



Trabajo Final:
Bases de Datos Avanzadas

Valentín Torassa Colombero¹

¹*Universidad Abierta Interamericana*

Autor: 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.*

Se 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
Teoria.....	3
Política de Backup.....	3
Vistas: definición, usos, ventajas y desventajas.....	3
Stored Procedures (Procedimientos almacenados).....	4
Práctica.....	6
Descripción.....	6
Vistas funcionales al modelo.....	6
Procedimiento Almacenado.....	7
Función.....	8
Trigger.....	9
Generación de archivo TXT plano.....	9
Ejemplo de salida (REGINFO_UAI_CBTE.txt).....	10
Política de Backups.....	11
Anexos.....	13
Docker Compose.....	13



Teoría

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 problemático.



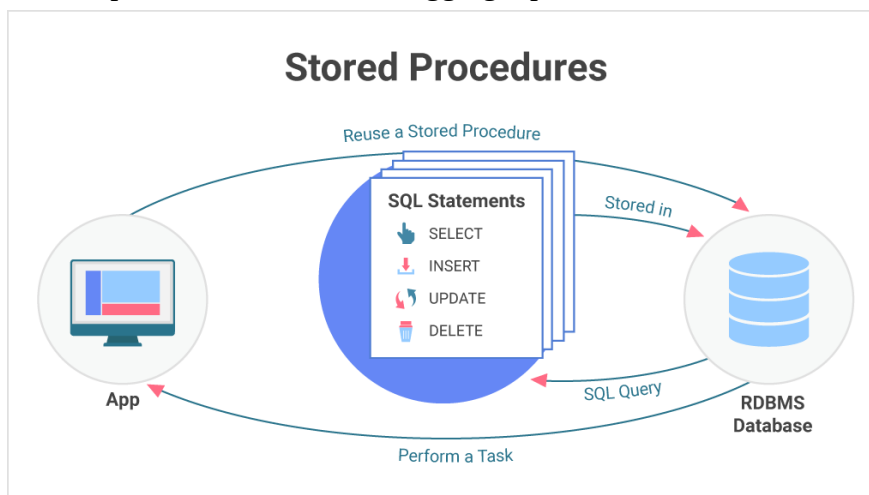
Stored Procedures (Procedimientos almacenados)

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.



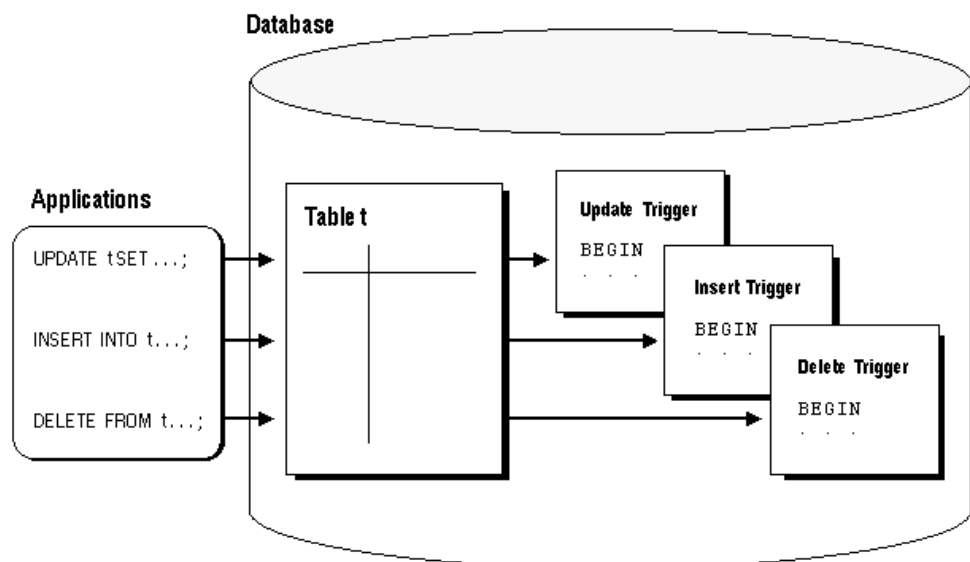
Trigger: definición, usos, ventajas y desventajas

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.

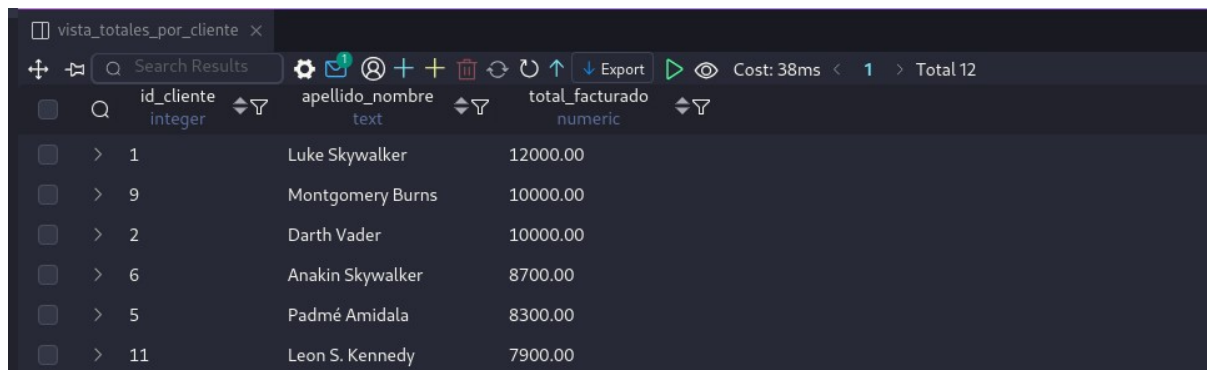
vista_comprobantes_detallada						
	nro_comprobante integer	fecha date	cliente text	nro_documento varchar(20)	descrip_comprobante text	importe numeric(12,2)
>	1	2025-06-18	Luke Skywalker	10000001	Factura A	15000.00
>	2	2025-06-19	Luke Skywalker	10000001	Nota Credito A	-3000.00
>	3	2025-06-18	Darth Vader	20000001	Nota Debito A	10000.00
>	4	2025-06-20	Han Solo	10000004	Factura A	7200.00
>	5	2020-06-20	Padmé Amidala	10000005	Factura B	9500.00
>	6	2022-06-21	Padmé Amidala	10000005	Nota Credito C	-1200.00



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.



id_cliente	apellido_nombre	total_facturado
1	Luke Skywalker	12000.00
9	Montgomery Burns	10000.00
2	Darth Vader	10000.00
6	Anakin Skywalker	8700.00
5	Padmé Amidala	8300.00
11	Leon S. Kennedy	7900.00

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.

```
2025-06-22 20:21:38 [ INFO] Executing: CALL total_facturado_rango('2025-06-18', '2025-06-22')
2025-06-22 20:21:38 [ INFO] Total facturado del 2025-06-18 al 2025-06-22: 35400.00
2025-06-22 20:21:38 [ INFO] Result: Completed in 4ms
```

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 $$
BEGIN
    RETURN QUERY
    SELECT
        c.nro_comprobante,
        c.fecha,
        cl.apellido_nombre,
        tc.descripcion_comprobante,
        c.importe,
        ROUND(c.importe / 1.21, 2) AS base_imponible
    FROM comprobantes c
    JOIN clientes cl ON c.id_cliente = cl.id_cliente
    JOIN tipos_comprobantes tc ON c.id_tipo_comprobante =
tc.id_tipo_comprobante
    WHERE c.nro_comprobante = nro;
END;
$$ 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%.

Result					
Search Results					
nro_comprobante integer	fecha	cliente	tipo_comprobante	importe	base_imponible
> 1	2025-06-18	Luke Skywalker	Factura A	15000.00	12396.69



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.

	id_cliente integer	apellido_nombre text	id_tipo_documento integer	nro_documento varchar(20)	acumulado_compras numeric(12,2)
1		Luke Skywalker	1	10000001	15000.00

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, 'FM9999999999990.00'), '.', ''), 15,
'0') AS linea_txt
FROM comprobantes c
JOIN clientes cl ON c.id_cliente = cl.id_cliente
ORDER BY c.nro_comprobante;
```

The screenshot shows a SQL query editor with a search bar containing 'linea_txt'. The search results are displayed in a table with two columns: 'linea_txt' and a value consisting of a long string of zeros followed by 'Luke Skywalker'.



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"
fi

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

