

Bases de Datos

Clase 9: Transacciones y Locks

Hasta ahora

Solo nos hemos dedicado a la Base de datos

- Diseño
- Estructura lógica
- Lenguaje de consulta SQL
- Lógica en la Base de datos (SP, Triggers, vistas)

¿Y qué pasa con el DBMS?

Desde esta clase nos dedicaremos a eso...

Transacciones

¿Que es una transacción?

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

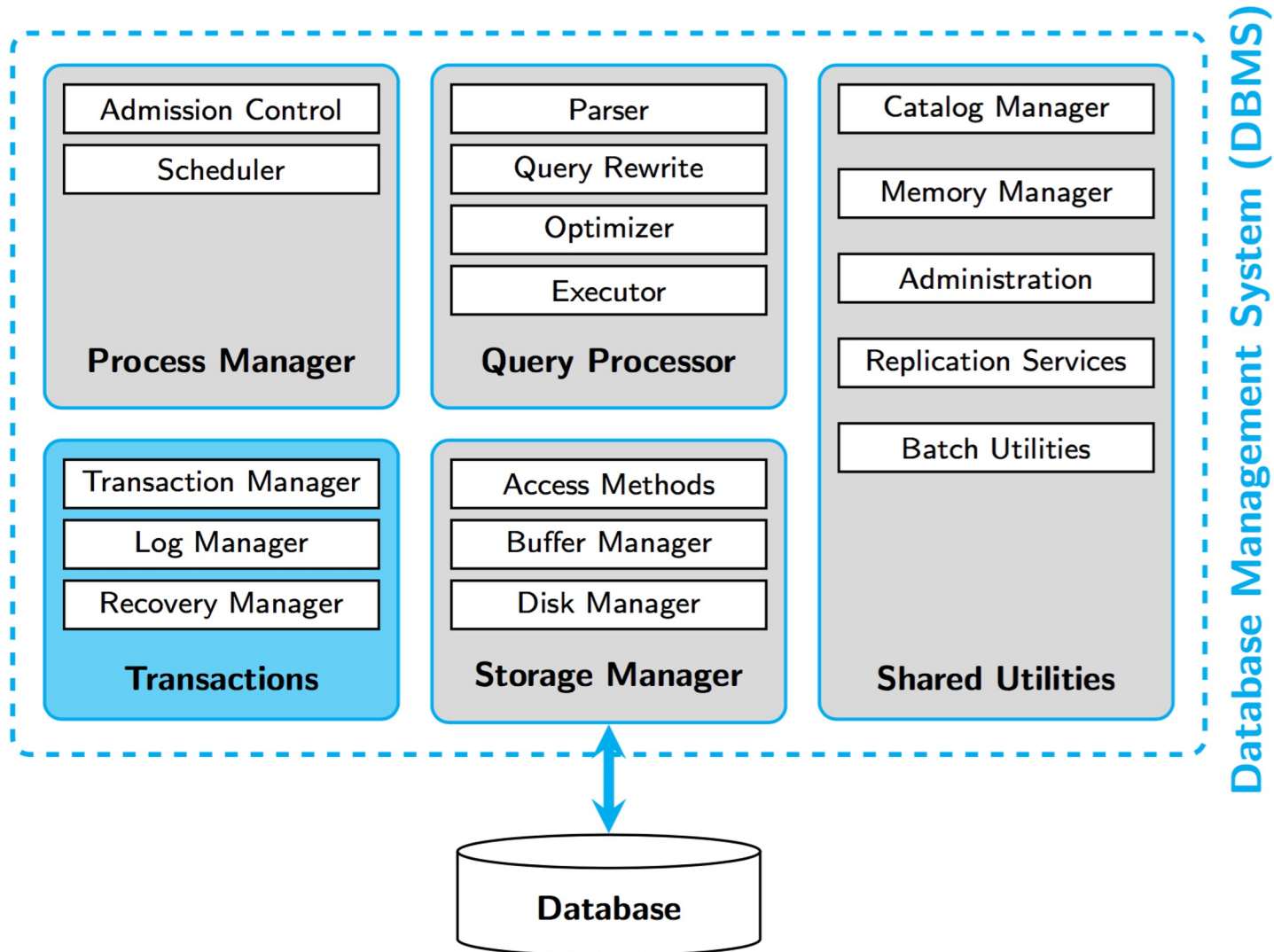
- Transferencias de dinero entre cuentas
- Compra por internet
- Registrar un curso

El principal propósito de las transacciones es garantizar la integridad de los datos y la consistencia del estado de la base de datos, incluso en presencia de fallos y errores

Transacciones

Ventajas	Desventajas
Integridad de los datos - Asegura que todos los cambios en la base de datos son correctos y se preserva la consistencia, incluso ante errores o fallos del sistema.	Complejidad en el manejo - Requiere mecanismos sofisticados de control de concurrencia y recuperación, lo que puede complicar el diseño y el mantenimiento de la base de datos.
Consistencia - Garantiza que la base de datos siempre refleje un estado válido según las reglas definidas, gracias a la ejecución completa de las transacciones.	Rendimiento - Las transacciones, especialmente cuando son grandes o complejas, pueden afectar el rendimiento debido al bloqueo de recursos y a la gestión del control de concurrencia.
Aislamiento - Permite que múltiples transacciones se ejecuten simultáneamente sin interferir entre ellas, manteniendo la exactitud de los datos.	Uso de recursos - Puede requerir un uso intensivo de recursos, como memoria y procesador, para gestionar el estado y la sincronización de las transacciones.
Durabilidad - Una vez que una transacción se completa, sus efectos son permanentes, incluso en caso de fallo del sistema.	Tiempo de espera - Los bloqueos necesarios para mantener el aislamiento pueden llevar a tiempos de espera o a deadlocks que deben ser gestionados cuidadosamente.

DBMS



Módulo de transacciones

- Es uno de los componentes fundamentales de un DBMS
- Fundamental para aplicaciones que requieren seguridad (C.I.A.)

Transacciones

Componente que asegura las propiedades **ACID**



Atomicity
Consistency
Isolation
Durability

Atomicidad
Consistencia
Islamiento
Durabilidad

ACID

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.

Módulo de Transacciones



Transaction Manager se encarga de asegurar Isolation y Consistency

Log y Recovery Manager se encargan de asegurar Atomicity y Durability

Transacciones – Ejemplo:

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
```

Transacciones

Pero el acceso de ambos UPDATE es **concurrente**

veamos un ejemplo...

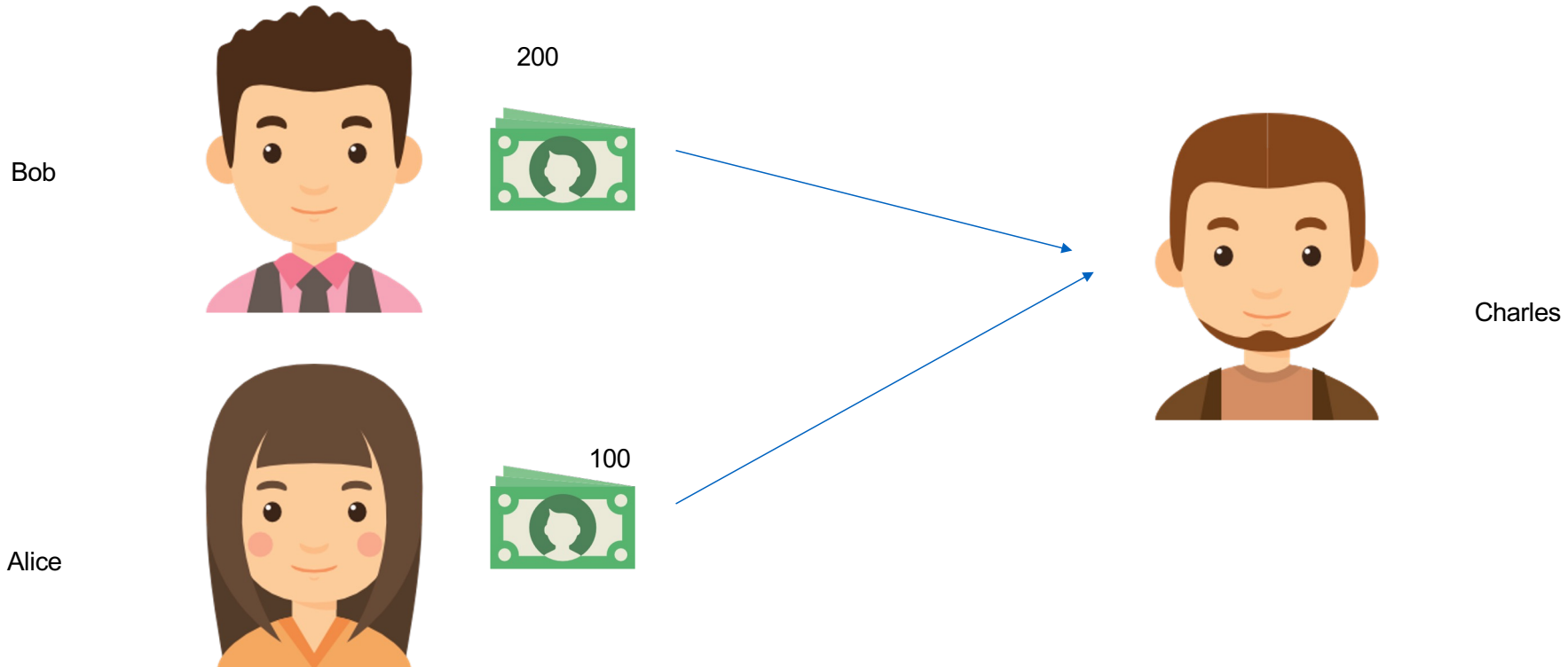
Concurrente: Capacidad de permitir que múltiples usuarios o procesos accedan o modifiquen la base de datos al **mismo tiempo**

Transacciones

Transferencia doble

Supongamos que Alice y Bob tienen una cuenta bancaria en común

Alice quiere transferirle 100 a su amigo Charles
Bob quiere transferirle 200 a su amigo Charles



Transacciones

Transferencia doble

Acceso en SERIE

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

Transacciones

Transferencia doble

Pero el acceso es CONCURRENTE opción 1

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

Transacciones

Transferencia doble

Pero el acceso es CONCURRENTE opción 2

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

¡Se perdieron 200!

¿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? Asegurando las propiedades ACID



Sin ACID

¿Que ocurre si no se siguen las propiedades 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 datos vuelve a su estado pero perdemos la transacción.

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

Sin ACID

¿Que ocurre si no se siguen las propiedades 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

¿Que ocurre si no se siguen las propiedades ACID?

Sin Isolation:

El sistema de base de datos planifica el orden de operaciones

Resultado no es igual a haber corrido transacciones en serie

Transacciones en SQL

Transacciones en SQL

START TRANSACTION

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

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

COMMIT

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

Definamos operaciones

Read(X) o R(X)

Lectura del
elemento X

Write(X) o W(X)

Escritura del
elemento X

Abort o A

Abortar
transacción

Commit o C

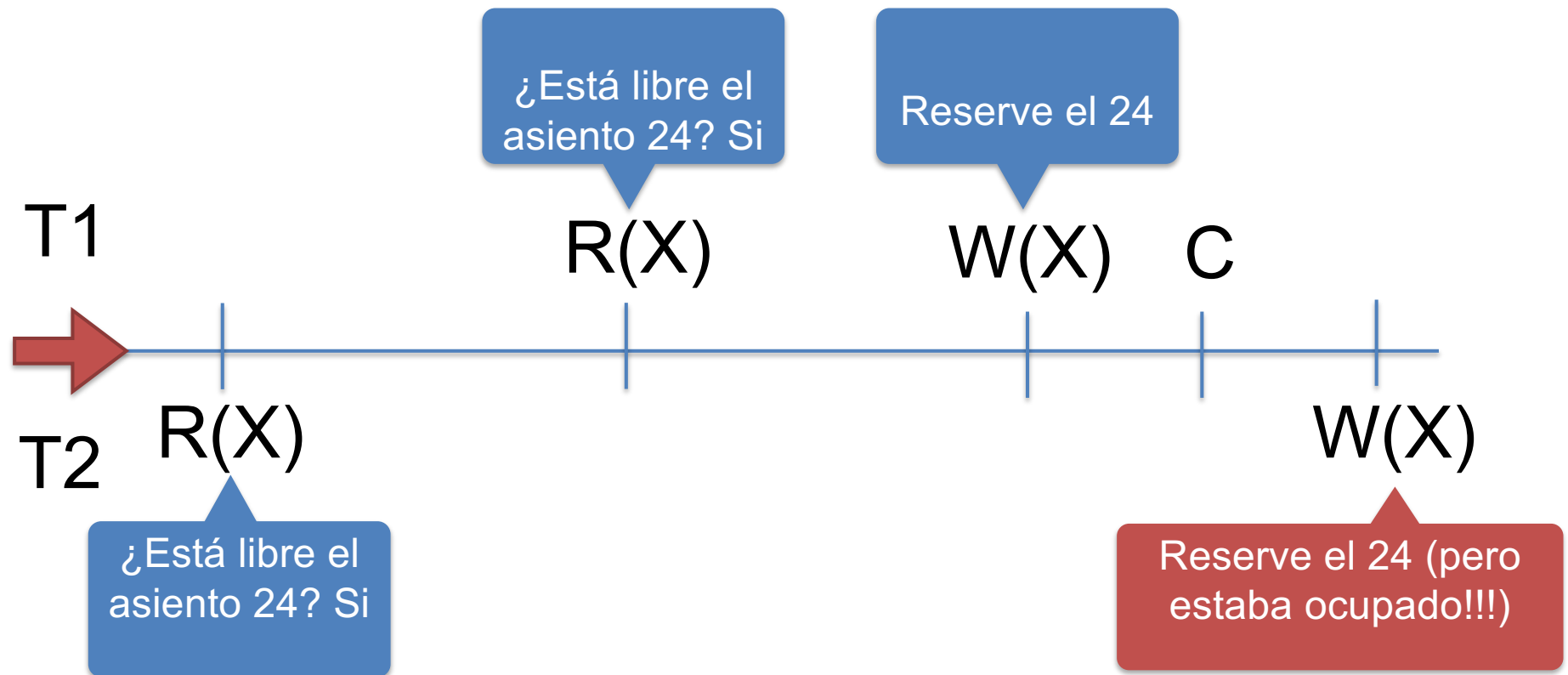
Finalizar
transacción

Conflictos con Transacciones

- Lecturas sucias (Write - Read): ocurren cuando una transacción accede a datos que **han sido modificados** por otra transacción que aún no se ha terminado.
- Lecturas no repetibles (Read - Write): ocurren cuando una fila es leída dos veces y el valor **cambia entre ambas** lecturas. Esto se debe a que otra transacción actualiza o modifica la fila entre las dos lecturas individuales.
- Actualización perdida o reescritura de datos temporales (Write - Write): ocurren cuando dos transacciones que intentan actualizar la misma fila son procesadas en un tiempo que permite que una de las actualizaciones **sobrescriba a la otra**.

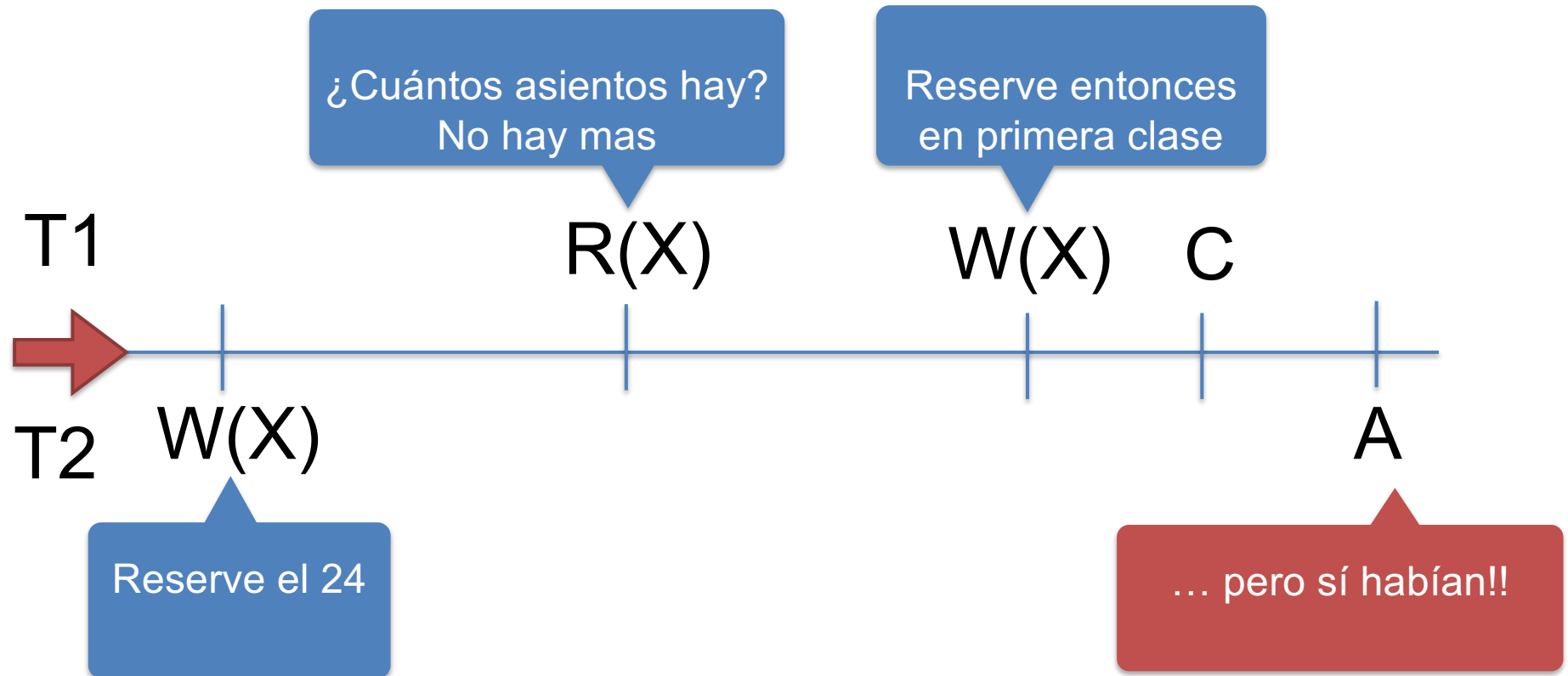
Ejemplo de actualización perdida (WW)

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



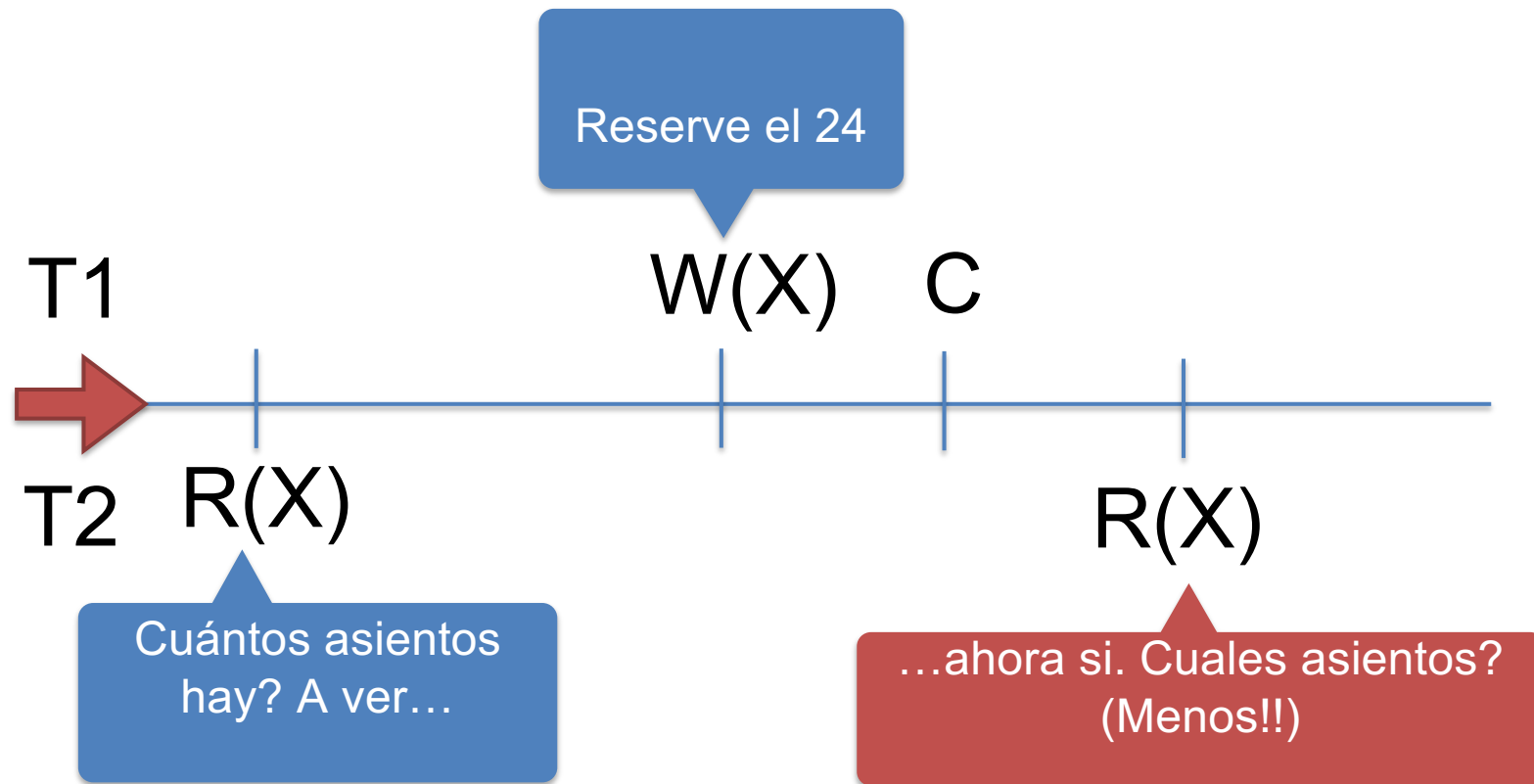
Ejemplo de lectura sucia (WR)

“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.”



Schedules

¿Que es un 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)

Definamos 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)

Definamos 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)

Como las operaciones para A en x e y no afectan al orden de x e y de B, entonces SI es serializable

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)	

En T2, la transaccion de WRITE(B, y) realiza cambios que afectan a READ(B,x), por lo que el orden si cambian los resultados y por lo mismo el schedule No es serializable

Transacciones



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

¿Que problemas pueden surgir si es que un schedule no 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)

Transacciones

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 :

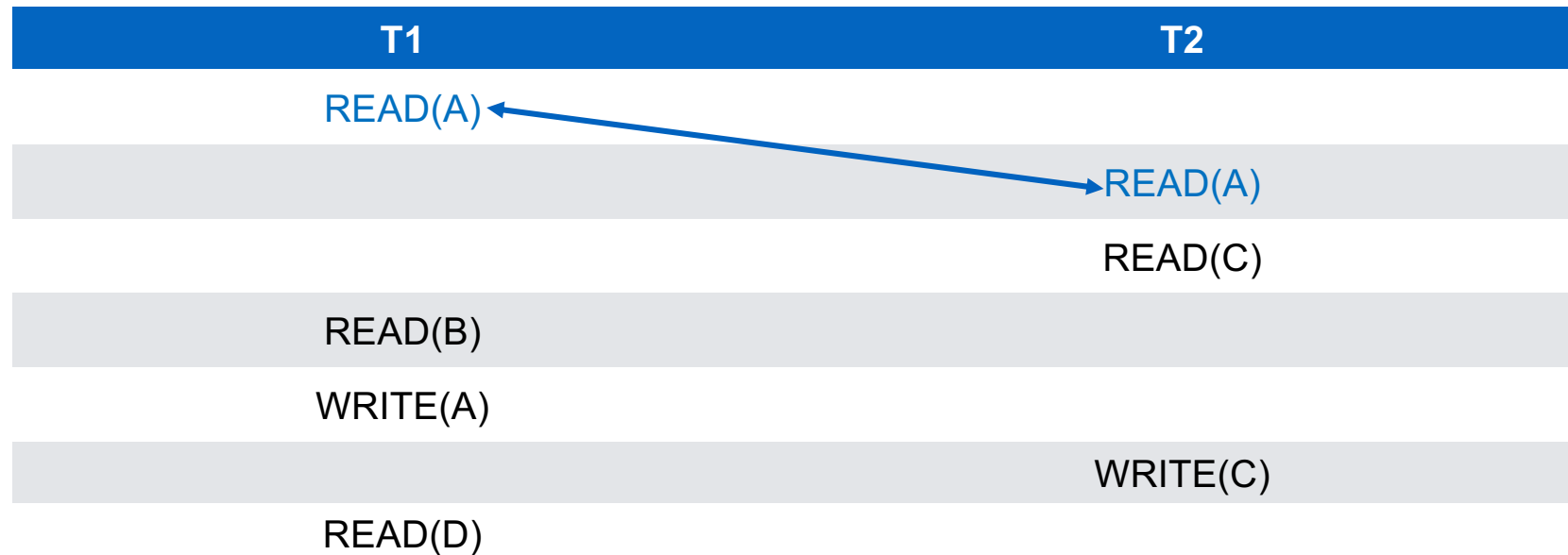
- $R_i(X), R_j(Y)$
- $R_i(X), W_j(Y)$ con $X \neq Y$
- $W_i(X), R_j(Y)$ con $X \neq Y$
- $W_i(X), W_j(Y)$ con $X \neq Y$

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

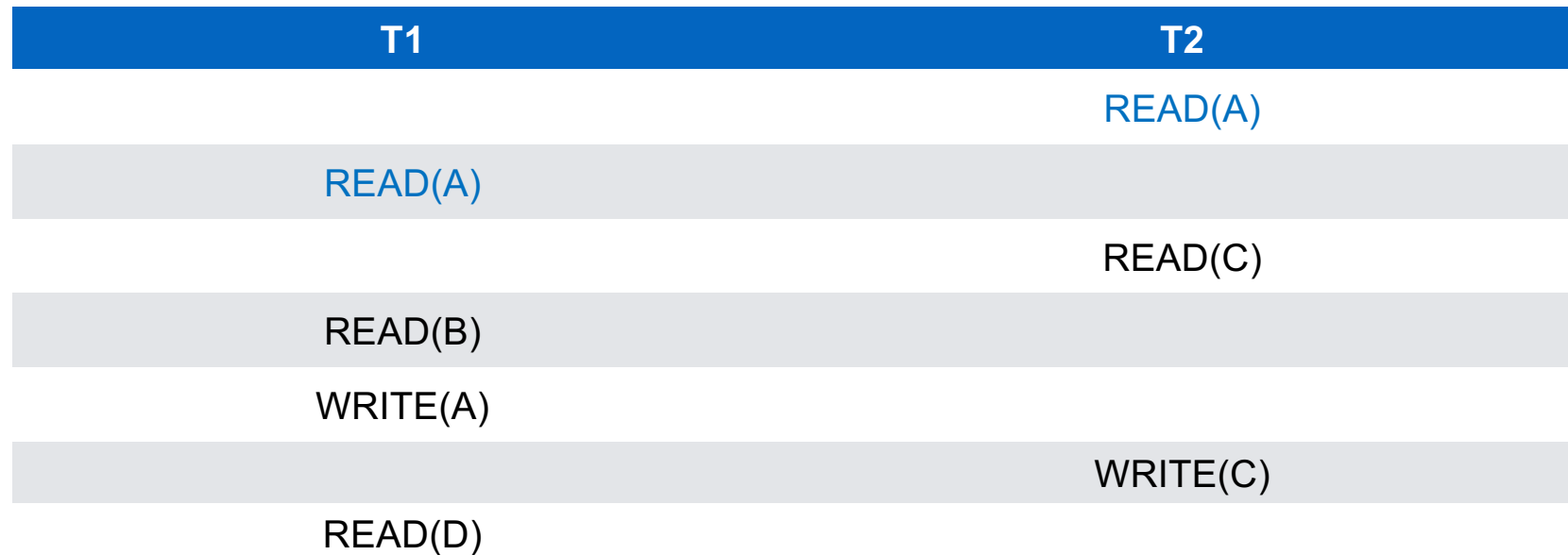
Permutaciones permitidas

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

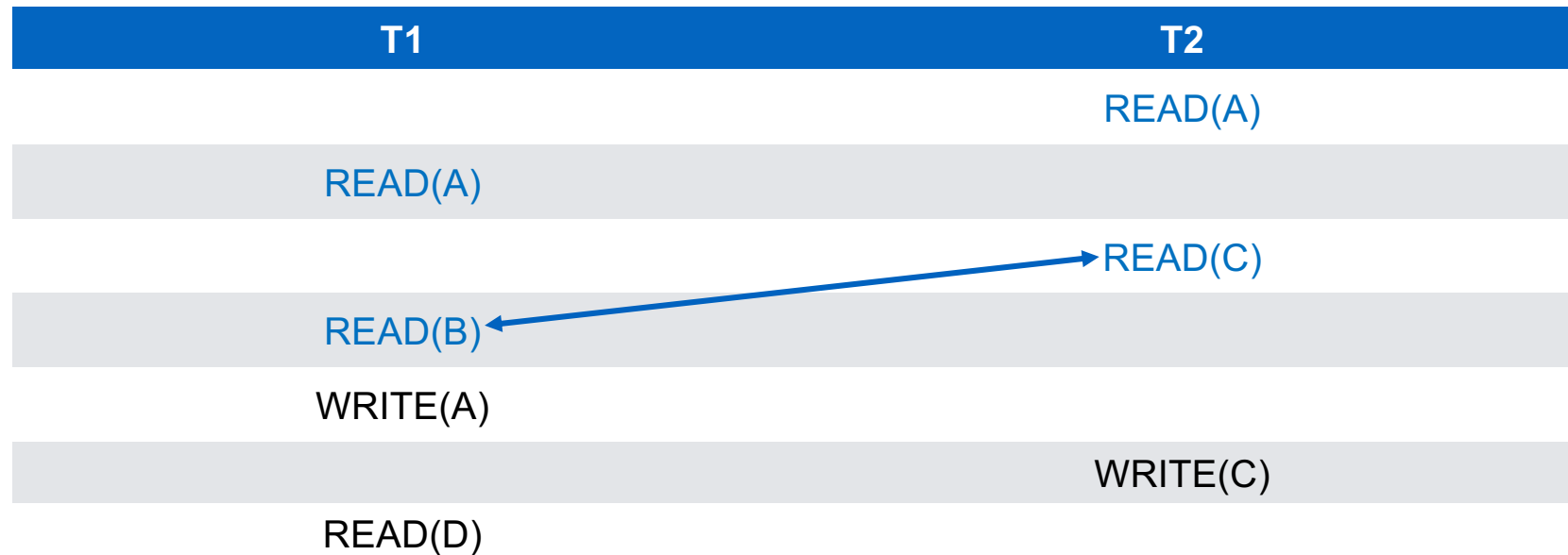
Permutaciones permitidas



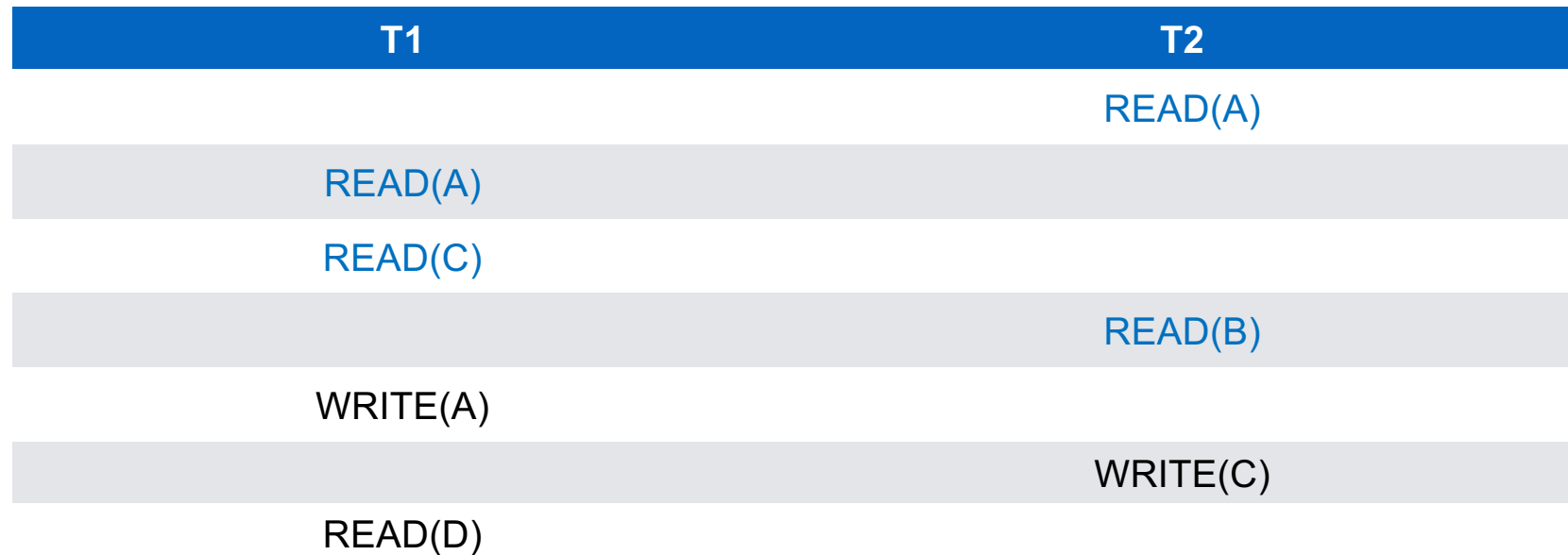
Permutaciones permitidas



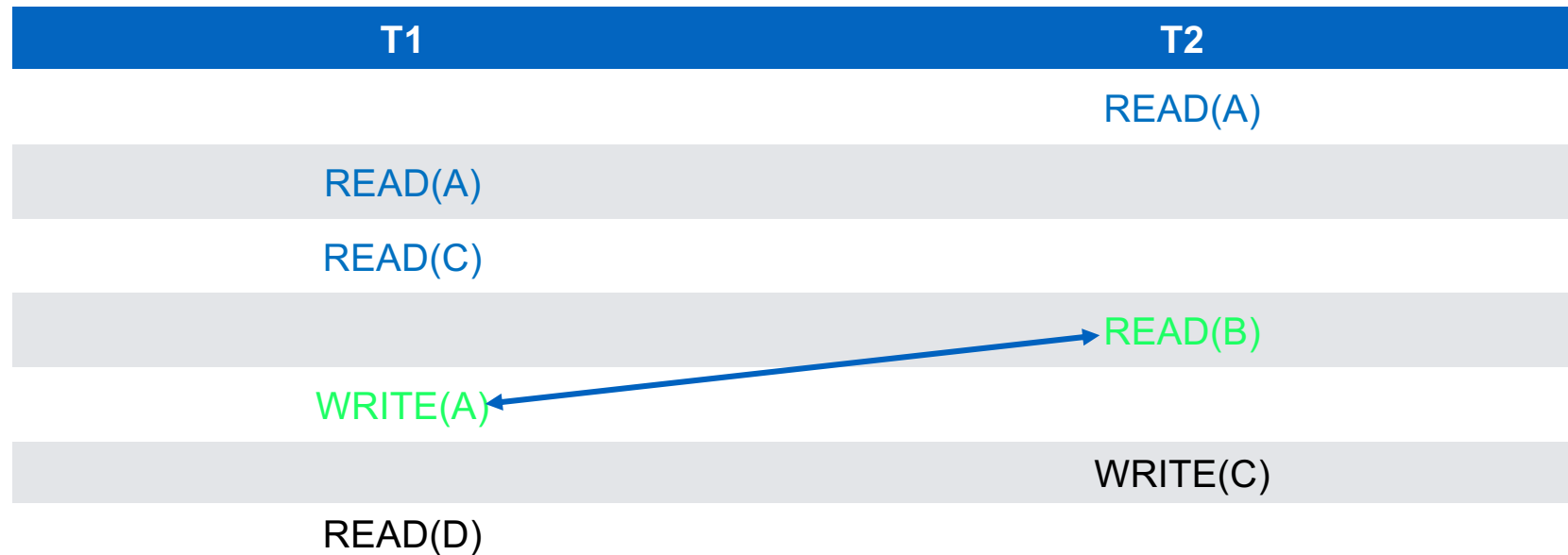
Permutaciones permitidas



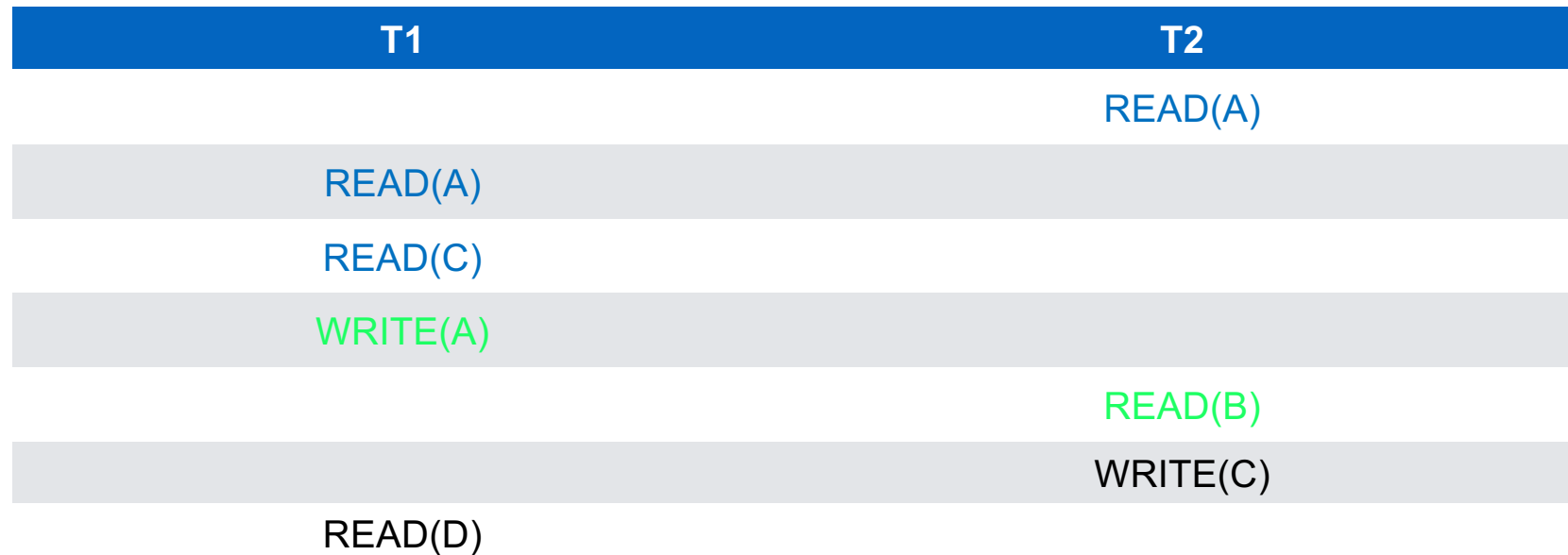
Permutaciones permitidas



Permutaciones permitidas



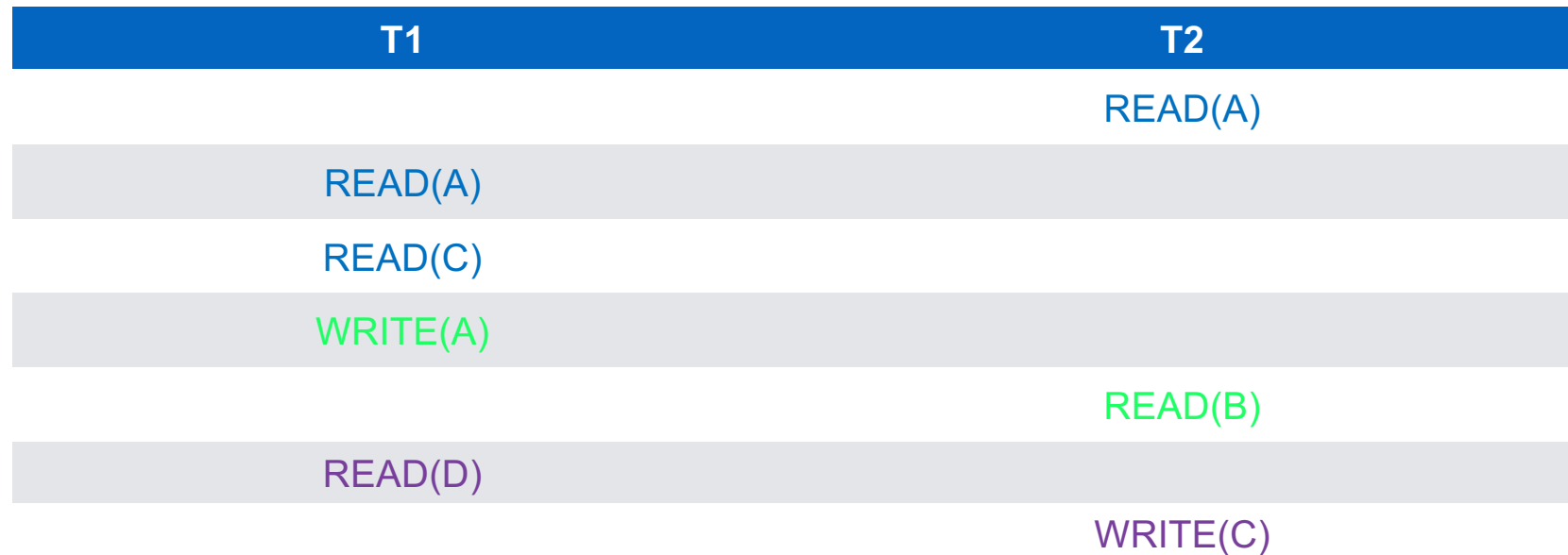
Permutaciones permitidas



Permutaciones permitidas



Permutaciones permitidas



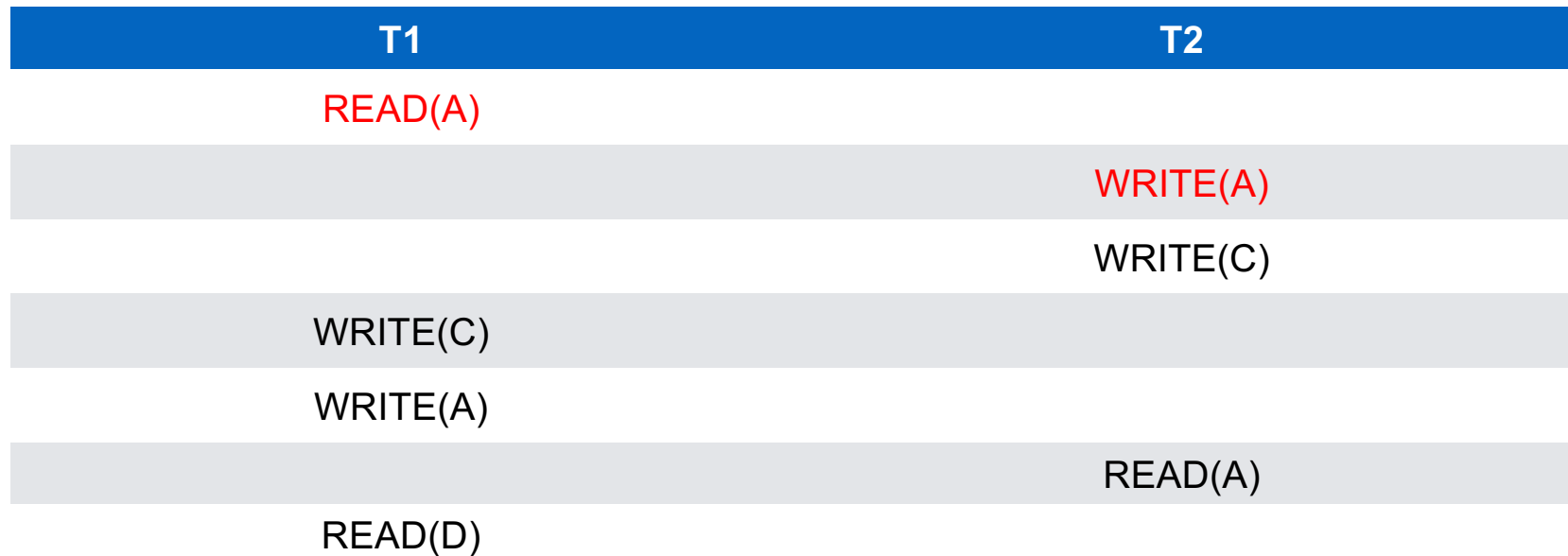
Acciones Conflictivas

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

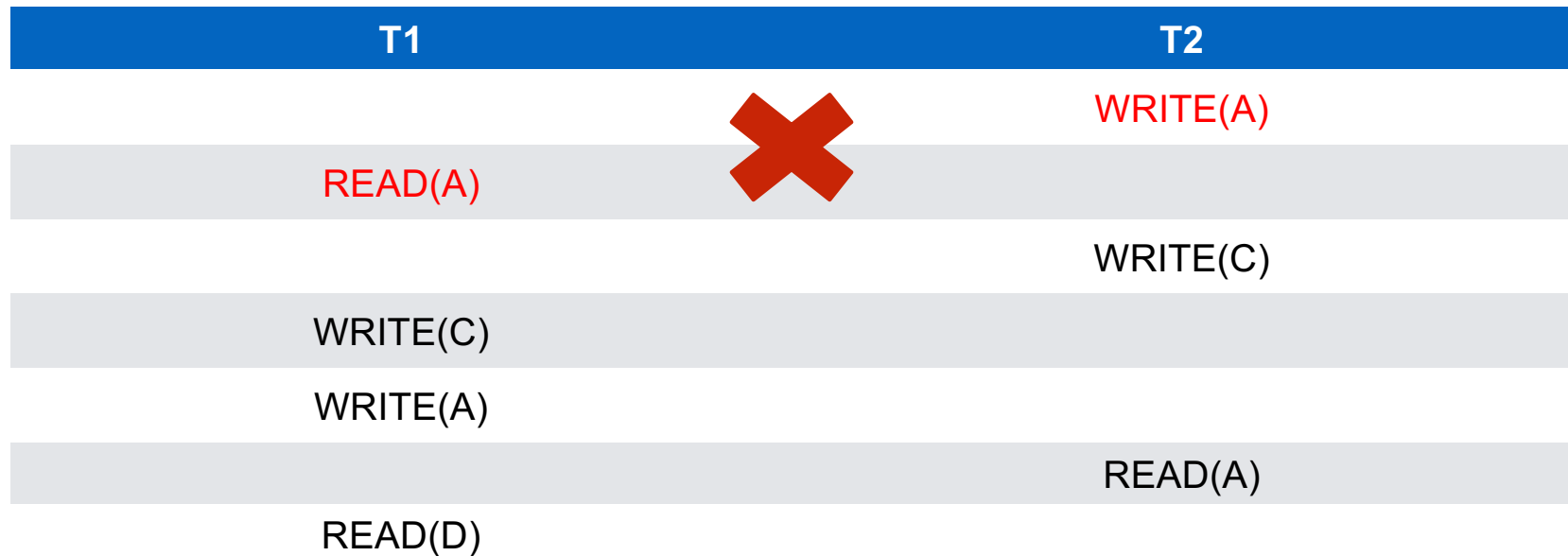
- Lectura-Escritura ($R_i(X), W_j(X)$): Transacción i lee X , transacción j escribe X .
- Escritura-Lectura ($W_i(X), R_j(X)$): Transacción i escribe X , transacción j lee X .
- Escritura-Escritura ($W_i(X), W_j(X)$): Ambas transacciones, i y j , escriben en X .

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

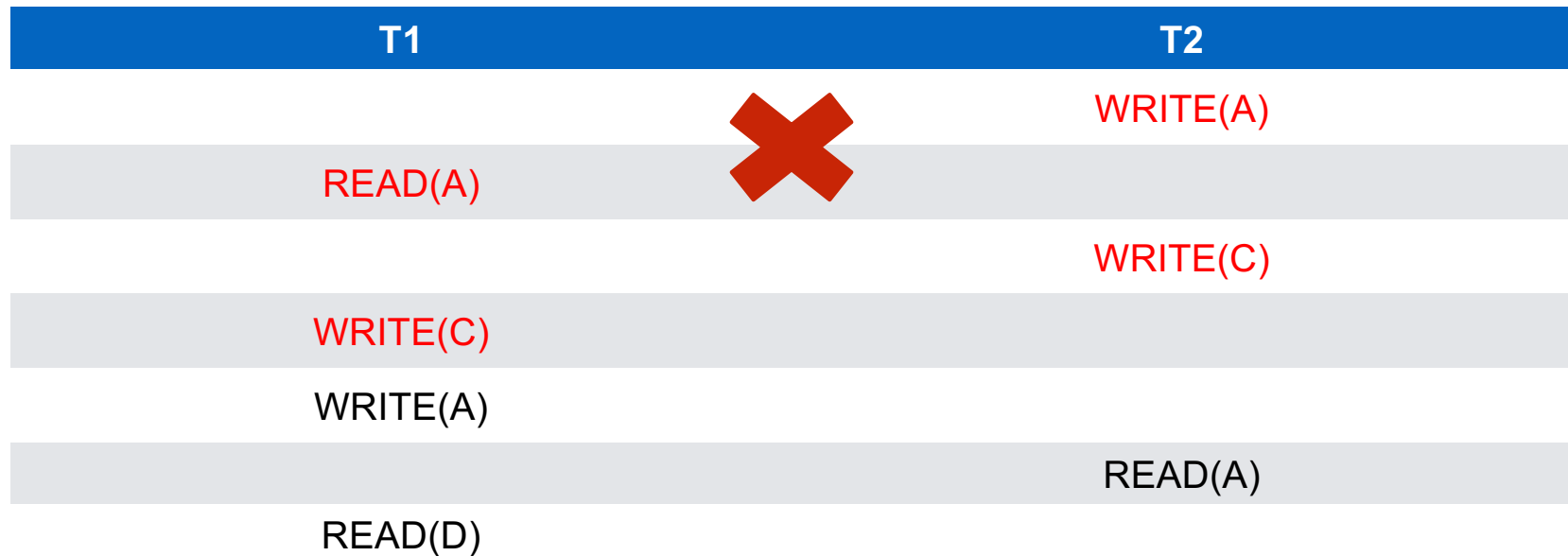
Permutaciones no permitidas



Permutaciones no permitidas

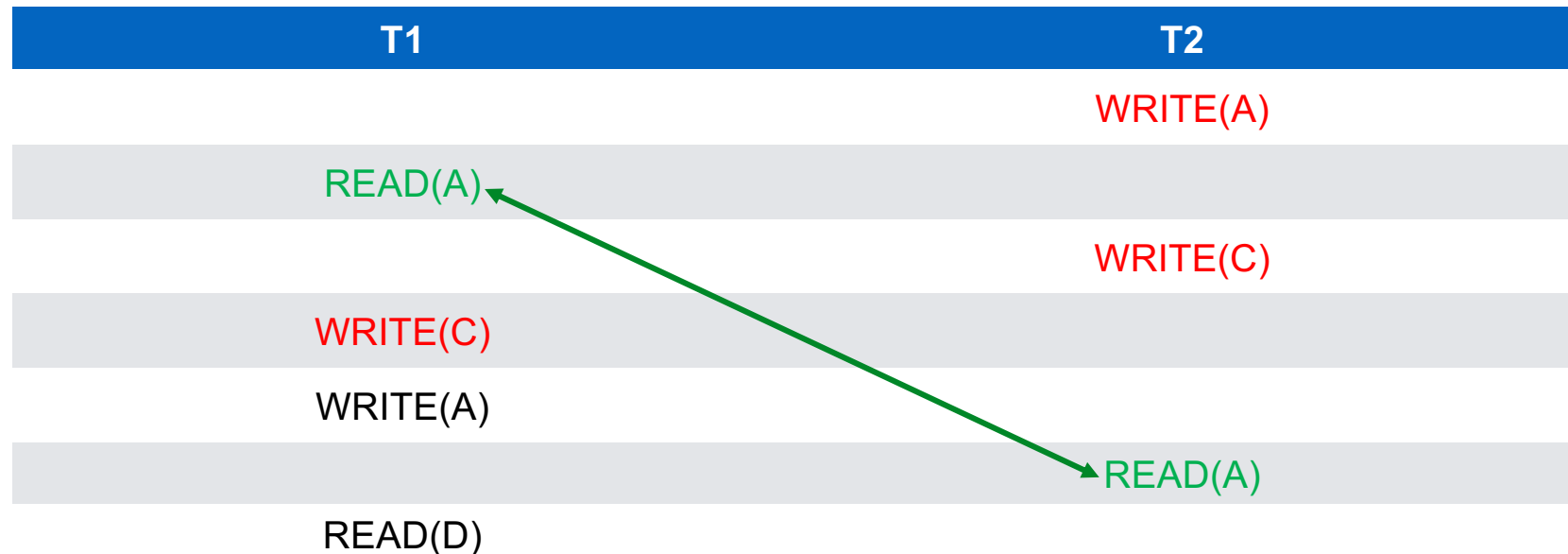


Permutaciones no permitidas



Permutaciones no permitidas

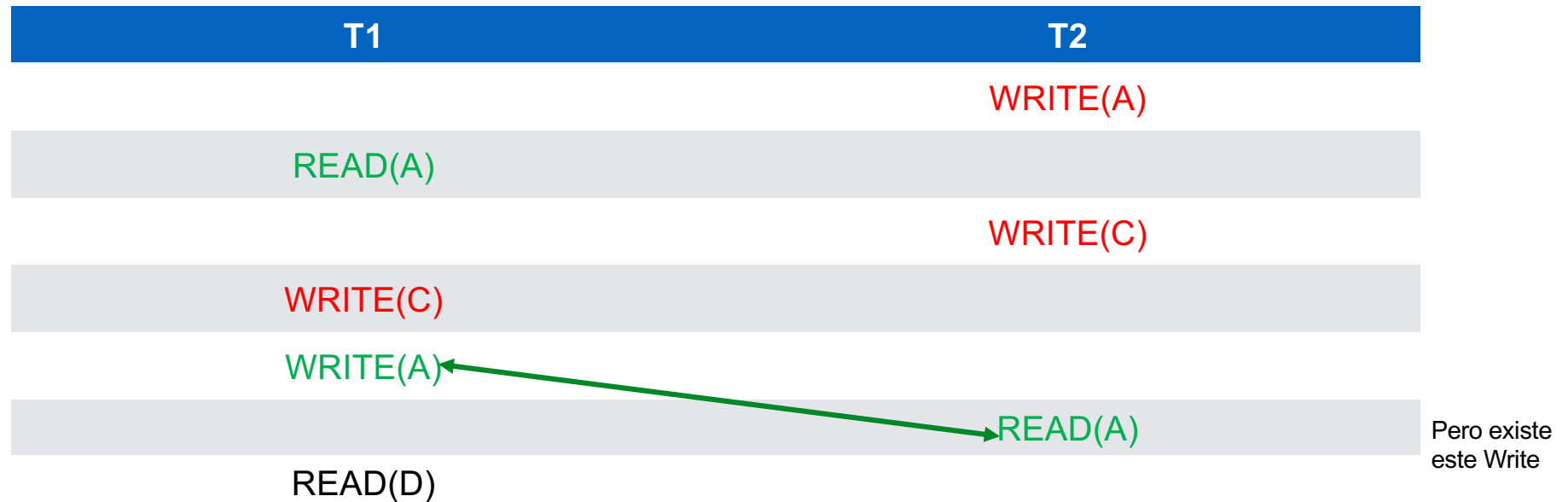
Cuidado!!!



Se podría ?

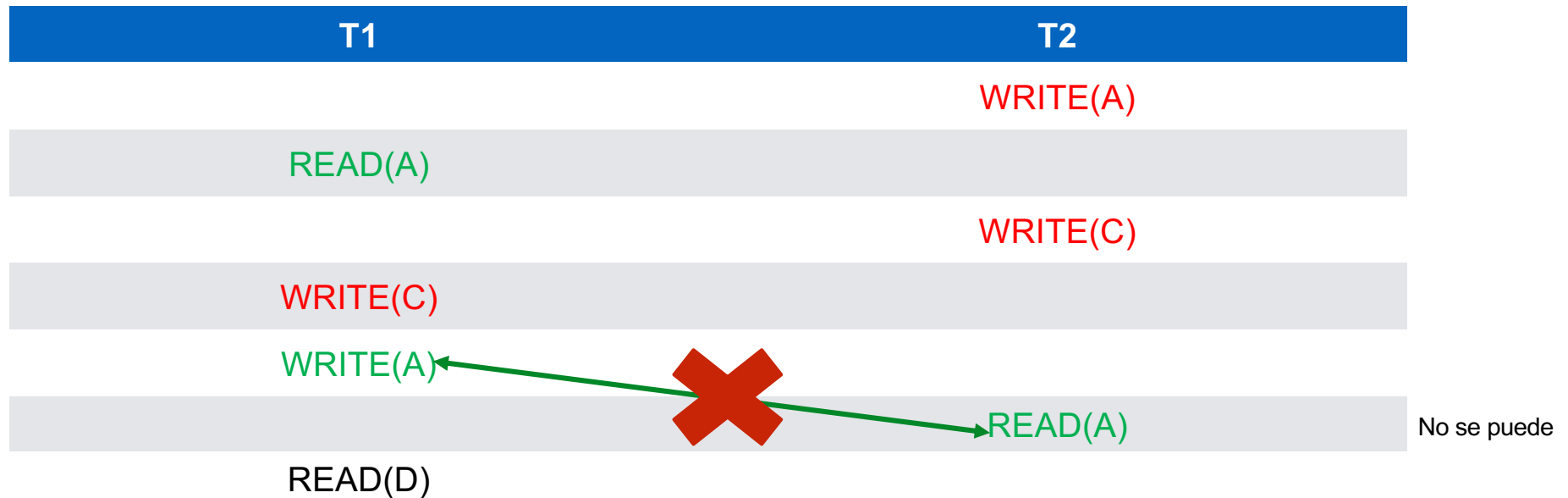
Permutaciones no permitidas

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

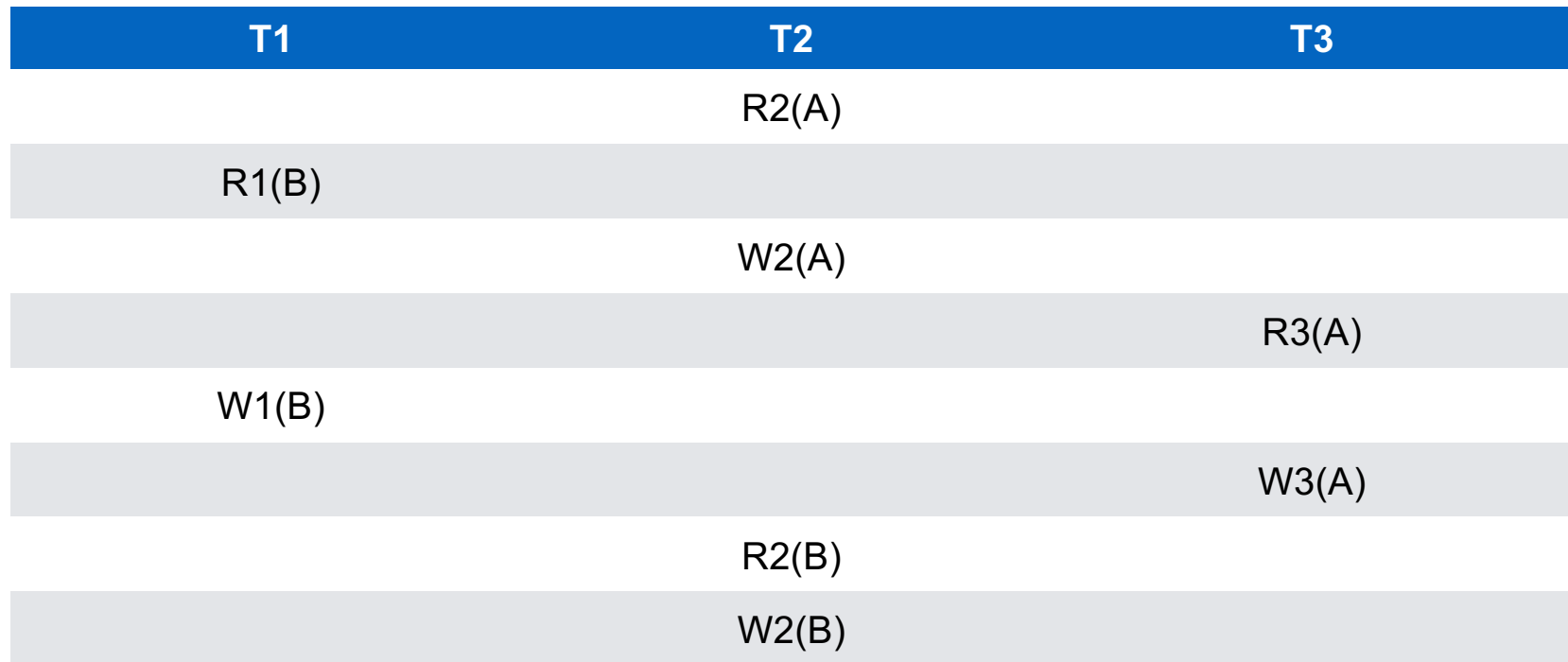
Con este proceso de permutaciones:

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

Permutando a serial

Ejemplo

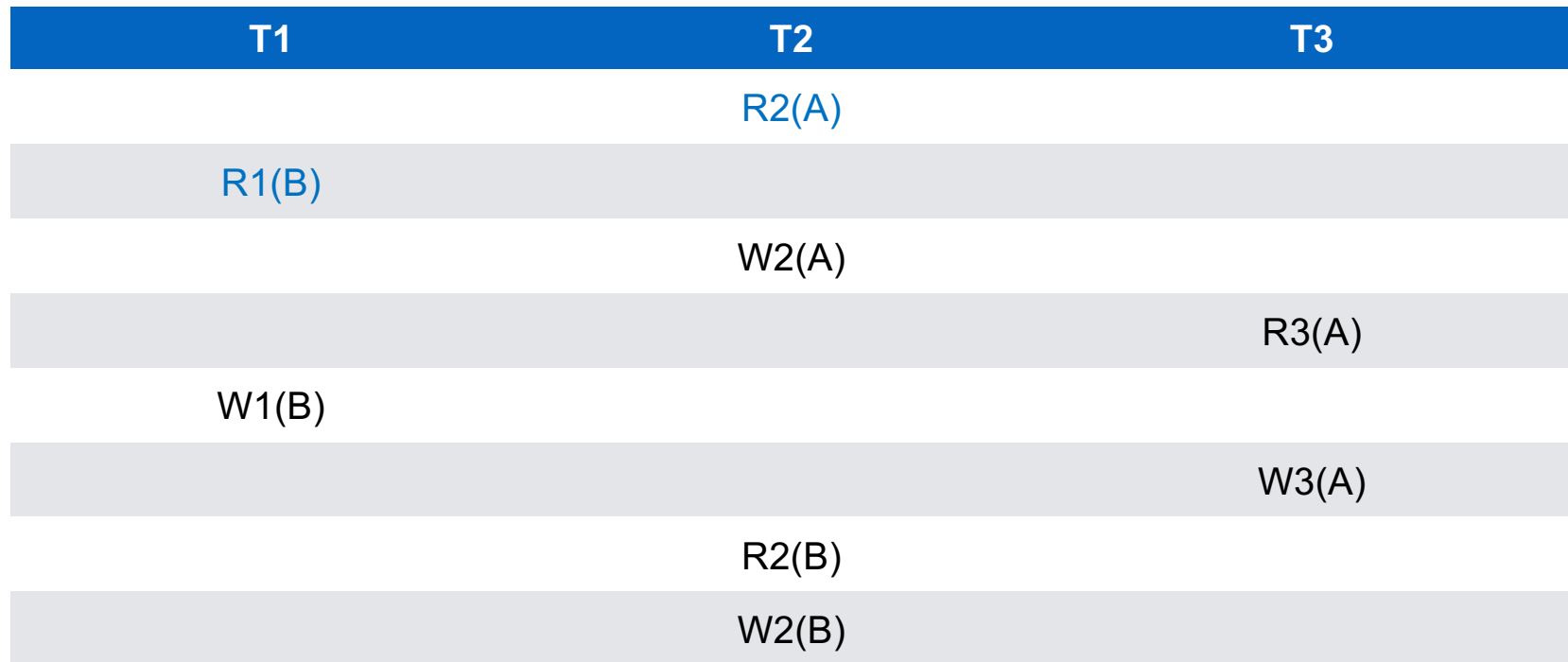
¿Es serializable?



Permutando a serial

Ejemplo

Permutemos!



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 conflict serializable serializable

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

Grafo de precedencia y serialización

Grafo de precedencia

Un grafo de precedencia es un grafo dirigido y sin ciclos donde los nodos corresponden a instrucciones (transacciones en este caso...)

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

Grafo de precedencia

¿Es *conflict serializable*?

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

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

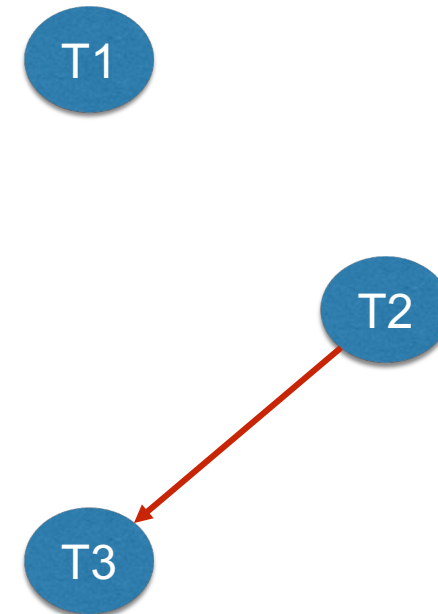
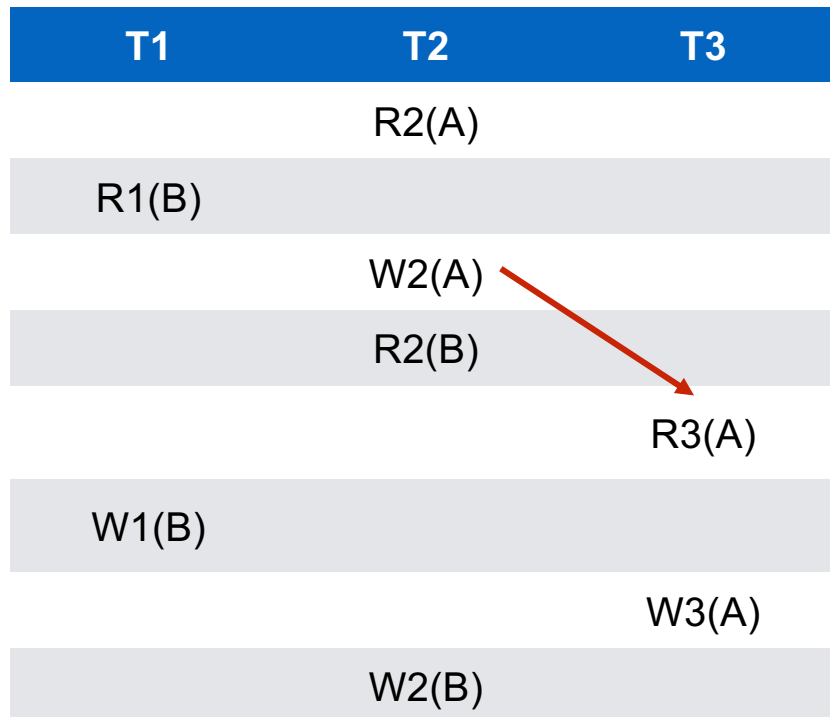
¿Es *conflict serializable*?

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



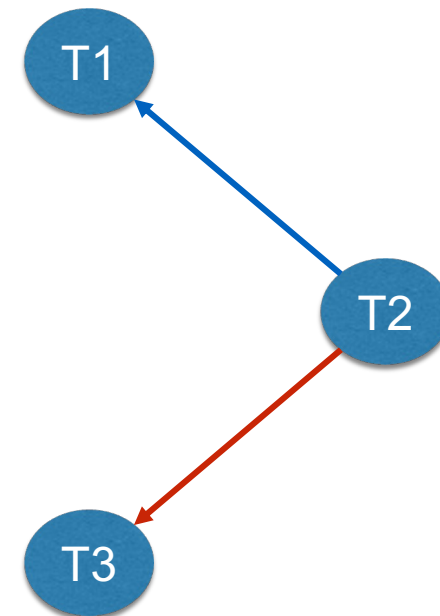
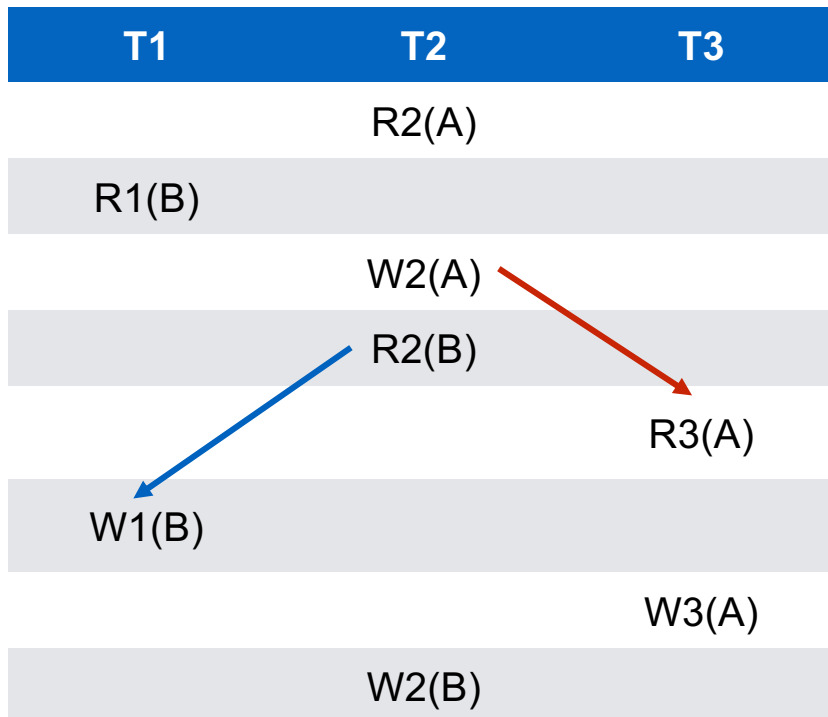
Grafo de precedencia

¿Es *conflict serializable*?



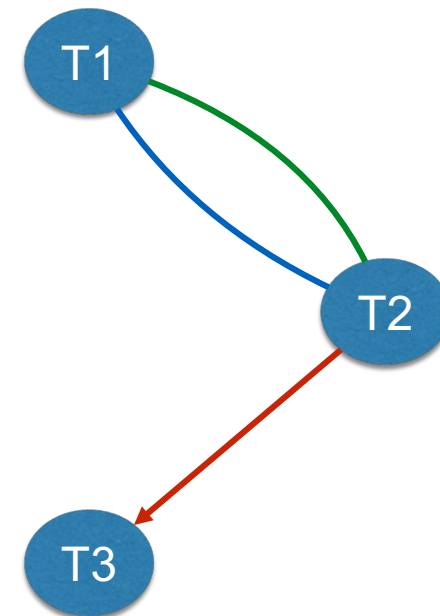
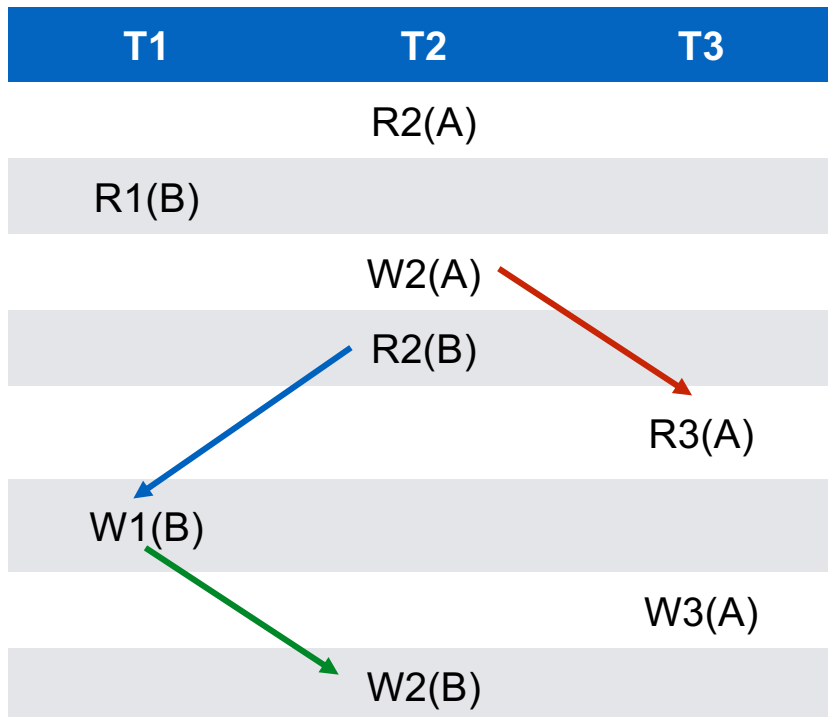
Grafo de precedencia

¿Es *conflict serializable*?



Grafo de precedencia

¿Es *conflict serializable*?

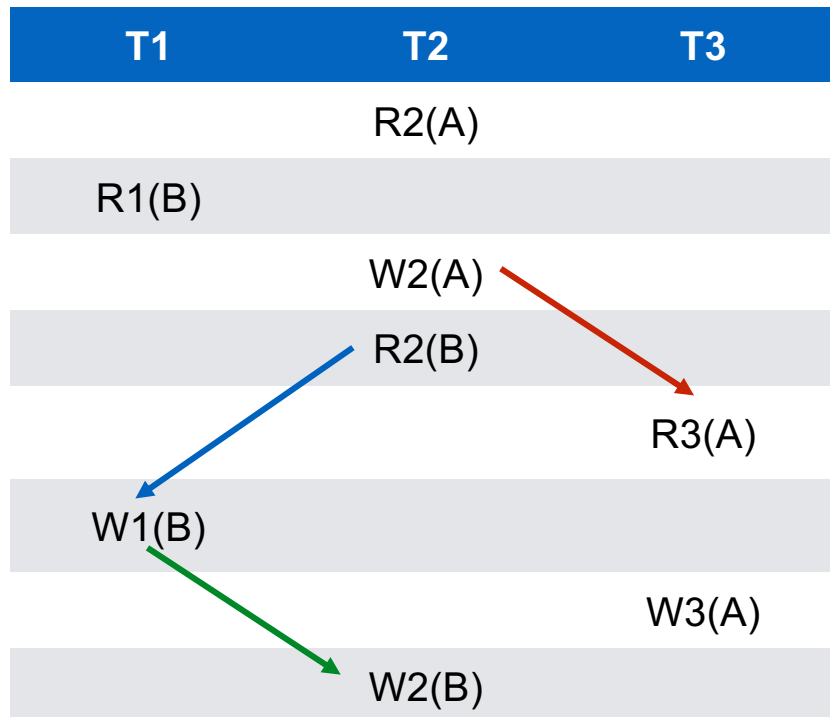


Grafo de precedencia

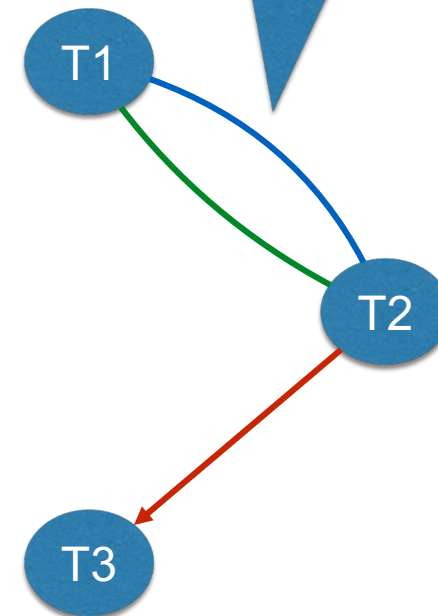
Teorema Un schedule es *conflict serializable* ssi el grafo de precedencia es acíclico

Grafo de precedencia

¿Es *conflict serializable*?



Ciclo == no es conflict serializable



Strict 2PL

Two-Phase Locking

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/modificar un objeto, primero pide un **shared lock** / **exclusive lock** sobre el objeto

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

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

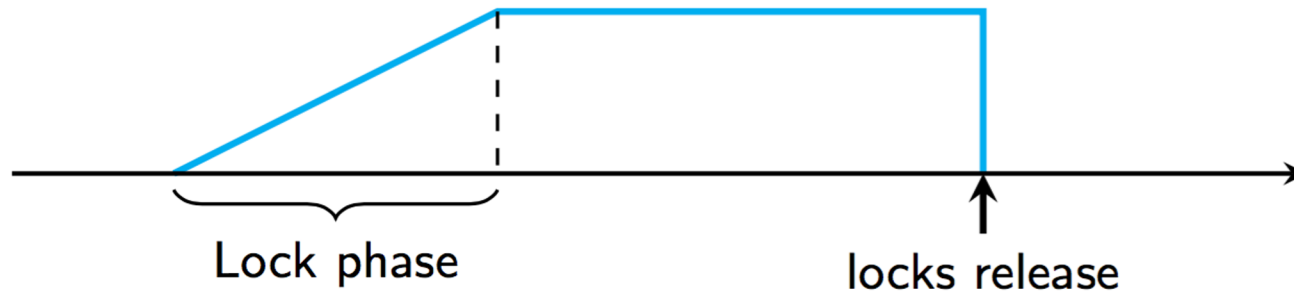
Strict 2PL

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:	T1	T2	Locks de T2:
Shared lock (A)	R(A)		
		W(A)	
		W(B)	
	W(B)		

Cómo funciona 2PL



Cómo funciona 2PL



Cómo funciona 2PL



Cómo funciona 2PL



Cómo funciona 2PL



Problemas con locks

Problemas con locks:

Long Transactions

Long Transactions

Son transacciones que requieren mucho tiempo para completarse debido a la complejidad de las operaciones que realizan, el volumen de datos que manejan, o las interacciones que requieren con otros sistemas o recursos.

Este tipo de transacciones pueden crear problemas significativos en términos de rendimiento, uso de recursos y gestión de bloqueos, lo cual puede afectar negativamente la concurrencia y la eficiencia del sistema.

Long Transactions - Ejemplo

Supongamos que un banco desea realizar un análisis financiero complejo al final del día, que involucra:

- Revisar todas las transacciones del día para cada cuenta, con el objetivo de detectar patrones de fraude.
- Calcular los intereses acumulados durante el día para todas las cuentas de ahorro.
- Actualizar los balances de cada cliente basándose en los resultados del día.

Esta operación podría tomar horas debido a la gran cantidad de datos y las complejas consultas de bases de datos necesarias.

Durante este tiempo, cualquier intento de acceder o modificar los datos de las cuentas afectadas por otros usuarios o procesos puede quedar bloqueado hasta que la transacción larga se complete.

Problemas con locks:

Dead locks

Dead locks

Ocurren cuando dos o más transacciones se encuentran en un estado de espera permanente porque cada una posee un bloqueo en un recurso que la otra transacción intenta bloquear.

En resumen, cada transacción está esperando a que la otra libere su bloqueo, creando así un ciclo de dependencias de bloqueo que nunca se resuelve por sí solo.

Deadlock en 2PL



Deadlock en 2PL

Locks de T1:	T1	T2	Locks de T2:
Shared lock (A)	R(A)		
		W(B)	
		W(A)	
	W(B)		

Deadlock en 2PL



Deadlock en 2PL



Deadlock en 2PL



Deadlock en 2PL



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

Transacciones en SQL

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 mayor actor'
```

```
WHERE nombre = 'Adrian S'
```

```
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 que 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;
```

Lock seguro: La tabla S

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

Lock razonable: Tuplas de S con rating = 8

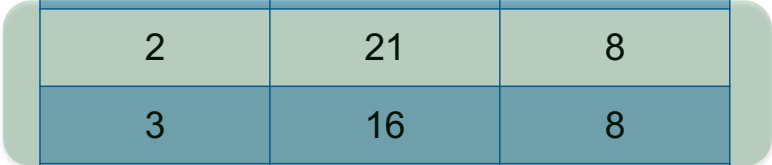
SQL y transacciones

Granularidad y "fantasmas"

T1

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

shared lock

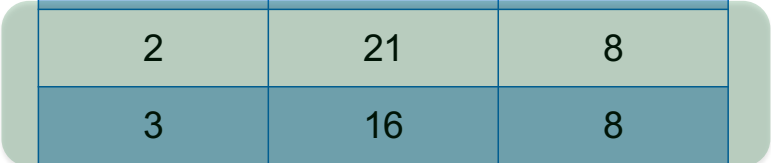


sid	age	rating
1	34	9
2	21	8
3	16	8
4	26	10

T2

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

shared lock



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

"fantasma"




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 (granularidad)

SET TRANSACTION ISOLATION LEVEL <level> READ
ONLY

SET TRANSACTION ISOLATION LEVEL <level> READ
WRITE

SQL y transacciones

Nivel de aislamiento (granularidad)

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 (granularidad)

SET TRANSACTION ISOLATION LEVEL Level READ
ONLY

SET TRANSACTION ISOLATION LEVEL 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
```