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, ejecución y portabilidad 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**](#_fjxf18wv2341)

[**Introduccion 3**](#_8w9jlfb0xjhk)

[**Desarrollo 4**](#_c9qg9tpvbg3a)

[4.1 Contexto organizacional 4](#_bqy5vpae492o)

[4.2 Politica de negocios 6](#_h4e7skltfe2b)

[4.3 Cultura y valores 9](#_9zfd4nshidc2)

[4.4 Cambio organizacional 15](#_kldqv6d8uw85)

[4.5 Estructura organizacional y diseño de puestos 16](#_i8h8apdhmwro)

[4.6 Toma de decisiones 19](#_vrqlq49iwy5m)

[**Conclusion 20**](#_pw03zjdohxcg)

[**Referencias 21**](#_1l7rwjsxqo9a)

# Teoria

**Política de Backup**

Una política de backup en bases de datos no se trata únicamente de tener una copia de seguridad, sino 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 se vuelve una responsabilidad irrenunciable.

Una política bien pensada contempla backups completos, incrementales, diferenciales y políticas de retención. Además, 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. En resumen, 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 poderosa 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. Además, las vistas permiten mejorar la experiencia del usuario, ya que enmascaran la complejidad y organizan la información de forma clara y precisa.

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. 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. Además, 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 complejo si no se documentan adecuadamente.

**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. Además, 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. También es importante mencionar que permiten realizar operaciones atómicas y coordinadas bajo un mismo contexto transaccional.

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 sobre ellos. Aún así, su valor en entornos profesionales es incuestionable.

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

Las **funciones** en bases de datos son similares a los SPs pero con una diferencia clave: 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. También pueden ser declaradas como IMMUTABLE o STABLE para mejorar el rendimiento.

Entre sus **desventajas** se encuentra el hecho de que, al estar diseñadas para devolver resultados, no pueden modificar datos directamente como un SP, y que su abuso puede introducir sobrecarga de cálculo en grandes volúmenes de datos si no se implementan de forma eficiente.

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

Los **triggers** son uno de los componentes más potentes —y delicados— de una base de datos. Se trata de mecanismos que se ejecutan automáticamente en respuesta a eventos específicos sobre una tabla: inserciones, actualizaciones o eliminaciones. Representan el paradigma de programación reactiva a nivel de datos.

Los triggers se utilizan para **automatizar tareas**, validar reglas de negocio, mantener consistencia entre tablas, auditar cambios o actualizar campos derivados. Por ejemplo, si queremos que cada vez que se inserte un comprobante se actualice automáticamente el acumulado de compras de ese cliente, un trigger es la herramienta adecuada.

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 **riesgos y 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. Además, al ejecutarse automáticamente, los errores pueden ser silenciosos si no se monitorean adecuadamente.

En resumen, los triggers son una herramienta potentísima, pero que debe usarse con criterio, planificación y una documentación clara que permita mantener el sistema sostenible a lo largo del tiempo.

¡Perfecto, Valentín! A continuación te dejo la **documentación completa** en formato markdown o Word-ready, ideal para incluir como **sección de desarrollo práctico** en tu entrega final.

# 📄 Desarrollo Práctico

## 🔍 Descripción General

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. El objetivo fue integrar distintos componentes de SQL avanzado y demostrar su aplicación práctica sobre un modelo relacional.

## 📊 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;

🔍 **Descripción**: 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;

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

## 🛠 Procedimiento Almacenado

### 💰 SP – 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;

$$;

-- Ejecución

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

🔍 **Descripción**: Calcula el total facturado en un período definido, utilizando una única 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;

-- Ejecución

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

🔍 **Descripción**: 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();

🔍 **Descripción**: 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 (formato fijo)

### 📋 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 esperada (REGINFO\_UAI\_CBTE.txt)

202307010010000000000000000010000000000000000000011020000000003201237991HOMERO SIMPSON 000000000750000

¿Querés que te prepare un README.md listo para subir al repo y que resuma todo esto con secciones claras y ejemplos? También podemos incluir capturas de pantalla o fragmentos del .txt generado.

Perfecto, Valentín. A continuación te presento la **implementación documentada profesionalmente de la política de backup**, con secciones claras, explicaciones entre los fragmentos de código, y una estructura pensada para ser incluida en tu entrega final o repositorio (README.md o documento PDF/Word).

# 🛡 Política de Backups – Implementación práctica en Debian

## 📌 Objetivo

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

Los backups se almacenan organizados por tipo y fecha:

/home/valentin/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 – backup\_pgsql.sh

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

📍 **Ubicación sugerida**: /home/valentin/scripts/backup\_pgsql.sh

#!/bin/bash

# Configuración de rutas

BASE\_DIR="/home/valentin/backups"

LOG\_DIR="$BASE\_DIR/logs"

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

DAY\_OF\_WEEK=$(date +%u) # 1 = lunes ... 7 = domingo

# Crear carpeta de logs si no existe

mkdir -p "$LOG\_DIR"

# Determinar tipo de backup según día

if [ "$DAY\_OF\_WEEK" -eq 7 ]; then

# Domingo → backup total

TARGET\_DIR="$BASE\_DIR/semanales/$DATE"

FILENAME="finaldb\_full.sql"

TYPE="FULL"

else

# Lunes a sábado → backup incremental

TARGET\_DIR="$BASE\_DIR/diarios/$DATE"

FILENAME="finaldb\_incremental.sql"

TYPE="INCREMENTAL"

fi

# Crear carpeta destino si no existe

mkdir -p "$TARGET\_DIR"

# Ejecutar backup lógico con pg\_dump dentro del contenedor Docker

docker exec -t postgres-final pg\_dump -U admin -d finaldb > "$TARGET\_DIR/$FILENAME"

# Escribir log

echo "[$(date '+%Y-%m-%d %H:%M:%S')] Backup $TYPE generado en $TARGET\_DIR/$FILENAME" >> "$LOG\_DIR/backup.log"

## 🔑 Permisos de ejecución

Asignar permisos de ejecución al script:

chmod +x /home/valentin/scripts/backup\_pgsql.sh

## 📅 Automatización con cron

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

Editar las tareas del usuario:

crontab -e

Agregar la siguiente línea:

0 2 \* \* \* /home/valentin/scripts/backup\_pgsql.sh

✔ Esto asegura la **ejecución automática diaria**, sin necesidad de intervención manual.

## ♻️ (Opcional) Rotación de backups antiguos

Se puede extender el script para eliminar automáticamente backups viejos y liberar espacio.

Agregar al final del script:

# Eliminar backups incrementales con más de 7 días

find "$BASE\_DIR/diarios/" -type d -mtime +7 -exec rm -rf {} \;

# Eliminar backups semanales con más de 30 días

find "$BASE\_DIR/semanales/" -type d -mtime +30 -exec rm -rf {} \;

## 🔄 Restauración del backup

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

cat /home/valentin/backups/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.

## ✅ Conclusión

Esta política de backup ofrece una solución práctica, automatizada y estructurada para proteger la integridad de la base de datos. Es **fácilmente adaptable** a entornos productivos y académicos, y asegura que siempre exista una copia reciente tanto para restauración parcial como total.

¿Querés que esta documentación se genere como archivo README\_BACKUP.md, o que preparemos el backup\_pgsql.sh con los cambios y lo empaquetemos todo listo para el repo?