UNIVERSIDAD DE CUENCA

FACULTAD DE INGENIERÍA

Sep 2025 - Feb 2026

Base Datos II

Capítulo 2: SQL Procedural y Triggers (Funciones)

Juan Carlos Pesántez Ing., MgT., MBA., MTD.

Tipos de Funciones

Tipo de Función	Devuelve	Características clave	Dificultad
Función Escalar	Valor único	Simple, para cálculos o transformaciones	🟢 Básico
Función que retorna tabla	Conjunto de filas	Puede usarse como una tabla, acepta filtros	🟡 Medio
Función con parámetros por defecto	Valor o tabla	Parametrizable, flexible en la invocación	🟡 Medio
Función recursiva	Valor escalar	Llama a sí misma, útil para algoritmos o estructuras jerárquicas	🔴 Avanzado
Función con SQL dinámico	Valor dinámico	Construye y ejecuta una consulta sobre la marcha	🔴 Avanzado
Función con manejo de errores	Valor controlado	Captura excepciones, ideal para robustecer procesos	🟡 Medio
Función según volatilidad (IMMUTABLE, STABLE, VOLATILE)	Cualquier tipo	Clasificación para optimización y predictibilidad	🟡 Medio

Función Escalar

Devuelve un solo valor. Se usa para cálculos simples, conversiones, validaciones u operaciones básicas.

Ejemplo real:
En un sistema de facturación, esta función se usa cada vez que se muestra el total del IVA.

```
CREATE FUNCTION calcular_iva(precio NUMERIC)
RETURNS NUMERIC AS $$
BEGIN
    RETURN precio * 0.15;
END;
$$ LANGUAGE plpgsql;
```

Función de Tabla

Devuelve una tabla como resultado. Útil para crear consultas dinámicas, reportes o filtrar datos en base a parámetros.

Ejemplo real:
En un sistema e-commerce, el backend usa esta función para listar productos de una categoría elegida por el usuario.

```
CREATE OR REPLACE FUNCTION productos_por_categoria(cat_id INT)
RETURNS TABLE(id INT, nombre TEXT, precio NUMERIC)

AS $$
BEGIN
  RETURN QUERY
  SELECT p.id, p.nombre, p.precio
  FROM productos p
  WHERE p.categoria_id = cat_id;

END;
$$
LANGUAGE plpgsql;
```

```
CREATE OR REPLACE FUNCTION productos_por_categoria(cat_id INT)
RETURNS TABLE(id INT, nombre TEXT, precio NUMERIC)
AS $$
  SELECT p.id, p.nombre, p.precio
  FROM productos p
  WHERE p.categoria_id = cat_id;  -- también podrías usar $1
$$ LANGUAGE sql;
```

Función con Parámetros

Permite omitir argumentos al invocar la función, si se desea usar valores estándar.

Ejemplo real:
En promociones de una tienda online, si no se especifica un descuento personalizado, se aplica el estándar del 10%.

```
CREATE FUNCTION aplicar_descuento(precio NUMERIC, descuento NUMERIC DEFAULT 0.1)
RETURNS NUMERIC AS $$
BEGIN
    RETURN precio - (precio * descuento);
END;
$$ LANGUAGE plpgsql;
```

```
SELECT aplicar_descuento(100, 0.15);
SELECT aplicar_descuento(100);
SELECT aplicar_descuento( descuento => 0.2,precio => 250);
```

Función Recursiva

Se llama a sí misma. Útil para recorrer estructuras como árboles o realizar cálculos repetitivos.

Ejemplo real:
Una empresa cuenta con una estructura jerárquica en la que cada empleado puede tener un jefe directo. Esta relación se representa en la misma tabla empleados, mediante el campo id_jefe, que hace referencia a otro empleado. Se desea crear una función que, dado el ID de un empleado, retorne todos sus jefes (es decir, la cadena jerárquica ascendente hasta llegar al nivel más alto, que no tiene jefe).

id_empleado	nombre	id_jefe
1	Ana	NULL
2	Carlos	1
3	Diana	2
4	Esteban	2
5	Fernanda	3

Función Recursiva

```sql
CREATE TABLE empleados (
    id_empleado INT PRIMARY KEY,
    nombre TEXT NOT NULL,
    id_jefe INT REFERENCES empleados(id_empleado)
);
```

```sql
INSERT INTO empleados (id_empleado, nombre, id_jefe) VALUES
(1, 'Ana', NULL),        -- Jefa general
(2, 'Carlos', 1),        -- Jefe intermedio
(3, 'Diana', 2),         -- Subjefa
(4, 'Esteban', 2),       -- Otro subordinado de Carlos
(5, 'Fernanda', 3);      -- Subordinada de Diana
```

Función Recursiva

```sql
CREATE OR REPLACE FUNCTION obtener_jefes_sql(
  id_empleado_consultado INT
)
RETURNS TABLE(
  id_empleado INT,
  nombre      TEXT,
  id_jefe     INT,
  nivel       INT
)
AS $$
  WITH RECURSIVE jerarquia AS (
    SELECT e.id_empleado, e.nombre, e.id_jefe, 0 AS nivel
    FROM empleados e
    WHERE e.id_empleado = id_empleado_consultado

    UNION ALL

    SELECT e.id_empleado, e.nombre, e.id_jefe, j.nivel + 1
    FROM empleados e
    JOIN jerarquia j ON e.id_empleado = j.id_jefe
  )

  SELECT id_empleado, nombre, id_jefe, nivel
  FROM jerarquia
  WHERE nivel > 0
  ORDER BY nivel;
$$ LANGUAGE sql;
```

SELECT * FROM obtener_jefes(5);

id_empleado	nombre	id_jefe
3	Diana	2
2	Carlos	1
1	Ana	NULL

Función Recursiva

🔷 Paso 1: Caso base (la semilla de la recursión)

```sql
SELECT e.id_empleado, e.nombre, e.id_jefe
FROM empleados e
WHERE e.id_empleado = id_empleado_consultado
```

- Este `SELECT` toma como punto de partida el **empleado que estamos consultando**.
- Por ejemplo, si llamas `obtener_jefes(5)`, selecciona a Fernanda (5).

(1ª iteración)

```sql
SELECT e.id_empleado, e.nombre, e.id_jefe
FROM empleados e
JOIN jerarquia j ON e.id_empleado = j.id_jefe
```

Como `j` contiene `id_jefe = 3`, busca `e.id_empleado = 3`, o sea, **Diana**.

📌 Resultado:

id_empleado	nombre	id_jefe
3	Diana	2

Función Recursiva

2ª iteración

Ahora, `jerarquia` contiene también a Diana (`id_jefe = 2`), así que se busca `e.id_empleado = 2` , que es **Carlos**.

📌 Resultado:

id_empleado	nombre	id_jefe
2	Carlos	1

3ª iteración

`jerarquia` ahora incluye a Carlos (`id_jefe = 1`), así que busca `e.id_empleado = 1` , que es **Ana**.

📌 Resultado:

id_empleado	nombre	id_jefe
1	Ana	NULL

Función Recursiva

SI UTILIZAMOS LANGUAGE SQL

```sql
CREATE OR REPLACE FUNCTION obtener_jefes_sql(id_empleado_consultado INT)
RETURNS TABLE(id_empleado INT, nombre TEXT, id_jefe INT, nivel INT)
AS $$
  WITH RECURSIVE jerarquia AS (
    SELECT e.id_empleado, e.nombre, e.id_jefe, 0 AS nivel
    FROM empleados e
    WHERE e.id_empleado = id_empleado_consultado
    UNION ALL
    SELECT e.id_empleado, e.nombre, e.id_jefe, j.nivel + 1
    FROM empleados e
    JOIN jerarquia j ON e.id_empleado = j.id_jefe
  )
  SELECT id_empleado, nombre, id_jefe, nivel
  FROM jerarquia
  WHERE nivel > 0
  ORDER BY nivel;
$$ LANGUAGE sql;
```

Función con SQL Dinámico

Construye consultas rápidas, útil cuando los nombres de tablas o columnas se definen en tiempo de ejecución.

Ejemplo real:
Un dashboard de administración permite consultar dinámicamente el número de registros de cualquier tabla del sistema.

```
CREATE FUNCTION contar_filas(tabla TEXT)
RETURNS INT AS $$
DECLARE
    resultado INT;
BEGIN
    EXECUTE format('SELECT COUNT(*) FROM %I', tabla) INTO resultado;
    RETURN resultado;
END;
$$ LANGUAGE plpgsql;
```

Probar : SELECT contar_filas('productos');

- Usa `EXECUTE` para ejecutar una consulta dinámica (`SELECT COUNT(*) FROM <nombre_tabla>`).
- `%I` en `format` es para identificar nombres de objetos SQL (como nombres de tabla, columna, etc.), lo cual previene inyecciones SQL.
- El resultado se almacena en la variable `resultado` y se retorna.

Función manejo errores

Permite capturar y controlar errores, mejorando la robustez del sistema ante condiciones inesperadas.

Ejemplo real:
En un sistema contable, se evita que una operación falle si el denominador es cero (ej. promedio de valores no registrados).

```
CREATE FUNCTION dividir_seguro(a NUMERIC, b NUMERIC)
RETURNS NUMERIC AS $$
BEGIN
    RETURN a / b;
EXCEPTION
    WHEN division_by_zero THEN
        RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```

```
SELECT dividir_seguro(10, 2);    -- Devuelve 5
SELECT dividir_seguro(10, 0);    -- Devuelve NULL (sin error)
SELECT dividir_seguro(8, 4);     -- Devuelve 2
```

Función con Clasificación

PostgreSQL permite clasificar una función según su estabilidad, para que el optimizador de consultas decida cuándo y cómo ejecutarla. Hay tres niveles:

Clasificación	Significado
`IMMUTABLE`	Siempre devuelve el mismo resultado para los mismos argumentos.
`STABLE`	No cambia durante una misma consulta, pero sí entre ejecuciones.
`VOLATILE`	Puede cambiar en cualquier momento, incluso dentro de una consulta.

Función con Clasificación

Función IMMUTABLE – Cálculo de precio con descuento fijo

Situación real:
El cálculo de un 10% de descuento sobre un precio siempre devuelve lo mismo para el mismo precio. No cambia.

```
CREATE OR REPLACE FUNCTION precio_con_descuento(precio NUMERIC)
RETURNS NUMERIC IMMUTABLE AS $$
BEGIN
    RETURN precio * 0.90;
END;
$$ LANGUAGE plpgsql;
```

Uso:

```
sql

SELECT nombre, precio, precio_con_descuento(precio)
FROM productos;
```

Función con Clasificación

Función STABLE – Fecha actual de consulta

Situación real:
Tabla con productos en dólares y quieres mostrar su precio en moneda local, usando el tipo de cambio del día de la consulta.

```sql
CREATE OR REPLACE FUNCTION tipo_cambio_dolar()
RETURNS numeric STABLE
AS $$
  SELECT tasa_local
  FROM tipo_cambio
  WHERE fecha = CURRENT_DATE;
$$ LANGUAGE sql;

SELECT id, nombre, precio,
precio * tipo_cambio_dolar() AS precio_local
FROM productos;
```

Esta función devuelve un **valor fijo dentro de la consulta**, pero puedes cambiarlo cada día si actualizas la función.

nombre	precio_usd	precio_moneda_local
Queso	5.00	8.75
Leche	1.20	2.10
Yogur	0.80	1.40

📌 Aunque se llama `tipo_cambio_dolar()` en cada fila, **solo se evalúa una vez**, porque es STABLE.

Función con Clasificación

Función VOLATILE – Descuento sorpresa con número aleatorio

Situación real:
En una promoción, cada producto recibe un descuento aleatorio diferente entre 5% y 15%.

```
CREATE OR REPLACE FUNCTION descuento_sorpresa()
RETURNS NUMERIC VOLATILE AS $$
BEGIN
    RETURN 1 - (0.05 + random() * 0.10); -- genera entre 0.85 y 0.95
END;
$$ LANGUAGE plpgsql;
```

Uso:

```
sql

SELECT nombre, precio, precio * descuento_sorpresa() AS precio_final
FROM productos;
```

Aplicación práctica:
Cada producto tiene un descuento aleatorio distinto, incluso en una misma consulta. Perfecto para una promoción dinámica.

Control de Flujos

El control de flujo permite que una función tome decisiones, repita acciones o ejecute condiciones específicas, según los valores de entrada u otras reglas lógicas.

🧰 Herramientas disponibles en PL/pgSQL

Estructura	¿Para qué sirve?
`IF / ELSIF / ELSE`	Decisión condicional simple
`CASE`	Decisión múltiple más limpia que varios `IF`
`LOOP`, `EXIT`, `CONTINUE`	Bucle genérico
`WHILE`	Bucle que repite mientras una condición sea verdadera
`FOR IN`	Iteración sobre un rango o conjunto de resultados

Control de Flujos

🔷 1. IF / ELSIF / ELSE

📜 Ejemplo: Clasificación de calificaciones

```sql
CREATE OR REPLACE FUNCTION clasificar_nota(nota NUMERIC)
RETURNS TEXT AS $$
BEGIN
    IF nota >= 9 THEN
        RETURN 'Excelente';
    ELSIF nota >= 7 THEN
        RETURN 'Bueno';
    ELSIF nota >= 5 THEN
        RETURN 'Regular';
    ELSE
        RETURN 'Reprobado';
    END IF;
END;
$$ LANGUAGE plpgsql;
```

🧪 Uso:

```sql
SELECT clasificar_nota(8); -- 'Bueno'
```

🔷 2. CASE

📜 Ejemplo: Bonificación por cargo

```sql
CREATE OR REPLACE FUNCTION bono_por_cargo(cargo TEXT)
RETURNS NUMERIC AS $$
BEGIN
    RETURN CASE
        WHEN cargo = 'Gerente' THEN 1000
        WHEN cargo = 'Analista' THEN 500
        WHEN cargo = 'Asistente' THEN 200
        ELSE 0
    END;
END;
$$ LANGUAGE plpgsql;
```

🧪 Uso:

```sql
SELECT bono_por_cargo('Analista'); -- 500
```

Control de Flujos

🔷 3. LOOP + EXIT

📜 Ejemplo: Sumar los primeros N números

```sql
CREATE OR REPLACE FUNCTION suma_numeros(n INT)
RETURNS INT AS $$
DECLARE
    total INT := 0;
    i INT := 1;
BEGIN
    LOOP
        total := total + i;
        i := i + 1;

        EXIT WHEN i > n;
    END LOOP;

    RETURN total;
END;
$$ LANGUAGE plpgsql;
```

🧪 Uso:

```sql
SELECT suma_numeros(5); -- 15 (1+2+3+4+5)
```

🔷 4. WHILE

📜 Ejemplo: Buscar el primer número divisible entre 7

```sql
CREATE OR REPLACE FUNCTION primer_divisible_entre_7(desde INT)
RETURNS INT AS $$
BEGIN
    WHILE desde % 7 != 0 LOOP
        desde := desde + 1;
    END LOOP;

    RETURN desde;
END;
$$ LANGUAGE plpgsql;
```

🧪 Uso:

```sql
SELECT primer_divisible_entre_7(10); -- 14
```

Control de Flujos

🔷 5. FOR IN

📜 Ejemplo: Concatenar nombres de productos

```sql
CREATE OR REPLACE FUNCTION concatenar_nombres()
RETURNS TEXT AS $$
DECLARE
    resultado TEXT := '';
    prod RECORD;
BEGIN
    FOR prod IN SELECT nombre FROM productos LOOP
        resultado := resultado || prod.nombre || ', ';
    END LOOP;

    RETURN TRIM(TRAILING ', ' FROM resultado);
END;
$$ LANGUAGE plpgsql;
```

🧪 Uso:

```sql
SELECT concatenar_nombres();
-- Resultado: 'Laptop, Mouse, Teclado'
```

```sql
SELECT TRIM(TRAILING ', ' FROM 'A, B, C, ');    -- 'A, B, C'
SELECT TRIM(TRAILING ', ' FROM 'A, B, C,, ');   -- 'A, B, C'   (quita comas y espacios finales)
SELECT TRIM(TRAILING '0' FROM '1200');          -- '12'
SELECT TRIM(TRAILING FROM '  hola  ');          -- '  hola'    (quita solo espacios finales)
```

Ejercicios

Nivel Básico

1. Clasificación de edades
   Crea una función que clasifique a una persona como "Niño", "Adolescente", "Adulto" o "Adulto mayor" según su edad.
2. Determinar si un número es par o impar
   Crea una función que reciba un número entero y retorne "Par" o "Impar".

Nivel Intermedio

3. Suma de números del 1 hasta N
   Crea una función que retorne la suma acumulada de los números desde 1 hasta N.
4. Contar múltiplos de 3 hasta N
   Crea una función que cuente cuántos números divisibles entre 3 hay desde 1 hasta N.

Nivel Avanzado

5. Concatenar nombres de empleados
   Crea una función que recorra la tabla empleados y devuelva una cadena con todos los nombres separados por comas.
6. Buscar el primer múltiplo de 9 a partir de un número dado
   Crea una función que reciba un número y retorne el primer número igual o mayor que sea divisible entre 9.

Manejo de Errores

¿Qué es un bloque EXCEPTION?

Es una sección dentro de una función en PL/pgSQL que permite capturar y controlar errores en tiempo de ejecución, evitando que la función falle completamente.

Funciona como un try...catch en otros lenguajes de programación.

¿Para qué sirve?

Evitar caídas del sistema ante errores inesperados.
Personalizar mensajes de error.
Proteger datos de operaciones incorrectas.
Devolver valores por defecto o controlados.

Manejo de Errores

Ejemplo 1: División segura

Función que divide dos números. Si el divisor es cero, en lugar de lanzar un error, retorna NULL.

```
CREATE OR REPLACE FUNCTION dividir_seguro(a NUMERIC, b NUMERIC)
RETURNS NUMERIC AS $$
BEGIN
    RETURN a / b;
EXCEPTION
    WHEN division_by_zero THEN
        RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```

Uso:

```
sql

SELECT dividir_seguro(10, 2);   -- Resultado: 5
SELECT dividir_seguro(10, 0);   -- Resultado: NULL (no error)
```

Manejo de Errores

Ejemplo 2: Inserción con control de clave duplicada

Supón que tienes una tabla usuarios con un campo único correo. Esta función intenta insertar, pero si el correo ya existe, no lanza error: simplemente ignora.

```
CREATE OR REPLACE FUNCTION registrar_usuario(nombre TEXT, correo TEXT)
RETURNS TEXT AS $$
BEGIN
    INSERT INTO usuarios(nombre, correo)
    VALUES (nombre, correo);
    RETURN 'Usuario registrado correctamente';
EXCEPTION
    WHEN unique_violation THEN
        RETURN 'Correo ya registrado. No se insertó.';
END;
$$ LANGUAGE plpgsql;
```

Manejo de Errores

Lista de errores comunes que puedes capturar

Error PostgreSQL	Descripción
`division_by_zero`	División entre cero
`unique_violation`	Violación de clave única
`foreign_key_violation`	Violación de clave foránea
`null_value_not_allowed`	Se intentó insertar NULL no permitido
`others`	Cualquier otro error no específico

```plpgsql
CREATE OR REPLACE FUNCTION funcion_general()
RETURNS TEXT AS $$
BEGIN
    -- Alguna lógica que puede fallar
    RAISE NOTICE 'Ejecutando...';
    PERFORM 1 / 0;
    RETURN 'OK';
EXCEPTION
    WHEN OTHERS THEN
        RETURN 'Ocurrió un error inesperado';
END;
$$ LANGUAGE plpgsql;
```

Manejo de Errores

Buenas prácticas

- Siempre maneja errores esperados (como división por cero o claves duplicadas).
- Usa RAISE NOTICE para depuración sin interrumpir la función.
- Evita WHEN OTHERS salvo que realmente quieras capturar cualquier cosa (puede ocultar errores importantes).
- Documenta lo que hace tu bloque EXCEPTION.

Aplicación real

En sistemas reales, el manejo de errores es útil para:

- Procesar cargas masivas sin detener todo por una fila errónea.
- Devolver respuestas personalizadas en APIs.
- Validar datos antes de modificar tablas críticas.

Buenas prácticas

¿Por qué es importante seguir buenas prácticas?

- Para escribir código más claro, mantenible y reutilizable.
- Para evitar errores, mejorar el rendimiento y facilitar el trabajo en equipo.
- Para crear funciones que se integren correctamente con procedimientos, triggers y servicios externos.

Buenas prácticas

A. Cohesión

Una función = una sola responsabilidad

Evita que una función haga múltiples cosas. Por ejemplo:

❌ **Mal diseño:**

```sql
-- Inserta, calcula y actualiza al mismo tiempo
CREATE FUNCTION procesar_venta(...) ...
```

✅ **Buen diseño:**

- Una función para registrar venta.
- Otra para calcular totales.
- Otra para aplicar descuento.

📌 Esto hace que cada función sea más fácil de entender, probar y reutilizar.

B. Reutilización

Evita repetir lógica que ya tienes en otras funciones.

✅ **Recomendado:** Encapsular lógica común en funciones pequeñas reutilizables.

Ejemplo:

```sql
CREATE FUNCTION calcular_igv(monto NUMERIC) RETURNS NUMERIC ...
CREATE FUNCTION total_con_igv(monto NUMERIC) RETURNS NUMERIC AS $$
BEGIN
    RETURN monto + calcular_igv(monto);
END;
$$ LANGUAGE plpgsql;
```

Buenas prácticas

C. Nombrado claro y consistente

- Usa nombres **claros, verbales y descriptivos**.
- Prefiere verbos en funciones (`obtener_`, `calcular_`, `registrar_`).
- Si devuelven un dato, debe ser evidente en su nombre.

Ejemplo:

```sql
-- Claro
SELECT calcular_total_con_descuento(100, 0.1);

-- Confuso
SELECT func123(100, 0.1);
```

D. Documentación mínima dentro del código

Usa comentarios breves en partes importantes del código:

```sql
-- Calcula el descuento aplicable
descuento := monto * porcentaje;
```

También puedes documentar al inicio de cada función:

```sql
-- Función que calcula el IGV del monto dado
```

Buenas prácticas

E. Testeo básico con casos esperados y límites

Prueba tus funciones con:

- Datos típicos
- Casos límite (0, NULL, negativos)
- Entradas erróneas controladas (para ver si se manejan bien)

Ejemplo:

```sql
SELECT dividir_seguro(10, 2);    -- OK
SELECT dividir_seguro(10, 0);    -- Debe retornar NULL sin error
```

F. Evita lógica innecesaria dentro de funciones pequeñas

❌ **No recomendado:**

Usar bucles si solo necesitas una operación simple.

✅ **Mejor:**

Mantén simples las funciones que solo aplican una fórmula o un SELECT.

G. Evita errores comunes

- No dejes funciones sin `RETURN`.
- Si usas `RETURNS TABLE`, asegúrate de seleccionar los mismos campos.
- No mezcles demasiadas operaciones en una sola función (divide y vencerás).

Optimización

¿Por qué es importante optimizar funciones?

- Las funciones mal diseñadas pueden ser muy costosas en términos de rendimiento.
- PostgreSQL no siempre puede optimizar funciones como lo hace con SQL directo, especialmente si están mal clasificadas o usan malas prácticas.
- Optimizar funciones mejora la escalabilidad, la velocidad de respuesta y la experiencia del usuario final (APIs, reportes, dashboards, etc.).

Optimización

¿Qué es EXPLAIN?

Es una instrucción que te muestra cómo PostgreSQL planea ejecutar una consulta (o cómo la ejecutó, si usas EXPLAIN ANALYZE).

Sintaxis básica:

```sql
EXPLAIN ANALYZE SELECT mi_funcion(…);
```

Esto muestra:

Tipo de operación (scan, join, sort)

- Uso de índices o no
- Costo estimado y tiempo real
- Cuántas veces se llama la función (ideal para ver si hay bucles ocultos)

Optimización

B. Clasifica correctamente la función (`IMMUTABLE`, `STABLE`, `VOLATILE`)

Esto ayuda al planificador a **optimizar mejor la ejecución**:

Clasificación	Comportamiento	Optimización posible
`IMMUTABLE`	Siempre mismo resultado para mismos argumentos	Alta (se puede cachear)
`STABLE`	Misma ejecución, pero puede cambiar entre consultas	Media
`VOLATILE`	Puede cambiar en cada llamada	Baja (se evalúa siempre)

Evita marcar como `VOLATILE` una función que en realidad es `IMMUTABLE` o `STABLE`.

C. Usa funciones en SQL puro cuando sea posible

Las funciones PL/pgSQL tienen sobrecarga.

Si solo necesitas un `SELECT`, puedes definir una función **SQL pura**, que es mucho más rápida:

```
CREATE FUNCTION obtener_nombre(id INT)
RETURNS TEXT
AS $$ SELECT nombre FROM empleados WHERE id = $1; $$
LANGUAGE sql;
```

Esto es más eficiente que una función con `BEGIN...END` si no necesitas lógica adicional.

Optimización

D. Evita SELECT dentro de bucles si puedes usar `JOIN`

Ejemplo ineficiente (N consultas dentro de un bucle):

```sql
FOR r IN SELECT * FROM pedidos LOOP
    SELECT nombre INTO cliente FROM clientes WHERE id = r.id_cliente;
END LOOP;
```

Mejor usar un `JOIN` externo con `SELECT ... FROM pedidos JOIN clientes`.

E. Usa índices y verifica que se estén utilizando

Si tus funciones consultan tablas grandes, asegúrate de que:

- Haya índices en las columnas correctas
- Las condiciones usen las columnas indexadas
- No se estén haciendo `CASTS` innecesarios que impidan el uso del índice

---

F. Evita funciones que devuelven muchas filas sin paginación o filtros

Si tu función devuelve una tabla, permite usar parámetros opcionales como `limite`, `orden`, `condición`, etc.

Seguridad

¿Por qué es importante?

En entornos reales (empresas, bancos, universidades), **no todos los usuarios deben tener el mismo nivel de acceso** a los datos o funciones.

Las funciones en PostgreSQL pueden configurarse para que:

- Se ejecuten con los privilegios del **usuario que llama** (modo **invoker**, por defecto), o
- Se ejecuten con los privilegios del **creador de la función** (modo **definer**), lo que permite **ocultar la lógica y restringir el acceso directo a las tablas**.

A. Modos de ejecución de funciones

Modo	Descripción	Sintaxis
`INVOKER`	La función se ejecuta con los privilegios del usuario que la llama	`SECURITY INVOKER` (valor por defecto)
`DEFINER`	La función se ejecuta con los privilegios de su creador	`SECURITY DEFINER`

Seguridad

Ejemplo comparativo

1. Función con `SECURITY DEFINER`

Supón que tienes una tabla `empleados` que contiene datos confidenciales:

```sql
CREATE TABLE empleados (
    id SERIAL PRIMARY KEY,
    nombre TEXT,
    salario NUMERIC
);
```

Y solo el rol `admin_rrhh` tiene acceso a `salario`.

Ahora, quieres permitir que otros roles vean solo nombres de empleados, sin exponer la tabla real.

```sql
CREATE OR REPLACE FUNCTION listar_nombres()
RETURNS TABLE(nombre TEXT)
AS $$
BEGIN
    RETURN QUERY SELECT nombre FROM empleados;
END;
$$ LANGUAGE plpgsql SECURITY DEFINER;
```

🤔 Esto permite que cualquier usuario ejecute esta función, pero **sin tener permiso directo sobre la tabla** `empleados`.

Seguridad

B. Buenas prácticas de seguridad con funciones

1. Usa `SECURITY DEFINER` para funciones que exponen datos de forma controlada

Permite aplicar lógica de negocio (filtros, condiciones) antes de entregar los datos.

2. Revoca el acceso directo a las tablas sensibles

```sql
REVOKE SELECT ON empleados FROM public;
```

Así los usuarios no podrán ver la tabla, pero sí podrán llamar funciones que la consultan de forma segura.

3. Limita qué funciones puede ejecutar cada rol

```sql
GRANT EXECUTE ON FUNCTION listar_nombres() TO rol_consultas;
```

Esto da control preciso sobre qué operaciones puede realizar cada usuario.

Seguridad

4. No uses `SECURITY DEFINER` si la función modifica datos según entradas del usuario, a menos que estés seguro de que están validadas (riesgo de inyección).

D. ¿Qué pasa si no controlas esto?

- Cualquier usuario podría hacer `SELECT * FROM empleados` y ver sueldos, correos o datos sensibles.
- Funciones mal configuradas pueden convertirse en **puertas de acceso a datos críticos**.
- Se rompe el principio de **mínimo privilegio**, base de la seguridad de sistemas.

Seguridad

EJERCICIO PRÁCTICO:

🎯 Objetivo:

Permitir que un **usuario de tipo "consultor"** pueda acceder a información parcial de una tabla sensible (`empleados`) **solo a través de una función, sin tener acceso directo a la tabla.**

Tabla: `empleados`

```sql
CREATE TABLE empleados (
    id SERIAL PRIMARY KEY,
    nombre TEXT NOT NULL,
    dni TEXT NOT NULL,
    salario NUMERIC,
    puesto TEXT
);
```

- Esta tabla tiene datos sensibles como `dni` y `salario`.
- Solo el **rol** `admin_rrhh` tiene acceso total.
- Queremos que el **rol** `consultor_rrhh` **pueda ver solo los nombres y puestos, pero no acceda directamente a la tabla.**

Seguridad

1. Crear función segura

```sql
CREATE OR REPLACE FUNCTION obtener_publico_empleados()
RETURNS TABLE(nombre TEXT, puesto TEXT) AS $$
BEGIN
    RETURN QUERY
    SELECT nombre, puesto FROM empleados;
END;
$$ LANGUAGE plpgsql SECURITY DEFINER;
```

💪 Esta función se ejecutará con los privilegios de su creador (por ejemplo, admin_rrhh), no del usuario que la llama.

2. Revocar el acceso directo a la tabla

```sql
REVOKE ALL ON empleados FROM public;
REVOKE ALL ON empleados FROM consultor_rrhh;
```

Así, el rol `consultor_rrhh` **no puede hacer esto:**

```sql
SELECT * FROM empleados; -- ❌ Error: permiso denegado
```

Seguridad

3. Conceder solo el uso de la función

```sql
GRANT EXECUTE ON FUNCTION obtener_publico_empleados() TO consultor_rrhh;
```

Ahora, ese usuario **puede ejecutar**:

```sql
SELECT * FROM obtener_publico_empleados();
```

Y verá:

nombre	puesto
Ana Pérez	Analista
Juan Ríos	Jefe de área

Sin acceso a `salario`, `dni`, ni siquiera a la tabla directamente.

✅ Resultado

Acción	¿Permitido?	Explicación
`SELECT * FROM empleados`	❌ No	Tabla restringida
`SELECT nombre, puesto FROM empleados`	❌ No	No tiene acceso a columnas directamente
`SELECT * FROM obtener_publico_empleados()`	✅ Sí	Solo accede a datos controlados por la función