

Concurrencia



Transacciones

Una transacción es una unidad lógica de trabajo de la base de datos que incluye una o más operaciones de acceso a la base de datos, que pueden ser de inserción, modificación o recuperación.

Las transacciones pueden delimitarse de forma explícita con sentencias de tipo “iniciar transacción” y “terminar transacción”

Como manejamos en SQL las transacciones:

Transacciones en SQL - COMMIT

```
BEGIN TRANSACTION;
```

```
SELECT nombre, saldo FROM cuentas WHERE cod_cli = 2564;
```

Nombre		saldo
--------	--	-------

Alberto		2.200
---------	--	-------

```
UPDATE cuentas SET saldo = 8.000 WHERE cod_cli = 2564 ;
```

```
COMMIT;
```

AQUÍ TERMINA LA TRANSACCIÓN

```
SELECT nombre, saldo FROM cuentas WHERE cod_cli = 2564;
```

Nombre		saldo
--------	--	-------

Alberto		8.000
---------	--	-------

Transacciones en SQL - ROLLBACK

```
BEGIN TRANSACTION;
```

```
SELECT nombre, saldo FROM cuentas WHERE cod_cli = 2564;
```

Nombre		saldo
Alberto		2.200

```
UPDATE cuentas SET saldo = 8.000 WHERE cod_cli = 2564 ;
```

```
ROLLBACK;
```

SE ANULA LA TRANSACCIÓN

```
SELECT nombre, saldo FROM cuentas WHERE cod_cli = 2564;
```

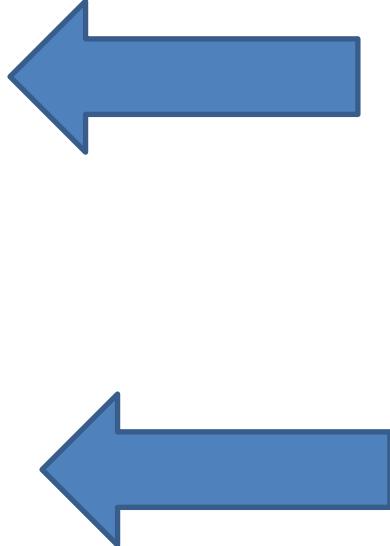
Nombre		saldo
Alberto		2.200

EL SALDO NO SE ACTUALIZÓ PORQUE
SE ABORTÓ LA TRANSACCION

**Característica básica de
las transacciones:**

En las transacciones tenemos operaciones básicas como leer o escribir elemento y cálculos sobre los datos leídos.

```
begin  
leer(A)  
leer (B)  
A = A + B  
B = B * 1.1  
escribir(A)  
escribir(B)  
end
```



Para nuestro análisis lo que nos va a interesar son las operaciones de lectura y de escritura en la base de datos.

Concurrencia de transacciones

Concurrencia

*Llamamos **concurrencia** a la situación donde muchas transacciones acceden a la misma Base de Datos al mismo tiempo, y comparten los mismos datos de la misma.*

Esto ocurre en un sistema multiusuario, donde el DBSM es el encargado de administrar estos múltiples accesos.

TRANSACTIONS EN CONFLICTO



Conflictos

La concurrencia de transacciones puede provocar situaciones inesperadas, por conflictos en el procesamiento de varias transacciones, que utilizan los mismos recursos.

Ejemplo de conflicto

Estoy haciendo una compra en el supermercado:
Pago con tarjeta de débito y concurrentemente
se produce un débito automático a mi cuenta
para pagar el servicio eléctrico de mi casa.

Saldo cuenta= \$8.000

A pagar supermercado= \$4.300

Factura eléctrica=\$ 1.500

Transacciones en conflicto

T0 (SUPERMERCADO)	T1 (EDESUR)
leer(A) $A = A - 4.300$	$A = 8.000$
escribir(A)	$A = 3.700$
leer(B) $B = B + 4.300$	$B = 850.000$
escribir(B)	$B = 854.300$
	$A = 8.000$
	$A = 6.500$
	$C = 59.000$
	$C = 60.500$

Como terminó el procesamiento?

VALORES INICIALES

A=8.000 B = 850.000 C=59.000

TOTAL=917.000

VALORES FINALES

A=6.500 B = 854.300 C=60.500

TOTAL=921.300

Se “crearon” 4.300!

Que es lo que pasó?

T0 (SUPERMERCADO)	T1 (EDESUR)
<p>leer(A) $A = A - 4.300$</p> <p>escribir(A) leer(B) $B = B + 4.300$ escribir(B)</p>	<p>Se perdió esta actualización</p> <p>leer(A) $A = A - 1.500$</p> <p>escribir(A) leer(C) $C = C + 1.500$ escribir(C)</p> <p>“pisada” por esta otra</p>

Se llama anomalía de actualización perdida.

Procesando en “serie”

T0 (SUPERMERCADO)	T1 (EDESUR)
<pre>leer(A) A = A - 4.300 escribir(A) leer(B) B = B + 4.300 escribir(B)</pre>	<pre>A= 8.000 A= 3.700 B=850.000 B=854.300</pre> <pre>leer(A) A = A - 1.500 escribir(A) leer(C) C = C + 1.500 escribir(C)</pre> <pre>A= 3.700 A= 2.200 C= 59.000 C= 60.500</pre>

Se ejecuta T0 y luego T1 (en serie)

Que ocurrió en el procesamiento en serie?

VALORES INICIALES

A=8.000 B = 850.000 C=59.000

TOTAL=917.000

VALORES FINALES

A=2.200 B = 854.300 C=60.500

TOTAL=917.000

Se mantiene la consistencia

Porque no procesamos siempre en serie?



Tendríamos que bloquear todas las demás transacciones hasta que se termine la anterior.



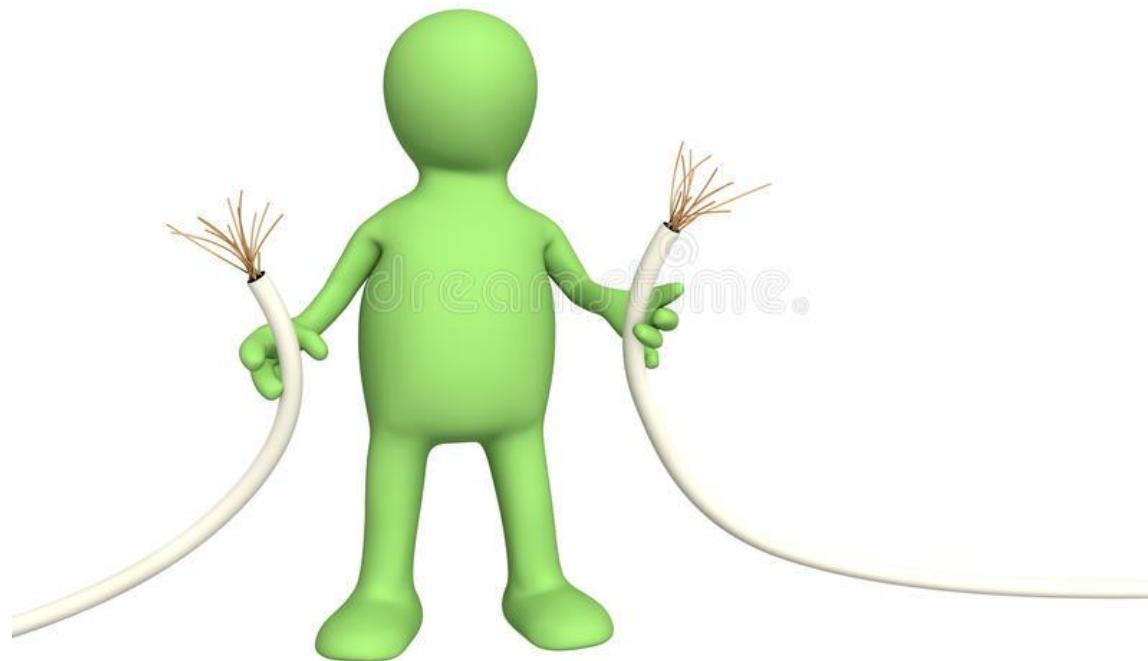
Estaríamos desaprovechando recursos, cada lectura demora y el procesador queda inactivo



Podría darse el caso de que un proceso tarde mucho y bloquee todos los accesos a la base.

Más anomalías

Anomalías por transacción abortada



Anomalías por transacción abortada

T0 (SUPERMERCADO)	T1 (EDESUR)
leer(A) A = A - 4.300 escribir(A)	A = 8.000 A = 3.700
leer(B) ... abort	A = 3.700 A = 2.200
Cuando T0 aborta, se anulan todas las operaciones de T0	

Actualización Temporal (lectura sucia)

Anomalías por transacción abortada

T0 se puede haber abortado por problemas en el sistema del supermercado, T1 lee el valor de A que dejó T0.

El valor de A quedó en 2.200, como si se hubiera pagado la compra de supermercado, a pesar de que se abortó la misma.

Y lo peor es que no voy a poder llevarme la compra del super!

Pero todavía hay más anomalías...

Vimos:

- ✓ *Actualización perdida*
- ✓ *Lectura sucia*

... pueden ocurrir resúmenes incorrectos



Resumen incorrecto

T0 (SUPERMERCADO)	T1 (BANCO HACIENDO SUMA)
leer(A) $A = A - 4.300$ escribir(A)	leer(A) $suma = suma + A$ leer(B) $suma = suma + B$
leer(B) $B = B + 4.300$ escribir(B)	 <div data-bbox="1152 835 1766 1108"><p>En esta suma van a faltar los 4.300 que no se sumaron a B</p></div>

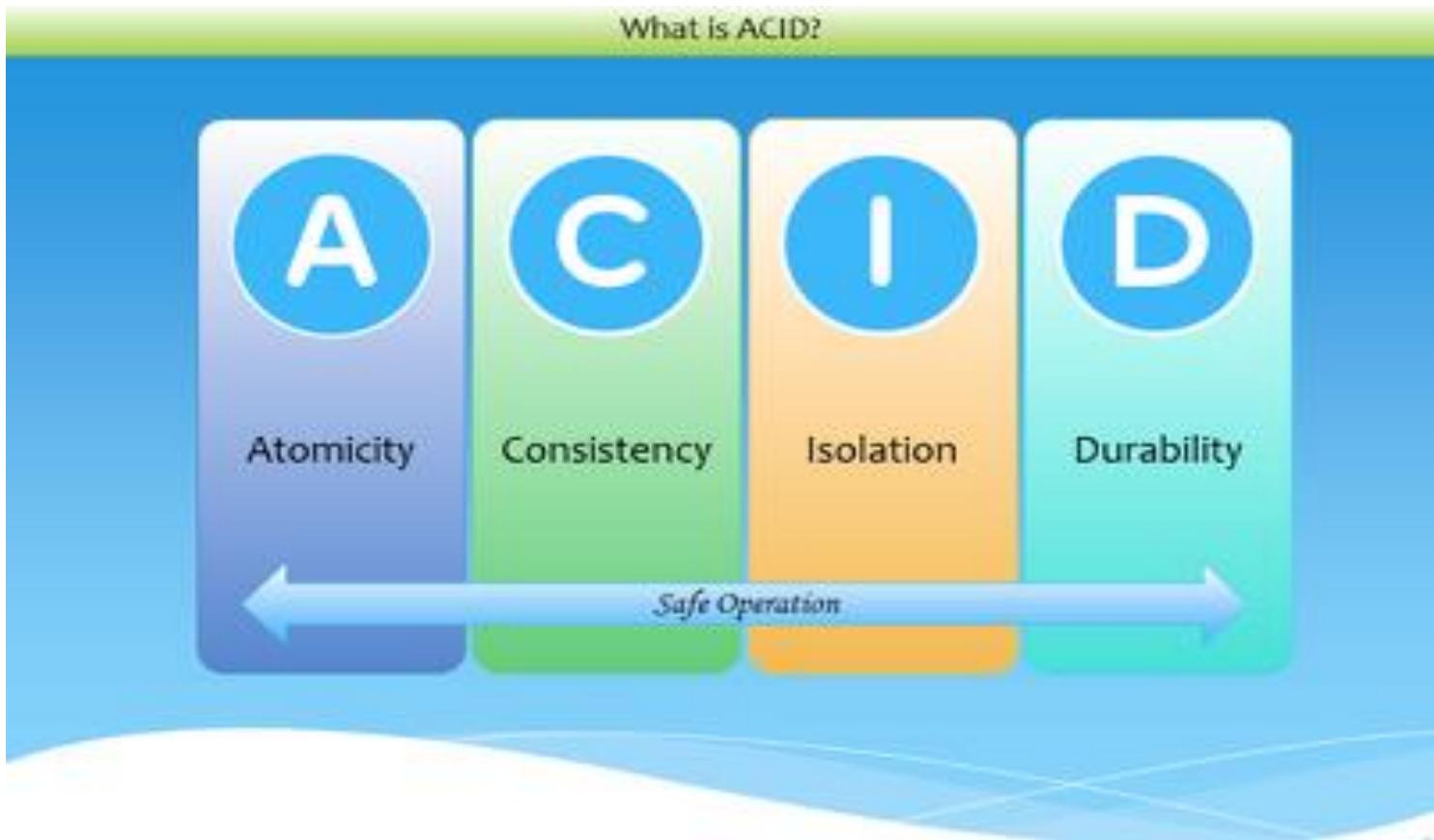
Es parecido a una actualización perdida, pero la base queda consistente.

Pero hay propiedades de las transacciones, que nos ayudan a evitar las anomalías:

Que propiedades tendrían que tener idealmente las transacciones?

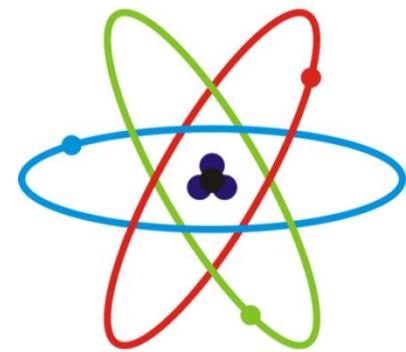


Propiedades ACID



Estas son las propiedades que las transacciones deberían cumplir al ser ejecutadas por el SGBD.

Atomicity (atomicidad)



La transacción se ejecuta por completo o no se ejecuta ninguna instrucción de la misma.

Las operaciones asociadas a una transacción comparten normalmente un objetivo y son interdependientes.

Si el sistema ejecutase únicamente una parte de las operaciones, podría poner en peligro el objetivo final de la transacción.

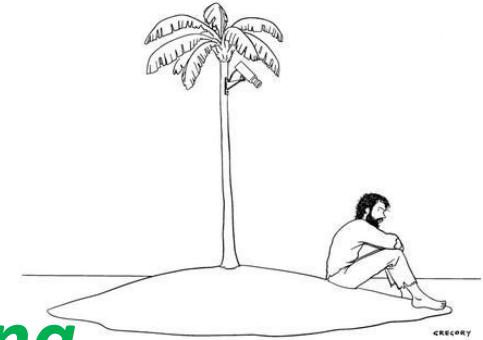
Consistency (consistencia)



Podría definirse como la coherencia entre todos los datos de la base de datos.

Consistencia es un término más amplio que el de integridad. Cuando se pierde la integridad también se pierde la consistencia. Pero la consistencia también puede perderse por razones de funcionamiento. Por ejemplo, como vimos antes, el saldo defectuoso del banco al haberse abortado una transacción.

Isolation (aislamiento)

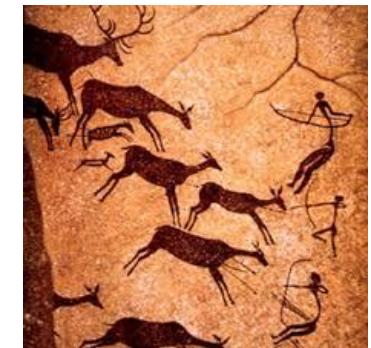


Es la propiedad que asegura que una transacción no puede afectar a otras.

Esto asegura que la realización de dos transacciones sobre la misma información nunca generará ningún tipo de error.

Esta propiedad se cumple cuando las transacciones son equivalentes a alguna ejecución serial.

Durability (permanencia)

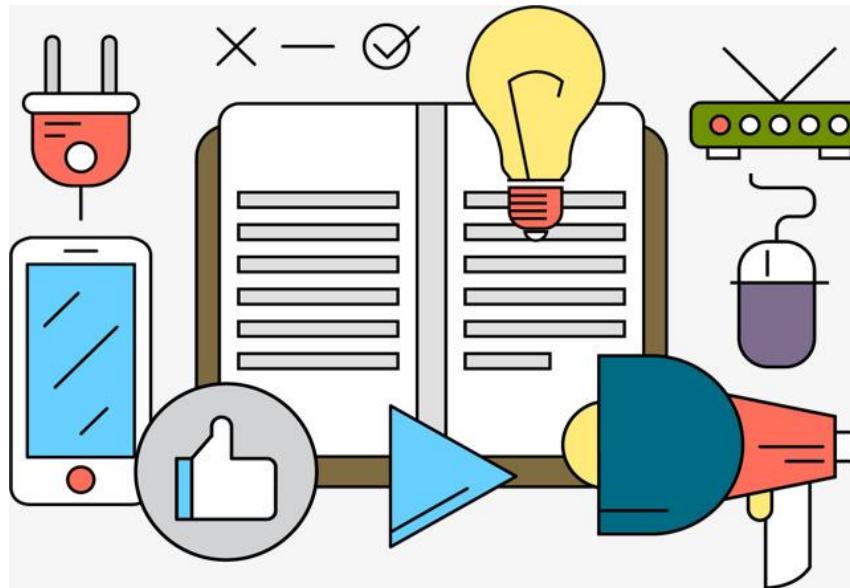


Una vez que el SGBD informa que la transacción se ha completado, después de un COMMIT, se garantiza la persistencia de la misma.

Esta propiedad es independiente de toda falla que pudiese ocurrir.

Como podemos hacer para tener procesos con las virtudes de los seriales?

Hay técnicas para crear un plan de procesos equivalente a uno serial, con sus virtudes.



Planificar ...



Planificaciones

Una planificación S de n transacciones $T_1 \dots T_n$ es una ordenación de las operaciones de esas transacciones.

S debe conservar todas las instrucciones de la transacción y además debe conservar el orden de las instrucciones dentro de la transacción.

En las planificaciones sólo tomaremos en cuenta las lecturas (r) y escrituras (w). También tendremos inicio (b), commit (c) y abort (a).

Ejemplo de planificación

T0 (SUPERMERCADO)	T1 (EDESUR)
leer(A) $A = A - 4.300$ escribir(A) leer(B) $B = B + 4.300$ escribir(B)	leer(A) $A = A - 1.500$ escribir(A) leer(C) $C = C + 1.500$ escribir(C)

Ejemplo de planificación

T0 (SUPERMERCADO)	T1 (EDESUR)
<p>leer(A) $A = A - 4.300$ escribir(A) leer(B) $B = B + 4.300$ escribir(B)</p>	<p>leer(A) $A = A - 1.500$ escribir(A) leer(C) $C = C + 1.500$ escribir(C)</p>

S: $r_0(A)$

Ejemplo de planificación

T0 (SUPERMERCADO)	T1 (EDESUR)
<p>leer(A) $A = A - 4.300$ escribir(A) leer(B) $B = B + 4.300$ escribir(B)</p>	<p>leer(A) $A = A - 1.500$ escribir(A) leer(C) $C = C + 1.500$ escribir(C)</p>

S: $r_0(A)$; **w₀(A)**

Ejemplo de planificación

T0 (SUPERMERCADO)	T1 (EDESUR)
leer(A) A = A – 4.300 escribir(A) leer(B) B = B + 4.300 escribir(B)	leer(A) A = A – 1.500 escribir(A) leer(C) C = C + 1.500 escribir(C)

S: $r_0(A);w_0(A) ;\textcolor{red}{r_0(B)}$

Ejemplo de planificación

T0 (SUPERMERCADO)	T1 (EDESUR)
leer(A) A = A – 4.300 escribir(A) leer(B) B = B + 4.300 escribir(B)	leer(A) A = A – 1.500 escribir(A) leer(C) C = C + 1.500 escribir(C)

S: $r_0(A);w_0(A);r_0(B) ;\textcolor{red}{w_0(B)}$

Ejemplo de planificación

T0 (SUPERMERCADO)	T1 (EDESUR)
leer(A) A = A – 4.300 escribir(A) leer(B) B = B + 4.300 escribir(B)	leer(A) A = A – 1.500 escribir(A) leer(C) C = C + 1.500 escribir(C)

S: $r_0(A);w_0(A);r_0(B);w_0(B) ;\textcolor{red}{r_1(A)}$

Ejemplo de planificación

T0 (SUPERMERCADO)	T1 (EDESUR)
leer(A) A = A – 4.300 escribir(A) leer(B) B = B + 4.300 escribir(B)	leer(A) A = A – 1.500 escribir(A) leer(C) C = C + 1.500 escribir(C)

S: $r_0(A);w_0(A);r_0(B);w_0(B);r_1(A) ;\textcolor{red}{w_1(A)}$

Ejemplo de planificación

T0 (SUPERMERCADO)	T1 (EDESUR)
leer(A) A = A – 4.300 escribir(A) leer(B) B = B + 4.300 escribir(B)	leer(A) A = A – 1.500 escribir(A) leer(C) C = C + 1.500 escribir(C)

S: $r_0(A);w_0(A);r_0(B);w_0(B);r_1(A);w_1(A) ;\textcolor{red}{r_1(C)}$

Ejemplo de planificación

T0 (SUPERMERCADO)	T1 (EDESUR)
leer(A) A = A – 4.300 escribir(A) leer(B) B = B + 4.300 escribir(B)	leer(A) A = A – 1.500 escribir(A) leer(C) C = C + 1.500 escribir(C)

S: $r_0(A);w_0(A);r_0(B);w_0(B);r_1(A);w_1(A);r_1(C) ;\textcolor{red}{w_1(C)}$

Ejemplo de planificación

T0 (SUPERMERCADO)	T1 (EDESUR)
leer(A) A = A – 4.300 escribir(A) leer(B) B = B + 4.300 escribir(B)	leer(A) A = A – 1.500 escribir(A) leer(C) C = C + 1.500 escribir(C)

S: $r_0(A);w_0(A);r_0(B);w_0(B);r_1(A);w_1(A);r_1(C);w_1(C)$

Ejecución serial

Como ya vimos, en las ejecuciones seriales evitamos las interferencias entre las transacciones y obtenemos resultados correctos.

*Es muy costoso procesar de esta manera, por ello lo que hacemos es permitir planificaciones solapadas, pero buscamos que sean **equivalentes a las seriales**.*

Para ello definiremos como deben ser las planificaciones para ser equivalentes a una serial.

Equivalencia de solapamientos

Existen distintas nociones de equivalencias entre planificaciones:

- *Equivalencia de resultados: cuando ambas planificaciones dejan la base de datos en el mismo estado (poco confiable)*
- *Equivalencia de conflictos: ambas planificaciones poseen los mismos conflictos entre instrucciones. Es interesante porque no depende del estado inicial de la base.*

Equivalencia de solapamientos

- *Equivalencia de vistas: en cada planificación, cada lectura de Ti debe leer el valor escrito por la misma transacción Tj. Además se pide que en ambas planificaciones la última modificación de cada ítem haya sido hecha por la misma transacción.*

Equivalencia de conflictos



Operaciones Conflictivas

$\{w_0(A); w_1(A)\}$

- *lo que escribe T0 es reescrito por T1.*

$\{r_0(A); w_1(A)\}$

- *el orden importa, T0 está leyendo un valor y T1 lo modifica.*

$\{w_0(A); r_1(A)\}$

- *esta combinación no presenta anomalías, pero no son intercambiables.*

Operaciones Conflictivas

Entonces tenemos un conflicto cuando dos transacciones distintas ejecutan instrucciones sobre un mismo ítem X, y al menos una de las instrucciones es una escritura.

Todo par de instrucciones consecutivas de una planificación, que no constituye un conflicto puede invertir su orden de ejecución, obteniendo un solapamiento equivalente al inicial.

Serialización por conflicto

T0	T1
leer(A) A = A – 4.300 escribir(A)	leer(A) A = A – 1.500 escribir(A) leer(C)
leer(B) B = B + 4.300 escribir(B)	C = C + 1.500 escribir(C)

Podemos demostrar que es equivalente a una ejecución serial?

Serialización por conflicto

T0	T1
<pre>leer(A) A = A - 4.300 escribir(A)</pre> <pre>leer(B) B = B + 4.300 escribir(B)</pre>	<pre>leer(A) A = A - 1.500 escribir(A) leer(C)</pre> <pre>C = C + 1.500 escribir(C)</pre>

Esta instrucción puede “subir” porque no está en conflicto, dado que en T1 no procesan a B

Trataremos de “mover” las instrucciones que no tengan conflicto para llevarla a un proceso serial.

Serialización por conflicto

T0	T1
<p>leer(A) A = A – 4.300 escribir(A) leer(B)</p> <p>B = B + 4.300 escribir(B)</p>	<p>leer(A) A = A – 1.500 escribir(A) leer(C)</p> <p>C = C + 1.500 escribir(C)</p>

Lo mismo ocurre con
escribir(B) , no tiene
conflicto con las
instrucciones de T1

Serialización por conflicto

T0	T1
leer(A) A = A – 4.300 escribir(A) leer(B) B = B + 4.300 escribir(B)	leer(A) A = A – 1.500 escribir(A) leer(C) C = C + 1.500 escribir(C)

Resulta por lo tanto una planificación serializada.

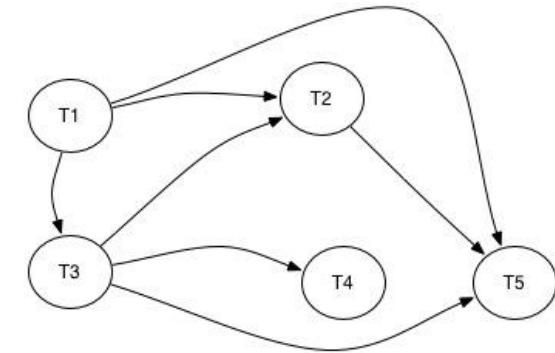
Esta planificación es serial?

T0	T1
<p>leer(A) $A = A - 4.300$</p> <p>escribir(A) leer(B) $B = B + 4.300$ escribir(B)</p>	<p>leer(A) $A = A - 1.500$</p> <p>escribir(A) leer(C) $C = C + 1.500$ escribir(C)</p> <p>No puedo mover escribir(A) dado que está en conflicto con leer(A) de T1</p>

Por lo que esta planificación no es serial.

Grafo de precedencias

La serialización por conflictos de una planificación puede ser evaluado con la construcción de un grafo de precedencias.



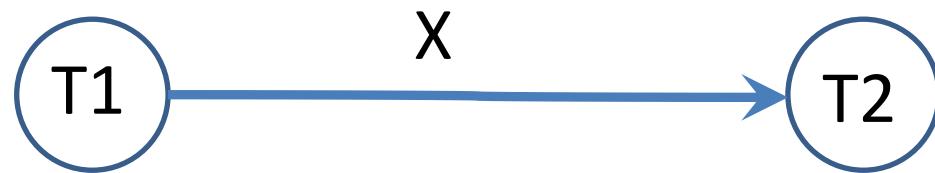
Se construye de la siguiente forma:

- *Se crea un nodo por cada transacción T_i .*
- *Se agrega un arco dirigido entre los nodos T_i y T_j si existe algún conflicto de la forma vista antes.*

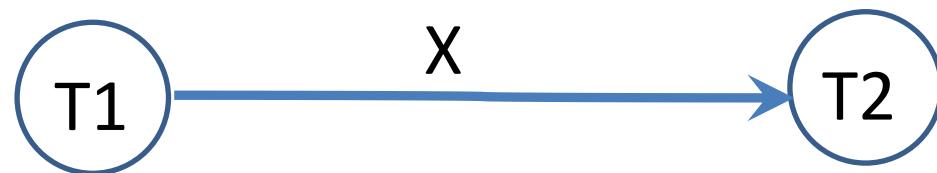
Grafo de precedencias

Podemos etiquetar el arco con el nombre del recurso en conflicto.

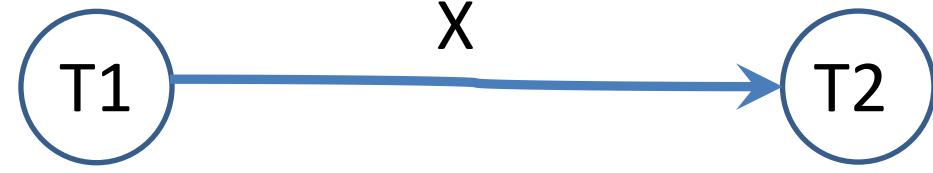
$r1(X);w2(X)$



$w1(X);r2(X)$



$w1(X);w2(X)$



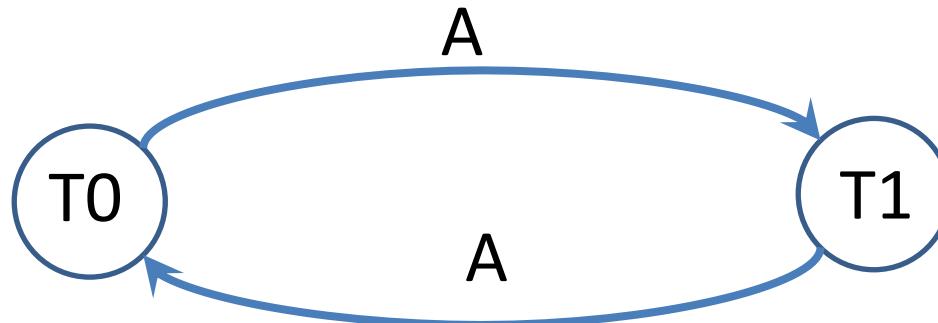
Tomamos este caso

T0	T1
leer(A) $A = A - 4.300$	leer(A) $A = A - 1.500$
escribir(A) leer(B) $B = B + 4.300$ escribir(B)	escribir(A) leer(C) $C = C + 1.500$ escribir(C)

S:r0(A);r1(A);w0(A);r0(B);w0(B);w1(A);r1(C);w1(C)

Resulta:

S:r0(A);r1(A);w0(A);r0(B);w0(B);w1(A);r1(C);w1(C)



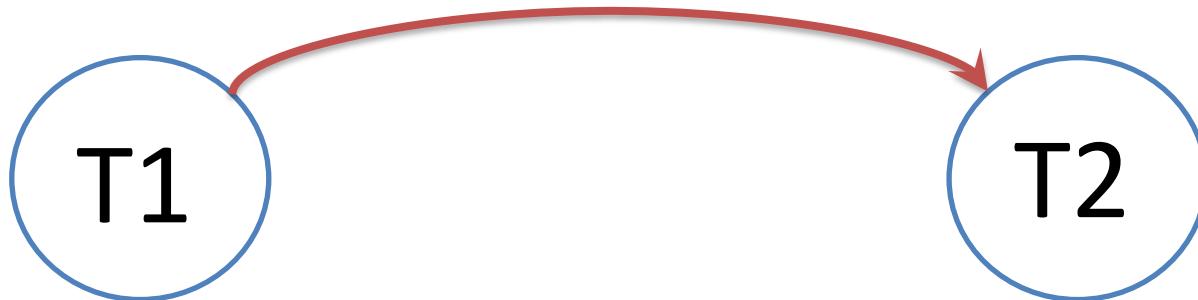
Resulta un ciclo, entonces no es serial

Ejercicio: comprobar si es serializable:

$S_1: r_1(X); r_2(Z); r_1(Z); r_2(X); w_2(X); w_1(Z); r_3(Y); w_2(Y);$
 $w_3(Y); r_1(Y);$

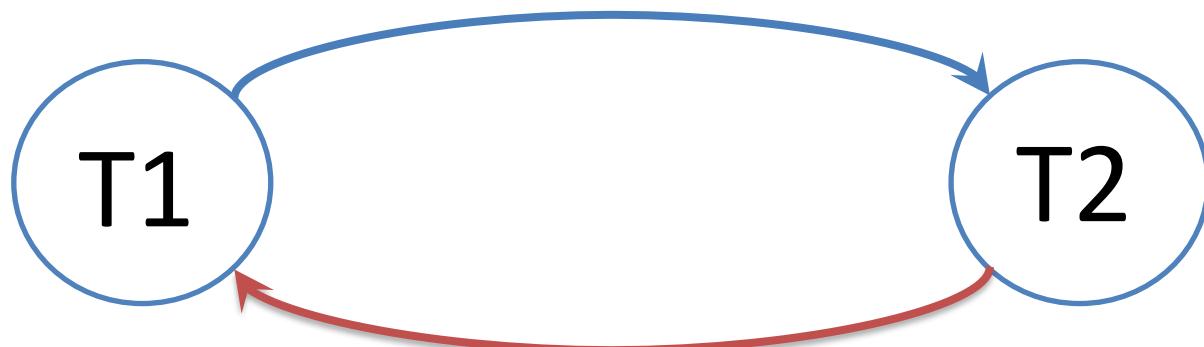
Ejercicio: comprobar si es serializable:

S_1 : $r_1(X)$; $r_2(Z)$; $r_1(Z)$; $r_2(X)$; $w_2(X)$; $w_1(Z)$; $r_3(Y)$; $w_2(Y)$; $w_3(Y)$; $r_1(Y)$;



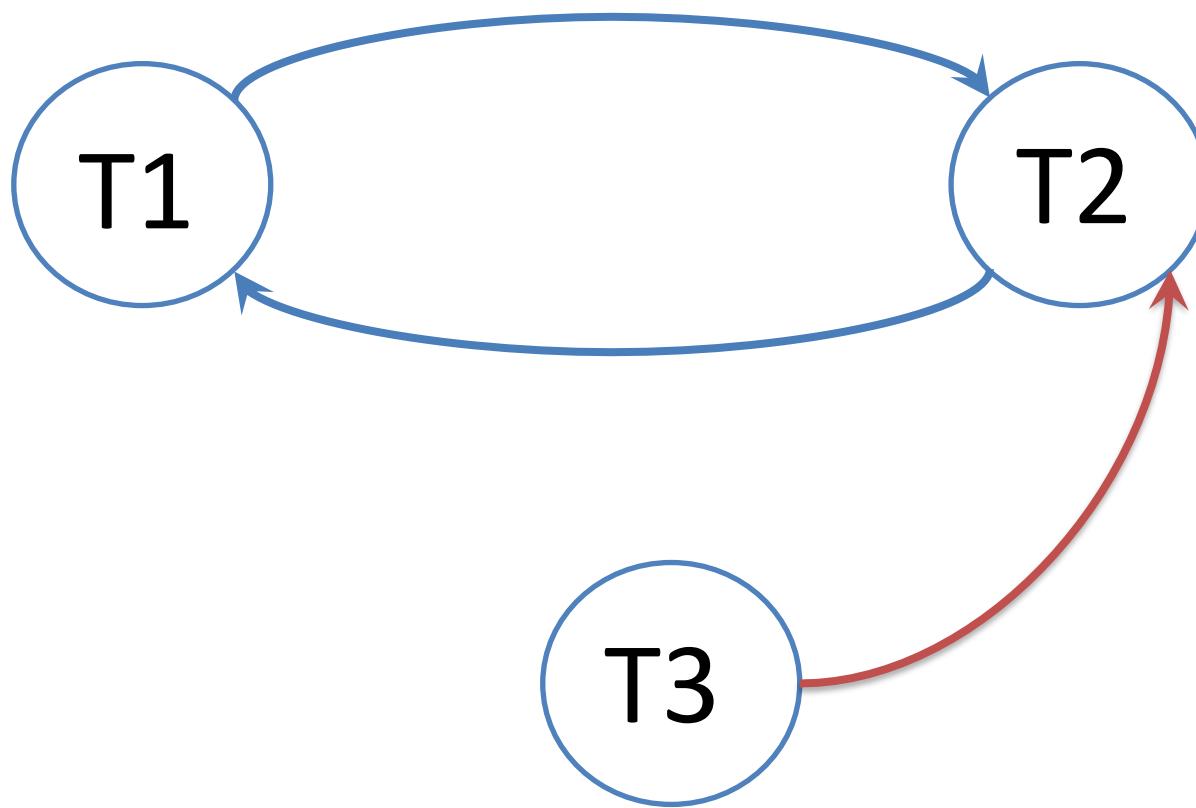
Ejercicio: comprobar si es serializable:

$S_1: r_1(X); \textcolor{red}{r}_2(Z); r_1(Z); r_2(X); w_2(X); \textcolor{red}{w}_1(Z); r_3(Y); w_2(Y); w_3(Y); r_1(Y);$



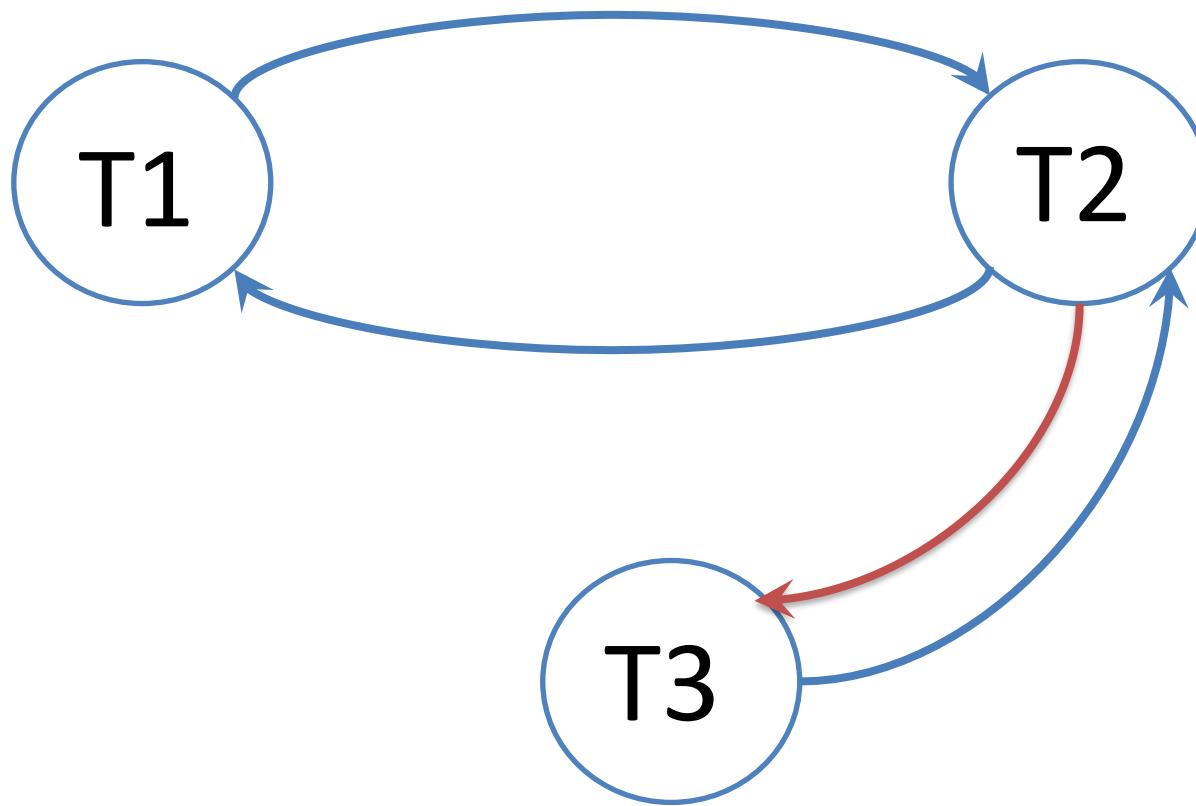
Ejercicio: comprobar si es serializable:

$S_1: r_1(X); r_2(Z); r_1(Z); r_2(X); w_2(X); w_1(Z); \color{red}{r_3(Y)}; w_2(Y); w_3(Y); r_1(Y);$



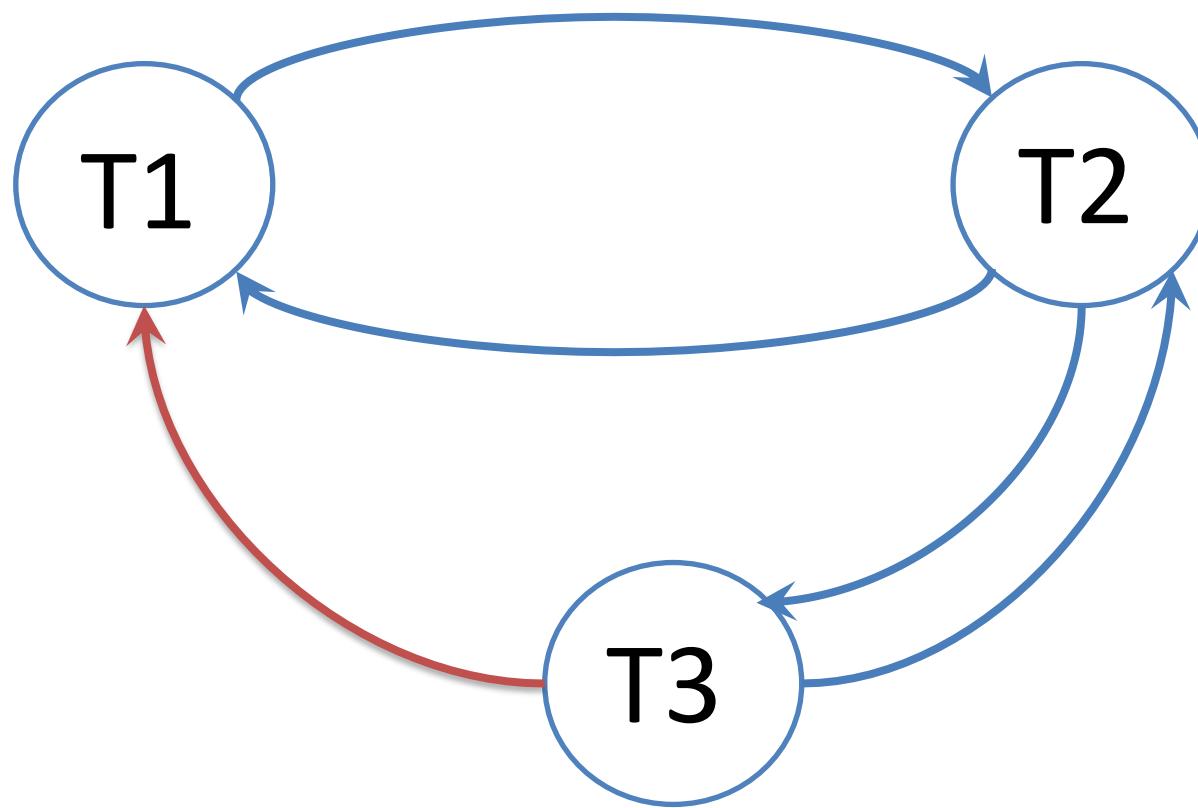
Ejercicio: comprobar si es serializable:

$S_1: r_1(X); r_2(Z); r_1(Z); r_2(X); w_2(X); w_1(Z); r_3(Y); \textcolor{red}{w_2(Y)}; \textcolor{red}{w_3(Y)}; r_1(Y);$



Ejercicio: comprobar si es serializable:

$S_1: r_1(X); r_2(Z); r_1(Z); r_2(X); w_2(X); w_1(Z); r_3(Y); w_2(Y);$
 $w_3(Y); r_1(Y);$



Dibuje los grafos de precedencia y determine cuales de las siguientes planificaciones son serializables por conflicto.

1. $S_1: r_1(X); r_2(Z); r_1(Z); r_3(X); r_3(Y); w_1(X); w_3(Y);$
 $r_2(Y); w_2(Z); w_2(Y);$
2. $S_2: r_1(X); r_2(Z); r_3(X); r_1(Z); r_2(Y); r_3(Y); w_1(X);$
 $w_2(Z); w_3(Y); w_2(Y);$

Resumiendo

Si se verifica la seriabilidad de las planificaciones, entonces no se producirá ninguna de las anomalías mostradas anteriormente.

Pero en la práctica, es imposible comprobar la seriabilidad, dado que no se conocen de antemano todas las operaciones y las secuencias que se realizarán en una transacción, ni tampoco si abortará o terminará exitosamente.

Entonces...

Los SGDB en lugar de verificar la seriabilidad, implementan protocolos que garantizan que los planes resultantes son equivalentes a planes seriales.

Control de Concurrencia

Control de concurrencia

Para lograr el objetivo de que se cumpla el aislamiento hay dos enfoques:

- 1. Optimista, deja hacer y luego validar, si falló hace rollback.*
- 2. Pesimista, busca garantizar que no se produzcan conflictos. Hay dos técnicas:*

- *Control de concurrencia basada en locks.*
- *Control de concurrencia basado en timestamps.*

Elegiremos esta técnica, dado que es la más utilizada en los SGBD

Control de concurrencia

Los locks o candados son variables asociadas a determinados recursos, que permiten regular el acceso a los mismos en los sistemas concurrentes.



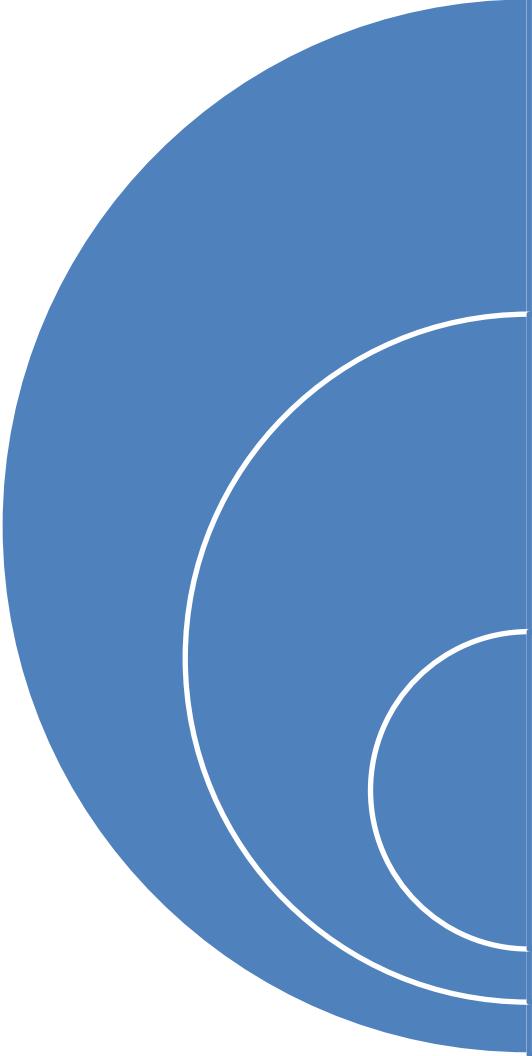
Control de concurrencia con locks

Un lock debe disponer de dos primitivas de uso, que permiten tomar $lock(X)$ $L(X)$ y liberar el recurso $unlock(X)$ $U(X)$.

Cuando una transacción tiene un lock sobre un ítem X , ninguna otra transacción puede adquirir un lock sobre el mismo hasta tanto la primera no lo libere.

$Lock(X)$ requiere escribir y leer una variable, por lo tanto no puede estar solapada con una ejecución similar en otra transacción.

Control de concurrencia con locks



Los tipos de locks principales son de lectura o “compartidos” y de escritura o de “acceso exclusivo”.

Cuando una transacción tiene un lock de acceso exclusivo ninguna otra transacción puede tener un lock de ningún tipo sobre él.

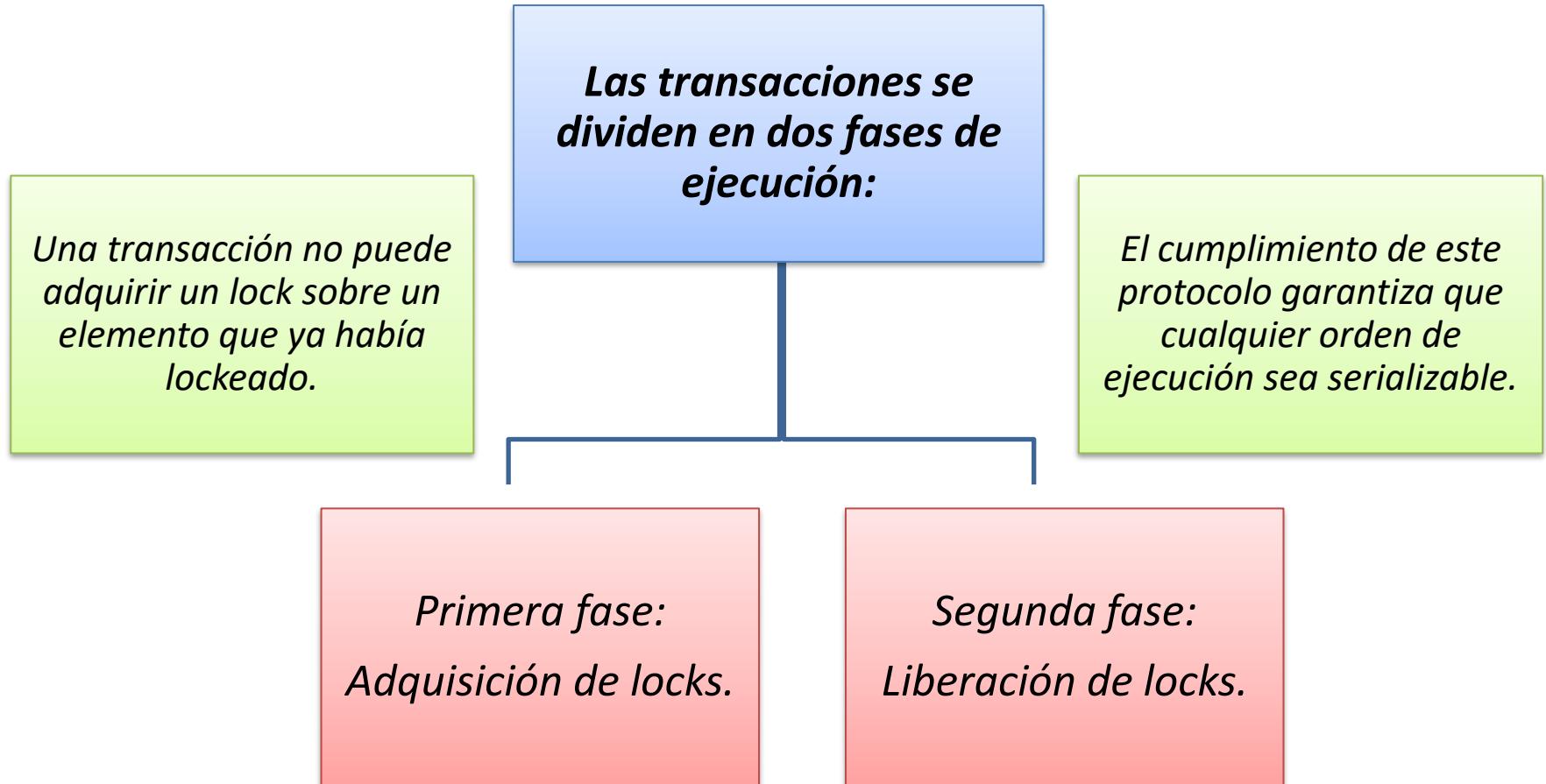
Pero muchas transacciones pueden poseer locks de acceso compartido simultáneamente.

Pero con los locks solo no alcanza...

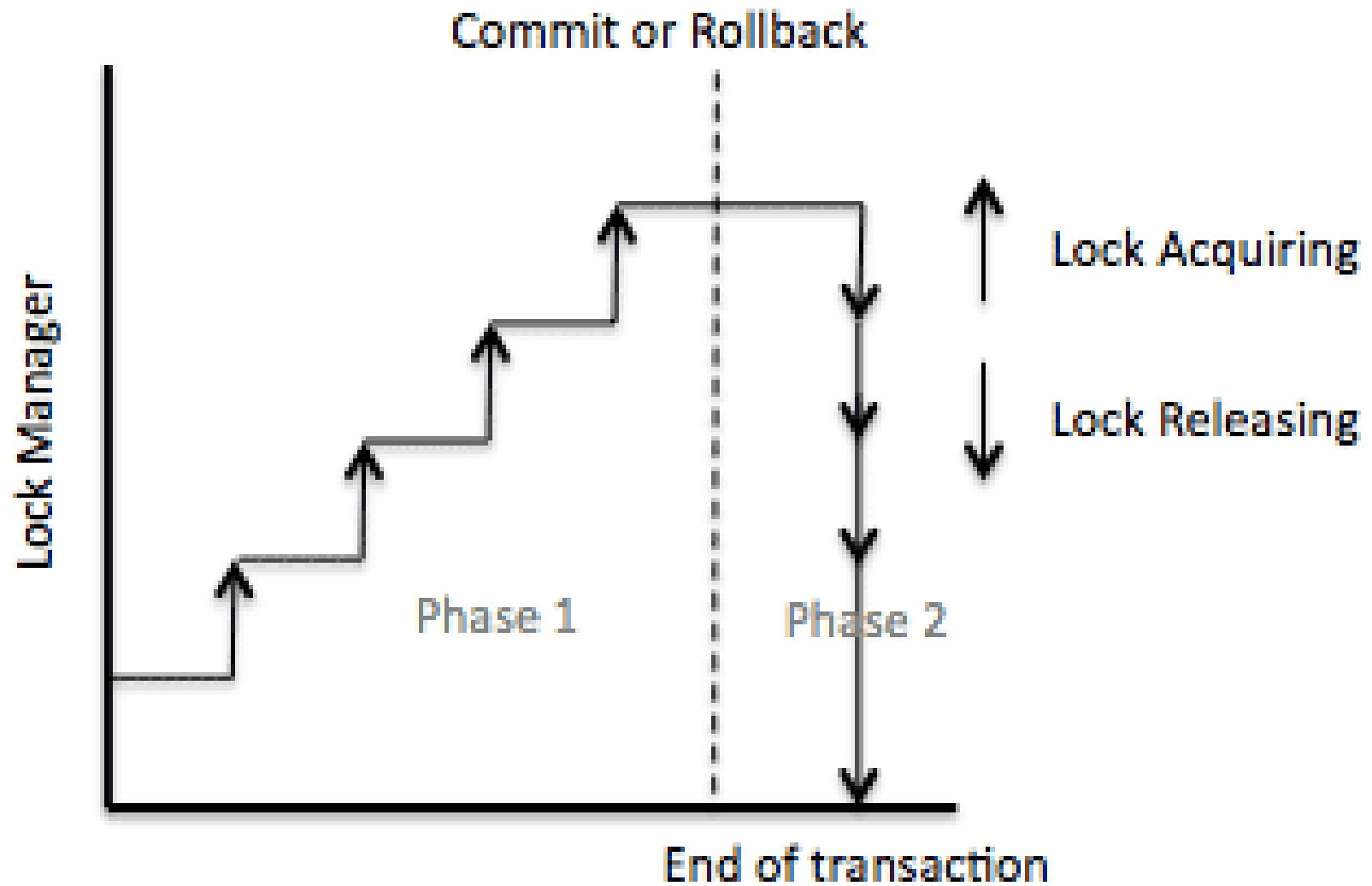
Una transacción podría adquirir un lock sobre un elemento para leerlo, liberarlo y luego volver a tomar un lock para modificarlo. Si en el medio otra transacción lo lee y escribe usando locks, seguramente se produciría alguna anomalía.

Entonces hace falta un protocolo que nos garantice la seriabilidad.

Protocolo de lock de dos fases básico 2PL



Protocolo de lock de dos fases básico 2PL



Ejemplo de 2PL básico

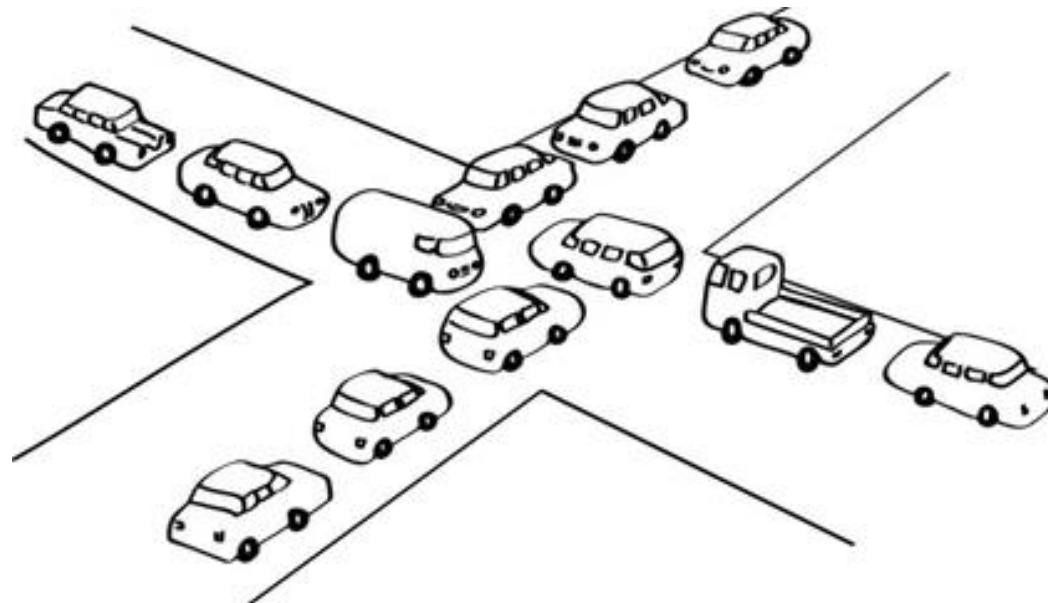
T1	T2	A	B
I(A); r(A) A = A +100 w(A); I(B); u(A)		25	25
r(B); B + B + 100 w(B); u(B)	I(A); r(A) A = A * 2 w(A) I(B) DENIED	125	
	I(B); u(A); r(B) B = B * 2 w(B); u(B)	250	125
			250

El resultado es el mismo que una ejecución serial.

Problemas...

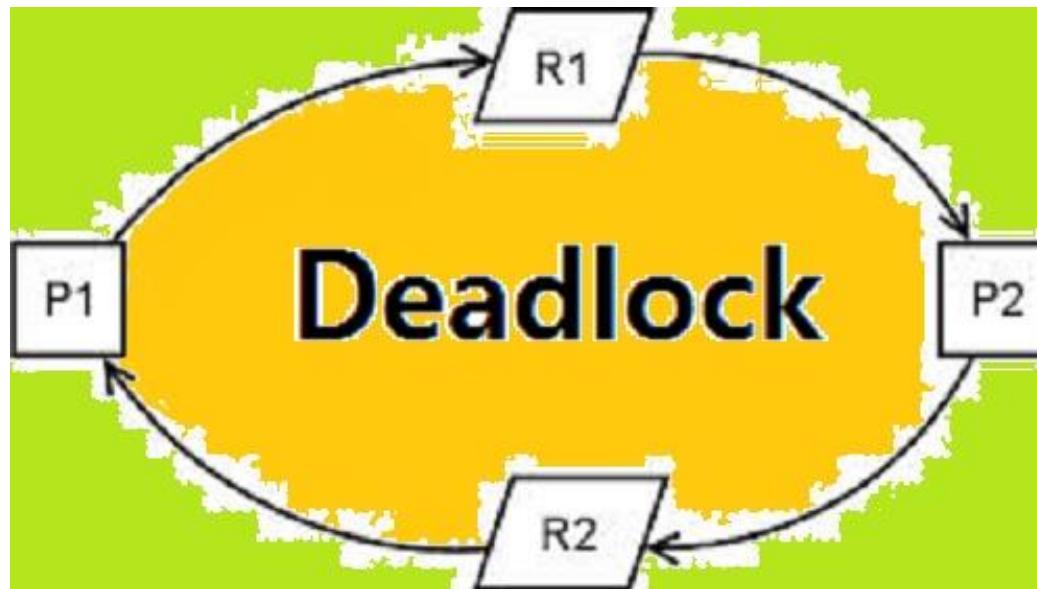
La utilización de locks introduce otros dos posibles problemas que antes no teníamos:

- *Bloqueo, interbloqueo o deadlock.*
- *Inanición, espera indefinida o livelock.*



Deadlock

Un deadlock se produce cuando un conjunto de transacciones queda cada una de ellas a la espera de recursos que otra de ellas ha bloqueado y ninguna puede continuar ejecutándose.



Ejemplo de deadlock

T0	T1
<pre>begin lock(A) leer(A) lock(B) leer(B)</pre>	<pre>begin lock(B) leer(B) lock(A) leer(A)</pre>

A diagram illustrating the deadlock situation. A central rounded rectangle contains the text "FALLAN AMBOS BLOQUEOS". Two blue arrows point from the bottom of this box to the "lock(B)" and "lock(A)" statements in their respective columns.

Protocolos de prevención de deadlocks

Existen otros protocolos que utilizan marcas de tiempo para administrar que transacción puede tomar los recursos o esperar, o en algunos casos abortar para evitar el deadlock.



Detección de deadlocks

Por medio del grafo de alocación de recursos podemos analizar si se produce un ciclo es que hay un deadlock.

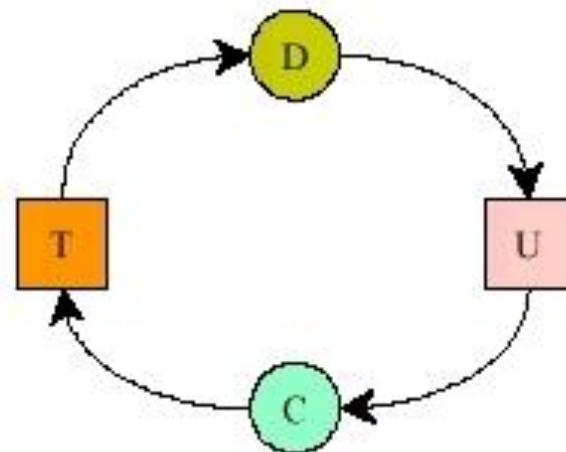
POSESION DE UN RECURSO



SOLICITUD DE UN RECURSO



BLOQUEO



Detección de deadlocks

Si una transacción lleva un tiempo de espera superior a un tiempo definido por el sistema o timeout, se aborta la transacción. Tiene la ventaja de ser muy simple.



Inanición

Es similar al deadlock, pero en este caso es una sola transacción que queda a la espera de recursos y no logre en un período de tiempo acceder a los mismos.

Una solución es dar prioridad por orden de llegada en los pedidos de locks.



Protocolos de prevención de deadlocks

2PL conservador: que cada transacción *bloquee con antelación* todos los elementos que necesita. Si alguno no puede bloquearlo, espere y reintenta. Limita la concurrencia.

-

Ordena todos los elementos de la base y se asegura de que una transacción que necesite varios elementos los bloqueará en ese orden. Obliga a conocer este orden.

-

Ejemplo de 2PL conservador

T1	T2	A	B
I(A); I(B) A = A +100 w(A); u(A) r(B); B + B + 100 w(B); u(B)	I(A); I(B) DENIED I(A); I(B) r(A) A = A * 2 w(A) u(A); r(B) B = B * 2 w(B); u(B)	25 125 250 125 250	25

El resultado es el mismo que una ejecución serial.

Problemas...

Aunque tengamos un solapamiento serializable de transacciones, si una transacción T_i es abortada, el SGBD debe mantener la consistencia de la base. Si las modificaciones hechas por T_i fueron leídas por otras transacciones, entonces será necesario hacer el rollback de todas las transacciones involucradas en una cascada de transacciones.

**Necesitamos
un nuevo
protocolo...**

Protocolo de lock de 2PL estricto y riguroso

Estricto S2P: *Una transacción no puede adquirir un lock luego de haber liberado uno que había adquirido, y los locks de escritura sólo pueden ser liberados después de haber commiteado la transacción.*

-

Riguroso R2PL: *no diferencia los tipos de locks. Los locks sólo pueden ser liberados después del commit.*

-

Protocolo de lock de 2PL estricto y riguroso

S2PL y R2PL garantizan que todo solapamiento sea no sólo serializable, sino también recuperable, y garantiza además que no se producirán cascadas de rollbacks al deshacer una transacción.

Pero no evitan los interbloqueos, por lo que se combinan con 2PL conservador o por marcas de tiempo.

Ejercicios

Organice las siguientes transacciones usando bloqueo de dos fases básico y conservador, con bloqueos que pueden ser exclusivos (escritura) o compartidos (lectura), minimizando el tiempo de atención (o inicio) promedio de cada transacción. Que pasaría en cada caso?

1) T1=(r(A),w(A),r(B),w(B),r(C),w(C))

T2=(r(B),w(B),r(C),w(C),r(A),w(A))

T3=(r(C),w(C),r(D),w(D),r(B),w(B))

2) T1=(r(A),r(B),w(A),r(C),w(C))

T2=(r(B),r(A),w(A),w(B))

3) T1=(r(A),r(B),w(B),w(C))

T2=(r(D),r(E),w(E),w(D))

T3=(r(B),r(D),w(B),w(D))

T4=(w(A),w(B),w(C),w(D))

4) T1=(r(A),w(A),w(B),r(C),w(C))

T2=(r(C),w(C))

T3=(r(B),w(B),w(C),r(D),w(D))

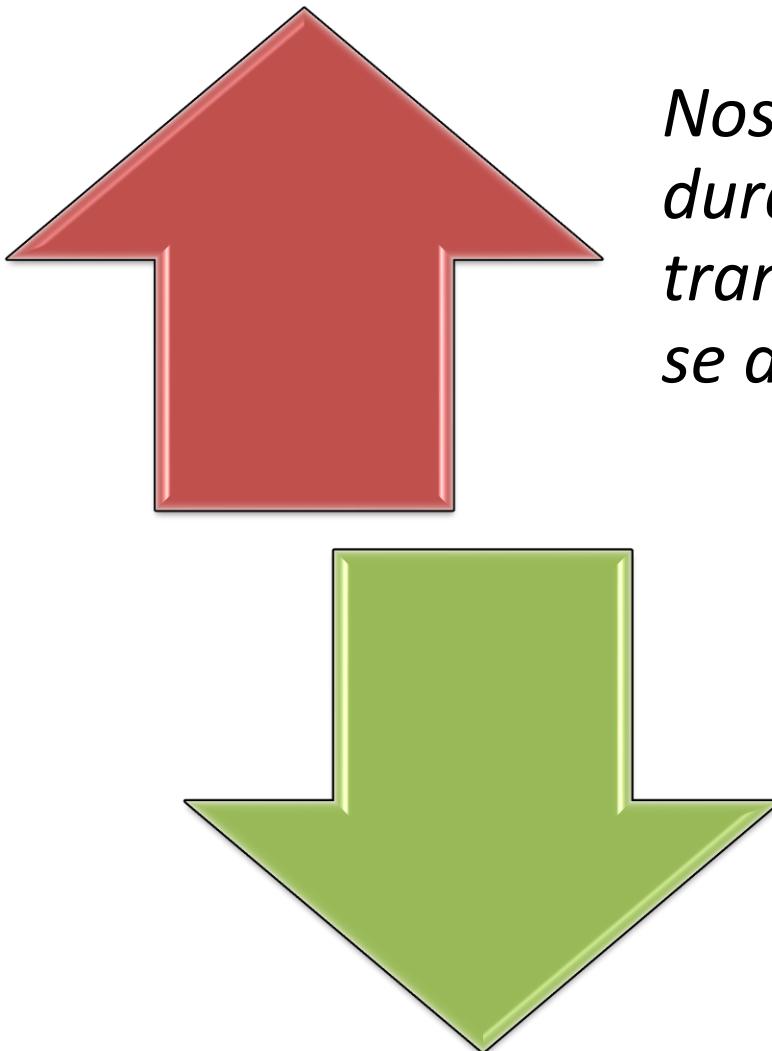
5) T1=(r(A),r(B),r(C),w(B),w(C))

T2=(r(A),r(B),r(D),w(B),w(D))

Recuperabilidad



Recuperabilidad

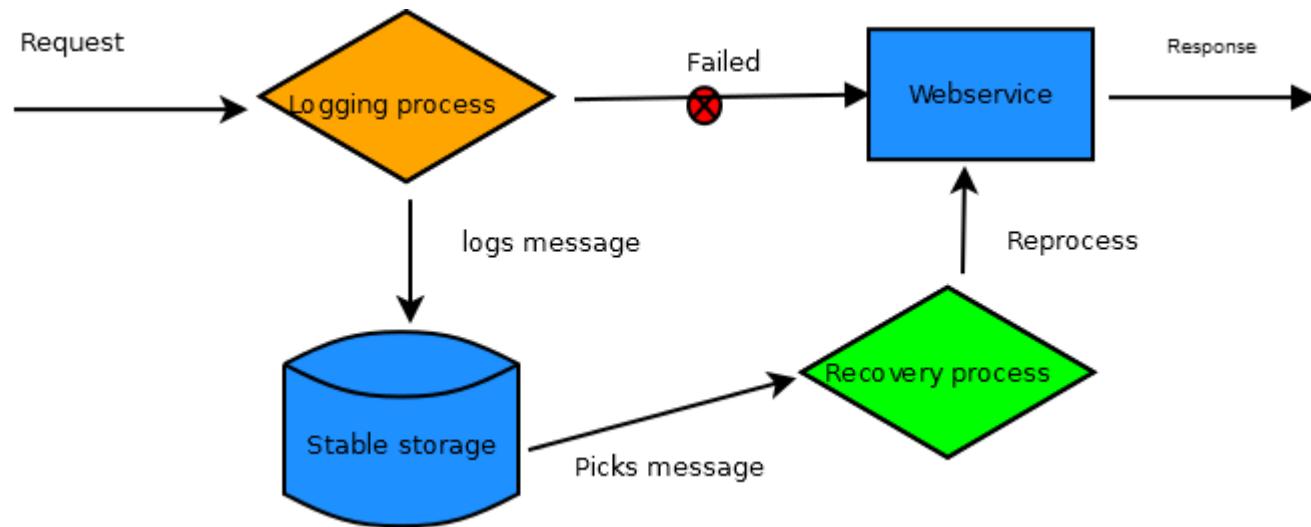


Nos interesa que se cumpla la durabilidad, esto es que una transacción que commitió no se deshaga.

Un solapamiento es recuperable si y sólo si ninguna transacción T_i realiza el commit hasta tanto todas las transacciones que escribieron datos antes de que T_i los leyera hayan commitreado.

Recuperabilidad

Dado un solapamiento recuperable, puede ser necesario abortar (deshacer) una transacción antes de llegar a su commit, y para ello el SGBD deberá contar con una serie de información que se almacena por su gestor de recuperación en un log o bitácora.



Datos almacenados en el log

(BEGIN T_i): indica que T_i comenzó.

(WRITE, T_i , X , X_{old} , X_{new}): indica que la transacción T_i escribió X , cambiando X_{old} por X_{new} .

(READ, T_i , X): indica que la transacción T_i leyó X .

(COMMIT, T_i): indica que la transacción T_i commitió.

(ABORT, T_i): indica que la transacción T_i abortó.

Con esta información se puede deshacer los efectos de una transacción en el momento de hacer un rollback.

Aislamiento en SQL



Grados de aislamiento en SQL

Hay distintos niveles de aislamiento que podemos parametrizar en SQL.

SET TRANSACTION

ISOLATION LEVEL [READ UNCOMMITTED |
READ COMMITTED | REPEATABLE | SERIALIZABLE]

Cada opción tiene su costo.

READ UNCOMMITTED

Permite hacer lecturas sucias (dirty reads), donde las consultas dentro de una transacción son afectadas por cambios no confirmados (not committed) de otras transacciones.

Esta opción es apenas transaccional, es como no tener transacciones.

READ UNCOMMITTED (dirty reads)

T0	T1
<pre>SELECT COUNT(*) AS numerosas FROM personal WHERE hijos > 3; numerosas ----- 15</pre>	<pre>UPDATE personal SET hijos=4 WHERE legajo=32;</pre>
<pre>SELECT COUNT(*) AS numerosas FROM personal WHERE hijos > 3; numerosas ----- 16</pre>	<pre>ROLLBACK;</pre>

READ COMMITTED (Non-repeatable reads)

Los cambios confirmados son visibles dentro de otra transacción, esto significa que dos consultas dentro de una misma transacción pueden retornar diferentes resultados. Generalmente este es el comportamiento por defecto en los SGBD.

READ COMMITTED (Non-repeatable reads)

T0	T1
<pre>SELECT COUNT(*) AS numerosas FROM personal WHERE hijos > 3; numerosas ----- 15</pre>	<pre>UPDATE personal SET hijos=4 WHERE legajo=32;</pre>
<pre>SELECT COUNT(*) AS numerosas FROM personal WHERE hijos > 3; numerosas ----- 15</pre>	<pre>COMMIT;</pre>

READ COMMITTED (Non-repeatable reads)

T0	T1
<pre>SELECT COUNT(*) AS numerosas FROM personal WHERE hijos > 3; numerosas ----- 15</pre>	<pre>UPDATE personal SET hijos=4 WHERE legajo=32; COMMIT;</pre>
<pre>SELECT COUNT(*) AS numerosas FROM personal WHERE hijos > 3; numerosas ----- 16</pre>	

REPEATABLE READS (Phantom reads)

Dentro de una transacción todas las lecturas son consistentes. En este nivel de aislamiento, el SGBD implementa el control de concurrencia basado en bloqueos, mantiene los bloqueos de lectura y escritura -de los datos seleccionados- hasta el final de la transacción. Sin embargo, no se gestionan los bloqueos de rango, por lo que las lecturas fantasma pueden ocurrir

REPEATABLE READS (Phantom reads)

T0	T1
<pre>SELECT COUNT(*) AS numerosas FROM personal WHERE hijos > 3; numerosas ----- 15</pre>	<pre>UPDATE personal SET hijos=4 WHERE legajo=32; COMMIT;</pre>
<pre>SELECT COUNT(*) AS numerosas FROM personal WHERE hijos > 3; numerosas ----- 15</pre>	<div style="border: 1px solid black; padding: 10px; width: fit-content; margin-left: auto;"><p>NO SE PUEDE REALIZAR PORQUE LOS DATOS ESTÁN BLOQUEADOS POR T0</p></div>

REPEATABLE READS (Phantom reads)

T0	T1
<pre>SELECT COUNT(*) AS numerosas FROM personal WHERE hijos > 3; numerosas ----- 15</pre>	<pre>INSERT INTO personal (legajo, hijos) VALUES (1302,4); COMMIT;</pre>
<pre>SELECT COUNT(*) AS numerosas FROM personal WHERE hijos > 3; numerosas ----- 16</pre>	

SERIALIZABLE

No se permiten actualizaciones en otras transacciones si una transacción ha realizado una consulta sobre ciertos datos. En este caso las distintas transacciones no se afectan entre sí.

Las transacciones están completamente aisladas entre sí, lo que conlleva un costo asociado.

Que puede hasta detener todos los procesos!!

SERIALIZABLE

T0	T1
<pre>SELECT COUNT(*) AS numerosas FROM personal WHERE hijos > 3; numerosas ----- 15</pre>	 <p>NO SE PUEDE REALIZAR NINGUNA TRANSACCION PORQUE LOS DATOS ESTÁN BLOQUEADOS POR T0</p>
<pre>SELECT COUNT(*) AS numerosas FROM personal WHERE hijos > 3; numerosas ----- 15</pre>	

En resumen

Nivel de aislamiento	Lectura sucia	Lectura no repetible	Lectura fantasma
READ UNCOMMITTED	SI	SI	SI
READ COMMITTED	--	SI	SI
REPEATABLE READ	--	--	SI
SERIALIZABLE	--	--	--