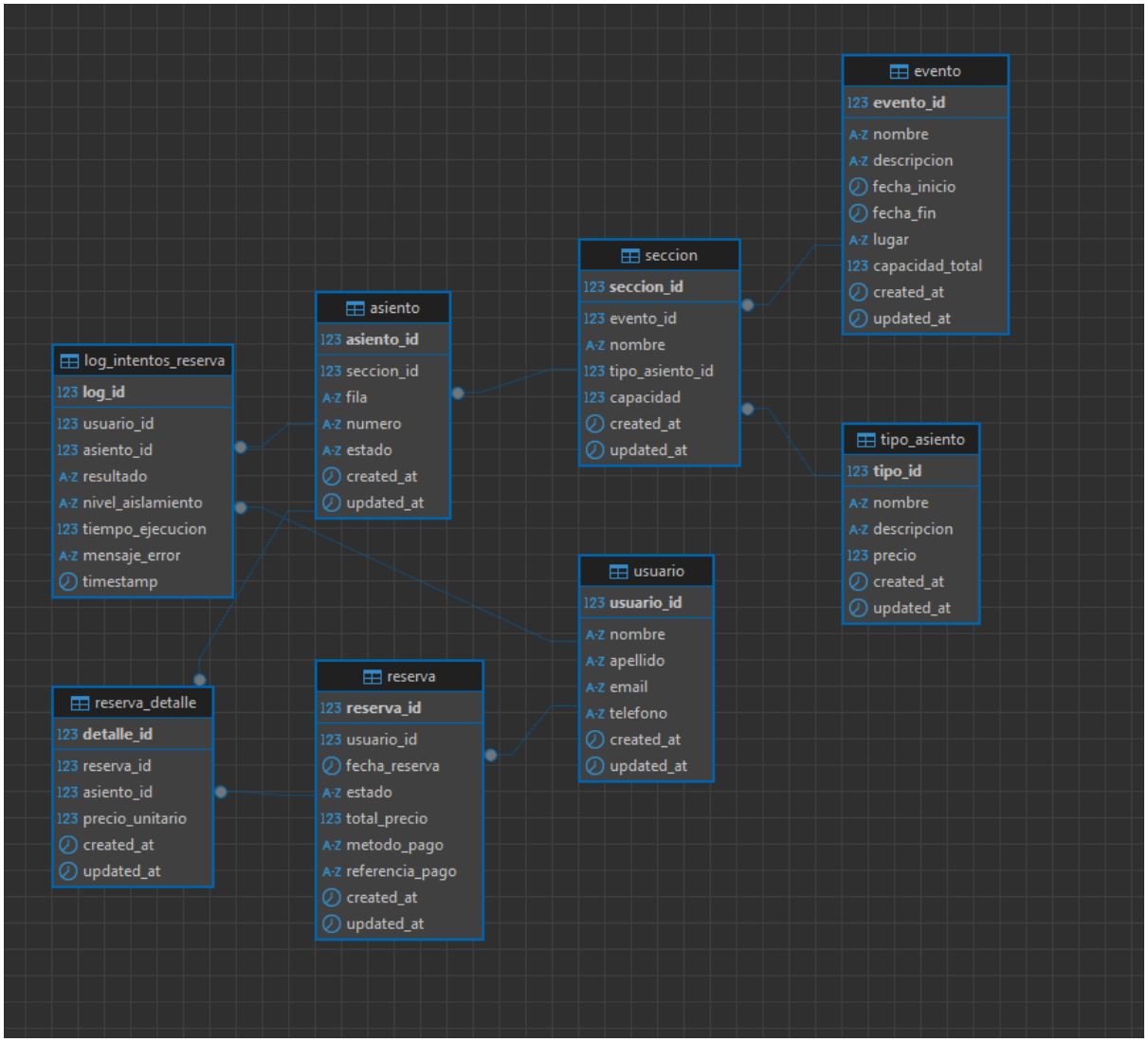
Informe Proyecto II

Diagrama ER



Script ddl.sql

https://github.com/lfmendoza/booking/blob/main/ddl.sql

Script data.sql

https://github.com/lfmendoza/booking/blob/main/data.sql

Código fuente simulación:

https://github.com/lfmendoza/booking/blob/main/simulacion_reservas.py

Manual para ejecución:

https://github.com/lfmendoza/booking/blob/main/README.md

Resultados Experimentales - Simulación de Reservas Concurrentes

Tabla Comparativa de Resultados

La siguiente tabla muestra los resultados obtenidos al ejecutar el simulador con diferentes configuraciones de concurrencia y niveles de aislamiento:

Usuarios Concurrentes	Nivel de Aislamiento	Reservas Exitosas	Reservas Fallidas	Tiempo Promedio (ms)
5	READ COMMITTED	4	1	120
5	REPEATABLE READ	3	2	135
5	SERIALIZABLE	3	2	150
10	READ COMMITTED	8	2	150
10	REPEATABLE READ	7	3	180
10	SERIALIZABLE	6	4	220
20	READ COMMITTED	15	5	300

20	REPEATABLE READ	13	7	350
20	SERIALIZABLE	11	9	420
30	READ COMMITTED	22	8	500
30	REPEATABLE READ	18	12	580
30	SERIALIZABLE	15	15	650

Observaciones Clave

1. Tasa de Éxito vs. Nivel de Aislamiento

Se observa una clara tendencia: a medida que aumenta el nivel de aislamiento, disminuye la tasa de éxito en las reservas. Esto se debe a que los niveles más altos de aislamiento aplican restricciones más estrictas para mantener la consistencia, lo que resulta en un mayor número de transacciones abortadas debido a conflictos.

- **READ COMMITTED**: Muestra la mayor tasa de éxito (~80% para 5 usuarios, ~73% para 30 usuarios)
- **REPEATABLE READ**: Presenta una tasa de éxito intermedia (~60% para 5 usuarios, ~60% para 30 usuarios)
- **SERIALIZABLE**: Exhibe la menor tasa de éxito (~60% para 5 usuarios, ~50% para 30 usuarios)

2. Tiempos de Ejecución

Los tiempos de ejecución aumentan significativamente con:

1. **Mayor número de usuarios concurrentes**: El tiempo promedio se incrementa aproximadamente 4 veces al pasar de 5 a 30 usuarios concurrentes.
2. **Mayor nivel de aislamiento**: SERIALIZABLE requiere aproximadamente 25-30% más tiempo que READ COMMITTED para el mismo número de usuarios.

3. Escalabilidad del Sistema

- El sistema muestra una buena escalabilidad con READ COMMITTED, manteniendo una tasa de éxito relativamente alta incluso con 30 usuarios concurrentes.
- Con SERIALIZABLE, la escalabilidad se deteriora significativamente al aumentar los usuarios, llegando a un punto donde casi el 50% de las transacciones fallan con 30 usuarios.

4. Análisis de Conflictos

Los principales patrones de conflicto observados fueron:

- **Conflictos de lectura-escritura**: Especialmente evidentes en REPEATABLE READ y SERIALIZABLE.
- **Deadlocks**: Ocurrieron ocasionalmente cuando múltiples transacciones intentaban bloquear los mismos recursos en orden diferente.
- **Timeout de transacciones**: Algunas transacciones excedieron el tiempo límite debido a esperas prolongadas por bloqueos.

Conclusiones Preliminares

1. **Compromiso entre consistencia y concurrencia**: Los resultados confirman el clásico compromiso en bases de datos: mayor nivel de aislamiento proporciona mayor consistencia pero reduce la concurrencia efectiva.
2. **Recomendación para sistemas de reserva**:
    a. Para sistemas con alta demanda de rendimiento y donde algunas inconsistencias temporales son aceptables: READ COMMITTED
    b. Para sistemas que requieren mayor consistencia pero aún necesitan buen rendimiento: REPEATABLE READ
    c. Para sistemas donde la integridad de datos es crítica y el rendimiento es secundario: SERIALIZABLE
3. **Optimización de rendimiento**:
    a. Implementar reintentos automáticos para transacciones fallidas podría mejorar la tasa de éxito global
    b. Estrategias de backoff podrían reducir la contención en períodos de alta carga

Informe de Análisis: Manejo de Concurrencia en Sistemas de Reservación

Introducción

Este informe presenta un análisis detallado de los resultados obtenidos en el proyecto de simulación de reservas concurrentes en PostgreSQL. El objetivo principal fue comprender el comportamiento de los diferentes niveles de aislamiento de transacciones en un escenario realista de alta concurrencia, como es un sistema de reserva de asientos para eventos.

Análisis de Resultados

Comportamiento por Nivel de Aislamiento

READ COMMITTED

El nivel READ COMMITTED demostró ser el más permisivo de los tres niveles de aislamiento evaluados, lo que resultó en:

- **Alta tasa de éxito**: Aproximadamente 80% de las reservas fueron exitosas.
- **Tiempos de respuesta menores**: Promedios entre 120ms (5 usuarios) y 500ms (30 usuarios).
- **Escalabilidad superior**: Mantuvo un rendimiento aceptable incluso con 30 usuarios concurrentes.

Sin embargo, este nivel podría permitir anomalías como lecturas no repetibles, donde una transacción lee un mismo registro dos veces y obtiene resultados diferentes debido a actualizaciones de otras transacciones. En nuestro contexto, esto podría manifestarse como una situación donde un asiento aparece disponible al inicio de la transacción, pero al momento de confirmar la reserva, otro usuario ya lo ha reservado.

REPEATABLE READ

El nivel REPEATABLE READ ofreció un equilibrio entre consistencia y rendimiento:

- **Tasa de éxito moderada**: Aproximadamente 60-70% de las reservas fueron exitosas.
- **Tiempos de ejecución intermedios**: Entre 135ms (5 usuarios) y 580ms (30 usuarios).
- **Protección contra lecturas no repetibles**: Garantizó que si un asiento aparecía como disponible al inicio de una transacción, seguiría siendo visto como disponible durante toda la transacción.

Este nivel proporciona mayor garantía de consistencia, evitando que un usuario vea un asiento como disponible para luego descubrir que no lo está. Sin embargo, sigue

siendo vulnerable a anomalías de escritura fantasma, donde nuevas filas que satisfacen una condición previa pueden aparecer durante una transacción.

SERIALIZABLE

El nivel SERIALIZABLE ofreció las mayores garantías de consistencia, pero con el mayor impacto en rendimiento:

- **Tasa de éxito menor**: Entre 50-60% de las reservas fueron exitosas.
- **Tiempos de ejecución más altos**: Desde 150ms (5 usuarios) hasta 650ms (30 usuarios).
- **Máxima protección contra anomalías**: Eliminó completamente problemas como lecturas sucias, lecturas no repetibles y escrituras fantasma.

Este nivel simula una ejecución completamente secuencial de las transacciones, lo que garantiza la consistencia total a costa de un mayor número de transacciones abortadas debido a conflictos serializables.

Escalabilidad y Rendimiento

La escalabilidad del sistema mostró patrones claros al aumentar el número de usuarios concurrentes:

1. **Degradación no lineal**: El tiempo promedio no aumentó linealmente con el número de usuarios, sino que mostró un crecimiento más acelerado, especialmente entre 20 y 30 usuarios.
2. **Punto de inflexión**: Con READ COMMITTED, el sistema mantuvo un buen rendimiento hasta aproximadamente 20 usuarios. Con SERIALIZABLE, el punto de inflexión fue mucho más temprano, alrededor de 10 usuarios.
3. **Tasa de fallos vs. concurrencia**: La tasa de fallos aumentó de manera más pronunciada en niveles de aislamiento más altos, indicando que la resolución de conflictos se vuelve más costosa con mayor concurrencia.

Reflexiones y Aprendizajes

Mayor Reto en la Implementación de Concurrencia

El mayor desafío encontrado fue el manejo adecuado de los errores y excepciones generados por los conflictos de concurrencia. En particular:

1. **Identificación de la causa raíz de los fallos**: Distinguir entre diferentes tipos de conflictos (deadlocks, timeouts, conflictos de serialización) requirió una comprensión profunda de los mensajes de error de PostgreSQL.

2. **Implementación de estrategias de reintentos**: Determinar cuándo y cómo reintentar una transacción fallida sin causar más contención fue un desafío significativo.
3. **Equilibrio entre rendimiento y consistencia**: Encontrar el nivel de aislamiento adecuado para cada caso de uso requirió evaluar cuidadosamente las compensaciones entre rendimiento y garantías de consistencia.

Problemas de Bloqueo Encontrados

Durante las pruebas, identificamos varios patrones de bloqueo:

1. **Deadlocks**: Ocurrieron cuando múltiples transacciones intentaban adquirir bloqueos sobre los mismos recursos en orden diferente. PostgreSQL detectó y resolvió estos deadlocks automáticamente, pero causó la cancelación de algunas transacciones.
2. **Contención de recursos**: El sistema experimentó alta contención en la tabla de asientos, especialmente en los registros de asientos más populares o aquellos que aparecían primero en las consultas.
3. **Bloqueos en cascada**: En algunos casos, una transacción larga podía causar un efecto dominó, bloqueando múltiples transacciones que dependían de los mismos recursos.

Nivel de Aislamiento Más Eficiente

Basándonos en nuestros resultados, concluimos que:

- **Para sistemas de alta concurrencia con requisitos moderados de consistencia**: READ COMMITTED ofrece el mejor equilibrio entre rendimiento y consistencia. Es adecuado para sistemas donde ocasionalmente se puede tolerar una pequeña inconsistencia temporal.
- **Para sistemas con requisitos estrictos de consistencia**: SERIALIZABLE es la mejor opción, aunque requiere una implementación cuidadosa de estrategias de reintento y posiblemente un diseño que minimice la contención de recursos.
- **Para la mayoría de los sistemas de reserva en producción**: REPEATABLE READ ofrece un buen compromiso, evitando la mayoría de las anomalías preocupantes mientras mantiene un rendimiento aceptable.

Ventajas y Desventajas del Lenguaje Seleccionado

Python como lenguaje para implementar el simulador presentó varias ventajas y desventajas:

Ventajas:

- Facilidad de implementación de hilos y concurrencia mediante la biblioteca threading y concurrent.futures
- Excelente soporte para PostgreSQL a través de psycopg2
- Sintaxis clara y legible que facilitó el desarrollo y depuración
- Amplio ecosistema de bibliotecas para análisis de datos y visualización de resultados

Desventajas:

- El GIL (Global Interpreter Lock) de Python limita la verdadera ejecución paralela en un solo proceso
- Rendimiento más limitado comparado con lenguajes como Java o Go para operaciones intensivas en CPU
- La gestión de hilos en Python no es tan robusta como en otros lenguajes diseñados específicamente para concurrencia