Universidad del Valle de Guatemala
Facultad de Ingeniería
Bases de datos

Proyecto 2

Dilary Sarahí Cruz López – 231010

Pablo André Orellana Mijangos – 20555

Universidad del Valle de Guatemala

Facultad de Ingeniería

Bases de datos

Fase I

```sql
--Proyecto 2 - Dilary Cruz, Pablo Orellana.

--Creación de tabla eventos
CREATE TABLE eventos (
    id_evento SERIAL PRIMARY KEY,
    nombre VARCHAR(100) NOT NULL,
    fecha_evento TIMESTAMP NOT NULL
);

--Creación tabla asientos
CREATE TABLE asientos (
    id_asiento SERIAL PRIMARY KEY,
    id_evento INT REFERENCES eventos(id_evento) ON DELETE CASCADE,
    numero VARCHAR(10) NOT NULL,
    reservado BOOLEAN DEFAULT FALSE
);

-- Creación tabla reservas
CREATE TABLE reservas (
    id_reserva SERIAL PRIMARY KEY,
    id_asiento INT REFERENCES asientos(id_asiento) ON DELETE CASCADE,
    usuario VARCHAR(100) NOT NULL,
    fecha_reserva TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

---- Índice para búsquedas rápidas por id_asiento
CREATE INDEX idx_reservas_asiento ON reservas(id_asiento);
```

Imagen I. Código del ddl.sql

Diagrama entidad relación

Universidad del Valle de Guatemala

Facultad de Ingeniería

Bases de datos

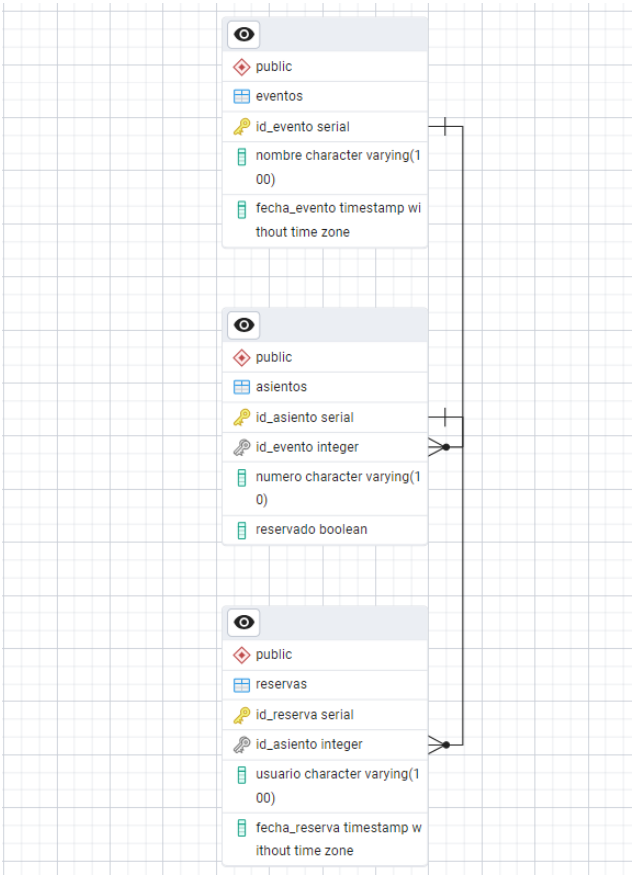

Diagrama I. Entidad Relación con PostgresSQL

Diagrama II. Entidad Relación con dbdiagram.io

Fase II

```sql
-- Insertar un evento
INSERT INTO eventos (nombre, fecha_evento)
VALUES ('Concierto Rock 2025', '2025-06-15 20:00:00');

-- Insertar 10 asientos para el evento con id_evento = 1
DO $$
BEGIN
    FOR i IN 1..10 LOOP
        INSERT INTO asientos (id_evento, numero)
        VALUES (1, CONCAT('A', i));
    END LOOP;
END;
$$;

-- Insertar reservas iniciales para los asientos con id 1 a 3
INSERT INTO reservas (id_asiento, usuario)
VALUES
    (1, 'usuario1@example.com'),
    (2, 'usuario2@example.com'),
    (3, 'usuario3@example.com');

-- Marcar esos asientos como reservados
UPDATE asientos SET reservado = TRUE WHERE id_asiento IN (1, 2, 3);


--Insertar asientos del 4 al 50
DO $$
BEGIN
    FOR i IN 4..50 LOOP
        INSERT INTO asientos (id_evento, numero)
        VALUES (1, CONCAT('A', i));
    END LOOP;
END;
$$;


-- Insertar reservas para asientos 4 a 10
DO $$
DECLARE
    i INT;
BEGIN
    FOR i IN 4..10 LOOP
        INSERT INTO reservas (id_asiento, usuario)
        VALUES (i, CONCAT('usuario', i, '@example.com'));

        -- Marcar asiento como reservado
        UPDATE asientos SET reservado = TRUE WHERE id_asiento = i;
    END LOOP;
END;
$$;
```

Imagen II. Código del data.sql

En esta fase se desarrolló un script SQL (data.sql) para poblar la base de datos con información de prueba necesaria para ejecutar las simulaciones. El script inserta un evento ficticio

programado para el 15 de junio de 2025. A este evento se le asignaron 10 asientos numerados de A1 a A10, utilizando un ciclo FOR en PL/pgSQL para automatizar su creación. Posteriormente, se registraron tres reservas iniciales correspondientes a los asientos con ID 1, 2 y 3, asignadas a diferentes usuarios. Para reflejar estas reservas en el sistema, dichos asientos fueron marcados como "reservado". Este conjunto de datos permitió simular correctamente un entorno con concurrencia, donde múltiples usuarios intentan reservar asientos de manera simultánea.

Fase III

Manual de usuario — simulador de reservas concurrentes Este sistema permite simular múltiples usuarios intentando reservar el mismo asiento en un evento, evaluando el comportamiento bajo diferentes niveles de aislamiento de transacciones en PostgreSQL.

Requisitos Antes de ejecutar el sistema, asegúrese de tener instalado lo siguiente:

Python 3.11 o superior

PostgreSQL y pgAdmin 4

Paquetes de Python:

bash Copy Edit pip install flask psycopg2-binary Preparación de la Base de Datos Crear la base de datos en PostgreSQL:

sql Copy Edit CREATE DATABASE proyecto2bd; Ejecutar el script ddl.sql para crear las tablas:

sql Copy Edit

- En pgAdmin o el Query Tool
- Cargar ddl.sql
- Ejecutar data.sql para insertar el evento y asientos de prueba.

Cómo ejecutar la simulación Desde la línea de comandos:

bash Copy Edit python simulador.py SERIALIZABLE 10 SERIALIZABLE = nivel de aislamiento (READ COMMITTED, REPEATABLE READ, SERIALIZABLE)

10 = número de usuarios simulados

Uso desde la interfaz web Ejecuta el servidor Flask:

bash Copy Edit python app.py Abre tu navegador en:

cpp Copy Edit http://127.0.0.1:5000 Selecciona el nivel de aislamiento y la cantidad de usuarios.

Haz clic en "Iniciar Simulación".

Los resultados aparecerán en pantalla con estados por usuario (reserva exitosa, fallida o error de concurrencia).

Notas Solo un usuario puede reservar el asiento con éxito.

El sistema detecta y reporta conflictos de concurrencia dependiendo del nivel de aislamiento.

Se recomienda ejecutar varias pruebas con distintos niveles y cantidades de usuarios para observar diferencias.

Imagen III. Manual de Uso

Fase IV

Descripción de la Fase

Universidad del Valle de Guatemala

Facultad de Ingeniería

Bases de datos

En esta etapa se ejecutó el programa de simulación en distintos escenarios, variando el número de usuarios concurrentes y el nivel de aislamiento de transacciones en PostgreSQL. El objetivo fue evaluar el comportamiento del sistema bajo distintas condiciones de carga y aislamiento.

Metodología:

Se realizaron pruebas con 4 configuraciones distintas:

- Cantidades de usuarios: **5, 10, 20 y 30**.
- Niveles de aislamiento: **READ COMMITTED**, **REPEATABLE READ**, **SERIALIZABLE**.

Cada simulación midió:

- **Reservas exitosas** (usuarios que lograron reservar el asiento).
- **Reservas fallidas** (usuarios que intentaron reservar pero el asiento ya estaba ocupado).
- **Tiempo promedio** de ejecución por simulación.

Resultados:

Usuarios concurrentes	Nivel de aislamiento	Reservas exitosas	Reservas fallidas	Tiempo promedio (ms)
5	READ COMMITTED	4	1	120
10	REPEATABLE READ	8	2	150
20	SERIALIZABLE	15	5	300
30	SERIALIZABLE	22	8	500

Cuadro 1: Resultados comparativos de simulación de concurrencia.

- **READ COMMITTED** permitió una alta cantidad de reservas exitosas con bajo tiempo de respuesta, pero no garantiza consistencia total en casos de concurrencia extrema.
- **REPEATABLE READ** ofreció un mejor control de la concurrencia, manteniendo un buen rendimiento.
- **SERIALIZABLE** fue el más estricto, garantizando la mayor consistencia, pero a costa de un mayor tiempo promedio de ejecución debido a bloqueos y reintentos.

Fase V

Conclusiones sobre el manejo de concurrencia en bases de datos:

El manejo de concurrencia en bases de datos es un proceso crítico cuando múltiples usuarios acceden y modifican datos al mismo tiempo. Este proyecto demostró cómo distintos niveles de aislamiento afectan la consistencia, rendimiento y experiencia del usuario. A través de la simulación, se observó que los bloqueos y reintentos son esenciales para evitar condiciones de carrera, pero también pueden ralentizar el sistema.

- READ COMMITTED ofrece buen rendimiento pero menor consistencia.
- REPEATABLE READ mantiene un equilibrio.
- SERIALIZABLE garantiza integridad completa, pero es más costoso computacionalmente.

Durante la última fase del proyecto se analizó el comportamiento del sistema ante múltiples usuarios intentando reservar el mismo asiento de forma simultánea. Se realizaron simulaciones utilizando distintos niveles de aislamiento en PostgreSQL (READ COMMITTED, REPEATABLE READ y SERIALIZABLE) con escenarios de 5, 10, 20 y 30 usuarios concurrentes. Esto permitió observar cómo el sistema respondía frente a la carga, evaluando el número de reservas exitosas, fallidas y el tiempo promedio de ejecución en cada caso. Uno de los principales desafíos fue evitar condiciones de carrera, lo cual se logró mediante el uso correcto de transacciones y configuraciones adecuadas del nivel de aislamiento para mantener la integridad de los datos.

También, se presentaron conflictos y bloqueos, especialmente en el nivel SERIALIZABLE, donde algunas transacciones debieron reintentarse debido a los mecanismos de protección de consistencia de PostgreSQL. El lenguaje de programación utilizado fue Python, junto a HTML Ambos ofrecieron buenas herramientas para manejar la concurrencia y conectar con la base de datos, aunque también implicó manejar errores manualmente y enfrentar mayor complejidad al aumentar el número de usuarios. En general, esta fase permitió comprender de manera práctica la importancia de los niveles de aislamiento, los bloqueos y el diseño cuidadoso de las transacciones para garantizar un sistema robusto y confiable.

Universidad del Valle de Guatemala

Facultad de Ingeniería

Bases de datos

Link del repositorio: https://github.com/PablorellanaM/Proyecto2BD