

1. Definición de Tablas y Relaciones

Tablas Principales:

- **Eventos:**

Contendrá la información básica del evento (nombre, fecha, lugar, etc.).

- **id_evento** (clave primaria)
- **nombre**
- **fecha**
- **lugar**

- **Asientos:**

Representa los asientos disponibles para cada evento.

- **id_asiento** (clave primaria)
- **id_evento** (clave foránea que referencia a Eventos)
- **numero_asiento**
- **seccion**

- **Reservas:**

Registra cada intento de reserva, vinculando el evento, el asiento y el usuario.

- **id_reserva** (clave primaria)
- **id_evento** (clave foránea)
- **id_asiento** (clave foránea)
- **usuario**
- **fecha_reserva**
- **Restricción única:** Para evitar la reserva doble de un mismo asiento, (id_evento, id_asiento).

Fase 2: Generación de Datos de Prueba

Data.sql

-- Verifica los datos de la tabla eventos:

```
SELECT * FROM eventos;
```

Query

Query History

1
SELECT * FROM eventos;

Data Output

Messages

Notifications

≡+

📄

▼

📋

▼





🗑️

🗄️

⬇️

📈

SQL

	id_evento [PK] integer 	nombre character varying (100) 	fecha date 	lugar character varying (100) 
1	1	Concierto de Rock	2025-06-15	Estadio Nacional
2	2	Teatro Clásico	2025-07-20	Teatro Central

-- Verifica los datos de la tabla asientos:

SELECT * FROM asientos;

Query

Query History

1

SELECT * FROM asientos;

Data Output

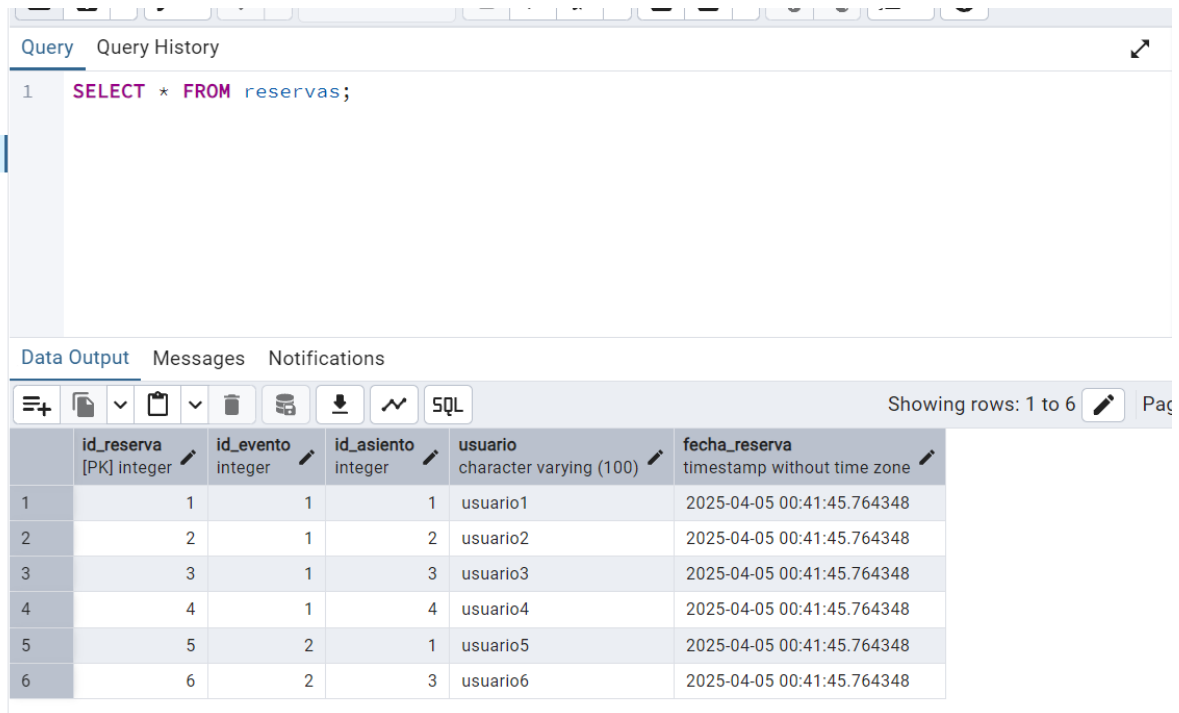
Messages

Notifications

<

-- Verifica los datos de la tabla reservas:

SELECT * FROM reservas;



Query Query History

```
1 SELECT * FROM reservas;
```

Data Output Messages Notifications

Showing rows: 1 to 6

	id_reserva [PK] integer	id_evento integer	id_asiento integer	usuario character varying (100)	fecha_reserva timestamp without time zone
1	1	1	1	usuario1	2025-04-05 00:41:45.764348
2	2	1	2	usuario2	2025-04-05 00:41:45.764348
3	3	1	3	usuario3	2025-04-05 00:41:45.764348
4	4	1	4	usuario4	2025-04-05 00:41:45.764348
5	5	2	1	usuario5	2025-04-05 00:41:45.764348
6	6	2	3	usuario6	2025-04-05 00:41:45.764348

Fase 3: Implementación del Programa de Simulación

Link de repositorio:

[MiltonPolanco/Proyecto-2-base-de-datos](#)

Manual de uso para ejecutar la simulación:

1. Se debe tener configurado PostgreSQL con la base de datos reservas_eventos, con las tablas creadas con ddl.sql.
2. Se abre el programa y se cambia la configuración del programa de acuerdo con la base de datos que se está usando.

```
def attempt_reservation(event_id, available_seats, user_name, isolation_level):  
    try:  
        conn = psycopg2.connect(  
            host="localhost",  
            database="reservas_evento",  
            user="postgres",  
            password="0512"  
        )  
        conn.set_client_encoding('UTF8')
```

3. Se instala **psycopg2** en la terminal con: "pip install psycopg2"
4. Para probar diferentes **niveles de aislamiento** se debe cambiar lo que está en negrita entre, **READ COMMITED**, **REPEATABLE READ** y **SERIALIZABLE** : python simulation_extra.py **SERIALIZABLE** 10.
5. Para probar diferentes **Usuarios Concurrentes** se deben cambiar **los últimos 2 números** del comando: python simulation_extra.py **SERIALIZABLE 10 10**. El **primer** número es el número de usuarios concurrentes y el **segundo** es la cantidad de sillas. Aunque si solo se pone el número de usuarios, la cantidad de sillas por defecto son **10**.
6. Al terminar cada prueba se puede ejecutar DELETE FROM reservas WHERE id_evento = 1; en PostgreSQL para dejar limpia la tabla de los asientos con id de evento 1, para poder hacer las pruebas de forma más limpia.

Fase 4: Experimentación y pruebas

Usuarios Concurrentes	Nivel de aislamiento	Reservas exitosas	Reservas fallidas	Tiempo promedio
5	READ COMMITED	4	1	0.08 SEGUNDOS
10	READ COMMITED	3	7	0.16 SEGUNDOS
20	READ COMMITED	2	18	0.29 SEGUNDOS
30	READ COMMITED	8	22	0.38 SEGUNDOS
5	REPEATABLE READ	4	1	0.08 SEGUNDOS
10	REPEATABLE READ	6	4	0.16 SEGUNDOS
20	REPEATABLE READ	11	9	0.27 SEGUNDOS
30	REPEATABLE READ	9	21	0.44 SEGUNDOS
5	SERIALIZABLE	4	1	0.08 SEGUNDOS
10	SERIALIZABLE	6	4	0.13 SEGUNDOS
20	SERIALIZABLE	10	10	0.26 SEGUNDOS
30	SERIALIZABLE	9	21	0.36 SEGUNDOS

Cuadro 1: Resultados Comparativos de Simulación de Concurrencia

Fase 5: Análisis y Reflexión**Conclusiones sobre el Manejo de Concurrency en Bases de Datos**

El manejo de la concurrencia es esencial para garantizar la integridad y consistencia de los datos cuando múltiples usuarios realizan operaciones simultáneas. La implementación correcta de transacciones, bloqueos y niveles de aislamiento permite prevenir problemas como las lecturas sucias, lecturas no repetibles y fenómenos fantasma. A través de la simulación se pudo observar cómo, en escenarios de alta contención, el uso de una restricción única (por ejemplo, para evitar la reserva duplicada de un asiento) y distintos niveles de aislamiento garantiza que solo una transacción se complete exitosamente, a pesar de los múltiples intentos concurrentes.

Cada nivel de aislamiento tiene su propio equilibrio entre consistencia y rendimiento, lo que influye en la aparición de bloqueos y en el tiempo de respuesta de las operaciones. La correcta selección y configuración del nivel de aislamiento es crucial para que la base de datos responda a las necesidades de la aplicación sin comprometer la integridad de los datos.

- **¿Cuál fue el mayor reto al implementar la concurrencia?**

El mayor reto fue asegurar que las transacciones se ejecutaran de manera consistente sin afectar el rendimiento. Esto incluyó:

- La correcta configuración y uso de niveles de aislamiento para prevenir problemas de lectura sucia o no repetible.
- La gestión de bloqueos en situaciones de alta concurrencia, donde múltiples hilos intentan acceder y modificar el mismo recurso.
- La sincronización de hilos y el manejo de reintentos o abortos de transacción, especialmente en escenarios con el nivel `SERIALIZABLE`.

- **¿Qué problemas de bloqueo encontraron?**

Se observaron algunos bloqueos que afectaron el rendimiento y causaron abortos en transacciones:

- **Bloqueos en nivel SERIALIZABLE:**
Este nivel, al ser el más estricto, generó abortos en transacciones cuando se detectó alta contención, obligando a reintentar la operación.
- **Bloqueos menores en niveles inferiores:**
Aunque en READ_COMMITTED y REPEATABLE_READ se presentaron menos abortos, en transacciones más complejas podían surgir bloqueos que retrasaban la ejecución o provocaban inconsistencias en la lectura de datos.

- **¿Cuál fue el nivel de aislamiento más eficiente?**

En el escenario de la simulación (donde la operación es simple, es decir, reservar un asiento con restricción única):

- **READ_COMMITTED** mostró un rendimiento ligeramente superior, ya que implica menor sobrecarga en la gestión de bloqueos y transacciones.
- **REPEATABLE_READ** y **SERIALIZABLE** proporcionan mayor consistencia, pero a costa de una mayor latencia y, en el caso de SERIALIZABLE, posibles abortos y reintentos.

Por lo tanto, para operaciones simples, READ_COMMITTED es generalmente suficiente; sin embargo, en sistemas donde la consistencia es crítica, podría justificarse el uso de niveles más estrictos a pesar del impacto en el rendimiento.

- **¿Qué ventajas y desventajas tuvo el lenguaje seleccionado?**

Se utilizó **Python** en la simulación, lo que trajo varias ventajas y algunas desventajas:

Ventajas:

- **Facilidad de desarrollo:**
Python es un lenguaje de alto nivel, con sintaxis clara y fácil de aprender, lo que facilita la implementación y mantenimiento del código.
- **Bibliotecas robustas:**
Librerías como *psycopg2* permiten la conexión y manipulación de bases de datos PostgreSQL de forma eficiente.
- **Soporte para concurrencia:**
Aunque Python tiene limitaciones en la ejecución paralela debido al

Global Interpreter Lock (GIL), en este caso la tarea se centra en operaciones de I/O (acceso a la base de datos), donde el impacto es menor.

Desventajas:

- **Limitaciones del GIL:**
En escenarios de alta carga de CPU, el GIL puede limitar la verdadera ejecución paralela, lo cual puede ser un factor si la aplicación requiere procesamiento intensivo.
- **Rendimiento comparado con lenguajes compilados:**
Python, al ser interpretado, puede ser más lento en comparación con lenguajes como Java o C#, especialmente en tareas de alta concurrencia y procesamiento intensivo.