Concurrencia y Seguridad de Bases de Datos en Tiempo Real

Agustín Gurvich

FCEIA, UNR

19/04/2022

Control de concurrencia

En las bases de datos convencionales, la noción de correctitud está asociada a las propiedades ACID.

Control de concurrencia

En las bases de datos convencionales, la noción de correctitud está asociada a las propiedades ACID.

El control de concurrencia se encarga de asegurar que las transacciones concurrentes sean serializables.

Control de concurrencia

En las bases de datos convencionales, la noción de correctitud está asociada a las propiedades ACID.

El control de concurrencia se encarga de asegurar que las transacciones concurrentes sean serializables.

Generamos nuevas estrategias para mantener la consistencia sin necesidad de acatar las propiedades ACID

Problemas de concurrencia

Los problemas de concurrencia más importantes son:

- Deadlocks
- Inversión de Prioridades

Proponemos nuevas estrategias para evitar estas situaciones

Inversión de Prioridades

Supongamos dos transacciones T_i, T_j con prioridades k, k' tales que $k < k'$.

Supongamos ahora que T_i tiene un recurso que T_j necesita para poder ejecutar (un lock de write por ejemplo).

Inversión de Prioridades

Supongamos dos transacciones T_i, T_j con prioridades k, k' tales que $k < k'$.

Supongamos ahora que T_i tiene un recurso que T_j necesita para poder ejecutar (un lock de write por ejemplo).

En este caso, T_j tiene mayor prioridad que T_i, pero no puede ejecutarse! T_i está acaparando el recurso, y (quizás) no podrá ejecutarse hasta que no termine T_j.

Priority Inheritance

Para solucionar el problema de inversión de prioridades se propuso la estrategia de *heredamiento de prioridades*.

En este caso, T_i tendrá momentáneamente una prioridad mayor a la mayor de las prioridades de los procesos activos.

Evitando deadlocks

La mejor forma de evitar deadlocks es generar estrategias que los eviten. El problema es que no siempre es posible.

Un deadlock se puede pensar como un grafo circular, y la solución es cortar un nodo para resolver el ciclo. Para detectarlos se utilizan algoritmos de detección de ciclos.

Resolviendo deadlocks

¿Cómo podemos resolverlos?

1. Abortando la transacción que genera el deadlock
2. Encontrar el ciclo y abortar la primer transacción que falló su deadline. Si ninguna cumple esa característica, abortar la que tenga la deadline más lejana
3. Encontrar el ciclo y abortar la primer transacción que falló su deadline. Si ninguna cumple esa característica, abortar la que tenga la deadline más cercana
4. Encontrar el ciclo y abortar la primer transacción que falló su deadline. Si ninguna cumple esa característica, abortar la que tenga menor criticidad
5. Abortar la transacción no feasible con menor criticidad. Si todas son feasibles, abortar la que tenga menor criticidad

Control de Concurrencia

Para mantener la consistencia de los datos cuando se están ejecutando múltiples transacciones es necesario introducir alguna forma de control de concurrencia.

Control de Concurrencia

Para mantener la consistencia de los datos cuando se están ejecutando múltiples transacciones es necesario introducir alguna forma de control de concurrencia.

En general, podemos hablar de dos estrategias importantes:

- Pesimistas: Antes de realizar operaciones sobre la información, debemos pedir permiso
- Optimistas: No hay que pedir permisos. Se ejecuta toda la transacción y luego se valida al commitear para ver si se mantuvo la serialización

Control de Concurrencia

Vamos a disctutir las siguientes técnicas de control de concurrencia:

- Locking
- Optimistas
- Especulativas
- Multiversión
- Ajuste Dinámico de Serialización

Locking

Las estrategias de locking son las más utilizadas en las bases de datos convencionales.

Se bloquea el acceso a un recurso para que no más de un actor pueda utilizarlo.

Locking

Las estrategias de locking son las más utilizadas en las bases de datos convencionales.

Se bloquea el acceso a un recurso para que no más de un actor pueda utilizarlo. El uso indiscriminado de locks no asegura consistencia.

Two-Phase-Locking (2PL)

El protocolo pesimista 2PL divide una transacción en dos fases:

1. Fase de crecimiento: Adquiere todos los locks necesarios, pero no puede soltar ninguno
2. Fase de decreceimiento: Empieza a liberar locks, pero no puede adquirir ninguno nuevo

Two-Phase-Locking (2PL)

El protocolo pesimista 2PL divide una transacción en dos fases:

1. Fase de crecimiento: Adquiere todos los locks necesarios, pero no puede soltar ninguno
2. Fase de decreceimiento: Empieza a liberar locks, pero no puede adquirir ninguno nuevo

Si bien en las bases de datos convencionales es muy útil, las bases de datos en tiempo real no pueden darse el lujo de perder tiempo en estas fases.

2PL Wait Promote (2PL-WP)

Es un protocolo idéntico a 2PL, pero tiene un mecanismo de heredamiento de prioridades.

Cuando una transacción bloquea a otra de mayor prioridad por acaparar un recurso, la primera adquiere la prioridad de la segunda.

Reduce los tiempos de bloqueo por el uso de locks, y mantiene las ventajas del uso de los recursos del esquema 2PL original.

2PL Wait Promote (2PL-WP)

Es un protocolo idéntico a 2PL, pero tiene un mecanismo de heredamiento de prioridades.

Cuando una transacción bloquea a otra de mayor prioridad por acaparar un recurso, la primera adquiere la prioridad de la segunda.

Reduce los tiempos de bloqueo por el uso de locks, y mantiene las ventajas del uso de los recursos del esquema 2PL original.
El problema es que los tiempos de bloqueo siguen siendo inciertos. Aún más, puede degradar el sistema a una base de datos convencional.

2PL High-Priority (2PL-HP)

Modificación del protocolo 2PL que agrega la prioridad de la transacción para resolver conflictos. Siempre se resuelven en favor de la transacción de mayor prioridad.

Cuando se pide un lock de un objeto que tiene una transacción de menor prioridad, esta se reinicia o aborta. Si el objeto estaba siendo usado por una de mayor prioridad, simplemente se espera.

2PL High-Priority (2PL-HP)

Modificación del protocolo 2PL que agrega la prioridad de la transacción para resolver conflictos. Siempre se resuelven en favor de la transacción de mayor prioridad.

Cuando se pide un lock de un objeto que tiene una transacción de menor prioridad, esta se reinicia o aborta. Si el objeto estaba siendo usado por una de mayor prioridad, simplemente se espera.
2PL-HP no tiene inversión de prioridades ni deadlock, pero puede abortar transacciones en favor de otras que no podrían completar su deadline.

Conditional Priority-Inheritance (CPI)

Si se detecta inversión de prioridades, pueden suceder dos cosas:

- Si la transacción de menor nivel está *a punto de ser completada*, entonces hereda la prioridad de la transacción de mayor nivel
- Caso contrario, la transacción es abortada

Conditional Priority-Inheritance (CPI)

Si se detecta inversión de prioridades, pueden suceder dos cosas:

- Si la transacción de menor nivel está *a punto de ser completada*, entonces hereda la prioridad de la transacción de mayor nivel
- Caso contrario, la transacción es abortada

Lo complicado de este esquema es definir qué significa para una transacción estar *a punto de ser completada*

Control de concurrencia optimista (OCC)

Los chequeos no se realizan mientras se ejecuta la transacción, sino al final.

La idea es poder trabajar sin trabas, accediendo libremente a los recursos, y realizar una sola verificación grande al terminar el trabajo.

Control de concurrencia optimista (OCC)

Los chequeos no se realizan mientras se ejecuta la transacción, sino al final.

La idea es poder trabajar sin trabas, accediendo libremente a los recursos, y realizar una sola verificación grande al terminar el trabajo.

Al terminar de ejecutarse una transacción, se chequea que esta no entre en conflicto con otras que hayan commiteado mientras se ejecutaba.

Control de concurrencia optimista (OCC)

Los chequeos no se realizan mientras se ejecuta la transacción, sino al final.

La idea es poder trabajar sin trabas, accediendo libremente a los recursos, y realizar una sola verificación grande al terminar el trabajo.

Al terminar de ejecutarse una transacción, se chequea que esta no entre en conflicto con otras que hayan commiteado mientras se ejecutaba.

Como todas las escrituras se realizan al commitear, es equivalente a un write atómico.

OCC-Broadcast Commit (OCC-BC)

Utiliza el esquema clásico de OCC, pero agrega una función de *broadcast* para adaptarse a los sistemas de tiempo real.

Al commitear, si existen transacciones activas que puedan tener conflictos con la actual, le notifica a dichas transacciones para que se reinicien.

OCC-Broadcast Commit (OCC-BC)

Utiliza el esquema clásico de OCC, pero agrega una función de *broadcast* para adaptarse a los sistemas de tiempo real.

Al commitear, si existen transacciones activas que puedan tener conflictos con la actual, le notifica a dichas transacciones para que se reinicien.

No hace falta chequear con las transacciones ya commiteadas, pues si hubiera conflictos ya hubiera sido reiniciada bajo este esquema. Validar una transacción es equivalente a commitearla.

OCC-Sacrifice

Una extensión del sistema OCC-BC que agrega el concepto de prioridades, el OCC-Sacrifice chequea las prioridades de las transacciones a abortar.

Si hay al menos una cuya prioridad es mayor a la que está a punto de commitear, se evita el commit y se reinicia la transacción.

OCC-Sacrifice

Una extensión del sistema OCC-BC que agrega el concepto de prioridades, el OCC-Sacrifice chequea las prioridades de las transacciones a abortar.

Si hay al menos una cuya prioridad es mayor a la que está a punto de commitear, se evita el commit y se reinicia la transacción.

Si bien añade conceptos de prioridades al esquema, OCC-Sacrifice puede cancelar transacciones terminadas en favor de otras que quizás no puedan alcanzar su deadline.

OCC-Wait

Otra extensión del OCC-BC, pero esta vez se añade una cola de espera para las transacciones que estén en estado conflictivo.

La transacción se mantiene en cola esperando que las otras de mayor prioridad realicen su commit. Si no lo hacen, entonces se reinicia la transacción que estaba esperando. Caso contrario, se commitea y se abortan las conflictivas.

OCC-Wait

Otra extensión del OCC-BC, pero esta vez se añade una cola de espera para las transacciones que estén en estado conflictivo.

La transacción se mantiene en cola esperando que las otras de mayor prioridad realicen su commit. Si no lo hacen, entonces se reinicia la transacción que estaba esperando. Caso contrario, se commitea y se abortan las conflictivas.

Efectivamente le estamos dando a las transacciones de mayor prioridad una oportunidad para que se ejecuten plenamente, pero aumentamos el número de operaciones pendientes de la base de datos.

Wait 50

En esta extensión del OCC-Wait, se define un *conjunto conflictivo* como un conjunto de transacciones que tienen conflictos con la que se quiere validar.

Si al menos el 50% de dichas transacciones tiene mayor prioridad que la validante, entonces se reinicia. Caso contrario, se commitea.

Wait

Podemos ver que Wait 50, OCC-Wait y OCC-BC son muy similares!

Podemos considerar un nuevo esquema *Wait k*, donde *k* es el porcentaje de transacciones conflictivas que superan la prioridad de la validante. En este caso tenemos:

- OCC-Wait : Wait 100, pues espera que todas las transacciones conflictivas terminen
- Wait 50, que espera que la mitad de transacciones conflictivas terminen
- OCC-BC : Wait 0, pues no espera a ninguna transacción conflictiva

OCC contra Locking

Dos investigadores trabajaron sobre las ventajas de cada estilo de conctrol de concurrencia:

- Haritsa descubrió que OCC era mejor que 2PL
- Huang descubrió que 2PL era mejor que OCC

¿A qué se debe esta contradicción?

OCC contra Locking

Se debe a las hipótesis del estudio:

- Haritsa había comparado los controles de concurrencia en una base de datos pequeña
- Huang había comparado los controles de concurrencia en una base de datos grande

OCC contra Locking

Se debe a las hipótesis del estudio:

- Haritsa había comparado los controles de concurrencia en una base de datos pequeña
- Huang había comparado los controles de concurrencia en una base de datos grande

Existe un punto de cruce! En 1992, Harista y sus compañeros publican "Data Access Scheduling in Firm Real-Time Database Systems", un paper donde encuentran discrepancias en los resultados y concluyen que se debía al tamaño de las bases de datos de prueba.

El problema fueron las suposiciones que hicieron!

Control de Concurrencia Especulativo (SCC)

Bestavros plantea este nuevo estilo de control de concurrencia, que intenta combinar los estilos pesimistas y optimistas en uno solo, utilizando el poder de las computaciones redundantes.

Control de Concurrencia Especulativo (SCC)

Bestavros plantea este nuevo estilo de control de concurrencia, que intenta combinar los estilos pesimistas y optimistas en uno solo, utilizando el poder de las computaciones redundantes.

Cuando se detecta un problema de concurrencia, se inicia una nueva transacción llamada *sombra* (shadow).

Sombras

Mientras que la transacción original se ejecuta con un modelo optimista, la sombra se ejecuta bajo un modelo pesimista.

Cuando una transacción finaliza y fuerza a que otra transacción T aborte, T no debería volver a comenzar desde el inicio. En este caso, la sombra de T se promovería a la versión principal.

De esta forma, cancelar una transacción no significa perder todo el progreso realizado.

Protocolos SCC

Bestavros planteó varias versiones de protocolos SCC, siempre motivado por la complejidad de la generación de sombras:

- SCC-Ordered Based: $\mathcal{O}(n!)$ sombras
- SCC-Conflict Based: $\mathcal{O}(n^2)$ sombras
- k-Shadow SCC: Permite a lo sumo k sombras por cada transacción
- SCC-Two Shadows: Una versión principal y una sombra de repuesto

Protocolos SCC

Bestavros planteó varias versiones de protocolos SCC, siempre motivado por la complejidad de la generación de sombras:

- SCC-Ordered Based: $\mathcal{O}(n!)$ sombras
- SCC-Conflict Based: $\mathcal{O}(n^2)$ sombras
- k-Shadow SCC: Permite a lo sumo k sombras por cada transacción
- SCC-Two Shadows: Una versión principal y una sombra de repuesto

Hong desarrolló en la universidad de Florida un modelo llamado *Alternative Version Concurrency Control*, que es básicamente similar a SCC.

Control de Concurrencia Multiversion

Cada actualización sobre los valores crean una nueva versión, y se guarda la vieja.

Este esquema no es muy factible, pues introduce tres problemas importantes:

1. Un gran requerimiento de espacio de almacenamiento
2. La necesidad de especificar sobre qué versión de los datos se quiere trabajar
3. Es muy difícil comprobar la serialización de los datos

Control de Concurrencia Basado en Similaridad

Pensemos en el caso de un sistema que toma constantemente medidas de temperatura en un campo. Es razonable pensar que el cambio entre dos muestras consecutivas no es muy exagerado.

En realidad, esta es una práctica muy común en algunos sistemas: asumir que un conjunto de muestras tomado en intervalos de tiempo cercanos son idénticas.

Control de Concurrencia Basado en Similaridad

Los estudios sobre el *Control de Concurrencia Basado en Similaridad* intenta formalizar este concepto: es posible obviar las muestras más nuevas y utilizar medidas viejas, siempre y cuando se establezca un período razonable (y justificable) de cuándo es despreciable la diferencia.

Procesamiento de Transacciones Seguro

Para áreas con información sensible (bancos, la bolsa, criptomonedas, etc.) se está estudiando el concepto de **Secure Real-Time Database Systems (SRTDBS)**.

La idea es poder tener las ventajas de las Bases de Datos en Tiempo Real, y agregarles capas de seguridad para poder mantener la información inaccesible a usuarios no autorizados.

Modelo Bell-LaPadula

El modelo Bell-LaPadula se usa para implementar control de acceso a los *objetos* de la base de datos según las *acciones*. Se basa en dos principios:

1. Una accion puede leer un objeto si su autorización es mayor o igual que la clasificación del objeto
2. Una accion puede escribir en un objeto si su autorización es menor o igual que la del objeto

Vectores para control de concurrencia

Bajo este principio, debemos asignar tanto prioridades como autorizaciones a las transacciones (accciones).

Se plantea un modelo donde a cada transacción T se le asigna un vector $P = (level, prio)$, donde *level* es el nivel de autorización y *prio* es el nivel de prioridad de dicha transacción.

Así, si dos transacciones tienen el mismo nivel de autorización, desempatan por prioridad.

Vectores para control de concurrencia

Bajo este principio, debemos asignar tanto prioridades como autorizaciones a las transacciones (accciones).

Se plantea un modelo donde a cada transacción T se le asigna un vector $P = (level, prio)$, donde *level* es el nivel de autorización y *prio* es el nivel de prioridad de dicha transacción.

Así, si dos transacciones tienen el mismo nivel de autorización, desempatan por prioridad. Lograr *fairness* es algo inalcanzable, y debe mitigarse utilizando una estrategia de control de concurrencia adecuada.

Referencias

Aldarmi, Saud A. (1998). Real-Time Database Systems: Concepts and Design.

Kuo, Tei-Wei, Lam, Kam-Yiu. (2002). Real-Time Database Systems Architecture and Techniques

J. R. Haritsa, M. J. Carey, M. Livny (1992). Data Access Scheduling in Firm Real-Time Database Systems

A. Bestavros (1993). Speculative Concurrency Control