similar a **console.log** de los demás lenguajes de programación, el cual sirve para **mostrar el resultado de una variable en consola**. Si se quiere concatenar texto

con una variable (la cual debe haber sido previamente inicializada con el comando **DECLARE**), se debe utilizar una coma para separar el texto de la variable a concatenar y usar el símbolo de porcentaje **%** para que en ese punto del texto se coloque el valor de la variable.

- Si indicamos dentro de la instrucción **RAISE NOTICE** que se muestre un valor en consola, este aparecerá en la pestaña inferior llamada **Messages** de **pgAdmin**.
- **LANGUAGE**: Después de los símbolos dobles de peso **\$\$** que indican el final de la ejecución de un código PL, se debe utilizar el comando **LANGUAGE** para indicar el lenguaje que se utilizó, ya que dentro de **PostgreSQL** se permite utilizar no solo el lenguaje **PL/PgSQL**, sino también se puede utilizar **SQL** (PL/PgSQL limitado), **Python** o **C++** al añadir ciertas librerías a **pgAdmin**.

--Función anónima PL/PgSQL:

DO \$\$

DECLARE

nombre_variable tipo_de_dato := valor_inicial;

BEGIN

FOR **nombre_variable** **IN** **Consulta_SQL** **LOOP**

--Código PL que se quiere ejecutar con los valores obtenidos de la consulta SQL.

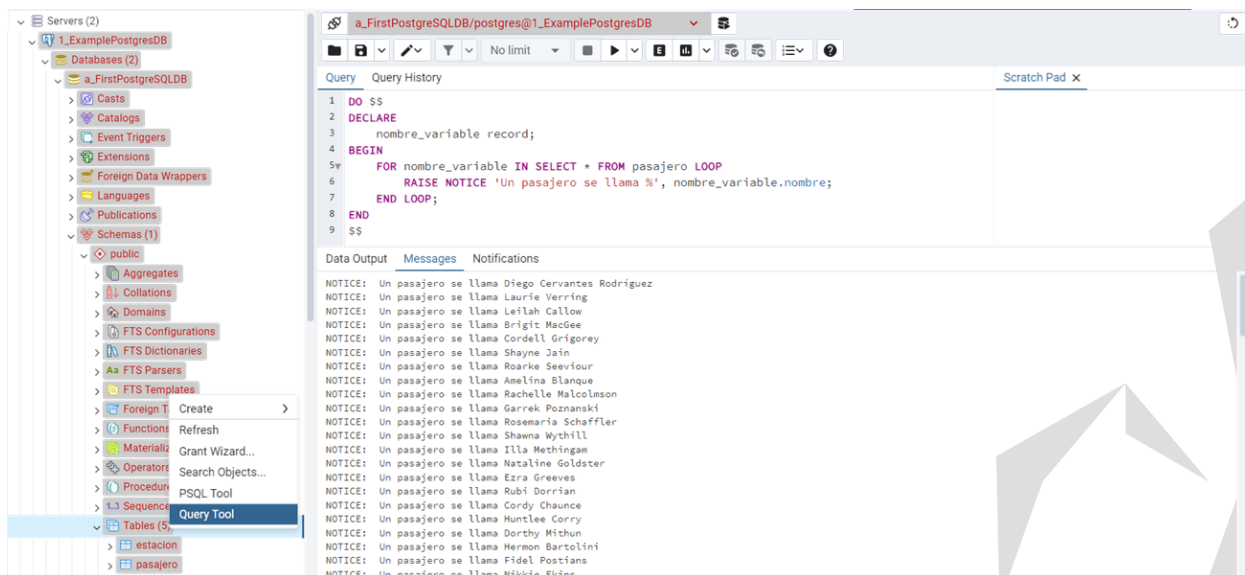
RAISE NOTICE 'Texto para mostrar en consola %', **nombre_variable.atributo_consulta**;

END LOOP;

END

\$\$

Para ejecutar los códigos **PL/PgSQL** se debe seleccionar la opción de: **Servers** → **Motor de PostgreSQL** → **Databases** → **nombre_base_de_datos** → **Schemas** → **public** → **Tables** → Clic derecho → **Query Tool** → Introducir código **PL/PgSQL** → Dar clic en botón de **Execute script**.



--Función con nombre PL/PgSQL:

CREATE OR REPLACE FUNCTION nombre_funcion_PL/PgSQL()

RETURNS tipo_de_dato

AS \$\$

DECLARE

nombre_variable tipo_de_dato := valor_inicial;

BEGIN

FOR nombre_variable **IN** Consulta_SQL **LOOP**

--Código PL que se quiere ejecutar con los valores obtenidos de la consulta SQL.

RAISE NOTICE 'Texto para mostrar en consola %', nombre_variable.atributo_consulta;

END LOOP;

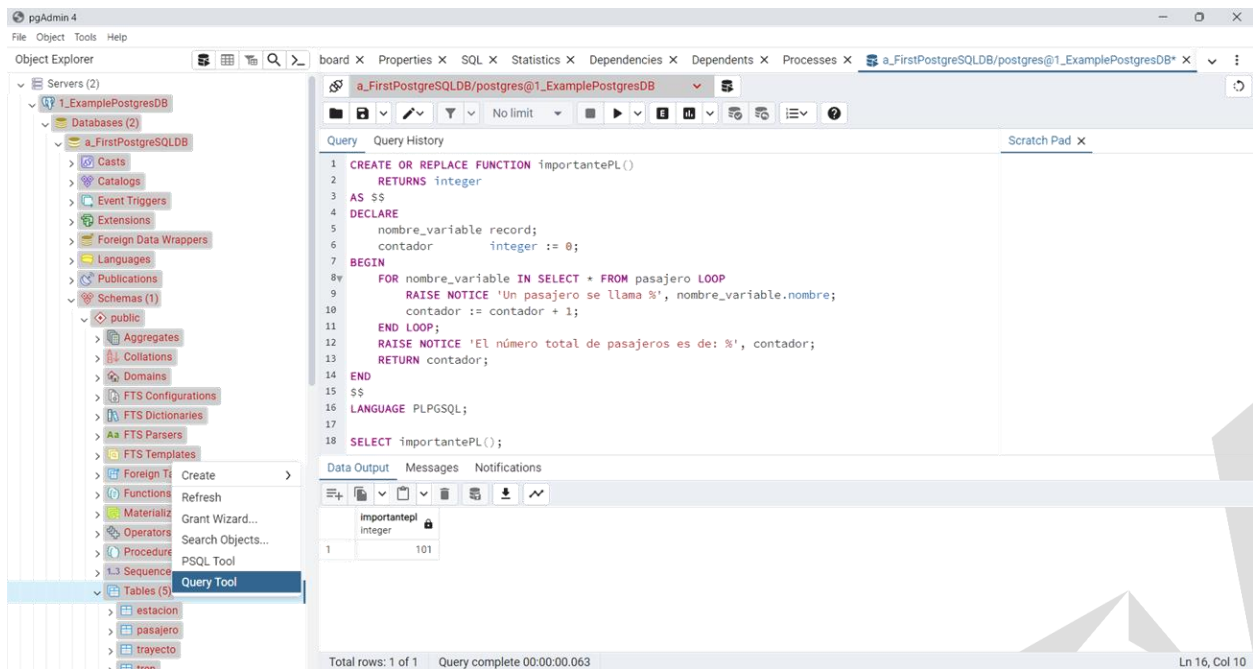
END

\$\$

LANGUAGE 'PLPGSQL, SQL, PYTHON o C++'

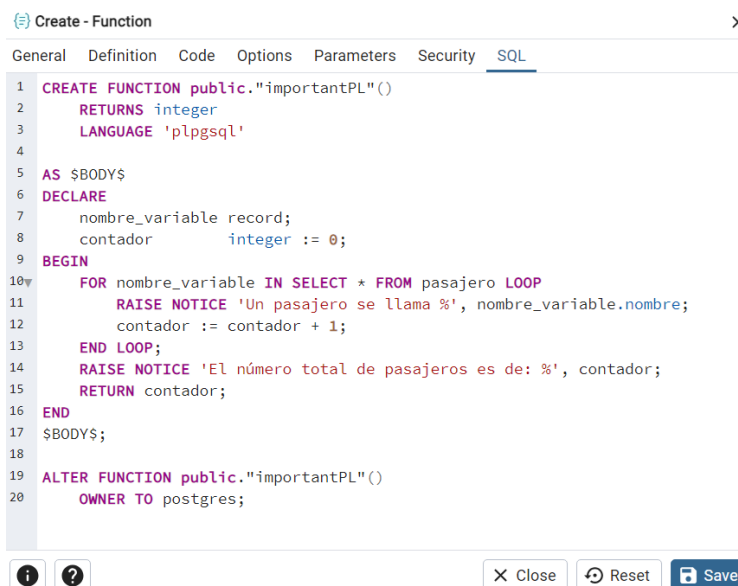
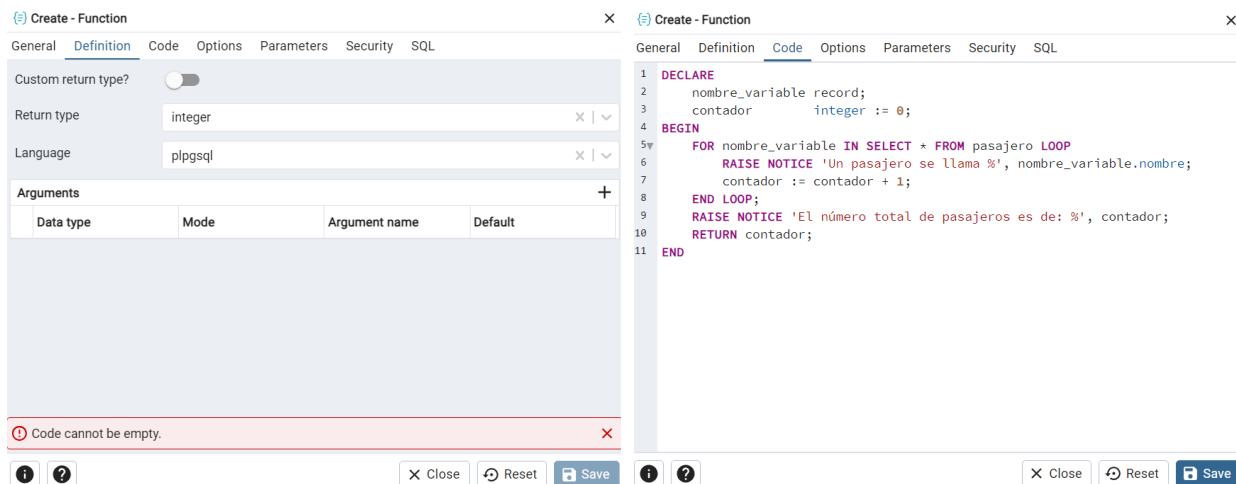
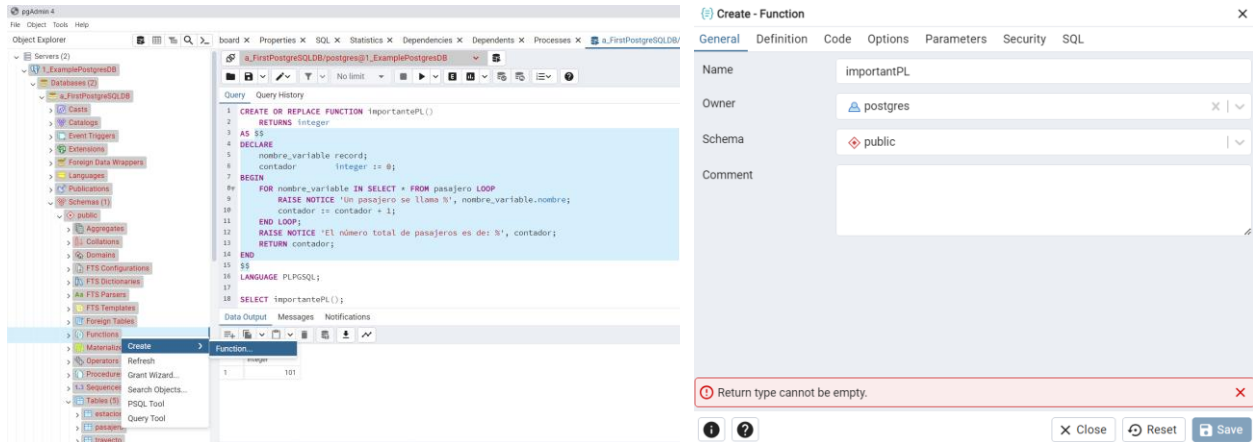
--Código de ejecución de una función PL/PgSQL:

SELECT nombre_funcion();

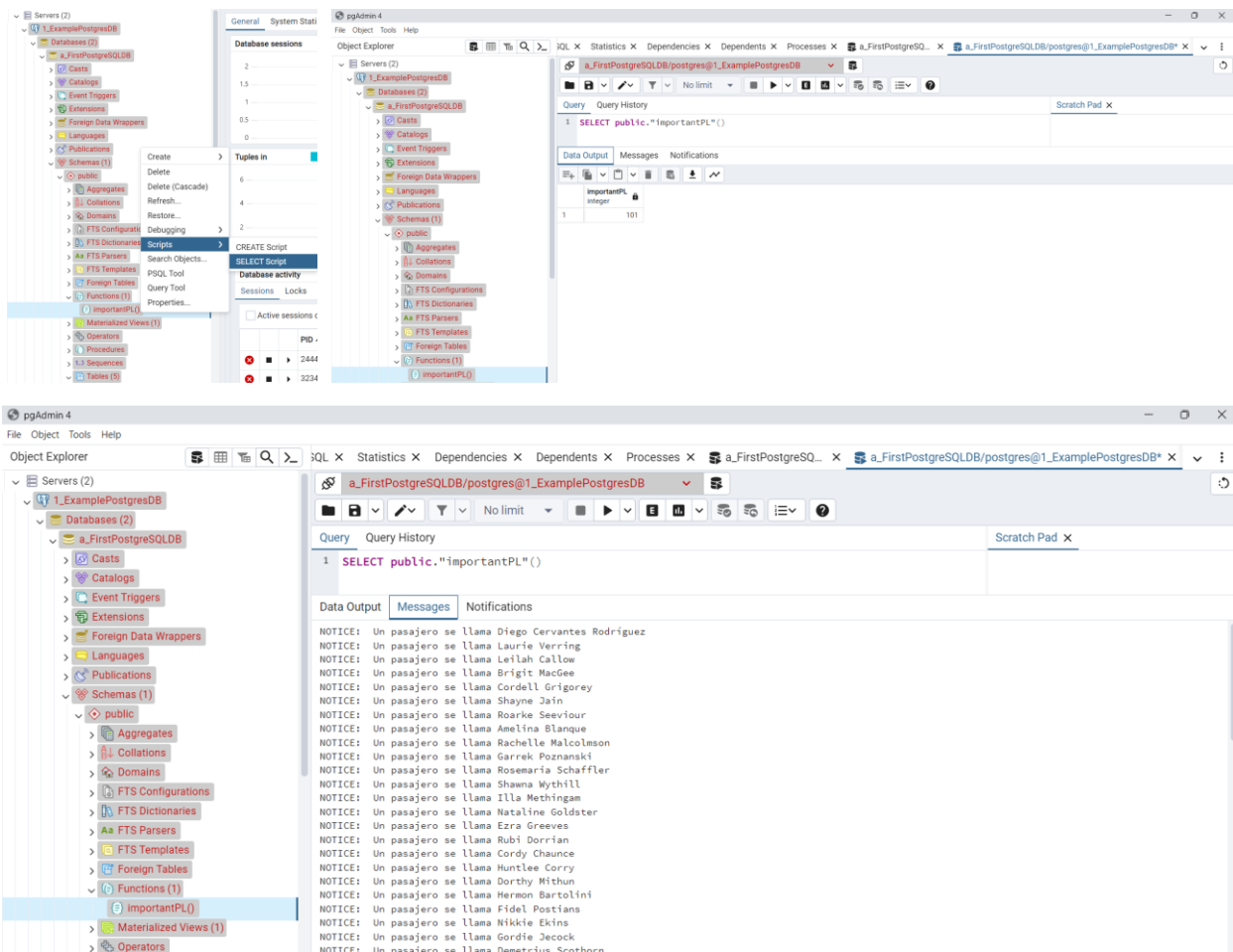


Cuando se quiera guardar una **función** dentro de **pgAdmin**, lo que se debe hacer es seleccionar la opción de: **Servers** → **Motor de PostgreSQL** → **Databases** → **nombre_base_de_datos** → **Schemas** → **public** →

Functions → Clic derecho → Create → Function... → **Pestaña General** → Name: **nombre_función** → **Pestaña Definition** → Return type: **tipo_de_dato_variable_return** → Language: **Lenguaje de Programación** → **Pestaña Code** → Código **PL/PgSQL** que se encuentra entre los símbolos de **\$\$** o **\$BODY\$** → **Pestaña SQL** → Ver código SQL → Save.



Para ejecutar la **función PL/PgSQL** creada se debe seleccionar la opción de: **Servers** → **Motor de PostgreSQL** → **Databases** → **nombre_base_de_datos** → **Schemas** → **public** → **Functions** → **nombre función** → Clic derecho → **Scripts** → **SELECT Script** → Dar clic en botón de **Execute script**.



Triggers: Código PL/PgSQL ejecutado automáticamente en tablas

Los **triggers** (también conocidos como **disparadores**) son **funciones** de código **PL/PgSQL** que se aplican de forma automática sobre una **tabla** cuando en ella se ejecuten ciertos comandos SQL de tipo **DML** (**Data Manipulation Language**), los cuales pueden ser:

- **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**.

Para ello antes que nada se debe crear y guardar en **pgAdmin** alguna **función especial PL/PgSQL** que se pueda ejecutar con un **trigger**, a esta se le deben hacer configuraciones especiales, como que el **tipo de**

dato indicado en el comando **RETURNS** sea **trigger** (indicado o no en mayúsculas) y después en la instrucción **RETURN** se utilice alguna de las variables **OLD** o **NEW**, las cuales sirven para lo siguiente:

- **RETURN:** Cuando la **función PL** se vaya a ejecutar a través de un **trigger** se deben utilizar alguna de las dos siguientes variantes en este comando:
 - **OLD:** Indica que el tipo de dato **TRIGGER** debe **devolver el mismo dato que se encontraba en la fila** antes de aplicar el cambio hecho por la **función PL**. En términos simples, **NO permite el cambio de datos** en la **tabla**, ejecutados por el **trigger**.
 - **NEW:** Indica que el tipo de dato **TRIGGER** debe **devolver el dato actualizado o modificado** después de aplicar el cambio hecho por la **función PL**. En términos simples, **permite la modificación de los datos** en la **tabla**, ejecutados por el **trigger**.

--Función con nombre PL/PgSQL:

```
CREATE FUNCTION nombre_funcion_PL_trigger()

    RETURNS    TRIGGER

AS $$

DECLARE

    nombre_variable tipo_de_dato := valor_inicial;

BEGIN

    FOR    nombre_variable IN    Consulta_SQL    LOOP

        --Código PL que se quiere ejecutar con los valores obtenidos de la consulta SQL.

        RAISE NOTICE 'Texto para mostrar en consola %', nombre_variable.atributo_consulta;

    END LOOP;

    --Justo antes del fin del código, vale la pena introducir, actualizar o borrar los
    --resultados del código en una tabla nueva, la cual es distinta a la tabla que
    --acciona la ejecución del trigger.

    --Para ello se pueden usar los comandos INSERT, UPDATE o DELETE.

    INSERT INTO    Nombre_Entidad_Resultados(Atributo_1, ..., Columna_n)

    VALUES("Valor_Atributo_1", ..., "Valor_Columna_n");

    UPDATE Nombre_Tabla_Resultados

    SET    "Columna_1" = "Valor_Columna_1", "Atributo_2" = "Valor_Atributo_2"

    WHERE "Nombre_Atributo_o_Columna" = "Valor_Fila_o_Identificador";
```

```
DELETE FROM "Nombre_Entidad_o_Tabla"

WHERE "Nombre_Atributo_o_Columna" = "Valor_Fila_o_Identificador";
```

END

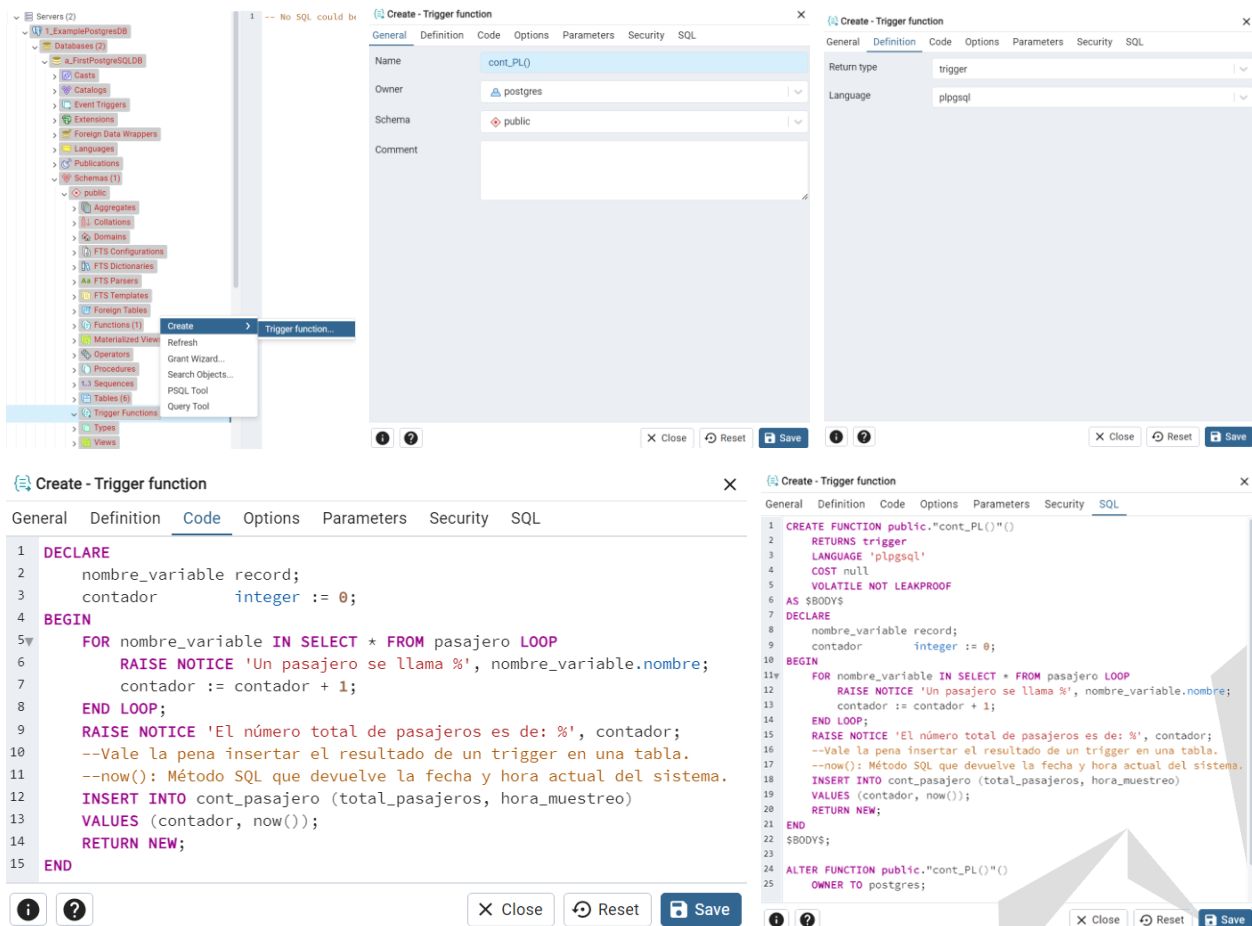
\$\$

LANGUAGE 'PLPGSQL, SQL, PYTHON o C++'

--Código de ejecución de una función PL/PgSQL:

```
SELECT nombre_funcion_PL_trigger();
```

Para crear una **función especial PL/PgSQL** que se pueda ejecutar con un **trigger** con el fin de cambiar automáticamente el estado de una **tabla** al detectar que se le aplica un comando **INSERT**, **UPDATE** o **DELETE**, se debe seleccionar la opción de: **Servers** → **Motor de PostgreSQL** → **Databases** → **nombre_base_de_datos** → **Schemas** → **public** → **Trigger Functions** → Clic derecho → **Create** → **Trigger function...** → **Pestaña General** → **Name:** **nombre_función** → **Pestaña Definition** → **Return type:** **trigger** → **Language:** **Lenguaje de Programación** → **Pestaña Code** → Código **PL/PgSQL** que se encuentra entre los símbolos de **\$\$** o **\$BODY\$** → **Pestaña SQL** → Ver código SQL → **Save**.



Como vale la pena que los resultados del código sean guardados, se creará una **nueva tabla** para almacenarlos, la cual debe tener su propio **atributo id** y cuando se realice esto, dentro de la **función trigger PL/PgSQL** se debió haber utilizado algún comando **INSERT**, **UPDATE** o **DELETE** para modificar dicha **entidad nueva**. Para ejemplificar el uso de los **triggers**, se creó una **nueva tabla** llamada **cont_pasajero**, la cual sirve para guardar el **número actual de pasajeros** y la **fecha/hora a la que corresponde el conteo**, siempre que se ejecute el comando **INSERT**, la **función cont_PL()** hecha con **código PL** se ejecutará automáticamente, obteniendo así el **número actual de pasajeros** y la **fecha/hora donde se recabó el dato**; para finalmente almacenar su resultado en la tabla **cont_pasajero**.

The screenshot shows two windows in a database IDE. The left window, titled 'Create - Table', displays the SQL code for creating a table named 'public.cont_pasajero'. The code includes columns for 'id_conteo' (serial), 'total_pasajeros' (integer), and 'horaFecha_muestreo' (time with time zone), along with a primary key constraint on 'id_conteo'. The right window, titled 'Create - Trigger function', shows the code for a PL/pgSQL trigger function named 'cont_PL()'. This function is designed to be executed after an INSERT operation on the 'pasajero' table. It declares a record variable, loops through the 'pasajero' table to count the number of passengers, and then inserts this count along with the current timestamp into the 'cont_pasajero' table.

```

1 CREATE TABLE public.cont_pasajero
2 (
3     id_conteo serial,
4     total_pasajeros integer,
5     "horaFecha_muestreo" time with time zone,
6     CONSTRAINT conteo_pasajero_pkey PRIMARY KEY (id_conteo)
7 );
8
9 ALTER TABLE IF EXISTS public.cont_pasajero
10 OWNER to postgres;

1 CREATE FUNCTION public."cont_PL()"()
2 RETURNS trigger
3 LANGUAGE 'plpgsql'
4 COST null
5 VOLATILE NOT LEAKPROOF
6 AS $BODY$
7 DECLARE
8     nombre_variable record;
9     contador integer := 0;
10 BEGIN
11     FOR nombre_variable IN SELECT * FROM pasajero LOOP
12         RAISE NOTICE 'Un pasajero se llama %', nombre_variable.nombre;
13         contador := contador + 1;
14     END LOOP;
15     RAISE NOTICE 'El número total de pasajeros es de: %', contador;
16     --Vale la pena insertar el resultado de un trigger en una tabla.
17     --now(): Método SQL que devuelve la fecha y hora actual del sistema.
18     INSERT INTO cont_pasajero (total_pasajeros, hora_muestreo)
19     VALUES (contador, now());
20     RETURN NEW;
21 END
22 $BODY$;
23
24 ALTER FUNCTION public."cont_PL()"()
25 OWNER TO postgres;

```

El código que se debe ejecutar para crear un **trigger** es el siguiente:

- **BEFORE, AFTER o INSTEAD OF:** Estas palabras reservadas indican el momento en el que queremos se ejecute un **trigger**, al percibir que se ha ejecutado un comando **INSERT**, **UPDATE** o **DELETE** sobre una **tabla** en específico.
 - **BEFORE:** Tiempo utilizado cuando queremos que el **trigger** se ejecute **antes** de algún comando **INSERT**, **UPDATE** o **DELETE** detectado.
 - **AFTER:** Tiempo utilizado cuando queremos que el **trigger** se ejecute **después** de algún comando **INSERT**, **UPDATE** o **DELETE** detectado.
 - **INSTEAD OF:** Tiempo utilizado cuando queremos que el **trigger** se ejecute **en lugar de** algún comando **INSERT**, **UPDATE** o **DELETE** detectado.
- **ON Nombre_Tabla_Trigger:** Recordemos que el **trigger** se ejecuta automáticamente cuando se detecte que se corre un comando **INSERT**, **UPDATE**, o **DELETE** sobre una **tabla** en específico, en esta parte del código es donde se indica a qué tabla se aplica dicha acción y normalmente dentro del mismo **trigger**, se guardan los resultados de la función **PL/PgSQL** dentro de una **nueva tabla adicional**.

```

CREATE      TRIGGER      nombre_trigger

AFTER      comando_INSERT,UPDATE_o_DELETE_a_detectar

ON          Nombre_Tabla_Trigger

```



FOR EACH ROW

EXECUTE PROCEDURE nombre_funcion_PL/PgSQL();

Y finalmente crear el **trigger** con el fin de cambiar automáticamente el estado de una **tabla** al detectar que se le aplica un comando **INSERT**, **UPDATE** o **DELETE**, se debe seleccionar la opción de: **Servers** → **Motor de PostgreSQL** → **Databases** → **nombre_base_de_datos** → **Schemas** → **public** → **Tables** → Clic derecho → **Query Tool** → Añadir y seleccionar el código de creación del **trigger** → Dar clic en botón de **Execute script** → Observar que en la **tabla monitoreada** aparece el nuevo **trigger** creado → Al **ejecutar la acción** descrita en el **trigger**, se **añaden datos** a la **nueva tabla creada para almacenar sus resultados**.

The screenshot shows the DBeaver interface with three main panels illustrating the process of creating and using a trigger.

Top Left Panel (Database Explorer): Shows the tree structure of the database. The 'public' schema is selected, and the 'Query Tool' is highlighted in the context menu.

Top Right Panel (Query Editor): Displays the SQL code for creating a trigger:

```
1 CREATE TRIGGER trigContPasajeros
2 AFTER INSERT
3 ON pasajero
4 FOR EACH ROW
5 EXECUTE PROCEDURE public."cont_PL"();
```

Bottom Left Panel (Query Editor): Shows the SQL code for inserting data into the 'pasajero' table:

```
1 INSERT INTO public.pasajero(
2 nombre, direccion,residencia, fecha_nacimiento)
3 VALUES ('Nombre trigger', 'Dirección trigger', '2000-01-01');
```

Bottom Right Panel (Query Results): Shows the results of a query executed on the 'cont_pasajero' table:

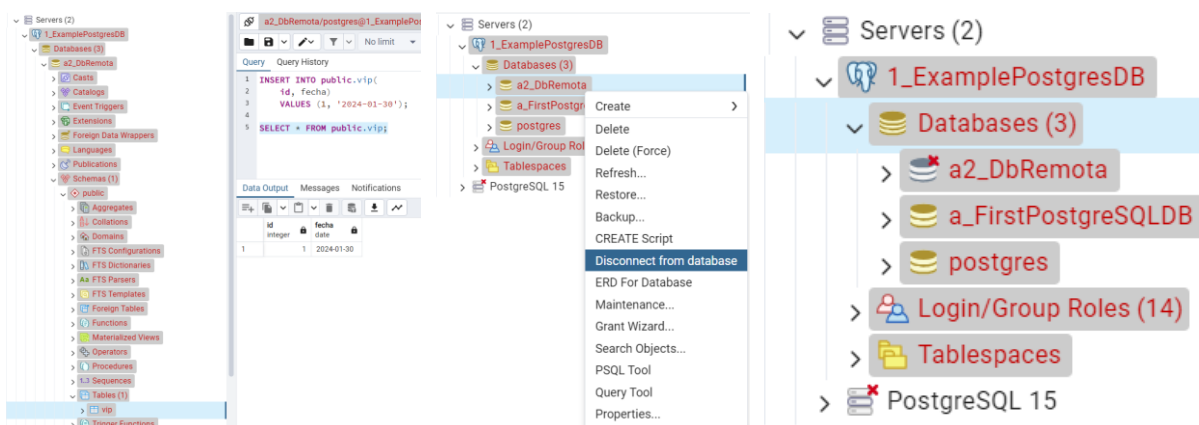
id_conteo	total_pasajeros	hora_muestreo
1	6	04:07:42.434648+00:00
2	9	04:09:56.571357+00:00
3	11	05:22:56.323118+00:00
4	12	05:23:03.183615+00:00

Librerías Externas de PostgreSQL

Conexión a bases de datos remotas e importación de datos: Dblink

PostgreSQL ofrece la herramienta Dblink que nos permite conectarnos a **bases de datos remotas** para así poder importar datos dentro de nuestra **database** a través de una consulta SQL.

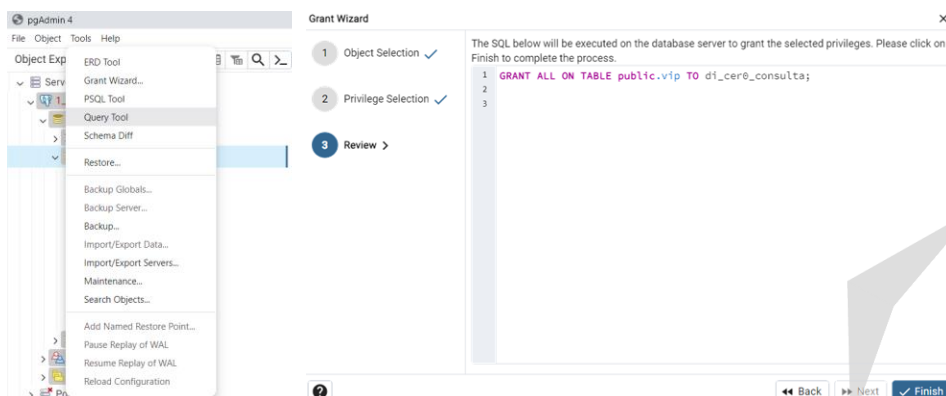
Para ejemplificar esto podemos crear una **DB** nueva dentro de nuestro propio **servidor** y **motor PostgreSQL** para simular como si fuera una **base de datos remota**. La llamaremos **a2_DbRemota** y dentro de ella crearemos una **tabla** llamada **vip**, la cual tendrá los **atributos id** y **fecha**, esta contendrá una fila de datos y nos conectaremos a ella a través de nuestra base de datos **a1_FirtsPostgreSQLDB**, para ello nos debemos desconectar de la base de datos **a2_DbRemota** al darle clic derecho y seleccionar la opción de Disconnect from database después de haberla creado y llenado de datos.



Importación y uso de librerías externas de PostgreSQL : Dblink

Como la herramienta **Dblink** es adicional al paquete por defecto de **PostgreSQL** se debe importar a través del siguiente código **CREATE EXTENSION**, el cual se debe agregar al inicio de la consulta SQL con la que se busca importar datos de una **database externa**. Para poder ejecutar el Query debemos ingresar a la opción del menú superior: Tools → Query Tool, para ello igual debemos dar **accesos** a un **usuario** a la **DB**. Todas las extensiones disponibles se encuentran en este enlace:

<https://www.postgresql.org/docs/11/contrib.html>



Y luego el código que se debe ejecutar en la consulta con el comando **SELECT** no se referirá a una tabla, sino a una instancia de la herramienta **DBLink**, la cual recibe los siguientes parámetros:

- **dblink()**: Es similar a un método constructor, el cual recibe los siguientes parámetros para realizar una conexión a una base de datos remota, de la cual puede extraer información a través de una consulta **SELECT**:
 - **dbname = nombre_database_remota**: Esta parte del 1er parámetro llamada **dbname** recibe el nombre de la base de datos remota a la que nos queremos conectar a través de la librería **DBLink**.
 - **port = puerto_de_conexión_de_la_database_remota**: Esta parte del 1er parámetro llamada **port** recibe el puerto de conexión de la **base de datos remota**, este suele ser el 5432.
 - **host = dirección_IP**: Esta parte del 1er parámetro llamada **host** recibe la dirección IP o el nombre de dominio de donde está alojada la **DB remota**, si esta se encuentra en **localhost** (nuestro propio ordenador), la dirección IP será **127.0.0.1**.
 - **user = nombre_usuario_db_remota**: Esta parte del 1er parámetro llamada **user** indica el **nombre de usuario** de la **database remota** con el que nos queremos conectar a ella.
 - **password = contraseña_usuario_db_remota**: El parámetro **password** indica la contraseña del **usuario** con el que nos queremos conectar a la **DB remota**.
- Este primer ejemplo solo sirve para mostrar cómo importar los datos de la **base de datos remota**.

--Esta primera línea se deberá comentar después de ejecutar una vez el código.

```
CREATE EXTENSION nombre_librería;
```

```
SELECT * FROM
```

--El 1er parámetro de la herramienta DBLink configura la conexión con la DB remota:

```
dblink ('dbname = nombre_database_remota
port = puerto_de_conexión_de_la_database_remota
host = dirección_IP
user = nombre_usuario_db_remota
password = contraseña_usuario_db_remota',
```

--El 2do parámetro ejecuta una Query sencilla que traiga los datos de la db

--remota, después se debe usar la instrucción AS para asignarle un nombre a

--dichos datos y en su paréntesis se debe indicar los nombres de los atributos

--que trae la tabla importada y el tipo de dato correspondiente a cada una:

```
'SELECT Atr_1, ..., Atr_n FROM Nombre_Tabla_Importada')
```

```
AS Nuevo_Nombre_Datos_Importados(Atr_1 tipo_dato,..., Atr_n tipo_dato)
```

- Este segundo ejemplo muestra la sintaxis para poder utilizar los datos importados de la **base de datos remota** con los propios y ejecutar una consulta que utilice el comando **JOIN**.

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

```
CREATE EXTENSION nombre_librería;

SELECT Atr_1, ..., Atr_n FROM Nombre_Tabla_Izq

JOIN

dblink ('dbname = nombre_database_remota

port = puerto_de_conexión_de_la_database_remota

host = dirección_IP

user = nombre_usuario_db_remota

password = contraseña_usuario_db_remota',

'SELECT Atr_1, ..., Atr_n FROM Nombre_Tabla_Importada')

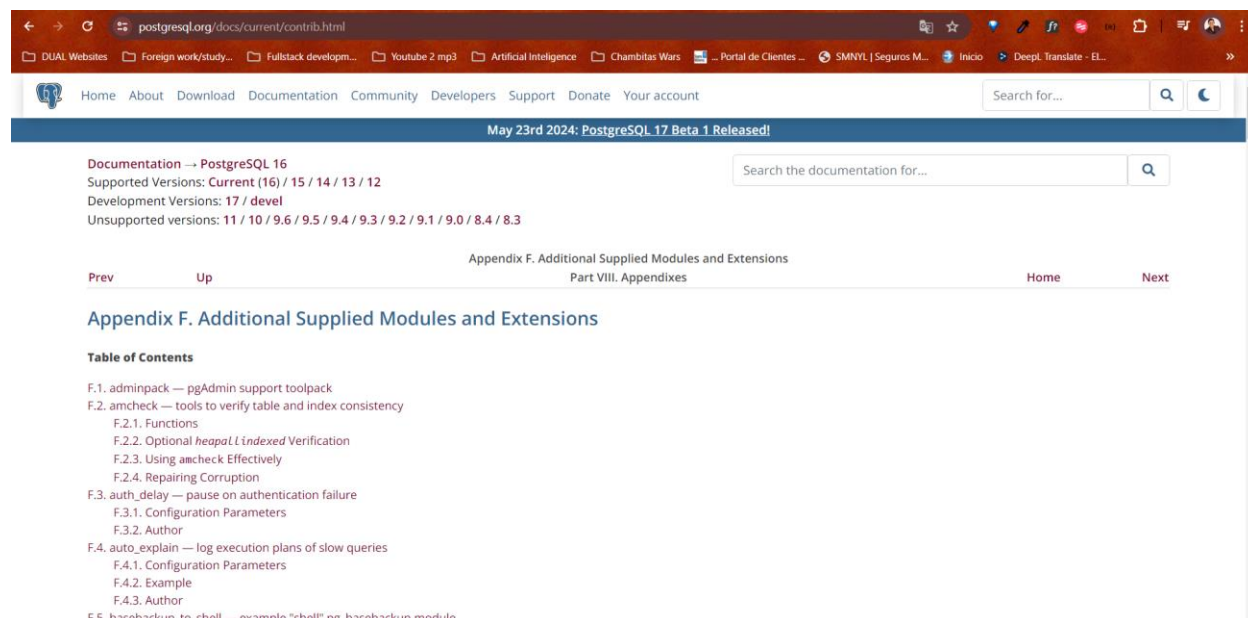
AS Tabla_Datos_Importados(Atr_1 tipo_dato,..., Atr_n tipo_dato)

ON Nombre_Tabla_Izq.PRIMARY_KEY = Tabla_Datos_Importados.FOREIGN_KEY;
```


Uso de la API de extensiones o librerías de PostgreSQL

Las extensiones de **PostgreSQL** nos permiten realizar cálculos, análisis de datos, machine learning, etc. pero para ello debemos instalarlas y ver su documentación de uso, la cual se incluye dentro del siguiente enlace:

<https://www.postgresql.org/docs/current/contrib.html>



Importación y uso de librerías externas de PostgreSQL : **Levenstein y Difference**

A continuación, utilizaremos dos librerías distintas de **PostgreSQL** para realizar tareas específicas:

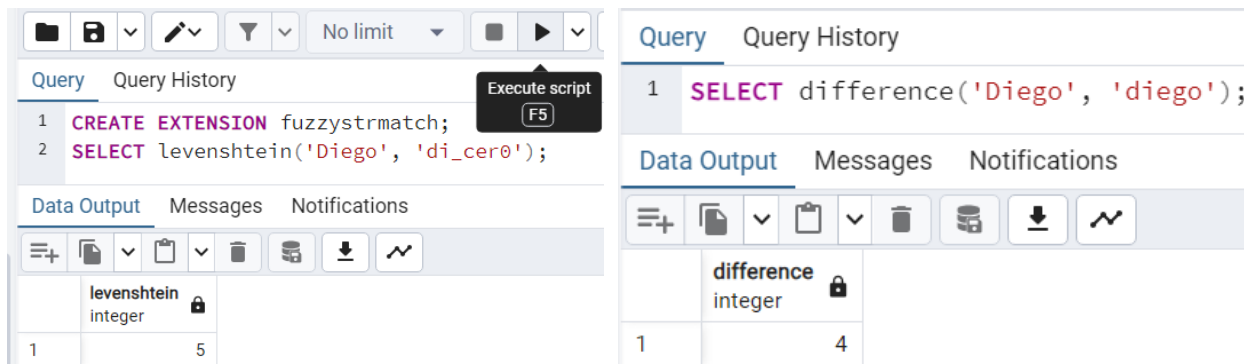
- **Levenstein**: Librería que sirve para **comparar dos palabras letra por letra** a través de un algoritmo llamado distancia de levenstein y obtener un número para ejemplificar su diferencia, siendo 0 el resultado cuando son iguales.
- **Difference**: Librería que a través de machine learning permite **comparar dos palabras y la forma en la que suenan en inglés**, para corregirla si es que se captó mal, este algoritmo lo usa SIRI de Apple. Retorna un valor de 0 a 4 dependiendo de su similitud, siendo 4 si son iguales.

Siempre que queramos utilizar una **extensión**, para saber cuál es el **nombre de librería** que se debe utilizar en el código SQL nos debemos dirigir a la documentación de PostgreSQL y el nombre que se encuentre en mayor jerarquía y contenga el método que se quiera utilizar, ese es el nombre que se debe indicar en el código:

CREATE EXTENSION nombre_librería;

F.17. fuzzystrmatch — determine string similarities and distance
F.17.1. Soundex
F.17.2. Daitch-Mokotoff Soundex
F.17.3. **Levenshtein**
F.17.4. Metaphone
F.17.5. Double Metaphone

Para ejecutar métodos de **librerías externas** debemos ingresar a la opción del menú superior: Tools → Query Tool → Comando **SELECT** que utilice los métodos de librerías externas **PostgreSQL**.



The screenshot shows the Query Tool interface with the following components:

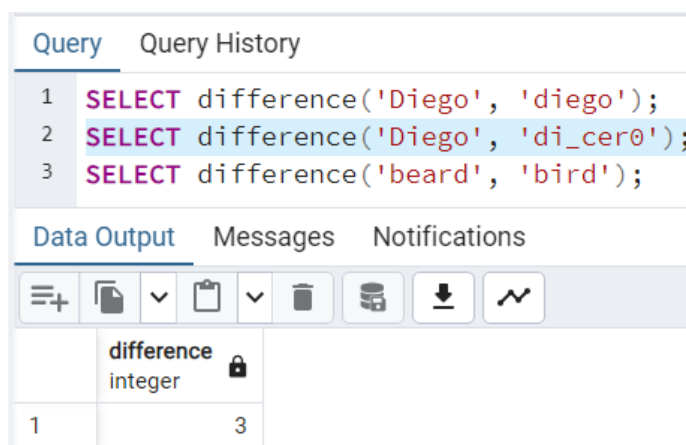
- Query Editor:** Contains two lines of SQL:


```
1 CREATE EXTENSION fuzzystmatch;
2 SELECT levenshtein('Diego', 'di_cer0');
```
- Execute script (F5):** A button to execute the query.
- Data Output:** A table showing the result of the query:

	levenshtein integer
1	5
- Query History:** Shows the executed query:


```
1 SELECT difference('Diego', 'diego');
```
- Data Output (History):** A table showing the result of the query in the history:

	difference integer
1	4



The screenshot shows the Query Tool interface with the following components:

- Query Editor:** Contains three lines of SQL:


```
1 SELECT difference('Diego', 'diego');
2 SELECT difference('Diego', 'di_cer0');
3 SELECT difference('beard', 'bird');
```
- Data Output:** A table showing the result of the query:

	difference integer
1	3
- Query History:** Shows the executed query:


```
1 SELECT difference('Diego', 'diego');
```

Transacciones

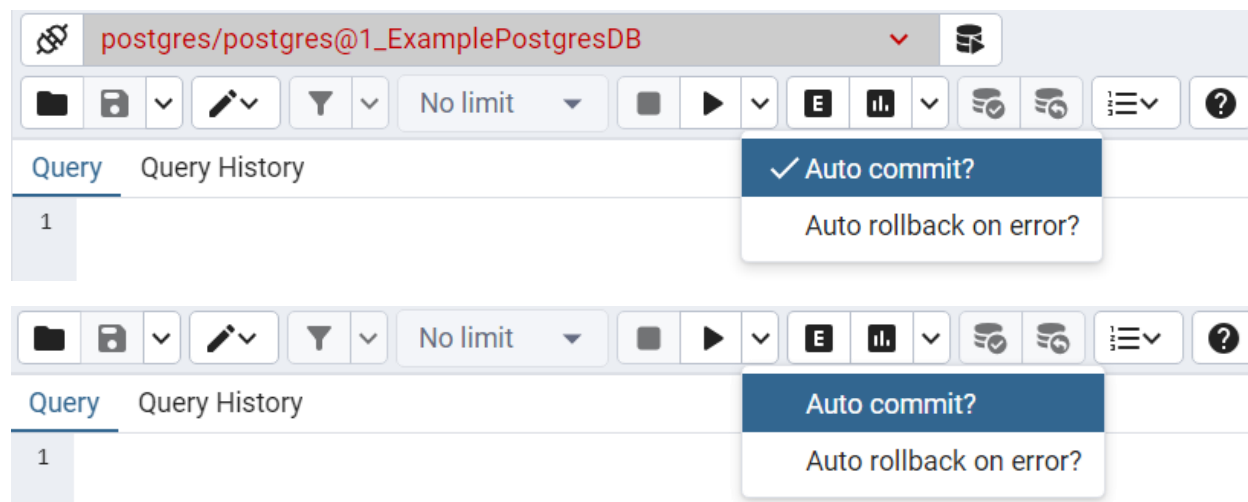
Las **transacciones** son operaciones donde se empaquetan varios comandos en una operación, de forma que se completen todos o ninguno, ya que, en el caso de que ocurra algún fallo que impida que se complete la transacción, ninguno de los pasos se ejecuta y no se afectan a la **base de datos**. Para ello nos apoyamos de los comandos **BEGIN**, **COMMIT** y **ROLLBACK**, los cuales realizan las siguientes funciones:

- **BEGIN:** Este comando se encarga de indicar **el inicio y contenido** de una **transacción**, permitiendo que se ejecuten sus sentencias SQL y se registren en la memoria de la **database**.
- **COMMIT:** Este comando se encarga de indicar **la correcta ejecución** de una **transacción**, permitiendo que las sentencias SQL previamente guardadas en la memoria de la **DB** a través del comando **BEGIN**, puedan afectar sus **relaciones**, **atributos** o **entidades**.
- **ROLLBACK:** Este comando se encarga de indicar **cuando no queremos que los cambios generados por las sentencias SQL durante una transacción se vean reflejados en la base de datos**, logrando así que no se afecten sus **relaciones**, **atributos** o **entidades**.

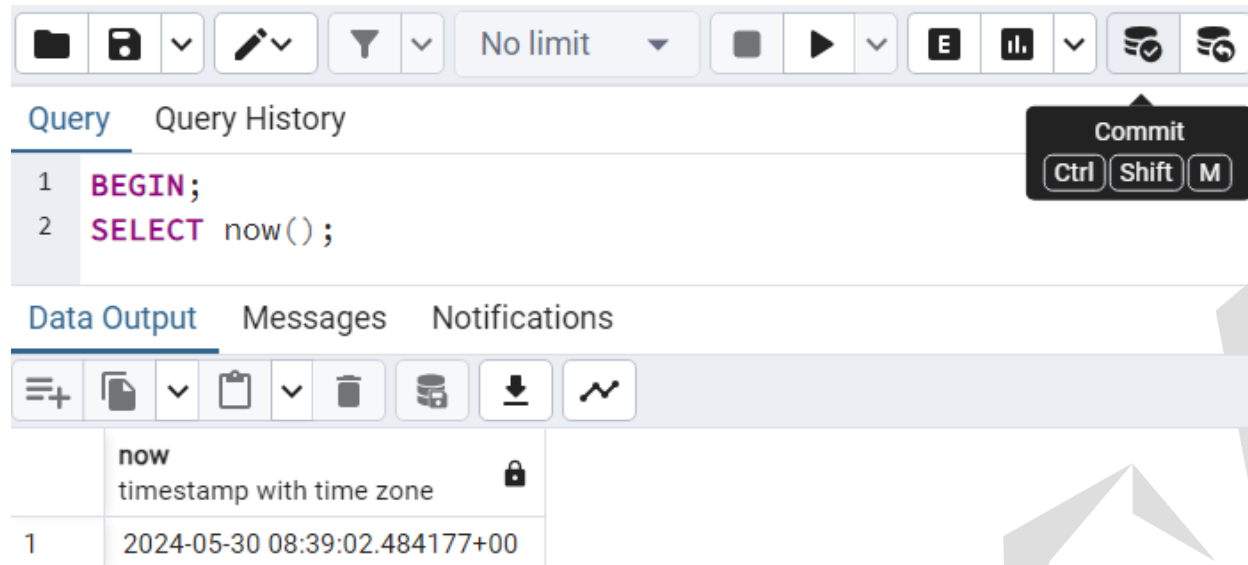
Para ejecutar la **transacción** debemos ingresar a la opción del menú superior: Tools → Query Tool.

Donde vale la pena mencionar que alado del botón de Execute Script, se encuentra seleccionada la opción de **Auto commit?**, la cual por defecto siempre ejecuta el comando **COMMIT** después de cualquier instrucción SQL, permitiendo así que se afecte a la base de datos. Si seleccionamos la opción de **Auto rollback on error?**, le estaremos diciendo a **pgAdmin** que, si ocurre una excepción, por defecto se ejecute el comando **ROLLBACK** al final, logrando así que los efectos del código SQL no se vean reflejados en la **database**.

Siempre que vayamos a ejecutar una transacción, se debe des seleccionar la opción de **Auto commit?**:



Si ejecutamos una operación sin utilizar de forma explícita el comando **COMMIT** o **ROLLBACK**, aparecerán dos opciones nuevas en el menú, donde de forma manual se puede ejecutar este comando en la **base de datos** con el botón de **Commit** o se pueden descartar los cambios hechos con el código SQL al seleccionar la opción de **Rollback**.



Query Query History

```

1 BEGIN;
2 SELECT now();

```

Rollback
Ctrl Shift R

Data Output Messages Notifications

	now timestamp with time zone
1	2024-05-30 08:39:02.484177+00

Ahora para denotar la utilidad de las **transacciones**, vamos a hacer un **INSERT** doble de datos, donde se ingresarán al mismo tiempo datos en las tablas **estación** y **tren**.

Query Query History

```

1 BEGIN;
2 INSERT INTO public.estacion(
3     nombre_estacion, direccion_estacion)
4     VALUES ('Estación transacción', 'Dirección transacción');
5 INSERT INTO public.tren(
6     modelo_tren, capacidad_pasajeros)
7     VALUES ('Modelo transacción', 10);
8 --Como no se ha agregado la instrucción COMMIT, esta inserción
9 --de datos no se verá reflejada en la DB.

```

Execute script
F5

Auto commit?
Auto rollback on error?

Después de haber ejecutado la **transacción** (inicializada por el comando **BEGIN**), tenemos la opción de ejecutarla en la **base de datos** para ver los resultados de su código SQL reflejados en ella al dar clic en el botón de **Commit** (o agregar el comando **COMMIT**) o de rechazarlos al seleccionar la opción de **Rollback** (o agregar el comando **ROLLBACK**).

Query Query History

```

1 BEGIN;
2 INSERT INTO public.estacion(
3     nombre_estacion, direccion_estacion)
4     VALUES ('Estación transacción', 'Dirección transacción');
5 INSERT INTO public.tren(
6     modelo_tren, capacidad_pasajeros)
7     VALUES ('Modelo transacción', 10);
8 --Como no se ha agregado la instrucción COMMIT, esta inserción
9 --de datos no se verá reflejada en la DB.

```

Commit
Ctrl Shift M

Data Output Messages Notifications

INSERT 0 1

Query returned successfully in 49 msec.

Query Query History

```

1 BEGIN;
2 INSERT INTO public.estacion(
3     nombre_estacion, direccion_estacion)
4     VALUES ('Estación transacción', 'Dirección transacción');
5 INSERT INTO public.tren(
6     modelo_tren, capacidad_pasajeros)
7     VALUES ('Modelo transacción', 10);
8 --Como no se ha agregado la instrucción COMMIT, esta inserción
9 --de datos no se verá reflejada en la DB.

```

Rollback
Ctrl Shift R

Data Output Messages Notifications

INSERT 0 1

Query returned successfully in 49 msec.

QueryQuery History

```

1 BEGIN;
2 INSERT INTO public.estacion(
3     nombre_estacion, direccion_estacion)
4     VALUES ('Estación transacción', 'Dirección transacción');
5 INSERT INTO public.tren(
6     modelo_tren, capacidad_pasajeros)
7     VALUES ('Modelo transacción', 10);
8 --Como no se ha agregado la instrucción COMMIT, esta inserción
9 --de datos no se verá reflejada en la DB.
10 COMMIT;

```

Data OutputMessagesNotifications

WARNING: no hay una transacción en curso

COMMIT

Query returned successfully in 33 msec.

QueryQuery History

```

1 SELECT * FROM public.estacion
2 ORDER BY id_estacion ASC

```

Data OutputMessagesNotifications

id_estacion	nombre_estacion	direccion_estacion
[PK] integer	character varying	character varying
87	Stark LLC	24324 Gina Lane
88	Goyette, Ferry and Rice	2 Superior Center
89	Gislason and Sons	368 Summer Ridge Drive
90	Bins and Sons	94495 Arrowood Terrace
91	Vandervort-Rempel	40911 Melvin Park
92	Keeling, Heller and Heaney	9 Oak Plaza
93	Schumm, Wilkinson and Wolf	8 Nelson Circle
94	Boyer, Reynolds and Reinger	83 Moulton Hill
95	Lemke, Orn and Terry	94539 Milwaukee Lane
96	Considine and Sons	8 Mockingbird Trail
97	Kuhman-Crist	24067 Mesta Court
98	Effertz LLC	475 Ronald Reagan Court
99	Bogisich-Stamm	0 Sherman Terrace
100	Hill, West and Predovic	5 Drewry Court
101	Strosin LLC	81913 Main Terrace
102	Lemke LLC	09 Beilfuss Parkway
103	Estación transacción	Dirección transacción
104	Estación transacción	Dirección transacción

Manejo de Excepciones

La gran utilidad de las **transacciones** se denota al realizar un **manejo de excepciones**, donde por ejemplo al hacer un **INSERT** de datos doble, **uno de ellos puede ser correcto y el otro no**, pero nosotros **queremos que cuando esto pase, ninguna de las dos inserciones se ejecute en la base de datos, a menos que toda la instrucción sea correcta**.

En este caso eso se ejemplifica al tratar de insertar manualmente un **id** que ya existe solo en uno de los comandos de inserción, mientras que el otro realizará su función de forma correcta.

Si esto se ejecutará sin el uso de **transacciones**, una de las **inserciones** lanzaría un error y no se ejecutaría, pero la otra sí y en este caso, aunque se indique explícitamente el comando **COMMIT** al final de la instrucción, como todo se encuentra después del comando **BEGIN**, ningún comando se ejecutará porque ocurrió un error al ejecutar la **transacción**, por lo cual las **relaciones**, **atributos** o **entidades** de la **database** no se verán afectadas.

QueryQuery History

```

1 BEGIN;
2 INSERT INTO public.tren(
3     modelo_tren, capacidad_pasajeros)
4     VALUES ('Modelo transacción 2', 100);
5 INSERT INTO public.estacion(
6     id_estacion, nombre_estacion, direccion_estacion)
7     VALUES (1, 'Estación transacción', 'Dirección transacción');
8 --El manejo de excepciones se realiza con el comando ROLLBACK,
9 --el cual detiene la ejecución de un comando SQL sobre una DB.
10 COMMIT;

```

Data OutputMessagesNotifications

ERROR: Ya existe la llave (id_estacion)=(1).llave duplicada viola restricción de unicidad «estacion_pkey»

ERROR: llave duplicada viola restricción de unicidad «estacion_pkey»

SQL state: 23505

Detail: Ya existe la llave (id_estacion)=(1).

Y como podemos ver, ni en la tabla de **estación** (donde ocurrió el error), ni en la tabla de **tren** (donde no ocurrió un error), se ven reflejados los cambios ejecutados por los comandos SQL, debido a que se ejecutó dentro de una **transacción**, donde o todo se ejecuta bien o nada se ejecuta.

Query Query History			
1 SELECT * FROM public.estacion			
2 ORDER BY id_estacion DESC			

Query Query History			
1 SELECT * FROM public.tren			
2 ORDER BY id_tren DESC			

Data Output Messages Notifications			
	id_estacion [PK] integer	nombre_estacion character varying	direccion_estacion character varying
1	106	Estación transacción	Dirección transacción
2	105	Estación transacción	Dirección transacción
3	102	Lemke LLC	09 Beilfuss Parkway
4	101	Strosin LLC	81913 Main Terrace
5	100	Hill, West and Predovic	5 Drewry Court
6	99	Bogisch-Stamm	0 Sherman Terrace
7	98	Effertz LLC	475 Ronald Regan Court
8	97	Kuhlman-Crist	24067 Mesta Court
9	96	Considine and Sons	8 Mockingbird Trail
10	95	Lemke, Orn and Terry	94539 Milwaukee Lane
11	94	Boyer, Reynolds and Reinger	83 Moulton Hill
12	93	Schumm, Wilkinson and Wolf	8 Nelson Circle
13	92	Keeling, Heller and Heaney	9 Oak Plaza
14	91	Vandervort-Rempel	40911 Melvin Park
15	90	Bins and Sons	94495 Arrowood Terrace
16	89	Gislason and Sons	368 Summer Ridge Drive
17	88	Goyette, Ferry and Rice	2 Superior Center
18	87	Stark LLC	24324 Gina Lane

Data Output Messages Notifications			
	id_tren [PK] integer	modelo_tren character varying	capacidad_pasajeros integer
1	105	Modelo transacción	10
2	104	Modelo transacción	10
3	101	Explorer Sport Trac	100
4	100	X3	99
5	99	Jimmy	98
6	98	Discovery	97
7	97	NSX	96
8	96	Fortwo	95
9	95	Laser	94
10	94	Montero Sport	93
11	93	New Beetle	92
12	92	VUE	91
13	91	Outback Sport	90
14	90	Diamante	89
15	89	X-Type	88
16	88	Marauder	87
17	87	Camaro	86
18	86	Sebring	85

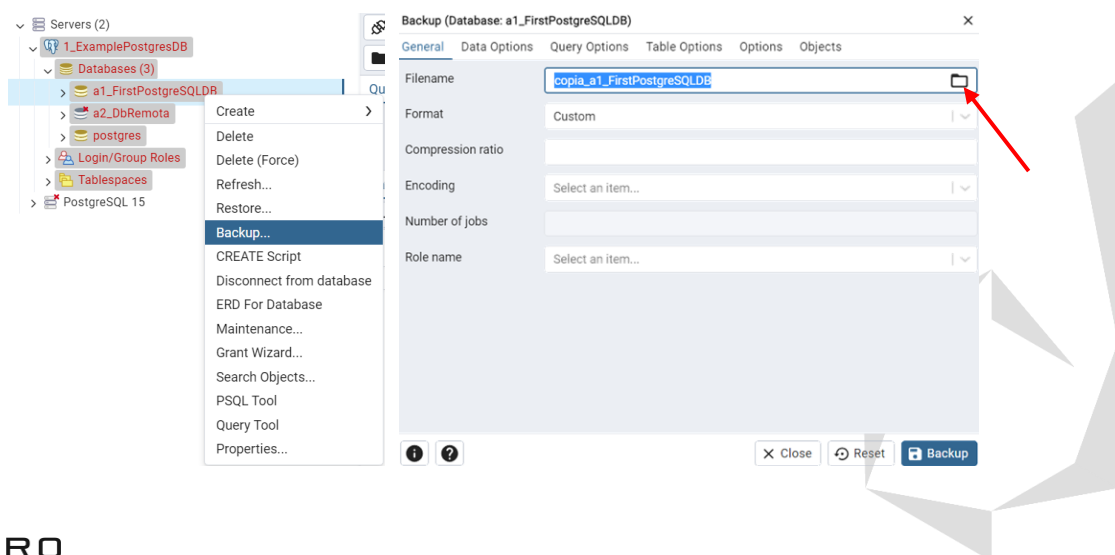
Total rows: 104 of 104 Query complete 00:00:00.096

Total rows: 103 of 103 Query complete 00:00:00.124

Como pudimos observar, la instrucción **ROLLBACK** nunca se utilizó de forma explícita en el código, esto se debe a que no es tan común encontrarla a menos que se coloquen condicionales con código **PL/PgSQL**, esta casi no se usa en códigos SQL sencillos.

Backups y Restauraciones

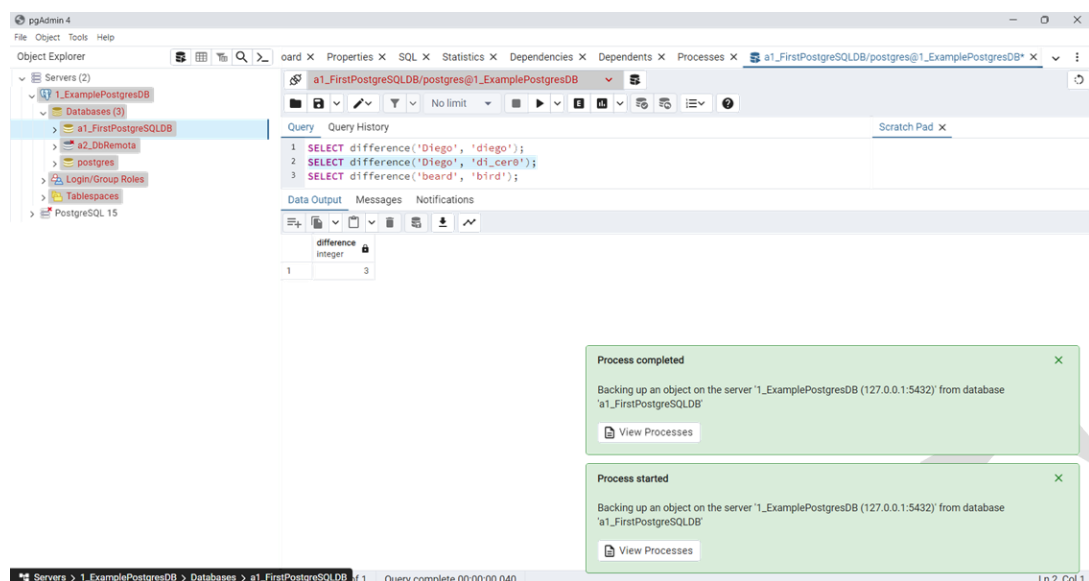
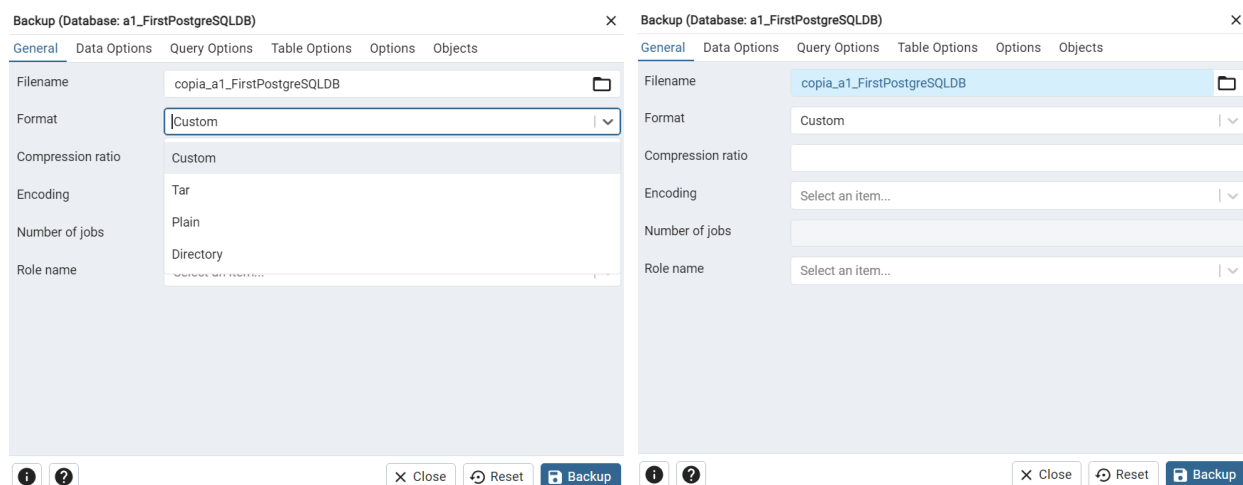
Las **copias de seguridad (DUMP)** son super importantes al manejar bases de datos en caso de que un fallo catastrófico ocurra, para ello **PostgreSQL** cuenta con los servicios de restauración. Para ello debemos seleccionar la opción de: **Servers** → **Motor de base de datos PostgreSQL** → **Databases** → **Nombre_Database** → Clic Derecho → Backup... → **Pestaña General** → **Filename**: Nombre y directorio copia.



Luego es muy importante que entendamos los formatos que puede tener el **backup** de la **base de datos**, el cual puede ser uno de los siguientes:

- **Custom:** Formato único que usa **PostgreSQL** para guardar la información de la base de datos y que solo puede ser restaurada a través del **RDBMS pgAdmin**.
- **Tar:** Es un archivo comprimido que contiene la estructura de la base de datos.
- **Plain:** Es un archivo que contiene SQL plano, este contendrá todos los comandos de **creación de tablas, relaciones, inserción de datos, consultas**, etc.
- **Directory:** Es un archivo sin comprimir que contiene la estructura de la base de datos.

En este caso se seleccionará la opción de: **Pestaña General** → **Format:** Custom → **Compression ratio:** Es la cantidad de veces que el ciclo de compresión se ejecuta para crear un archivo de copia más pequeño → **Role name:** Nombre del usuario que puede acceder a la copia, por defecto es postgres → **Pestaña Data Options** → **Data:** No, para que no solo nos dé los datos, sino la estructura de la **DB** → **Blobs:** Yes, para que se guarden datos binarios como PDF, imágenes, etc. Aunque vale la pena analizar la necesidad de esto, ya que aumentará mucho el peso de la **base de datos** → **?:** Si queremos ver una documentación más detallada de la funcionalidad de cada opción, daremos clic en el botón de interrogación → **Backup**.



Para ver los detalles del **backup**, podemos acceder a la opción de: **Pestaña Processes** → View details, sabiendo así la ubicación donde se encuentra, cabe mencionar que copiar y pegar esta dirección en el explorador de archivos no servirá, sino que debemos seguir la ruta manualmente.

The screenshot shows the pgAdmin 4 interface with the 'Processes' tab selected. A table lists the backup process:

	PID	Type	Server	Object	Start Time	Status	Time Taken (sec)
	5208	Backup Object	1_ExamplePostgresDB (127.0.0.1:5432)	a1_FirstPostgreSQLDB	30/5/2024, 04:16:50	Finished	0.29

Below the table, the 'Process Watcher' window shows the command used to back up the database:

```
C:\Program Files\PostgreSQL\15\pgAdmin 4\runtime\pg_dump.exe --file "C:\Users\diego\DOCUMENTS\1\COPIA_~1" --host "127.0.0.1" --port "5432" --username "postgres" --no-password --role "postgres" --format=c --blobs --encoding "UTF8" --section=data --verbose "a1_FirstPostgreSQLDB"
```

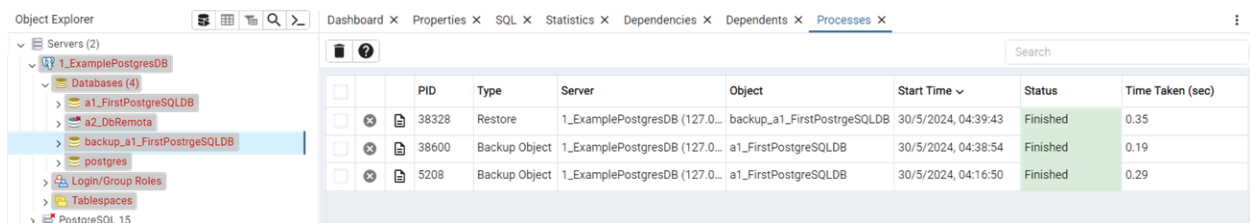
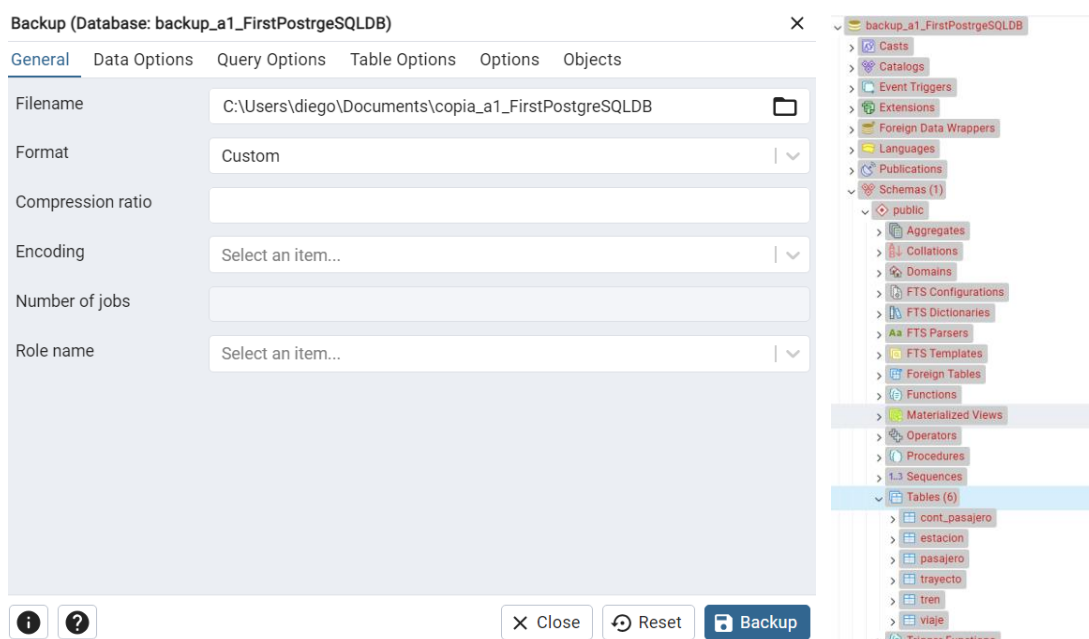
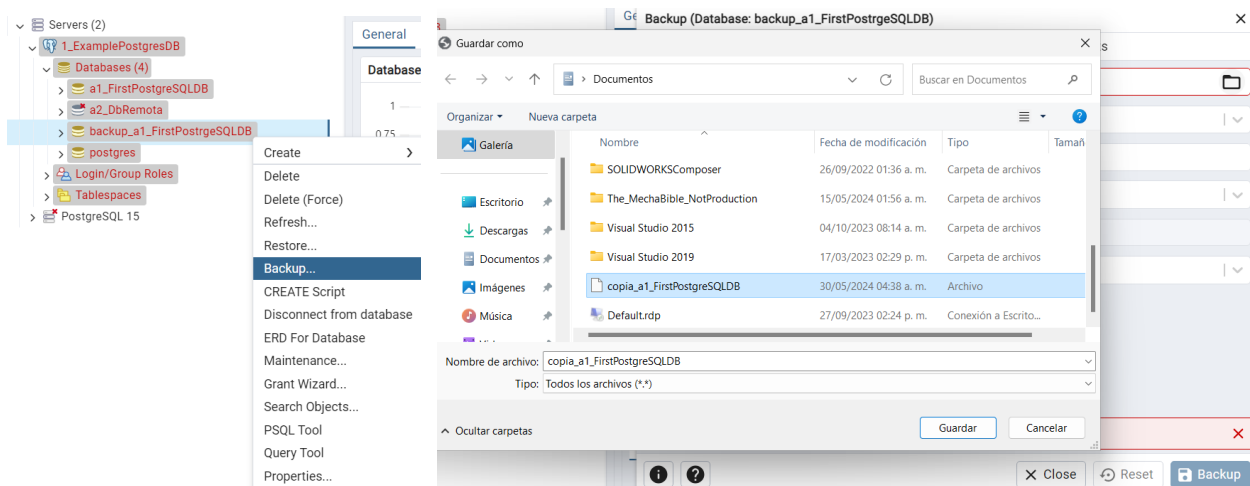
The status is 'Successfully completed.' and the execution time is 0.29 seconds.

Ahora para tener una **copia de seguridad**, debemos crear una **nueva base de datos** que contenga la información del **backup** al seleccionar la opción de: **Servers** → **Motor de base de datos PostgreSQL** → **Databases** → Clic Derecho → **Create** → **Database...** → **Pestaña General** → Database: **Nombre_Database** → Clic Derecho en database nueva → **Restore...** → **Pestaña General** → Filename: Elegir ruta de la copia (Todos los archivos (*.*) en el explorador) → **Restore**.

The first screenshot shows the 'Create Database' dialog box in pgAdmin 4. The 'Database' field is set to 'backup_a1_FirstPostgreSQLDB' and the 'Owner' is 'postgres'.

The second screenshot shows the 'Restore (Database: backup_a1_FirstPostgreSQLDB)' dialog box. The 'Filename' field is set to 'C:\Users\diego\Documents\copia_a1_FirstPostgreSQLDB'.

The third screenshot shows the 'Abrir' (Open) dialog box in Windows Explorer, where the file 'copia_a1_FirstPostgreSQLDB' is selected.



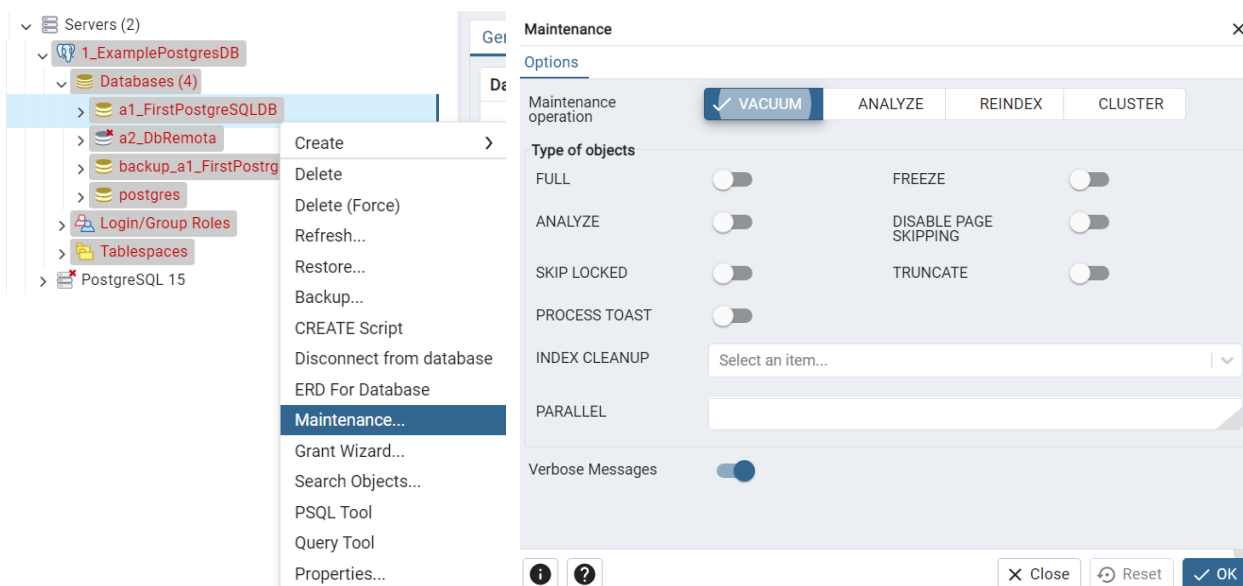
Mantenimiento

PostgreSQL ejecuta en segundo plano de forma automática una serie de funciones, mientras nosotros trabajamos con una **base de datos**, estas actividades son las de mantenimiento y se ejecutan específicamente a través del comando de vaciado o **VACUUM**, ya que este se dedica a quitar todas las

filas, **columnas** o **ítems** del disco duro, que no estén funcionando correctamente. Existen distintos usos del comando **VACUUM** para el mantenimiento de la **database**, los cuales son:

- **VACUUM**: Libera el espacio ocupado por las **filas eliminadas** o **actualizadas**.
- **VACUUM FULL**: Realiza un mantenimiento más completo de una **tabla**, liberando más espacio al liberar memoria ocupada por **filas eliminadas** o **actualizadas**, pero bloqueando la **entidad** afectada y requiriendo más tiempo de procesamiento.
- **ANALYZE**: Recalcula las estadísticas de la **DB** para ayudar al optimizador de consultas a tomar decisiones más eficientes.

El mantenimiento se puede hacer en cada **tabla** individualmente o en la **database** completa, pero si se hace en la **DB** completa, esto podría llegar a bloquear todas sus **entidades** durante un periodo de tiempo. Para analizar el mantenimiento de una **base de datos (o tabla)** debemos seleccionar la opción de: **Servers** → **Motor de base de datos PostgreSQL** → **Databases** → **Nombre_Database** → Clic Derecho → **Maintenance...** → **Pestaña VACUUM**: Cuenta con varias opciones de mantenimiento → **Full**: Limpieza completa de la **DB** o **entidad**, pero bloqueándola durante su proceso de mantenimiento → **Freeze**: Bloquea de forma manual la **base de datos** o **tabla** durante su proceso de mantenimiento → **Analyze**: Ejecuta una revisión del estado actual de la **database** o **entidad**, pero sin realizar ninguna limpieza sobre ella → **Verbose Messages**: Se selecciona si queremos que se muestre un mensaje con el estado actual del mantenimiento → **Pestaña Analyze**: Realiza una revisión del estado actual de la **DB** o **tabla**, pero sin ejecutar ninguna limpieza sobre ella → **Pestaña Reindex**: Se utiliza para realizar un análisis cuando contamos con muchos **índices** o **relaciones** en nuestra **tabla** → **Pestaña Cluster**: Al ejecutar nuestro mantenimiento a través de esta pestaña, indicamos que los cambios queremos que se vean reflejados en el disco duro del servidor.



No es recomendable realizar mantenimiento manualmente, hay que dejar que **PostgreSQL** lo ejecute de forma automática. Aunque si se llega a tener que efectuar, debemos usar solo el comando **VACUUM FULL** en un horario que no afecte a producción que se bloquee la **DB** o **entidad**.

Réplicas

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 al mismo tiempo, existen límites electrónicos o físicos, que limitan 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**.

Ejemplo Réplicas: Jelastic & Cloudjiffy

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

Referencias

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

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

