# Tema extra (Continuacion Tema 5) - Triggers y demás

David Moreno Lumbreras & Daniela Patricia Feversani
GSyC, EIF. URJC.

Laboratorio de Bases de Datos (BBDD)

Curso 2024-2025


# Contenidos

# E.1 Triggers

# ¿Qué son los Triggers?

- Los **Triggers** son mecanismos de bases de datos que ejecutan acciones automáticamente cuando ocurre un evento específico (*INSERT*, *UPDATE*, *DELETE*) en una tabla.
- Tipos de triggers según el momento de ejecución:
  - **BEFORE**: Se ejecutan antes del evento.
  - **AFTER**: Se ejecutan después del evento.
- Se utilizan para:
  - Automatizar procesos.
  - Garantizar la integridad de los datos.
  - Registrar auditorías o aplicar reglas de negocio.

# Motivación para el Trigger: Auditoría en Clientes (BPSimple)

- **Objetivo:** Registrar los cambios realizados en los datos de los clientes para fines de auditoría.
- **Uso:** Controlar quién realiza los cambios y qué valores se modificaron.

# Auditoría de Cambios en Clientes (BPSimple)

Registrar actualizaciones en la tabla `customer`

```
CREATE TABLE customer_audit (
        audit_id SERIAL PRIMARY KEY,
        customer_id INT,
        old_fname VARCHAR(32),
        old_lname VARCHAR(32),
        new_fname VARCHAR(32),
        new_lname VARCHAR(32),
        change_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE OR REPLACE FUNCTION audit_customer_update()
RETURNS TRIGGER AS $$
BEGIN
        INSERT INTO customer_audit (customer_id, old_fname, old_lname,
            new_fname, new_lname)
        VALUES (OLD.customer_id, OLD.fname, OLD.lname, NEW.fname, NEW.lname);
        RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER after_update_customer
AFTER UPDATE ON customer
FOR EACH ROW
EXECUTE FUNCTION audit_customer_update();
```

# Explicación del Trigger: Auditoría en Clientes

- Este trigger se ejecuta después de cada actualización en la tabla `customer`.
- Inserta un registro en la tabla `customer_audit` con los valores anteriores y nuevos de las columnas actualizadas.

# Motivación para el Trigger: Validación de Matrículas (Universidad)

- **Objetivo:** Evitar que un estudiante se matricule más de una vez en la misma asignatura.
- **Uso:** Garantizar la consistencia e integridad de los datos de matrículas.

# Validación de Matrículas (Universidad)

Evitar duplicados en la tabla `Matriculas`

```
CREATE OR REPLACE FUNCTION validar_matricula()
RETURNS TRIGGER AS $$
DECLARE
        matricula_existente INT;
BEGIN
        SELECT COUNT(*) INTO matricula_existente
        FROM Matriculas
        WHERE Matricula = NEW.Matricula AND IDAsignatura = NEW.IDAsignatura;

        IF matricula_existente > 0 THEN
                RAISE EXCEPTION 'Estudiante ya matriculado';
        END IF;

        RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER before_insert_matriculas
BEFORE INSERT ON Matriculas
FOR EACH ROW
EXECUTE FUNCTION validar_matricula();
```

# Explicación del Trigger: Validación de Matrículas

- Este trigger se ejecuta antes de insertar una nueva matrícula.
- Valida que no exista ya un registro para el estudiante y la asignatura.
- En caso de duplicado, lanza un error y evita la inserción.

# Motivación para el Trigger: Control de Inventario (DVDRental)

- **Objetivo:** Actualizar automáticamente el inventario cuando se realiza un alquiler.
- **Uso:** Mantener consistencia en los datos del inventario sin necesidad de procesos manuales.

# Control de Inventario en Alquileres (DVDRental)

Actualizar inventario en la tabla `inventory`

```
CREATE OR REPLACE FUNCTION actualizar_inventario()
RETURNS TRIGGER AS $$
BEGIN
        UPDATE inventory
        SET quantity = quantity - 1
        WHERE inventory_id = NEW.inventory_id;

        RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER after_insert_rental
AFTER INSERT ON rental
FOR EACH ROW
EXECUTE FUNCTION actualizar_inventario();
```

# Explicación del Trigger: Control de Inventario

- Este trigger se ejecuta después de insertar un nuevo alquiler.
- Decrementa automáticamente el inventario disponible para el artículo alquilado.
- Asegura que el inventario esté siempre actualizado.

# E.2 Procedimientos Almacenados

# Procedimientos Almacenados

- Los **procedimientos almacenados** (*Stored Procedures*) son bloques de código que se almacenan y se ejecutan directamente en la base de datos.
- Se usan para automatizar procesos repetitivos, reglas de negocio y operaciones complejas.
- Ventajas principales:
  - Mejora del rendimiento al reducir el tráfico entre cliente y servidor.
  - Reutilización del código.
  - Facilita la gestión y el mantenimiento de reglas de negocio.
- Soportan parámetros de entrada y salida.

# Motivación para el Procedimiento: Total de Ventas por Cliente (BPSimple)

- **Objetivo:** Calcular el total de ventas realizadas por un cliente.
- **Uso:** Generar reportes rápidos sobre el comportamiento de compra de los clientes.

# Total de Ventas por Cliente (BPSimple)

```
CREATE OR REPLACE FUNCTION obtener_total_ventas(cliente_id INT)
RETURNS NUMERIC AS $$
DECLARE
        total NUMERIC;
BEGIN
        SELECT SUM(total_amount) INTO total
        FROM invoice
        WHERE customer_id = cliente_id;

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

**Ejecución:**

```
SELECT obtener_total_ventas(1);
```

# Explicación del Procedimiento: Total de Ventas por Cliente

- AS $$ ... $$ Es un delimitador utilizado para encapsular el bloque de código.
- LANGUAGE plpgsql; Especifica el lenguaje de programación que se usará para definir el procedimiento o función, en este caso es el lenguaje de procedimientos de PostgreSQL basado en SQL.
- Este procedimiento recibe el ID de un cliente como parámetro de entrada.
- Calcula la suma del campo `total_amount` de la tabla `invoice` para ese cliente.
- Devuelve el total como un valor de retorno.

# Motivación para el Procedimiento: Promedio de Notas por Estudiante (Universidad)

- **Objetivo:** Calcular el promedio de las calificaciones obtenidas por un estudiante en todas sus asignaturas.
- **Uso:** Evaluar el desempeño académico de los estudiantes.

# Promedio de Notas por Estudiante (Universidad)

```
CREATE OR REPLACE FUNCTION calcular_promedio_notas(matricula INT)
RETURNS NUMERIC AS $$
DECLARE
        promedio NUMERIC;
BEGIN
        SELECT AVG(Calificacion) INTO promedio
        FROM Matriculas
        WHERE Matricula = matricula;

        RETURN promedio;
END;
$$ LANGUAGE plpgsql;
```

**Ejecución:**

```
SELECT calcular_promedio_notas(1001);
```

# Explicación del Procedimiento: Promedio de Notas por Estudiante

- Este procedimiento recibe el ID de un estudiante (`Matricula`) como parámetro de entrada.
- Calcula el promedio de las calificaciones almacenadas en la tabla `Matriculas` para ese estudiante.
- Devuelve el promedio como un valor de retorno.

# Motivación para el Procedimiento: Actualizar Inventario (DVDRental)

- **Objetivo:** Disminuir la cantidad disponible de un artículo tras realizar un alquiler.
- **Uso:** Mantener actualizado el inventario de manera eficiente.

# Actualizar Inventario (DVDRental)

```
CREATE OR REPLACE PROCEDURE actualizar_inventario(inventario_id INT)
LANGUAGE plpgsql
AS $$
BEGIN
        UPDATE inventory
        SET quantity = quantity - 1
        WHERE inventory_id = inventario_id;
END;
$$;
```

**Ejecución:**

```
CALL actualizar_inventario(20);
```

# Explicación del Procedimiento: Actualizar Inventario

- Este procedimiento recibe el ID del inventario como parámetro de entrada.
- Disminuye en 1 la cantidad disponible (`quantity`) para el artículo correspondiente en la tabla `inventory`.
- Ayuda a mantener el inventario sincronizado tras un alquiler.

# E.3 Transacciones (aun más)

## Conceptos clave

- Una **transacción** es una unidad de trabajo que agrupa una o más operaciones de base de datos, tratándolas como una sola operación lógica.
- Propiedades fundamentales: **ACID**.
  - **Atomicidad**: La transacción es indivisible; o se ejecuta por completo o no se ejecuta.
  - **Consistencia**: La base de datos pasa de un estado válido a otro estado válido.
  - **Aislamiento**: Las transacciones concurrentes no interfieren entre sí.
  - **Durabilidad**: Los cambios realizados por una transacción confirmada son permanentes.
- Las transacciones son gestionadas con `BEGIN`, `COMMIT` y `ROLLBACK`.

# Manejo de Transacciones con BEGIN, COMMIT y ROLLBACK

- **BEGIN:** Indica el inicio de una transacción.
- **COMMIT:** Confirma los cambios realizados durante la transacción, haciéndolos permanentes.
- **ROLLBACK:** Revierte todos los cambios realizados desde el inicio de la transacción.
- **Casos comunes de errores:**
  - Fallo de integridad referencial (clave foránea).
  - Violación de restricciones `UNIQUE` o `NOT NULL`.
  - Errores en los cálculos o lógica de negocio.
- Si ocurre un error en cualquier paso, `ROLLBACK` asegura que la base de datos vuelva a su estado original.

# Motivación para el Uso de Transacciones: Gestión de Clientes (BPSimple)

- **Objetivo:** Asegurar la integridad de los datos al actualizar información de un cliente.
- **Uso:** Evitar inconsistencias como actualizar parcialmente los datos de un cliente.

# Gestión de Clientes (BPSimple)

```
BEGIN;

-- Actualizar nombre y direccion del cliente
UPDATE customer
SET fname = 'John',
    address = '123 New Street'
WHERE customer_id = 1;

-- Confirmar la transaccion
COMMIT;
```

**Si ocurre un error:**

```
ROLLBACK;
```

# Explicación de la Transacción: Gestión de Clientes

- La transacción asegura que todos los cambios en los datos del cliente se realicen juntos.
- Si ocurre un error, todos los cambios se revierten.

# Motivación para el Uso de Transacciones: Gestión de Matrículas (Universidad)

- **Objetivo:** Asegurar que los datos de matrícula y las actualizaciones relacionadas sean consistentes.
- **Uso:** Garantizar que las asignaturas se registren correctamente y se actualicen los cupos.

# Gestión de Matrículas (Universidad)

```
BEGIN;

-- Insertar matricula del estudiante
INSERT INTO Matriculas (Matricula, IDAsignatura, Fecha)
VALUES (1001, 101, CURRENT_DATE);

-- Reducir cupo disponible de la asignatura
UPDATE Asignaturas
SET CuposDisponibles = CuposDisponibles - 1
WHERE IDAsignatura = 101;

-- Confirmar la transaccion
COMMIT;
```

**Si ocurre un error:**

```
ROLLBACK;
```

# Explicación de la Transacción: Gestión de Matrículas

- La transacción asegura que la matrícula y la reducción del cupo se ejecuten juntas.
- Si ocurre un error al actualizar el cupo, la matrícula también se revierte.

# Motivación para el Uso de Transacciones: Gestión de Inventario (DVDRental)

- **Objetivo:** Asegurar que el registro de un alquiler y la actualización del inventario sean consistentes.
- **Uso:** Evitar errores como registrar un alquiler sin descontar el inventario.

# Gestión de Inventario en Alquileres (DVDRental)

```
BEGIN;

-- Insertar nuevo alquiler
INSERT INTO rental (rental_date, inventory_id, customer_id, return_date)
VALUES (CURRENT_DATE, 10, 1, NULL);

-- Reducir inventario disponible
UPDATE inventory
SET quantity = quantity - 1
WHERE inventory_id = 10;

-- Confirmar la transaccion
COMMIT;
```

**Si ocurre un error:**

```
ROLLBACK;
```

# Explicación de la Transacción: Gestión de Inventario

- La transacción asegura que el registro del alquiler y la reducción del inventario se realicen juntos.
- Si ocurre un error en cualquiera de los pasos, todos los cambios se revierten.

# Bibliografía I


[Connolly & Begg, 2015] Connolly, T., Begg, C.
*Database Systems*.
Pearson, 6th Global Edition. 2015.


[Petrov, 2019] Petrov, A.
*Database Internals*.
O'Reilly Media, 2019.