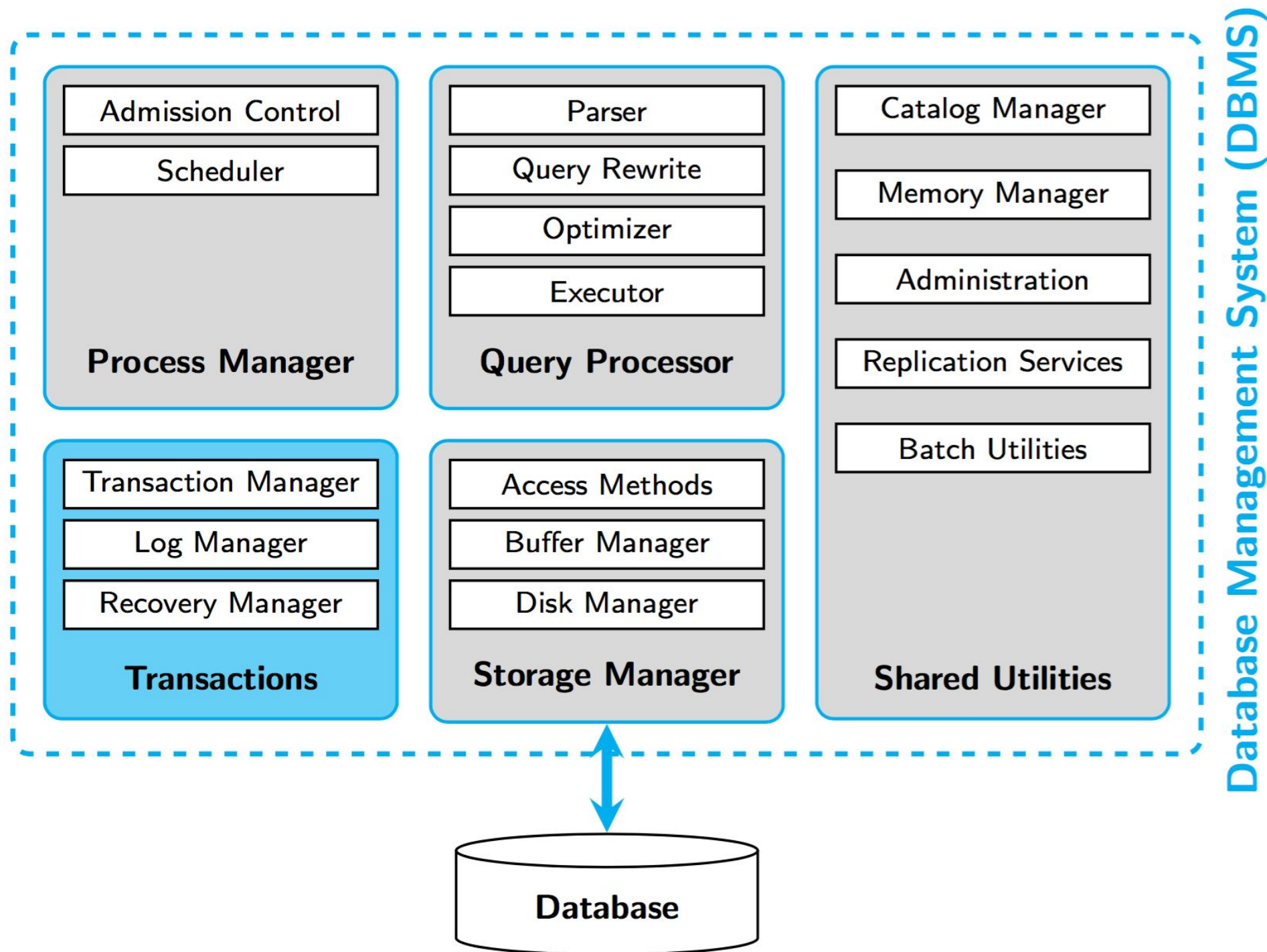


Bases de Datos

Clase 11: Transacciones y Recuperación de Fallas

Transactions



Transactions

Componente que asegura las propiedades **ACID**



Atomicity
Consistency
Isolation
Durability

Transactions

Transaction Manager se encarga de asegurar Isolation y Consistency

Log y Recovery Manager se encargan de asegurar Atomicity y Durability

Transactions

Supongamos las siguientes consultas (transferencia de dinero entre dos cuentas):

```
UPDATE cuentas  
SET saldo = saldo - v  
WHERE cid = 1
```

```
UPDATE cuentas  
SET saldo = saldo + v  
WHERE cid = 2
```

Transactions

¿Qué pasa cuando el acceso es concurrente?

Transactions

Transferencia doble

Supongamos que Alice y Bob están casados y tienen una cuenta común

Alice quiere transferirle 100 a su amigo Charles

Bob quiere transferirle 200 a su amigo Charles

Transactions

Transferencia doble

¿Qué puede salir mal?



Transactions

Transferencia doble

Proceso Alice	Proceso Bob	Saldo Cuenta A & B	Saldo Cuenta C
READ(saldoAB, x)		1000	1000
WRITE(saldoAB, x - 100)		900	
READ(saldoC, x)			
WRITE(saldoC, x + 100)			1100
	READ(saldoAB, y)		
	WRITE(saldoAB, y - 200)	700	
	READ(saldoC, y)		
	WRITE(saldoC, y + 200)	700	1300

Transactions

Transferencia doble

¿Qué puede salir mal?



Transactions

Transferencia doble

Proceso Alice	Proceso Bob	Saldo Cuenta A & B	Saldo Cuenta C
READ(saldoAB, x)		1000	1000
WRITE(saldoAB, x - 100)		900	
	READ(saldoAB, y)		
	WRITE(saldoAB, y - 200)	700	
	READ(saldoC, y)		
	WRITE(saldoC, y + 200)		1200
READ(saldoC, x)			
WRITE(saldoC, x + 100)		700	1300

Transactions

Transferencia doble

¿Qué puede salir mal?



Transactions

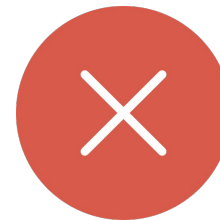
Transferencia doble

Proceso Alice	Proceso Bob	Saldo Cuenta A & B	Saldo Cuenta C
READ(saldoAB, x)		1000	1000
WRITE(saldoAB, x - 100)		900	
READ(saldoC, x)			
	READ(saldoAB, y)		
	WRITE(saldoAB, y - 200)	700	
	READ(saldoC, y)		1000
	WRITE(saldoC, y + 200)		1200
WRITE(saldoC, x + 100)		700	1100

Transactions

Transferencia doble

¿Qué puede salir mal?



Transactions

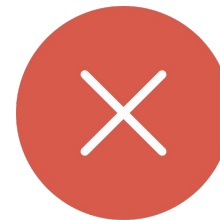
Transferencia doble

Proceso Alice	Proceso Bob	Saldo Cuenta A & B	Saldo Cuenta C
READ(saldoAB, x)		1000	1000
WRITE(saldoAB, x - 100)		900	
	ERROR	900	1000

Transactions

Transferencia doble

¿Qué puede salir mal?



¿Qué está pasando?

Mezclamos las operaciones a realizar
(en cada depósito)

El Ideal: cada depósito se ejecuta en el orden que fue solicitado

Lo real: Para optimizar accesos a disco, nos conviene mezclar operaciones.

¿Cómo hacerlo sin dejar la escoba?

Asegurar las propiedades **ACID**

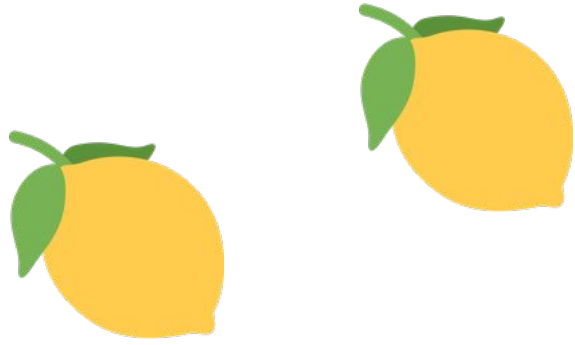


Atomicity
Consistency
Isolation
Durability



Atomicity:

O se ejecutan todas las operaciones de la transacción, o no se ejecuta ninguna.



Consistency:

Cada transacción preserva la consistencia de la BD
(restricciones de integridad, etc.).



Isolation:

Cada transacción debe ejecutarse como si se estuviese ejecutando sola, de forma aislada.



Durability:

Los cambios que hace cada transacción son permanentes en el tiempo, independiente de cualquier tipo de falla.

Sin ACID

Sin Atomicity, Durability:

Se corta la luz y la transacción quedó en la mitad

Se corta la luz cuando la transacción estaba en la mitad. La base de dato vuelve a su estado pero perdemos la transacción.

Un cambio hecho en la transacción no se ve reflejado en la BD.

Sin ACID

Sin Consistency:

La base de datos viola las restricciones momentáneamente

Al ejecutar una transacción, queda la BD que no cumple con las restricciones

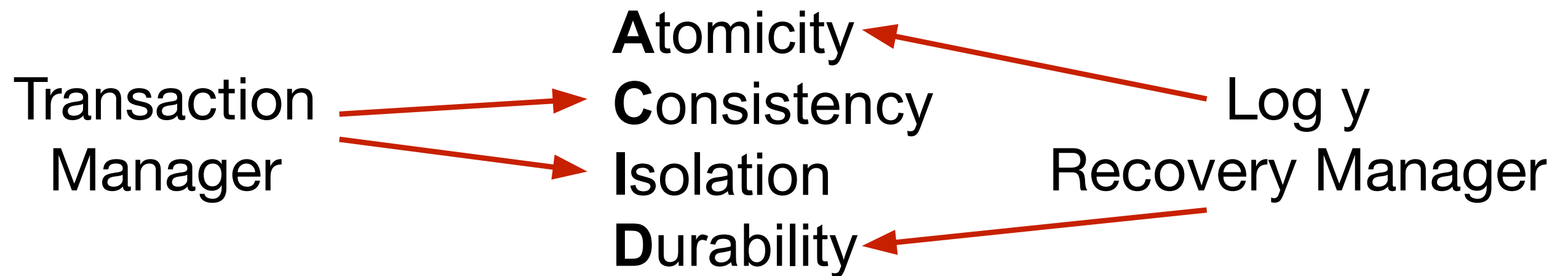
Sin ACID

Sin Isolation:

El sistema de base de datos planifica el orden de operaciones

Resultado no es igual a haber corrido transacciones en serie

Asegurar las propiedades **ACID**



Transacciones

Necesitamos transacciones

Una **transacción** es una secuencia de 1 o más operaciones que modifican o consultan la base de datos

- Transferencias de dinero entre cuentas
- Compra por internet
- Registrar un curso
- ...

Transacciones en SQL

START TRANSACTION

UPDATE cuentas
SET saldo = saldo - v
WHERE cid = 1

UPDATE cuentas
SET saldo = saldo + v
WHERE cid = 2

COMMIT

Transacciones en SQL

START TRANSACTION y **COMMIT** nos permiten agrupar operaciones en una sola transacción

Sobre transacciones

- Uno de los componentes fundamentales de una DBMS
- Fundamental para aplicaciones que requieren seguridad
- Uno de los **Turing Award** en Bases de Datos

Operaciones

$R(X)$

Lectura del
elemento X

$W(X)$

Escritura del
elemento X

A

Abortar
transacción

C

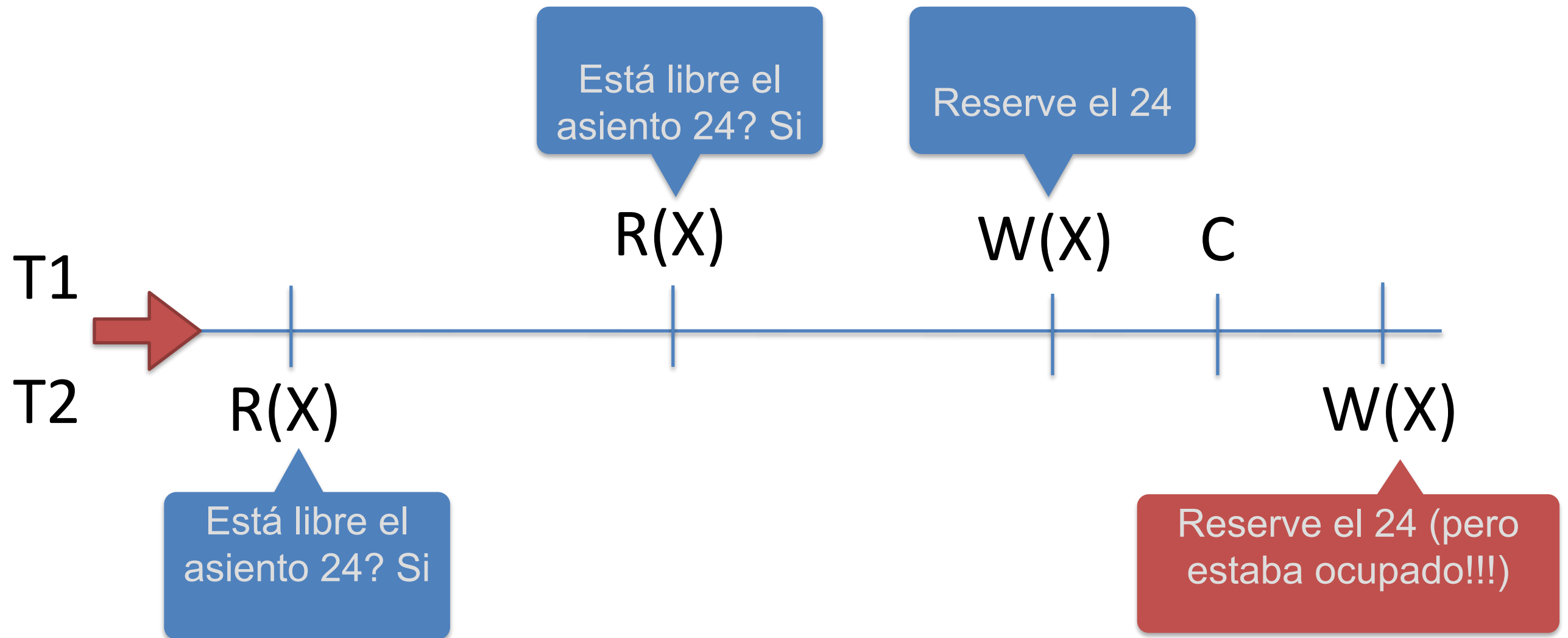
Finalizar
transacción

Conflictos con Transacciones

- Lecturas sucias (Write - Read)
- Lecturas no repetibles (Read - Write)
- Actualización perdida o reescritura de datos temporales (Write - Write)

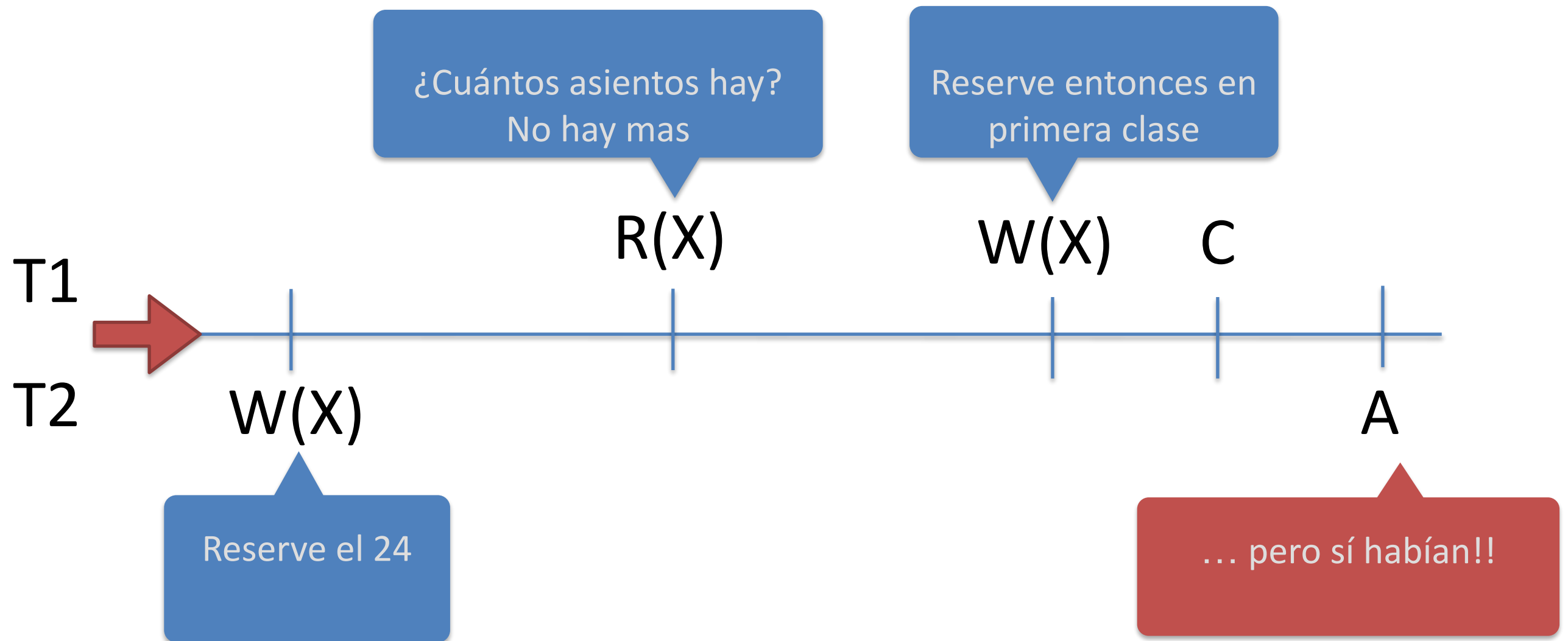
Ejemplo de actualización perdida (WW)

“Una transacción sobrescribe los datos que otra tx ya había escrito”



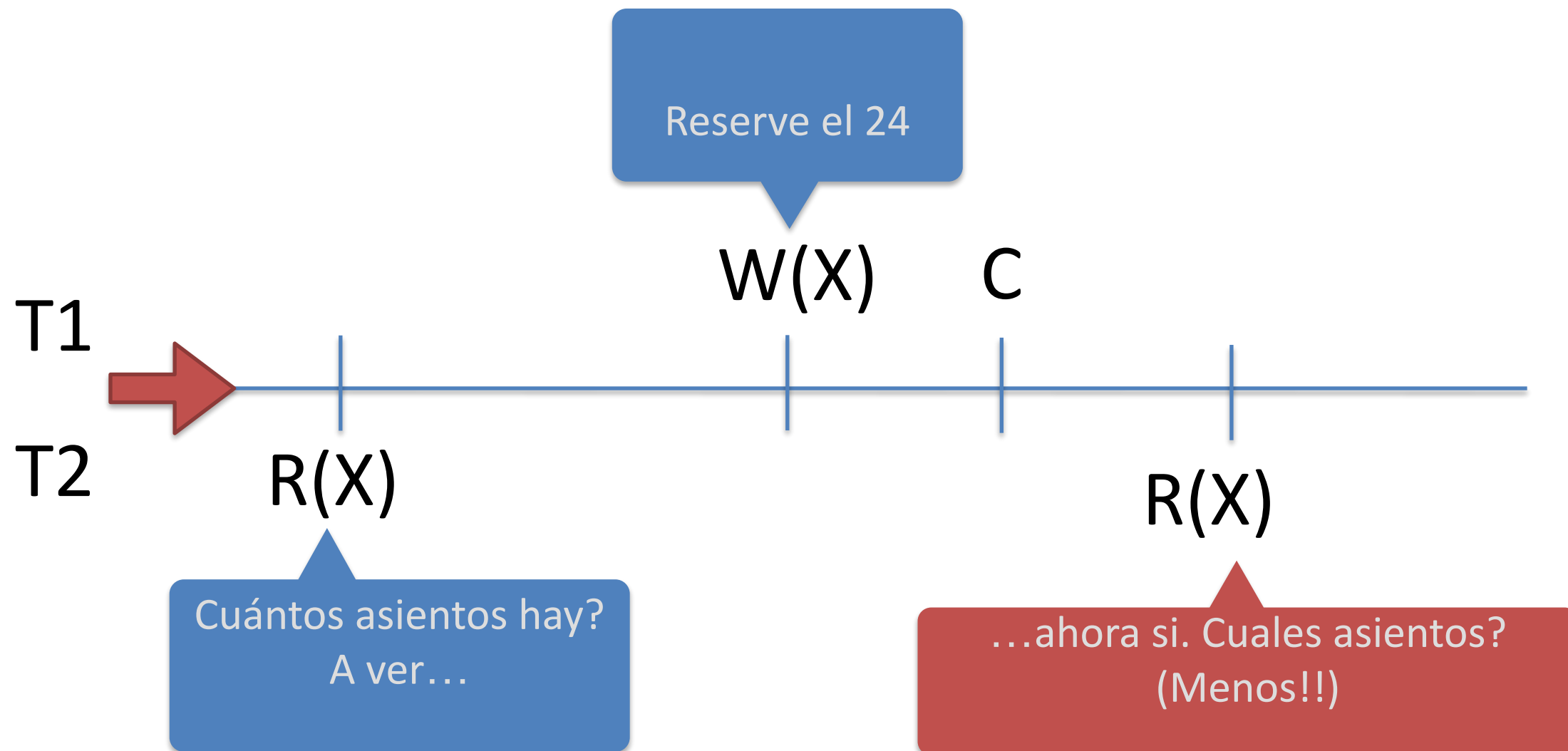
Ejemplo de lectura sucia (WR)

“Una tx. lee lo que otra tx escribió pero no se había confirmado aún.”



Ejemplo de lectura no repetible (RW)

“Una tx. sobrescribe un dato que otra ya había leído antes pero no había confirmado.”



Schedule

Un **schedule S** es una secuencia de operaciones primitivas de una o más transacciones, tal que para toda transacción, las acciones de ella aparecen en el mismo orden que en su definición

Schedule

Transacciones de un schedule

T1	T2
READ(A,x)	READ(A,y)
x:= x + 100	y:= y * 2
WRITE(A,x)	WRITE(A,y)
READ(B,x)	READ(B,y)
x:= x + 200	y:= y * 3
WRITE(B,x)	WRITE(B,y)

Schedule

Un schedule

T1	T2
READ(A,x)	
x:= x + 100	
WRITE(A,x)	
READ(B,x)	
x:= x + 200	
WRITE(B,x)	
	READ(A,y)
	y:= y * 2
	WRITE(A,y)
	READ(B,y)
	y:= y * 3
	WRITE(B,y)

Schedule

Otro schedule

T1	T2
READ(A,x)	
x:= x + 100	
WRITE(A,x)	
	READ(A,y)
	y:= y * 2
	WRITE(A,y)
READ(B,x)	
x:= x + 200	
WRITE(B,x)	
	READ(B,y)
	y:= y * 3
	WRITE(B,y)

Schedule Serial

Un **schedule S** es **serial** si no hay intercalación entre las acciones

Schedule Serial

Un schedule serial

T1	T2
READ(A,x)	
x:= x + 100	
WRITE(A,x)	
READ(B,x)	
x:= x + 200	
WRITE(B,x)	
	READ(A,y)
	y:= y * 2
	WRITE(A,y)
	READ(B,y)
	y:= y * 3
	WRITE(B,y)

Schedule Serializable

Un **schedule S** es **serializable** si existe algún **schedule S'** serial con las mismas transacciones, tal que el resultado de **S** y **S'** es el mismo para todo estado inicial de la BD

Schedule Serializable

T1	T2
READ(A,x)	
x:= x + 100	
WRITE(A,x)	
	READ(A,y)
	y:= y * 2
	WRITE(A,y)
READ(B,x)	
x:= x + 200	
WRITE(B,x)	
	READ(B,y)
	y:= y * 3
	WRITE(B,y)

Schedule No Serializable

T1	T2
READ(A,x)	
x:= x + 100	
WRITE(A,x)	
	READ(A,y)
	y:= y * 2
	WRITE(A,y)
	READ(B,y)
	y:= y * 3
	WRITE(B,y)
READ(B,x)	
x:= x + 200	
WRITE(B,x)	

Transactions

La tarea del Transaction Manager es permitir solo schedules que sean **serializables**

¿Cómo determinamos de manera rápida si un schedule es serializable?

Posibles problemas

Lo que queremos

T1	T2
READ(A,x)	
x:= x + 100	
WRITE(A,x)	
	READ(A,y)
	y:= y + 100
	WRITE(A,y)

Posibles problemas

Lo que el sistema quiere

T1	T2
READ(A,x)	
	READ(A,y)
x:= x + 100	
	y:= y + 100
WRITE(A,x)	
	WRITE(A,y)

Transactions

Notación

Si la transacción i ejecuta $READ(X,t)$ escribimos $R_i(X)$

Si la transacción i ejecuta $WRITE(X,t)$ escribimos $W_i(X)$

Acciones No Conflictivas

Las siguientes acciones son NO conflictivas para dos transacciones distintas i, j :

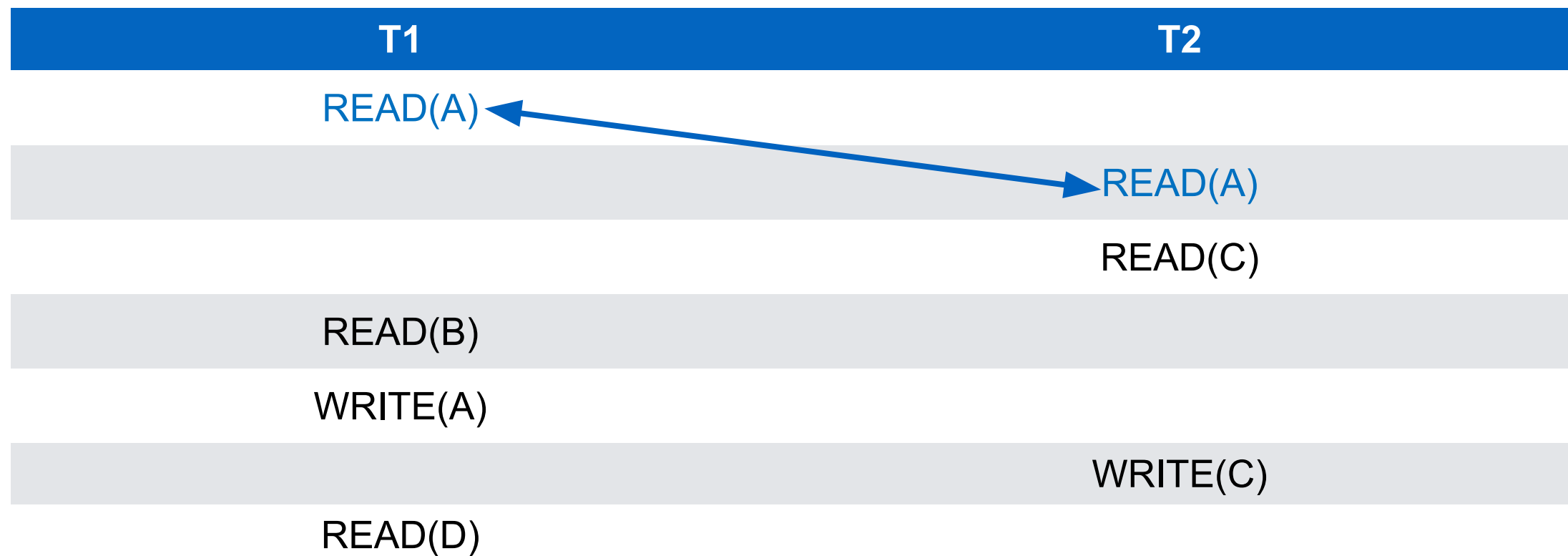
- $R_i(X), R_j(Y)$
- $R_i(X), W_j(Y)$ con $X \neq Y$
- $W_i(X), R_j(Y)$ con $X \neq Y$
- $W_i(X), W_j(Y)$ con $X \neq Y$

Podemos cambiarlas de orden en un **schedule**!

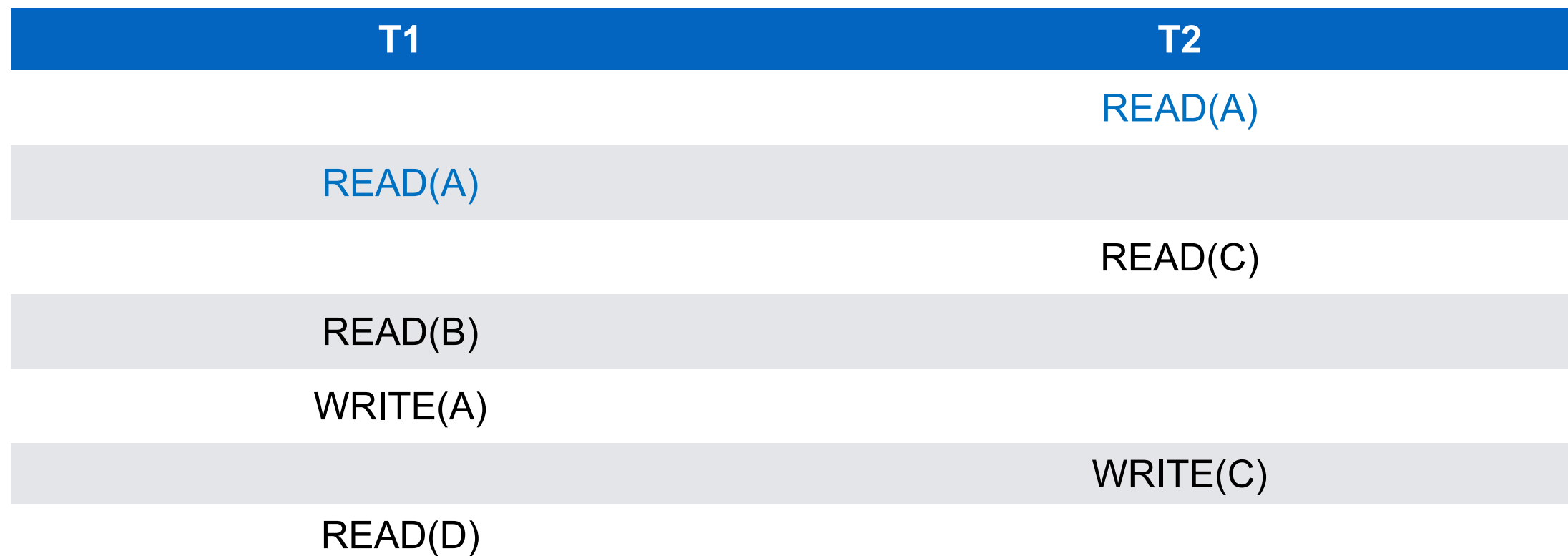
Permutaciones permitidas

T1	T2
READ(A)	
	READ(A)
	READ(C)
READ(B)	
WRITE(A)	
	WRITE(C)
READ(D)	

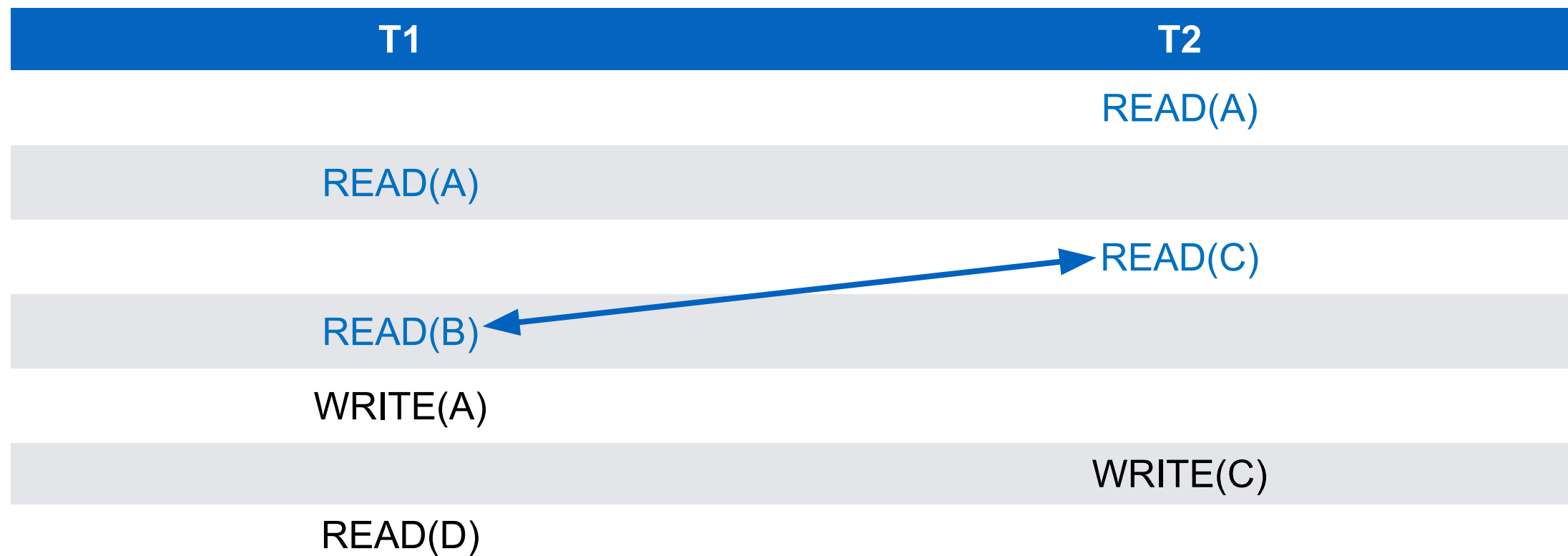
Permutaciones permitidas



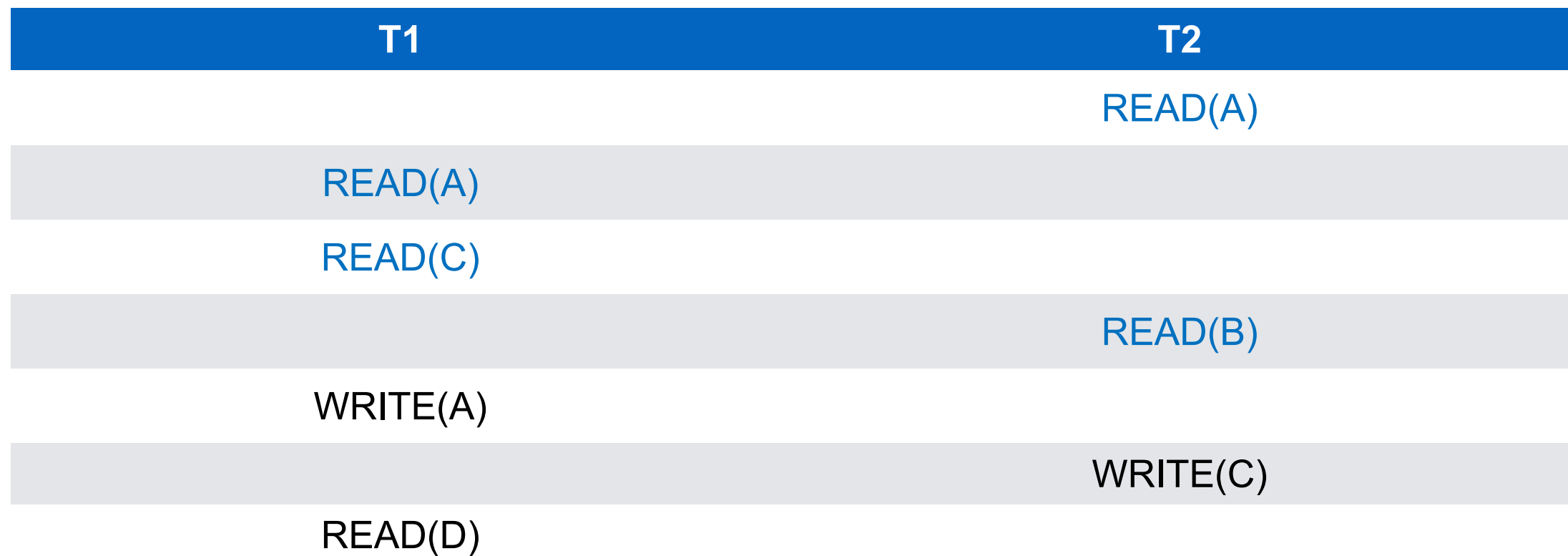
Permutaciones permitidas



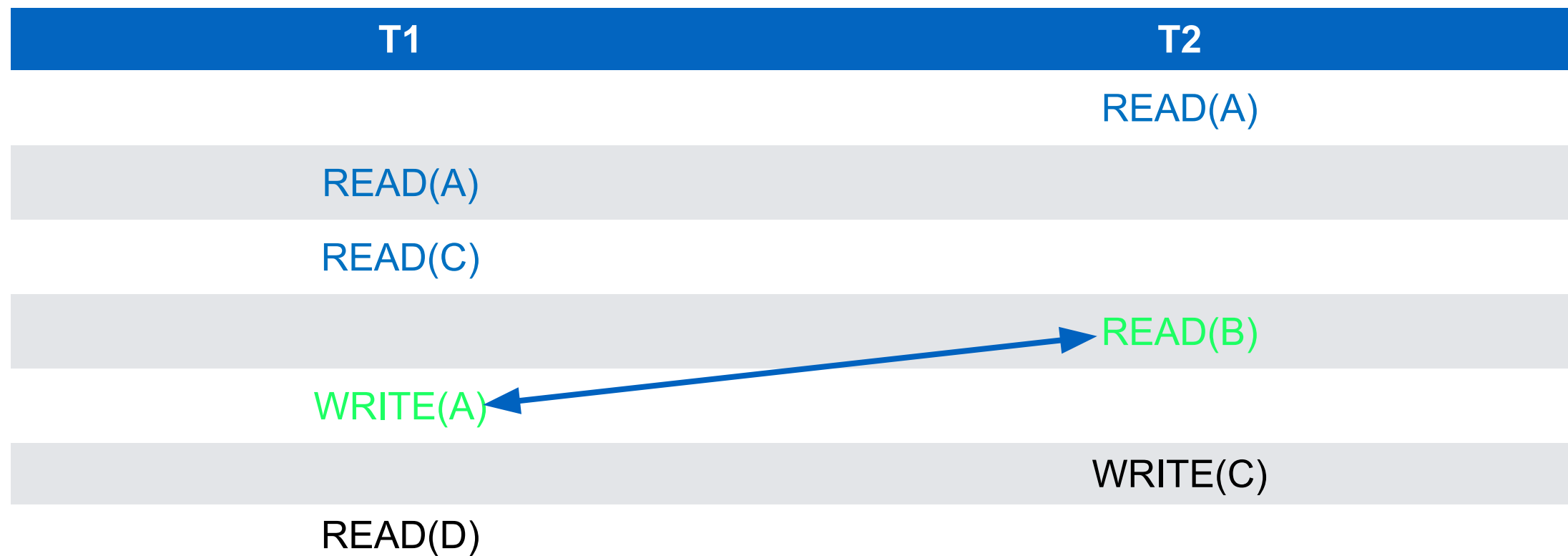
Permutaciones permitidas



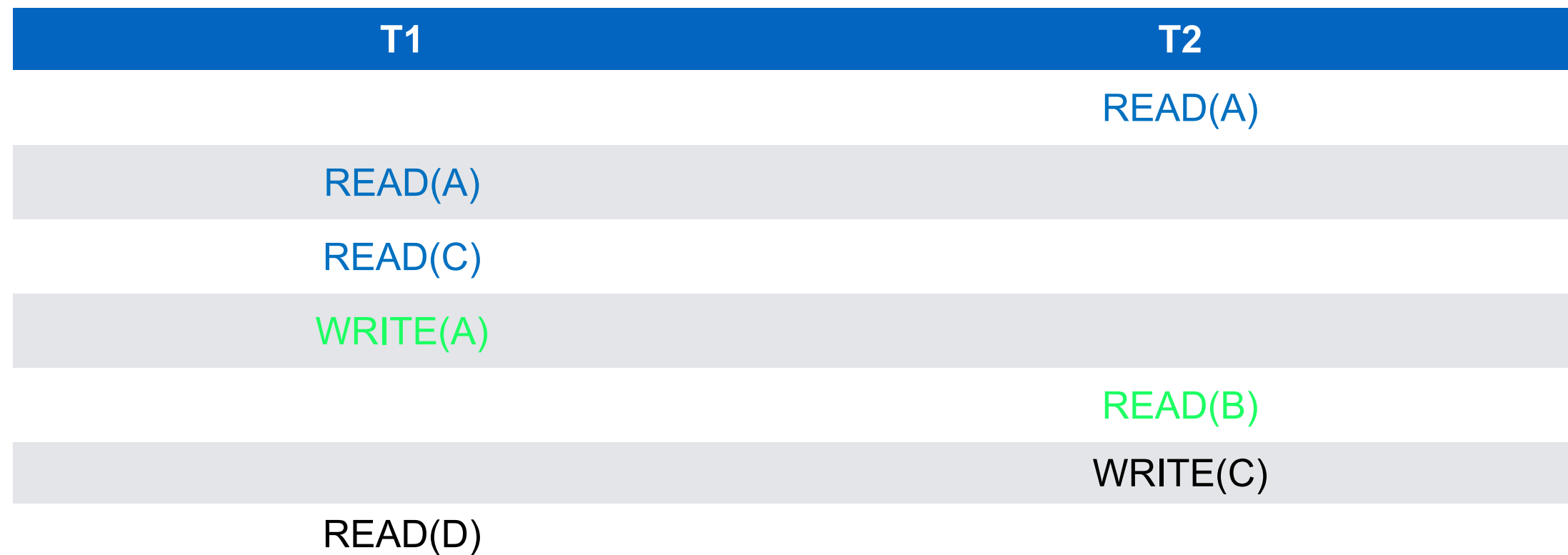
Permutaciones permitidas



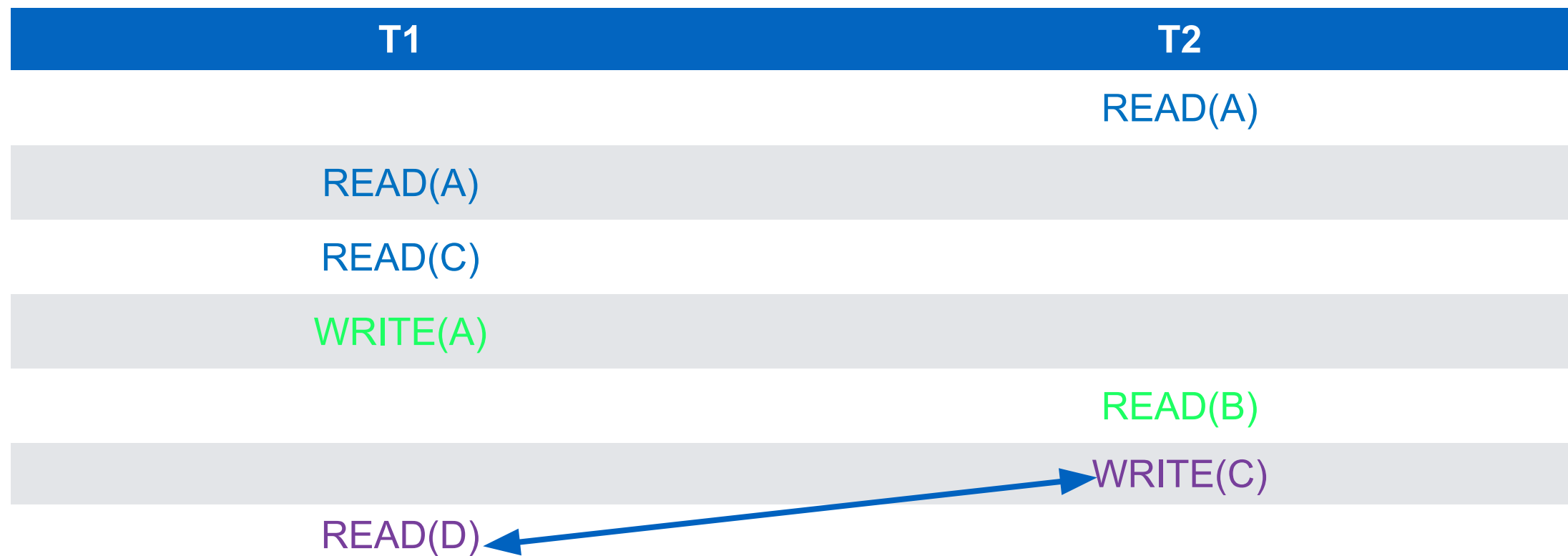
Permutaciones permitidas



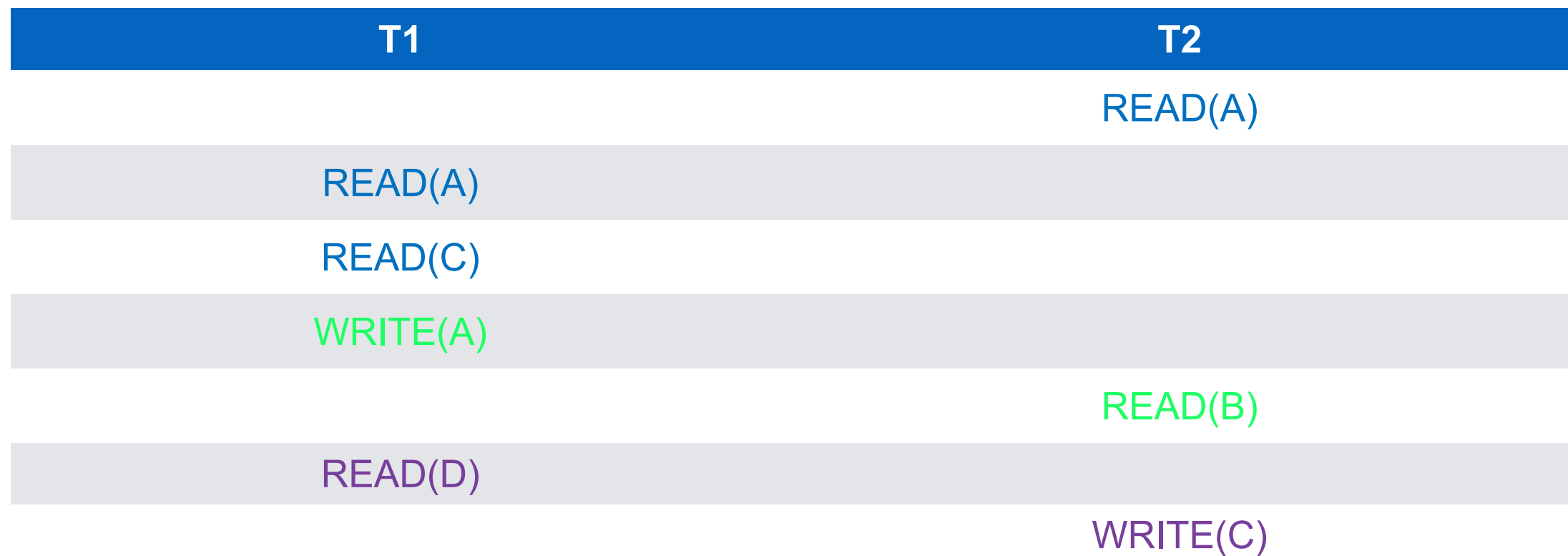
Permutaciones permitidas



Permutaciones permitidas



Permutaciones permitidas



Permutaciones permitidas

T1	T2
READ(A)	
	READ(A)
	READ(C)
READ(B)	
WRITE(A)	
	WRITE(C)
READ(D)	

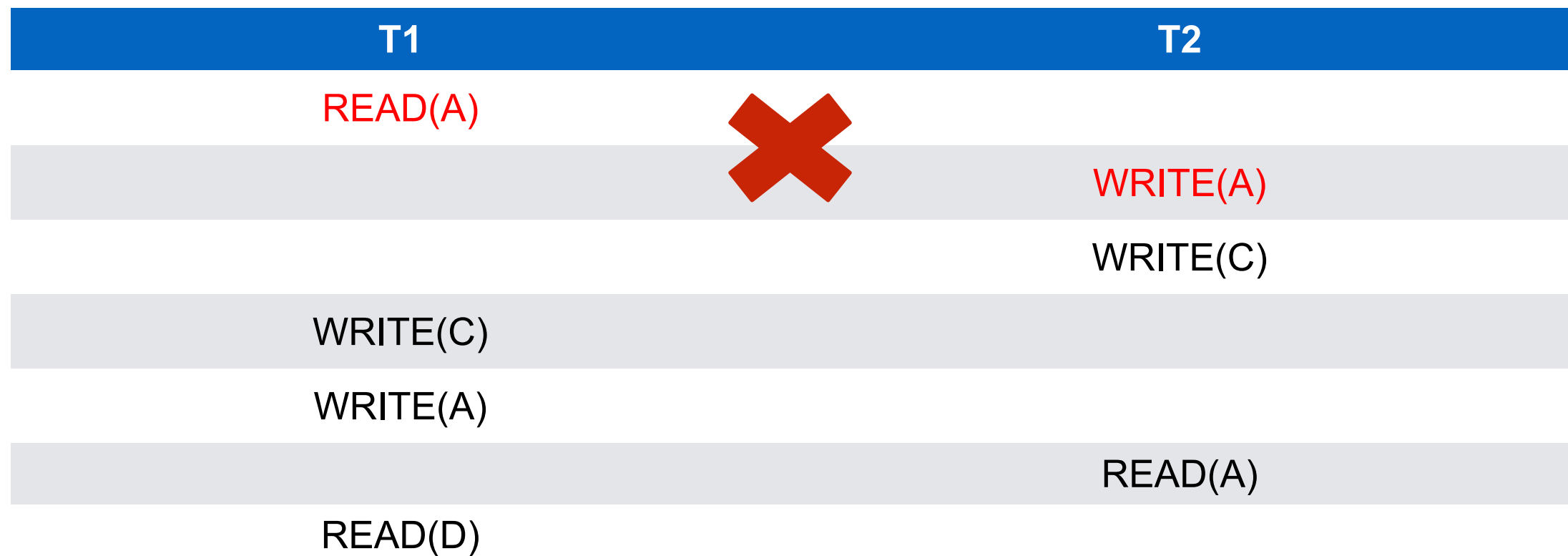
Acciones Conflictivas

Las siguientes acciones son conflictivas para dos transacciones distintas i, j :

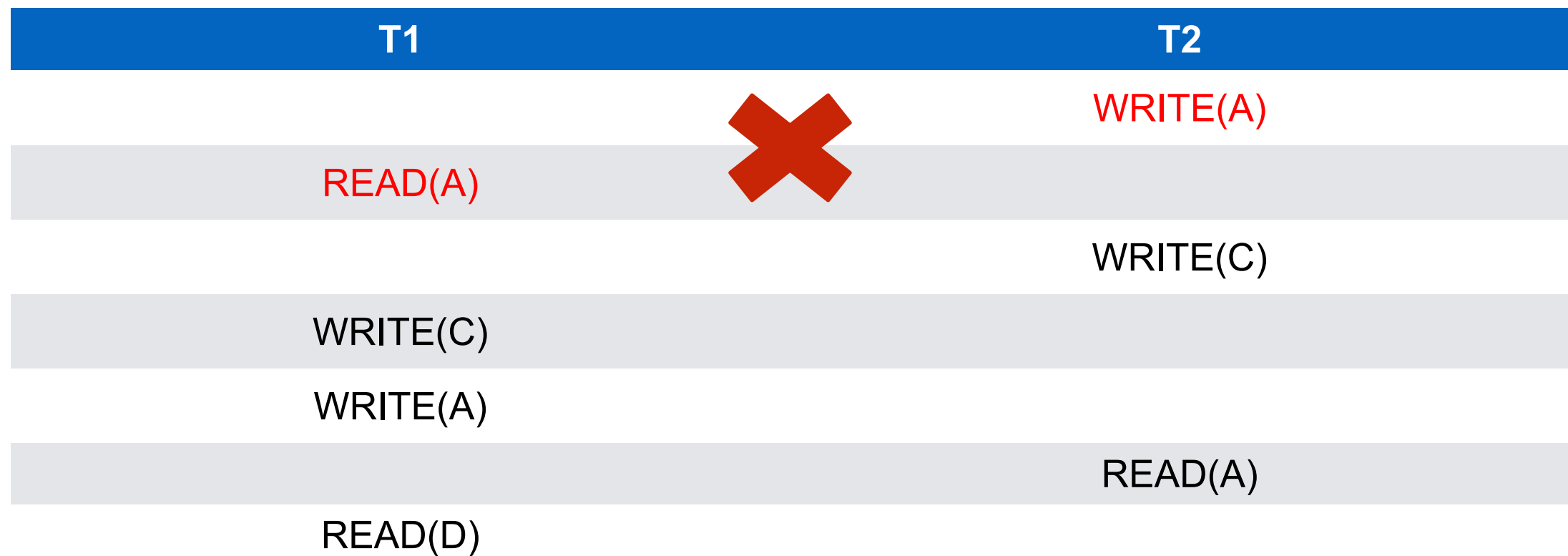
- $P_i(X), Q_i(Y)$ con P, Q en $\{R, W\}$
- $R_i(X), W_j(X)$
- $W_i(X), R_j(X)$
- $W_i(X), W_j(X)$

No podemos cambiar su orden en un **schedule** a la ligera!

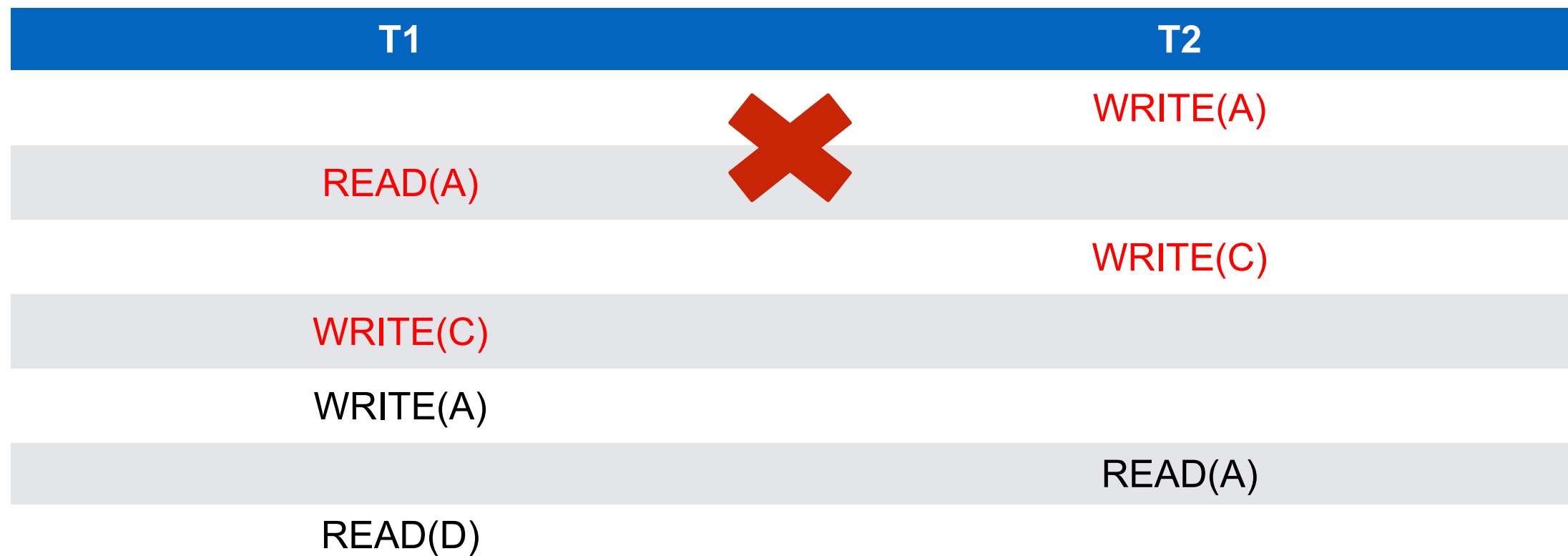
Permutaciones no permitidas



Permutaciones no permitidas

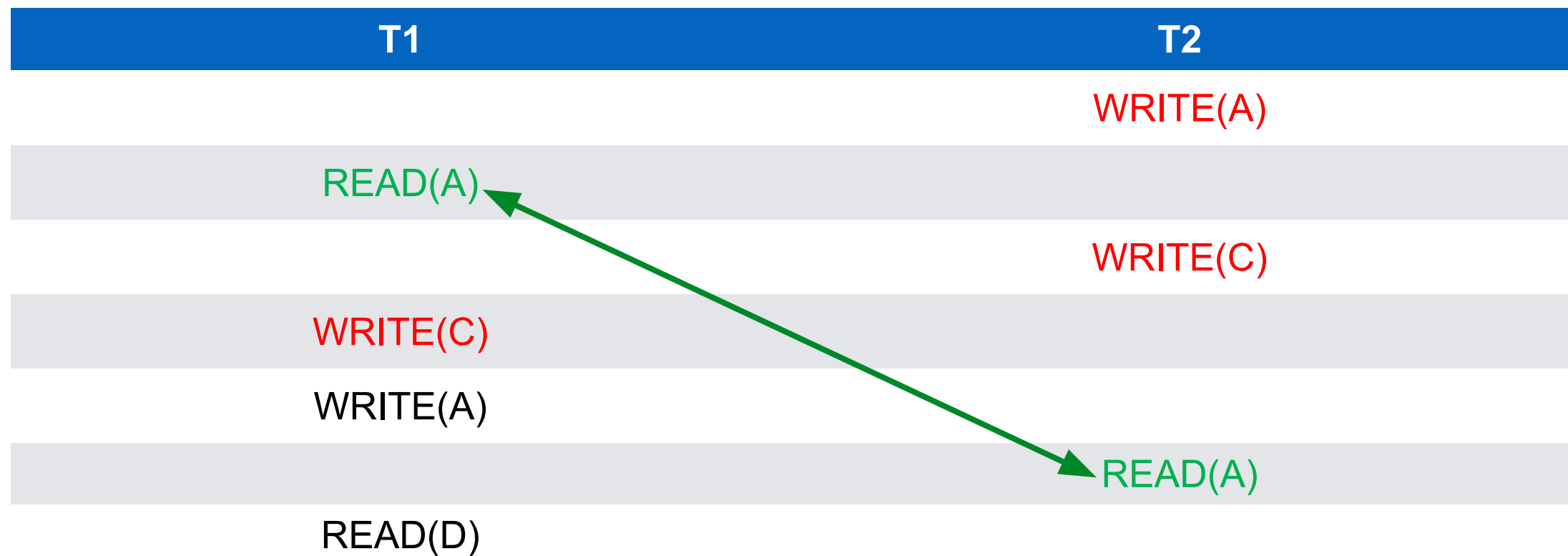


Permutaciones no permitidas



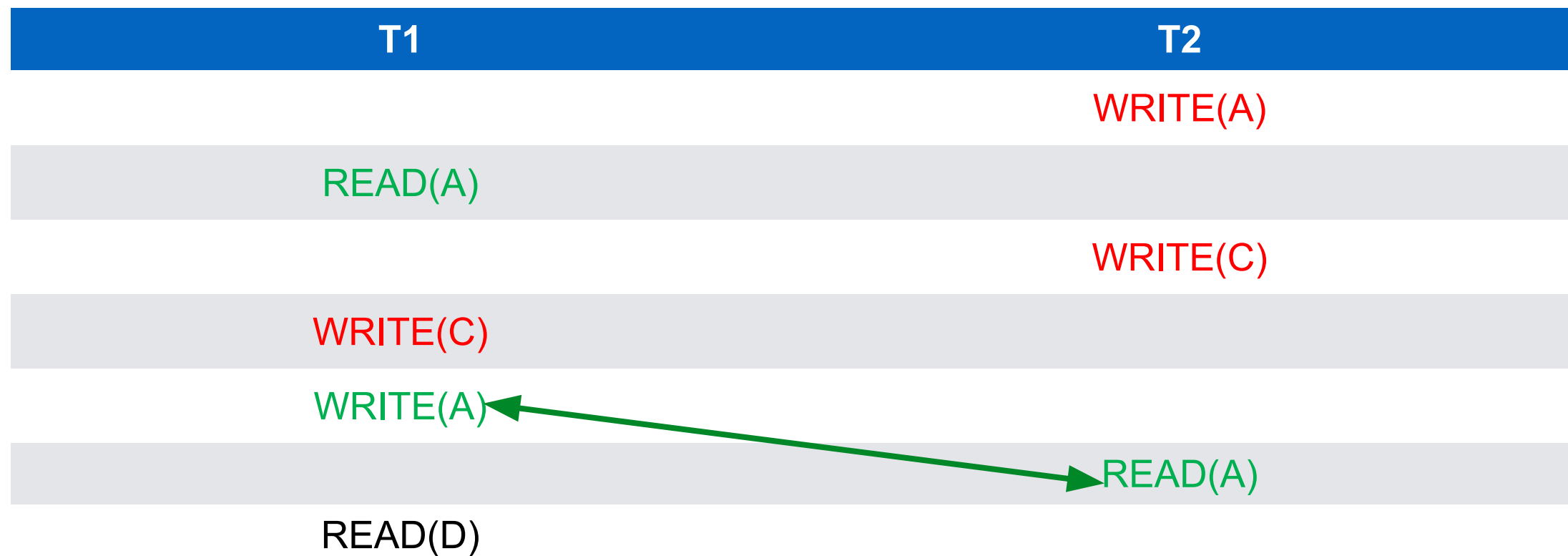
Permutaciones no permitidas

Cuidado!!!



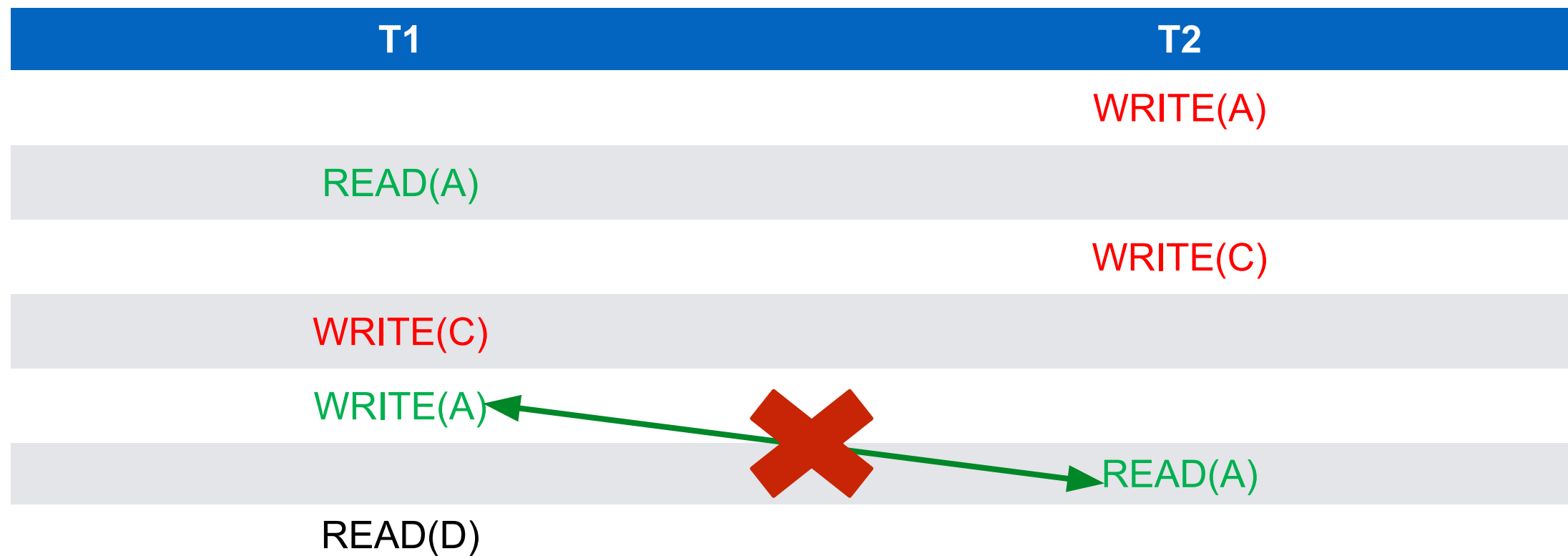
Permutaciones no permitidas

Cuidado!!!



Permutaciones no permitidas

Cuidado!!!



Acciones Conflictivas

Puedo permutar un par de operaciones consecutivas si:

- No usan el mismo recurso
- Usan el mismo recurso pero ambas son de lectura

Un **schedule** es *conflict serializable* si puedo transformarlo a uno **serial** usando permutaciones.

Conflict serializable

Si un **schedule** es *conflict serializable* implica que también es serializable, pero hay schedules serializables que no son *conflict serializable*

Conflict serializable

Con este proceso de permutaciones:

- Llevamos nuestro schedule a uno serial
- Preservamos el orden de **todos** los conflictos

Permutando a serial

Ejemplo

¿Es serializable?

T1	T2	T3
	R2(A)	
R1(B)		
	W2(A)	
		R3(A)
W1(B)		
		W3(A)
	R2(B)	
	W2(B)	

Permutando a serial

Ejemplo

Permutemos!

T1	T2	T3
	R2(A)	
R1(B)		
	W2(A)	
		R3(A)
W1(B)		
		W3(A)
	R2(B)	
	W2(B)	

Permutando a serial

Ejemplo

Permutemos!

T1	T2	T3
R1(B)		
	R2(A)	
	W2(A)	
		R3(A)
W1(B)		
		W3(A)
	R2(B)	
	W2(B)	

Permutando a serial

Ejemplo

Permutemos!

T1	T2	T3
R1(B)		
	R2(A)	
	W2(A)	
W1(B)		
		R3(A)
		W3(A)
	R2(B)	
	W2(B)	

Permutando a serial

Ejemplo

Permutemos!

T1	T2	T3
R1(B)		
	R2(A)	
	W2(A)	
W1(B)		
		R3(A)
		W3(A)
	R2(B)	
	W2(B)	

Permutando a serial

Ejemplo

Permutemos!

T1	T2	T3
R1(B)		
	R2(A)	
W1(B)		
	W2(A)	
		R3(A)
		W3(A)
	R2(B)	
	W2(B)	

Permutando a serial

Ejemplo

Permutemos!

T1	T2	T3
R1(B)		
	R2(A)	
W1(B)	W2(A)	
		R3(A)
		W3(A)
	R2(B)	
	W2(B)	

Permutando a serial

Ejemplo

Permutemos ...

T1	T2	T3
R1(B)		
W1(B)		
	R2(A)	
	W2(A)	
		R3(A)
		W3(A)
	R2(B)	
	W2(B)	

Permutando a serial

Ejemplo

T1;T2;T3 conflicto serializable serializable

T1	T2	T3
R1(B)		
W1(B)		
	R2(A)	
	W2(A)	
	R2(B)	
	W2(B)	
		R3(A)
		W3(A)

Grafo de precedencia

Teorema: Este proceso es exponencial ($n!$, n número de operaciones en todas las transacciones).

Además, determinar si un *schedule* es serializable (no necesariamente permutando) es **NP-Completo!**

Grafo de precedencia

Dado un **schedule** puedo construir su grafo de precedencia

- Nodos: **transacciones** del sistema
- Aristas: hay una arista de **T** a **T'** si **T** ejecuta una operación op1 antes de una operación op2 de **T'**, tal que op1 y op2 no se pueden permutar

Grafo de precedencia

Ejemplo (Pizarra)

¿Es *conflict serializable*?

T1	T2	T3
	R2(A)	
R1(B)		
	W2(A)	
	R2(B)	
		R3(A)
W1(B)		
		W3(A)
	W2(B)	

T1

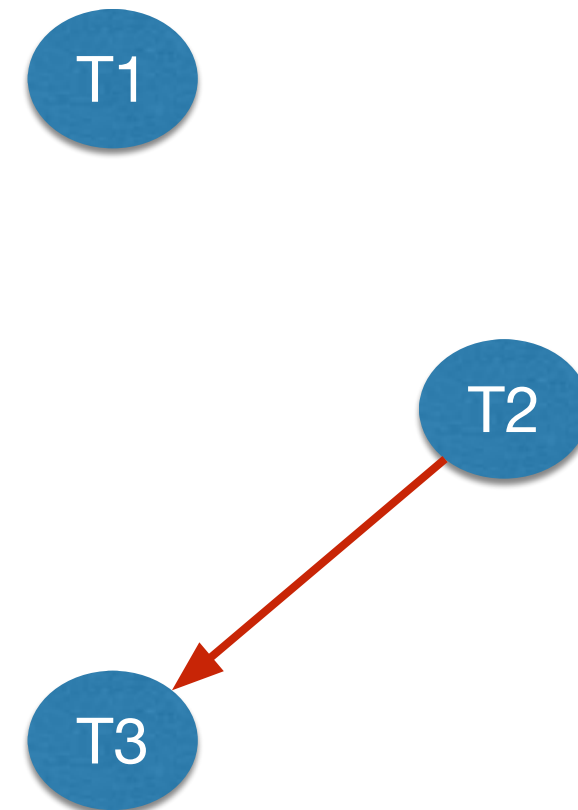
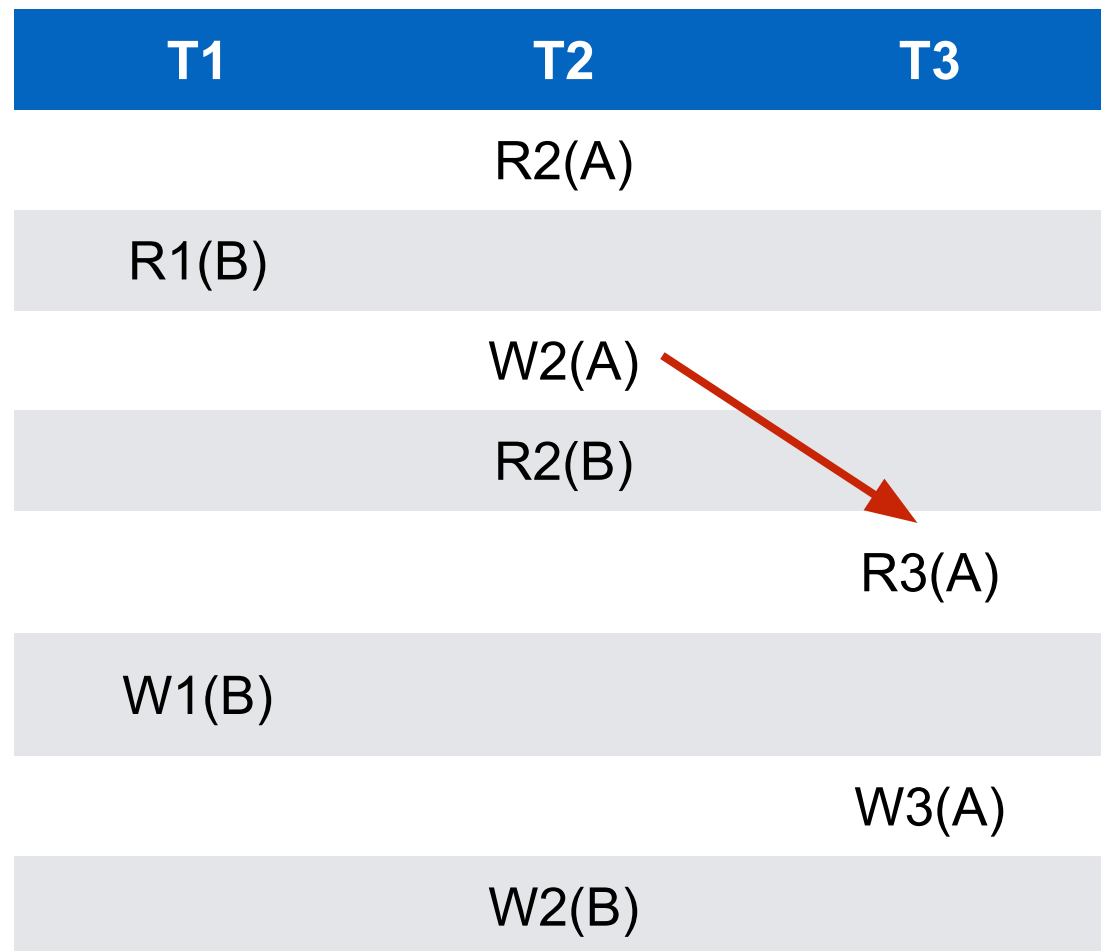
T2

T3

Grafo de precedencia

Ejemplo (Pizarra)

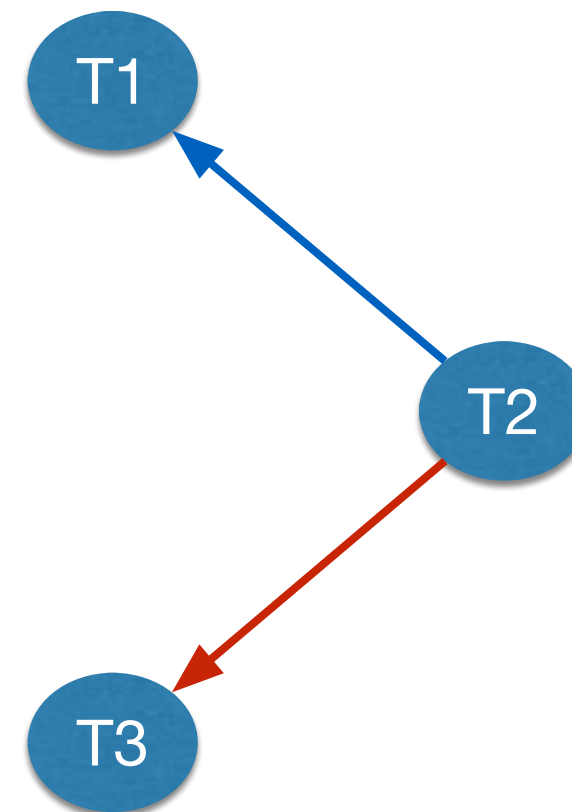
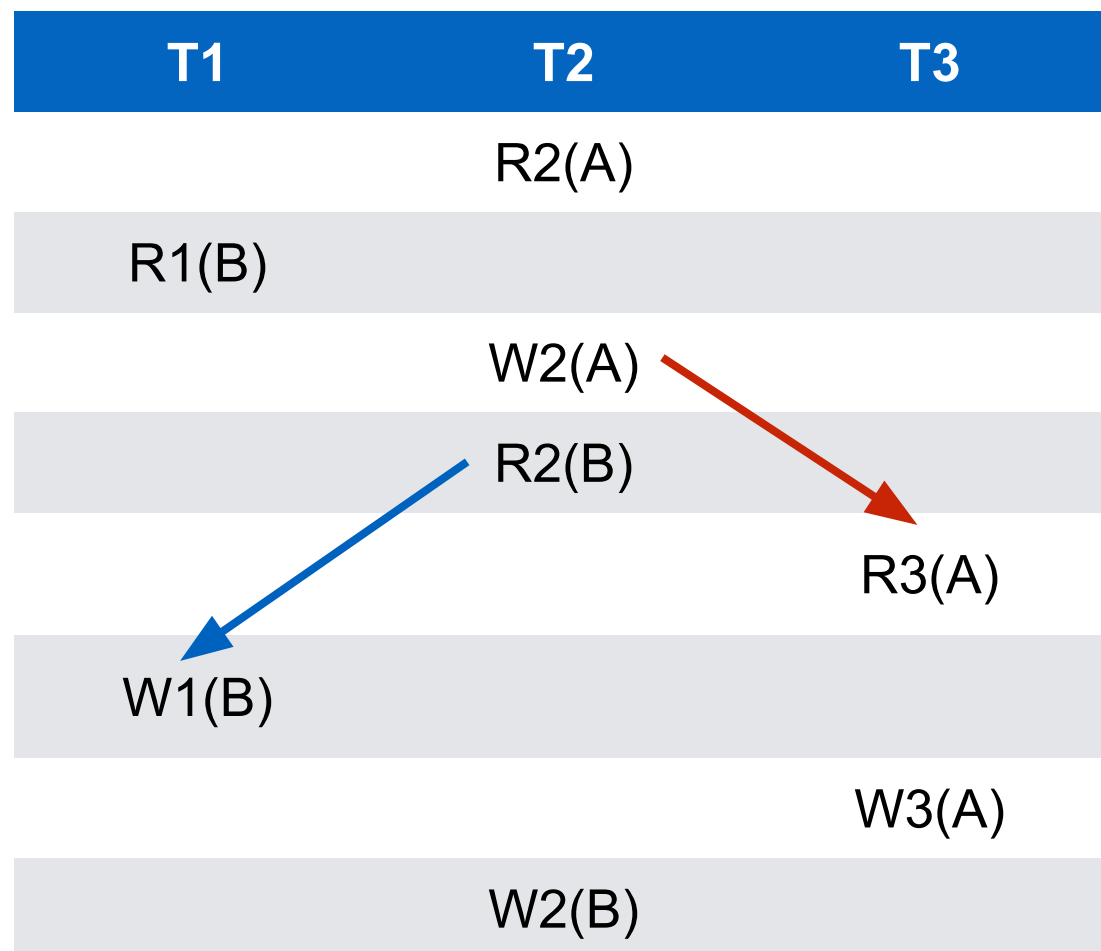
¿Es *conflict serializable*?



Grafo de precedencia

Ejemplo (Pizarra)

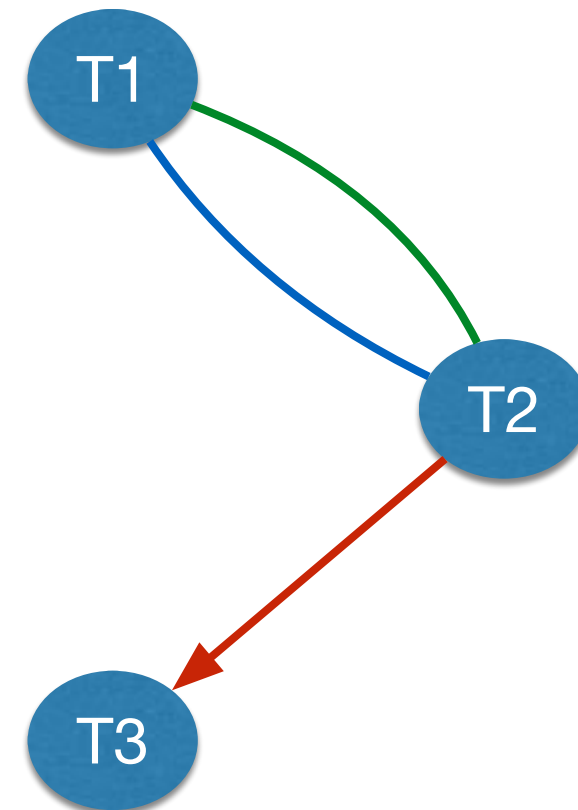
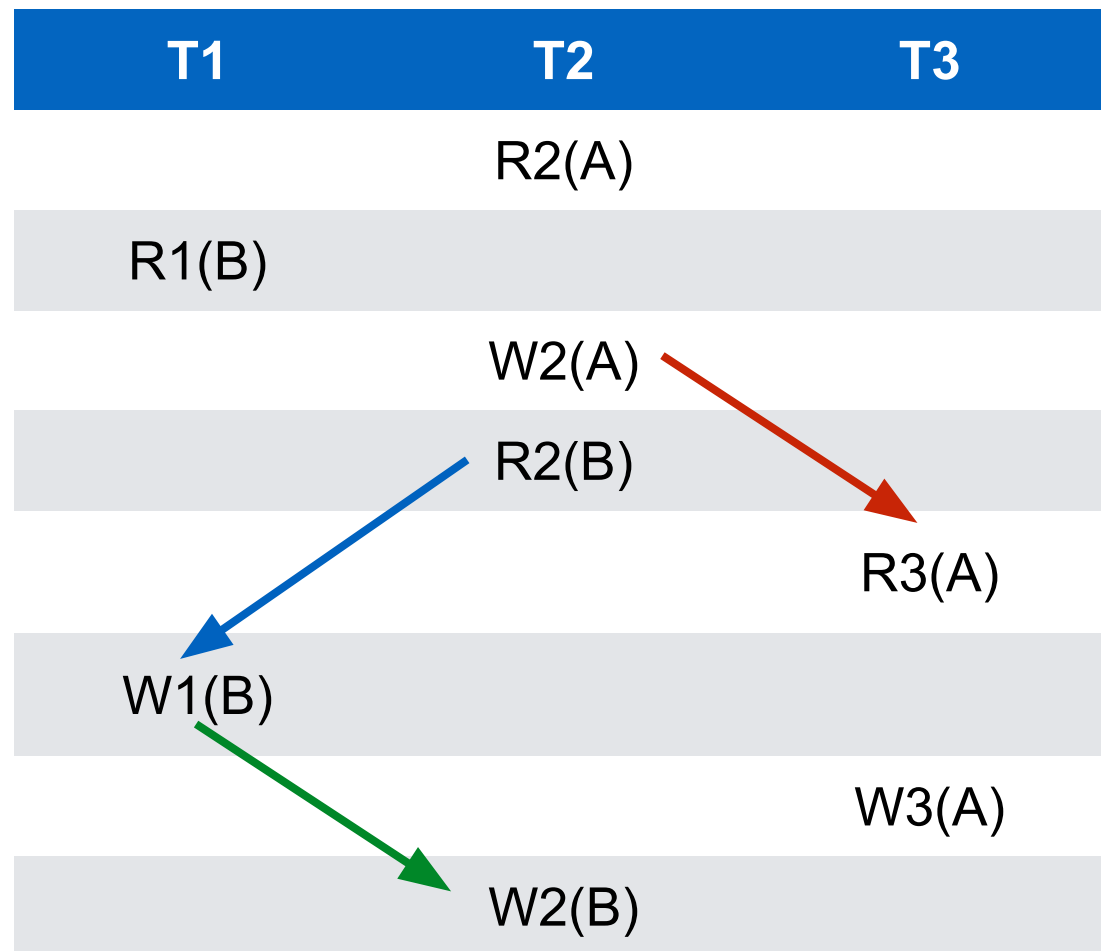
¿Es *conflict serializable*?



Grafo de precedencia

Ejemplo (Pizarra)

¿Es *conflict serializable*?



Grafo de precedencia

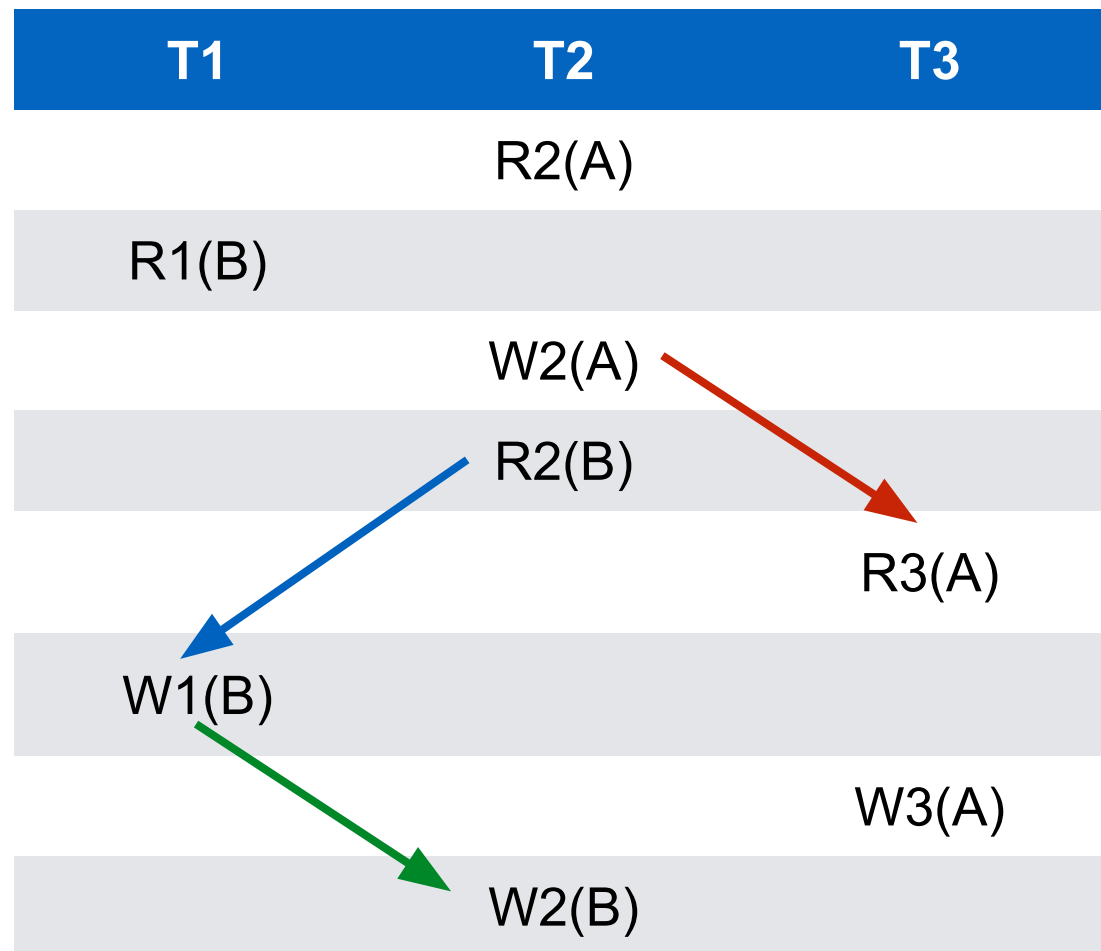
Teorema Un *schedule* es *conflict serializable* ssi el grafo de precedencia es acíclico

Además, determinar si un *schedule* es serializable es NP-Completo!

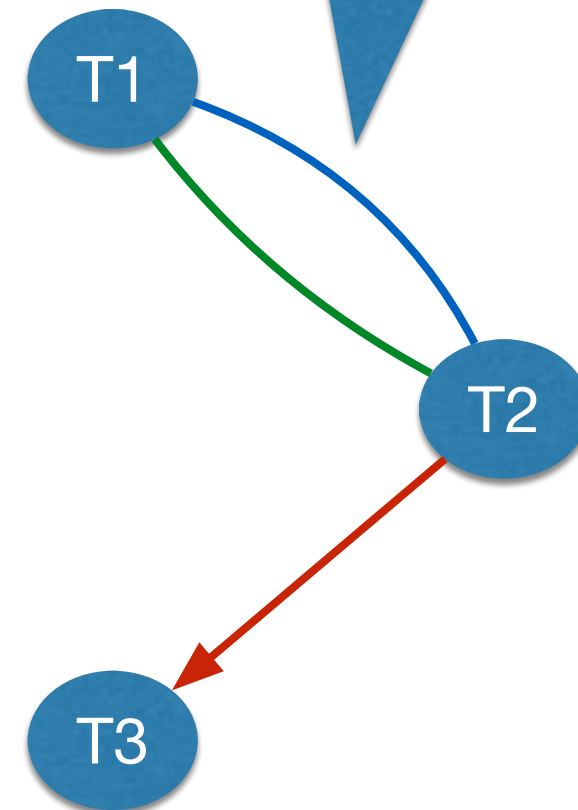
Grafo de precedencia

Ejemplo (Pizarra)

¿Es *conflict serializable*?



Ciclo == no es conflict serializable



Strict 2PL

Es el protocolo para control de concurrencia más usado en los DBMS

Está basado en la utilización de locks

Tiene dos reglas

Strict 2PL

Regla 1

Si una transacción T quiere leer (resp. modificar) un objeto, primero pide un **shared lock** (resp. **exclusive lock**) sobre el objeto

Una transacción que pide un lock se suspende hasta que el lock es otorgado

Strict 2PL

Regla 1

Si una transacción mantiene un exclusive lock de un objeto, ninguna otra transacción puede mantener un shared o exclusive lock sobre el objeto

Es importante notar que por lo anterior, para obtener el exclusive lock, no debe haber ningún lock sobre el objeto

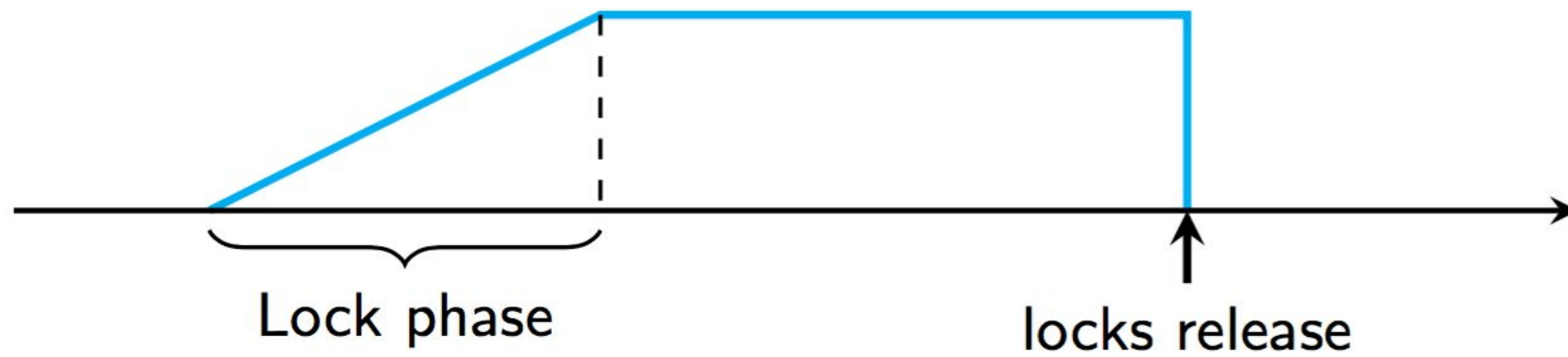
Strict 2PL

Regla 2

Cuando la transacción se completa, libera todos los locks que mantenía

Strict 2PL

Strict 2PL.



Strict 2PL

Estas reglas aseguran solo **schedules**
conflict serializables

Cómo funciona 2PL

Locks de T1:



Locks de T2:

Cómo funciona 2PL

Locks de T1:

Shared lock (A)

T1	T2
R(A)	
	W(A)
	W(B)
W(B)	

Locks de T2:

Cómo funciona 2PL

Locks de T1:

Shared lock (A)



Locks de T2:

← Espera a T1

Cómo funciona 2PL

Locks de T1:

Shared lock (A)

T1	T2
R(A)	
W(B)	
	W(A)
	W(B)

Locks de T2:

← Espera a T1

Cómo funciona 2PL



Cómo funciona 2PL

Locks de T1:

Listo con T1



Locks de T2:

← Espera a T1

Cómo funciona 2PL



Deadlock en 2PL

Locks de T1:

T1	T2
R(A)	
	W(B)
	W(A)
W(B)	

Locks de T2:

Deadlock en 2PL

Locks de T1:

Shared lock (A)

T1	T2
R(A)	
	W(B)
	W(A)
W(B)	

Locks de T2:

Deadlock en 2PL

Locks de T1:

Shared lock (A)

T1

R(A)

T2

W(B)

W(A)

W(B)

Locks de T2:

Exclusive lock (B)

Deadlock en 2PL

Locks de T1:

Shared lock (A)

T1	T2
R(A)	
	W(B)
	W(A)
W(B)	

Locks de T2:

Exclusive lock (B)

← Espera a T1

Deadlock en 2PL



Deadlock en 2PL



Deadlock:
T2 espera que termine T1
T1 espera que termine T2

SQL y transacciones

Lo básico

```
START TRANSACTION;
```

```
    SELECT a_nombre
```

```
    FROM Actores;
```

```
COMMIT;
```

SQL y transacciones

Lo básico

Se hace automáticamente cuando se ejecuta una consulta.

¡O cuando uno se conecta a la DB con un lenguaje de programación!

START TRANSACTION;

SELECT a_nombre

FROM Actores;

COMMIT;

SQL y transacciones

Cancelar una transacción

START TRANSACTION;

UPDATE Actores

SET bio = 'El mejor actor'

WHERE nombre = 'Adrian Soto';

ROLLBACK;



Para deshacer una transacción

SQL y transacciones

Savepoints

```
START TRANSACTION;
```

```
UPDATE Actores
```

```
SET bio = 'El mejor actor'
```

```
WHERE nombre = 'Adrian Soto';
```

```
SAVEPOINT MejorActor;
```

```
UPDATE Actores
```

```
SET bio = 'El peor actor'
```

```
WHERE nombre = 'Juan Reutter';
```

```
ROLLBACK TO SAVEPOINT MejorActor;
```

SQL y transacciones

Savepoints

START TRANSACTION;

UPDATE Actores

SET bio = 'El major actor'

WHERE nombre = 'Adrian Soto'

SAVEPOINT **MejorActor**;

UPDATE Actores

SET bio = 'El peor actor'

WHERE nombre = 'Juan Reutter';

ROLLBACK TO SAVEPOINT **MejorActor**;

Al ejecutar, se borra el SAVEPOINT

Útil en un programa qué hace varias transacciones y verifica condiciones

SQL y transacciones

Granularidad de locks

```
SELECT S.rating, MIN(S.age)
FROM Sailors AS S
WHERE S.rating = 8;
```

sid	age	rating
1	34	9
2	21	8
3	16	8
4	26	10

SQL y transacciones

Granularidad de locks

```
SELECT S.rating, MIN(S.age)
FROM Sailors AS S
WHERE S.rating = 8;
```

sid	age	rating
1	34	9
2	21	8
3	16	8
4	26	10

Lock seguro: La tabla S

SQL y transacciones

Granularidad de locks

```
SELECT S.rating, MIN(S.age)
FROM Sailors AS S
WHERE S.rating = 8;
```

sid	age	rating
1	34	9
2	21	8
3	16	8
4	26	10

Lock seguro: La tabla S

Lock razonable: Tuplas de S con rating = 8

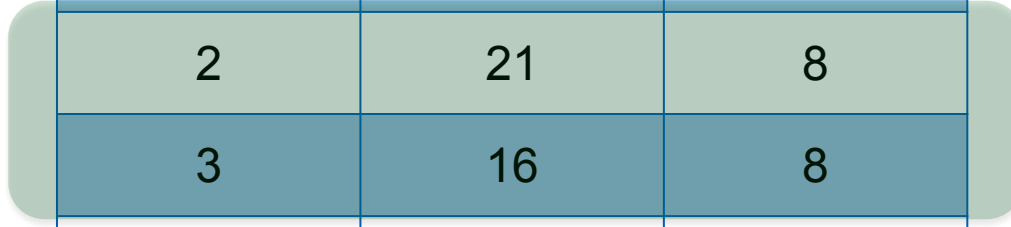
SQL y transacciones

Granularidad y "fantasmas"

T1

```
SELECT S.rating, MIN(S.age)
FROM Sailors AS S
WHERE S.rating = 8;
```

shared lock




sid	age	rating
1	34	9
2	21	8
3	16	8
4	26	10

shared lock

T2

```
INSERT INTO Sailors AS S
VALUES (5,22,8);
```

"fantasma"



sid	age	rating
1	34	9
2	21	8
3	16	8
4	26	10
5	22	8


SQL y transacciones

Granularidad y "fantasmas"

T1

```
SELECT S.rating, MIN(S.age)
FROM Sailors AS S
WHERE S.rating = 8;
```

shared lock



sid	age	rating
1	34	9
2	21	8
3	16	8
4	26	10

T2

```
INSERT INTO Sailors AS S
VALUES (5,22,8);
```

espera a T1



SQL y transacciones

Nivel de aislamiento

SET TRANSACTION ISOLATION LEVEL READ
ONLY

SET TRANSACTION ISOLATION LEVEL READ
WRITE

SQL y transacciones

Nivel de aislamiento

SET TRANSACTION ISOLATION LEVEL READ
ONLY

SET TRANSACTION ISOLATION LEVEL READ
WRITE



¿Qué puedo hacer sobre las tablas en mi transacción?

SQL y transacciones

Nivel de aislamiento

SET TRANSACTION ISOLATION LEVEL READ ONLY

SET TRANSACTION ISOLATION LEVEL READ WRITE



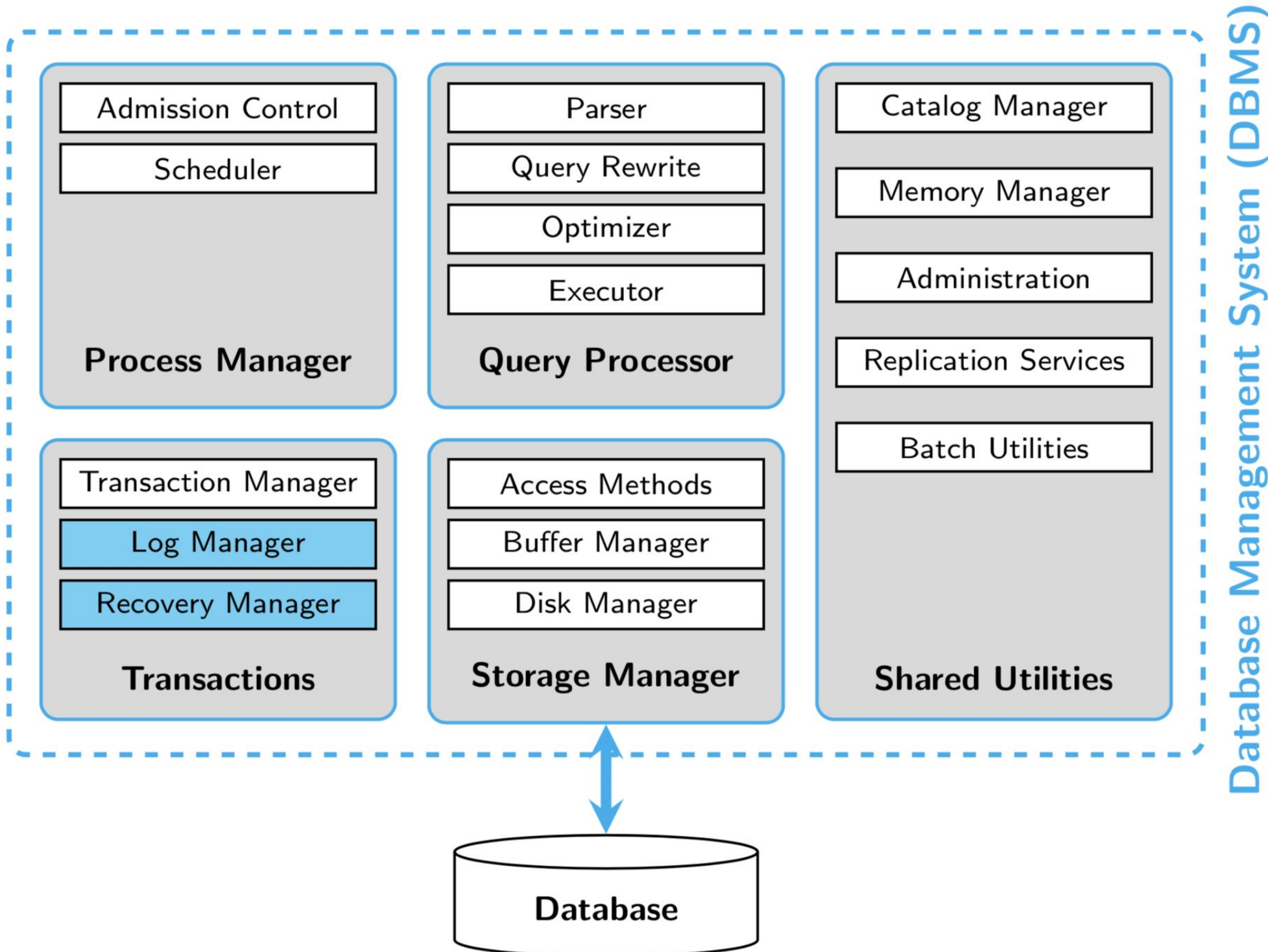
Dirty Read	Unrepeatable Read	Phantom
No	No	No
No	No	Maybe
No	Maybe	Maybe
Maybe	Maybe	Maybe

SQL y transacciones

Por defecto

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE  
READ WRITE
```


Recuperación de Fallas



Recuperación de Fallas

Log y Recovery Manager se encargan de asegurar Atomicity y Durability

¿Pero qué puede salir mal?

Fallas en la ejecución:

- Datos erróneos
 - Solución: restricciones de integridad, data cleaning
- Fallas en el disco duro
 - Solución: RAID, copias redundantes

¿Pero qué puede salir mal?

Fallas en la ejecución:

- Catástrofes
 - Solución: copias distribuidas
- Fallas del sistema
 - Solución: **Log y Recovery Manager**

Log Manager

Una página se va llenando secuencialmente con *logs*

Cuando la página se llena, se almacena en disco

Todas las transacciones escriben el *log* de manera concurrente

Log Manager

Registra todas las acciones de las transacciones

Log Records

Los *logs* comunes son:

- **<START T>**
- **<COMMIT T>**
- **<ABORT T>**
- **<T UPDATE>**

¿Cómo los usamos?

Undo Logging

Forma de escribir los *logs* para poder hacer *recovery* del sistema

Undo Logging

Los *logs* son:

- $\langle \text{START } T \rangle$
- $\langle \text{COMMIT } T \rangle$
- $\langle \text{ABORT } T \rangle$
- $\langle T, X, t \rangle$ donde t es el valor **antiguo** de X

Undo Logging

Regla 1: si **T** modifica X , el *log* $\langle \mathbf{T}, X, t \rangle$ debe ser escrito antes que el valor X sea escrito en disco

Regla 2: si **T** hace *commit*, el log $\langle \text{COMMIT } \mathbf{T} \rangle$ debe ser escrito justo después de que todos los datos modificados por **T** estén almacenados en disco

Undo Logging

En resumen:

- **T** cambia el valor de X (t valor antiguo)
 - Escribir el log $\langle \mathbf{T}, X, t \rangle$ (write-ahead logging)
 - Escribir valor nuevo de X a disco
 - Escribir $\langle \text{COMMIT } \mathbf{T} \rangle$
- } repeat

Recuperación con Undo Logging

Detectando fallas en el *log*:

... <START T> ... <COMMIT T> ...



Recuperación con Undo Logging

Detectando fallas en el *log*:

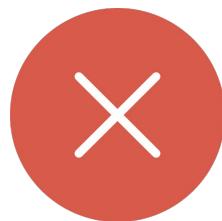
... <START T> ... <ABORT T> ...



Recuperación con Undo Logging

Detectando fallas en el *log*:

... <START T> ...



Recuperación con Undo Logging

Supongamos que mientras usamos nuestro sistema, se apagó de forma imprevista

Leyendo el *log* podemos hacer que la base de datos quede en un estado consistente

Recovery

Algoritmo para un *Undo Logging*

Procesamos el log desde el final hasta el principio:

- Si leo $\langle \text{COMMIT } T \rangle$, marco T como realizada
- Si leo $\langle \text{ABORT } T \rangle$, marco T como realizada
- Si leo $\langle T, X, t \rangle$, debo restituir $X := t$ en disco, si no fue realizada.
- Si leo $\langle \text{START } T \rangle$, lo ignoro

Recovery

Algoritmo para un *Undo Logging*

- ¿Hasta dónde tenemos que leer el *log*?
- ¿Qué pasa si el sistema falla en plena recuperación?
- ¿Cómo trucamos el *log*?

Recovery

Uso de *Checkpoints*

Utilizamos *checkpoints* para no tener que leer el *log* entero y para manejar las fallas mientras se hace *recovery*

Recovery

Uso de *Checkpoints*

- Dejamos de escribir transacciones
- Esperamos a que las transacciones actuales terminen
- Se guarda el *log* en disco
- Escribimos <CKPT> y se guarda en disco
- Se reanudan las transacciones

Recovery

Uso de *Checkpoints*

Ahora hacemos *recovery* hasta leer un <CKPT>

Problema: es prácticamente necesario apagar el sistema para guardar un *checkpoint*

Recovery

Uso de *Nonquiescent Checkpoints*

Nonquiescent Checkpoints son un tipo de *checkpoint* que no requiere "apagar" el sistema

Recovery

Uso de *Nonquiescent Checkpoints*

- Escribimos un *log* $\langle \text{START CKPT } (T_1, \dots, T_n) \rangle$, donde T_1, \dots, T_n son transacciones activas
- Esperamos hasta que T_1, \dots, T_n terminen, sin restringir nuevas transacciones
- Cuando T_1, \dots, T_n hayan terminado, escribimos $\langle \text{END CKPT} \rangle$

Undo Recovery

Uso de *Nonquiescent Checkpoints*

- Avanzamos desde el final al inicio
- Si encontramos un $\langle \text{END CKPT} \rangle$, hacemos *undo* de todo lo que haya después del inicio del *checkpoint*
- Si encontramos un $\langle \text{START CKPT } (T_1, \dots, T_n) \rangle$ sin su $\langle \text{END CKPT} \rangle$, debemos analizar el log desde el inicio de la transacción más antigua entre T_1, \dots, T_n

Ejemplo

Uso de *Checkpoints* en *Undo Logging*

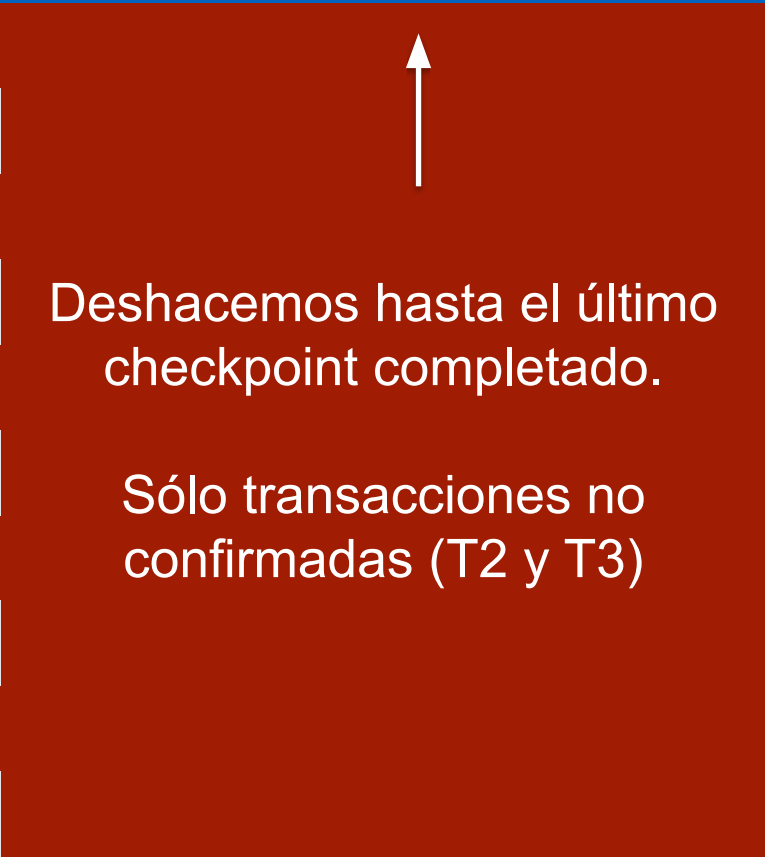
Considere este *log* después de una falla:

Log	
<START T1>	Podemos truncar esta parte
<T1, a, 5>	
<START T2>	
<T2, b, 10>	
<START CKPT (T1, T2)>	T1 y T2 activas
<T2, c, 15>	Deshacemos solo esta parte
<START T3>	
<T1, d, 20>	
<COMMIT T1>	Noten que T3 partió después del checkpoint
<T3, e, 25>	
<COMMIT T2>	
<END CKPT>	

Ejemplo

Uso de *Checkpoints* en *Undo Logging*

Ahora considere este *log* después de una falla:

Log	
<START T1>	 <p>Deshacemos hasta el último checkpoint completado.</p> <p>Sólo transacciones no confirmadas (T2 y T3)</p>
<T1, a, 5>	
<START T2>	
<T2, b, 10>	
<START CKPT (T1, T2)>	
<T2, c, 15>	
<START T3>	
<T1, d, 20>	
<COMMIT T1>	
<T3, e, 25>	

Undo Logging

Problema: no es posible hacer COMMIT antes de almacenar los datos en disco

Por lo tanto las transacciones se toman más tiempo en terminar!

Redo Logging

Los *logs* son:

- $\langle \text{START } T \rangle$
- $\langle \text{COMMIT } T \rangle$
- $\langle \text{ABORT } T \rangle$
- $\langle T, X, v \rangle$ donde v es el valor **nuevo** de X
- $\langle T, \text{END} \rangle$

Redo Logging

Regla 1: Antes de modificar cualquier elemento X en disco, es necesario que todos los *logs* estén almacenados en disco, *incluido* el COMMIT

Esto es al revés respecto a *Undo Logging*

Redo Logging

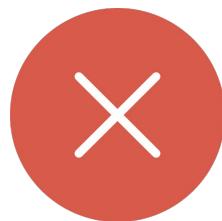
En resumen:

- Escribir el log $\langle \mathbf{T}, X, v \rangle$
- Escribir $\langle \text{COMMIT } \mathbf{T} \rangle$
- Hacer *flush* a disco del *log*
- Escribir los datos en disco
- Escribir $\langle \mathbf{T}, \text{END} \rangle$ en log (en disco)

Recuperación con Redo Logging

Detectando fallas en el *log*:

... <START T> ... <COMMIT T> ...



Recuperación con Redo Logging

Detectando fallas en el *log*:

... <START T> ... <ABORT T> ...



Recuperación con Redo Logging

Detectando fallas en el *log*:

... <START T> ...



Recovery

Algoritmo para un *Redo Logging*

Procesamos el *log* desde el principio hasta el final:

- Identificamos las transacciones que hicieron COMMIT *sin hacer END* (si hicieron END todo OK)
- Hacemos un *scan* desde el principio
- Si leo $\langle T, X, v \rangle$:
 - Si **T** no hizo COMMIT, no hacer nada
 - Si **T** hizo COMMIT, reescribir con el valor v
- Para cada transacción sin COMMIT, escribir $\langle \text{ABORT } T \rangle$

Recovery

Uso de *Checkpoints* en *Redo Logging*

¿Cómo utilizamos los checkpoints en el Redo Logging?

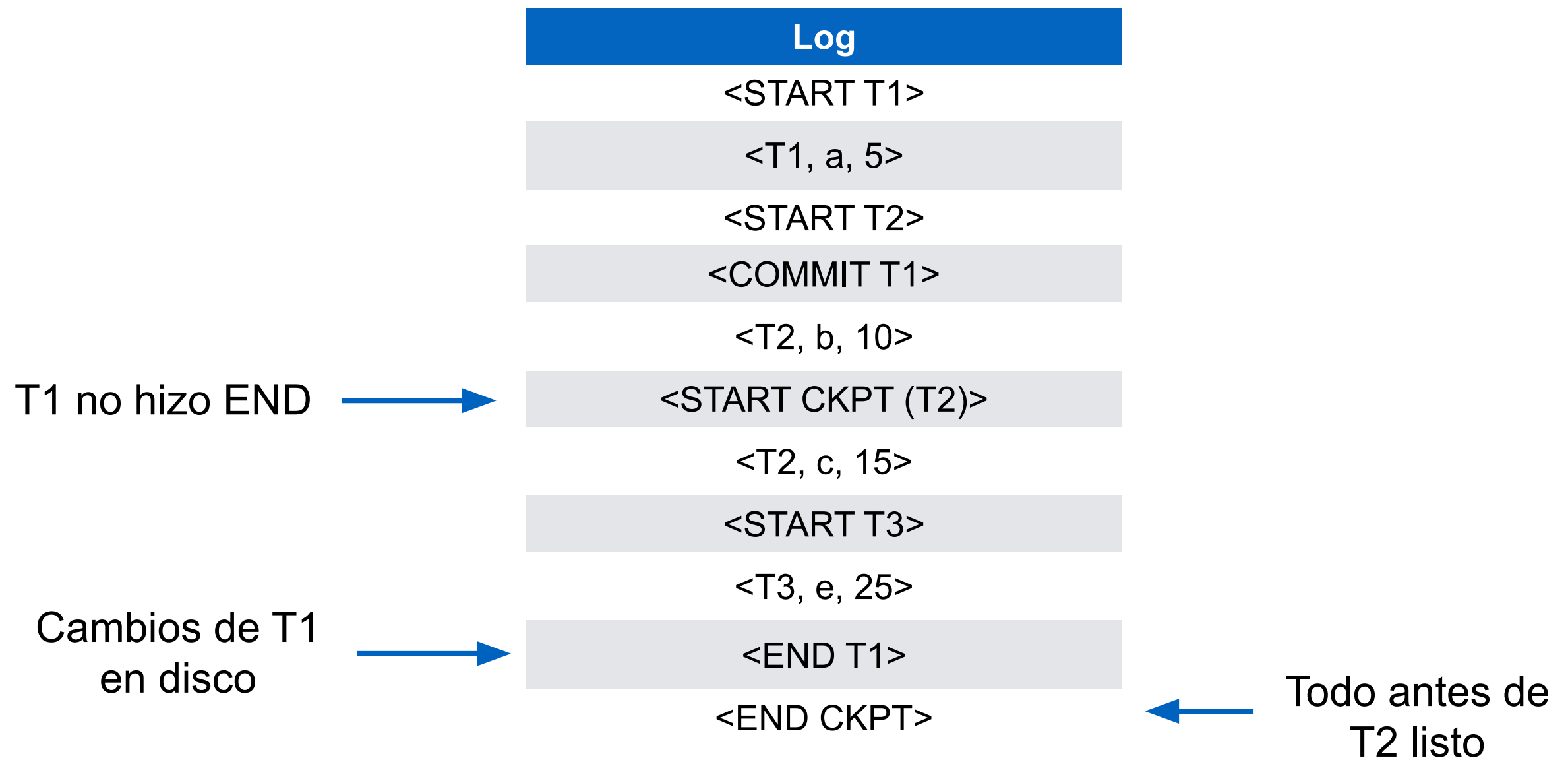
Recovery

Uso de *Checkpoints* en *Redo Logging*

- Escribimos un *log* $\langle \text{START CKPT } (T_1, \dots, T_n) \rangle$, donde T_1, \dots, T_n son transacciones activas y sin COMMIT
- Guardar todo el log en el disco
- Guardar en disco todo lo que haya hecho COMMIT hasta ese punto; escribir en log END al finalizar
- Una vez hecho, escribir $\langle \text{END CKPT} \rangle$

Ejemplo

Cuand finalizamos un checkpoint



Redo Recovery

Uso de *Checkpoints* en *Redo Logging*

- Revisar el *log* desde el final al inicio
- Si encontramos un $\langle \text{END CKPT} \rangle$, debemos retroceder hasta su respectivo $\langle \text{START CKPT } (T_1, \dots, T_n) \rangle$, y comenzar a hacer *redo* desde la transacción más antigua entre T_1, \dots, T_n
- No se hace *redo* de las transacciones con COMMIT antes del $\langle \text{START CKPT } (T_1, \dots, T_n) \rangle$

Redo Recovery

Uso de *Checkpoints* en *Redo Logging*

- Revisar el *log* desde el final
- Si encontramos un $\langle \text{END CKPT} (T_1, \dots, T_n) \rangle$, retroceder hasta su respectivo $\langle \text{START CKPT} (T_1, \dots, T_n) \rangle$, y comenzar a hacer *redo* desde la transacción más antigua entre T_1, \dots, T_n
- No se hace *redo* de las transacciones con COMMIT antes del $\langle \text{START CKPT} (T_1, \dots, T_n) \rangle$

Su END aparece antes de
 $\langle \text{END CKPT} \rangle$

Redo Recovery

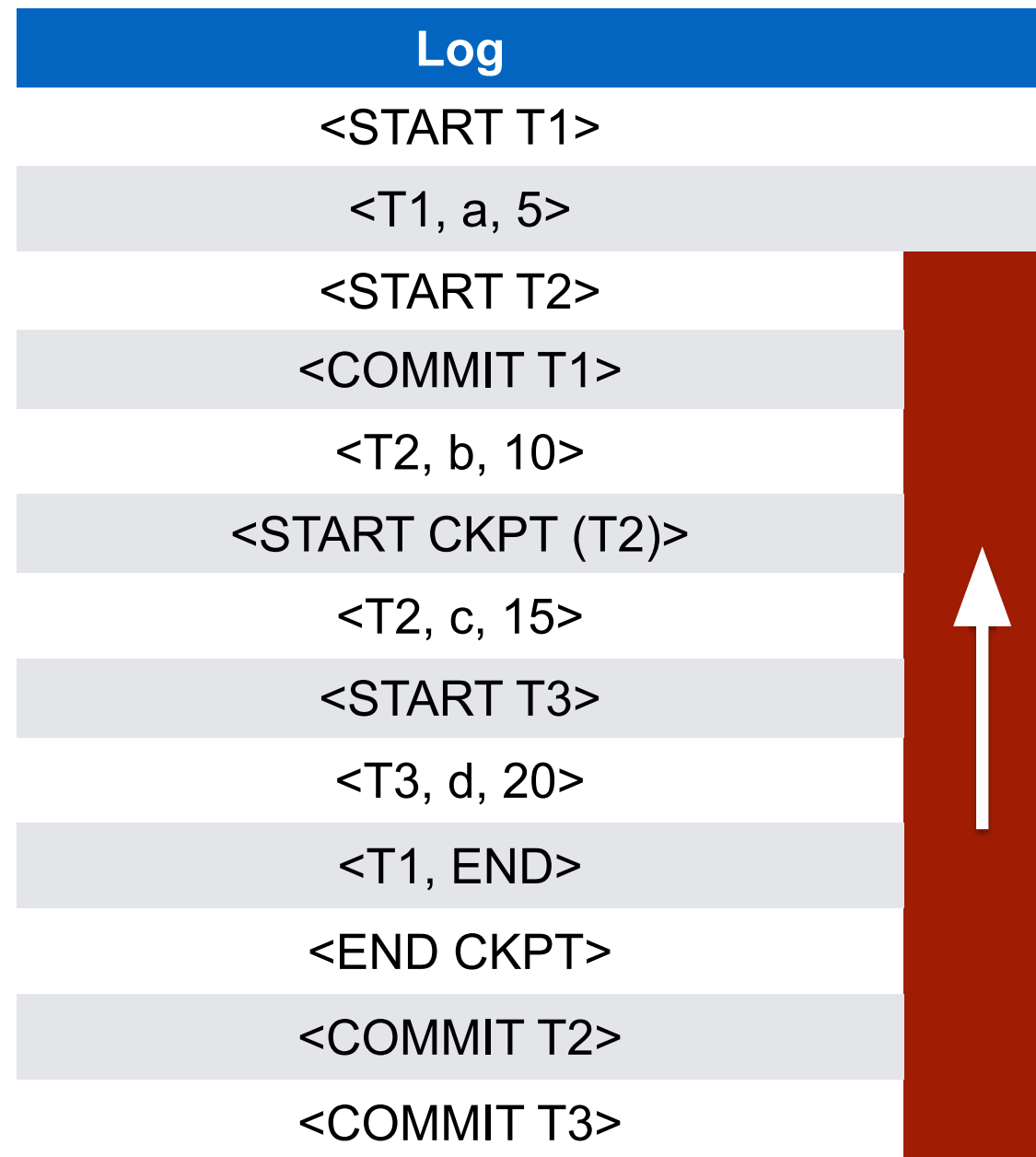
Uso de *Checkpoints* en *Redo Logging*

Si encontramos un $\langle \text{START CKPT } (T_1, \dots, T_n) \rangle$ sin su $\langle \text{END CKPT} \rangle$, debemos retroceder hasta encontrar un $\langle \text{END CKPT} \rangle$ más antiguo

Ejemplo

Uso de *Checkpoints* en *Redo Logging*

Considere este *log* después de una falla:



Ejemplo

Uso de *Checkpoints* en *Redo Logging*

Considere este *log* después de una falla:

Log
<START T1>
<T1, a, 5>
<START T2>
<COMMIT T1>
<T2, b, 10>
<START CKPT (T2)>
<T2, c, 15>
<START T3>
<T3, d, 20>
<T1, END>
<END CKPT>
<COMMIT T2>
<COMMIT T3>

Ahora REDO

Redo Logging

Problema: no es posible ir grabando los valores de X en disco antes que termine la transacción

Por lo tanto se congestiona la escritura en disco!

Undo/Redo Logging

Es la solución para obtener mayor performance que mezcla las estrategias anteriormente planteadas

Undo/Redo Logging

Es la solución para obtener mayor performance que mezcla las estrategias anteriormente planteadas

Usa $\langle T, X, v_{\text{antiguo}}, v_{\text{nuevo}} \rangle$ con write-ahead logging
 $\langle T, \text{commit} \rangle$ va de manera arbitraria al disco

Recuperación:

- Undo de transacciones sin COMMIT
- Redo transacciones con COMMIT (pero no en disco)

Técnicas de Logging

Resumen

Undo

Redo

Trans. Incompletas

Cancelarlas

Ignorarlas

Trans. Comiteadas

Ignorarlas

Repetirlas

Escribir COMMIT

Después de almacenar en disco

Antes de almacenar en disco

UPDATE *Log Record*

Valores antiguos

Valores nuevos

Un buen recurso sobre logging

- [http://mlwiki.org/index.php/Database Transaction Log](http://mlwiki.org/index.php/Database_Transaction_Log)