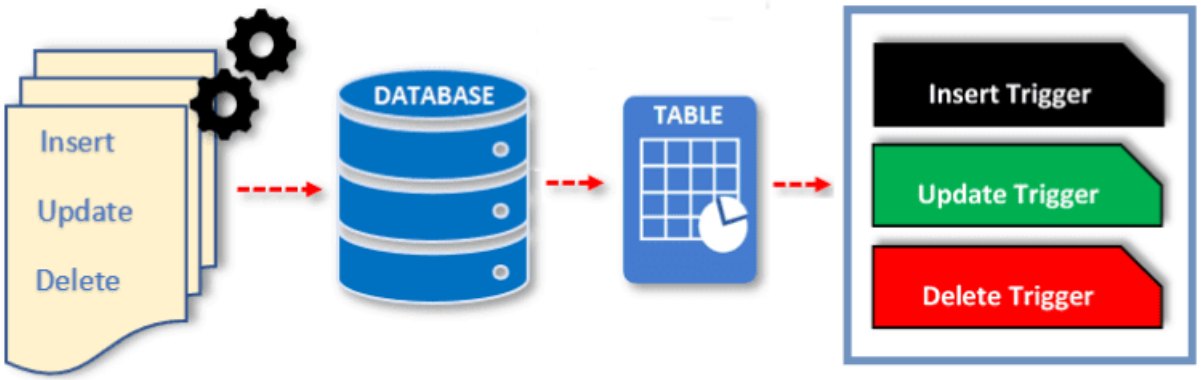
Tema 3

Construcción de guiones de administración



2º C.F.G.S. Administración de Sistemas Informáticos en Red

Administración de Sistemas Gestores de Bases de Datos

Índice

1. Introducción

En el ámbito de la administración de sistemas y bases de datos, la automatización de tareas resulta esencial para mejorar la eficiencia, la fiabilidad y la escalabilidad de los procesos. Este tema se centra en el uso de guiones como herramienta fundamental para optimizar la gestión y el mantenimiento de entornos informáticos, permitiendo reducir la intervención manual y minimizar errores. A lo largo de este bloque se abordarán los elementos necesarios para diseñar soluciones automatizadas que faciliten el control, la supervisión y la ejecución de tareas administrativas de forma estructurada y segura.

2. Guiones

Dentro del ámbito de las bases de datos, un guión es un archivo que contiene una serie de instrucciones o sentencias escritas en un lenguaje específico, normalmente SQL, que se ejecutan de forma automática y ordenada sobre el sistema gestor de bases de datos.

Estos guiones se utilizan para automatizar tareas administrativas y operativas como la creación o modificación de estructuras, la inserción o actualización de datos, la generación de copias de seguridad o la gestión de usuarios y permisos.

El uso de guiones en este contexto facilita la repetición de procesos, garantiza la coherencia de los cambios entre distintos entornos y permite documentar de manera clara las operaciones realizadas sobre la base de datos.

Ejemplo (mi_guión.sql):

```
-- Crear una nueva base de datos

CREATE DATABASE gestion_empleados;


-- Conectarse a la base de datos recién creada

\c gestion_empleados;


-- Crear una tabla para almacenar información de empleados
-- La palabra clave SERIAL indica que id será numérico y autoincremental

CREATE TABLE empleados (

    id SERIAL PRIMARY KEY,

    nombre VARCHAR(100) NOT NULL,

    puesto VARCHAR(50),
```

```
    salario NUMERIC(10, 2), -- Número con hasta 10 dígitos en total, de los cuales 2 son decimales.
    fecha_contratacion DATE DEFAULT CURRENT_DATE
);

-- Insertar algunos registros de ejemplo
INSERT INTO empleados (nombre, puesto, salario)
VALUES
    ('Ana López', 'Administrativa', 24000.00),
    ('Carlos Gómez', 'Técnico de sistemas', 30000.00),
    ('Lucía Martín', 'Analista de datos', 32000.00);

-- Consultar los empleados con salario superior a 25000
SELECT nombre, puesto, salario
FROM empleados
WHERE salario > 25000
ORDER BY salario DESC;
```

psql -U <usuario> -f <archivo>.sql

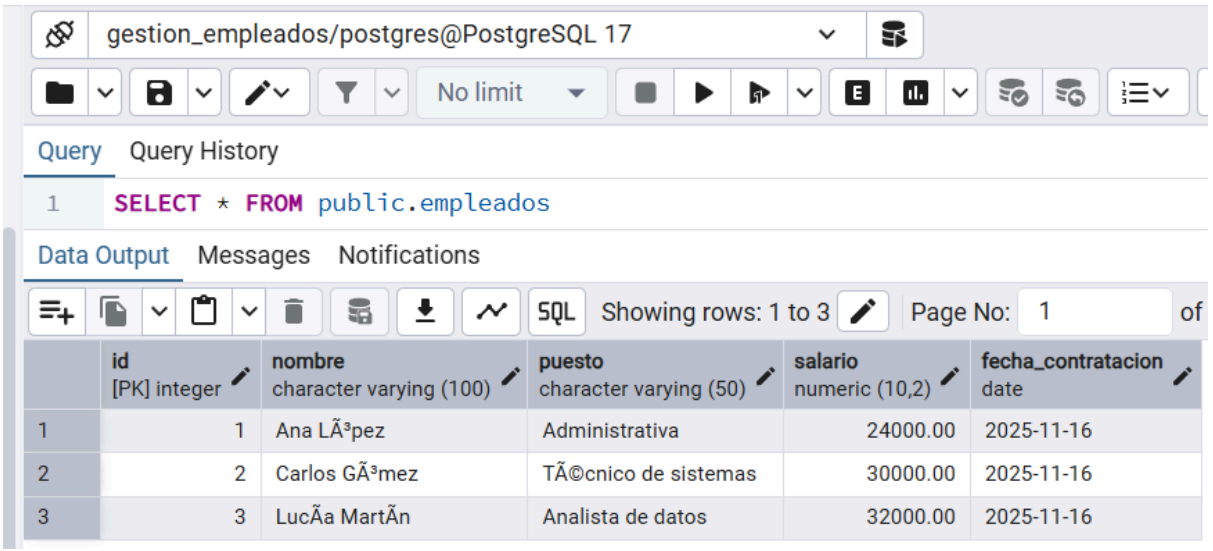
- Siendo <usuario> el nombre de tu usuario en postgreSQL, por defecto es postgres.
- Siendo <archivo> el nombre del archivo del guión.

Si no reconoce el comando psql hay que agregar la ruta del directorio bin de PostgreSQL (ej. C:\Program Files\PostgreSQL\<tu_version>\bin) a la variable de entorno PATH del sistema Windows, luego reinicia la terminal.

Como podemos observar en la siguiente imagen, al ejecutar el comando y revisar los datos introducidos, se aprecia que las tildes y otros caracteres propios del castellano no se visualizan correctamente.

```
C:\Users\josag\Documents\Docencia\2 ASIR\ASGBD>psql -U postgres -f mi_guión.sql
Contraseña para usuario postgres:

CREATE DATABASE
Ahora está conectado a la base de datos «gestion_empleados» con el usuario «postgres».
CREATE TABLE
INSERT 0 3
     nombre     |        puesto        | salario
----------------+----------------------+----------
Luc├¡a Mart├¡n | Analista de datos    | 32000.00
Carlos G├│mez  | T├®cnico de sistemas | 30000.00
(2 ╞ilas)
```

En un primer momento podríamos pensar que se trata únicamente de un problema de representación de la información en la terminal. Sin embargo, al visualizar los datos directamente en pgAdmin, podemos confirmar que, efectivamente, no se han almacenado correctamente.



Lo que estamos experimentando es un problema de codificación de caracteres. En Windows, la consola por defecto utiliza CP850 o CP437, mientras que PostgreSQL normalmente almacena los datos en UTF-8. Por eso, los caracteres con tildes como á, é u ó se muestran mal o se sustituyen por símbolos extraños como ├¡ o ├®.

El problema surge porque, al ejecutar un script SQL con psql -f archivo.sql, PostgreSQL interpreta el archivo usando la codificación que detecta de la consola. En Windows, si la consola no está en UTF-8, los caracteres especiales del archivo se traducen incorrectamente al almacenarlos en la base de datos.

Una solución común es cambiar la codificación de la consola a UTF-8 con “chcp 65001”. Sin embargo, incluso así a veces psql sigue interpretando mal el archivo, especialmente si el nombre del archivo contiene caracteres no ASCII.

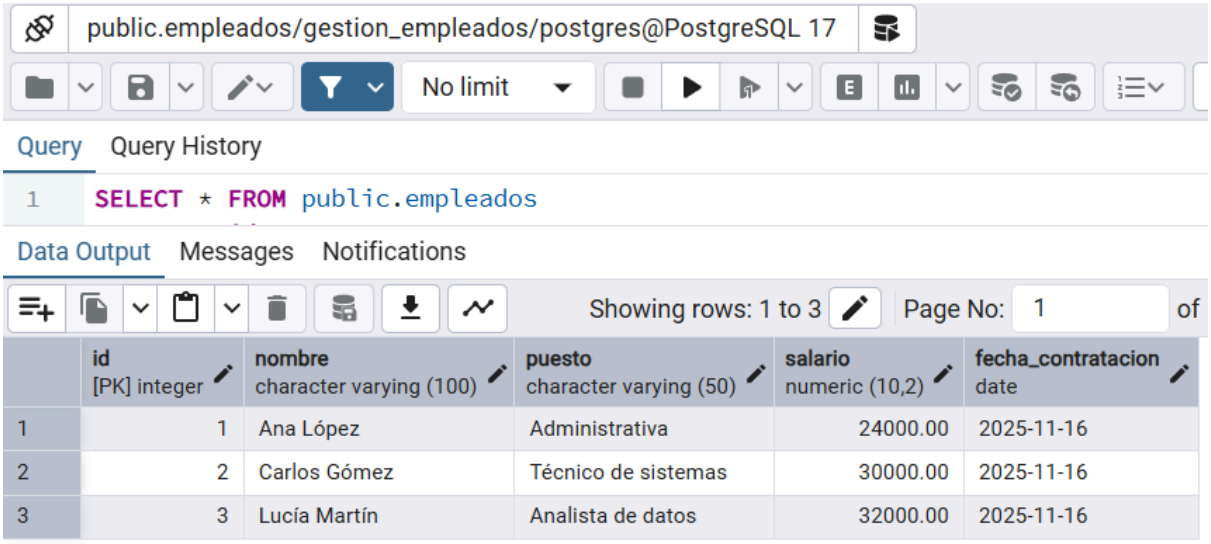
El método que siempre funciona en Windows es usar el comando *type* para enviar el contenido del archivo directamente a psql:

type <archivo>.sql | psql -U <usuario>

Esto funciona porque *type* envía los bytes exactos del archivo tal cual están en UTF-8, sin que Windows o la consola intenten convertirlos a otra codificación. De esta forma, PostgreSQL recibe correctamente los caracteres con tildes y los almacena bien en la base de datos.

```
C:\Users\josag\Documents\Docencia\2 ASIR\ASGBD>chcp 65001
Página de códigos activa: 65001

C:\Users\josag\Documents\Docencia\2 ASIR\ASGBD>type mi_guión.sql | psql -U postgres
Contraseña para usuario postgres:

SET
CREATE DATABASE
Ahora está conectado a la base de datos «gestion_empleados» con el usuario «postgres».
CREATE TABLE
INSERT 0 3
    nombre    |       puesto        | salario
--------------+---------------------+----------
 LucÃ-a MartÃ-n | Analista de datos   | 32000.00
 Carlos GÃ³mez | TÃ©cnico de sistemas | 30000.00
(2 filas)
```

Aunque hayamos cambiado el encoding de la terminal, los caracteres todavía se muestran incorrectamente; sin embargo, esto ya es únicamente un problema de visualización en la consola. Si lo consultamos en el pgAdmin, podemos comprobar que los datos se han guardado correctamente en la base de datos.



3. Estructuras de flujo de control

Las estructuras de flujo de control son mecanismos que permiten ejecutar instrucciones de forma condicional, iterativa o controlada, en lugar de ejecutarlas siempre de manera lineal.

En PostgreSQL, las estructuras de flujo de control se utilizan principalmente dentro de funciones o procedimientos almacenados (ya que los scripts SQL puros no soportan directamente IF…ELSE o WHILE fuera de bloques de código). PostgreSQL usa el lenguaje PL/pgSQL para este tipo de scripts.

3.1. Estructuras de decisión (condicionales)

Son mecanismos que permiten que un script decida qué instrucciones ejecutar según una condición. Sirven para comprobar datos, ejecutar acciones solo en casos específicos y hacer que los scripts reaccionen de forma más inteligente a distintas situaciones de la base de datos.

3.1.1. IF…ELSIF…ELSE

El bloque IF…ELSIF…ELSE permite que un script ejecute diferentes acciones según se cumplan ciertas condiciones. Primero se evalúa el IF; si su condición es verdadera, se ejecuta ese bloque. Si no lo es, se revisan las condiciones del ELSIF, una a una. Si ninguna resulta verdadera, se ejecuta el ELSE, que actúa como opción por defecto.

Ejemplo: Comprobar si hay películas anteriores al año 2000 en la tabla film y mostrar un mensaje distinto según exista alguna o no.

```
DO $$
BEGIN
   IF EXISTS (SELECT 1 FROM film WHERE release_year < 2000) THEN
      RAISE NOTICE 'Hay películas del siglo XX';
   ELSE
      RAISE NOTICE 'Todas las películas son del siglo XXI';
   END IF;
END
$$;
```

La consulta SELECT 1 FROM film WHERE … busca en la tabla film y no devuelve información de la película; simplemente devuelve un 1 por cada fila que cumpla la condición. Se está usando para comprobar si hay resultados.

Query | Query History

```
1  DO $$
2  BEGIN
3      IF EXISTS (SELECT 1 FROM film WHERE release_year < 2000) THEN
4          RAISE NOTICE 'Hay películas del siglo XX';
5      ELSE
6          RAISE NOTICE 'Todas las películas son del siglo XXI';
7      END IF;
8  END
9  $$;
```

Data Output | Messages | Notifications

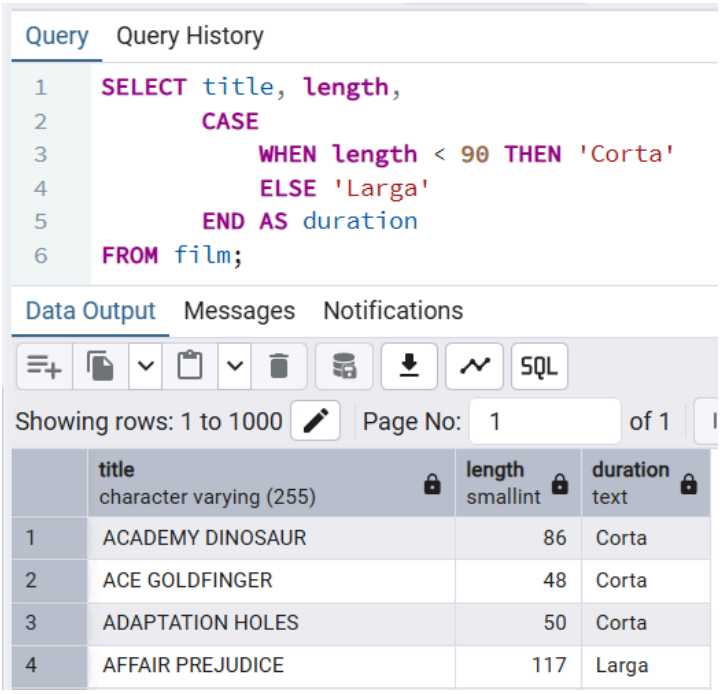
```
NOTICE:  Todas las películas son del siglo XXI
DO

Query returned successfully in 57 msec.
```

3.1.2. CASE

El CASE es una estructura que permite devolver distintos valores según una condición, normalmente dentro de una consulta SQL. Funciona como un selector: evalúa condiciones en orden y devuelve el resultado asociado a la primera que se cumpla. Si ninguna condición es verdadera, puede dar un valor por defecto mediante ELSE.

Ejemplo: Obtener el título y la duración de cada película con una columna calculada que clasifica cada una como “Corta” si dura menos de 90 minutos o “Larga” en caso contrario.

```
SELECT title, length,
    CASE
      WHEN length < 90 THEN 'Corta'
      ELSE 'Larga'
    END AS duration
FROM film;
```



3.2. Estructuras de bucle (iterativas)

Las estructuras de bucle permiten repetir un conjunto de instrucciones varias veces, ya sea mientras se cumpla una condición, durante un número determinado de iteraciones o recorriendo los resultados de una consulta. Su propósito es automatizar tareas repetitivas, como procesar filas una por una, realizar cálculos iterativos o ejecutar acciones que deben repetirse hasta alcanzar un objetivo.

3.2.1. WHILE

El WHILE es una estructura de bucle que permite repetir un bloque de instrucciones mientras una condición sea verdadera. Antes de cada iteración, se evalúa la condición; si sigue cumpliéndose, el código continúa ejecutándose una y otra vez. Se utiliza para tareas en las que no se sabe de antemano cuántas repeticiones serán necesarias y el número de iteraciones depende del estado de los datos o de un contador que se va actualizando dentro del propio bucle.

Ejemplo: Imprimir los números del 1 al 5.

```
DO $$

DECLARE
   i INT := 1;
BEGIN
   WHILE i <= 5 LOOP
      RAISE NOTICE 'Número: %', i;
      i := i + 1;
```

```
    END LOOP;
END
$$;
```

Query | Query History

```
1  DO $$
2  DECLARE
3      i INT := 1;
4  BEGIN
5      WHILE i <= 5 LOOP
6          RAISE NOTICE 'Número: %', i;
7          i := i + 1;
8      END LOOP;
9  END
10 $$;
```

Data Output | Messages | Notifications

```
NOTICE:  Número: 1
NOTICE:  Número: 2
NOTICE:  Número: 3
NOTICE:  Número: 4
NOTICE:  Número: 5
DO

Query returned successfully in 49 msec.
```

3.2.2. FOR

El FOR es una estructura de bucle que permite repetir un conjunto de instrucciones de manera controlada. Puede recorrer un rango de valores numéricos o iterar automáticamente sobre las filas que devuelve una consulta. Es útil cuando se conoce el conjunto de elementos a procesar, ya sea una secuencia de números o un listado de registros, y permite ejecutar acciones para cada uno de ellos sin necesidad de gestionar manualmente contadores o condiciones adicionales.

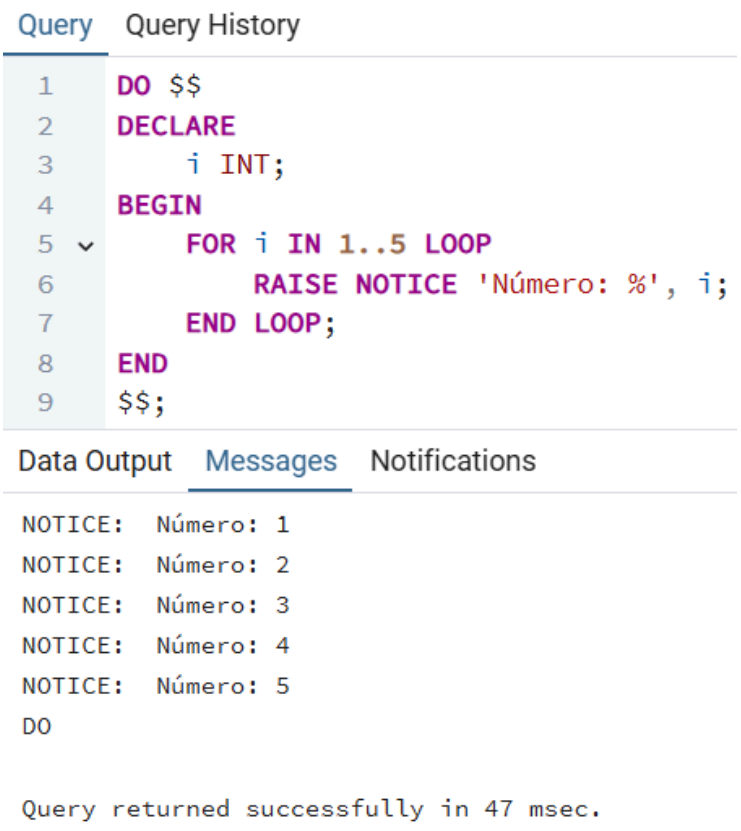
En PostgreSQL, la estructura FOR puede utilizarse de dos formas distintas, según lo que se necesite recorrer. Por un lado, existe el FOR por rango, que permite iterar sobre una secuencia de valores numéricos (por ejemplo, del 1 al 10). Por otro lado, está el FOR por consulta, que recorre automáticamente las filas devueltas por una sentencia SELECT.

FOR (iterando sobre rango)

Permite repetir un bloque de instrucciones recorriendo una secuencia de valores numéricos. Se define un rango inicial y final, y el bucle ejecuta su contenido una vez por cada número dentro de ese intervalo. Este tipo de FOR es útil cuando se conoce de antemano cuántas iteraciones se necesitan o cuando se desea ejecutar una acción un número fijo de veces sin depender de datos de una tabla.

Ejemplo: Imprimir los números del 1 al 5.

```
DO $$
DECLARE
   i INT;
BEGIN
   FOR i IN 1..5 LOOP
      RAISE NOTICE 'Número: %', i;
   END LOOP;
END
$$;
```



FOR (iterando sobre filas)

Permite recorrer automáticamente los resultados de una consulta y ejecutar un conjunto de instrucciones para cada fila obtenida. Dentro del bucle se dispone de un registro que representa la fila actual, con acceso directo a sus columnas. Este tipo de FOR es especialmente útil para procesar datos uno a uno sin necesidad de declarar ni gestionar cursores manualmente, simplificando tareas como validaciones, cálculos o actualizaciones basadas en los valores de cada registro.

Ejemplo: Mostrar el nombre de todas las filas de la tabla language.

```
DO $$
DECLARE
   record RECORD;
BEGIN
   FOR record IN SELECT * FROM language LOOP
      RAISE NOTICE 'Idioma: %', record.name;
   END LOOP;
END
$$;
```

Query | Query History

```
1  DO $$
2  DECLARE
3      record RECORD;
4  BEGIN
5      FOR record IN SELECT * FROM language LOOP
6          RAISE NOTICE 'Idioma: %', record.name;
7      END LOOP;
8  END
9  $$;
```

Data Output | Messages | Notifications

```
NOTICE:  Idioma: English
NOTICE:  Idioma: Japanese
NOTICE:  Idioma: Mandarin
NOTICE:  Idioma: French
NOTICE:  Idioma: German
NOTICE:  Idioma: Spanish
NOTICE:  Idioma: Italian
DO

Query returned successfully in 44 msec.
```

3.2. Control de transacciones

Las estructuras de control de transacciones permiten agrupar varias operaciones en un único bloque para garantizar que todas se realicen correctamente o, si ocurre un problema, se deshagan por completo. Con ellas se decide cuándo empiezan los cambios y cuándo deben confirmarse o revertirse.

Dentro de este mecanismo, COMMIT se utiliza para efectuar la transacción, es decir, para guardar de forma definitiva todos los cambios realizados. Por el contrario, si se necesita revertir lo ejecutado por algún error o condición inesperada, se utiliza ROLLBACK, que deshace todos los cambios y deja la base de datos en el estado anterior al inicio de la transacción.

Ejemplo: Crear una transacción para actualizar el nombre de dos idiomas en la tabla language, y finalmente confirma los cambios.

BEGIN;

UPDATE language SET name = ‘Inglés’ WHERE language_id = 1;
UPDATE language SET name = ‘Italiano’ WHERE language_id = 2;

COMMIT;
-- O usar ROLLBACK en caso de error manual

Hay que tener cuidado al trabajar con transacciones, porque si escribes mal una consulta dentro de un BEGIN y esta produce un error, la transacción queda abierta y en estado pendiente.

Query Query History

```
1  BEGIN;
2  UPDATE language SET name = 'Inglés' WHERE id = 1;
3  COMMIT;
```

Data Output Messages Notifications

```
ERROR:  no existe la columna «id»
LINE 2: UPDATE language SET name = 'Inglés' WHERE id = 1;
                                                  ^

SQL state: 42703
Character: 50
```

Mientras la transacción no se cierre, no podrás ejecutar nuevas operaciones en esa misma sesión.

Query Query History

```
2  BEGIN;
3  UPDATE language SET name = 'Inglés' WHERE id = 1;
4  COMMIT;
```

Data Output Messages Notifications

```
ERROR:  transacción abortada, las órdenes serán ignoradas hasta el fin de bloque de transacción

SQL state: 25P02
```

Para recuperar el control, es necesario ejecutar un ROLLBACK, que cancela la transacción y permite volver a trabajar con normalidad.

Query Query History

```
1  ROLLBACK;
2
3  BEGIN;
4  UPDATE language SET name = 'Inglés' WHERE language_id = 1;
5  COMMIT;
```

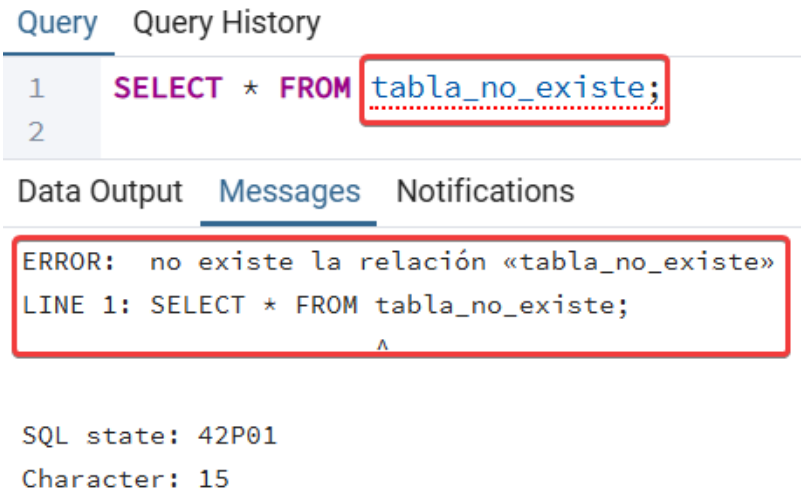
Data Output Messages Notifications

```
COMMIT

Query returned successfully in 48 msec.
```

4. Excepciones

Las excepciones son situaciones anómalas que ocurren durante la ejecución de un programa o de una instrucción y que impiden que el proceso continúe con normalidad. Cuando aparece una excepción, el sistema detiene el flujo normal de ejecución y genera un mensaje de error que indica qué ha fallado. En el contexto de las bases de datos y del lenguaje SQL, una excepción se produce cuando el motor no puede completar una operación, ya sea por un problema de sintaxis, por violar una regla de integridad, por conflictos entre transacciones o por cualquier condición que haga imposible finalizar la instrucción. Las excepciones permiten detectar y manejar errores de forma controlada, evitando resultados incorrectos y manteniendo la coherencia de los datos.

Ejemplo: Hacer una consulta a una tabla que no existe.



Lanzar una excepción

Lanzar una excepción significa generar de forma intencionada un error dentro de un bloque o función para detener la ejecución normal cuando se produce una situación no válida. Esto se hace mediante la instrucción RAISE EXCEPTION, que permite al desarrollador avisar explícitamente al motor de que una condición no debería continuar.
Cuando se lanza una excepción, se interrumpe inmediatamente la operación en curso, cancela cualquier cambio realizado dentro de la transacción activa y devuelve un mensaje de error al cliente o a la aplicación que ha ejecutado la instrucción. Esta instrucción puede incluir un mensaje personalizado que describa el problema, así como parámetros para insertar valores dinámicos dentro del texto del error, lo que ayuda a ofrecer información más precisa.
Este mecanismo es esencial para implementar validaciones de negocio directamente en la base de datos, garantizando que no se ejecuten operaciones inválidas y manteniendo la coherencia de los datos.

Ejemplo: Lanzar una excepción si la cantidad es mayor que el saldo actual.

```
DO $$
DECLARE
   cantidad numeric := 30;
   saldo_actual numeric := 10;
BEGIN
   IF cantidad > saldo_actual THEN
      RAISE EXCEPTION 'Saldo insuficiente: intentas retirar %, pero solo hay %',
                  cantidad, saldo_actual;
   END IF;
END;
$$;
```

Otro ejemplo: Lanzar una excepción con código de error personalizado cuando una variable booleana sea falsa.

```
DO $$
DECLARE
   existe boolean := false;
BEGIN
   IF NOT existe THEN
      RAISE EXCEPTION USING
         MESSAGE = 'El usuario no existe',
         ERRCODE = 'P1234';
   END IF;
END;
$$;
```

Query Query History

```
1   DO $$
2   DECLARE
3       existe boolean := false;
4   BEGIN
5       IF NOT existe THEN
6           RAISE EXCEPTION USING
7               MESSAGE = 'El usuario no existe',
8               ERRCODE = 'P1234';
9       END IF;
10  END;
11  $$;
```

Data Output Messages Notifications

```
ERROR:  El usuario no existe
CONTEXT:  función PL/pgSQL inline_code_block en la línea 6 en RAISE

SQL state: P1234
```

Manejar excepciones

Manejar excepciones consiste en interceptar los errores que se producen durante la ejecución de un bloque de código y decidir qué hacer con ellos. Esto se realiza mediante la estructura BEGIN … EXCEPTION … END, que permite capturar tanto errores específicos como cualquier error genérico.
Cuando se produce una excepción, ya sea lanzada manualmente o generada por el motor de la base de datos, la ejecución normal se detiene y el control pasa automáticamente a la sección EXCEPTION, donde puedes registrar información, ejecutar lógica alternativa, realizar un rollback parcial o incluso relanzar la excepción para que llegue a la aplicación llamadora.

En la práctica, el manejo de excepciones se usa para evitar que una función falle silenciosamente, para proporcionar mensajes más precisos, para aplicar reglas de negocio y para garantizar que las transacciones se comportan de forma coherente. Dentro del bloque EXCEPTION puedes capturar errores concretos utilizando códigos SQLSTATE (por ejemplo, WHEN unique_violation THEN …) o capturar cualquier error con WHEN OTHERS THEN …. Además, dentro de esta sección es posible consultar información del error mediante variables especiales como SQLERRM, que contiene el mensaje de error, o SQLSTATE, que indica el código exacto del error. En casos donde necesitas registrar el error pero deseas que siga propagándose, puedes relanzarlo simplemente utilizando de nuevo RAISE.

Ejemplo: Intentar insertar un dato en la tabla de idiomas y, si al hacerlo se viola un atributo único, mostrar un mensaje de error.

Query | Query History

```
1  DO $$
2  BEGIN
3      INSERT INTO language(language_id, name) VALUES (1, 'Japanese');
4  EXCEPTION WHEN unique_violation THEN
5      RAISE NOTICE 'Ya existe un lenguaje con ese id';
6  END
7  $$;
```

Data Output | Messages | Notifications

```
NOTICE:  Ya existe un lenguaje con ese id
DO
```

Otro ejemplo: Ejecutar una operación que lanza una excepción con código P1234, y cuando ocurre ese error específico, muestra el mensaje del error. Si ocurre cualquier otro error, muestra un aviso genérico.

```
DO $$
BEGIN
   -- Simulación de una operación que genera un error
   RAISE EXCEPTION USING MESSAGE = 'El usuario no existe', ERRCODE = 'P1234';

EXCEPTION
   WHEN SQLSTATE 'P1234' THEN
      RAISE NOTICE 'Error controlado: %.', SQLERRM;
   WHEN OTHERS THEN
      RAISE NOTICE 'Ocurrió un error no previsto';
END;
$$;
```

Query | Query History

```
1  DO $$
2  BEGIN
3      -- Simulación de una operación que genera un error
4      RAISE EXCEPTION USING MESSAGE = 'El usuario no existe', ERRCODE = 'P1234';
5
6  EXCEPTION
7      WHEN SQLSTATE 'P1234' THEN
8          RAISE NOTICE 'Error controlado: %.', SQLERRM;
9      WHEN OTHERS THEN
10         RAISE NOTICE 'Ocurrió un error no previsto';
11 END;
12 $$;
```

Data Output | Messages | Notifications

```
NOTICE:  Error controlado: El usuario no existe.
DO
```

El manejo de excepciones se puede usar para controlar errores dentro de una transacción. Esto permite detectar cuándo ocurre un fallo, mostrar un mensaje informativo o realizar acciones correctivas, y luego hacer un rollback para deshacer los cambios realizados hasta ese momento. De esta forma, aunque una operación falle, la transacción no deja la base de datos en un estado inconsistente y el proceso puede finalizar de manera controlada, asegurando la integridad de los datos y evitando que errores inesperados interrumpan todo el flujo de ejecución.

Ejemplo: Intentar cambiar el nombre de un idioma y, si la operación falla, revertir el cambio y mostrar un mensaje de error.

```
DO $$
BEGIN
   BEGIN  -- inicia la transacción interna
      UPDATE language SET name = 'Inglés' WHERE id_raro = 1;
      COMMIT;  -- si todo va bien, se confirma
   EXCEPTION WHEN OTHERS THEN
      ROLLBACK;  -- si algo falla, se revierte
      RAISE NOTICE 'La transacción ha fallado y se hizo ROLLBACK.';
   END;
END
$$;
```

Query Query History

```
1  DO $$
2  BEGIN
3      BEGIN  -- inicia la transacción interna
4          UPDATE language SET name = 'Inglés' WHERE id_raro = 1;
5          COMMIT;  -- si todo va bien, se confirma
6      EXCEPTION WHEN OTHERS THEN
7          ROLLBACK;  -- si algo falla, se revierte
8          RAISE NOTICE 'La transacción ha fallado y se hizo ROLLBACK.';
9      END;
10 END
11 $$;
```

Data Output Messages Notifications

```
NOTICE:  La transacción ha fallado y se hizo ROLLBACK.
DO
```

5. Funciones

Una función es un bloque de código que realiza una tarea específica y puede recibir parámetros, devolver un valor y ser reutilizada en consultas SQL o procedimientos más complejos. Son similares a funciones en otros lenguajes de programación, pero ejecutan lógica directamente dentro de la base de datos. En PostgreSQL los nombres de las funciones son, por defecto, insensibles a mayúsculas y minúsculas.

Ventajas:

- Reutilización de código.
- Encapsulación de lógica compleja.
- Optimización de consultas.
- Seguridad: se pueden restringir accesos a tablas y dejar solo funciones para manipular datos.

Para ver las funciones creadas en una base de datos en el pgadmin hay que ir a Servidor → Base de datos → Esquema (ej. public) → Functions.

5.1. Sintaxis básica

Para crear o actualizar una función:

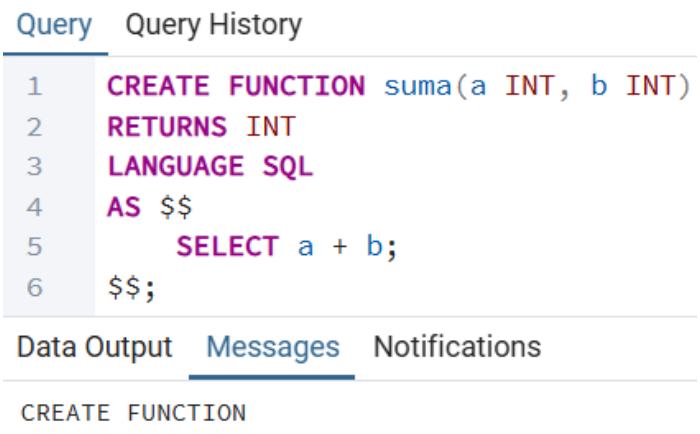
```
CREATE [OR REPLACE] FUNCTION nombre_funcion(param1 tipo, param2 tipo, ...)
RETURNS tipo_de_retorno
LANGUAGE lenguaje
AS $$
-- cuerpo de la función
$$;
```

- CREATE FUNCTION → crea la función.
- OR REPLACE → reemplaza la función si ya existe.
- param1 tipo, param2 tipo → parámetros de entrada (opcional).

- RETURNS tipo_de_retorno → tipo de dato que devuelve la función.
- LANGUAGE → lenguaje de programación usado (PL/pgSQL, SQL, C, Python con PL/Python, etc.)
- $$ ... $$ → delimitadores del cuerpo de la función.

Ejemplo:

```
CREATE FUNCTION suma(a INT, b INT)
RETURNS INT
LANGUAGE SQL
AS $$
   SELECT a + b;
$$;
```



Para eliminar una función:

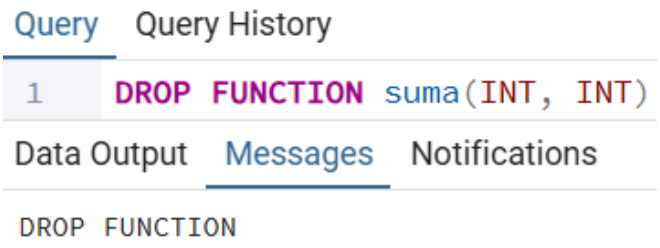
Hay que tener en cuenta que, a diferencia de las tablas, hay que especificar la lista de tipos de los parámetros, porque puede haber varias funciones con el mismo nombre (sobrecarga de funciones).

DROP FUNCTION [IF EXISTS] nombre_funcion(param_tipo1, param_tipo2, ...);

- IF EXISTS: evita error si la función no existe.
- nombre_funcion: el nombre de la función.
- param_tipo1, param_tipo2, ...: tipos de datos de los parámetros (obligatorio).

Ejemplo:

```
DROP FUNCTION suma(INT, INT);
```

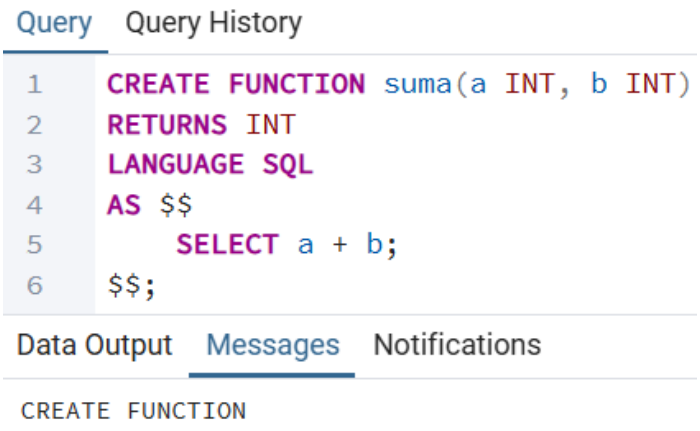


5.2. Funciones SQL

Son funciones simples que solo usan SQL puro. No requieren PL/pgSQL.

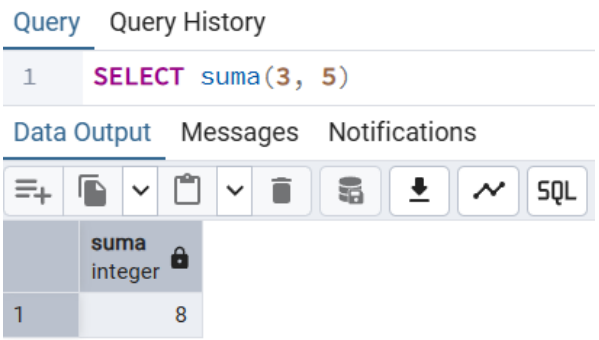
Creación:

```
CREATE FUNCTION suma(a INT, b INT)
RETURNS INT
LANGUAGE SQL
AS $$
   SELECT a + b;
$$;
```



Uso:

```
SELECT suma(3, 5); -- devuelve 8
```



5.3. Funciones PL/pgSQL

PL/pgSQL es el lenguaje procedural de PostgreSQL. Permite lógica compleja como condicionales, bucles, excepciones.

Creación:

```
CREATE FUNCTION factorial(n INT)
RETURNS INT
LANGUAGE plpgsql
AS $$
DECLARE
   resultado INT := 1;
BEGIN
   IF n < 0 THEN
      RAISE EXCEPTION 'No se puede calcular factorial de un número negativo';
   END IF;

   FOR i IN 1..n LOOP
      resultado := resultado * i;
   END LOOP;

   RETURN resultado;
END;
$$;
```

Query | Query History

```
1  CREATE FUNCTION factorial(n INT)
2  RETURNS INT
3  LANGUAGE plpgsql
4  AS $$
5  DECLARE
6      resultado INT := 1;
7  BEGIN
8      IF n < 0 THEN
9          RAISE EXCEPTION 'No se puede calcular factorial de un número negativo';
10     END IF;
11
12     FOR i IN 1..n LOOP
13         resultado := resultado * i;
14     END LOOP;
15
16     RETURN resultado;
17 END;
18 $$;
```

Data Output | Messages | Notifications

```
CREATE FUNCTION
```

Uso:

```
SELECT factorial(5); -- devuelve 120
```

Query | Query History

```
1  SELECT factorial(5)
```

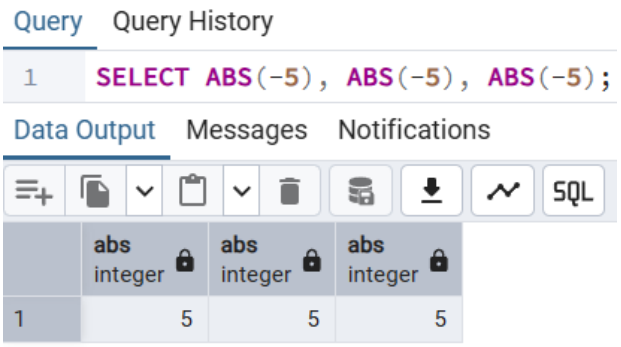
Data Output | Messages | Notifications

	factorial integer
1	120

5.4. Funciones Inmutables

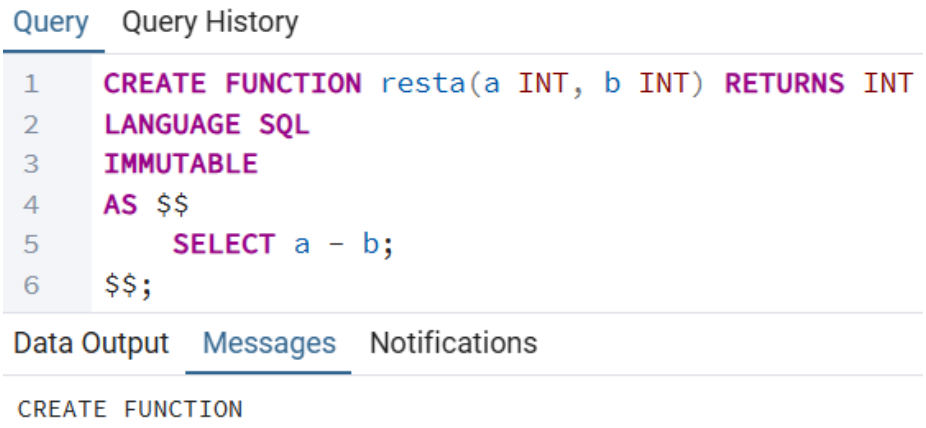
Siempre devuelve el mismo resultado para los mismos parámetros, útiles para indexación y optimización.

Ejemplo: ABS(-5) → siempre 5.



Creación:

```
CREATE FUNCTION resta(a INT, b INT) RETURNS INT
LANGUAGE SQL
IMMUTABLE
AS $$
   SELECT a - b;
$$;
```



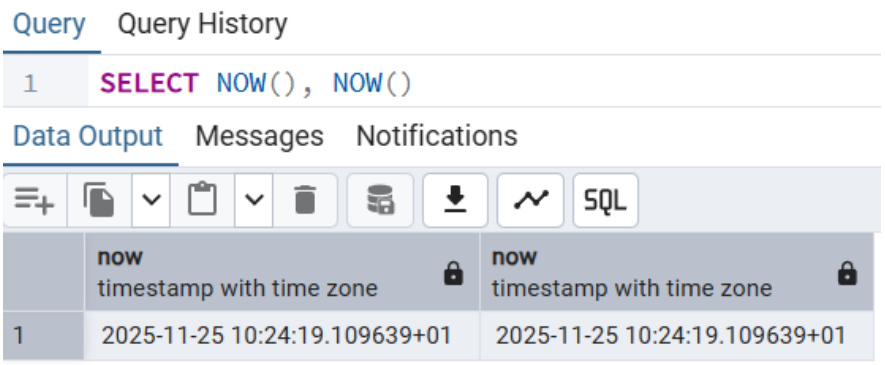
Uso:

```
SELECT resta(2, 5), resta(2, 5), resta(2, 5); -- siempre devuelve -3
```

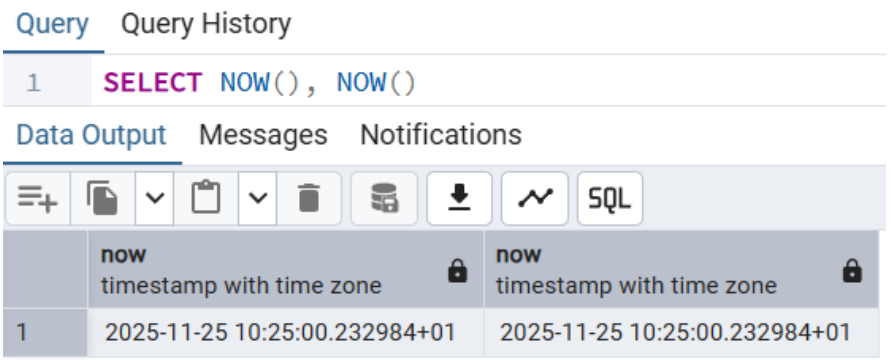
5.5. Funciones Estables

Devuelve el mismo resultado durante una misma transacción, pero puede cambiar entre transacciones.

Ejemplo: Obtener la fecha y hora actual con la función NOW()



Si lo vuelvo a ejecutar cambia los valores:



Creación:

```
CREATE FUNCTION obtener_fecha_actual() RETURNS TIMESTAMP
LANGUAGE SQL
STABLE
AS $$
   SELECT NOW();
$$;
```

Query Query History

```
1  CREATE FUNCTION obtener_fecha_actual() RETURNS TIMESTAMP
2  LANGUAGE SQL
3  STABLE
4  AS $$
5      SELECT NOW();
6  $$;
```

Data Output Messages Notifications

```
CREATE FUNCTION
```

Uso:

SELECT obtener_fecha_actual();

Query Query History

```
1  SELECT obtener_fecha_actual(), obtener_fecha_actual()
```

Data Output Messages Notifications

	obtener_fecha_actual timestamp without time zone	obtener_fecha_actual timestamp without time zone
1	2025-11-25 10:29:59.389923	2025-11-25 10:29:59.389923

Si lo vuelvo a ejecutar cambia los valores:

Query Query History

```
1  SELECT obtener_fecha_actual(), obtener_fecha_actual()
```

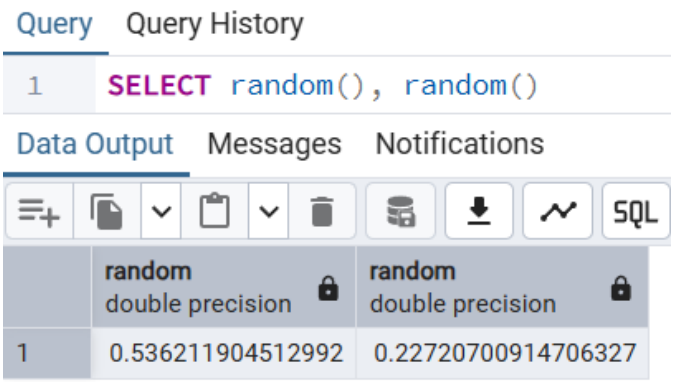
Data Output Messages Notifications

	obtener_fecha_actual timestamp without time zone	obtener_fecha_actual timestamp without time zone
1	2025-11-25 10:30:45.453392	2025-11-25 10:30:45.453392

5.6. Funciones Volátiles

Puede devolver resultados distintos incluso en la misma consulta.

Ejemplo: funciones que generan números aleatorios.



Creación:

```
CREATE FUNCTION generar_aleatorio(min INT, max INT) RETURNS INT
LANGUAGE SQL
VOLATILE
AS $$
   SELECT trunc(min + (max - min) * random());
$$;
```

Uso:

```
SELECT generar_aleatorio(1, 10); -- devuelve un número entero aleatorio entre 1 y
10
```

5.7. Funciones con parámetros opcionales y por nombre

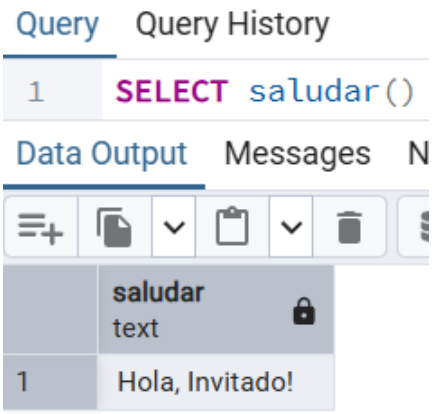
Puedes dar valores por defecto a los parámetros.

Creación:

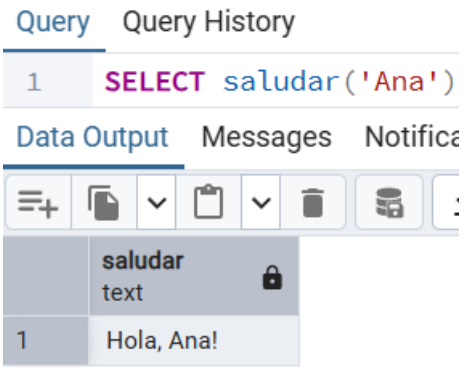
```
CREATE FUNCTION saludar(nombre TEXT DEFAULT 'Invitado')
RETURNS TEXT
LANGUAGE SQL
AS $$
   SELECT 'Hola, ' || nombre || '!';
$$;
```

Uso:

```
SELECT saludar(); -- Hola, Invitado!
```

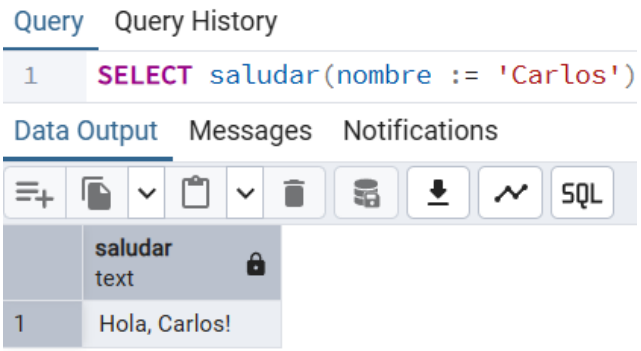


SELECT saludar('Ana'); -- Hola, Ana!



También se pueden pasar parámetros por nombre:

SELECT saludar(nombre := 'Carlos'); -- Hola, Carlos!



5.8. Funciones que devuelven tablas

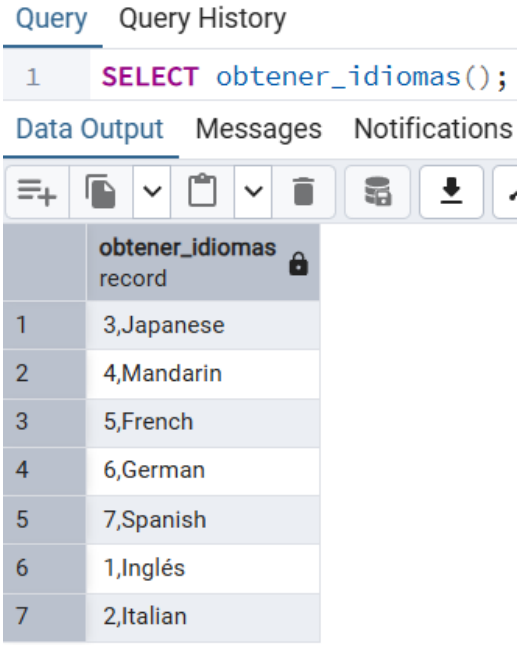
Puedes crear funciones que devuelvan múltiples filas.

Creación:

```
CREATE FUNCTION obtener_idiomas()
RETURNS TABLE(id INT, nombre TEXT)
LANGUAGE SQL
AS $$
   SELECT language_id, name FROM language;
$$;
```

Uso:

```
SELECT obtener_idiomas();
```

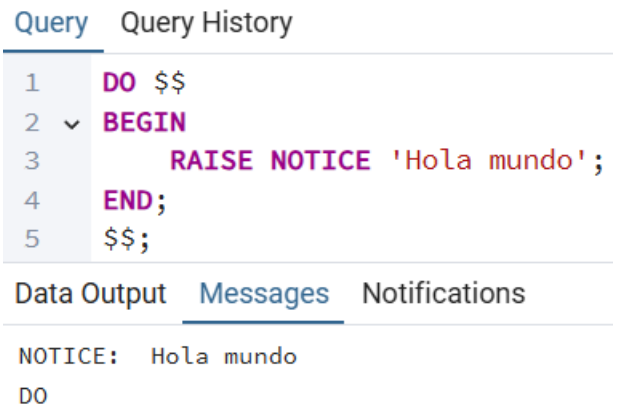


5.9. Funciones Anónimas

Para ejecutar código temporal sin crear una función permanente.

Uso:

```
DO $$
BEGIN
   RAISE NOTICE 'Hola mundo';
END;
$$;
```



6. Triggers

Un trigger es un objeto que contiene un bloque de código SQL que se ejecuta automáticamente cuando ocurre un evento específico en una tabla o vista.
Los eventos que suelen activar un trigger son:

- INSERT → cuando se inserta una fila.
- UPDATE → cuando se actualiza una fila.
- DELETE → cuando se elimina una fila.

Dependiendo del motor de base de datos (SQL Server, MySQL, PostgreSQL, Oracle…), también existen detalles como:

- BEFORE: se ejecuta antes del INSERT/UPDATE/DELETE.
- AFTER: se ejecuta después.
- INSTEAD OF: permite interceptar la operación y ejecutar otra cosa en su lugar. En PostgreSQL solo se puede con vistas.

Los triggers se utilizan principalmente para automatizar tareas o garantizar la integridad de los datos sin depender de la aplicación. Algunos ejemplos:

1. **Auditoría de cambios**
   Registrar quién hizo un cambio, cuándo y qué valores cambió.
   Ejemplo: guardar cada modificación en una tabla de historial.

2. **Validación de datos**
   Evitar operaciones incorrectas antes de que lleguen a la base de datos.
   Ejemplo: impedir que se elimine un empleado que tiene ventas asociadas.

3. **Mantenimiento automático**
   Actualizar automáticamente valores derivados.
   Ejemplo: recalcular el stock cuando se inserta una nueva venta.

4. **Implementación de reglas de negocio**
   Forzar reglas que deben cumplirse pase lo que pase, incluso si otra aplicación se conecta a la BD.
   Ejemplo: cuando se crea un usuario, insertar automáticamente su configuración inicial.

Ejemplo: Crear un trigger que valide que, antes de insertar o actualizar un registro en la tabla country, el valor de la columna country no contenga ningún dígito. Si el nombre del país incluye números, la operación debe abortarse y lanzarse un error. Si cumple la condición, la inserción o actualización debe continuar normalmente.

1. Primero, creamos la función que va a llamar el trigger.

```
CREATE OR REPLACE FUNCTION validate_country_name()
RETURNS TRIGGER AS $$
BEGIN
   IF NEW.country ~ '[0-9]' THEN
      RAISE EXCEPTION 'El nombre del país no puede contener dígitos.';
   END IF;

   RETURN NEW; -- Permite continuar la inserción si pasa la validación
END;
$$ LANGUAGE plpgsql;
```

El operador ~ en PostgreSQL permite hacer coincidencias con expresiones regulares.
En este caso, '[0-9]' busca cualquier dígito entre 0 y 9.

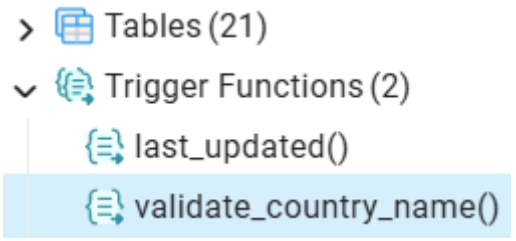
Query | Query History

```
1  CREATE OR REPLACE FUNCTION validate_country_name()
2  RETURNS TRIGGER AS $$
3  BEGIN
4      IF NEW.country ~ '[0-9]' THEN
5          RAISE EXCEPTION 'El nombre del país no puede contener dígitos.';
6      END IF;
7
8      RETURN NEW; -- Permite continuar la inserción si pasa la validación
9  END;
10 $$ LANGUAGE plpgsql;
```

Data Output | Messages | Notifications

```
CREATE FUNCTION
```

2. Podemos verificar en el pgadmin que se ha creado la función yendo a Servidor → Base de datos → Esquema (ej. public) → Trigger Functions.



3. Ahora tenemos que crear el trigger indicando que antes de insertar o actualizar un registro en la tabla country, se ejecute la función de validación que hemos creado anteriormente.

CREATE TRIGGER trg_validate_country_name
BEFORE INSERT OR UPDATE ON country
FOR EACH ROW
EXECUTE FUNCTION validate_country_name();

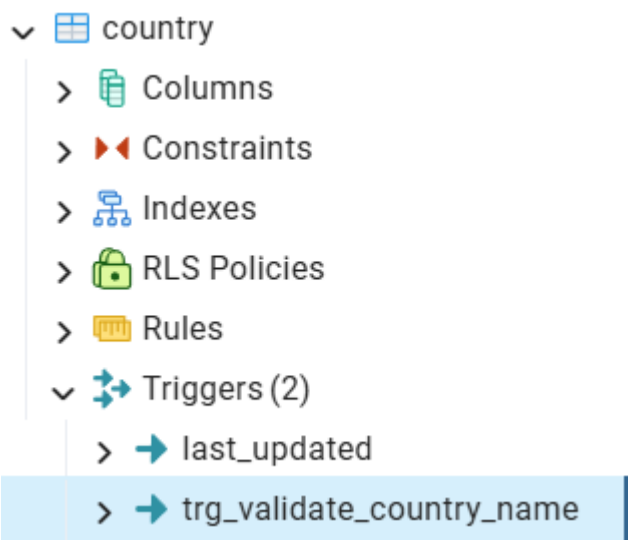
Query | Query History

```
1  CREATE TRIGGER trg_validate_country_name
2  BEFORE INSERT OR UPDATE ON country
3  FOR EACH ROW
4  EXECUTE FUNCTION validate_country_name();
```

Data Output | Messages | Notifications

```
CREATE TRIGGER
```

4. Podemos verificar en el pgadmin que se ha creado la función yendo a Servidor → Base de datos → Esquema (ej. public) → Tablas → Tabla para la que se creó el trigger → Triggers.



5. Comprobamos que el trigger creado funciona correctamente intentando añadir un país cuyo nombre contiene dígitos, verificando que la inserción falla y el registro no puede almacenarse.

INSERT INTO country (country) VALUES ('Españ4')