Control de Concurrencia Multiversión

Dr. Gerardo Rossel

FCEN, UBA

2021

Timestamping

Un planificador basado en Timestamping:

- Asigna un timestamp distinto a cada transacción
- Garantiza que el orden de ejecución se corresponda con el orden de los timpestamps

Timestamping

Un planificador basado en Timestamping:

- Asigna un timestamp distinto a cada transacción
- Garantiza que el orden de ejecución se corresponda con el orden de los timpestamps

Cómo lo hace:

- Define valores **RT**, **WT** y **C** para cada item
- Cuando llega una lectura o escritura, compara los timestamps del item con el timpestamp (**TS**) de la transacción
- Evita que se produzcan comportamientos físicamente irrealizables

Timestamping

Eventos que producen el abort de una transacción:

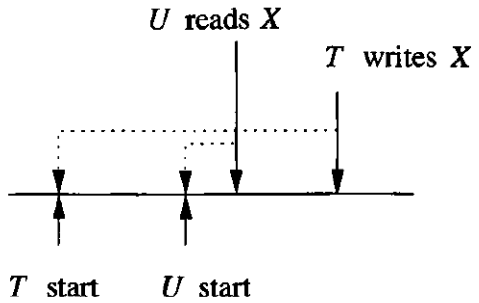


Figure 18.36: Transaction T tries to write too late

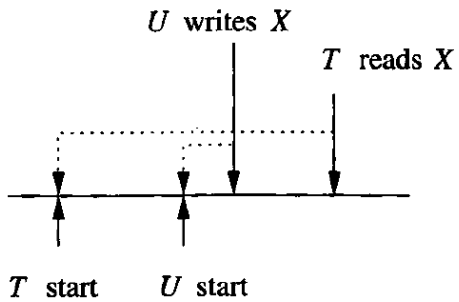


Figure 18.35: Transaction T tries to read too late

Timestamping

- Problema con el write-too-late:
  $\rightarrow$ Se intenta escribir un item que ya fue leído en el futuro
- Se puede solucionar?
  $\rightarrow$ Volver en el tiempo y escribir el valor antes de que sea leído
  Imposible... ✗

Timestamping

- Problema con el write-too-late:
  $\rightarrow$ Se intenta escribir un item que ya fue leído en el futuro
- Se puede solucionar?
  $\rightarrow$ Volver en el tiempo y escribir el valor antes de que sea leído
  Imposible... ✗

- Problema con el read-too-late:
  $\rightarrow$ Se intenta leer un item que posee un valor del futuro
- Se puede solucionar?
  $\rightarrow$ Si tenemos la versión anterior del item? Si! ✓

Timestamping Multiversión

Definición

- Variación/Extensión del planificadores monoversión
- Mantiene versiones históricas de los items
- Permite que las transacciones lean valores antiguos
- **Evita** aborts ocasionados por eventos **read-too-late**

Multiversión

Multiversión

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

Función de Versión

Definición

Sea s una historia. Una **función de versión** para s es una función h, que:

- Asocia cada operación de lectura con una operación de escritura anterior del mismo ítem
- Para operaciones de escritura es la identidad

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 .

Historia Multiversión

Schedule Multiversión

Una historia multiversión contiene operaciones de lecturas y escrituras versionadas para sus transacciones y el orden de las mismas respeta el orden de las tranasacciones individuales

Condición importante

Commits

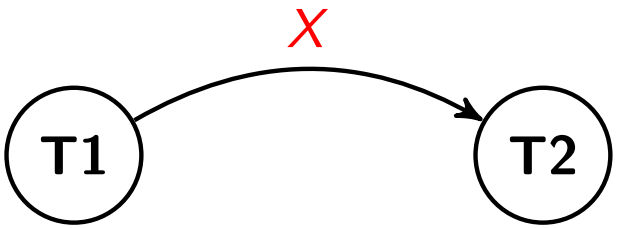
Si $h(r_j(x)) = r_j(x_i), i \neq j$, y c_j está en m, entonces c_i está en m y $c_i <_m c_j$

Pequeño Ejemplo

$$s = r_1(x), w_1(x), r_2(x), w_2(y), r_1(y), w_1(z), c_1, c_2$$

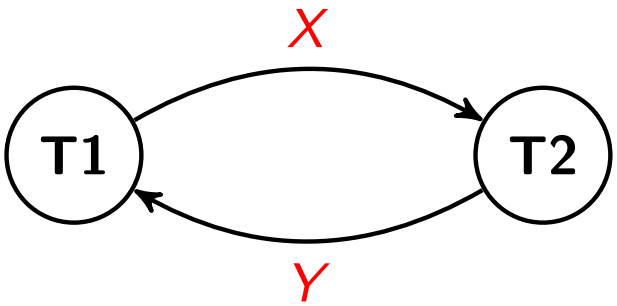
Pequeño Ejemplo

$$s = r_1(x), w_1(x), r_2(x), w_2(y), r_1(y), w_1(z), c_1, c_2$$



Pequeño Ejemplo

$$s = r_1(x), w_1(x), r_2(x), w_2(y), r_1(y), w_1(z), c_1, c_2$$



Serializabilidad Multiversión

- View serializabilidad
- Conflicto Serializabilidad

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: $RF(m) := (t_i, x, t_j) | r_j(x_i) \in op(m)$

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 $RF(m) = RF(m')$

Serializabilidad Multiversión

Multiversión view serializable

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

Conflict graph

$$s = w_0(x), w_0(y), c_0, r_1(x), w_1(x), r_2(x), w_2(y), r_1(y), w_1(z), c_1, c_2$$

Conflict graph

$s = w_0(x), w_0(y), c_0, r_1(x), w_1(x), r_2(x), w_2(y), r_1(y), w_1(z), c_1, c_2$

$s =$

$w_0(x), w_0(y), c_0, r_1(x_0), w_1(x_1), r_2(x_1), w_2(y_2), r_1(y_0), w_1(z_1), c_1, c_2$

El grafo de conflicto de una historia m denotado como $G(m)$ se construye con nodos por cada transacción de my con un eje $t_i \rightarrow t_j$ si $r_j(x_i)$ esta en m

Conflict graph

$s = w_0(x), w_0(y), c_0, r_1(x), w_1(x), r_2(x), w_2(y), r_1(y), w_1(z), c_1, c_2$
$s =$
$w_0(x), w_0(y), c_0, r_1(x_0), w_1(x_1), r_2(x_1), w_2(y_2), r_1(y_0), w_1(z_1), c_1, c_2$
El grafo de conflicto de una historia m denotado como $G(m)$ se construye con nodos por cada transacción de my con un eje $t_i \rightarrow t_j$ si $r_j(x_i)$ esta en m

Para cualquier par de schedulers multiversión, $m \approx_v m'$ entonces $G(m) = G(m)'$.

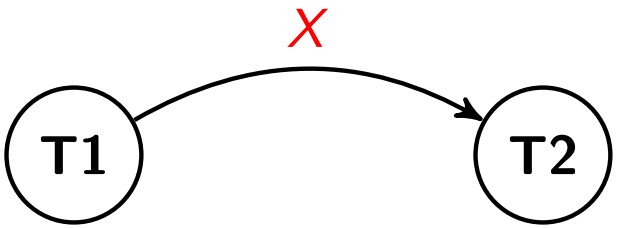
Conflict graph

$s = w_0(x), w_0(y), c_0, r_1(x), w_1(x), r_2(x), w_2(y), r_1(y), w_1(z), c_1, c_2$

$s =$

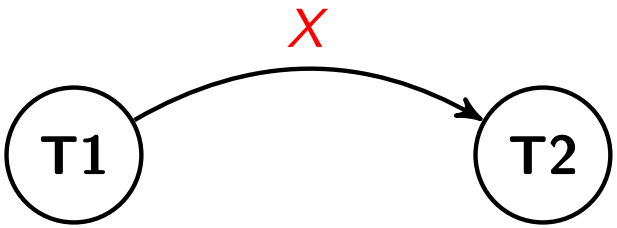
$w_0(x), w_0(y), c_0, r_1(x_0), w_1(x_1), r_2(x_1), w_2(y_2), r_1(y_0), w_1(z_1), c_1, c_2$

El grafo de conflicto de una historia m denotado como $G(m)$ se construye con nodos por cada transacción de my con un eje $t_i \rightarrow t_j$ si $r_j(x_i)$ esta en m



Conflict graph

$s = w_0(x), w_0(y), c_0, r_1(x), w_1(x), r_2(x), w_2(y), r_1(y), w_1(z), c_1, c_2$
$s =$
$w_0(x), w_0(y), c_0, r_1(x_0), w_1(x_1), r_2(x_1), w_2(y_2), r_1(y_0), w_1(z_1), c_1, c_2$
El grafo de conflicto de una historia m denotado como $G(m)$ se construye con nodos por cada transacción de my con un eje $t_i \rightarrow t_j$ si $r_j(x_i)$ esta en m



Para cualquier par de schedulers multiversión, $m \approx_v m'$ entonces $G(m) = G(m)'$.

Conflicto Serializable

Multiversión conflicto serializable

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

Conmutatividad

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 $i \neq j, x \neq y$

Una historia s es *reducible basada en conmutatividad* si hay una historia serial s' que es equivalente basada en conmutatividad

Multiversión conflicto serializable

Un schedule multiversión m es multiversión reducible si se puede transformar en un historial monoversión serial mediante una secuencia finita de pasos de transformación

Multiversión conflicto serializable

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,

Teorema

$$MCSR \subset MVSR$$

Protocolos

- MVTO multiversion timestamp ordering
- MV2PL multiversion two-phase locking
- 2V2PL two version two-phase locking

Multiversion Timestamp Ordering

Cada versión de un elemento de datos lleva un timestamp $ts(t_i)$ de la transacción t_i que fue la que creo la versión.

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 mas 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
   - 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*.
   - 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 ejecutados.

Multiversion two-phase Locking

Teorema

El MV2PL es un protocolo basado el locking usando *strong strict two-phase locking* o **2PL riguroso**.

1. Si el paso no es el final dentro de una transacción:
   1. 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
   2. 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, esta se retrasa hasta que commitean las siguientes transacciones
   1. todas aquellas transacciones t_j que hayan leído la versión actual de un elemento de datos escrito por t_i
   2. Todas aquellas t_j de las que t_i ha leído alguna versión.