

Tópicos Avanzados en Bases de Datos Relacionales

Protocolos para gestión de fallas o caídas del sistema

Protocolos de control de accesos concurrentes a la base de datos

Tipos de fallas

- Errores de Lógica: El programa no puede seguir ejecutándose debido a un problema interno, como un error I/O, un overflow, etc.
- Errores del Sistema: Por algún motivo el sistema operativo decide abortar un programa (por ejemplo deadlock).
- Caídas del Sistema: Un error de software o hardware que ocasiona pérdida de información en memoria volátil.
- Fallas del Disco: Error en el hardware del disco que ocasiona la pérdida parcial o total de su contenido.

Consideremos el siguiente programa P :

```
P : read(A,a)
    a = a - 50
    write(A,a)
    read(B,b)
    b = b + 50
    write(B,b)
```

A,B: identificadores de registros de la B.D.

a,b: variables del programa.

Supongamos que inicialmente $a = 1000$ y $b = 2000$, entonces después de la ejecución de P se obtiene $a = 950$ y $b = 2050$. Si ocurre una falla después de ejecutar `write(A,a)`, pero antes de ejecutar `write(B,b)`, obtenemos $a = 950$ y $b = 2000$, lo que impide dejar la base de datos en un estado consistente. Re-ejecutar P no es una solución adecuada porque obtendríamos $a = 900$ y $b = 2050$. Necesitamos que P sea considerado como una operación atómica y un mecanismo que nos asegure que P se ejecute completamente o no se ejecute completamente.

Transacciones

Es una unidad de programa cuya ejecución conserva la consistencia de la base de datos. Si antes de la ejecución, la base de datos estaba en un estado consistente, entonces después de la ejecución de la transacción la base de datos continúa en un estado consistente.

El DBMS debe asegurar que las transacciones sean atómicas. Si ocurre una falla durante la ejecución de una transacción, al “volver” el sistema, el DBMS debe deshacer todos los cambios que la transacción haya realizado en la base de datos. A esto se le llama ROLLBACK. Si la transacción alcanza a ejecutarse completamente se dice que está COMMITED (COMMIT=cometer, perpetrar).

Se definen los siguientes estados para una transacción (figura 6.1):

- ACTIVA: Está en ejecución.
- PARCIALMENTE TERMINADA: Después de ejecutarse la última instrucción de la transacción.
- CAÍDA: Cuando se descubre que no puede continuar su ejecución normal.
- TERMINADA: La transacción termina exitosamente. Si posteriormente ocurre una falla sus efectos no se pierden.

Para poder deshacer los cambios a la base de datos realizados por una transacción que es abortada, los cambios (writes) se registran en un archivo llamado BITÁCORA (LOG). Existen dos estrategias para realizar esto:

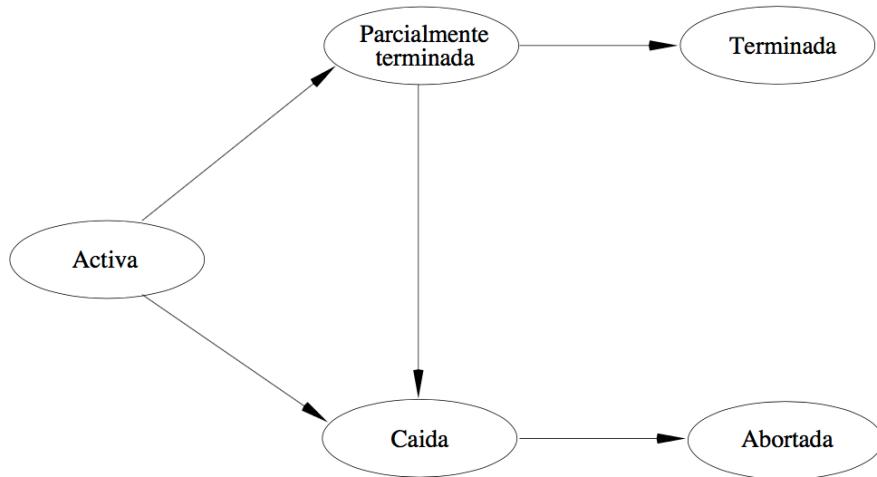


Figure 6.1: Estados de una transacción.

Note que en el programa de aplicación el usuario puede indicar el comienzo y término de una transacción con:

```

BEGIN TRANSACTION
    read(A,a)
    a = a - 50
    write(A,a)
END TRANSACTION

```

Postergar actualizaciones y mantener una bitácora

Durante la ejecución de la transacción, los “write” son postergados hasta llegar al estado “Parcialmente Terminada”. Las actualizaciones son registradas en un archivo llamado bitácora. Cuando la transacción llega a estado “Parcialmente Terminada” se usa el contenido de la bitácora para ejecutar los “writes” reales en la base de datos. Si ocurre una falla en la mitad de la ejecución de una transacción, simplemente se ignora el contenido de la bitácora.

En la bitácora se graba la siguiente información:

- Justo antes de comenzar la ejecución de una Transacción T :

$T \text{ start}$

- Por cada $\text{write}(X, x)$ que T ejecute:

T, X, x

- Cuando T llega al estado “Parcialmente Terminada”:

$T \text{ commit}$

Después de “ $T \text{ commit}$ ” el DBMS intenta ejecutar los writes de la bitácora a partir de “ $T \text{ start}$ ”. Antes de hacer esto el DBMS se asegura que la bitácora está realmente grabada en disco (buffer de memoria principal grabados a disco). Después de ejecutar los “writes”, la transacción pasa a estado “Terminada”.

Supongamos que después de una Falla, el contenido de la bitácora es:

```
T start  
T, A, 950  
T, B, 2050
```

En este caso el DBMS no ejecuta los writes en la base de datos. En cambio, sí lo hará si el contenido de la bitácora es:

```
T start  
T, A, 950  
T, B, 2050  
T commit
```

Al volver el sistema el DBMS no sabe si realmente se ejecutaron los “writes” de la transacción en la base de datos, y por lo tanto el DBMS hace:

REDO (T)

que ejecuta los “writes” correspondientes a T , basándose en el contenido de la bitácora.

Supongamos que después de una “caída del sistema”, el contenido de la bitácora es:

```
T1 start  
T1, A, 950  
T1, B, 2050  
T1 commit  
T2 start  
T2, C, 600  
T2 commit
```

Aunque se sabe que se realizaron los “writes” correspondientes a T_1 , el DBMS no sabe si los buffers de memoria principal alcanzaron a escribirse a disco antes de la caída del sistema, luego el DBMS hace:

```
REDO (T1)  
REDO (T2)
```

En general, debe hacerse un “REDO” de todas las transacciones para las que existe un “COMMIT” en la bitácora.

Ejecutar actualizaciones inmediatamente y mantener la bitácora

Esta técnica consiste en ejecutar las instrucciones “write” inmediatamente y simultáneamente mantener una bitácora de los “write” ejecutados. Si ocurre una falla en medio de una transacción, el contenido de la bitácora se usa para deshacer las actualizaciones que haya alcanzado a hacer la transacción.

En la bitácora se graba la siguiente información:

- Registros “`T start`” y “`T commit`”, donde T es una transacción, con el mismo significado que en el esquema anterior.
- Antes de ejecutar cada instrucción `write(X,x)` se graba un registro

`T,X,<valor anterior>,x`

Por ejemplo, si después de una falla el contenido de la bitácora es:

```
T start
T, A, 1000, 950
T, B, 2000, 2050
```

se debe ejecutar:

`UNDO(T)`

que deshace las actualizaciones efectuadas por T .

Si ocurre una falla durante el proceso de recuperación, entonces simplemente, al valor el sistema, se debe ejecutar el proceso desde un comienzo.

Note que tal como antes el asunto se complica debido a que ejecutar una instrucción `write`, no necesariamente significa escribir en el disco inmediatamente, sino que modificar un buffer en memoria principal. La copia de este buffer desde memoria principal al disco puede ocurrir bastante después. La idea es que el contenido de la bitácora sirve para asegurar que lo que no aparece en ella no se ejecutó. Sin embargo, lo que sí aparece en ella puede o no haber alcanzado a grabarse en disco.

Tal como en el esquema anterior, se debe efectuar un

`REDO(T)`

para cada transacción que aparezca con `commit` en la bitácora. La función `REDO` vuelve a efectuar los writes correspondientes a la transacción T .

Cabe destacar que en este esquema, al igual que en el esquema anterior en que se postergan las actualizaciones, se debe grabar en memoria estable los buffers correspondientes a la bitácora antes de grabar en disco cualquier otro buffer que esté en memoria principal.

Checkpoints

En principio, después de una falla se debe examinar toda la bitácora para saber cuando hacer *redo* y *undo* de una transacción. Esto tiene algunas desventajas:

- El proceso es costoso en tiempo y espacio.
- Muchas de las transacciones para las que se hace el *redo*, en realidad ya han grabado en disco las modificaciones.

Para disminuir el costo de este proceso se introduce el concepto de *checkpoint*. Esto considera los siguientes pasos:

- Escribir todos los registros de la bitácora que estén en memoria principal a memoria estable.
- Escribir en disco todos los buffers que hayan sido modificados.
- Escribir en la bitácora, en memoria estable, un registro

<checkpoint>

De este modo, después de una falla sólo se debe analizar la bitácora desde el <checkpoint> en adelante. En rigor desde la transacción que se estaba ejecutando durante el proceso de *checkpoint*.

Concurrencia en la base de datos

Se desea permitir que varias transacciones puedan estar operando simultáneamente sobre la base de datos. El DBMS debe coordinar las transacciones concurrentes para que no se pierda la consistencia de la base de datos.

Consideremos las transacciones siguientes:

T0: `read(A,a)`
 `a = a - 50`
 `write(A,a)`
 `read(B,b)`
 `b = b + 50`
 `write(B,b)`

T1: `read(A,a)`
 `temp = A*0.1`
 `a = a - temp`
 `write(A,a)`
 `read(B,b)`
 `b = b + temp`
 `write(B,b)`

La ejecución secuencial de T0 y T1 es:

T0 => $(A, a) = 950$
 $(B, b) = 2050$ $A + B = 3000$

T1 => $(A, a) = 855$
 $(B, b) = 2145$ $A + B = 3000$

Supongamos que T0 y T1 se ejecutan concurrentemente:

T0	T1	A	B
<code>read(A,a)</code>		1000	2000
<code>a = a - 50</code>			
	<code>read(A,a)</code>		
	<code>temp = a*0.1</code>		
	<code>a = a - temp</code>		
	<code>write(A,a)</code>	900	
	<code>read(B,b)</code>		
<code>write(A,a)</code>		950	
<code>read(B,b)</code>			
<code>b = b - 50</code>			
<code>write(B,b)</code>			2050
	<code>b = b + temp</code>		
	<code>write(B,b)</code>	950	2100

Si se tratara de un banco, la cantidad total de dinero ($A+B$) debería mantenerse constante, después de hacer traspasos de dineros entre las cuentas A y B . Pero en el ejemplo de ejecución concurrente de T0 y T1 el resultado final es $A + B = 3050$. Es decir, la ejecución concurrente de T0 y T1 ha dejado en un estado inconsistente a la base de datos.

Definimos esquema de un conjunto de transacciones como el orden en que se ejecutan las instrucciones de dichas transacciones.

Un esquema es *serial* si las instrucciones de cada transacción aparecen en forma consecutiva.

Un esquema es *serializable* si su efecto es el mismo que un esquema serial.

Por ejemplo:

T0
read(A,a)
a = a - 50
write(A,a)

T1

read(A,a)
temp = a*0.1
a = a - temp
write(A,a)

read(B,b)
b = b + 50
write(B,b)

read(B,b)
b = b + temp
write(B,b)

Decidir si un esquema es serializable es muy difícil y costoso. Lo que se hace es establecer criterios más estrictos que la simple comparación de los efectos finales de los esquemas.

Reads* antes de *writes

Supongamos un modelo en el cual una transacción tiene que leer un ítem antes de escribir en él. Es decir, el valor que escribe depende del valor anterior del ítem.

Para saber si un esquema es serializable se debe construir un grafo de precedencia del esquema.

El Grafo de precedencia es un grafo dirigido en que cada nodo corresponde a una transacción y los arcos se crean de acuerdo a las siguientes reglas:

- Si T_j lee el valor de un ítem A , que fue escrito por T_i , entonces se crea un arco $T_i \rightarrow T_j$.
- Si T_i lee el valor de un ítem A y a continuación T_j lo modifica entonces existe un arco $T_i \rightarrow T_j$.

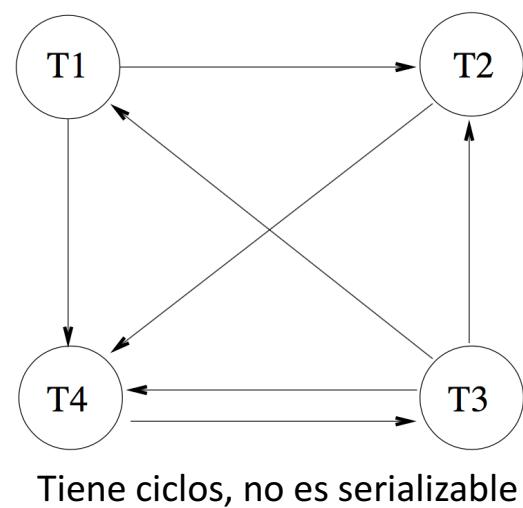
Un arco $T_i \rightarrow T_j$ en el grafo indica que en un esquema serial equivalente al analizado, T_i debería ejecutarse antes que T_j .

T1	T2	T3	T4
		read(A,a) write(A,a)	
		read(B,b)	read(B,b)
	read(A,a)		
read(A,a) read(B,b) write(B,b)			
		read(A,a) write(A,a)	
	read(B,b)		

Si T_j lee el valor de un ítem A , que fue escrito por T_i , entonces se crea un arco $T_i \rightarrow T_j$.

Si T_i lee el valor de un ítem A y a continuación T_j lo modifica entonces existe un arco $T_i \rightarrow T_j$.

Si el grafo de precedencia no tiene ciclos, entonces el esquema es serializable.

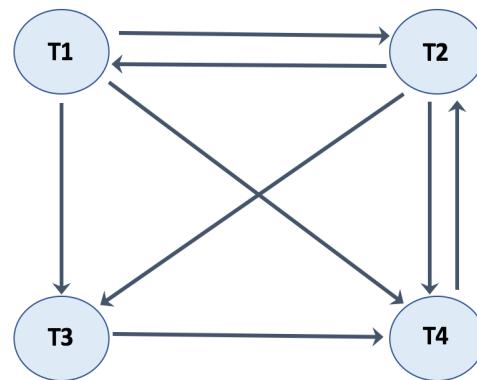
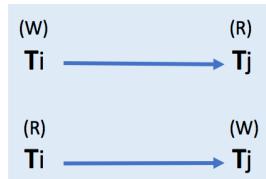


EJEMPLO

Para el esquema de ejecución concurrente de la figura copiada a continuación se pide dibujar el grafo *read antes que writes* para determinar si la ejecución es serializable.

T1	T2	T3	T4
read(A)			
		read(B)	
	read(C)		
		write(A)	
read(B)			
			read(B)
write(C)			
	read(B)		
		read(A)	
		write(C)	
			write(B)
	write(A)		

Solución



Si T_j lee el valor de un ítem A , que fue escrito por T_i , entonces se crea un arco $T_i \rightarrow T_j$.

Si T_i lee el valor de un ítem A y a continuación T_j lo modifica entonces existe un arco $T_i \rightarrow T_j$.

Dado el siguiente esquema de transacciones concurrentes, determine si el esquema es serializable utilizando los diagramas “*reads* antes de *writes*”. Además, si lo fuese indique un esquema en serie equivalente.

T1	T2	T3	T4
read(A)			
		read(B)	
	read(C)		
		write(C)	
read(B)			
			read(B)
write(C)			
	read(D)		
			read(D)
		write(C)	
			write(B)
	write(A)		

T1	T2	T3
read(A)		
write(B)		
	read(B)	
write(A)		
		write(A)
	read(A)	
read(A)		
		write(B)
	write(A)	
read(B)		
		read(A)

- Si T_j lee el valor de un ítem A , que fue escrito por T_i , entonces se crea un arco $T_i \rightarrow T_j$.
- Si T_i lee el valor de un ítem A y a continuación T_j lo modifica entonces existe un arco $T_i \rightarrow T_j$.

Protocolo de *timestamp*

El DBMS asigna a cada transacción T_i un valor único $T_s(i)$ que permita establecer un orden entre las transacciones.

A cada tupla Q siendo requerida por las transacciones se asocian dos valores:

- $W_{TS}(Q)$: Máximo de los timestamps de las transacciones que han hecho write(Q).
- $R_{TS}(Q)$: Máximo de los timestamps de las transacciones que han hecho read(Q).

El protocolo que asegura la correcta ejecución de las transacciones es el siguiente:

- Si T_i intenta ejecutar read(Q)
 - Si $T_s(i) < W_{TS}(Q)$ no se ejecuta la operación y T_i se aborta.
 - Si $T_s(i) > W_{TS}(Q)$ se ejecuta la operación y $R_{TS}(Q) = \max\{R_{TS}(Q), T_s(i)\}$.
- Si T_i intenta un write(Q)
 - Si $T_s(i) < R_{TS}(Q)$ no se ejecuta la operación y T_i se aborta.
 - Si $T_s(i) < W_{TS}(Q)$ no se ejecuta la operación y T_i continúa.
 - Si no, sí se ejecuta la operación y $W_{TS}(Q) = T_s(i)$.

EJEMPLO

Para el esquema de ejecución concurrente de la figura copiada a continuación se pide aplicar el protocolo de ejecución optimista de transacciones para determinar cuáles transacciones pueden finalizar con éxito bajo este protocolo de sincronización.

T1	T2	T3	T4
read(A)			
		read(B)	
		read(C)	
			write(A)
read(B)			
			read(B)
write(C)			
	read(B)		
			read(A)
		write(C)	
			write(B)
	write(A)		

Solución

WTS(A)	0	3		Abort T2
WTS(B)	0	4		
WTS(C)	0	3		Abort T1
RTS(A)	0	1	4	
RTS(B)	0	3	4	
RTS(C)	0	2		

- $W_{TS}(Q)$: Máximo de los timestamps de las transacciones que han hecho write(Q).
- $R_{TS}(Q)$: Máximo de los timestamps de las transacciones que han hecho read(Q).

El protocolo que asegura la correcta ejecución de las transacciones es el siguiente:

- Si T_i intenta ejecutar read(Q)
 - Si $T_s(i) < W_{TS}(Q)$ no se ejecuta la operación y T_i se aborta.
 - Si $T_s(i) > W_{TS}(Q)$ se ejecuta la operación y $R_{TS}(Q) = \max\{R_{TS}(Q), T_s(i)\}$.
- Si T_i intenta un write(Q)
 - Si $T_s(i) < R_{TS}(Q)$ no se ejecuta la operación y T_i se aborta.
 - Si $T_s(i) < W_{TS}(Q)$ no se ejecuta la operación y T_i continúa.
 - Si no, sí se ejecuta la operación y $W_{TS}(Q) = T_s(i)$.

El sistema debe procurar que el esquema resultante sea equivalente a un esquema serial en que las transacciones se ejecutan en el orden indicado por sus timestamps.

T1	T2	T3
write(A)		
write(B)		
	write(C)	
	read(A)	
		read(B)
		write(A)
read(C)		
write(C)		
	read(B)	
	write(B)	

TS(1)=1 TS(2)= 2 TS(3)= 3

WTS(Q)

A	0							
B	0							
C	0							

RTS(Q)

A	0							
B	0							
C	0							

Al abandonar una transacción se debe abortar (y hacer *rollback*) de todas las transacciones que dependen de ella, es decir, que hayan leído valores escritos por la primera. Esto es recursivo y se llama efecto cascada. El efecto cascada se puede evitar obligando a que las transacciones sólo puedan leer valores escritos por transacciones que hayan terminado, lo que introduce estados de espera.

T1	T2	T3
read(A)		
write(B)		
	read(B)	
write(A)		
		write(A)
	read(A)	
read(A)		
		write(B)
	write(A)	
read(B)		
		read(A)

TS(1)=1 TS(2)= 2 TS(3)= 3

WTS(Q)

A	0						
B	0						

RTS(Q)

A	0						
B	0						

- Si la transacción T_i intenta ejecutar un **read(Q)**
 - Si $T_s(i) < W_{TS}(Q)$ no se ejecuta la operación y T_i se re-ejecuta.
 - Si $T_s(i) > W_{TS}(Q)$ se ejecuta la operación y $R_{TS}(Q) = \max\{R_{TS}(Q), T_s(i)\}$.
- Si la transacción T_i intenta ejecutar un **write(Q)**
 - Si $T_s(i) < R_{TS}(Q)$ no se ejecuta la operación y T_i se re-ejecuta.
 - Si $T_s(i) < W_{TS}(Q)$ no se ejecuta la operación y T_i continúa.
 - Si no, sí se ejecuta la operación y $W_{TS}(Q) = T_s(i)$.

Protocolo de versiones múltiples

Se mantienen varias versiones para cada ítem. A cada transacción se asigna un timestamp igual que antes. Cada versión de un ítem contiene los siguientes campos:

- El valor del ítem.
- W_{TS} : El timestamp T_s de la transacción que creó la versión.
- R_{TS} : Máximo de los T_s de las transacciones que hayan leído la versión.

Supongamos que T_i quiere efectuar una operación sobre Q . Sea Q_k la versión de Q , tal que $W_{TS}(Q_k)$ es el mayor de los W_{TS} menores que $T_s(i)$. El protocolo es:

- Si T_i intenta un $\text{read}(Q)$, entonces se retorna el valor de Q_k y $R_{TS}(Q_s) = \max\{R_{TS}(Q_k), T_s(i)\}$.
- Si T_i intenta un $\text{write}(Q)$:
 - Si $T_s(i) < R_{TS}(Q_k)$, T_i se aborta.
 - Si no, se crea una nueva versión de Q con el valor escrito por T_i , con W_{TS} y R_{TS} iguales a $T_s(i)$.

Locks

Consideramos dos tipos de Locks:

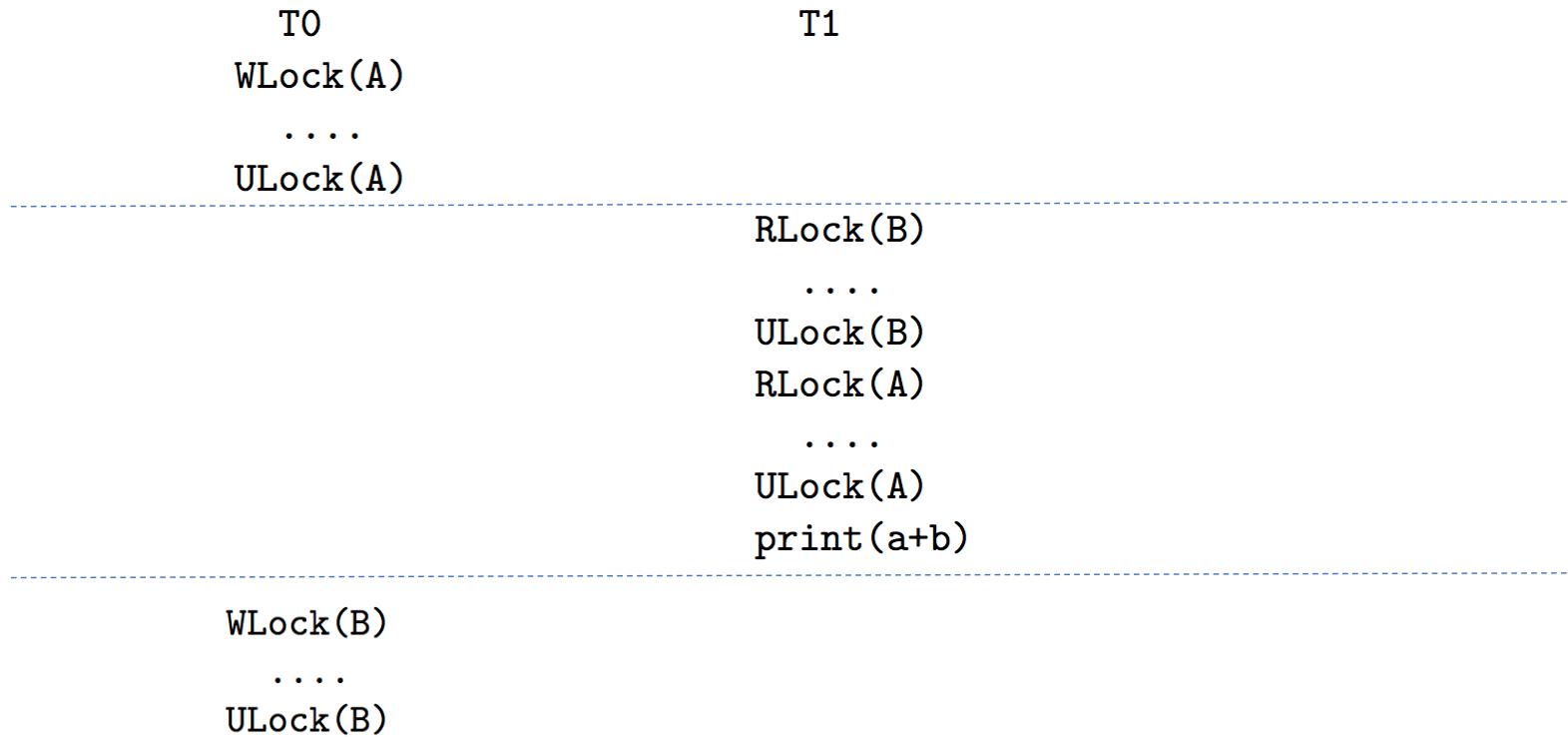
- LECTURA: Una transacción que tiene un lock de lectura para un ítem, puede leerlo pero no modificarlo. Ninguna otra Transacción puede modificar el valor del ítem mientras la transacción tenga lock de lectura.
- ESCRITURA: Si una transacción tiene un lock de escritura puede leer y escribir el ítem. Ninguna otra transacción puede tener un lock en el mismo ítem, al mismo tiempo.

Definamos las operaciones RLock(Q) , WLock(Q) y ULock(Q). Ejemplo:

T0: **WLock(A)**
 read(A,a)
 a = a - 50
 write(A,a)
 ULock(B)
 WLock(B)
 read(B,b)
 b = b + 50
 write(B,b)
 ULock(B)

T1: **RLock(B)**
 read(B,b)
 ULock(B)
 RLock(A)
 read(A,a)
 ULock(A)
 print(a+b)

Un esquema de ejecución concurrente podría ser:



Este esquema no es serializable ($T_0 \rightarrow T_1$, y $T_0 \leftarrow T_1$), es decir, esta ejecución no debería permitirse.

Protocolo de dos fases

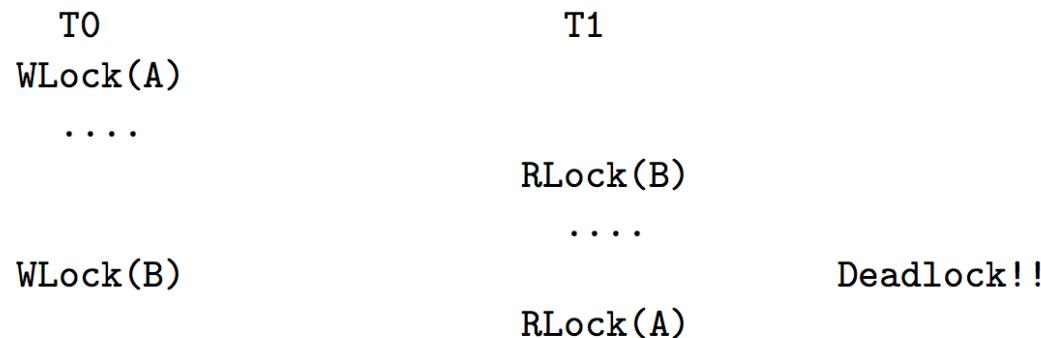
Este protocolo consiste en que todas las operaciones de Lock, deben preceder a las operaciones UnLock.

Podemos reescribir T0 como:

T0: `WLock(A)`
 `read(A,a)`
 `a = a - 50`
 `write(A,a)`
 `WLock(B)`
 `read(B,b)`
 `b = b + 50`
 `write(B,b)`
 `ULock(A)`
 `ULock(B)`

T1: `RLock(B)`
 `read(B,b)`
 `RLock(A)`
 `read(A,a)`
 `ULock(A)`
 `ULock(B)`
 `print(a+b)`

El siguiente esquema es válido:



El protocolo de dos fases, no evita los deadlocks.

Una forma de evitar los deadlocks, es imponer un orden a los ítems y exigir que las transacciones hagan los Locks respetando ese orden. Por ejemplo, si se van a usar *A* y *B*, hacen siempre Lock(*A*) antes que Lock(*B*).