

Concurrencia y Transacciones

(... o bien, transacciones y concurrencia ...)

Universidad de los Andes

Demián Gutierrez

Enero 2009

A transaction is a **complete unit of work**. It may comprise many computational tasks, which may include user interface, data retrieval, and communications. A typical transaction modifies shared resources.

DistributedTransactionProcessing_TheXA_Specification.PDF

Es una **unidad lógica de trabajo** (procesamiento) 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***”

iniciar T0

... operaciones ...

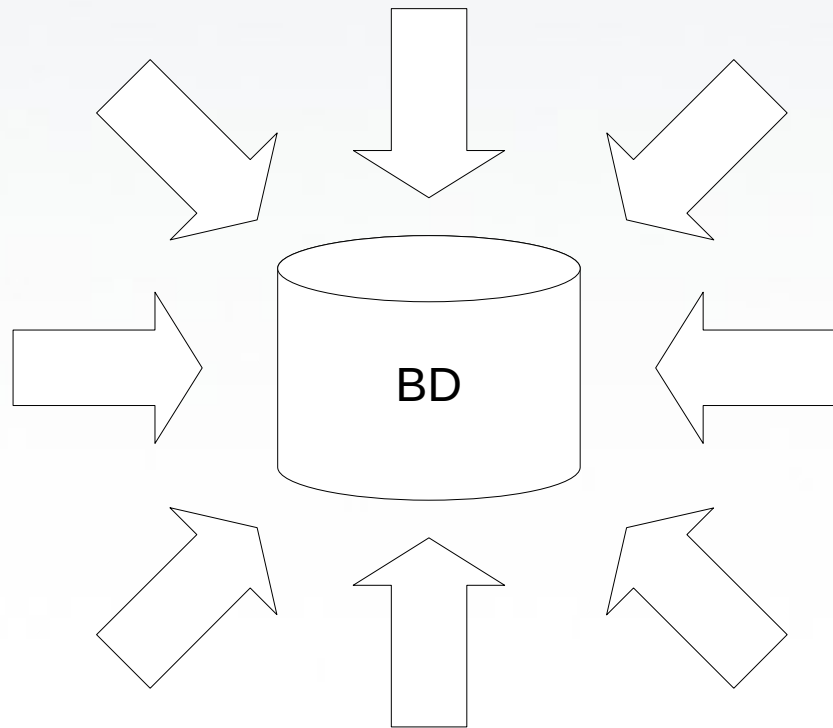
terminar T0

Además, en las transacciones tenemos operaciones básicas (“*leer elemento*”, “*escribir elemento*”), y cálculos sobre los datos leídos.

Las transacciones tienen otras propiedades “*deseables*” que veremos más adelante

```
iniciar T0
leer(A)
leer(B)
A = A + B
B = B * 1.1
escribir(A)
escribir(B)
terminar T0
```

Es cuando muchas transacciones acceden a la misma Base de Datos al mismo tiempo. Especialmente, cuando acceden a los mismos datos de la misma Base de Datos al mismo tiempo



Sean T0 y T1 dos transacciones:

T0:

```
leer(A)
A = A - 50
escribir(A)
leer(B)
B = B + 50
escribir(B)
```

T1:

```
leer(A)
temp = A * 0.1
A = A - temp
escribir(A)
leer(B)
B = B + temp
escribir(B)
```

Donde A y B son saldos de dos cuentas bancarias diferentes con valores de 1000 y 2000 BsF respectivamente
 $A + B = 3000$ BsF

T0 (Transf)	T1 (Gan. Premio)
leer(A)	leer(A)
A = A - 50	A = A + 100
escribir(A)	escribir(A)
leer(B)	
B = B + 50	
escribir(B)	

Esta lectura se hace antes de que T0 tenga oportunidad de actualizar A

Esta escritura hace que el valor de A de T0 se pierda

Actualización Perdida

T1 lee el valor de A antes de que T0 lo pueda actualizar, luego T0 escribe A, pero después T1 sobre escribe A con un valor incorrecto. Como consecuencia la actualización que hizo T0 se pierde.

T0	T1
leer(A)	
$A = A - 50$	
escribir(A)	
	leer(A)
	$A = A + 100$
	escribir(A)
leer(B)	
...	
¡T0 aborta!	

T0 va a realizar sus cálculos con un valor de A que no es válido porque T0 aborta

Actualización Temporal (Lectura Sucia)

T1 lee el valor de A luego de que T0 lo escribió, y realiza cálculos en base a dicho valor de A. Sin embargo, luego T0 aborta, por lo tanto el valor de A leído por T1 ya no es válido

T0	T1
<pre>leer(A) A = A - 50 escribir(A)</pre>	<pre>suma = 0</pre>
<pre>leer(B) B = B + 50 escribir(B)</pre>	<pre>leer(A) suma = suma + A leer(B) suma = suma + B</pre>

La suma se debería hacer completamente antes o después de T0, pero no en el medio

Resumen Incorrecto

T1 está calculando la suma (o cualquier otra función agregada) con un valor correcto de A, pero con un valor incorrecto (anterior) de B

```
postgres=# BEGIN TRANSACTION;
```

```
BEGIN
```

```
postgres=# SELECT * FROM departamento WHERE codigo=1;
```

```
codigo | nombre
-----+-----
        1 | Computaion
(1 row)
```

```
postgres=# UPDATE departamento SET nombre='Informatica';
```

```
UPDATE 3
```

```
postgres=# COMMIT;
```

```
COMMIT
```

```
postgres=# SELECT * FROM departamento WHERE codigo=1;
```

```
codigo | nombre
-----+-----
        1 | Informatica
(1 row)
```

```
postgres=#
```

Para que esto funcione en MySQL las tablas deben utilizar **InnoDB** y no **MyISAM**
(Motores de almacenamiento usados por MySQL)

```
postgres=# BEGIN TRANSACTION;
```

```
BEGIN
```

```
postgres=# SELECT * FROM departamento WHERE codigo=1
```

```
codigo | nombre  
-----+-----  
      1 | Informatica  
(1 row)
```

```
postgres=# UPDATE departamento SET nombre='Computacion';
```

```
UPDATE 3
```

```
postgres=# ROLLBACK;
```

```
ROLLBACK
```

```
postgres=# SELECT * FROM departamento WHERE codigo=1;
```

```
codigo | nombre  
-----+-----  
      1 | Informatica  
(1 row)
```

```
postgres=#
```

El dato no actualizó porque abortamos
(**ROLLBACK**) la transacción

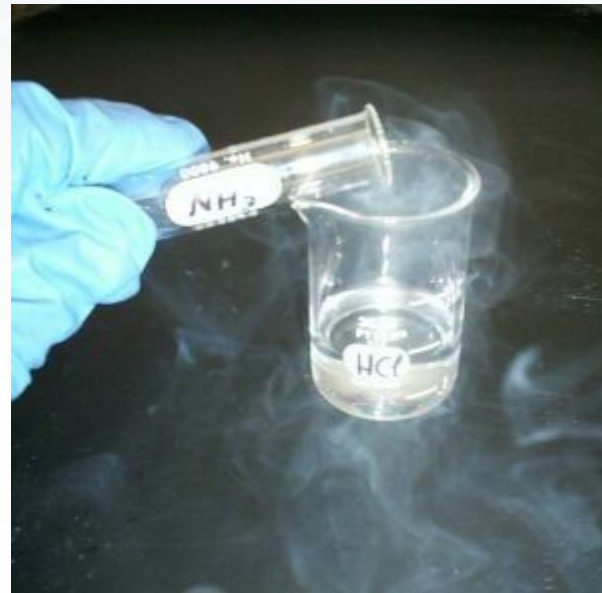
ACID:

Atomicity (Atomicidad)

Consistency (Consistencia)

Isolation (Aislamiento)

Durability (Durabilidad / Permanencia)



READ UNCOMMITTED:

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

READ UNCOMMITTED (dirty reads)

Transaction 1

```
/* Query 1 */  
SELECT * FROM users WHERE id = 1;
```

```
/* Query 1 */  
SELECT * FROM users WHERE id = 1;
```

¿Qué resultado tendremos luego del primer select?
¿y luego del segundo?

Transaction 2

```
/* Query 2 */  
UPDATE users SET age = 21 WHERE id = 1;  
/* No commit here */
```

```
ROLLBACK; /* lock-based DIRTY READ */
```

READ COMMITTED:

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)

Transaction 1

```
/* Query 1 */  
SELECT * FROM users WHERE id = 1;
```

```
/* Query 1 */  
SELECT * FROM users WHERE id = 1;  
COMMIT; /* lock-based REPEATABLE READ */
```

Transaction 2

```
/* Query 2 */  
UPDATE users SET age = 21 WHERE id = 1;  
COMMIT; /* in multiversion concurrency  
control, or lock-based READ COMMITTED */
```

¿Qué resultado
tendremos luego
del primer select?
¿y luego del
segundo?

REPEATABLE READ:

Dentro de una transacción todas las lecturas son consistentes (Esto es el comportamiento por defecto en MySQL usando tablas en InnoDB)

REPEATABLE READS (Phantom Reads)

Transaction 1

```
/* Query 1 */  
SELECT * FROM users  
WHERE age BETWEEN 10 AND 30;
```

```
/* Query 1 */  
SELECT * FROM users  
WHERE age BETWEEN 10 AND 30;
```

Transaction 2

```
/* Query 2 */  
INSERT INTO users VALUES ( 3, 'Bob', 27 );  
COMMIT;
```

¿La primera y la segunda consulta retornan la misma cantidad de registros?
¿Deberían?

SERIALIZABLE:

No se permiten las actualizaciones en otras transacciones si una transacción ha realizado una consulta sobre ciertos datos (las distintas transacciones no se afectan entre si)

Las transacciones están completamente aisladas entre si (Esto tiene un costo asociado...)

En resumen:

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read
Read uncommitted	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible
Repeatable read	Not possible	Not possible	Possible
Serializable	Not possible	Not possible	Not possible

Sin embargo, PostgreSQL sólo soporta “Read Committed” y “Serializable”

“In PostgreSQL, you can request any of the four standard transaction isolation levels. But internally, **there are only two distinct isolation levels**, which correspond to the levels **Read Committed and Serializable**. When you select the level **Read Uncommitted** you really get **Read Committed**, and when you select **Repeatable Read** you really get **Serializable**, so the actual isolation level may be stricter than what you select.”

<http://www.postgresql.org/docs/8.1/static/transaction-iso.html>

Transacciones y nivel de aislamiento en SQL (PostgreSQL)

```
postgres=# BEGIN TRANSACTION;  
BEGIN  
postgres=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED READ WRITE;  
SET  
postgres=# -- Otras operaciones... ---  
UPDATE 3  
postgres=# COMMIT;  
COMMIT  
postgres=#
```

O ROLLBACK, según sea necesario

SET TRANSACTION

Name

SET TRANSACTION -- set the characteristics of the current transaction

Synopsis

```
SET TRANSACTION transaction_mode [, ...]  
SET SESSION CHARACTERISTICS AS TRANSACTION transaction_mode [, ...]
```

where *transaction_mode* is one of:

```
ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED }  
READ WRITE | READ ONLY
```

Transacciones y nivel de aislamiento en SQL (PostgreSQL)

```
test=# BEGIN;
BEGIN
test=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED READ WRITE;
SET
test=# SELECT * FROM FOO WHERE ID=1;
 id | val
----+-----
  1 | xxx
(1 row)

test=# SELECT * FROM FOO WHERE ID=1;
 id | val
----+-----
  1 | www
(1 row)

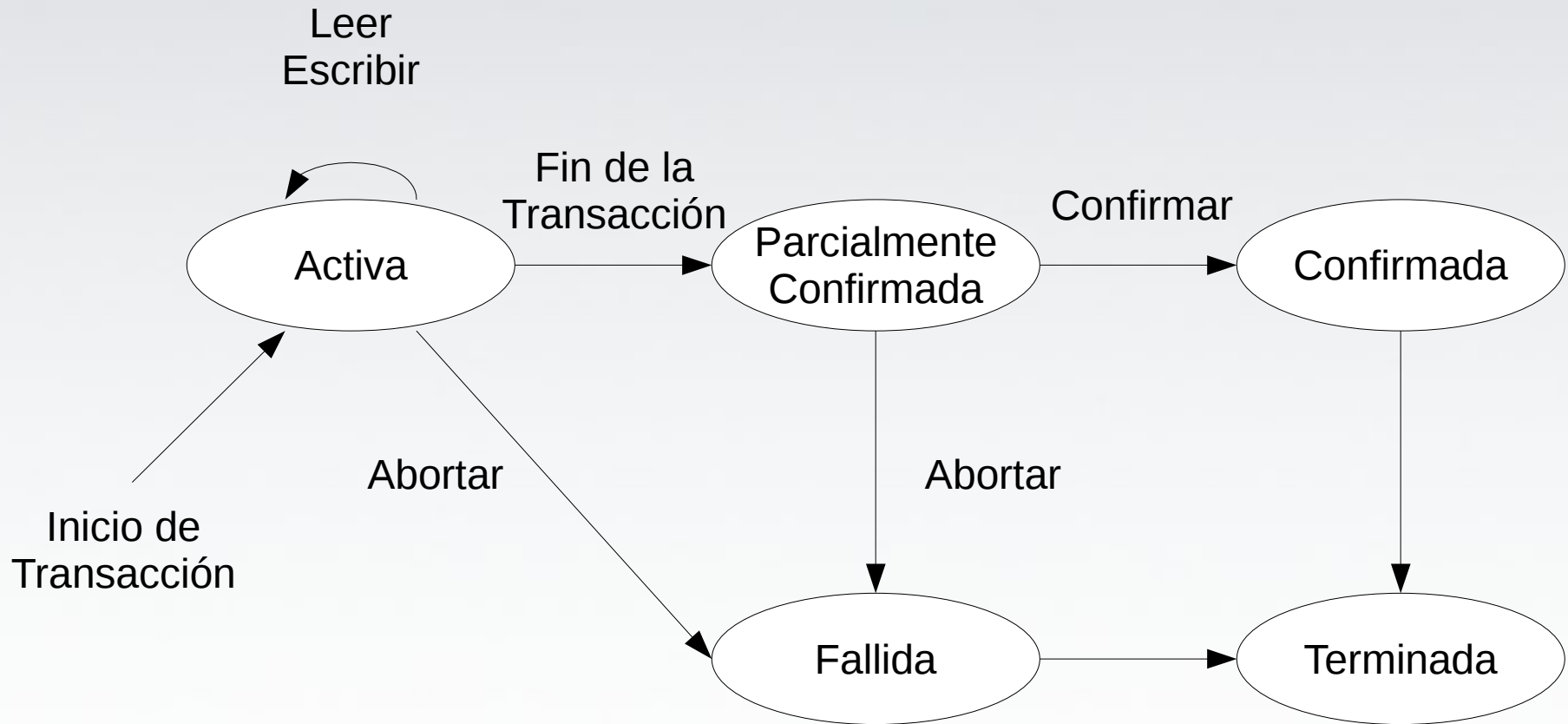
test=# COMMIT;
COMMIT
test=#
```

```
test=# BEGIN;
BEGIN
test=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED READ WRITE;
SET
test=# UPDATE foo SET val='www' WHERE id=1;
UPDATE 1
test=# COMMIT;
COMMIT
test=# □
```

¿Cómo logran los
SGBD esto?

En general, las transacciones tienen una serie de estados, que permiten hacerle un adecuado seguimiento

Estados de una Transacción



Confirmar = Commit
Abortar = Rollback

Una planificación representa el orden cronológico en que se ejecutan o se han ejecutado las instrucciones de un conjunto de transacciones en el SGBD

Una planificación para una transacción debe conservar todas las instrucciones de la transacción. Además, se debe conservar el orden de las instrucciones dentro de la transacción

Plan:

$\{T1, L(A)\}, \{T0, L(A)\}, \{T0, E(A)\}, \{T0, L(B)\}$
 $\{T1, E(A)\}, \{T0, E(B)\}, \{T1, L(B)\}, \{T1, E(B)\}$

Un plan en serie (serial) es aquel en que las transacciones se ejecutan completas en secuencia una detrás de otra

Un plan no en serie (no serial, o plan intercalado) es aquel en el que se intercalan de forma simultánea instrucciones de distintas transacciones

Plan en serie



Plan no en serie
(Intercalado)



Planificación en Serie y en Paralelo

T0	T1
leer(A)	
A = A - 50	
escribir(A)	
leer(B)	
B = B + 50	
escribir(B)	
	leer(A)
	temp = A * 0.1
	A = A - temp
	escribir(A)
	leer(B)
	B = B + temp
	escribir(B)

Planificación en Serie
(T0 primero, T1 después, una detrás de la otra)

El plan en serie es aquel en que **se ejecutan las transacciones en secuencia, una atrás de la otra** (nunca genera problemas)

Los valores finales de A y B son 855 y 2145 BsF respectivamente

$$A + B = 3000 \text{ BsF}$$

n! posibles planificaciones en serie (n = número de transacciones)

Planificación en Serie y en Paralelo

T0	T1
	leer(A)
	temp = A * 0.1
	A = A - temp
	escribir(A)
	leer(B)
	B = B + temp
	escribir(B)
leer(A)	
A = A - 50	
escribir(A)	
leer(B)	
B = B + 50	
escribir(B)	

Planificación en Serie
(T1 primero, T0 después, una detrás de la otra)

Los valores finales de A y B son 850 y 2150 BsF respectivamente
 $A + B = 3000$ BsF

Planificación en Serie y en Paralelo

T0	T1
leer(A) A = A - 50 escribir(A)	leer(A) temp = A * 0.1 A = A - temp escribir(A)
leer(B) B = B + 50 escribir(B)	leer(B) B = B + temp escribir(B)

La planificación es concurrente, y si bien no está en serie, el plan es "serializable"

La planificación NO está en serie (operaciones de T0 y T1 intercaladas)

Los valores finales de A y B son 855 y 2145 BsF respectivamente
A + B = 3000 BsF **(Mismo resultado que una en serie)**
posibles planificaciones en serie > n! (n = número de transacciones)

Planificación en Serie y en Paralelo

T0	T1
leer(A)	leer(A)
A = A - 50	temp = A * 0.1
	A = A - temp
	escribir(A)
escribir(A)	leer(B)
leer(B)	
B = B + 50	
escribir(B)	B = B + temp
	escribir(B)

Los planes en serie siempre son seguros (No presentan anomalías) mientras que no todos los planes intercalados son seguros

Los planes en serie suelen ser poco eficientes (*¿por qué?*) mientras que los planes intercalados suelen ser más eficientes (y realistas)

Los valores finales de A y B son 950 y 2100 BsF respectivamente

$A + B = 3050$ BsF (*¡El banco perdió 50 BsF!*)

En estos casos se dice que la planificación no es serializable

Operaciones Conflictivas...

- **{T0, leer(A)}, {T1, leer(A)}**: El orden no importa, ya que T0 y T1 leen el mismo valor de A.
- **{T0, escribir(A)}, {T1, escribir(A)}**: El orden no importa para T0 o T1, pero se producen problemas posteriormente porque el valor escrito por T0 se sobrescribe por T1

Operaciones Conflictivas...

- **{T0, leer(A)}, {T1, escribir(A)}**: El orden si importa, porque T0 está leyendo un valor que deja de ser válido cuando T1 realiza la escritura.
- **{T0, escribir(A)}, {T1, leer(A)}**: Aplican las mismas consideraciones que en el caso anterior, aun cuando esta combinación en concreto no genera anomalías.

Se dice que dos operaciones I_i e I_j de un plan S están en conflicto si operan sobre el mismo dato, pertenecen a transacciones distintas y al menos una de ellas es escribir

Si dos operaciones NO están en conflicto entonces **es posible** intercambiar el orden en que se ejecutan sin generar ningún tipo de anomalía

Es posible llevar una planificación paralela a un plan en serie equivalente.

Para esto, es necesario analizar las transacciones y las posibles operaciones en conflicto que pueden aparecer.

Es posible intercambiar el orden en el que se ejecutan dos operaciones que NO están en conflicto y de esta manera, es posible que se pueda obtener un plan en serie equivalente

Planes Equivalentes, Serialización (Por Conflicto)

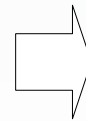
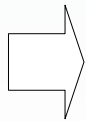
T0	T1
leer(A) escribir(A)	
leer(B) escribir(B)	leer(A) escribir(A)
	leer(B) escribir(B)



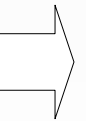
T0	T1
leer(A) escribir(A)	
leer(B) escribir(B)	leer(A) escribir(A)
	leer(B) escribir(B)



T0	T1
leer(A) escribir(A) leer(B)	
escribir(B)	leer(A) escribir(A)
	leer(B) escribir(B)



T0	T1
leer(A) escribir(A) leer(B)	
escribir(B)	leer(A) escribir(A)
	leer(B) escribir(B)



Planes Equivalentes, Serialización (Por Conflicto)

T0	T1
leer(A)	
escribir(A)	
leer(B)	
escribir(B)	
	leer(A)
	escribir(A)
	leer(B)
	escribir(B)

El orden de las operaciones en conflicto sigue siendo el mismo en este plan y en el anterior (Es decir, el orden de las operaciones en conflicto se conserva)

Dos planes son equivalentes por conflicto si el orden de las operaciones en conflicto entre estos dos planes es exactamente el mismo en ambos planes

Planes Equivalentes, Serialización (Por Conflicto)

T0	T1
leer(A)	
A = A - 50	
	leer(A)
	temp = A * 0.1
	A = A - temp
	escribir(A)
	leer(B)
escribir(A)	
leer(B)	
B = B + 50	
escribir(B)	
	B = B + temp
	escribir(B)

No todos los planes paralelos tienen un plan en serie equivalente...
¿Por qué este en particular NO es serializable?

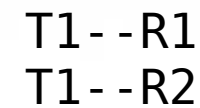
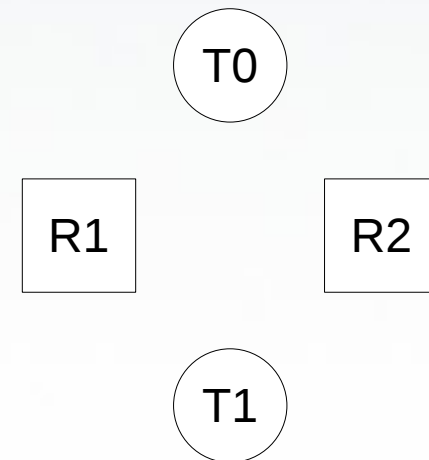
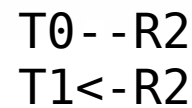
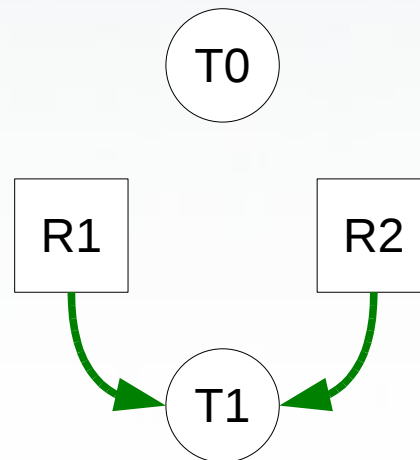
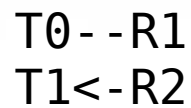
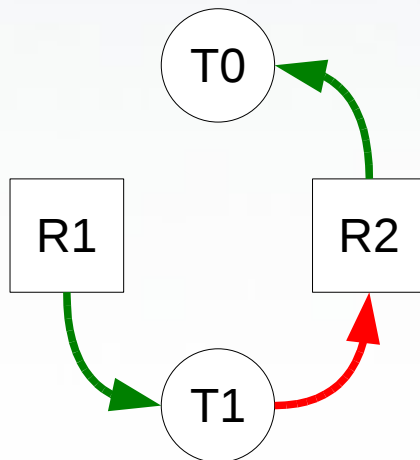
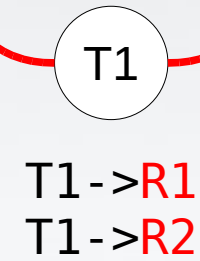
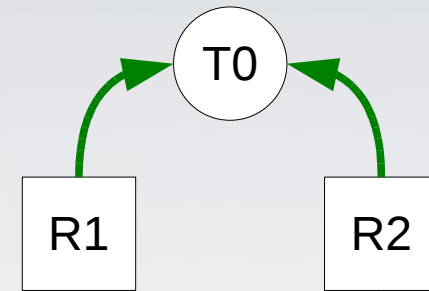
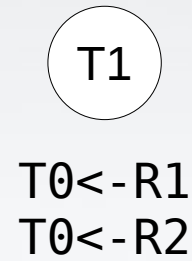
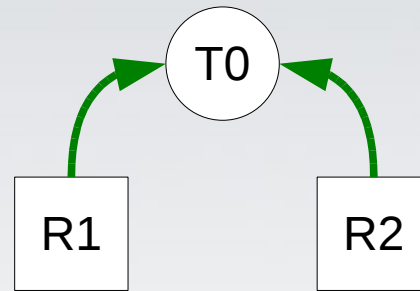
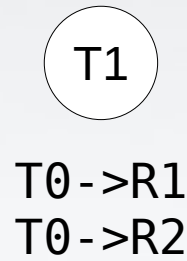
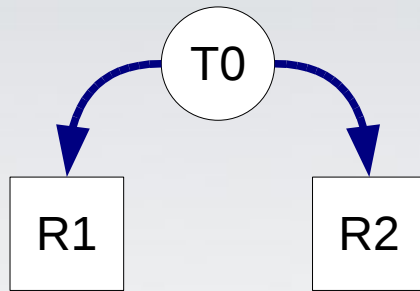
En teoría, si se verifica la seriabilidad de los planes, entonces no se producirá ninguna de las anomalías mostradas anteriormente

Sin embargo, en la práctica, comprobar la seriabilidad de los planes no es posible por razones de rendimiento y porque usualmente no se conocen de antemano todas las operaciones (y la secuencia de estas) que realizará una transacción (o si esta abortará o terminará satisfactoriamente)

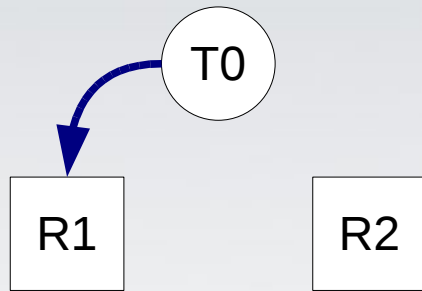
Normalmente, los SGBD en lugar de verificar la seriabilidad implementan “protocolos” que “garantizan” que los planes resultantes son “serializables”:

- **Protocolos Basados en Bloqueos:** Exigen que el acceso a los datos se haga de forma mutuamente excluyente.
- **Protocolos Basados en Marcas de Tiempo:**
Utilizan marcas de tiempo para determinar el orden de serializabilidad (No vamos a entrar en detalles).

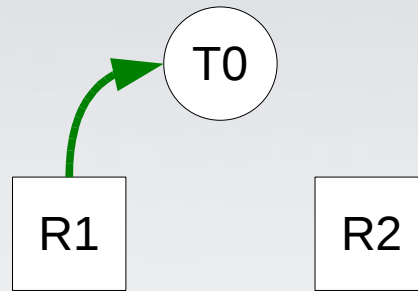
Bloqueo



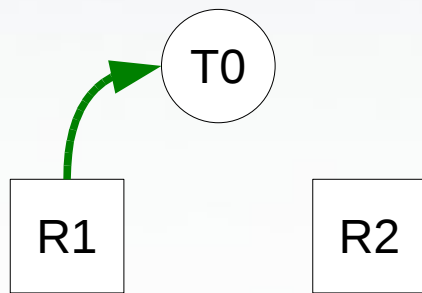
Bloqueo Mortal



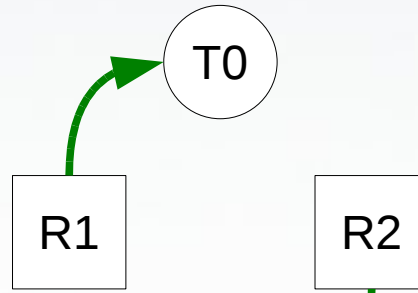
T1
T0 -> R1



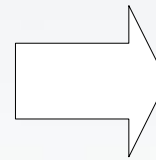
T1
T0 <- R1



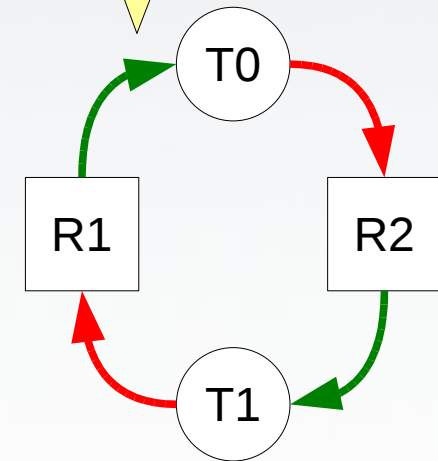
T1
T1 <- R2



T1
T1 <- R2

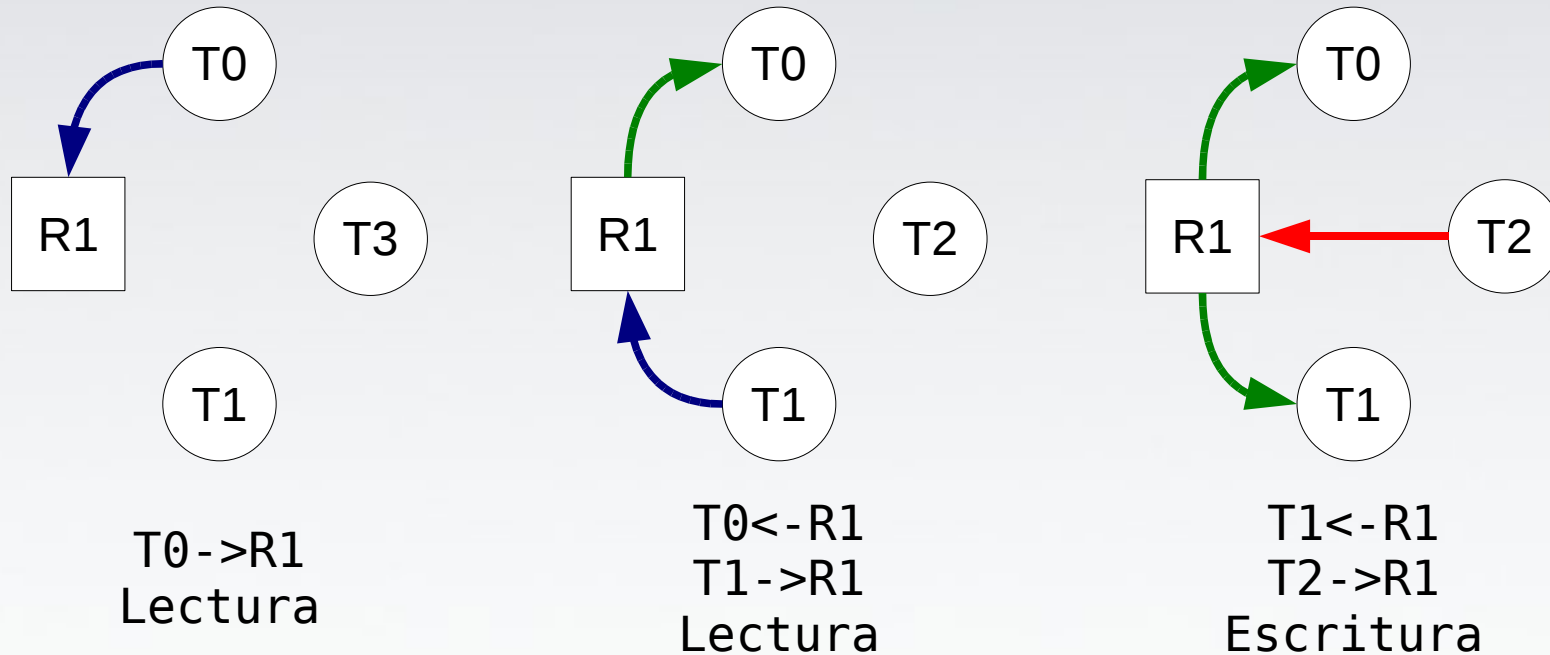


Hay un ciclo
(T0->R2->T1-> R1)
en el grafo, por lo
tanto hay un
bloqueo mortal



T0 -> R2
T1 -> R1

¿Como se resuelve?
¿Inanición?
¿Costo de Retroceso?



Hay distintos tipos de bloqueos:
Compartidos (de lectura) o
Exclusivos (de escritura)

Bloqueo (¿Que sale mal aquí?)

T0	T1
bloq-E(A) leer(A) A = A + 50 escribir(A) desbloq(A)	bloq-C(A) leer(A) desbloq(A) bloq-C(B) leer(B) desbloq(B) imprimir(A + B)
bloq-E(B) leer(B) B = B - 50 escribir(B) desbloq(B)	

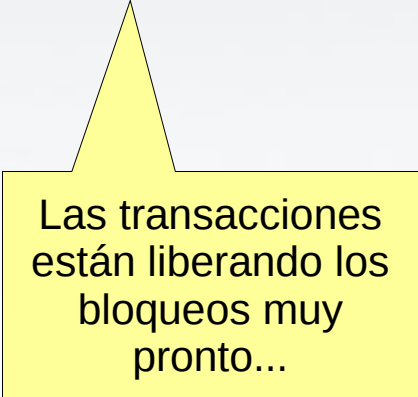
¿Cuanto debe valer A+B si los planes fueran en serie?

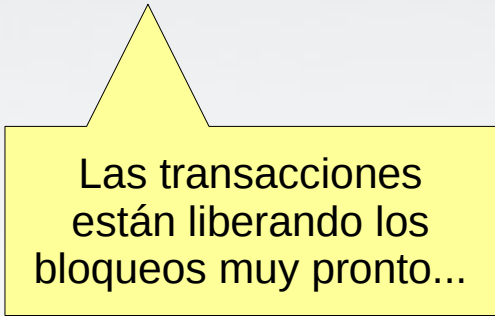
T0	T1
bloq-E(A) leer(A) A = A + 50 escribir(A) desbloq(A)	bloq-C(A) leer(A) desbloq(A) bloq-C(B) leer(B) desbloq(B) imprimir(A + B)
bloq-E(B) leer(B) B = B - 50 escribir(B) desbloq(B)	

¿Qué es lo que está funcionando mal?

Bloqueo

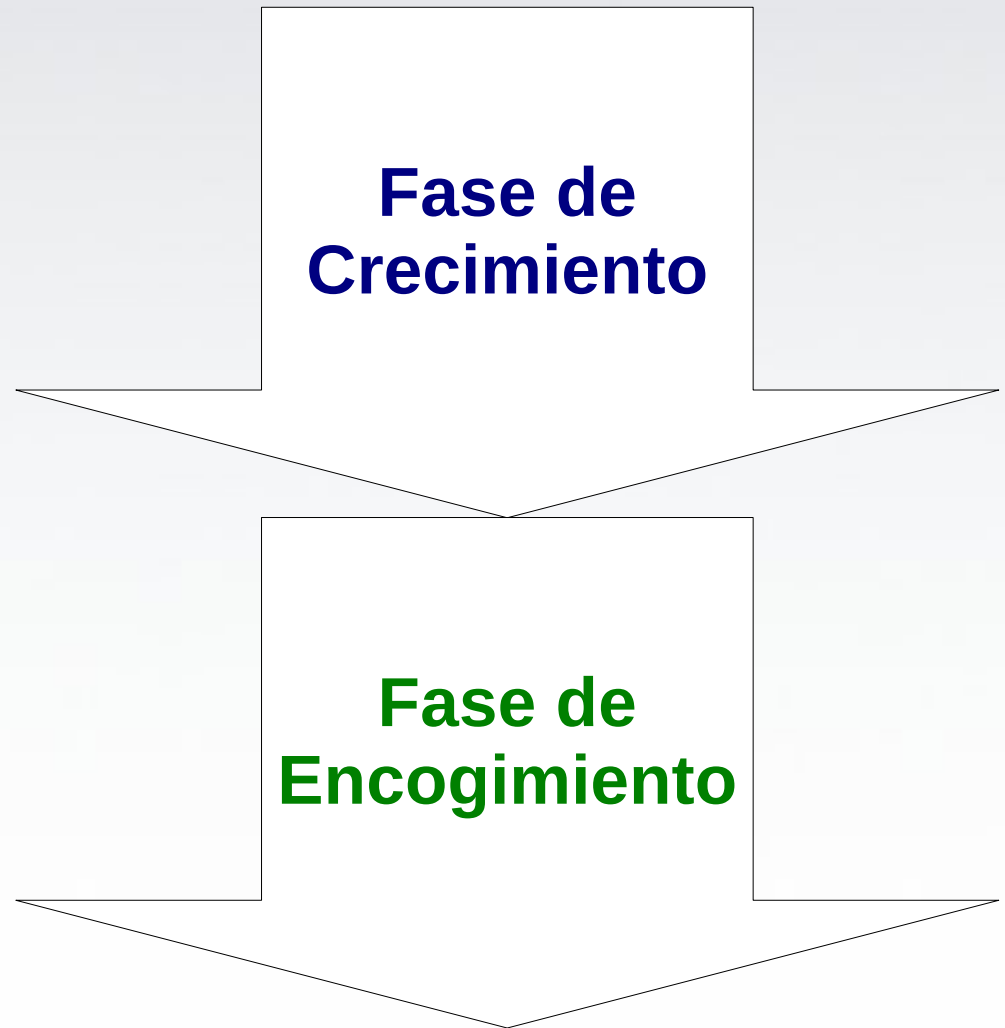
(¿Que sale mal aquí?)

T0	T1
bloq-E(A) leer(A) A = A + 50 escribir(A) desbloq(A)	
 <p>Las transacciones están liberando los bloqueos muy pronto...</p>	bloq-C(A) leer(A) desbloq(A) bloq-C(B) leer(B) desbloq(B) imprimir(A + B)
bloq-E(B) leer(B) B = B - 50 escribir(B) desbloq(B)	

T0	T1
	bloq-C(A) leer(A) desbloq(A)
bloq-E(A) leer(A) A = A + 50 escribir(A) desbloq(A)	 <p>Las transacciones están liberando los bloqueos muy pronto...</p>
bloq-E(B) leer(B) B = B - 50 escribir(B) desbloq(B)	bloq-C(B) leer(B) desbloq(B) imprimir(A + B)

Bloqueo en dos Fases (Solución al problema anterior)

T0	T1
bloq-E(A)	
leer(A)	
A = A + 50	
escribir(A)	
bloq-E(B)	
	bloq-C(A)
leer(B)	
B = B - 50	
escribir(B)	
desbloq(A)	
	leer(A)
	bloq-C(B)
desbloq(B)	
	leer(B)
	desbloq(B)
	imprimir(A + B)
	desbloq(A)



```
postgres=# BEGIN TRANSACTION;
...
postgres=# SELECT *
           FROM departamento
           WHERE codigo=1
           FOR UPDATE;
<resultados del select aquí>
...
...
...
...
postgres=# UPDATE departamento
           SET
           nombre='Computacion';
postgres=# COMMIT;
<termina y desbloquea>
...
...
...
...
```

```
...
postgres=# BEGIN TRANSACTION;
...
...
...
...
postgres=# SELECT *
           FROM departamento
           WHERE codigo=1
           FOR UPDATE;
<bloqueo hasta que termine T0>
...
...
...
...
<resultados del select aquí>
postgres=# UPDATE departamento
           SET
           nombre='Informatica';
postgres=# COMMIT;
```

Transacciones y Bloqueos en SQL (Bloqueos Mortales)

```
p=# BEGIN TRANSACTION;  
BEGIN  
p=# SELECT *  
      FROM departamento  
      WHERE codigo=1 FOR UPDATE;  
codigo | nombre  
-----+-----  
      1 | Informatica  
(1 row)
```

```
p=# SELECT *  
      FROM departamento  
      WHERE codigo=2 FOR UPDATE;  
codigo | nombre  
-----+-----  
      2 | Informatica  
(1 row)
```

```
p=# UPDATE departamento  
      SET nombre='Computacion';  
UPDATE 3  
postgres=# COMMIT;  
COMMIT
```

**AQUÍ FALTA UN
EJEMPLO DEL
UPDATE FOR
SHARE**

```
p=# BEGIN TRANSACTION;  
BEGIN  
p=# SELECT *  
      FROM departamento  
      WHERE codigo=2 FOR UPDATE;  
codigo | nombre  
-----+-----  
      2 | Informatica  
(1 row)
```

```
p=# SELECT *  
      FROM departamento  
      WHERE codigo=1 FOR UPDATE;  
ERROR:  deadlock detected  
DETAIL:  Process 3216 waits for  
ShareLock on  
transaction 19350;  
blocked by process 3219.  
Process 3219 waits for  
ShareLock on  
transaction 19351;  
blocked by process 3216.
```


Existen otras variantes y consideraciones a la hora de realizar bloqueos con el SGBD en SQL.

Para saber más consulte el manual de usuario del SGBD particular y cualquier otra documentación correspondiente.

Gracias

¡Gracias!



- Añadir:
 - El concepto de bitácora y estrategias elementales de recuperación cuando el sistema se cae (Silberschatz)
 - Hablar un poco más de los distintos estados de una transacción
 - Falta serialización por vistas (Incluirlo como parte de la clase o mandarlo como tarea)
 - Faltan protocolos basados en marcas de tiempo (incluirlo o dejarlo como tarea)
 - ¿Incluir bloqueo basado en índices y estructuras de acceso o similar?