FCEyN UBA

Bases de Datos

Control de Concurrencia
Multiversión

Índice

1. Introducción

Este apunte está tomado del libro Transactional Information Systems de Gerhard Weikum y Gottfried Vossen (ver referencias al final).

Muchas veces se usan en forma intercambiable los términos historia y *schedule*. Para los propósitos de este apunte tendrán una diferencia. Diferenciaremos los *schedules*, en los cuales se conoce el resultado de cada transacción de los *schedules* que están abiertos para alguna (o todas) las transacciones. A los *schedules* del primer tipo los llamaremos historias y conservamos el término *schedule* para los del segundo tipo. Resumiendo: una historia siempre está completa y aparecen todas las operaciones de cada transacción incluida la terminación (commit o abort). Las historias son vistas como *schedules* completos. En este sentido un *schedule* es un prefijo de una historia. Esta diferencia nos permite analizar la dinámica de la ejecución.

Cuando decimos que una operación p en una historia (o en un *schedule*) ocurre antes que otra operación q en la misma historia (o *schedule*) lo escribimos $p < q$, pero si se necesita expresar el contexto de un *schedule* s o una transacción t lo escribimos $p <_s q$ o $p <_t q$

Vamos a mantener el mismo modelo de transacciones que teníamos en los modelos anteriores sólo con un pequeño cambio de interpretación.

Básicamente, una operación de lectura de la forma $r(x)$ lee una versión (existente) de x, y una operación de escritura de la forma $w(x)$ (siempre) crea una nueva versión de x (o sobrescribe una existente). Asumimos que cada transacción escribe cada elemento de datos como máximo una vez; por lo tanto, si t_j contiene la operación $w_j(x)$, podemos denotar la versión de x creada por esta escritura como x_j. Ahora podemos definir

Definición

Sea s una historia con la transacción de inicialización t_0 y la transacción final t_∞. Una **función de versión** para s es una función h, que asocia con cada operación de lectura de s una operación de escritura anterior en el mismo elemento de datos, y que es la identidad en las operaciones de escritura.

Podemos entonces decir que:

1. $h(r_i(x)) = w_j(x)$ para algún $w_j(x) <_s r_i(x)$, y $r_i(x)$ lee x_j ,
2. $h(w_i(x)) = w_i(x)$, y $w_i(x)$ escribe x_i .

Por lo tanto, $h(r_i(x))$ denota esencialmente la versión de x que se ha asignado para ser leída por $r_i(x)$. En un ligero abuso de la notación, también escribimos $w_j(x_j)$

para la operación de escritura que crea x_j, y escribimos $r_i(x_j)$ para la operación lectura que se ha asociado con $w_j(x_j)$ por la función de versión h.

Una función de versión traduce cada operación de escritura en una operacón de creación de versión y cada operación de lectura en una operación de lectura de versión. Ahora estamos listos para introducir la noción formal de un *schedule* de múltiples versiones, que es esencialmente un *schedule* junto con una función de versión:

Sea $T = t_1, ..., t_n$ un conjunto finito de transacciones

1. Una historia multiversión (o un schedule multiversión completo) para T es un par $m = (op(m), <_m)$, donde $<_m$ es un orden sobre $op(m)$ y

   a) $op(m) = h(U_{i=1}^{n} op(t_i))$ para alguna función de versión h (aquí asumimos que h se ha extendido canónicamente de operaciones simples a conjuntos de operaciones)

   b) $\forall t \in T$ y para todas las operaciones $p, q \in op(t)$ se sostiene que:

   $$p <_t q \implies h(p) <_m h(q)$$

   c) Si $h(r_j(x)) = r_j(x_i), i \neq j$, y c_j está en m, entonces c_i está en m y $c_i <_m c_j$

2. Un **schedule multiversión** es un prefijo de una **historia multiversión**.

En palabras, un historial de multi versionado contiene operaciones de lectura y escritura versionadas para sus transacciones cuyo ordenamiento respeta los ordenamientos de transacciones individuales. La condición (1c) es necesaria ya que, de lo contrario, C(m), la proyección *commiteada* de un schedule m, no es necesariamente completa (es decir, no es una historia).

Claramente un *schedule* monoversión puede ser visto cómo un caso especial de un *schedule* multiversión.

2. Serializabilidad Multiversion

Hay dos nociones de serializabilidad, tenemos por un lado *view serializabilidad* y por otro *conflicto serializabilidad*. Se debe tomar en cuenta que el manejo de versiones debe ser transparente desde afuera del *scheduler*, de tal forma, que sea visto de la misma manera que un *schedule* normal sin versiones. Otro punto importante es que la definición de conflicto equivalencia no puede ser la misma que en un *scheduler* monoversión

Así cómo en los *schedules* monoversión, una transacción t_1 lee un elemento x de otra t_2 cuando t_2 es la última transacción que escribió x antes de que t_1 lo lea, en el caso de multiversión tenemos una defínción equivalente:

Leer de (Reads-From Relation)

Sea m un schedule multiversión, t_i y $t_j \in trans(m)$ La relación *lee-de* se define cómo: $RF(m) := (t_i, x, t_j)|r_j(x_i) \in op(m)$

$trans(m)$ denotan todas las transacciones ocurriendo en m.

2.1. Multiversión view serializabilidad

Tomemos en cuenta que un *schedule* multiversión se llama *schedule* monoversión si su función de versión mapea cada operación de lectura a la última operación de escritura precedente sobre el mismo elemento de datos.

Sean m y m' dos *schedules* multiversión tales que $trans(m') = trans(m)$ entonces m y m' son view equivalentes ($m \approx_v m'$) si $RF(m) = RF(m')$

Multiversión view serializable

Sea m una historia multiversión, se dice que m es multiversión view serializable si existe una historia m' monoversión tal que $m \approx_v m'$.

Denotamos a la clase de historias multiversión serializables como MVSR

2.2. Multiversión conflicto serializabilidad

Primero debemos definir el concepto de conflicto multiversión.

Definición

Un conflicto multiversión en un schedule multiversión m es un par de pasos $r_i(x_j)$ y $w_k(x_k)$ tales que $r_i(x_j) <_m w_k(x_k)$

En este caso estamos diciendo que, a diferencia de lo que veíamos en monoversión, los únicos conflictos que nos interesan son los pares lectura-escritura (rw) sobre el mismo ítem de datos, no necesariamente sobre la misma versión. Es directo ver que dos operaciones de escritura (ww) no tendrían conflicto porque cada una escribe su versión, también podría verse que un par wr tampoco afecta. Los pares importantes entonces son los de rw.

2.3. Conmutatividad

Antes de avanzar en la serializabilidad vamos a definir algunas reglas de conmutatividad que nos permite transformar un *schedule*. Estas reglas son usadas en schedules *monoversión*. Las reglas indican que se puede reemplazar el par de acciones de la mano izquierda con el de la mano derecha.

1. RC1: $r_i(x)r_j(y) \sim r_j(y)r_i(x)$ si $i \neq j$
2. RC2: $r_i(x)w_j(y) \sim w_j(y)r_i(x)$ si $i \neq j, x \neq y$
3. RC3: $w_i(x)w_j(y) \sim w_j(y)w_i(x)$ si $i \neq j, x \neq y$

Estas reglas nos permiten establecer una noción de equivalencia de *schedules* basado en la conmutatividad. Dos *schedules* s_1 y s_2 serían equivalentes basados en conmutatividad si puedo obtener s_2 a partir de s_1 mediante la aplicación de las reglas de conmutatividad. Una historia s es *reducible basada en conmutatividad* si hay una historia serial s' que es equivalente basada en conmutatividad. Para *schedules* monoversión la equivalencia basada en conmutatividad es simétrica y además ocurre que una historia reducible basada en conmutatividad es conflicto serializable.

2.4. Reducibilidad Multiversión

En el caso de multiversión, la conmutatividad no es simétrica dado que los pares conflictivos que nos interesan son los pares rw pero no los wr. La idea es poder producir una historia monoversión serial mediante la aplicación de operaciones de conmutación de modo que siempre se respete el orden los pares conflictivos rw.

Un *schedule* multiversión m es multiversión reducible si se puede transformar en un historial monoversión serial mediante una secuencia finita de pasos de transformación, cada uno de los cuales intercambia el orden de dos pasos adyacentes (es decir, los pasos p, q con $p < q$ tal que $o < p$ o $q < o$ para todos los demás pasos o) pero sin invertir el orden de un par de conflictos multiversión (es decir, rw).

Un historia multiversión m es conflicto multiversión serializable si hay una historia monoversión serial para el mismo conjunto de transacciones en el que todos los pares de operaciones en conflicto multiversión ocurren en el mismo orden que en m. MCSR denota la clase de todas las historias multiversión conflicto serializables.

Teorema

Una historia multiversión es multiversión reducible si y solo si es multiversión conflicto serializable

2.5. Grafo de conflictos multiversión (GCM)

Para verificar que una historia pertenece a la clase MCSR vamos a utilzar una herramienta conocida: el grafo de conflictos multiversión. Sea m una historia multiversión. El grafo del conflicto multiversión de m es un grafo que tiene a las transacciones de m como sus nodos y un eje desde t_i a t_k si hay pasos $r_i(x_j)$ y $w_k(x_k)$ para el mismo elemento de datos x en m tal que $r_i(x_j) <_m w_k(x_k)$

Teorema
Una historia multiversión es MCSR si y solo si su grafo de conflictos es acíclico.

3. Protocolo MVTO

El protocolo ***multiversion timestamp ordering*** lo denotamos cómo MVTO. Básicamente procesa las operaciones a medida que van llegando. Procesa cada operación transformándola en una operación multiversión pero logrando un resultado cómo si fuera un *schedule* monoversión serial. El orden serial equivalente es el orden de los timestamps que son asignados al comienzo de cada transacción. Cada versión de un elemento de datos lleva un timestamp $ts(t_i)$ de la transacción t_i que fue la que creó la versión.

El funcionamiento es el siguiente:

1. Una operación $r_i(x)$ se transforma en una operación multiversión $r_i(w_k)$ donde w_k es la versión de x que tiene el timestamp más grande menor o igual que $ts(t_i)$ y que fue escrita por $t_k, k \neq i$

2. Una operación $w_i(x)$ se procesa de la siguiente manera:

   a) Si una operación $r_j(x_k)$ tal que $ts(t_k) < ts(t_i) < ts(t_j)$ ya existe en el *schedule* entonces $w_i(x)$ es rechada y t_i es abortada. Se produce un *write too late*.

   b) en otro caso $w_i(x)$ se transforma en $w_i(x_i)$ y es ejecutada.

3. Un commit c_i se retrasa hasta que los commit c_j de todas las transacciones t_j que han escrito nuevas versiones de los elementos de datos leídos por t_i hayan sido ejecutadas.

Todos los *schedules* que pueden ser generados por el protocolo **MVTO**, que se denota Gen(**MVTO**), son view multiversión serializables.

Es decir $\text{Gen}(\mathbf{MVTO}) \subseteq \mathbf{MVSR}$

4. Protocolo MV2PL

El MV2PL es un protocolo basado el locking usando *strong strict two-phase locking* o **2PL riguroso**. Vamos a distinguir entre versiones commiteadas, versión actual y versiones no commiteadas. Las primeras son las que fueron escritas por transacciones que ya hicieron *commit*. La versión actual es la commiteada por la

transacción que commiteo último. Las versiones no commiteadas son las creadas por las transacciones aún activas. El planificador se asegura que, en cada momento, haya como máximo una versión no commiteada de cualquier elemento de datos.

La operatoria difiere si un paso de lectura permite leer solo la versión actual o permite leer versiones no commiteadas. El planificador trata el paso final de una transacción de manera diferente a los otros pasos, donde "final"se refiere a la última operación de datos antes del commit de la transacción o al commit en sí. La forma de trabajar es la siguiente:

1. Si el paso no es el final dentro de una transacción:

   a) Una lectura $r(x)$ se ejecuta de inmediato, asignándole la versión actual del elemento de datos solicitado, es decir, la versión commiteada más recientemente (pero no cualquier otra, previamente commiteada), o asignándole una versión no commiteada de x

   b) Un escritura $w(x)$ se ejecuta solamente cuando la transacción que ha escrito x por última vez finalizo, es decir no hay otra versión no commiteada de x

2. Si es el paso final de la transacción t_i, ésta se retrasa hasta que commitean las siguientes transacciones:

   a) todas aquellas transacciones t_j que hayan leído la versión actual de un elemento de datos escrito por t_i

   b) Todas aquellas t_j de las que t_i ha leído alguna versión.

Veamos un ejemplo para el schedule s:
$s = r_1(x); w_1(x); r_2(x); w_2(y); r_1(y); w_2(x); c_2; w_1(y); c_1$
Veamos cada paso

- $r_1(x)$ se le asigna x_0 y se ejecuta: $r_1(x_0)$
- $w_1(x)$ se ejecuta porque no hay otra transacción activa: $w_1(x_1)$
- $r_2(x)$ se le asigna x_1 y se ejecuta: $r_2(x_1)$
- $w_2(y)$ es ejecutada: $w_2(y_2)$
- $r_1(y)$ es asignada a y_0 y se ejecuta: $r_1(y_0)$
- si $w_2(x)$ no fuera el paso final de t_2, se retrasaría ya que t_1 todavía está activa y ha escrito x_1. Sin embargo, como es el paso final de t_2, es necesario aplicar las reglas del paso final. Resulta que t_2, no obstante, **tiene que esperar** por los siguientes motivos:

  - t_1 ha leído la versión actual de y o sea y_0 y t_2 sobrescribe esta versión
  - t_2 ha leído x_1 de t_1.

- $w_1(y)$, es el paso final de t_1, es ejecutado dado que:
  - t_2 no ha leído la versión actual de un ítem de datos que es escrito por t_1 (las versiones actuales son x_0 e y_0).
  - t_1 no ha leído una versión escrita t_2.

  Se ejecuta $w_1(y_1)$

- $w_2(x)$ puede ser ejecutada: $w_2(x_2)$

5. Protoclo 2V2PL

Un caso especial de **MV2PL** es el **2V2PL** el cual mantiene a lo sumo 2 versiones de cada ítem de datos en cada momento. Cuando una transacción t_i escribe un ítem de datos x se tienen dos versiones: la imagen anterior y la imagen posterior. Si t_i commitea la imagen anterior puede ser eliminada dado que la nueva versión de x es estable. Básicamente se mantiene la versión actual y la versión candidata a reemplazar la versión actual.

Este protocolo utiliza 3 tipos de locks:

- rl **read lock**: que se establece antes de una operación de lectura $r(x)$ respecto de la versión actual de x
- wl **write lock**; que se establece antes de una operación de escritura $w(x)$ para escribir una versión no commiteada de x.
- cl **commit o certify lock**:se establece un $cl(x)$ antes de la ejecución del paso final de una transacción en cada elemento de datos x que esta transacción ha escrito .

Las operaciones de unlock deben obedecer al protocolo **2PL**. La matriz de compatibilidad es la siguiente

	rl(x)	wl(x)	cl(x)
rl(x)	+	+	−
wl(x)	+	−	−
cl(x)	−	−	−

La función de los *write locks* es garantizar que, como máximo, pueda existir una versión no commiteada para cada elemento de datos en cada momento. El punto clave para garantizar que los schedulers resultantes sean multiversión serializables es la adquisición de *certify locks*. Las verificaciones relativas a los ordenamientos admisibles de lectura de versiones actuales y creación de nuevas, se expresan en la comprobación de compatibilidad de **read locks** y **certify locks**. En este sentido, los **certify locks** desempeñan el papel que tienen los **write locks** en el locking convencional no versionado. Sin embargo, el hecho de que los **certify locks** se adquieran solo al final de la transacción y, por lo tanto, generalmente se mantengan por un tiempo mucho más corto es una gran ventaja de rendimiento sobre la **2PL** monoversión convencional.

Referencias

[1] Gerhard Weikum and Gottfried Vossen. 2001. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.