Control de Concurrencia Multiversión

Dr. Gerardo Rossel

FCEN, UBA

1er. Cuat. 2024

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

Lectura y Escritura

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.

Motivación

$$s = r_1(x)\ w_1(x)\ r_2(x)\ w_2(y)\ r_1(y)\ w_1(z)\ c_1\ c_2 \qquad \rightarrow \notin \text{CSR}$$

pero el schedule sería tolerable si
$r_1(y)$ pudiera leer la versión y_0 de y
para ser consistente con $r_1(x)$
$\rightarrow$ s sería equivalente a el serial $s' = t_1 t_2$

Enfoque

- cada paso de escritura w crea una nueva versión
- cada paso de lectura r puede elegir qué versión quiere/necesita leer
- las versiones son transparentes para la aplicación y transitorias (es decir, sujetas a recolección de basura)

Sobre Historias y Schedules

Sea $T = \{t_1, \ldots, t_n\}$ un conjunto de transacciones, donde cada $t_i \in T$ tiene la forma $t_i = (op_i, <_i)$ con op_i denotando las operaciones de t_i y $<_i$ su orden.

Una **historia** para T es un par $s = (op(s), <_s)$ tal que:

(a) $op(s) \subseteq \bigcup_{i=1}^{n} op_i \cup \bigcup_{i=1}^{n} \{a_i, c_i\}$

(b) para todo i, $1 \leq i \leq n$: $c_i \in op(s) \iff a_i \notin op(s)$

(c) $\bigcup_{i=1}^{n} <_i \subseteq <_s$

(d) para todo i, $1 \leq i \leq n$, y todo $p \in op_i$: $p <_s c_i$ o $p <_s a_i$

(e) para todo $p, q \in op(s)$ tal que al menos uno de ellos es una escritura y ambos acceden al mismo elemento de datos: $p <_s q$ o $q <_s p$

Un **schedule** es un prefijo de una historia

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

U**na historia multiversión** (**mv**) para las transacciones $T = \{t_1, \ldots, t_n\}$ es un par $m = (op(m), <_m)$ donde $<_m$ es un orden en $op(m)$ y

(a) $op(m) = \bigcup_{i=1}^{n} h(op(t_i))$ para alguna función de versión h,

(b) para todo $t \in T$ y todo $p, q \in op(t_i)$:
$p <_t q \Rightarrow h(p) <_m h(q)$,

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

Un **schedule** multiversión (mv) es un prefijo de una historia multiversión.

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 sobre 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 $c_i <_m c_j$

Un planificador multiversión es un **planificador monoversión** si su función de versión asigna cada lectura a la última escritura precedente en el mismo elemento de datos.

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 $m \approx_v m'$.

MVSR es la clase de historias *view* serializables (serializables por vista) multiversión.

Decidir si una historia esta en MSVR es un problema NP-Completo

Grafo de Conflictos

El grafo de conflictos de una historia m denotado como $G(m)$ se construye con nodos por cada transacción de m 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** $G(m) = G(m)'$.

Grafo de Conflictos

El grafo de conflictos de una historia m denotado como $G(m)$ se construye con nodos por cada transacción de m 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** $G(m) = G(m)'$.

$m = w_1(x_1)r_2(x_0)w_1(y_1)r_2(y_1)c_1c_2$
$m' = w_1(x_1)w_1(y_1)c_1r_2(x_1)r_2(y_0)c_1$

$G(m) = G(m)'$ pero $m \not\approx_v m'$

MCSR

MCSR

Vamos a trabajar con una subclase de MVSR cuya pertenencia puede determinarse en tiempo polinómico.

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

Pasos de transformación

Un paso de transformación intercambia el orden de dos pasos adyacentes (es decir, pasos p, q con $p < q$ tal que $o < p$ y $q < o$ para todos los demás pasos o) pero sin invertir el orden de un par de conflicto multiversión (es decir, rw).

Reducibilidad multiversión

Una historia de multiversión es *multiversión reducible* si puede transformarse en una historia serial de monoversión mediante una secuencia de pasos de transformación.

Multiversión conflicto serializable

Un historia multiversión m es multiversión conflicto 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,

$$MCSR \subset MVSR$$

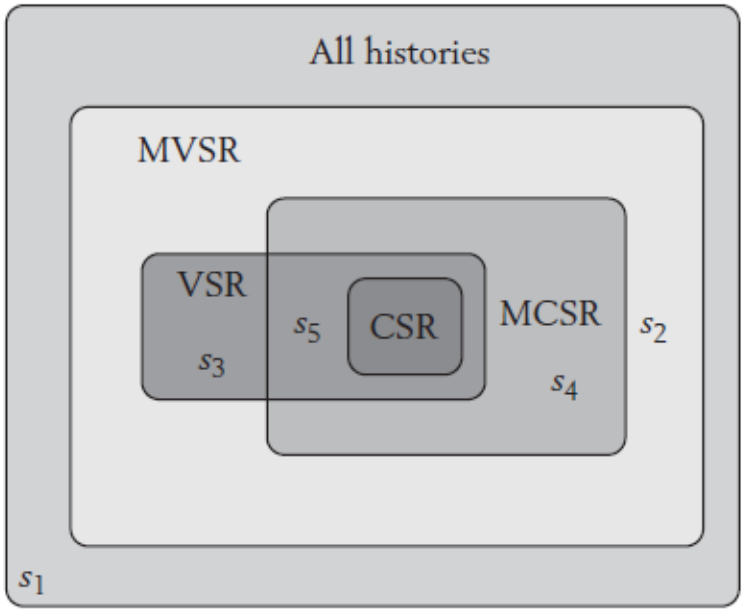
Grafo de conflicto multiversión

Sea m un schedule de multiversión. El *grafo de conflicto multiversión* de m es un grafo que tiene las transacciones de m como sus nodos y una arista de t_i a t_k si existen pasos $r_i(x_j)$ y $w_k(x_k)$ para el mismo elemento de datos x en m tal que $r_i(x_j) <_m w_k(x_k)$.

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

$m \in MCSR \iff$ multiversión reducible *iff* grafo conflicto multiversión es acíclico.

Resumen



$s_1 = r_1(x)r_2(x)w_1(x)w_2(x)c_1c_2$
$s_2 = w_1(x)c_1r_2(x)r_3(y)w_3(x)w_2(y)c_2c_3$
$s_3 = w_1(x)c_1r_2(x)r_3(y)w_3(x)w_2(y)c_2c_3w_4(x)c_4$
$s_4 = r_1(x)w_1(x)r_2(x)r_2(y)w_2(y)r_1(y)w_1(y)c_1c_2$
$s_5 = r_1(x)w_1(x)r_2(x)w_2(y)c_2w_1(y)w_3(y)c_1c_3$

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.

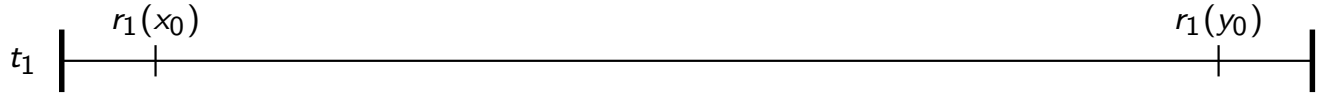
1. Una operación $r_i(x)$ se transforma en una operación multiversión $r_i(w_k)$ donde w_k es la versión de x que tiene el timestamp mas grande menor o igual que $ts(t_i)$ y que fue escrita por $t_k, k \neq i$
2. Una operación $w_i(x)$ se procesa de la siguiente manera
   - Si una operación $r_j(x_k)$ 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.

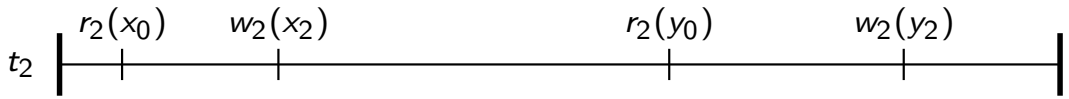
Ejemplo MVTO

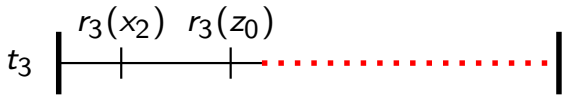
$r_1(x); r_2(x); w_2(x); r_3(x); r_2(y); r_4(x); r_3(Z); w_4(X); w_2(y); r_4(y); r_5(z); r_1(y); w_4(y)$

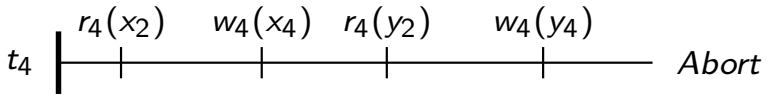
Ejemplo MVTO

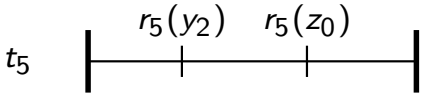
$r_1(x); r_2(x); w_2(x); r_3(x); r_2(y); r_4(x); r_3(Z); w_4(X); w_2(y); r_4(y); r_5(z); r_1(y); w_4(y)$











Multiversion two-phase Locking

MV2PL

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

1. *versiones commiteadas*, que han sido escritas por transacciones que ya están commiteadas,
2. *versión actual* es la versión commiteada de ese elemento de datos escrita por la última transacción commiteada;
3. *versiones no commiteadas*, que son todas las versiones restantes (creadas por transacciones que todavía están activas).

Multiversion two-phase Locking

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 algún elemento.

2V2PL protocol

El protocolo 2V2PL (bloqueo de dos versiones) mantiene como máximo dos versiones de cualquier elemento de datos en cada momento.

- Supongamos que t_i escribe el elemento de datos x, pero aún no está *comiteado*, las dos versiones de x son su imagen anterior y su imagen posterior.
- Tan pronto como t_i se comitea, la imagen anterior puede ser eliminada ya que la nueva versión de x ahora es estable, y las versiones antiguas ya no son necesarias ni se mantienen.

En 2V2PL las operaciones de lectura están restringidas a leer solo las versiones actuales, es decir, la última versión comiteada

2V2PL protocol - 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)	−	−	−

Ejercicio

Suponga el siguiente schedule que se quiere ejecutar sobre un motor que soporta **2V2PL**.

$$r_1(x); w_2(y); r_1(y); w_1(x); c_1; r_3(y); r_3(z); w_3(z); w_2(x); c_2; w_4(z); c_4; c_3$$

MV2PL

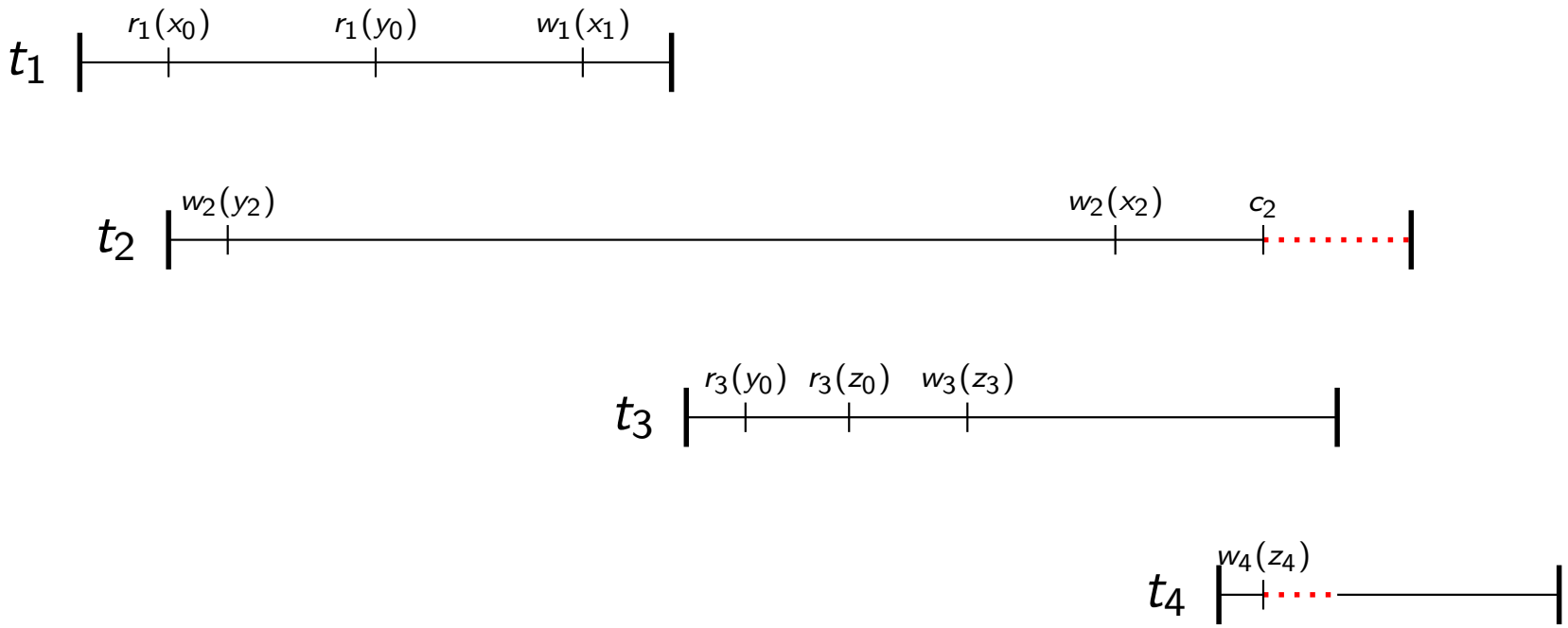
Indicar que ocurre en cada paso y en que lugar las operaciones deben ser demoradas. Escriba el *schedule* final indicando *locks y unlocks*.

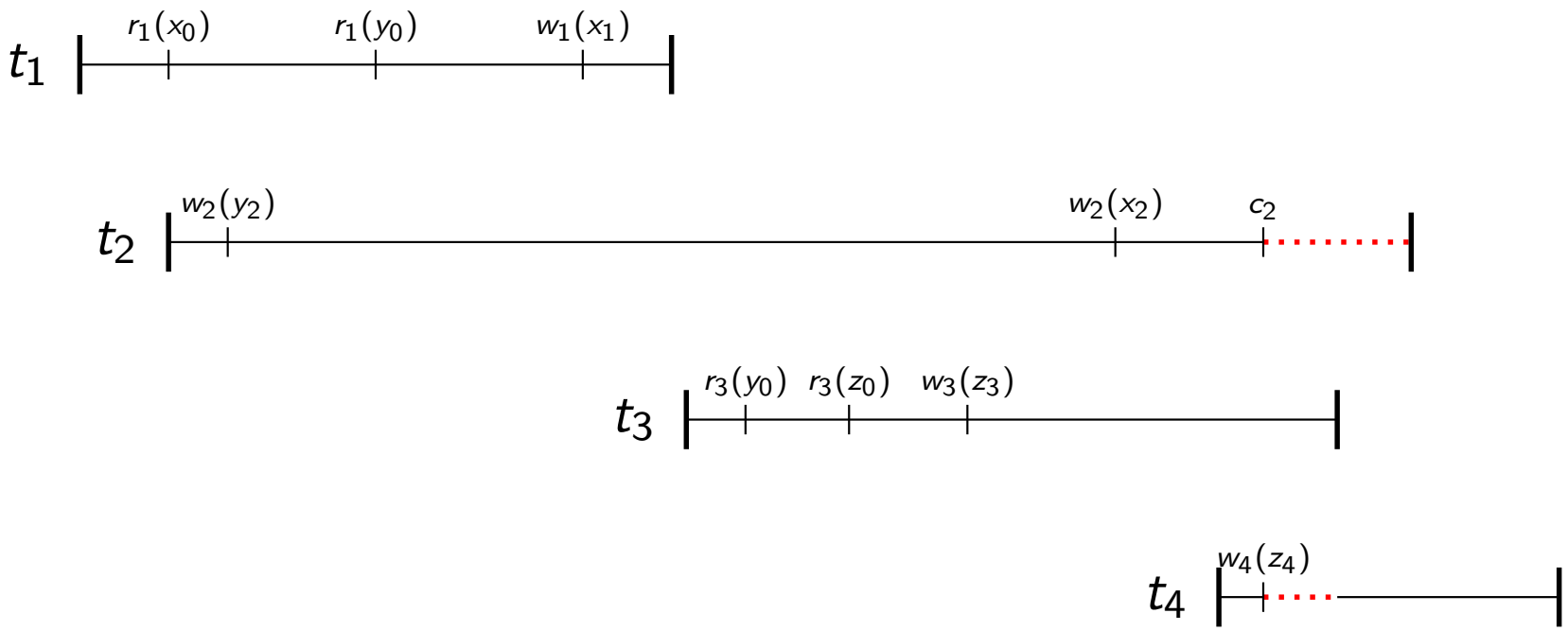
Solución

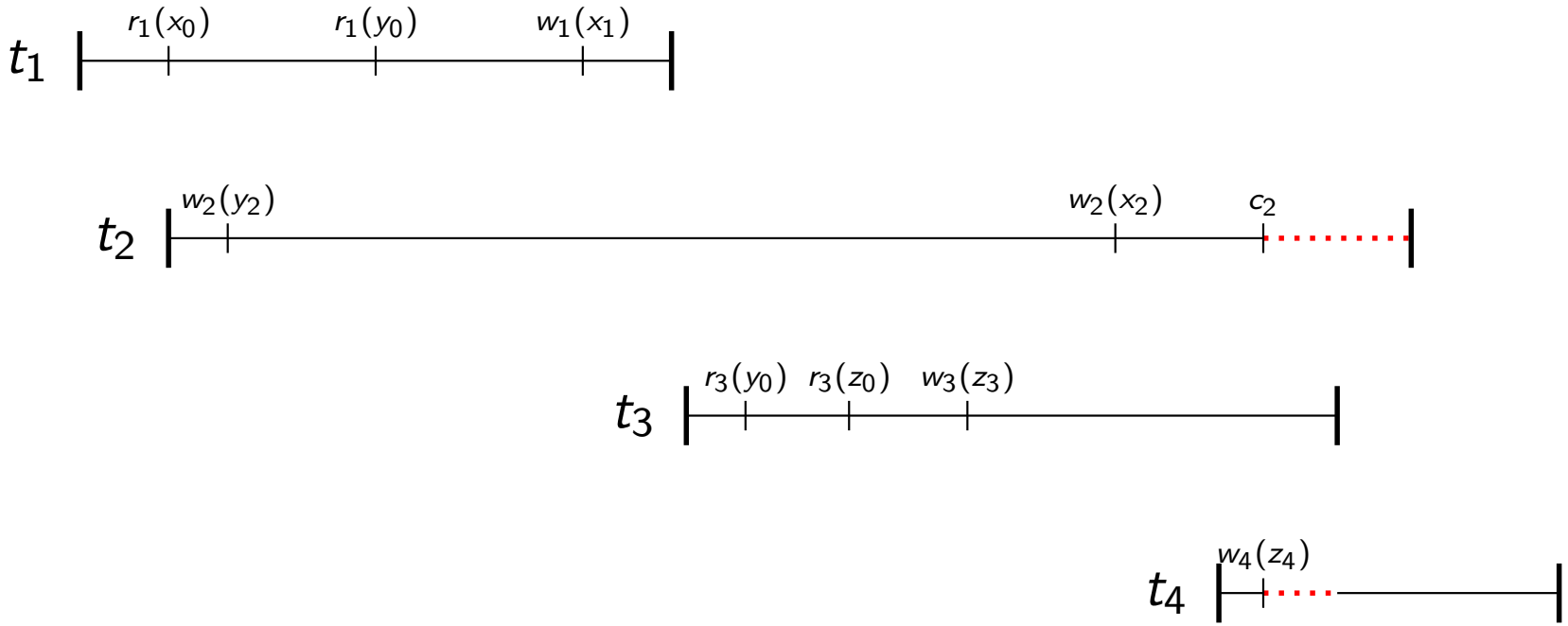
$$r_1(x); w_2(y); r_1(y); w_1(x); c_1; r_3(y); r_3(z); w_3(z); w_2(x); c_2; w_4(z); c_4; c_3$$

Solución

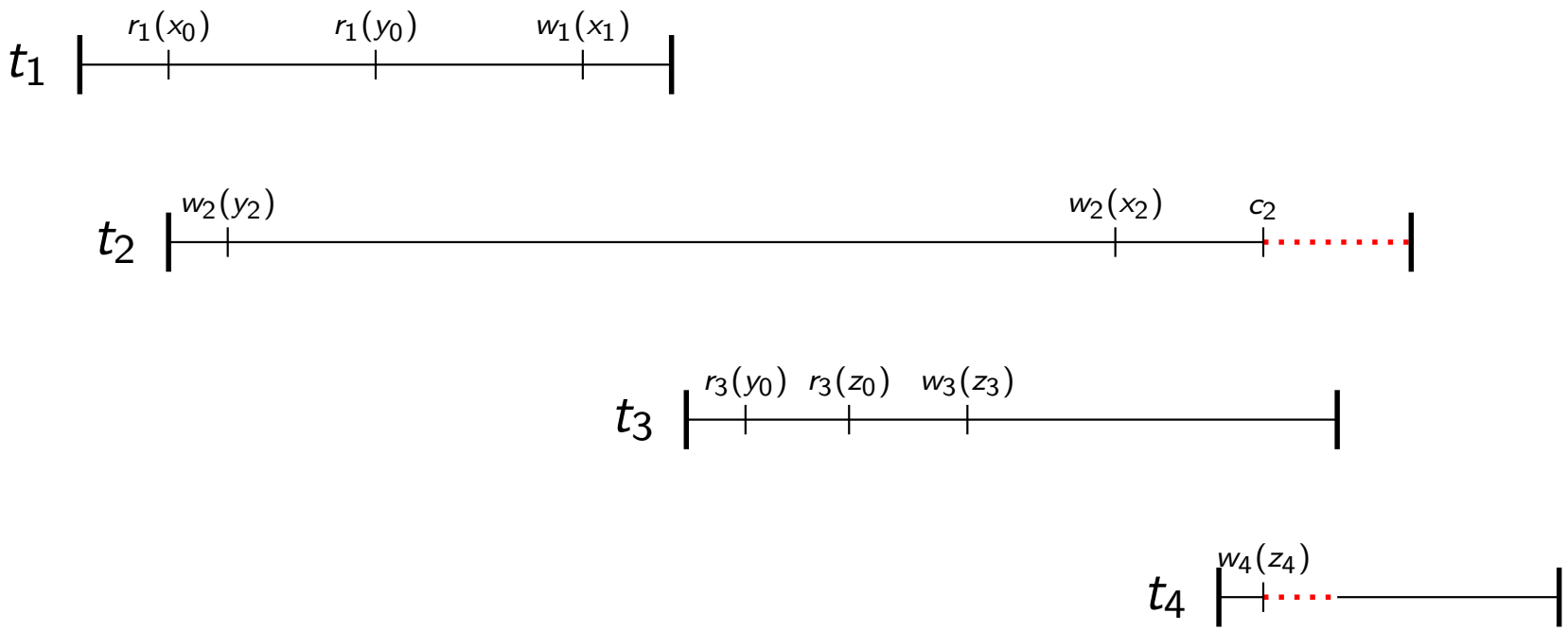
$$r_1(x); w_2(y); r_1(y); w_1(x); c_1; r_3(y); r_3(z); w_3(z); w_2(x); c_2; w_4(z); c_4; c_3$$



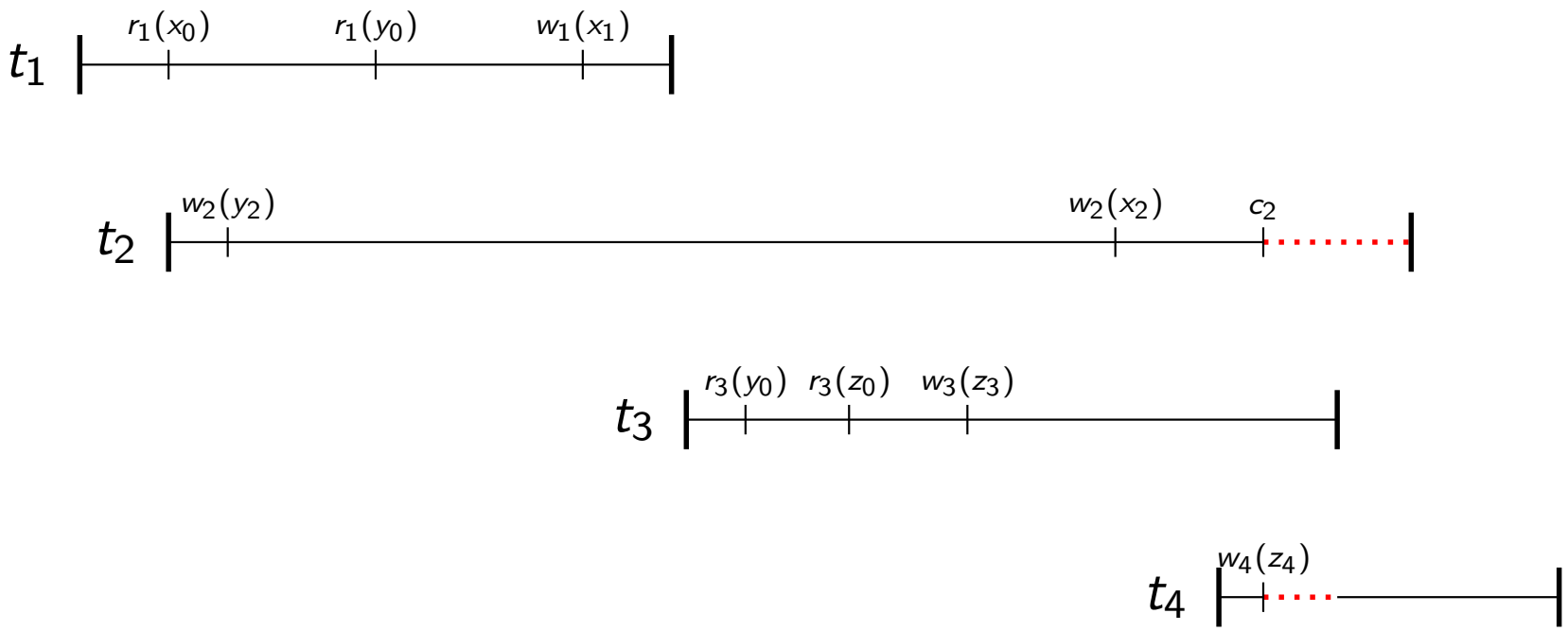

t_1
$r_1(x_0)$
$r_1(y_0)$
$w_1(x_1)$
t_2
$w_2(y_2)$
$w_2(x_2)$
c_2
t_3
$r_3(y_0)$
$r_3(z_0)$
$w_3(z_3)$
t_4
$w_4(z_4)$



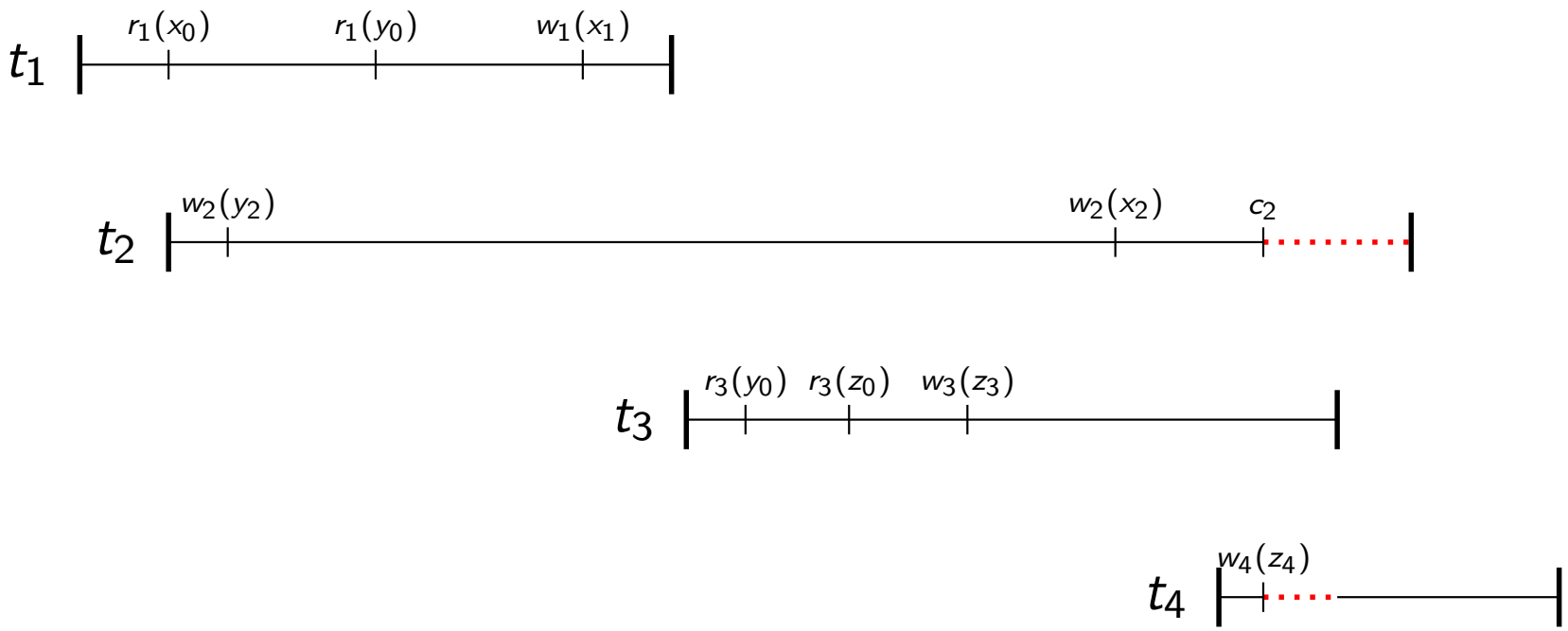
$rl_1(x); r_1(x_0); wl_2(y); w_2(y_2); rl_1(y); r_1(y_0); wl_1(x); w_1(x_1); cl_1(x), ul_1;$
$c_1; rl_3(y); r_3(y_0); rl_3(z); r_3(z_0); wl_3(z); w_3(z_3); wl_2(x); w_2(x_2);$
$cl_2(x); cl_3(z); ul_3; c_3; cl_2(y); ul_2; \ c_2; w_4(z_4); cl_4(z); ul_4; c_4$



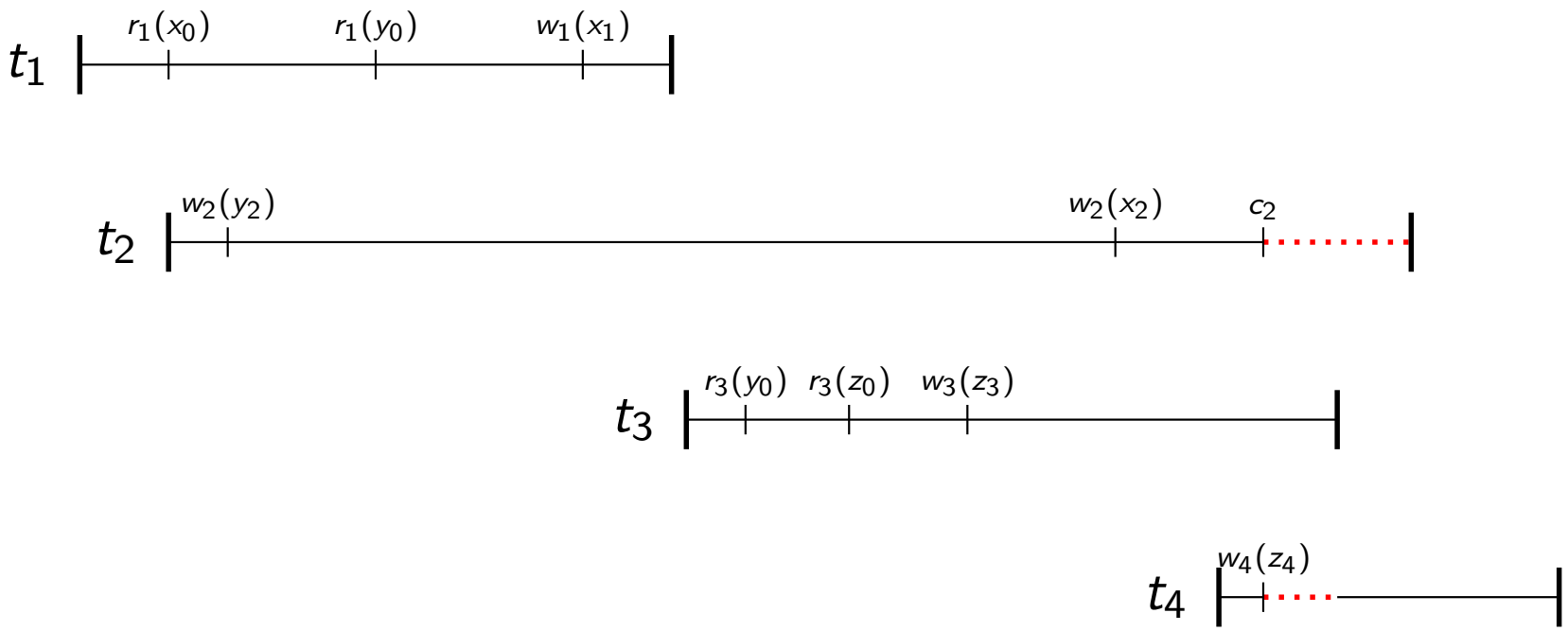
X	Y	Z
x_0	y_0	z_0



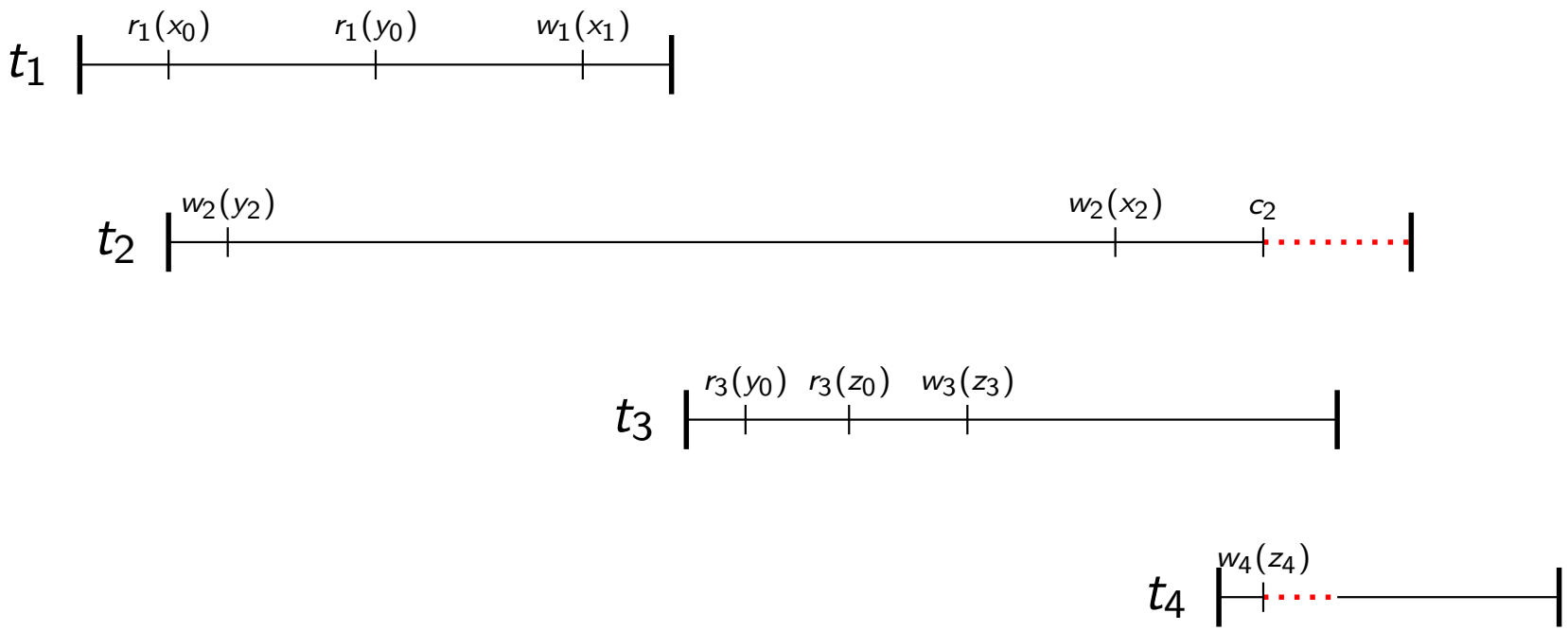
X	Y	Z
x_0	y_0 y_2	z_0



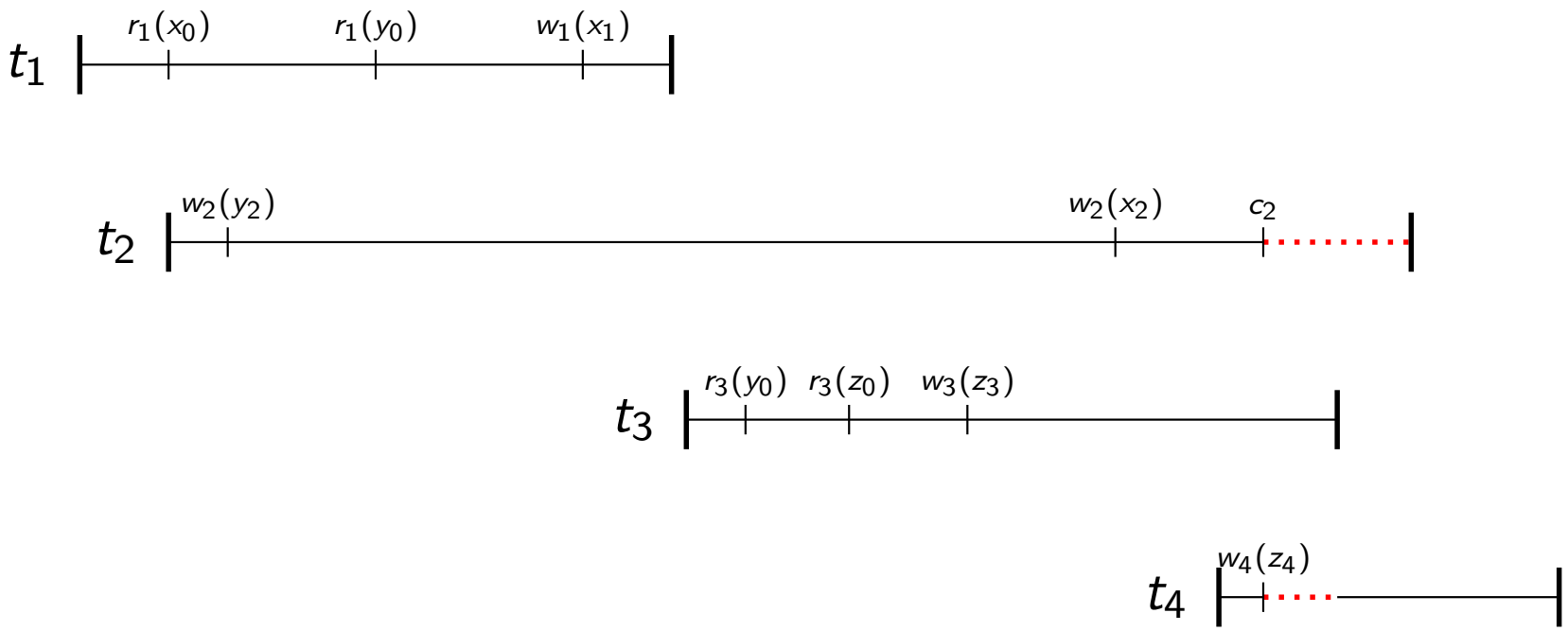
X		Y		Z
x_0	x_1	y_0	y_2	z_0



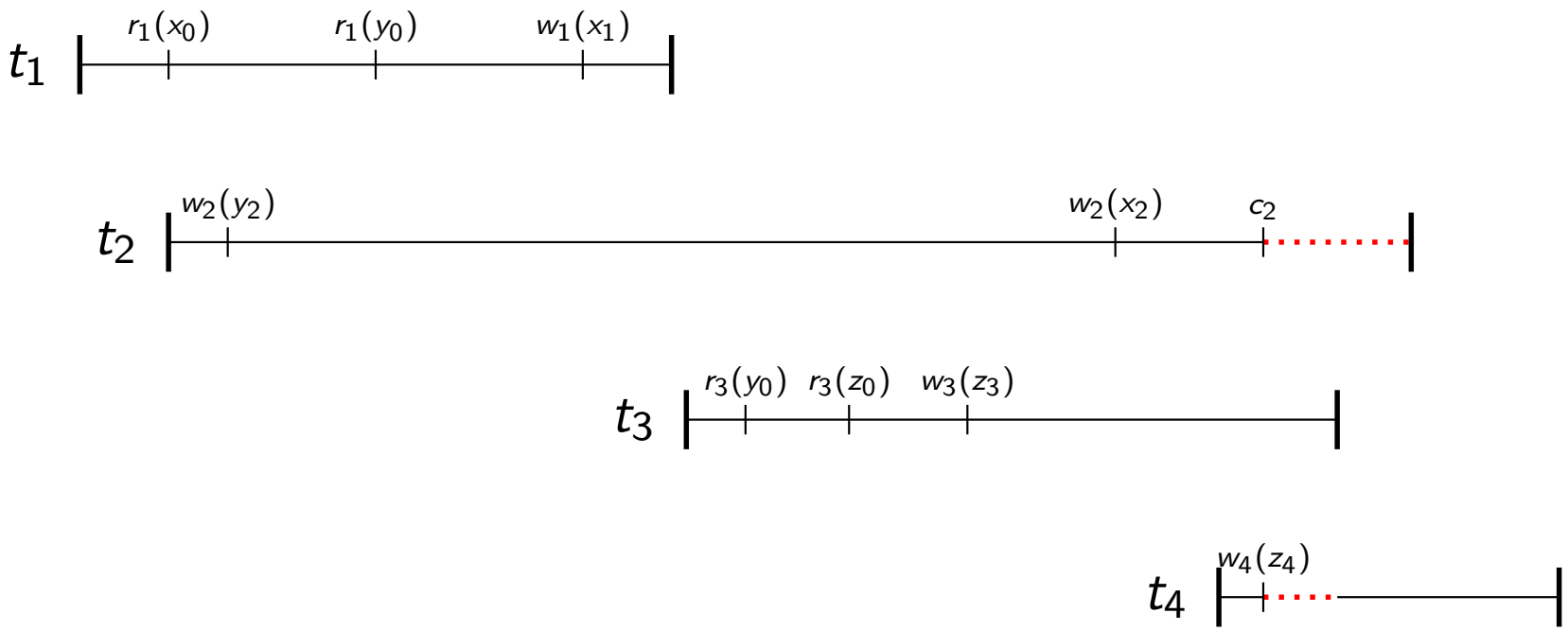
X	Y	Z
x_1	y_0 y_2	z_0



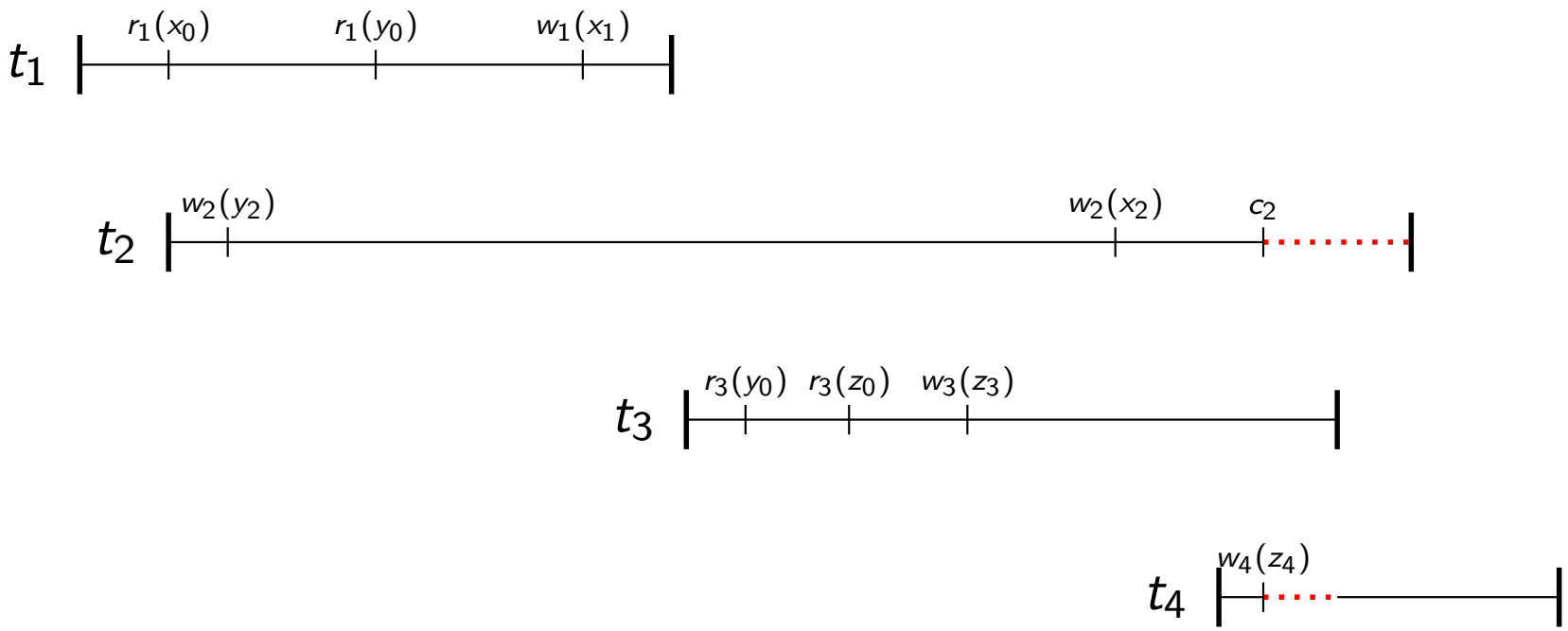
X	Y		Z	
x_1	y_0	y_2	z_0	z_3



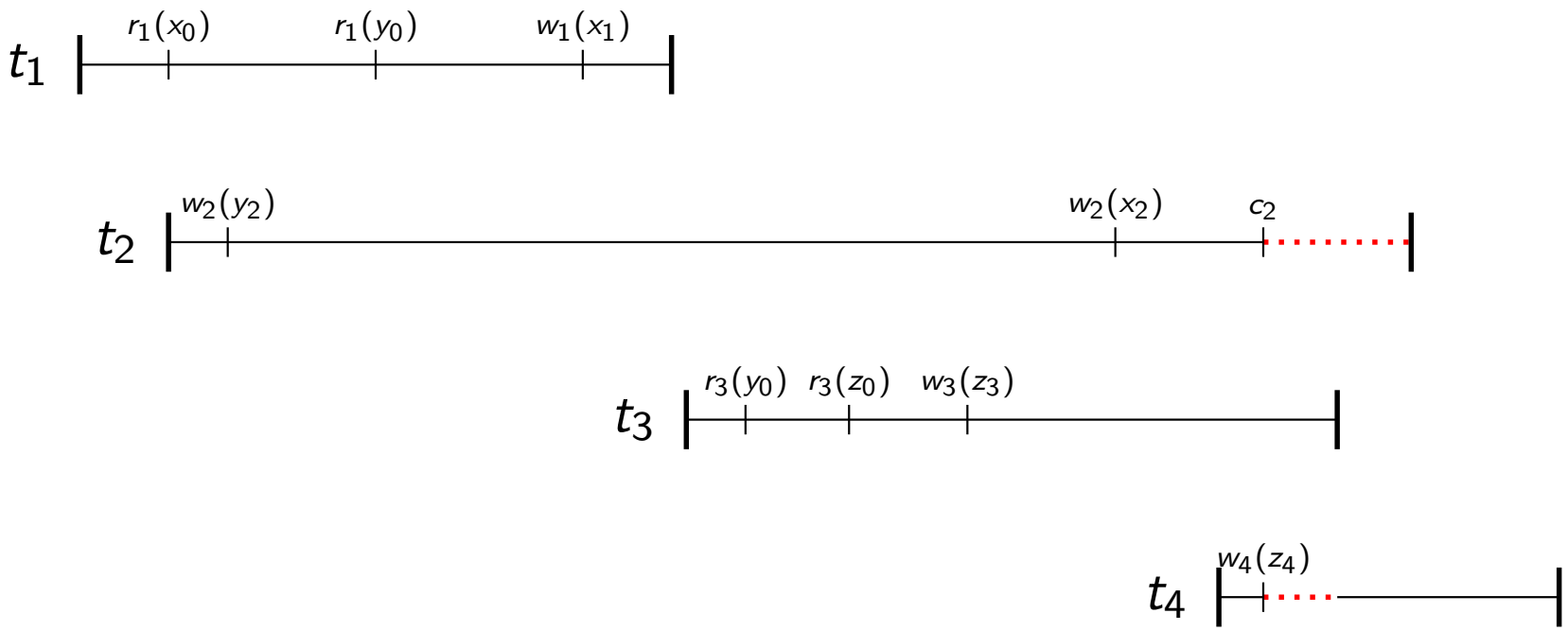
X		Y		Z	
x_1	x_2	y_0	y_2	z_0	z_3



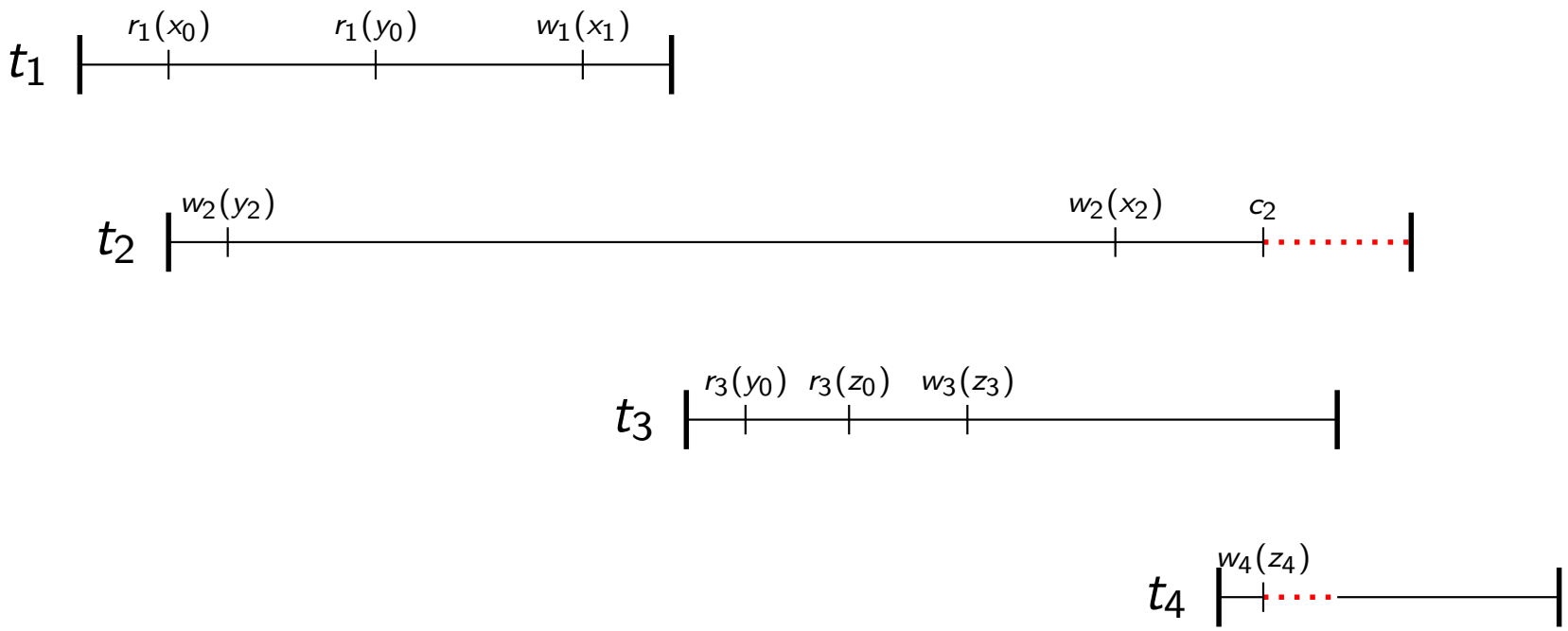
X		Y		Z
x_1	x_2	y_0	y_2	z_3



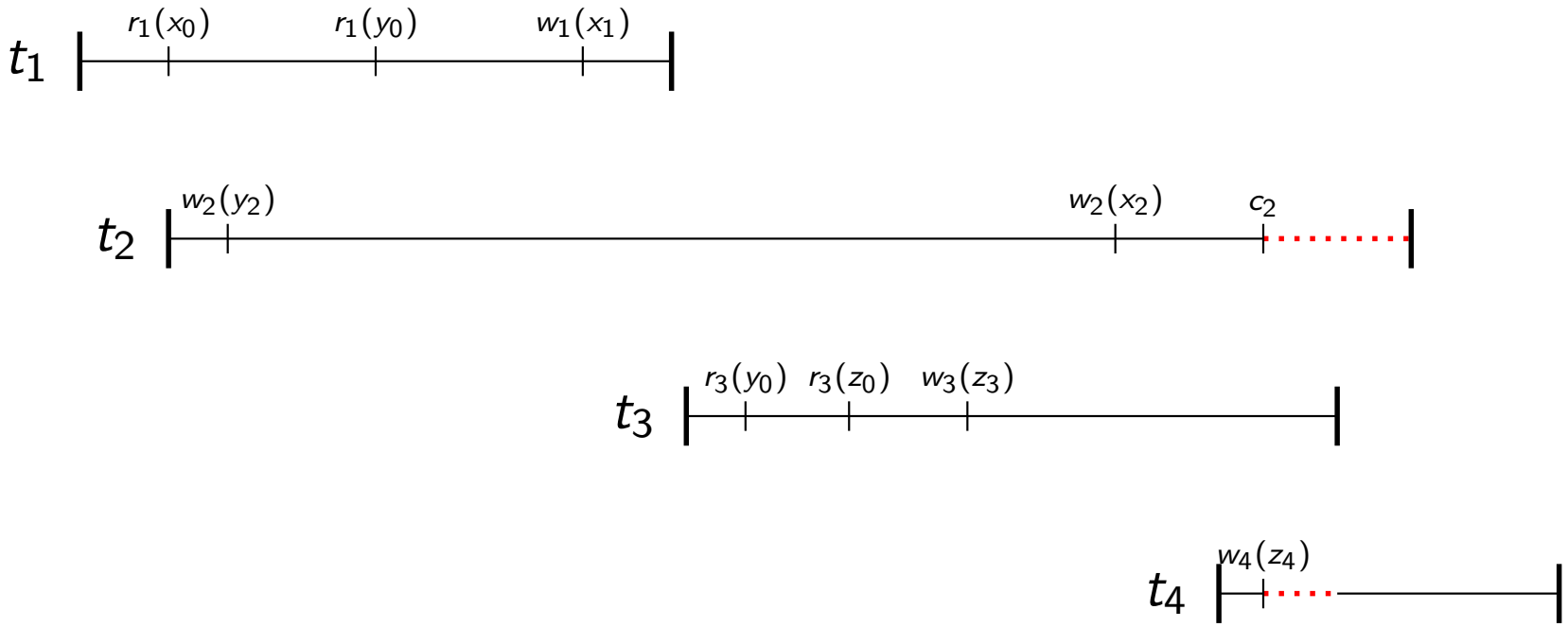
X	Y	Z
x_2	y_2	z_3



X	Y	Z	
x_2	y_2	z_3	z_4



X	Y	Z
x_2	y_2	z_4



X	Y	Z
x_2	y_2	z_4

Orden serial equivalente: $t_1; t_3; t_2; t_4$

Bibliografía

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.