UNIVERSIDAD DEL VALLE DE GUATEMALA

Bases de Datos 1
Sección 30

Proyecto 2 Base de Datos
ACID - Concurrencias

Facultad de Ingeniería

Anggie Quezada 23643
Gabriel Bran 23590
David Domínguez 23712

Guatemala
2025

Introducción

Este trabajo presenta el diseño completo de la base de datos que será utilizada en el sistema de gestión de eventos, incluyendo la administración de asientos y reservas, conforme a los principios fundamentales de bases de datos como la integridad de datos, las relaciones entre entidades, las restricciones necesarias, y la preparación para aplicar conceptos de transacciones, concurrencia y propiedades ACID. Este diseño establece la estructura sobre la cual se desarrollarán las funcionalidades del proyecto y se realizarán pruebas en fases posteriores.

Fase 1: Diseño de la Base de Datos

Entidades Identificadas

1. Evento:Representa un evento específico al que se le pueden asignar múltiples asientos.

Atributos:

- id: Identificador único del evento (clave primaria).
- nombre: Nombre del evento.
- fecha: Fecha y hora en la que se realizará el evento.

2. Asiento: Representa un asiento específico dentro de un evento.

Atributos:

- id: Identificador único del asiento (clave primaria).
- evento_id: Referencia al evento al que pertenece el asiento (clave foránea).
- numero: Número de asiento dentro del evento.
- estado: Estado del asiento (por ejemplo, 'disponible', 'reservado').

3. Reserva: Representa la acción de un usuario reservando un asiento para un evento.

Atributos:

- id: Identificador único de la reserva (clave primaria).
- usuario_id: ID del usuario que realiza la reserva (simulado en este proyecto).
- asiento_id: Referencia al asiento reservado (clave foránea).
- fecha_reserva: Fecha y hora en que se hizo la reserva.

Relaciones entre las Tablas

1. Un evento puede tener muchos asientos (1:N).
   Un asiento pertenece a un solo evento.
2. Un evento puede tener muchas reservas (1:N).
   Una reserva pertenece a un solo evento.
3. Un asiento puede estar relacionado con muchas reservas, pero solo una reserva puede estar activa para un asiento (1:1 por evento).

Se asegura que un asiento no pueda ser reservado más de una vez para el mismo evento.

Restricciones de Integridad

1. Identificadores únicos:
   - Cada entidad posee un campo id como **clave primaria**.

2. Integridad referencial:
   - asiento.evento_id es clave foránea que referencia a evento.id.
   - reserva.asiento_id es clave foránea que referencia a asiento.id.

3. Restricciones de unicidad en asientos:
   - Se asegura que no haya dos asientos con el mismo número dentro de un mismo evento mediante la restricción: UNIQUE (evento_id, numero)

4. Estados del asiento:
   - Se controla si un asiento está disponible o reservado mediante el atributo estado.

Script SQL

```
-- Tabla Evento
CREATE TABLE evento (
    id SERIAL PRIMARY KEY,
    nombre VARCHAR(100) NOT NULL,
    fecha TIMESTAMP NOT NULL
);
-- Tabla Asiento
CREATE TABLE asiento (
    id SERIAL PRIMARY KEY,
    evento_id INTEGER NOT NULL,
    numero INTEGER NOT NULL,
    estado VARCHAR(20) DEFAULT 'disponible',
    FOREIGN KEY (evento_id) REFERENCES evento(id),
    UNIQUE (evento_id, numero)
```

);

-- Tabla Reserva

CREATE TABLE reserva (

id SERIAL PRIMARY KEY,

usuario_id INTEGER NOT NULL,

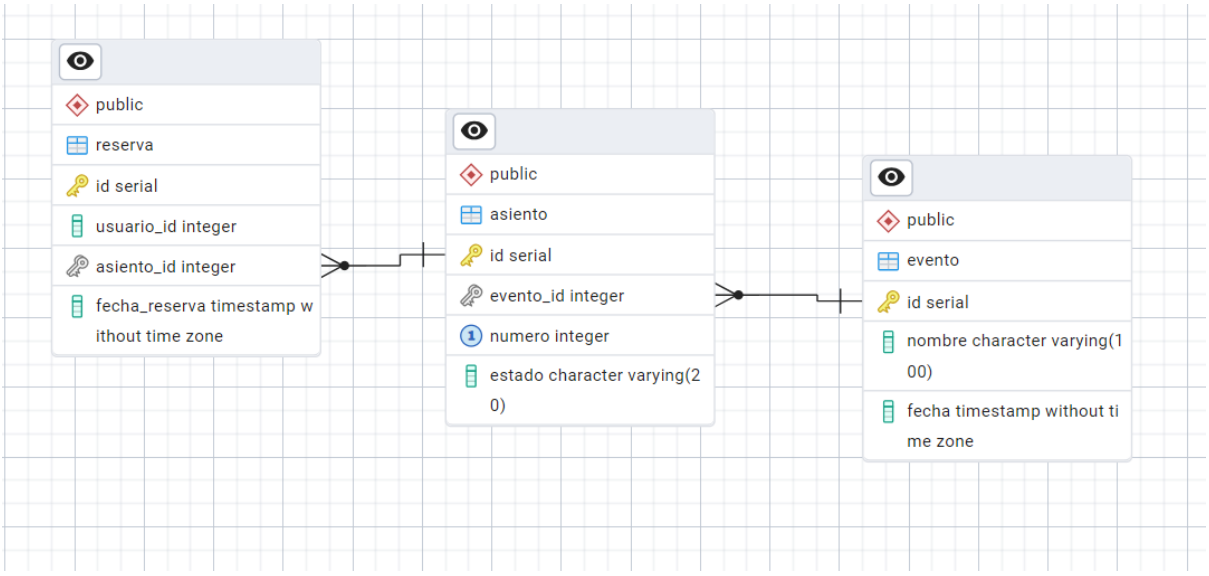
asiento_id INTEGER NOT NULL,

fecha_reserva TIMESTAMP DEFAULT NOW(),

FOREIGN KEY (asiento_id) REFERENCES asiento(id)

);

Diagrama entidad relación



Fase 2: Generación de datos de prueba

-- Insertar un evento

INSERT INTO evento (nombre, fecha) VALUES ('Concierto UVG 2025', '2025-05-20 18:00:00');

-- Insertar 100 asientos para el evento (id=1)

INSERT INTO asiento (evento_id, numero)

SELECT 1, generate_series(1, 100);

-- Reservar algunos asientos iniciales (en este caso 10 reservas)

INSERT INTO reserva (usuario_id, asiento_id)

SELECT floor(random() * 1000 + 1), id

FROM asiento

WHERE numero IN (1, 5, 10, 15, 20, 25, 30, 35, 40, 45);

Fase 3: Implementación del programa de simulación

Git del programa: https://github.com/Qu3zada22/proyecto2_basededatos.git

Instrucciones sobre la ejecución del programa dentro del README.md del github.

Fase 4: Experimentación y Pruebas

Se realizaron simulador desarrollado, con el fin de comparar el comportamiento del sistema bajo diferentes niveles de aislamiento y cantidades de usuarios simultáneos. Cada prueba consistió en lanzar múltiples hilos que intentaban reservar un asiento al mismo tiempo, evaluando cuántas reservas se completaban exitosamente, cuántas fallaban, y cuál era el tiempo promedio por transacción.

Usuarios	Aislamiento	Exitosas	Fallidas	Tiempo Promedio
5	READ COMMITTED	15	-10	407.92 ms
5	REPEATABLE READ	5	0	5.63 ms
5	SERIALIZABLE	1	4	5.35 ms
10	READ COMMITTED	10	0	6.65 ms
10	REPEATABLE READ	10	0	5.17 ms
10	SERIALIZABLE	3	7	5.24 ms
20	READ COMMITTED	20	0	5.90 ms
20	REPEATABLE READ	18	2	6.29 ms
20	SERIALIZABLE	6	14	5.98 ms
30	READ COMMITTED	30	0	7.56 ms
30	REPEATABLE READ	0	30	6.82 ms
30	SERIALIZABLE	0	30	10.34 ms

El nivel READ COMMITTED mostró el mejor rendimiento en términos de reservas exitosas y tiempo promedio, aunque con mayor riesgo de inconsistencias. Es ideal para sistemas que priorizan velocidad sobre consistencia estricta. REPEATABLE READ funcionó bien con baja concurrencia, pero falló completamente con 30 usuarios simultáneos debido a sus mecanismos de bloqueo más restrictivos. Por su parte, SERIALIZABLE, que ofrece la máxima consistencia,

presentó una alta tasa de fallos al aumentar la concurrencia ya que PostgreSQL cancela transacciones potencialmente conflictivas para mantener la integridad de los datos.

Fase 5: Análisis y Reflexión

Conclusiones sobre el manejo de concurrencia en bases de datos.

El manejo de concurrencia en bases de datos es un aspecto crucial al diseñar sistemas que permiten acceso simultáneo a recursos compartidos. A través de este proyecto, se evidenció cómo diferentes niveles de aislamiento impactan directamente en la integridad de los datos, la disponibilidad del sistema, el rendimiento general. READ COMMITTED permitió completar todas las reservas incluso con 30 usuarios concurrentes, lo hizo sin garantías fuertes de consistencia. En cambio, SERIALIZABLE, a pesar de ser el nivel más seguro, falló masivamente bajo alta concurrencia.

Análisis

El principal desafío fue garantizar que cada usuario reservara un asiento de forma segura y sin conflictos, especialmente al trabajar con múltiples hilos simultáneamente. Fue necesario manejar adecuadamente las transacciones y niveles de aislamiento.

Se presentaron varios tipos de errores relacionados con bloqueos y serialización, especialmente con SERIALIZABLE. Estos errores ocurrieron cuando múltiples transacciones intentaban leer y escribir los mismos datos al mismo tiempo. PostgreSQL, para preservar la consistencia, aborta las transacciones conflictivas, lo cual redujo el número de reservas exitosas.

Basado en los resultados, el nivel más eficiente en este contexto fue READ COMMITTED, ya que permitió la mayor cantidad de reservas exitosas y tiempos promedio bajos. Sin embargo, su falta de control sobre condiciones de carrera lo hace poco adecuado si se requiere una consistencia estricta.

Se eligió Python para implementar la simulación, junto con la librería psycopg2 para conectar con PostgreSQL. Las principales ventajas fueron: sintaxis clara y rápida de implementar, excelente soporte para concurrencia con threading y facilidad para manejar conexiones y errores. Sin embargo, también presentó desventajas: el módulo threading de Python está limitado por el Global Interpreter Lock (GIL), lo que puede limitar el verdadero paralelismo en CPUs multinúcleo, y para escenarios de alta concurrencia real.