

# Bases de Datos

Clase 9: Transacciones y Locks

# Hasta ahora

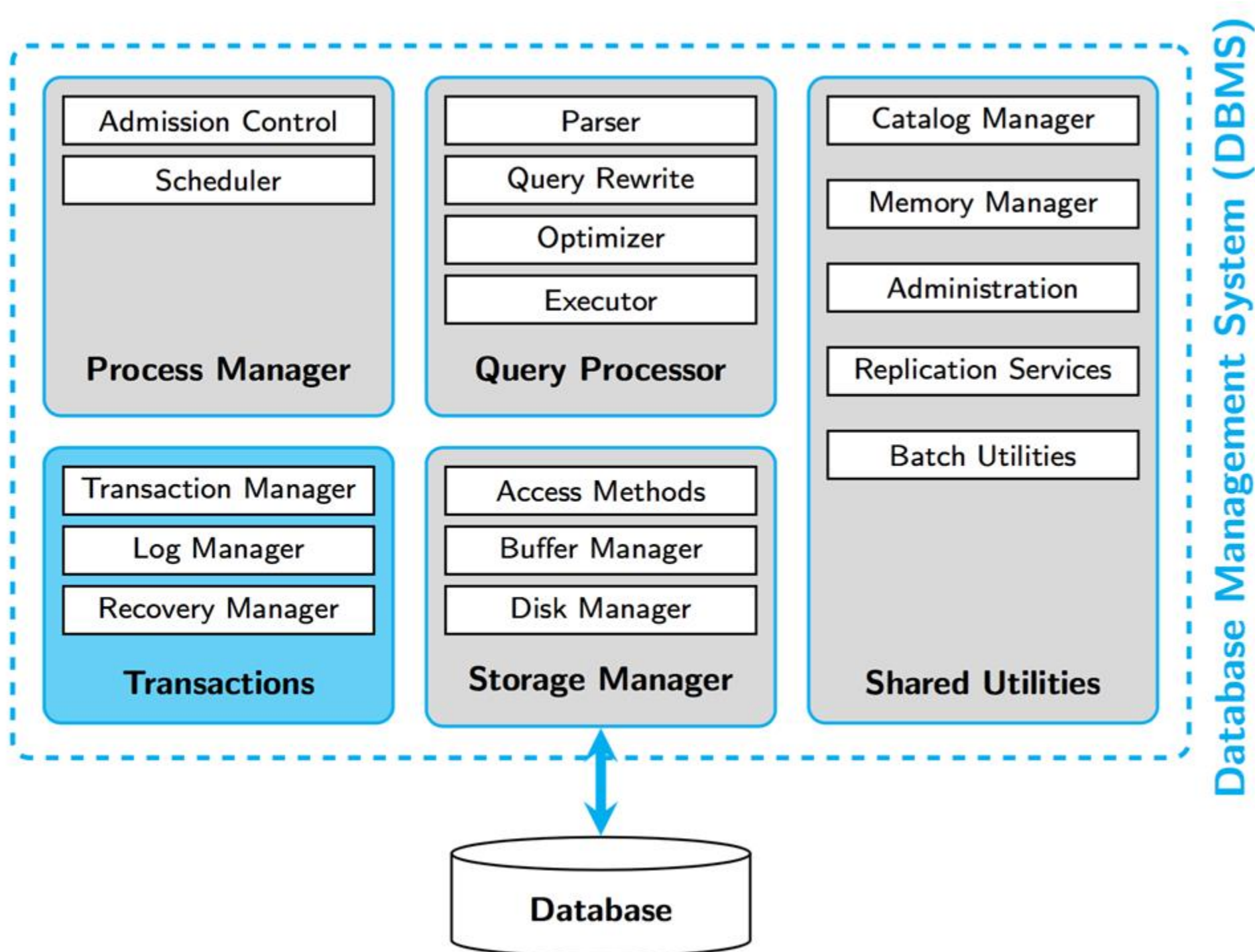
Solo nos hemos dedicado a la Base de datos

- Diseño
- Estructura lógica
- Lenguaje de consulta SQL

¿Y qué pasa con es DBMS?

Desde esta clase nos dedicaremos a eso...

# DBMS



# Transacciones

# ¿Que es una transacción?

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

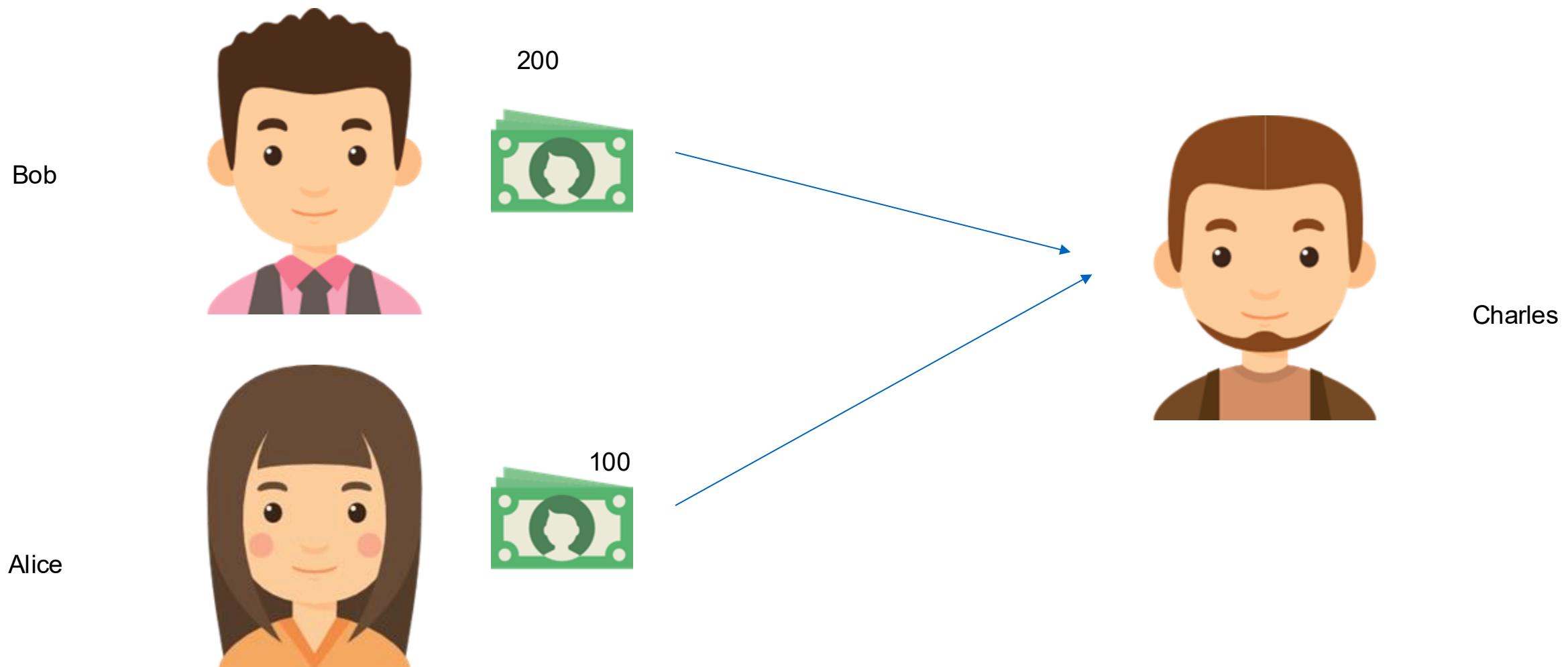
- Transferencias de dinero entre cuentas
- Compra por internet
- Inscribir un curso

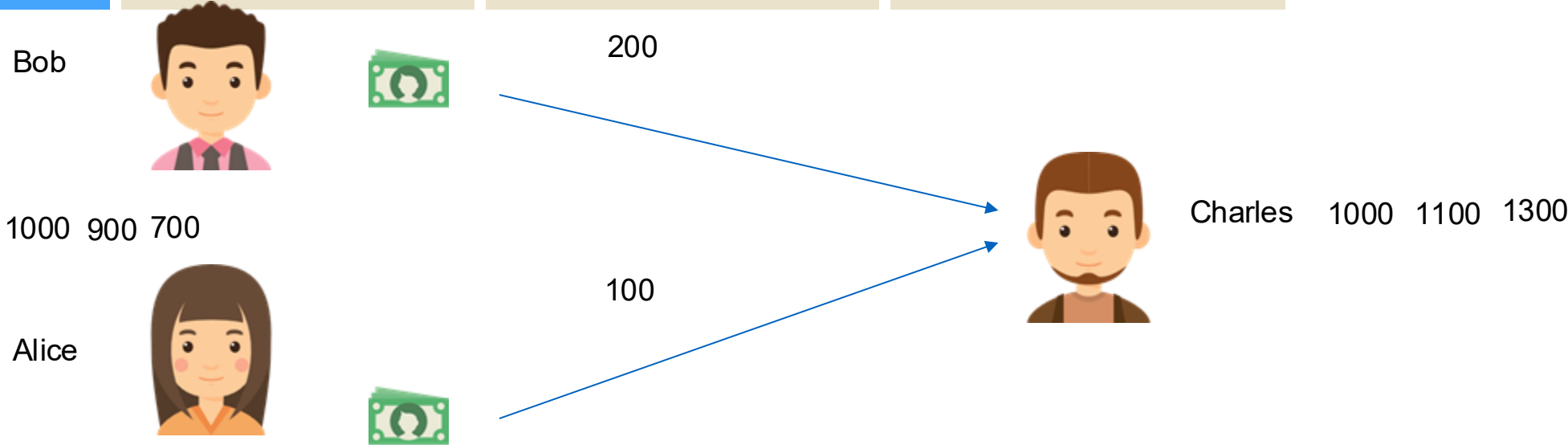
El principal propósito de las transacciones es garantizar la integridad de los datos y la consistencia del estado de la base de datos, incluso en presencia de fallos y errores

# Motivación

Supongamos que Alice y Bob tienen una cuenta bancaria en común y cada uno le va a transferir a Charles

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





	Proceso Alice	Proceso Bob	Saldo Cuenta A & B	Saldo Cuenta C
➡	READ(saldoAB, x)		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

Pero el acceso es CONCURRENTE opción 1

Proceso Alice	Proceso Bob	Saldo Cuenta A & B	Saldo Cuenta C
READ(saldoAB, x)		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



Pero el acceso es CONCURRENTE opción 2

Proceso Alice	Proceso Bob	Saldo Cuenta A & B	Saldo Cuenta C
READ(saldoAB, x)		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)			<b>1100</b>

¡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.

El DBMS mezcla las operaciones para optimizar la ejecución y al mismo tiempo mantiene las transacciones

# ¿Solución?

Agrupamos las operaciones de modo que se ejecuten como una sola, se hace entera o no se hace

Proceso Alice	Proceso Bob	Saldo Cuenta A & B	Saldo Cuenta C
Comienzo transacción T1		1000	1000
READ(saldoAB, x)			
WRITE(saldoAB, x - 100)			
READ(saldoC, x)			
WRITE(saldoC, x + 100)			
Fin T1		900	1100
Comienzo transacción T2		900	1100
READ(saldoAB, y)			
WRITE(saldoAB, y - 200)			
READ(saldoC, y)			
WRITE(saldoC, y + 200)			
Fin T2		700	1300

# Pero si todas las transacciones son concurrentes!!!

Proceso Alice	Proceso Bob	Saldo Cuenta A & B	Saldo Cuenta C
Comienzo transacción T1	Comienzo transacción T2	1000	1000
READ(saldoAB, x)	READ(saldoAB, y)		
WRITE(saldoAB, x - 100)	WRITE(saldoAB, y - 200)		
READ(saldoC, x)	READ(saldoC, y)		
WRITE(saldoC, x + 100)	WRITE(saldoC, y + 200)		
Fin T1	Fin T2	700	1300

## Entonces?

# ACID



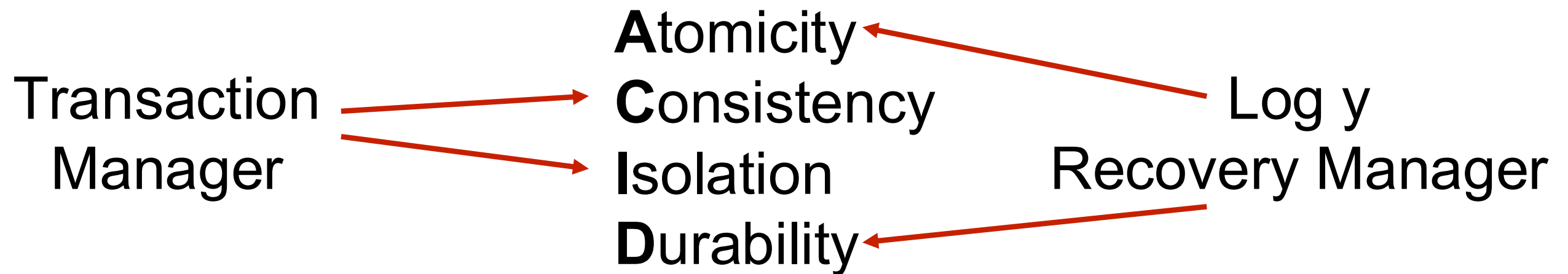
# Transactions



**Transaction Manager** se encarga de asegurar Isolation y Consistency

**Log y Recovery Manager** se encargan de asegurar Atomicity y Durability

# Asegurar las propiedades **ACID**



# Sobre Transactions



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

# 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.



# Sin ACID

¿Que ocurre si no se siguen las propiedades ACID?

Caso: Se corta la luz y la transacción quedó en la mitad

## **Sin Atomicity, Durability:**

- 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 Consistency:**

- La base de datos viola las restricciones momentáneamente
- Al ejecutar una transacción, queda la BD que no cumple con las restricciones

## **Sin Isolation:**

- El sistema de base de datos planifica el orden de operaciones
- Resultado no es igual a haber corrido transacciones en serie

# 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.

# Definamos operaciones

Read(X) o R(X)

Lectura del  
elemento X

Write(X) o W(X)

Escritura del  
elemento X

Abort o A

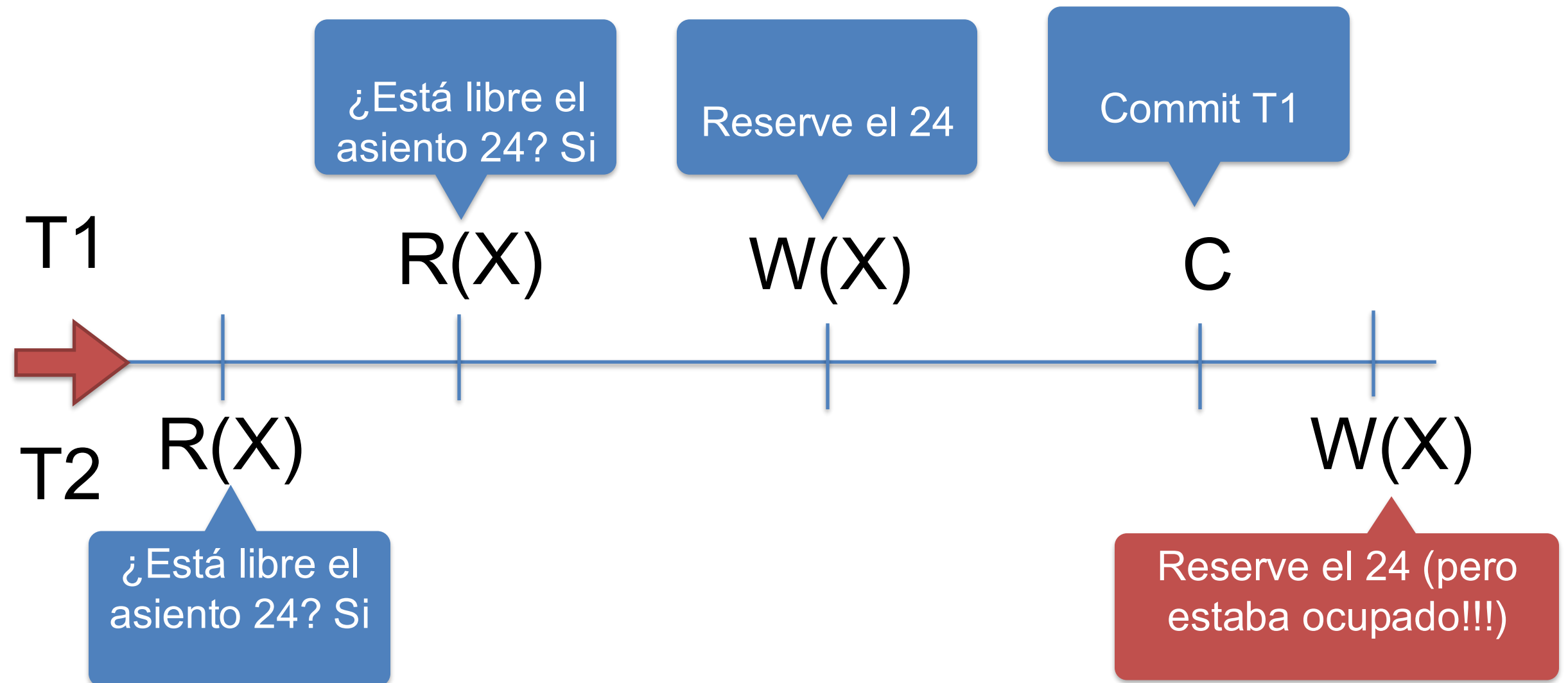
Abortar  
transacción

Commit o C

Finalizar  
transacción

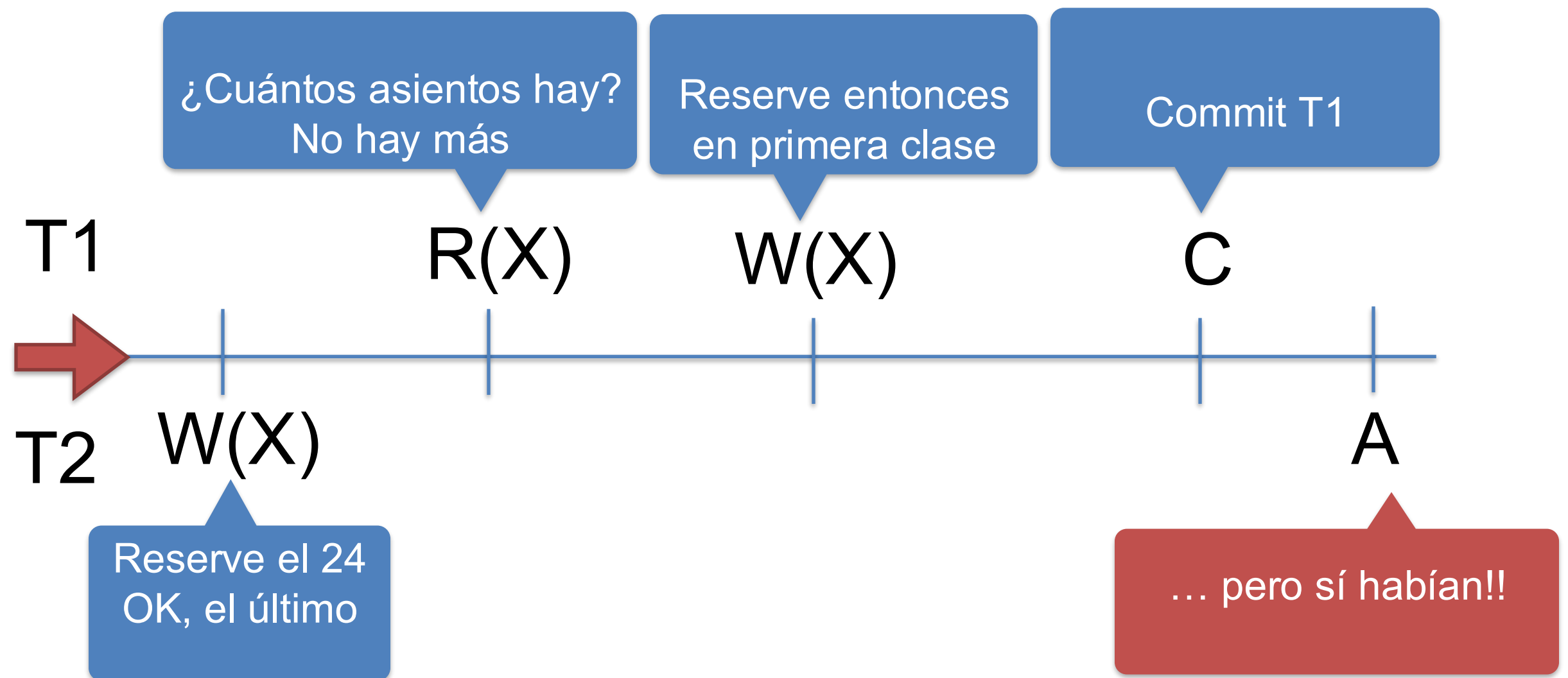
# Ejemplo de actualización perdida (WW)

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



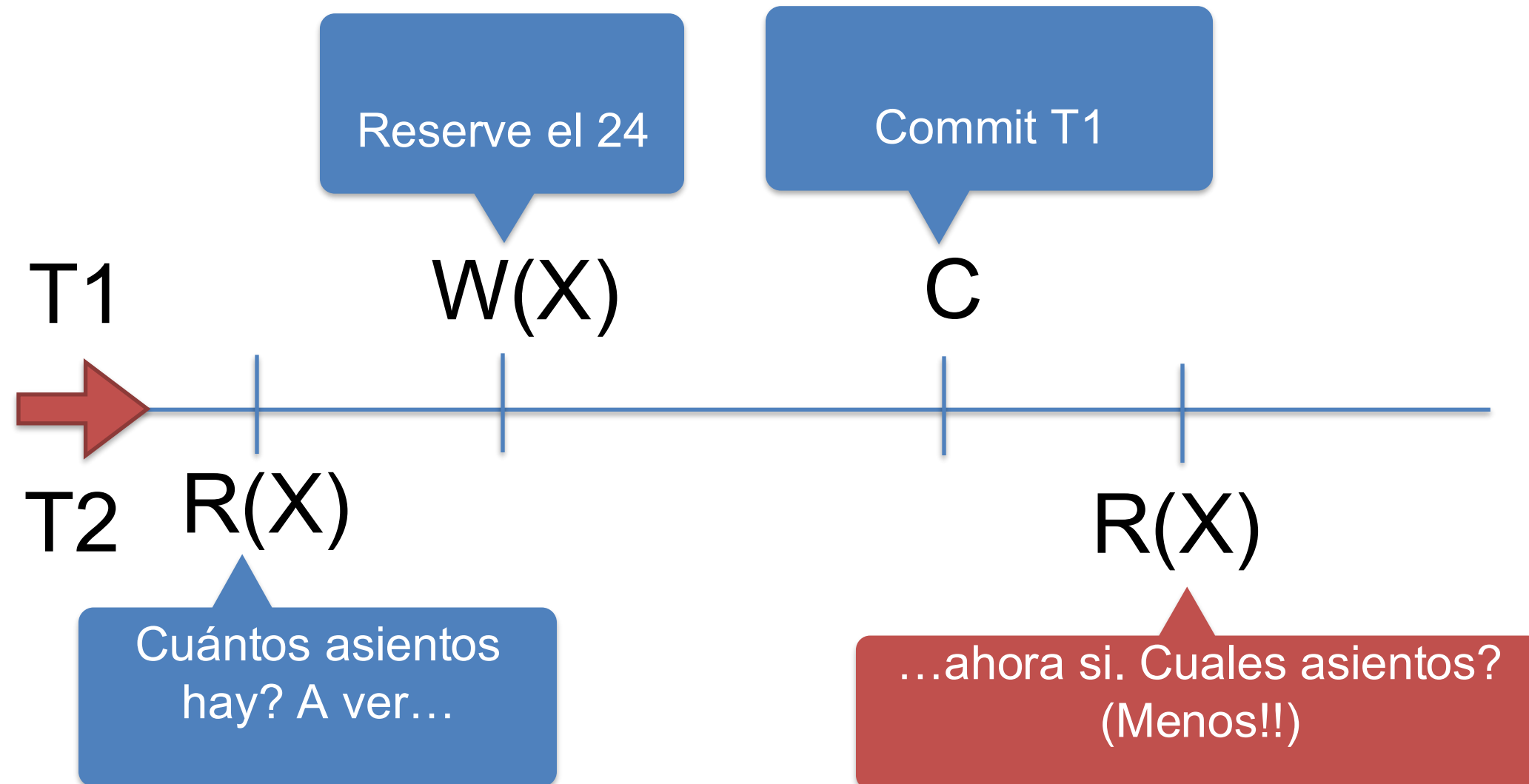
# Ejemplo de lectura sucia (WR)

“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.”



¿Qué reglas sigue  
entonces el DBMS?

Schedules

# ¿Qué 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 1

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,x)
	$x := x + 200$
	WRITE(B,x)
	READ(B,y)
	$y := y * 3$
	WRITE(B,y)

# Schedule 2

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)

¿Son equivalentes?

# Schedule 2

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 serial

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

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

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)

Las operaciones para A en x e y no afectan al orden de x e y de B, entonces S 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 READ(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)

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)



## Notación

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

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

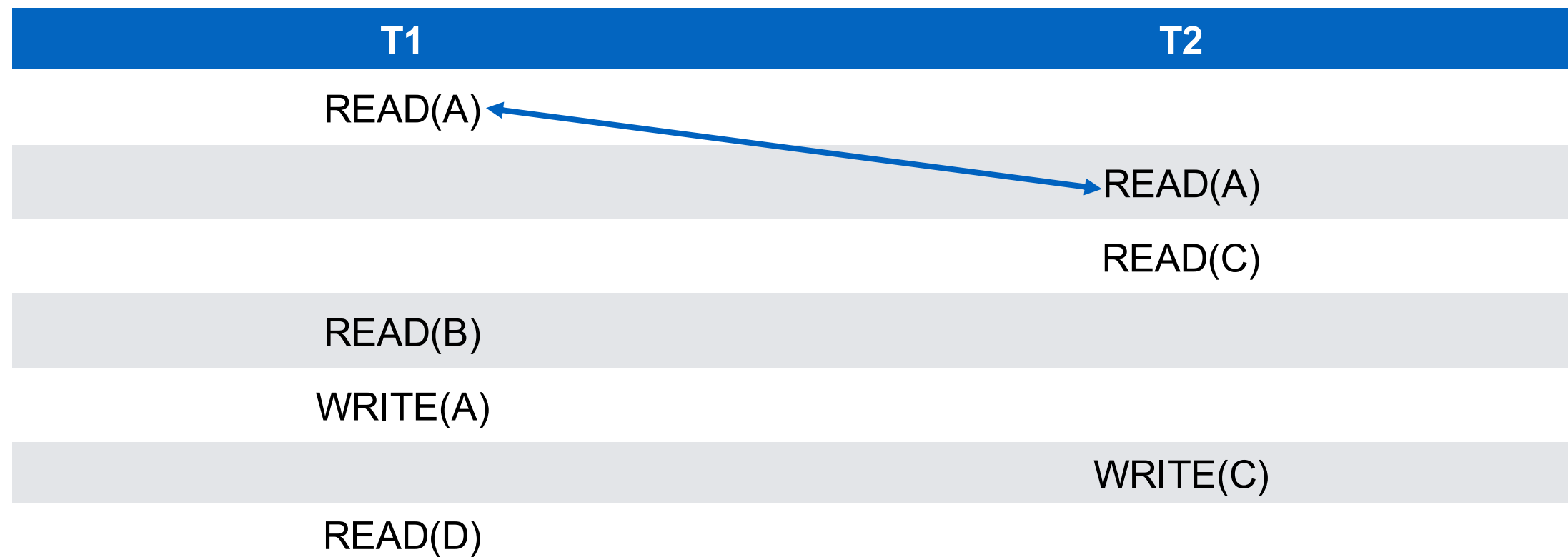
# Acciones No Conflictivas

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

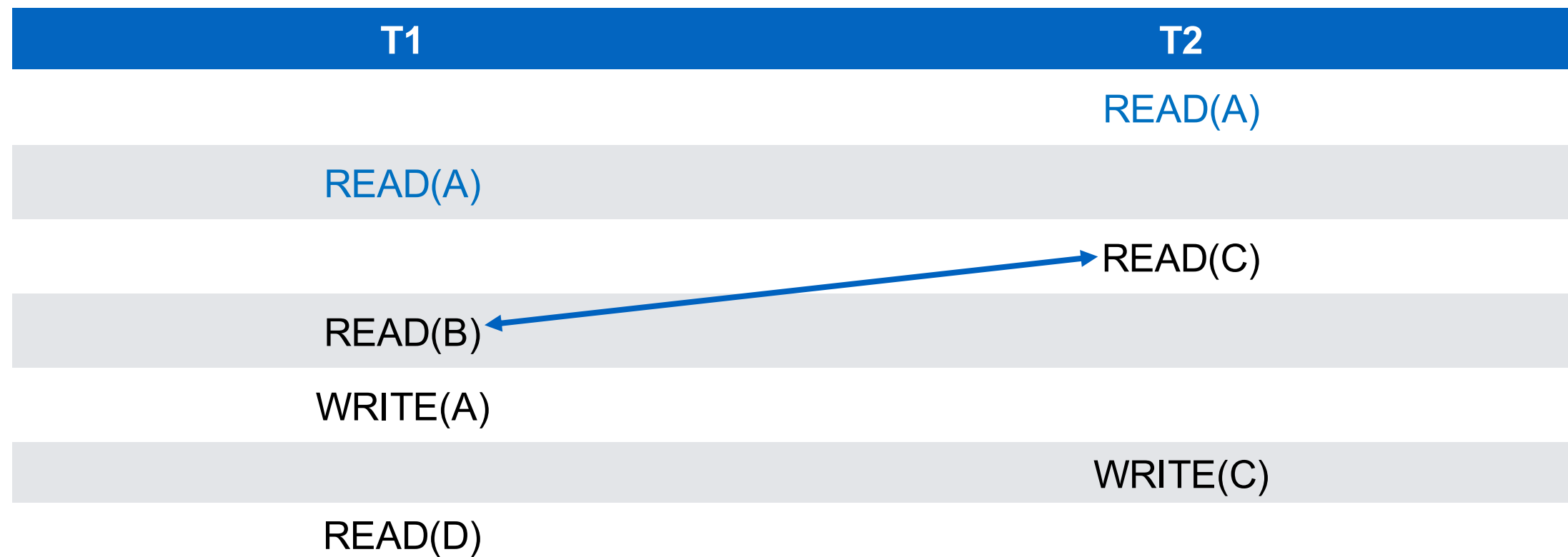
- $R_i(X), R_j(Y)$
- $R_i(X), W_j(Y)$  con  $X \neq Y$
- $W_i(X), R_j(Y)$  con  $X \neq Y$
- $W_i(X), W_j(Y)$  con  $X \neq Y$

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

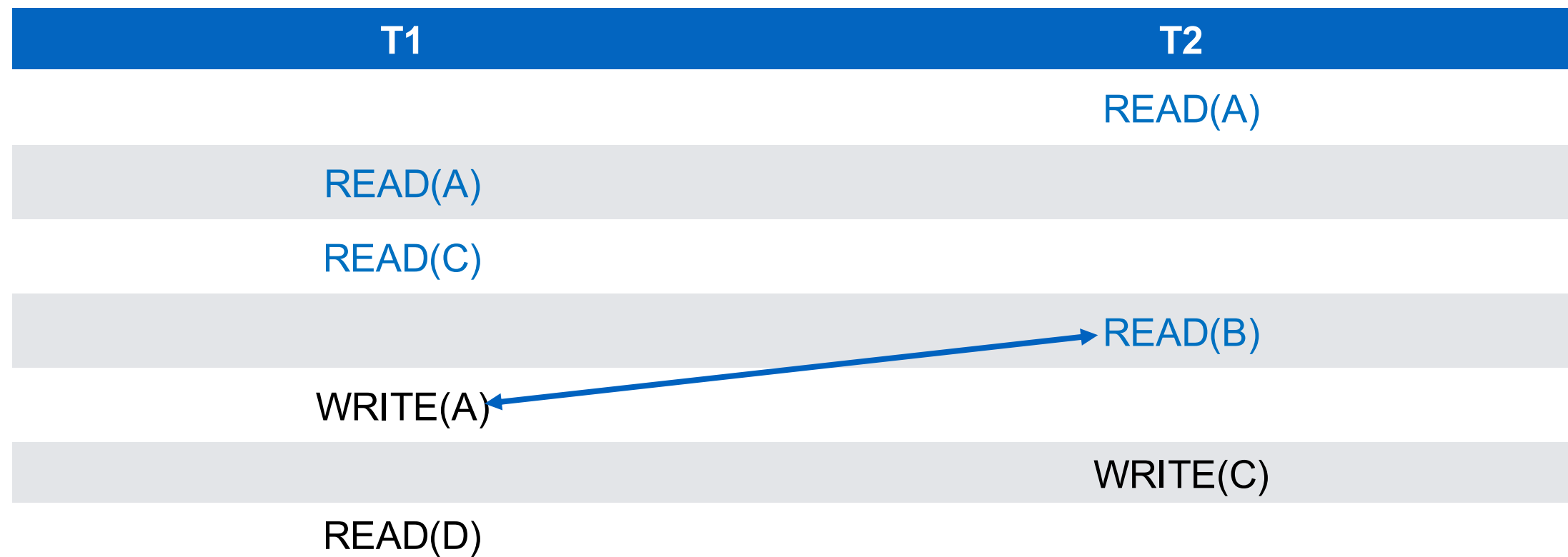
# Permutaciones permitidas



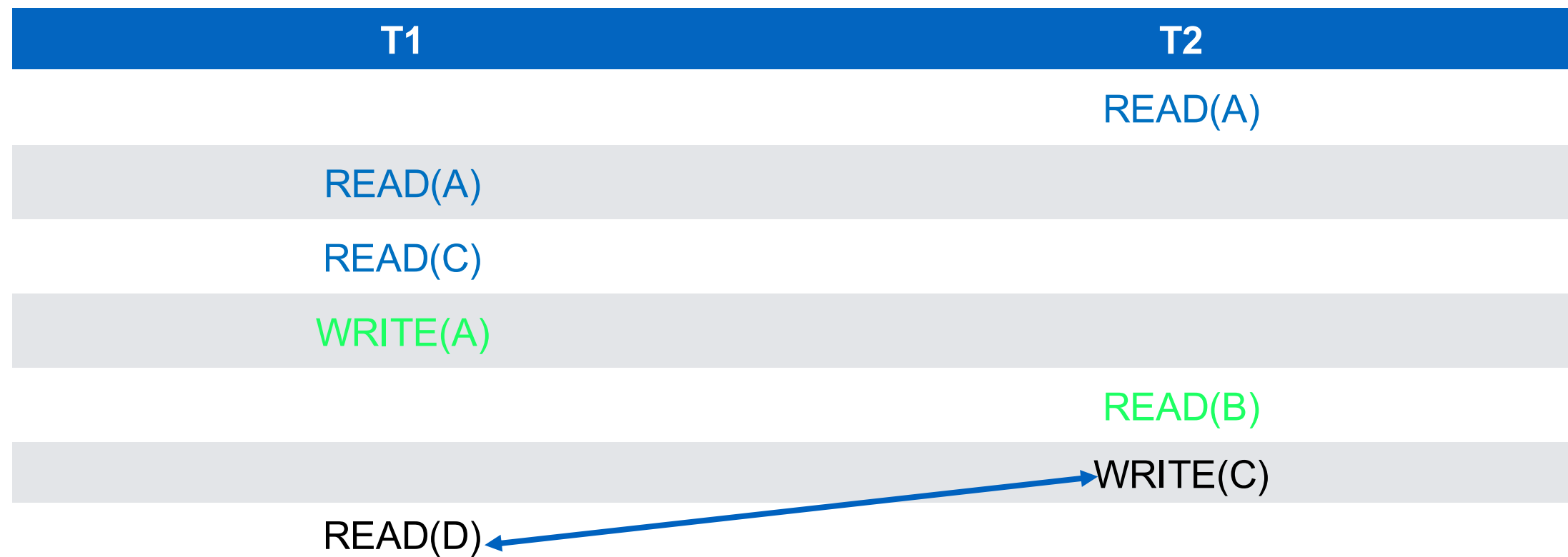
# Permutaciones permitidas



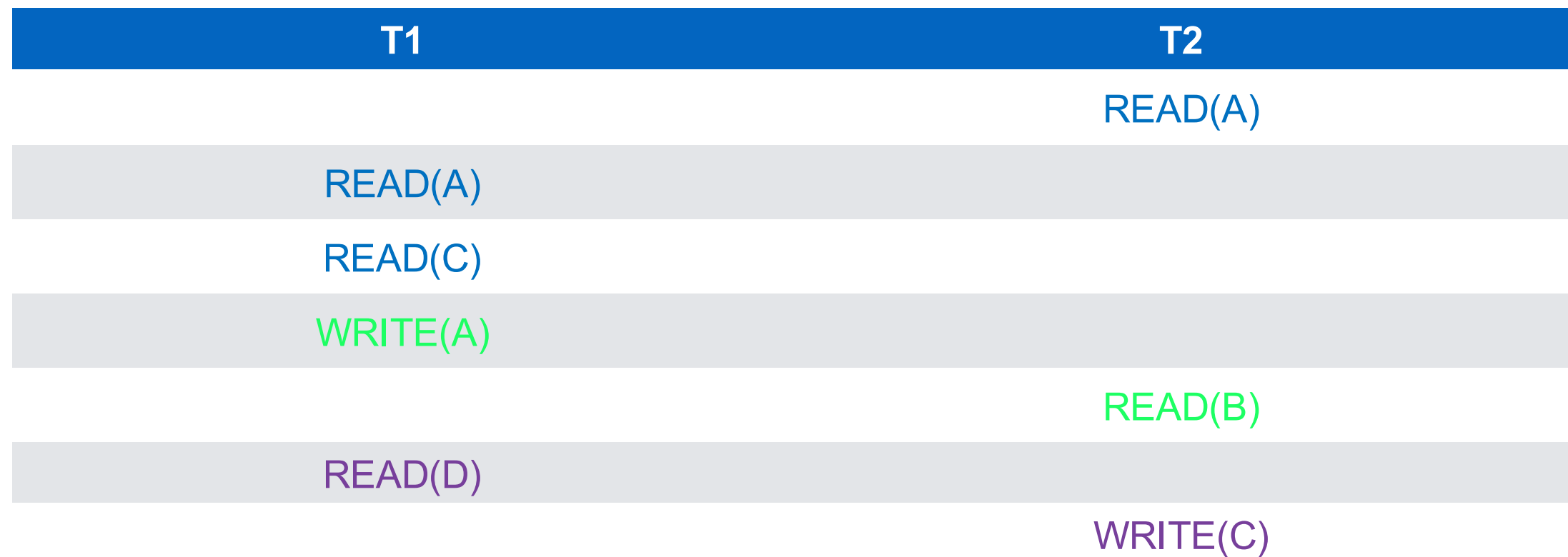
# Permutaciones permitidas



# Permutaciones permitidas



# Permutaciones permitidas



# Acciones Conflictivas

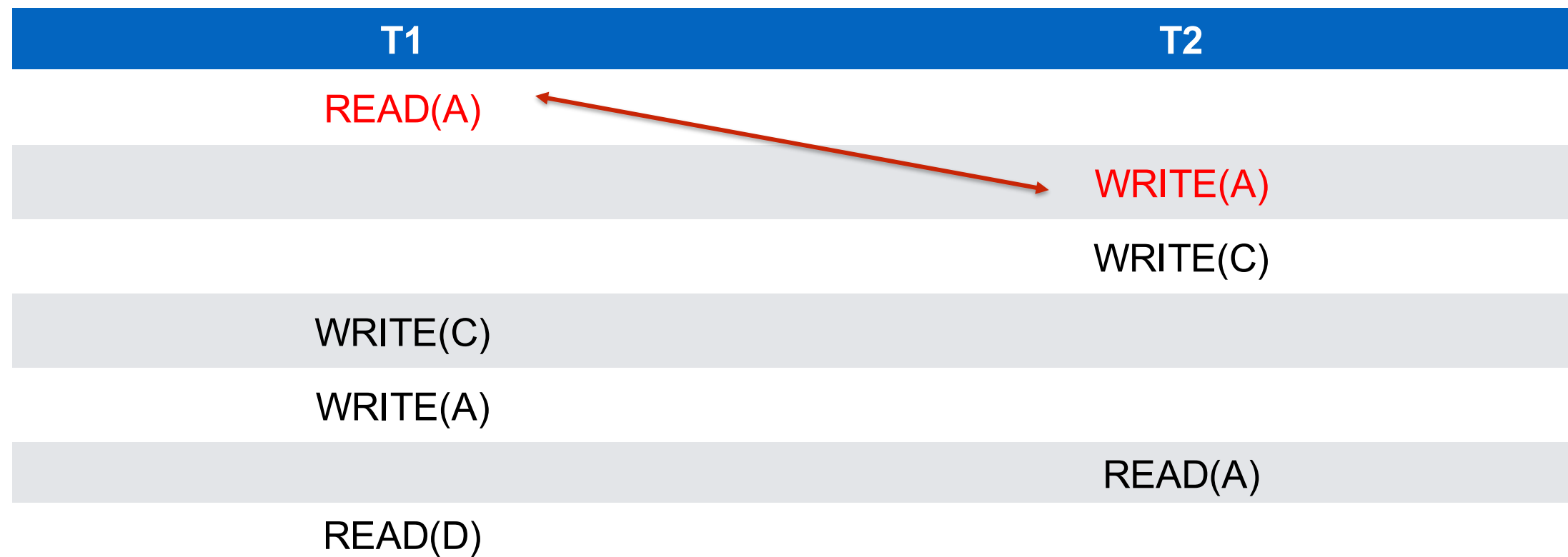
Las siguientes acciones son conflictivas para dos transacciones distintas  $i, j$ :

- 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  $X$ .

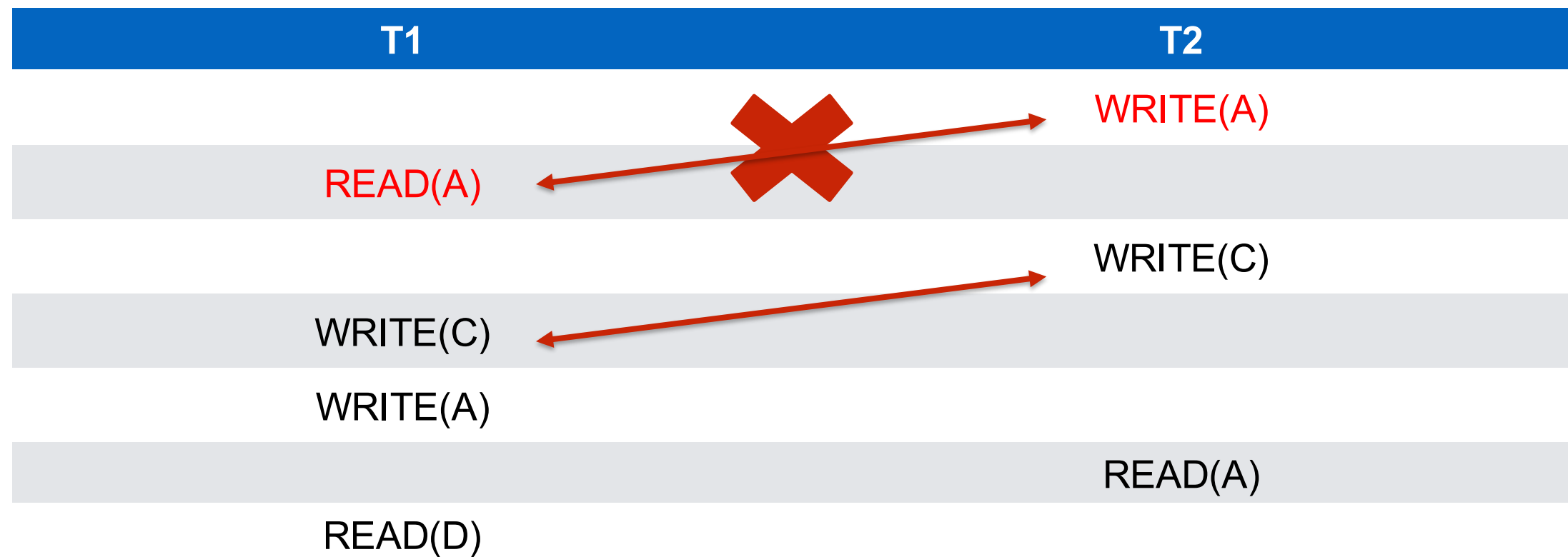
¡No podemos cambiar su orden en un **schedule** a la ligera!



# Permutaciones no permitidas

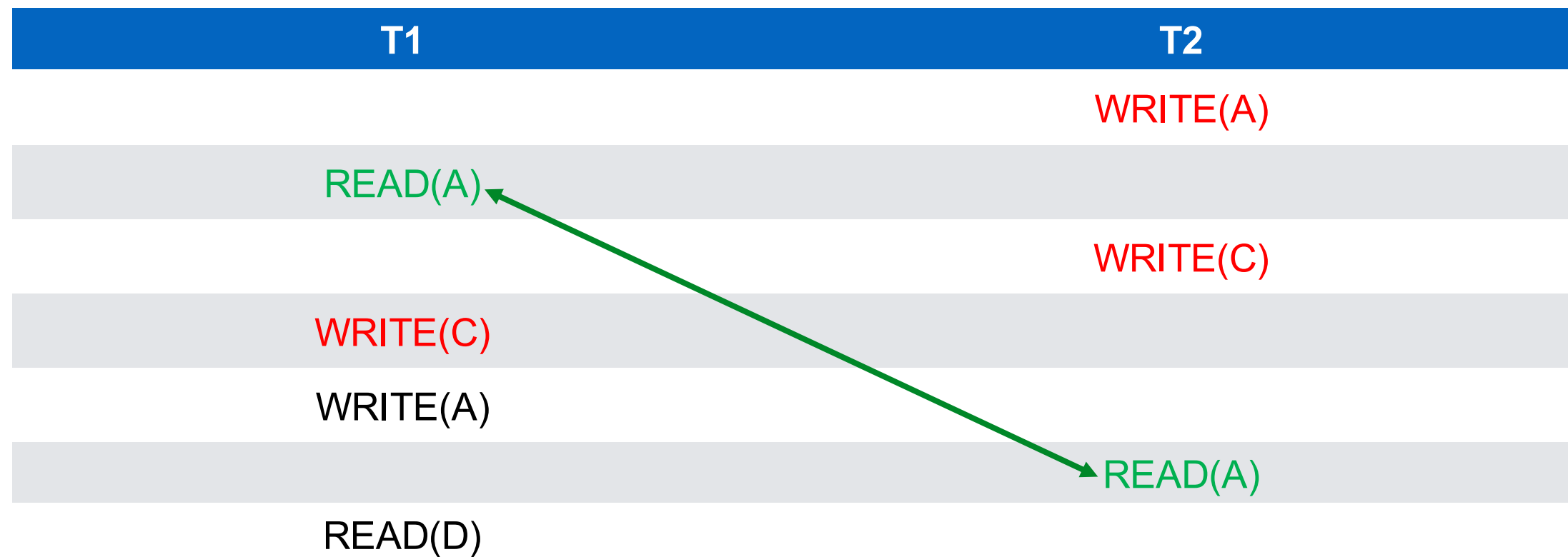


# Permutaciones no permitidas



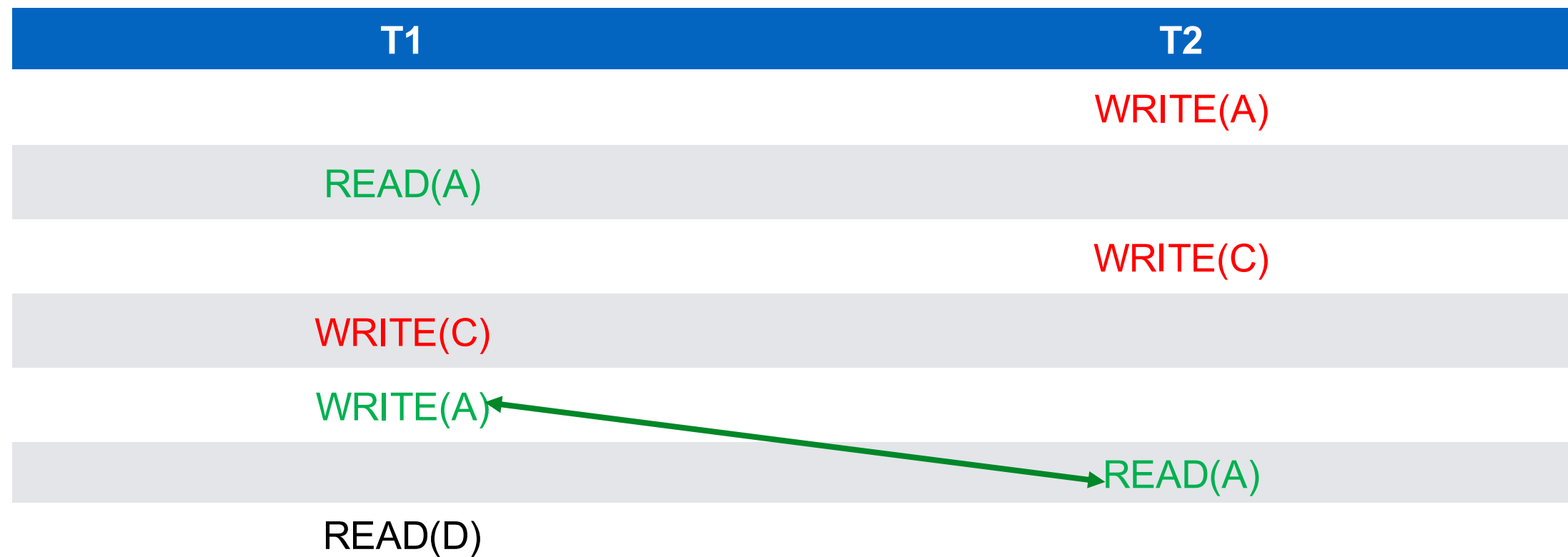
# Permutaciones no permitidas

Cuidado!!!



# Permutaciones no permitidas

Cuidado!!!



# Permutaciones no permitidas

Cuidado!!!



# Acciones no Conflictivas

Puedo permutar un par de operaciones consecutivas si:

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

# Conflict serializable

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

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

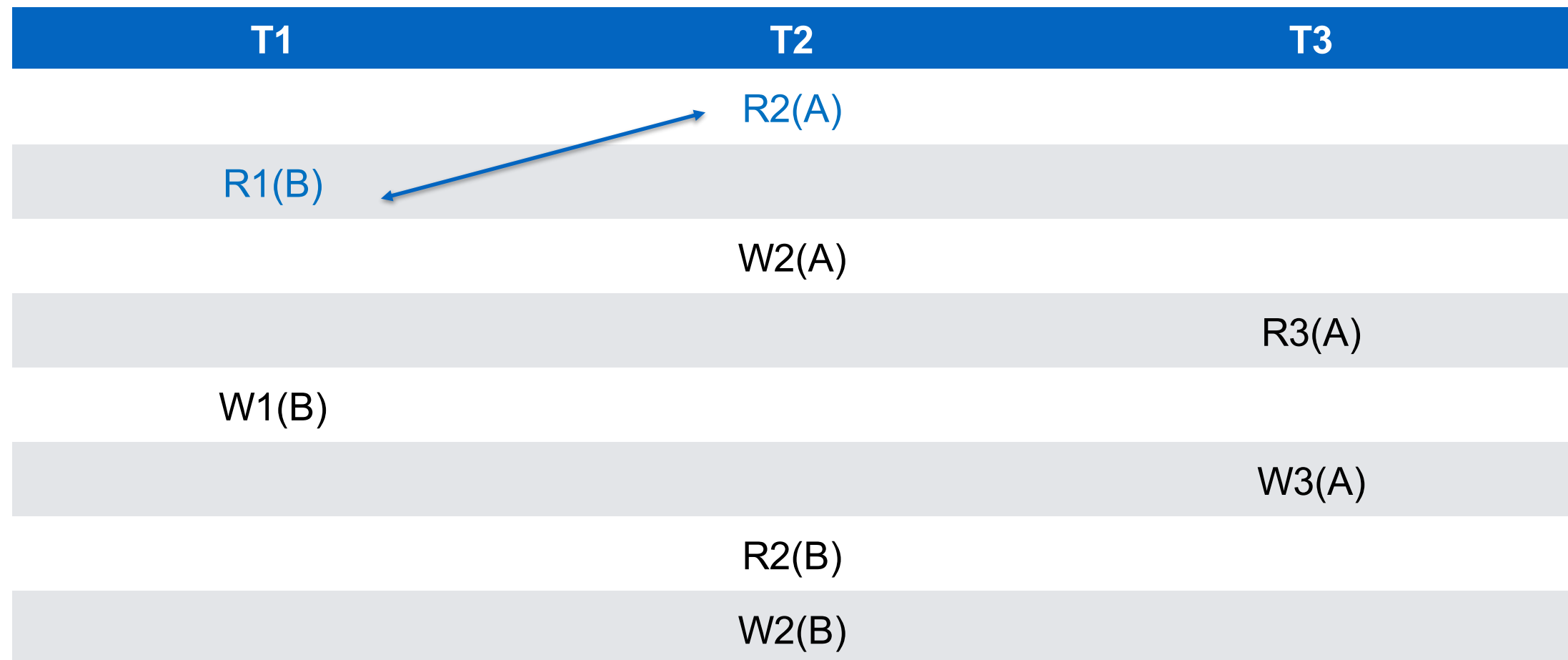


# Permutando a serial

Ejemplo

¿Es serializable?

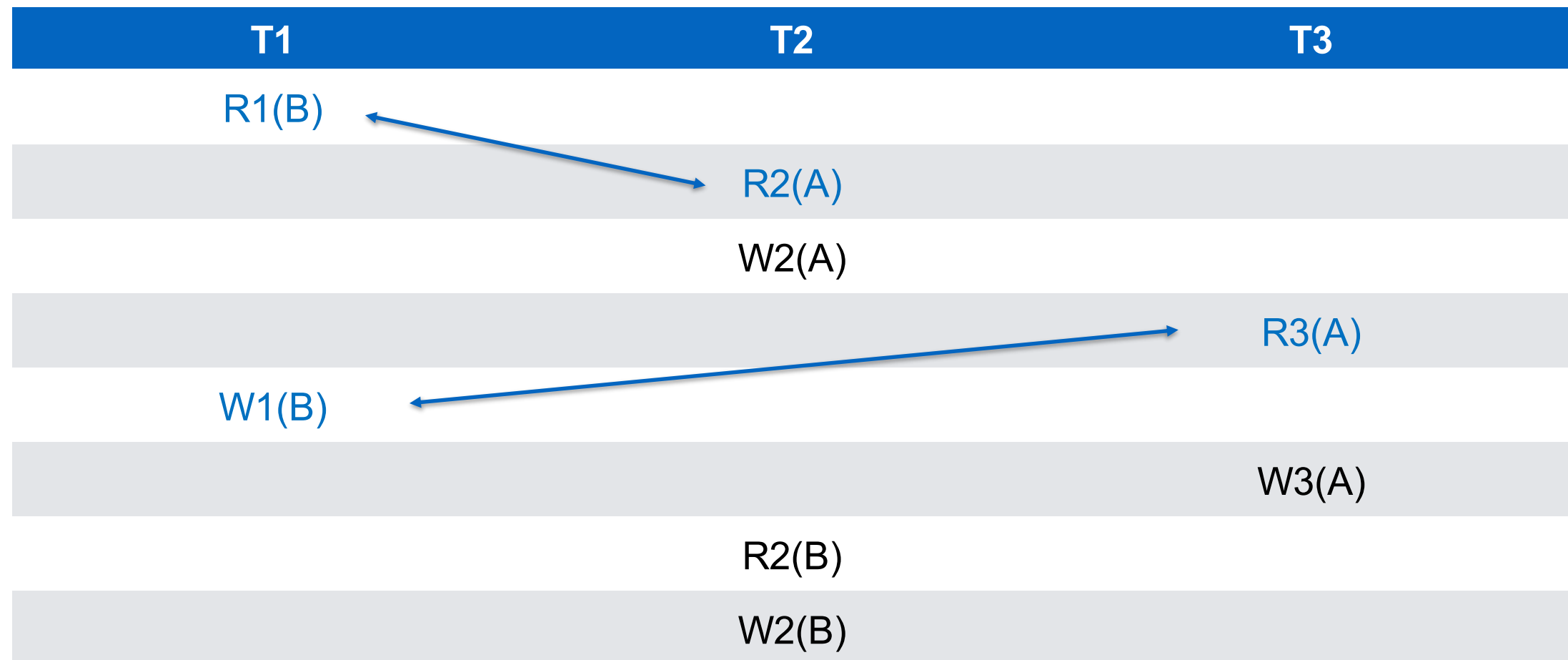
Permutemos!



# Permutando a serial

Ejemplo

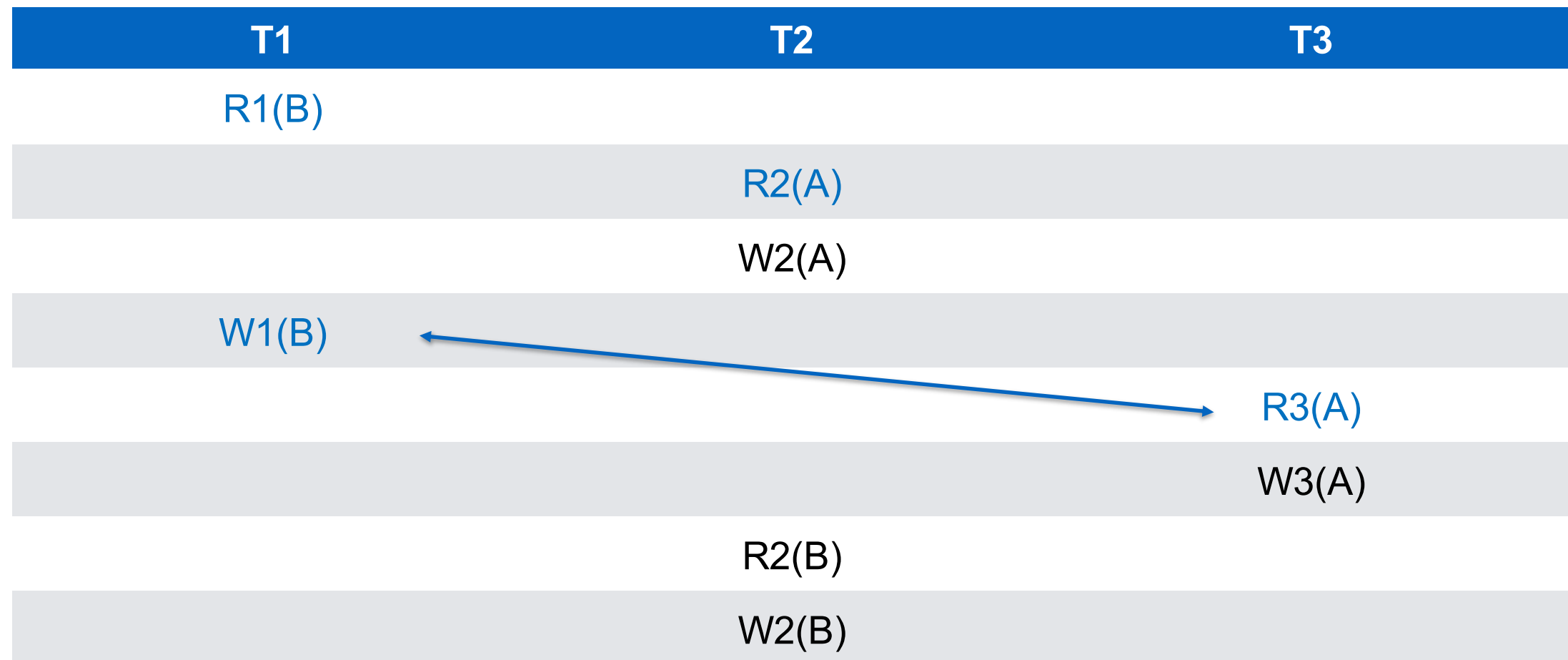
Permutemos!



# Permutando a serial

Ejemplo

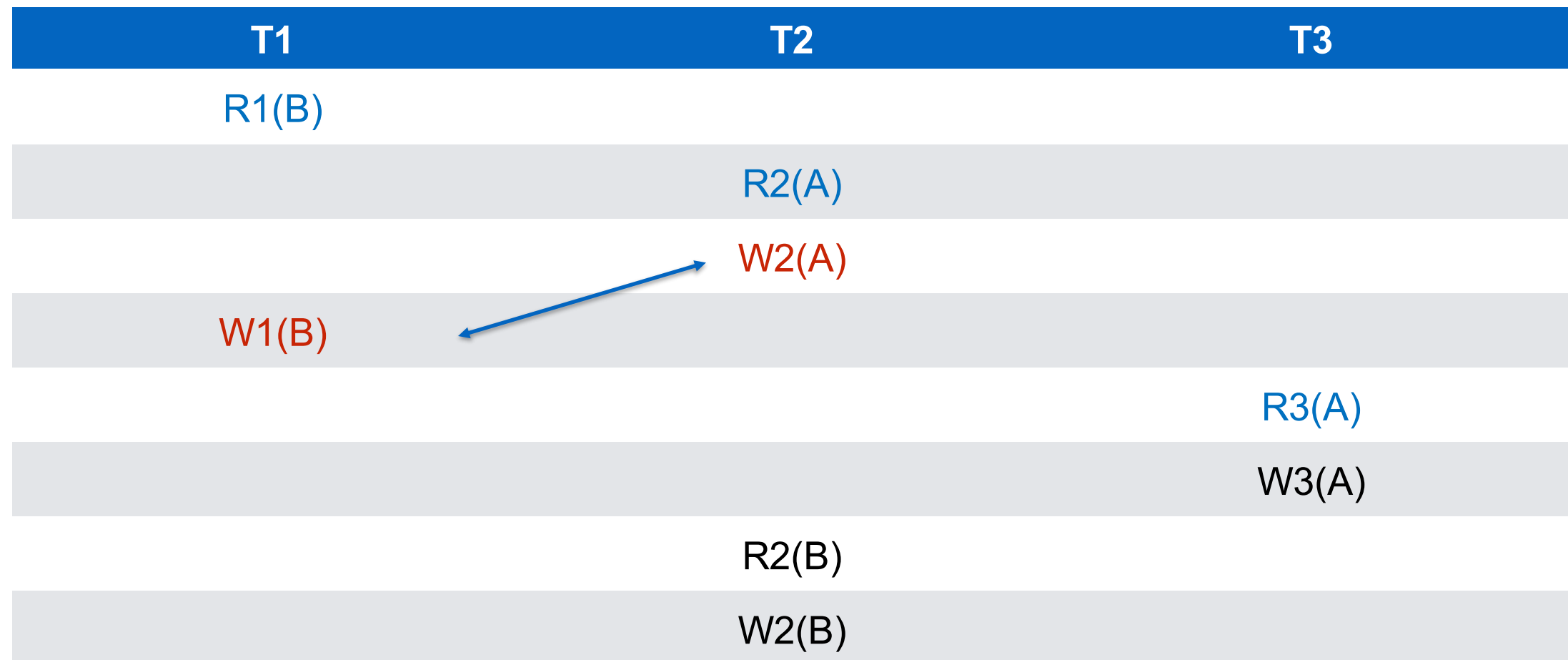
Permutemos!



# Permutando a serial

Ejemplo

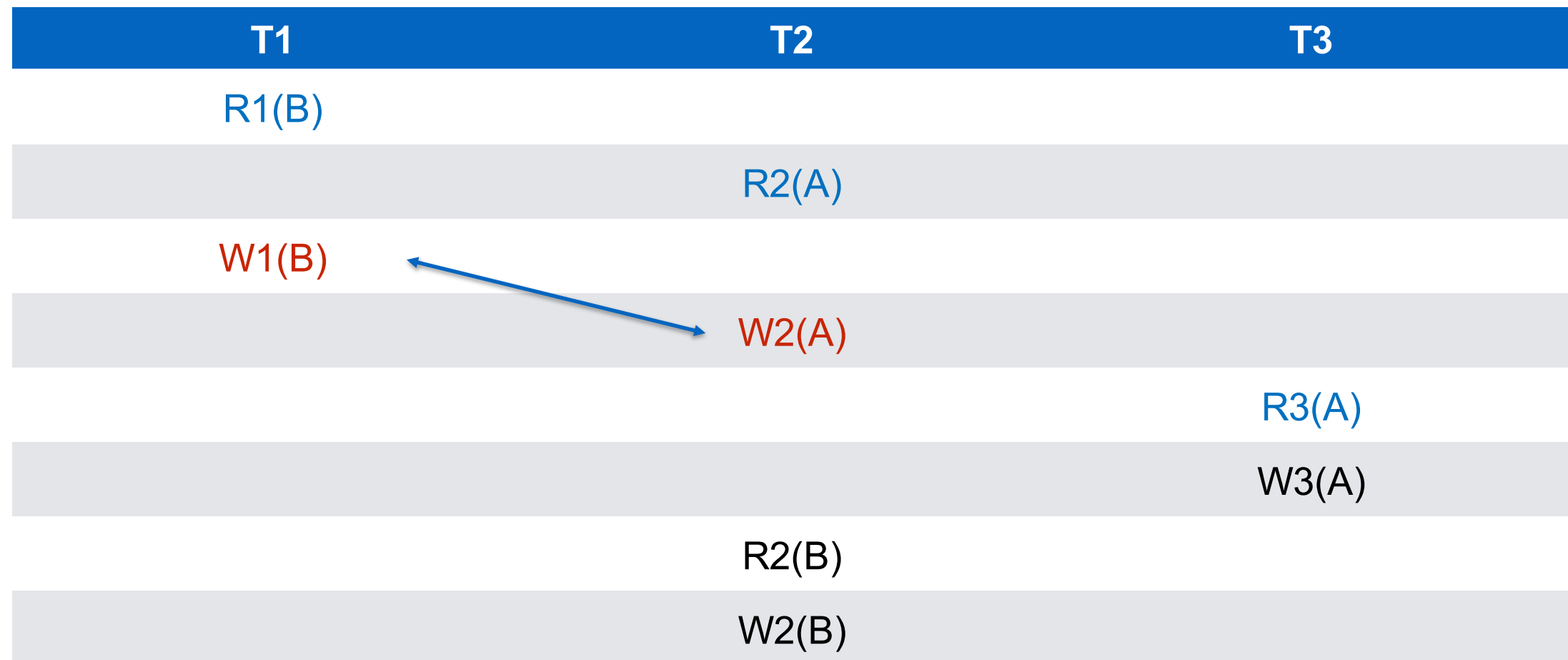
Permutemos!



# Permutando a serial

Ejemplo

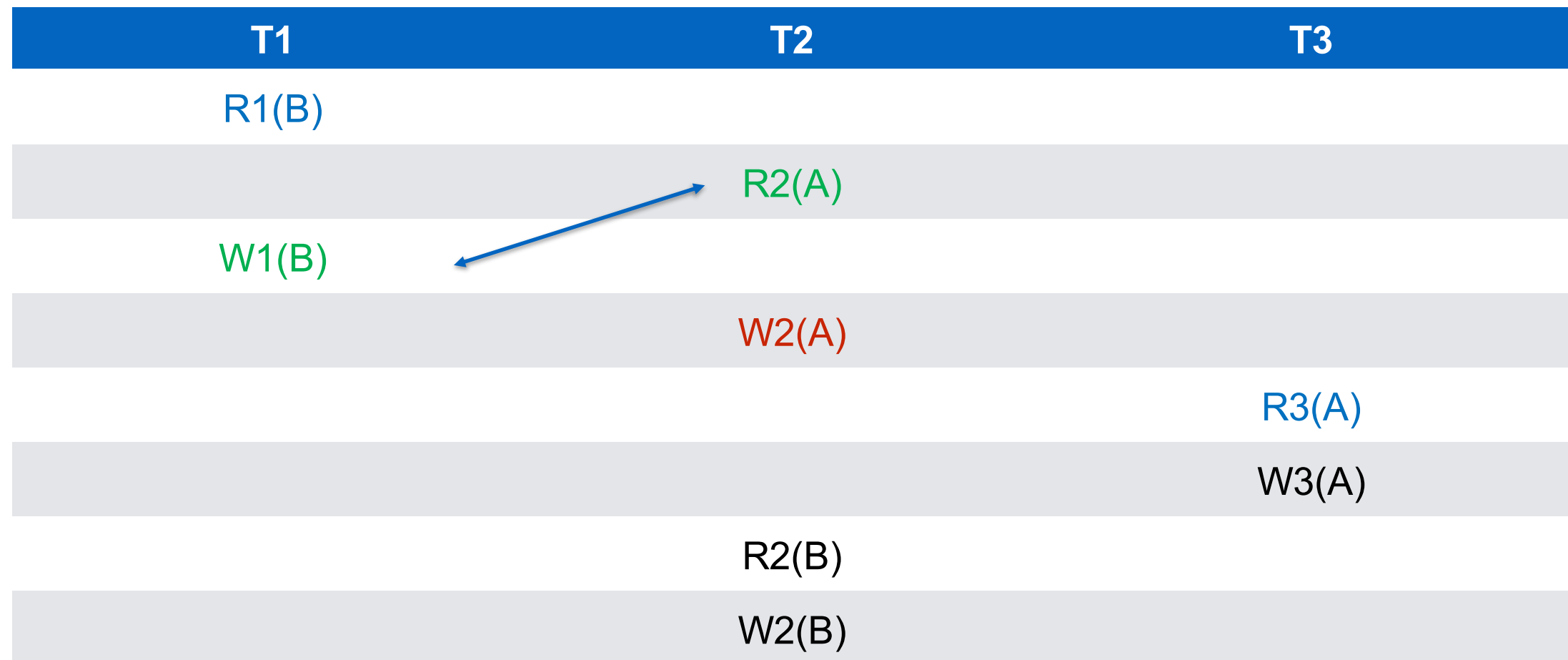
Permutemos!



# Permutando a serial

Ejemplo

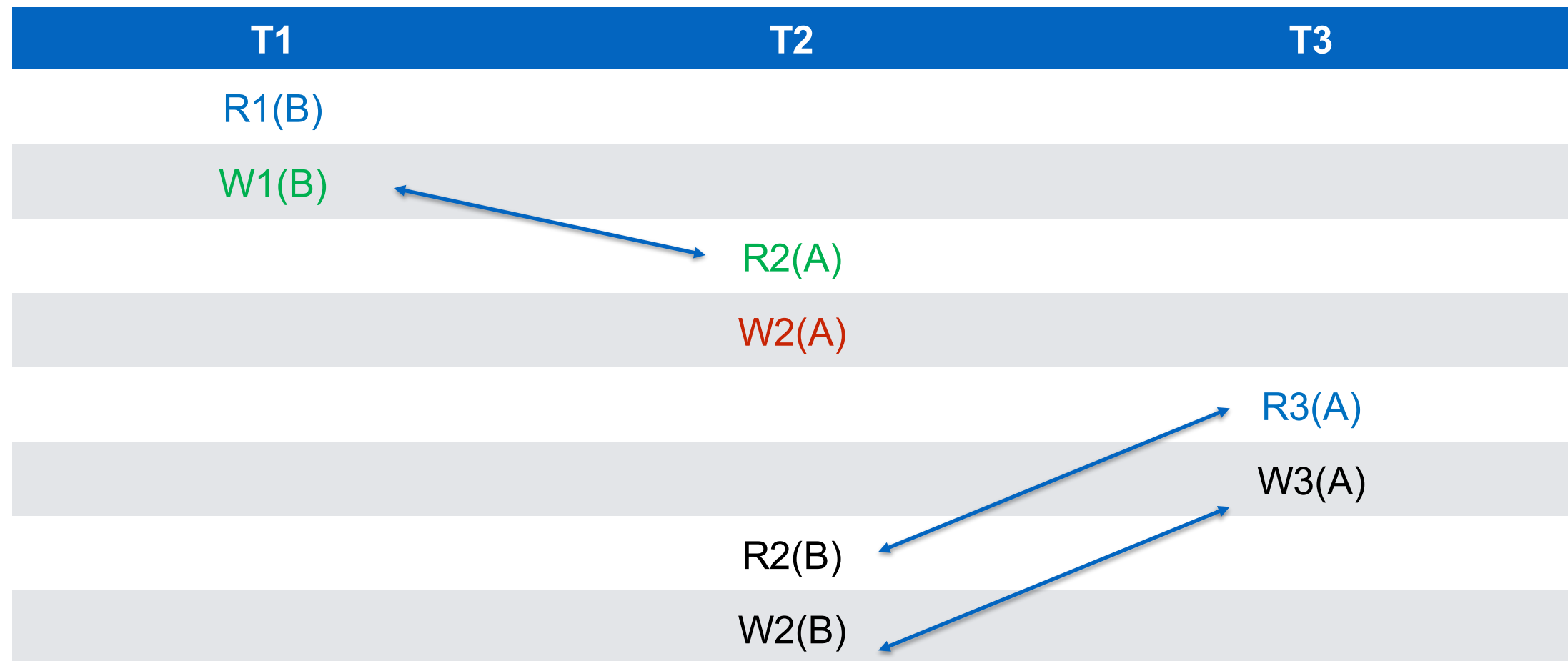
Permutemos!



# Permutando a serial

Ejemplo

Permutemos ...



# 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

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 ( $n!$ ) con  $n$  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)	

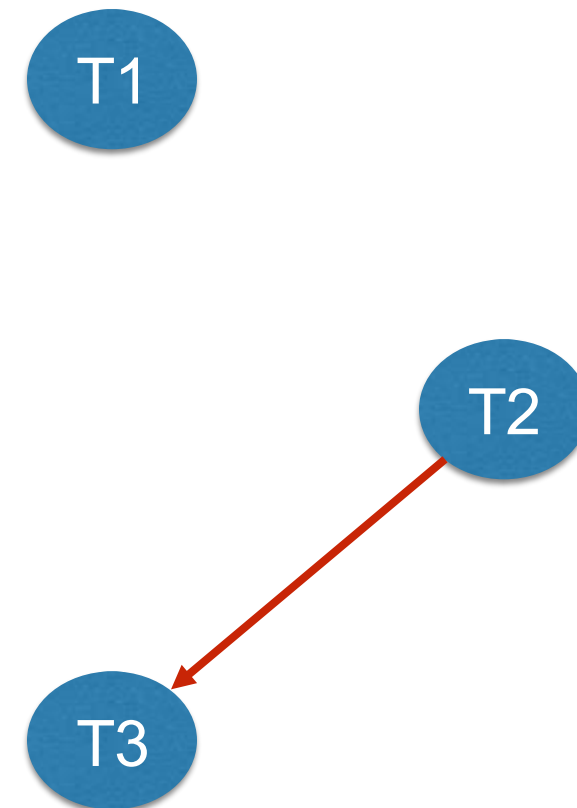
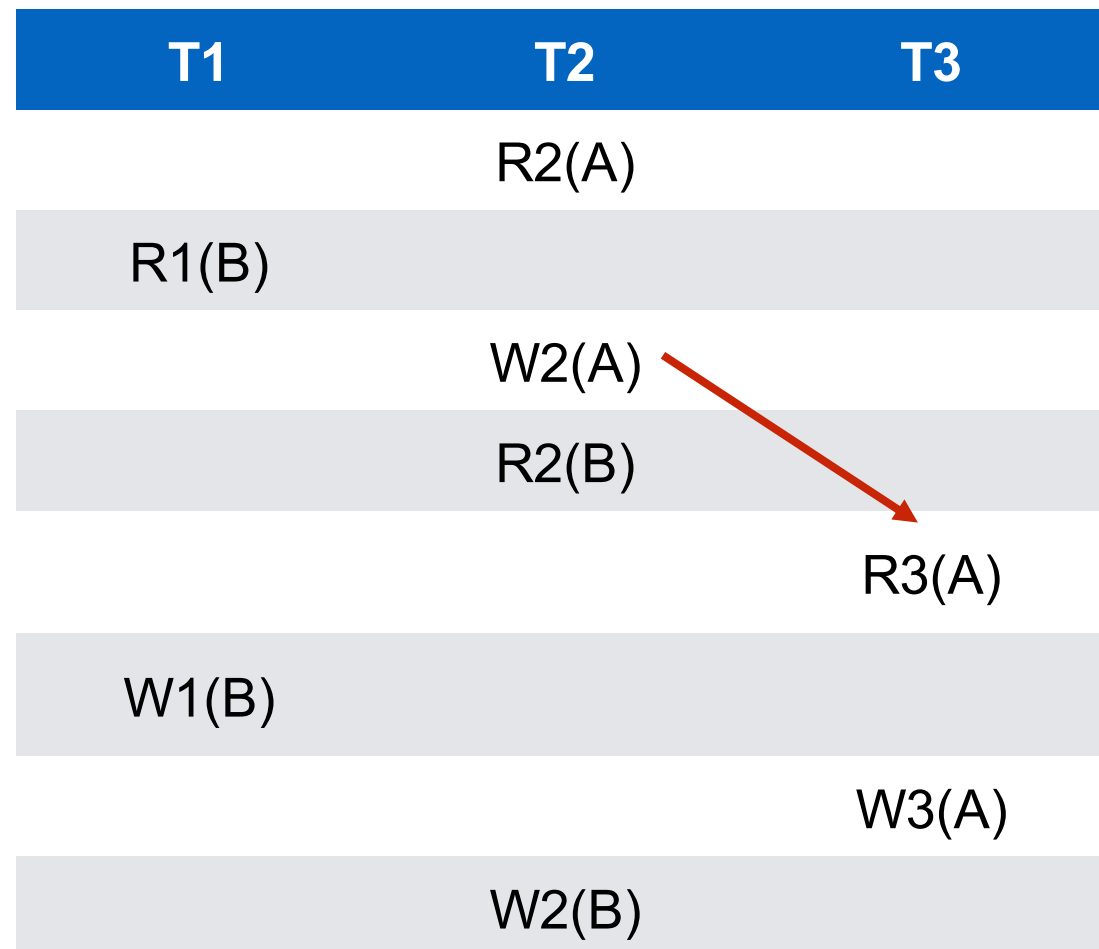
T1

T2

T3

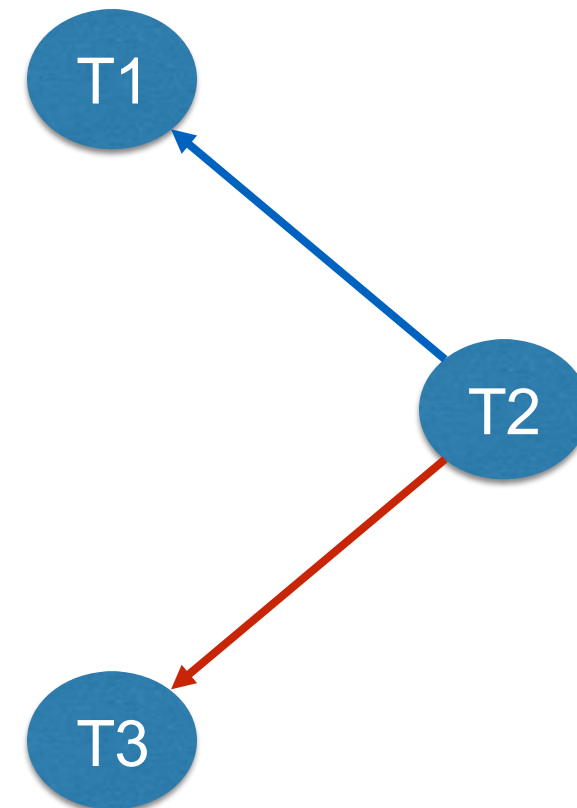
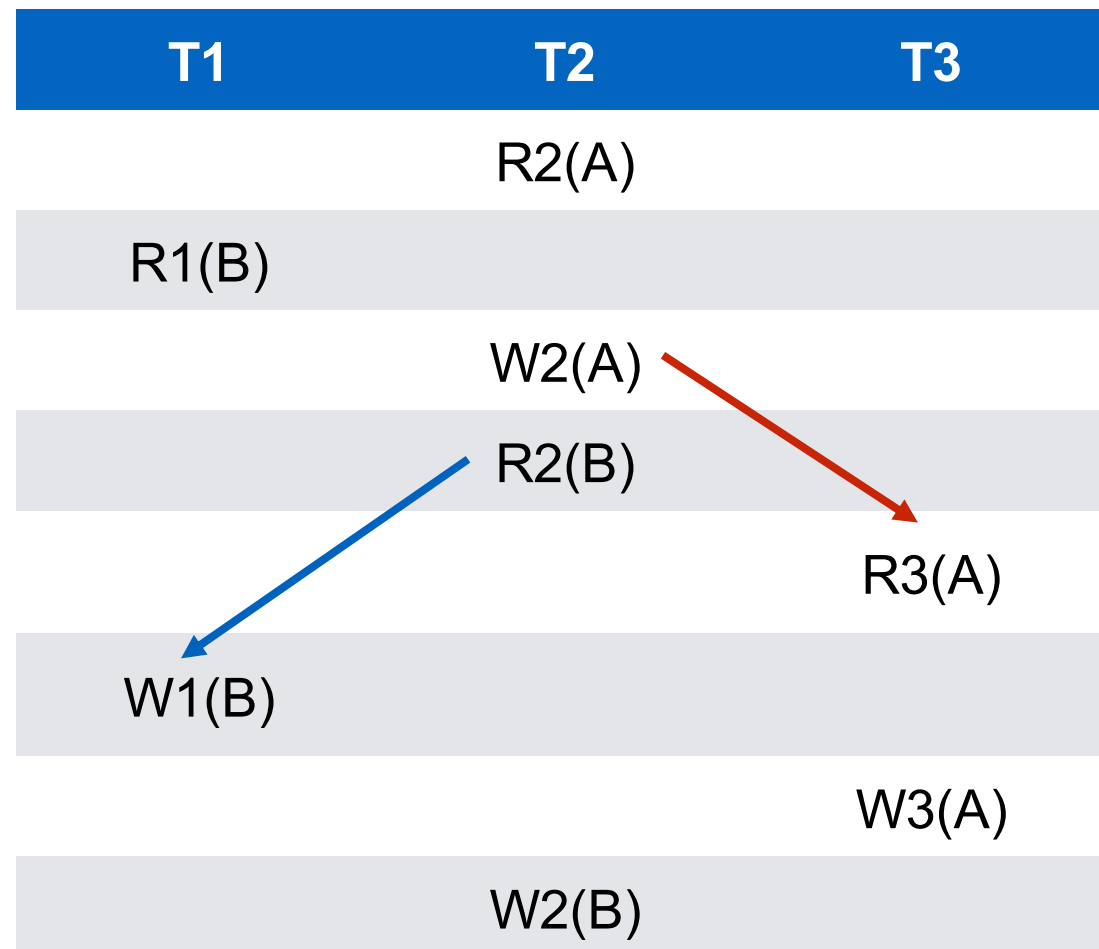
# Grafo de precedencia

¿Es *conflict serializable*?



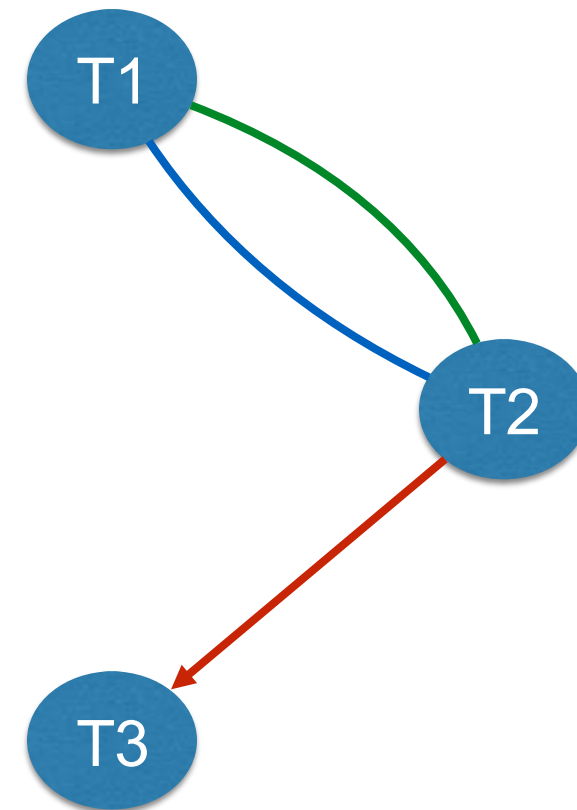
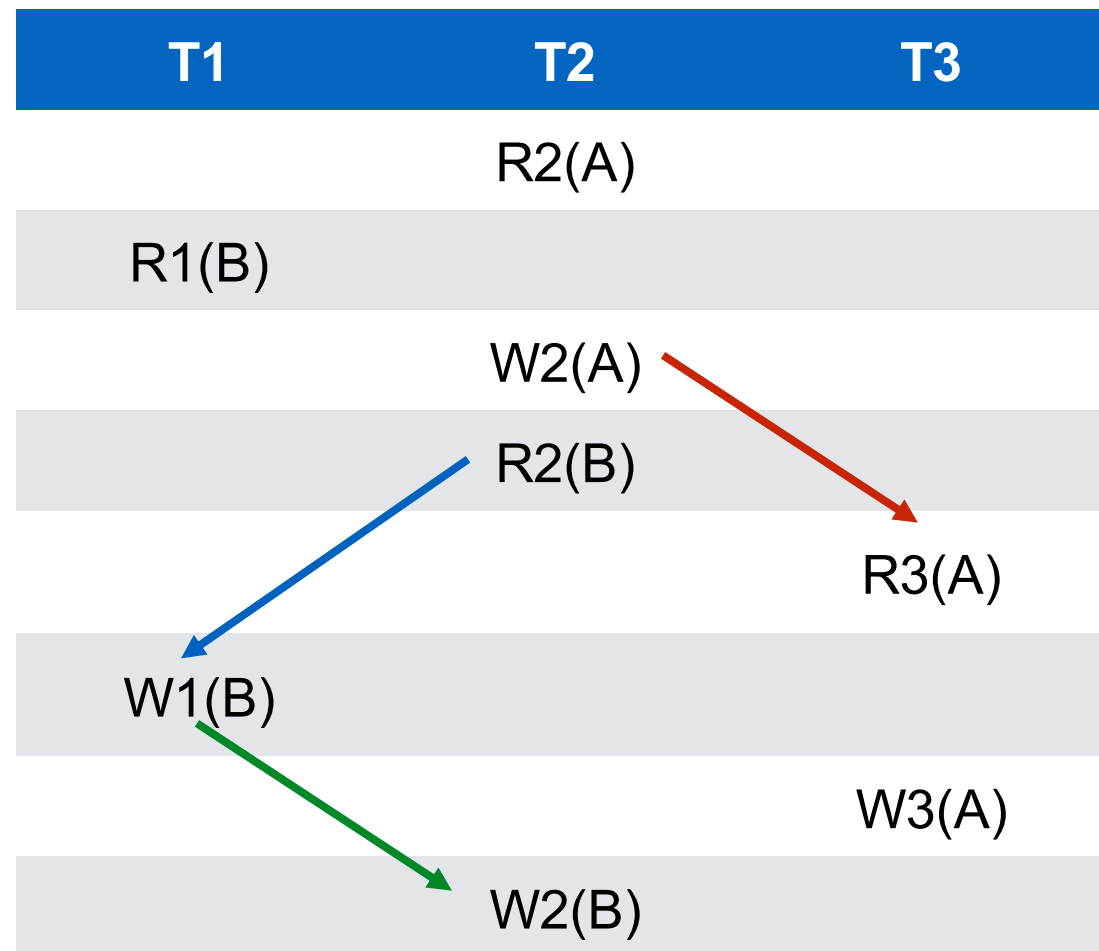
# Grafo de precedencia

¿Es *conflict serializable*?



# Grafo de precedencia

¿Es *conflict serializable*?



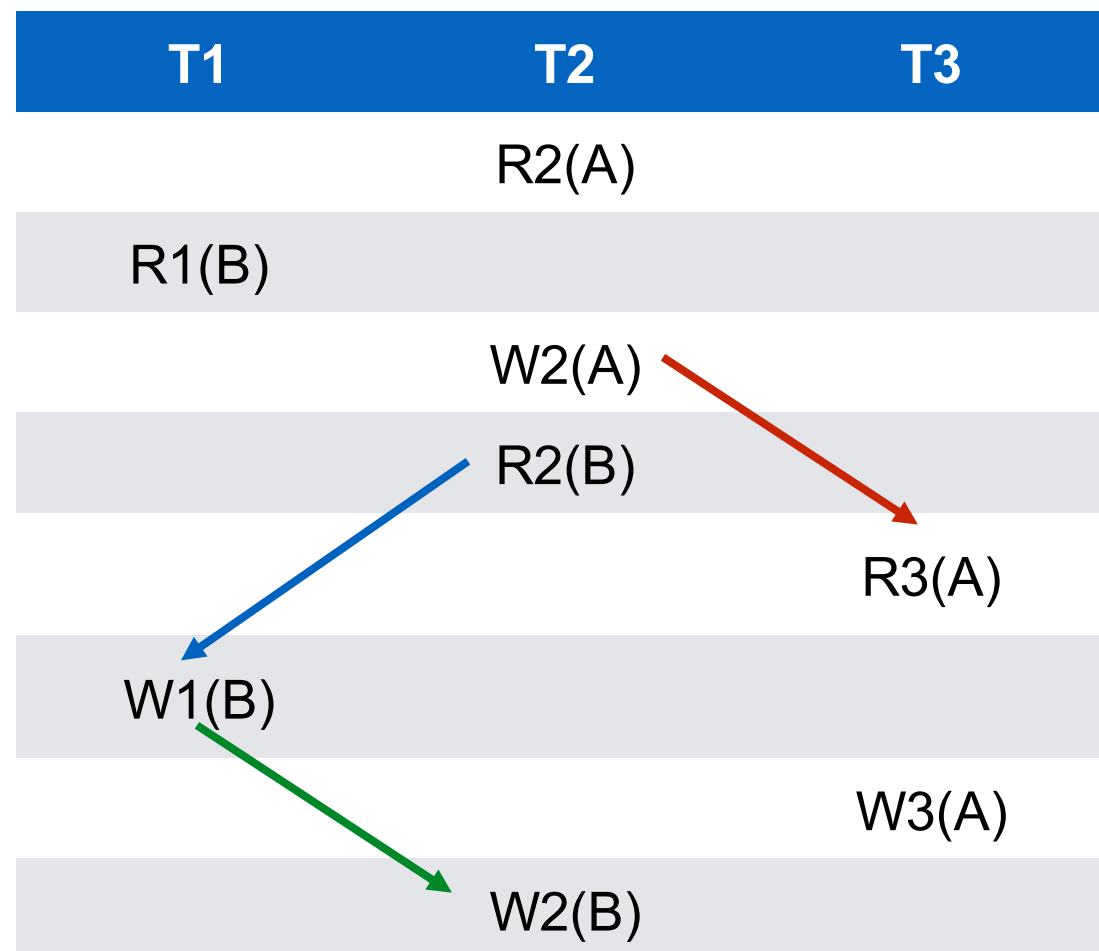
# Grafo de precedencia

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

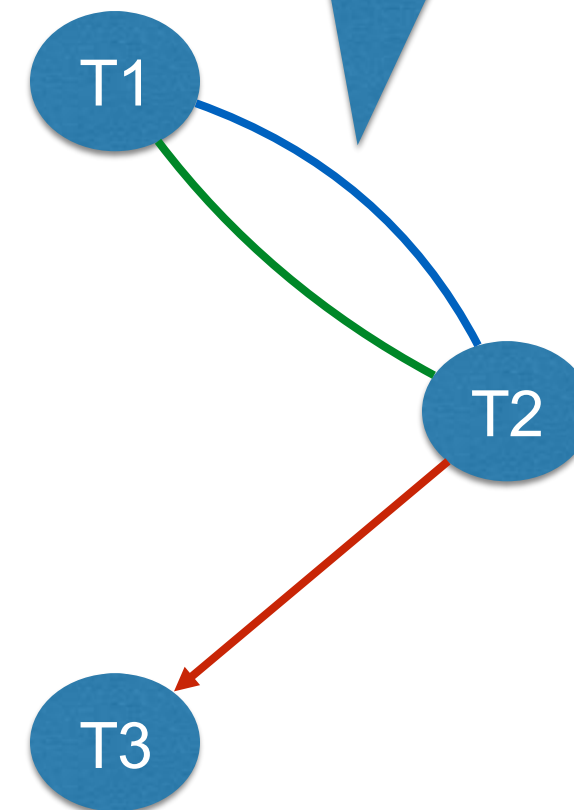


# Grafo de precedencia

¿Es *conflict serializable*?



Ciclo == no es conflict serializable



# Locks

## Strict 2PL

Two-Phase Locking

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

Está basado en la utilización de Bloqueos (locks)

Tiene dos reglas

# Reglas

## Regla 1:

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

## Regla 2:

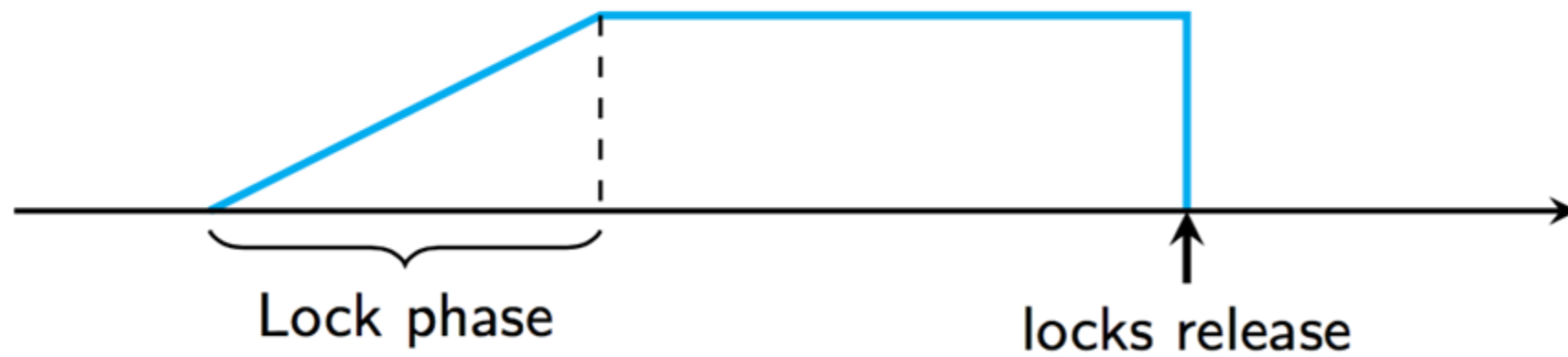
Cuando la transacción se completa, libera todos los locks que mantenía

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

Estas reglas aseguran solo **schedules**  
conflict serializables

**Strict 2PL.**

# Cómo funciona 2PL

**Locks de T1:**

Shared lock (A)

**T1**

R(A)

**T2**

W(A)

W(B)

W(B)

**Locks de T2:**

← **Espera a T1**

# Cómo funciona 2PL



# Cómo funciona 2PL

**Locks de T1:**

Listo con T1

**T1**

R(A)

W(B)

**T2**

W(A)

W(B)

**Locks de T2:**

← Espera a T1



# Cómo funciona 2PL

**Locks de T1:**

Listo con T1

**T1**

R(A)

W(B)

**T2**

W(A)

W(B)

**Locks de T2:**

← Exclusive lock (A)

...

# Problemas con locks

## 1.- 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.

# 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.

## 2.- 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:**

T2 espera que termine T1  
T1 espera que termine T2

# Transacciones en SQL

# SQL y transacciones

Se hace automáticamente cuando se ejecuta una consulta.  
¡O cuando uno se conecta a la DB con un lenguaje de programación!

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 major 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 hasta el  
SAVEPOINT

Útil en un programa qué hace varias  
transacciones y verifica condiciones



## Granularidad de locks

### Lock a nivel de tabla

```
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:** bloqueo toda la tabla

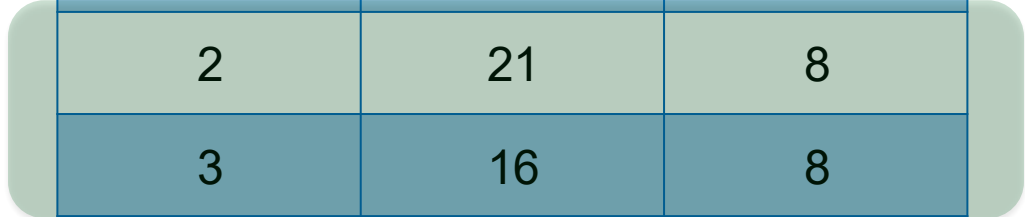
**Lock razonable:** bloqueo de tuplas con rating = 8

## Granularidad y "fantasmas"

**T1**

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

shared lock

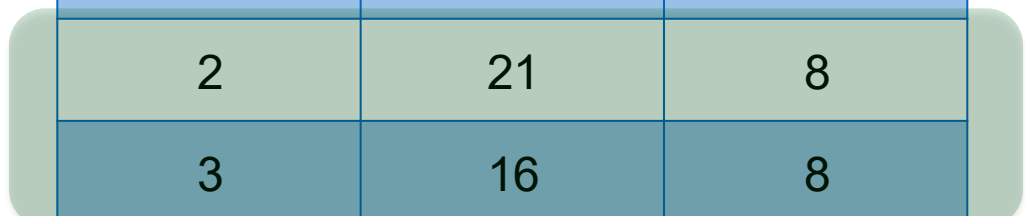


sid	age	rating
1	34	9
2	21	8
3	16	8
4	26	10

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

## 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

## Nivel de aislamiento (granularidad)

SET TRANSACTION ISOLATION LEVEL <level> READ  
ONLY

SET TRANSACTION ISOLATION LEVEL <level> READ  
WRITE

## 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?

## 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

**Lock razonable:** Tuplas de S con rating = 8