EDUCACIÓN
SECRETARÍA DE EDUCACIÓN PÚBLICA

TECNOLÓGICO NACIONAL DE MÉXICO

Instituto Tecnológico de Morelia

Reporte 12
Ejercicios conceptuales de concurrencia

Profesor: Bryan Eduardo Martinez Guzmán

Taller de Base de Datos

Aguilar Calderón Katia
Paz Alfaro Mariana
Rodriguez Tapia J Karlo

Morelia, Michoacán

Índice

Introducción

Los siguientes ejercicios realizados en clase son para detectar las anomalías y comprender mejor algunos errores muy específicos que pueden ocurrir dado a la naturaleza de la base de datos de ofrecer información de manera simultánea a muchos usuarios, se revisa cómo los datos dependen entre sí y lo que ocurre en diferentes niveles de acceso.

Desarrollo

Ejercicio 11

Identificar anomalía
T1: R(A), W(A); T2: R(A), W(A). Si T2 escribe sobre A después de que T1 leyó A pero antes de que T1 escriba, ¿qué anomalía ocurre?

Es actualización perdida, porque T1 escribirá sobre el valor que puso T2 en A

Ejercicio 12

Planificación conflictiva
Dada la planificación: R1(A), W1(A), R2(A), W2(A), R1(B), W1(B). Identifica los conflictos entre operaciones.
En B no hay ningún problema posible

En A no hay ningún problema si se ejecutan así pero tiene problema en el orden ya que si T2 ejecuta R(A) antes que T1 haga W(A) se produce un problema y T2 leería otro valor que no es (lectura sucia)
Si T2 ejecuta sus operaciones antes que T1 o viceversa se produce una actualización perdida, pues los datos escritos por el que hizo las operaciones primero se pierden.

Ejercicio 13

Lectura no repetible
Explica con un ejemplo cómo ocurre la lectura no repetible entre dos transacciones.

Ocurre cuando un usuario lee primero un valor sin terminar su consulta, luego otro usuario lo modifica y el primero vuelve a leer pero el mismo valor es diferente aunque sea la misma consulta.
Ejemplo:
U1: Lee el saldo con el id=12 // El saldo=4000 pero U1 aún no lo ve aunque ya se leyó
U2: Cambia el saldo de id=12 a 3500
U1: Vuelve a leer el saldo de id=12 en la misma consulta pero ahora ve el resultado
// Una lectura del saldo es 4000 y la otra 3500

Ejercicio 14

Estados transaccionales
Describe el flujo completo de estados por los que pasa una transacción exitosa o una que falla.

Activa: La transacción ha comenzado y está en progreso de realizarse
Parcialmente comprometida: Ha hecho la última operación que se requería pero no ha hecho commit

Fallida:
Fallida: Por algún motivo el sistema decide que la operación no puede realizarse
Abortada: Se ha suspendido la ejecución de la consulta

Exitosa:
Comprometida: Se ha hecho commit y todo ha salido muy bien.

Ejercicio 15

Análisis inconsistente
En un sistema bancario, T1 suma todos los saldos mientras T2 transfiere dinero entre cuentas. ¿Qué anomalía puede ocurrir?

Lectura sucia: Suponiendo que T1 solo suma los saldos para leerlos entonces T2 cambiará el valor real a penas T1 lo lea.

Ejercicio 16

Planificación serial
Dadas T1: R(X), W(X) y T2: R(X), W(X). ¿Es la planificación R1(X), R2(X), W1(X), W2(X) serial? Justifica.
No sería serial, ya que para ser serial primero se debe finalizar la transacción T1 para que pueda empezar T2, y al estar intercaladas las operaciones esto no es posible

Ejercicio 17

Operaciones mixtas
Para T1: R(A), W(A), R(B), W(B) y T2: R(A), R(B). Escribe una planificación donde se preserve el orden interno.
La planificación seria R1(A), R2(A), W1(A), R1(B), R2(B), W1(B) ya que según el orden establecido tanto en T1 como en T2, primero se lee y se escribe A y después B, por lo que podemos intercalarlos bajo este orden lo que nos da como resultado esta planificación

Ejercicio 18

Detección anomalías
En la planificación: W1(A), R2(A), W2(A), Commit2, Rollback1. ¿Qué anomalía específica ocurre?
Realiza lectura sucia, ya que primero T1 escribe una variable A, despues en T2 se lee esta variable A pero sin guardarla, para después volver a escribir una nueva variable A, pero esta vez si hace un Commit para confirmar los cambios para que al final con Rollback elimine los datos que se ingresaron en T1

Ejercicio 19

Transacción compuesta
Una transacción de "crear pedido" debe: verificar stock, reducir inventario, crear pedido, crear factura. ?¿Qué pasa si falla después de crear pedido pero antes de la factura?
Pueden pasar dos errores, el primero sería que el inventario se reduciría pero no hay alguna prueba (factura) que diga cuando fue vendido o porque falta y el segundo sería que se crea el pedido pero no tiene una factura, lo que haría que la transacción romperá con su toxicidad

Ejercicio 20

Concurrencia práctica
¿Por qué en sistemas reales no siempre se usa el nivel máximo de aislamiento?

Ejercicio 21

Planificación no serializable
Dada T1: R(A), W(A), R(B), W(B) y T2: R(A), W(A), R(B), W(B). Demuestra que la planificación R1(A), W1(A), R2(A), W2(A), R1(B), W1(B), R2(B), W2(B) no es serializable

En el dato A, T1 escribe antes de que T2 lea y escriba, por lo que T1 depende antes que T2. En el dato B, pasa lo mismo: T1 escribe antes de que T2 lea y escriba. Por tanto, T1 debe ir antes que T2 en todos los casos. No hay ninguna parte donde T2 tenga que ir antes que T1. El orden final es T1 → T2, y no hay contradicciones. Conclusión: La planificación sí es serializable, porque se comporta igual que si primero se ejecutara T1 completa y después T2.

Ejercicio 22

Análisis de dependencias
Para la planificación: R1(A), R2(B), W1(A), W2(B), R1(B), R2(A), W1(B), W2(A). Identifica todas las dependencias de conflicto y determina si es serializable.

En el dato A, T1 escribe antes de que T2 lea y escriba, por lo tanto T1 depende antes que T2 (T1 → T2). En el dato B, T2 escribe antes de que T1 lea y escriba, por lo tanto T2 depende antes que T1 (T2 → T1). Esto crea un ciclo: T1 → T2 → T1. Cuando hay un ciclo, significa que no se puede encontrar un orden donde una transacción vaya totalmente antes que la otra sin romper la lógica. Osea que la planificación no es serializable, porque hay dependencias en ambas direcciones que generan un conflicto.

Ejercicio 23

Lectura fantasma compleja
En un sistema de reservas, T1 busca asientos disponibles > 0, T2 reserva el 煤ltimo asiento. Explica cómo ocurre la lectura fantasma y cómo prevenirla.

En un sistema de reservas, la transacción T1 busca los asientos disponibles (por ejemplo, “asientos con disponibilidad > 0”). Mientras T1 aún está viendo los datos, otra transacción T2 reserva el último asiento disponible y hace commit. Cuando T1 intenta reservar, cree que todavía hay asientos porque su lectura fue antes del cambio, pero en realidad ya no hay ninguno. Esto se llama lectura fantasma, porque el resultado de la consulta cambió mientras la transacción seguía activa. Para evitarlo se usa el nivel de aislamiento serializable, que bloquea los rangos de datos y evita que otra transacción modifique el conjunto de resultados. También se puede usar SELECT ... FOR UPDATE para bloquear los asientos mientras se realiza la lectura. En sistemas modernos, se usa MVCC (control multiversión). Osea que la lectura fantasma ocurre cuando el conjunto de datos que ve una transacción cambia por otra que inserta o modifica filas. Para evitarlo, se usan bloqueos de rango o el nivel de aislamiento serializable.

Ejercicio 24

Recuperación ante fallos
T1: W(A), W(B); T2: W(C), W(D). El sistema falla después de Commit1 pero antes de Commit2. Explica el proceso de recuperación.

El sistema falla después del Commit1 pero antes del Commit2. Al recuperarse, el gestor de transacciones usa el registro de log para: Rehacer (redo) las operaciones de T1 (W(A) y W(B)), porque ya se había confirmado y sus efectos deben mantenerse. Deshacer (undo) las operaciones de T2 (W(C) y W(D)), ya que no llegó a hacer commit y sus cambios no deben conservarse. Así se cumple la durabilidad de T1 y la atomicidad de T2.

Ejercicio 25

Equivalencia de vistas
Dadas T1: R(A), W(B); T2: W(A), R(B); T3: R(A), R(B). Determina si dos planificaciones diferentes son equivalentes en vista.

Dos planificaciones son equivalentes en vista si: Cada lectura obtiene el valor del mismo write en ambas planificaciones y si el último write sobre cada dato es el mismo. Las lecturas iniciales (si no hay write previo) se hacen del mismo valor inicial.
Por lo tanto, si alguna lectura cambia de origen o el último write es distinto, las planificaciones no son equivalentes en vista.

Ejercicio 26

Planificación con abortos

Analiza: W1(A), R2(A), W2(B), Abort1, Commit2. ?Qué problemas de consistencia surgen y cómo debería manejarlos el sistema?

En W1(A), R2(A), W2(B), Abort1, Commit2, T2 lee un valor que T1 escribió antes de hacer commit. Cuando T1 aborta, el valor que leyó T2 ya no es válido, provocando una lectura sucia.
El sistema debe evitarlo usando bloqueo estricto (Strict 2PL) o MVCC, impidiendo que una transacción lea datos no confirmados. Si ya ocurrió, T2 debe abortarse también (cascading abort) para mantener la consistencia.

Ejercicio 27

Transacciones anidadas
En un sistema con transacciones anidadas, explica cómo se maneja el rollback de una subtransacción sin afectar la transacción padre.

Si una subtransacción falla, se deshacen solo sus propias operaciones sin afectar a la transacción padre.
Sus cambios se mantienen separados (por ejemplo, en un área temporal o mediante savepoints) hasta que el padre haga commit.
De esta forma, el rollback de una subtransacción no cancela todo el trabajo del padre, que puede continuar normalmente.

Ejercicio 28

Optimización concurrencia
Para un sistema de alto tráfico con 90% lecturas y 10% escrituras, ?qué estrategia de control de concurrencia recomendarías y por qué?

Con una carga del 90% de lecturas y 10% de escrituras, la mejor estrategia es usar MVCC (Multi-Version Concurrency Control) o Snapshot Isolation.
Esto permite que las lecturas no bloqueen las escrituras y viceversa, mejorando el rendimiento. Además, se pueden usar réplicas de solo lectura para distribuir la carga sin afectar la consistencia global.

Ejercicio 29

Deadlock detection
T1 tiene lock A y solicita B; T2 tiene lock B y solicita A. Describe el algoritmo que detectaría este deadlock.

El sistema construye un grafo de espera (Wait-For Graph) donde cada transacción que espera por un recurso bloqueado por otra crea una arista. En este caso, T1 espera por B (bloqueado por T2) y T2 espera por A (bloqueado por T1), formando un ciclo. El algoritmo

detecta el ciclo y el sistema elige una transacción víctima para abortar (normalmente la más joven o la que menos trabajo haya hecho), liberando los recursos y resolviendo el deadlock.

Ejercicio 30

Consistencia eventual vs fuerte
En un sistema distribuido, contrasta los enfoques de consistencia fuerte (ACID) vs consistencia eventual (BASE) para transacciones financieras.

La consistencia fuerte (ACID) asegura que todas las transacciones sean atómicas, consistentes, aisladas y duraderas. Es ideal para sistemas financieros donde no se permiten datos desactualizados. La consistencia eventual (BASE) prioriza la disponibilidad y el rendimiento: las actualizaciones pueden propagarse con retraso y diferentes nodos pueden tener datos distintos temporalmente. Por eso, en sistemas financieros se prefiere la consistencia fuerte, mientras que la eventual se usa en sistemas distribuidos donde la velocidad es más importante que la precisión inmediata.

Conclusión

Con esta práctica se comprendió la importancia del control de concurrencia en las bases de datos. Se pudo ver que cuando varias transacciones se ejecutan al mismo tiempo, pueden ocurrir errores si no se controlan correctamente, como lecturas incorrectas o pérdida de información. También se aprendió que existen diferentes métodos para evitar estos problemas, como los bloqueos, los niveles de aislamiento y los mecanismos de recuperación ante fallos. En general, esta práctica ayudó a entender cómo mantener la consistencia y seguridad de los datos en sistemas donde muchas operaciones se realizan al mismo tiempo.