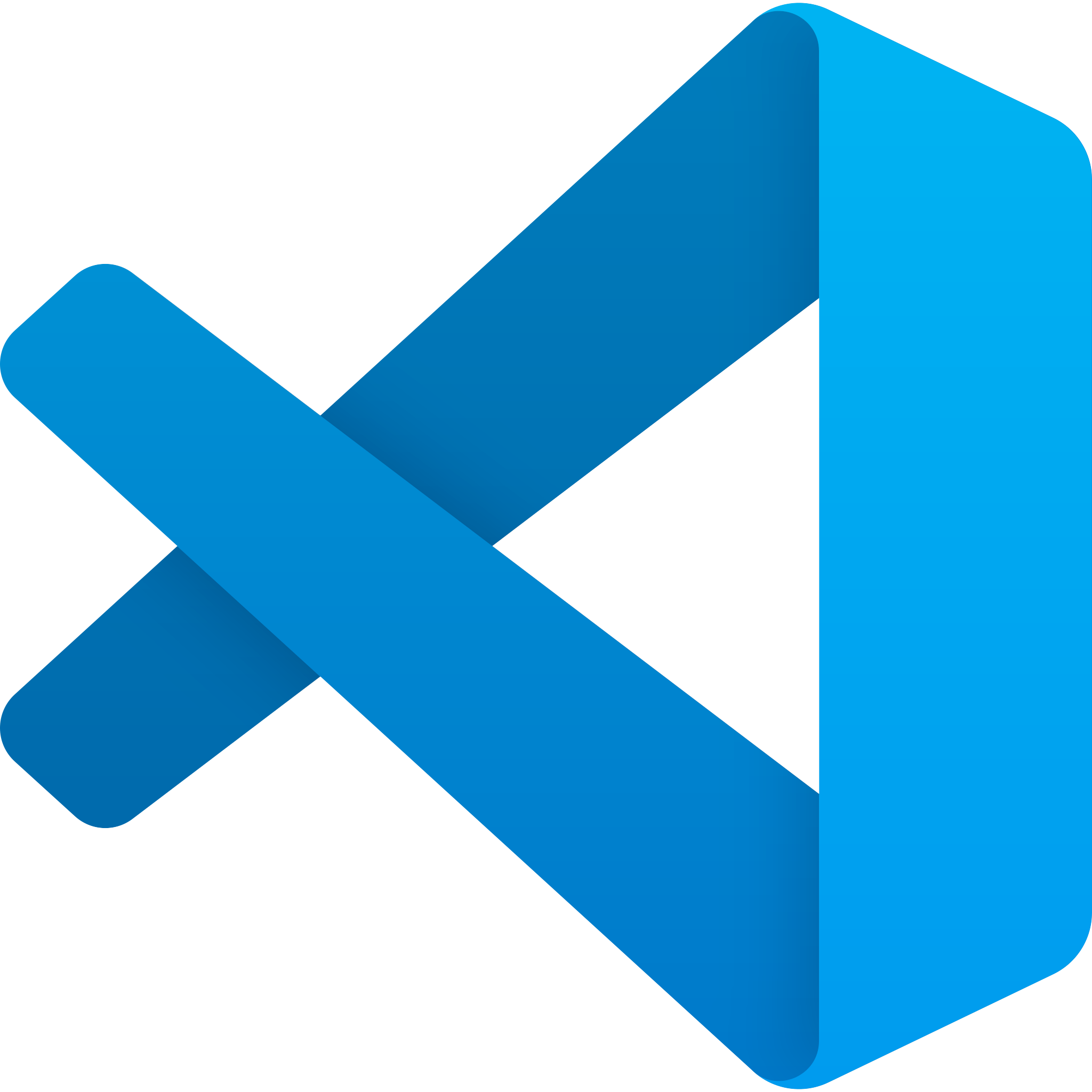
****

****

****

Trabajo Final:

Bases de Datos Avanzadas

Valentín Torassa Colombero1

*1Universidad Abierta Interamericana*

**Autor:** [*valentin.torassacolombero@alumnos.uai.edu.ar*](mailto:valentin.torassacolombero@alumnos.uai.edu.ar)

**Profesores:** Cristian Medin y Nicolas Alejandro Micheletti

**Materia:** Bases de Datos Avanzadas

**Turno:** Noche

# Abstract

*En el presente trabajo final, se ha montado una instancia de* ***PostgreSQL*** *utilizando* ***Docker*** *para facilitar la instalación y ejecución del entorno de base de datos. El documento incluye tanto la* ***teoría como el código SQL desarrollado****, abarcando la creación de tablas, carga de datos, vistas, SP, funciones, triggers y archivos.*

*S*e adjuntan los archivos .sql por separado definición de tablas, inserción de datos, y vistas.

Todo el contenido se encuentra disponible en el repositorio de GitHub:

🔗 <https://github.com/ValentinTorassa/FinalDBA-TorassaColomberoValentin>

**Índice**

[Abstract 2](#__RefHeading___Toc1355_1659058208)

[Teoria 3](#__RefHeading___Toc1357_1659058208)

[Política de Backup 3](#__RefHeading___Toc1359_1659058208)

[Vistas: definición, usos, ventajas y desventajas 3](#__RefHeading___Toc1361_1659058208)

[Stored Procedures (Procedimientos almacenados) 4](#__RefHeading___Toc1363_1659058208)

[Práctica 6](#__RefHeading___Toc1365_1659058208)

[Descripción 6](#__RefHeading___Toc1367_1659058208)

[Vistas funcionales al modelo 6](#__RefHeading___Toc1369_1659058208)

[Procedimiento Almacenado 7](#__RefHeading___Toc1371_1659058208)

[Función 8](#__RefHeading___Toc1373_1659058208)

[Trigger 9](#__RefHeading___Toc1375_1659058208)

[Generación de archivo TXT plano 9](#__RefHeading___Toc1377_1659058208)

[Ejemplo de salida (REGINFO\_UAI\_CBTE.txt) 10](#__RefHeading___Toc1379_1659058208)

[Política de Backups 11](#__RefHeading___Toc1381_1659058208)

[Anexos 13](#__RefHeading___Toc1383_1659058208)

[Docker Compose 13](#__RefHeading___Toc1385_1659058208)

# Teoria

## ****Política de Backup****

Una política de backup en bases de datos se trata de tener una copia de seguridad y de diseñar una estrategia integral que contemple cuándo, cómo, con qué frecuencia y dónde se realiza dicha copia. En entornos donde los datos representan activos críticos (lo cual, hoy en día, es prácticamente en todos), asegurar su disponibilidad y recuperación frente a fallas.

Una política bien pensada contempla backups completos, incrementales, diferenciales y políticas de retención. Debe estar acompañada de pruebas periódicas de restauración. No alcanza con respaldar; es fundamental comprobar que se puede recuperar. En la práctica real, muchas empresas descubren —demasiado tarde— que sus backups no eran restaurables. No es un punto técnico aislado, es parte del plan de continuidad operativa y del compromiso con la seguridad e integridad de la información.

## ****Vistas: definición, usos, ventajas y desventajas****

Las **vistas** representan una abstracción dentro del modelo relacional. En esencia, son consultas predefinidas que se almacenan y pueden usarse como si fueran tablas. Esto permite encapsular lógica compleja, simplificar consultas repetitivas y brindar una capa de seguridad o personalización sobre los datos.

Uno de sus principales usos es el de servir como intermediario entre la base de datos y el usuario. Al diseñar una vista, podemos elegir qué columnas mostrar, aplicar filtros, y hasta unir datos de múltiples tablas sin que el usuario final tenga que conocer la estructura interna del modelo. Las vistas permiten mejorar la experiencia del usuario, ya que enmascaran la complejidad y organizan la información.

Entre sus **ventajas** más importantes se encuentran la mejora en la seguridad (al evitar acceso directo a tablas sensibles), la facilidad de mantenimiento, y la optimización del tiempo de desarrollo al permitir reutilización y eficiencia. También son ideales para integración con sistemas de terceros.

Sin embargo, también presentan **desventajas**. Las vistas complejas pueden perjudicar el rendimiento, especialmente si se anidan muchas vistas unas dentro de otras. Modificar datos a través de vistas puede ser problemático o directamente no estar permitido, dependiendo de su definición y el motor de base de datos. Finalmente, su mantenimiento puede volverse problematico.

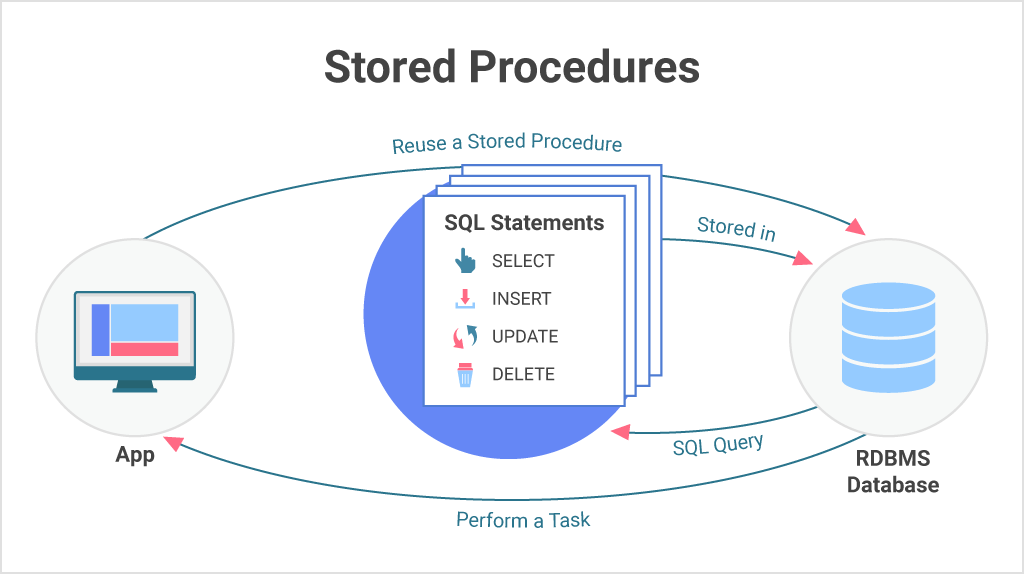
## ****Stored Procedures (Procedimientos almacenados)****

Los **procedimientos almacenados**, o SPs, son bloques de código SQL que encapsulan lógica de negocio dentro del motor de base de datos. Se ejecutan bajo un nombre específico y pueden recibir parámetros de entrada y salida. A diferencia de una consulta común, un SP permite ejecutar múltiples operaciones en una sola llamada, combinando instrucciones DML, control de flujo, validaciones y más.

Su **uso principal** es la automatización de tareas complejas, el procesamiento de transacciones, validaciones previas a la inserción, generación de reportes y centralización de reglas de negocio. Al estar precompilados, ofrecen una mejora notable en rendimiento, especialmente cuando se usan de forma recurrente.

Entre sus **ventajas**, se destaca la capacidad de encapsular lógica y reutilizarla, la reducción del tráfico entre cliente y servidor, y la mejora en la seguridad y control de accesos.

Sin embargo, también tienen **desventajas**: el código queda acoplado al motor (pérdida de portabilidad), su versionado puede ser más complicado que en una aplicación convencional, y no siempre es fácil realizar debugging o pruebas automatizadas.

**Function: definición, usos, ventajas y desventajas**

Las **funciones** en bases de datos son similares a los SPs pero con una diferencia: devuelven un valor (escalar o tabla) y pueden ser utilizadas directamente dentro de consultas SQL. Se usan comúnmente para realizar cálculos, validar datos, transformar valores o encapsular lógica que se repite.

Su **uso** se concentra en consultas complejas donde se requiere transformar o calcular datos, como por ejemplo obtener la base imponible de un comprobante aplicando una fórmula sobre el importe, sin necesidad de escribir dicha fórmula en cada SELECT.

Las **ventajas** de las funciones incluyen su reutilización dentro de cualquier query, su integración sencilla con otras funciones o cláusulas (WHERE, SELECT, GROUP BY) y su bajo impacto sobre la estructura general del sistema.

Entre sus **desventajas** se encuentra el hecho de que, al estar diseñadas para devolver resultados, no pueden modificar datos directamente como un SP.

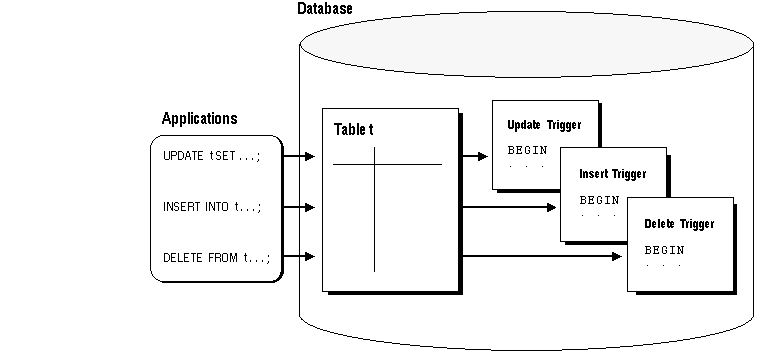
**Trigger: definición, usos, ventajas y desventajas**

Se trata de mecanismos que se ejecutan automáticamente en respuesta a eventos específicos sobre una tabla: inserciones, actualizaciones o eliminaciones.

Los triggers se utilizan para **automatizar tareas**, validar reglas de negocio, mantener consistencia entre tablas, auditar cambios o actualizar campos derivados.

Entre sus **ventajas** se destacan la transparencia (se ejecutan sin que el usuario tenga que intervenir), la centralización de lógica (evita replicar validaciones en todas las aplicaciones), y la capacidad de mantener la integridad referencial y lógica sin depender del cliente que realiza el cambio.

Sin embargo, también presentan **desventajas**. Pueden afectar seriamente el rendimiento si se abusa de ellos o si la lógica no está bien optimizada. También pueden generar cascadas difíciles de seguir, ya que un trigger puede activar otro, y así sucesivamente, lo cual complica el debugging y puede generar efectos colaterales inesperados.



# Práctica

## Descripción

En esta sección se detallan todas las implementaciones realizadas en el entorno PostgreSQL, incluyendo la creación de **vistas funcionales al modelo**, **procedimientos almacenados**, **funciones y triggers**, así como la generación de un **archivo TXT plano** a partir del esquema de comprobantes.

## Vistas funcionales al modelo

### Vista 1 – Comprobantes Detallados

**CREATE** **OR** **REPLACE** **VIEW** vista\_comprobantes\_detallada **AS**

**SELECT**

**c**.nro\_comprobante,

**c**.fecha,

cl.apellido\_nombre **AS** cliente,

cl.nro\_documento,

tc.descrip\_comprobante,

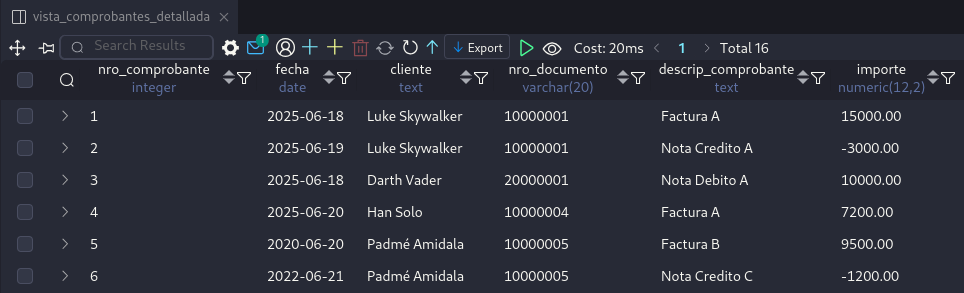
**c**.importe

**FROM** comprobantes **c**

**JOIN** clientes cl **ON** **c**.id\_cliente **=** cl.id\_cliente

**JOIN** tipos\_comprobantes tc **ON** **c**.id\_tipo\_comprobante **=** tc.id\_tipo\_comprobante;

Esta vista permite obtener todos los comprobantes con sus datos descriptivos asociados, incluyendo el cliente y el tipo de comprobante.



### Vista 2 – Totales por Cliente

**CREATE** **OR** **REPLACE** **VIEW** vista\_totales\_por\_cliente **AS**

**SELECT**

cl.id\_cliente,

cl.apellido\_nombre,

**SUM**(**c**.importe) **AS** total\_facturado

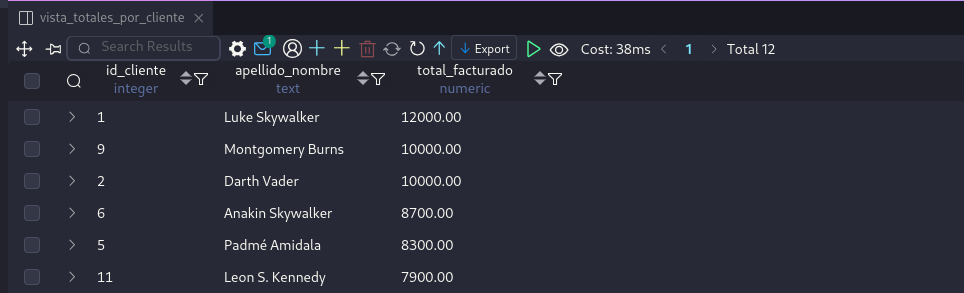
**FROM** comprobantes **c**

**JOIN** clientes cl **ON** **c**.id\_cliente **=** cl.id\_cliente

**GROUP** **BY** cl.id\_cliente, cl.apellido\_nombre

**ORDER** **BY** total\_facturado **DESC**;

Se calcula el total facturado por cada cliente en base a los comprobantes registrados, ordenado de mayor a menor.



## Procedimiento Almacenado

### Total facturado en rango de fechas

**CREATE** **OR** **REPLACE** **PROCEDURE** total\_facturado\_rango(

**IN** fecha\_inicio DATE,

**IN** fecha\_fin DATE

)

**LANGUAGE** plpgsql

**AS** $$

**BEGIN**

RAISE NOTICE 'Total facturado del % al %: %',

fecha\_inicio,

fecha\_fin,

**COALESCE**((

**SELECT** **SUM**(importe)

**FROM** comprobantes

**WHERE** fecha **BETWEEN** fecha\_inicio **AND** fecha\_fin

), 0);

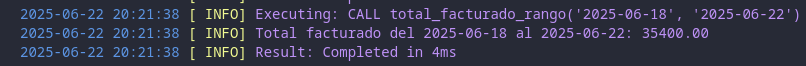
**END**;

$$;

*-- Ejecucion*

**CALL** total\_facturado\_rango('2025-06-18', '2025-06-22');

Calcula el total facturado en un período definido, utilizando una consulta agregada y mostrando el resultado como mensaje del servidor.



## Función

### Base Imponible (IVA 21%)

**CREATE** **OR** **REPLACE** **FUNCTION** calcular\_base\_imponible(nro INTEGER)

**RETURNS** **TABLE** (

nro\_comprobante INTEGER,

fecha DATE,

cliente TEXT,

tipo\_comprobante TEXT,

importe NUMERIC,

base\_imponible NUMERIC

) **AS** $$

**BEGIN**

**RETURN** QUERY

**SELECT**

**c**.nro\_comprobante,

**c**.fecha,

cl.apellido\_nombre,

tc.descrip\_comprobante,

**c**.importe,

ROUND(**c**.importe **/** 1.21, 2) **AS** base\_imponible

**FROM** comprobantes **c**

**JOIN** clientes cl **ON** **c**.id\_cliente **=** cl.id\_cliente

**JOIN** tipos\_comprobantes tc **ON** **c**.id\_tipo\_comprobante **=** tc.id\_tipo\_comprobante

**WHERE** **c**.nro\_comprobante **=** nro;

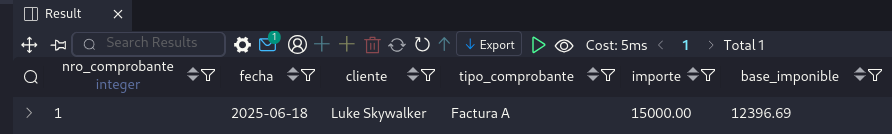
**END**;

$$ **LANGUAGE** plpgsql;

*-- Ejecucion*

**SELECT** **\*** **FROM** calcular\_base\_imponible(1);

Esta función devuelve todos los datos relevantes de un comprobante, incluyendo su base imponible neta de IVA calculada al 21%.



## Trigger

### Acumulado de Compras

#### Paso 1: Crear función del trigger

**CREATE** **OR** **REPLACE** **FUNCTION** actualizar\_acumulado()

**RETURNS** **TRIGGER** **AS** $$

**BEGIN**

**UPDATE** clientes

**SET** acumulado\_compras **=** acumulado\_compras **+** **NEW**.importe

**WHERE** id\_cliente **=** **NEW**.id\_cliente;

**RETURN** **NEW**;

**END**;

$$ **LANGUAGE** plpgsql;

#### Paso 2: Crear trigger sobre comprobantes

**CREATE** **TRIGGER** trg\_actualizar\_acumulado

**AFTER** **INSERT** **ON** comprobantes

**FOR** **EACH** **ROW**

**EXECUTE** **FUNCTION** actualizar\_acumulado();

Al insertar un nuevo comprobante, se actualiza automáticamente el atributo acumulado\_compras del cliente relacionado, manteniendo un valor agregado por cada carga.

## Generación de archivo TXT plano

### Consulta SQL para generación de líneas TXT

**SELECT**

LPAD(TO\_CHAR(**c**.fecha, 'YYYYMMDD'), 8, '0') **||**

LPAD(**c**.id\_tipo\_comprobante::text, 3, '0') **||**

LPAD(**c**.nro\_comprobante::text, 20, '0') **||**

LPAD(**c**.nro\_comprobante::text, 20, '0') **||**

LPAD(cl.id\_tipo\_documento::text, 2, '0') **||**

LPAD(cl.nro\_documento, 20, '0') **||**

RPAD(cl.apellido\_nombre, 30, ' ') **||**

LPAD(**REPLACE**(TO\_CHAR(**c**.importe, 'FM999999999990.00'), '.', ''), 15, '0') **AS** linea\_txt

**FROM** comprobantes **c**

**JOIN** clientes cl **ON** **c**.id\_cliente **=** cl.id\_cliente

**ORDER** **BY** **c**.nro\_comprobante;

### 

### Script Python para exportar el archivo

import psycopg2

conn = psycopg2.connect(

host="localhost",

port="5432",

dbname="finaldb",

user="admin",

password="admin123"

)

cursor = conn.cursor()

query = """

SELECT

LPAD(TO\_CHAR(c.fecha, 'YYYYMMDD'), 8, '0') ||

LPAD(c.id\_tipo\_comprobante::text, 3, '0') ||

LPAD(c.nro\_comprobante::text, 20, '0') ||

LPAD(c.nro\_comprobante::text, 20, '0') ||

LPAD(cl.id\_tipo\_documento::text, 2, '0') ||

LPAD(cl.nro\_documento, 20, '0') ||

RPAD(cl.apellido\_nombre, 30, ' ') ||

LPAD(REPLACE(TO\_CHAR(c.importe, 'FM999999999990.00'), '.', ''), 15, '0') AS linea\_txt

FROM comprobantes c

JOIN clientes cl ON c.id\_cliente = cl.id\_cliente

ORDER BY c.nro\_comprobante;

"""

cursor.execute(query)

with open("REGINFO\_UAI\_CBTE.txt", "w", encoding="utf-8") as file:

for row in cursor.fetchall():

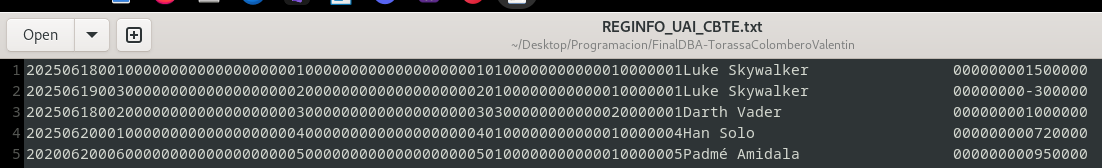
file.write(row[0] + "\n")

print("Archivo REGINFO\_UAI\_CBTE.txt generado correctamente.")

cursor.close()

conn.close()

## Ejemplo de salida (REGINFO\_UAI\_CBTE.txt)



## Política de Backups

Esta política garantiza la **disponibilidad, integridad y recuperabilidad de la base de datos** finaldb utilizada en el entorno Docker con PostgreSQL.

Se establecen dos tipos de backup:

* **Backups incrementales diarios**: ejecutados de lunes a sábado.
* **Backups totales (full)**: ejecutados todos los domingos.

### Estructura del sistema de backups

/home/valen/backups/

├── diarios/

│ ├── 2025-07-01/ finaldb\_incremental.sql

├── semanales/

│ ├── 2025-07-06/ finaldb\_full.sql

└── logs/ backup.log

* Los backups **incrementales** contienen datos actualizados por día, pensados para recuperación parcial.
* Los backups **totales** contienen un volcado completo de la base de datos, útil para restauraciones integrales.

### Script de Backup

Este script realiza automáticamente la copia de seguridad adecuada según el día de la semana, y deja un registro en un archivo de log.

#### backup\_pgsql\_Valen.sh

*#!/bin/bash*

**BASE**="/home/valen/backups"

**LOG**="**$BASE**/logs/backup.log"

**DATE**=**$(**date +%Y-%m-%d**)**

**DAY**=**$(**date +%u**)**

**if** [ "**$DAY**" -eq 7 ]; **then**

**DIR**="**$BASE**/semanales/**$DATE**"

**FILE**="finaldb\_full.sql"

**TYPE**="FULL"

**else**

**DIR**="**$BASE**/diarios/**$DATE**"

**FILE**="finaldb\_incremental.sql"

**TYPE**="INCREMENTAL"

mkdir -p "**$DIR**" "**$BASE**/logs"

docker exec -t postgres-final pg\_dump -U admin -d finaldb > "**$DIR**/**$FILE**"

echo "[**$(**date '+%F %T'**)**] Backup **$TYPE** guardado en **$DIR**/**$FILE**" >> "**$LOG**"

Este script realiza un backup automático de la base de datos finaldb en Docker. Detecta el día de la semana si es domingo (7), hace un backup completo; si no, hace uno incremental. Según eso, define el tipo (TYPE), el nombre del archivo y la carpeta destino.

Luego, ejecuta pg\_dump dentro del contenedor postgres-final y guarda el archivo en la ubicación correspondiente. Finalmente, registra la acción en un log indicando la fecha, hora, tipo de backup y ruta del archivo generado.

### Automatización con cron

Para que el script se ejecute todos los días a las **02:00 AM**, se configura una tarea cron.

2 \* \* \* /home/valen/scripts/backup\_pgsql.sh

### Restauración del backup

Para restaurar manualmente un archivo .sql generado, se puede usar:

cat /home/valenbackups/semanales/2025-07-06/finaldb\_full.sql | \

docker exec -i postgres-final psql -U admin -d finaldb

También puede aplicarse a un backup incremental si se requiere revertir cambios parciales.

# Anexos

## Docker Compose

version: '3.8'

services:

db:

image: postgres:15

container\_name: postgres-final

ports:

- "5432:5432"

environment:

POSTGRES\_DB: finaldb

POSTGRES\_USER: admin

POSTGRES\_PASSWORD: admin123

volumes:

- pgdata:/var/lib/postgresql/data

- ./init:/docker-entrypoint-initdb.d

restart: unless-stopped

volumes:

pgdata:

## Estructura Carpetas GIT

