

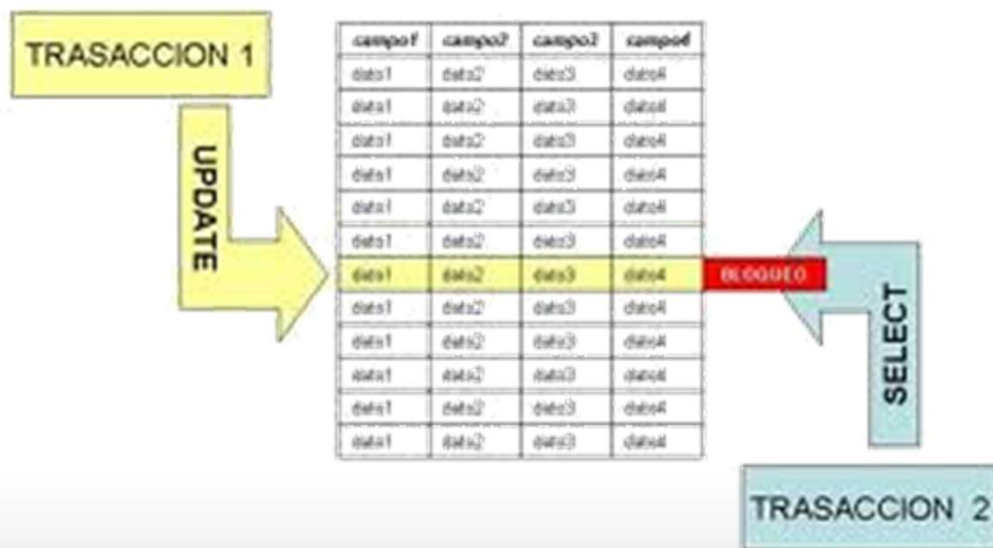


UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO
FACULTAD DE CIENCIAS
FUNDAMENTOS DE BASES DE DATOS

Transacciones

Gerardo Avilés Rosas
gar@ciencias.unam.mx

- Una transacción es una **unidad lógica de trabajo**.
- Manejar transacciones significa **asegurar que un conjunto de proposiciones SQL sea tratado como una entidad indivisible**.
- Las transacciones son necesarias para:
 - ✓ Permitir a los usuarios trabajar simultáneamente con la BD sin interferencia entre ellos.
 - ✓ Permitir que la BD se recupere a un estado consistente después de alguna falla de software.



Vamos a suponer que tenemos el siguiente esquema:

Ventas (bar, cerveza, precio)

que indica que una **cerveza** se vende a cierto **precio** en cierto **bar**.

- Vamos a suponer que el **Bar de Moe** vende solo **Duff** a \$450 y **Victoria** a \$400.
- **Homero** quiere preguntar por la cerveza más cara y más barata del **Bar de Moe**.
- Al mismo tiempo **Moe** elimina a **Duff** y **Victoria**, para comenzar a vender solo **Modelo** en \$500.



- En SQL, **Homero** ejecuta las instrucciones:

```
select max(precio) from Ventas where bar = 'MOE';  
select min(precio) from Ventas where bar = 'MOE';
```

que llamaremos **(max)** y **(min)** respectivamente.

- Por su parte **Moe** ejecuta:

```
delete from Ventas where bar = 'MOE';  
insert into Ventas values('MOE','Modelo',500);
```

que llamaremos **(del)** e **(ins)** respectivamente.

- Lo único que podemos asegurar con certeza es que **(max)** se ejecuta antes que **(min)** y que **(del)** se ejecuta antes que **(ins)**, pero nada más.
- ¿Qué pasaría si el orden de las instrucciones fuera **(max)**, **(del)**, **(ins)** y **(min)**.

```
select max(precio) from Ventas where bar = 'MOE';  
delete from Ventas where bar = 'MOE';  
insert into Ventas values('MOE','Modelo',500);  
select min(precio) from Ventas where bar = 'MOE';
```

- **Homero** lee cómo máximo el valor de **Duff** que es de \$450 y finalmente lee como precio mínimo el precio de **Modelo** que es de \$500... **¡¡¡el máximo es el menor que el mínimo!!!**



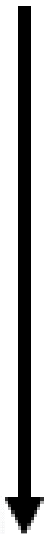
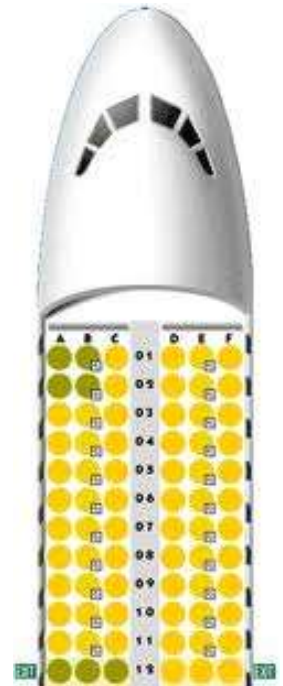
- En aplicaciones bancarias o de reservaciones en líneas áreas, cientos de operaciones se efectúan cada segundo en la BD.
- Éstas comienza en una gran variedad de sitios: **cajeros automáticos, ventanillas, computadoras de los agentes de viajes, de las compañías áreas e incluso de los mismos clientes.**



- Es posible que se realicen **simultáneamente** dos operaciones que afecten a la misma cuenta o al mismo vuelo y que además coincidan aproximadamente en el tiempo.
- Si esto ocurriera, podrían interactuar de forma extrañas; es decir, dos operaciones puede efectuarse correctamente y pese a ello, su resultado no sería correcto.

```
select ocupado
from Vuelos
where numVuelo = 200 and fecha = '2017-12-04'
      and asiento = 'A03';
```

```
update vuelos
set ocupado = true
where numVuelo = 200 and fecha = '2017-12-04'
      and asiento = 'A03';
```

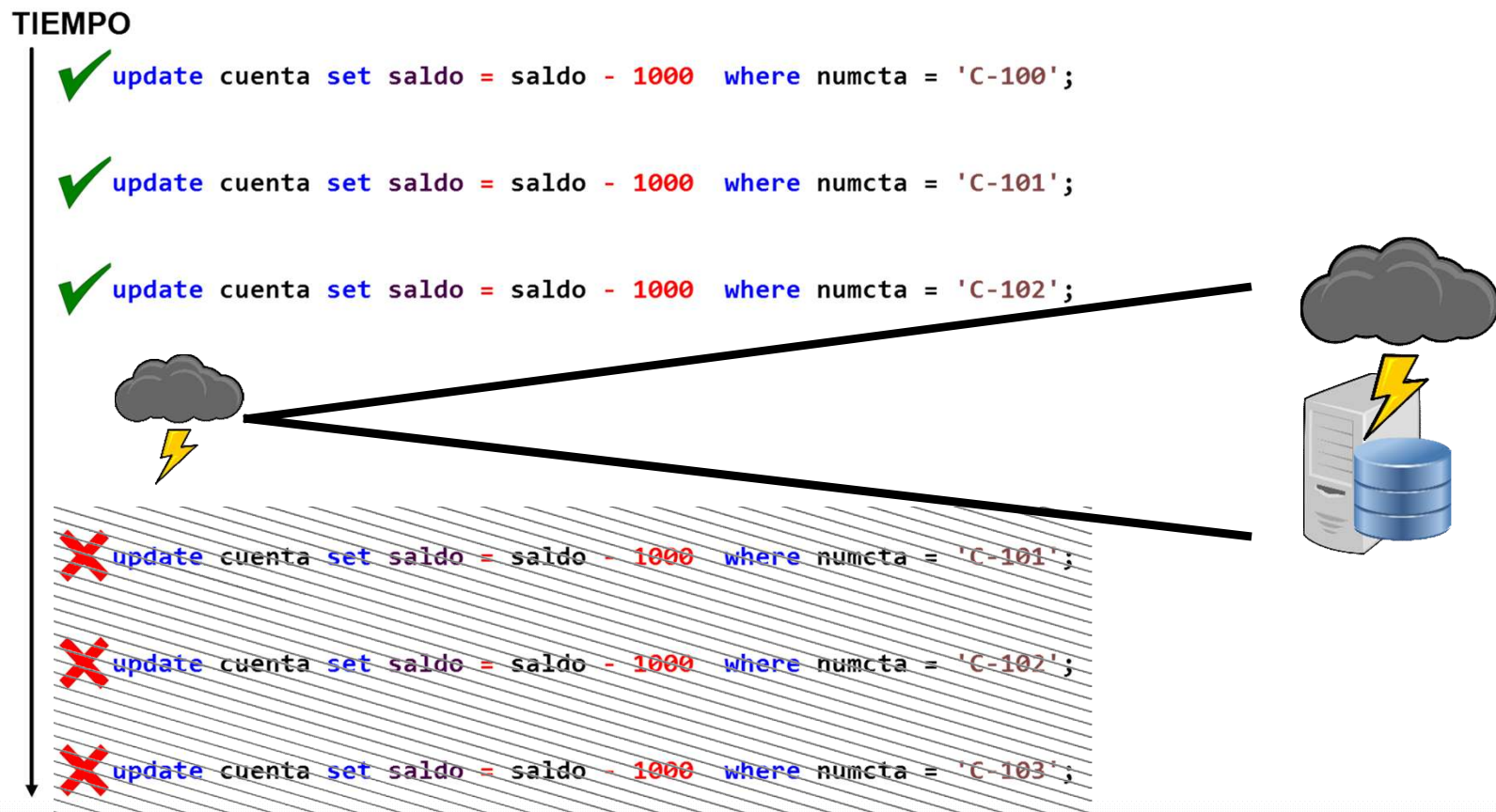


- El problema puede resolverse a través de varios mecanismos de SQL que permiten **serializar** la realización de dos ejecuciones.
- Decimos que la ejecución de dos funciones que operan sobre la misma BD es **serial**, si una se ejecuta completamente antes de que empiece la otra.
- Decimos que la ejecución es **serializable** si se comporta como si se ejecutara de forma serial, aunque sus ejecuciones puedan coincidir en el tiempo.

Tiempo	Transacciones		Valores	
	T1	T2	A	B
t1			25	25
t2	READ(A); A := A + 100; WRITE(A)		125	
t3		READ(A); A := A * 2; WRITE(A);	250	
t4	READ(B); B := B + 100; WRITE(B)			125
t5		READ(B); B := B * 2; WRITE(B);		250

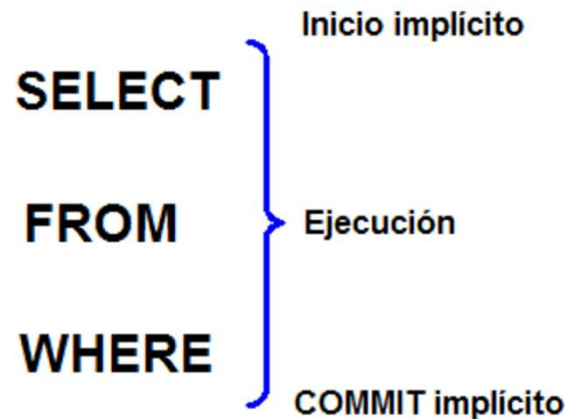
- Supongamos que tenemos una aplicación bancaria y un procedimiento para transferir fondo entre las cuentas **C1** y **C2**:
 1. **Se verifica que C1 tenga dinero suficiente.**
 2. **Se aumenta el saldo de C2 en el monto especificado**
 3. **Se disminuye el saldo de C1 en el monto especificado.**
- Supongamos que el sistema falla justo antes de comenzar a ejecutar el **paso 3**.
- La base de datos quedaría entonces en un estado **indeseable** (al menos para el banco).
- En este ejemplo, nos gustaría que las operaciones se ***ejecutaran todas o que ninguna de ellas se ejecutara.***

- La ejecución de una operación es **atómica** si el estado de la base de datos luego de la operación es como si todos sus componentes se hubiesen ejecutado o como si ninguno de ellos lo hubiera hecho.



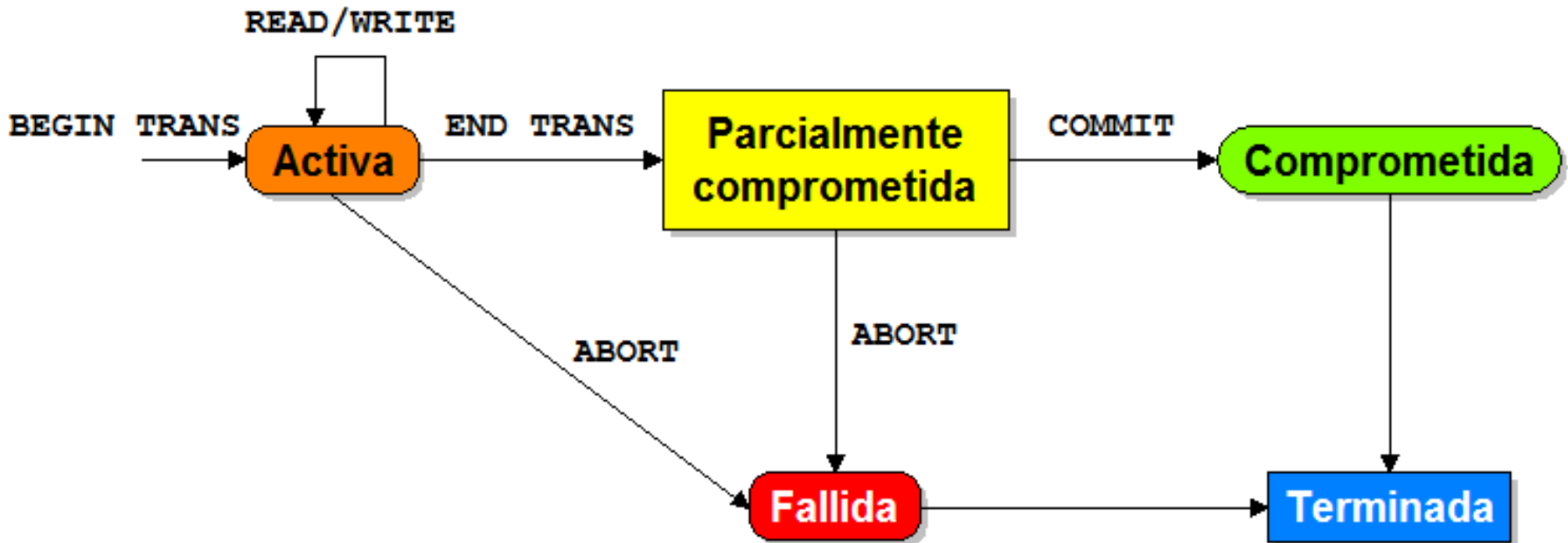
- **Atomicidad** (**A**tomicity). Se realizan todas las operaciones incluidas en la transacción o no se realiza ninguna.
- **Consistencia** (**C**onsistencia). Una base de datos está en estado consistente si los datos cumplen las expectativas.
- **Aislamiento** (**I**solation). Los efectos causados por la ejecución de transacciones simultáneas no debe ser distinta que si se realizaran en forma secuencial.
- **Persistencia** (**D**urability). El efecto de la transacción no se debe perder aunque el sistema falle, aún si la falla ocurre en cuanto termina la transacción.

- SQL **no tiene** una instrucción de inicio de transacción. Éstas empiezan con **cualquier instrucción** de SQL.
- Para terminar la transacción hay dos posibilidades:
 - ✓ **COMMIT** para especificar que una transacción llegó a buen fin.
 - ✗ **ROLLBACK** para especificar un fin anormal o con fracaso.
- SQL maneja todas sus instrucciones como una transacción. Toda instrucción SQL empieza con un ***begin transaction*** implícito:



Si algo falla, entonces hace **ROLLBACK**.

Estados de una transacción



...Estados de una transacción

- Si la transacción se compromete, las modificaciones hechas a los datos permanecen en la BD incluso si hay fallas en el sistema.
- Una transacción abortada no debe tener efecto en la BD. El aborto puede ser:
 - ✓ **Voluntario.** Se detecta un error fatal. No hay fondos suficientes.
 - ✓ **Involuntario.** Falta de energía eléctrica.
- Antes que inicie la transacción, la **BD está en estado consistente**, al terminar la ejecución de la misma se debe tener **otro estado consistente**.

- La ejecución concurrente de transacciones permite que los usuarios de la BD accedan simultáneamente a ella sin colisiones entre ellos.
- La propiedad de aislamiento se aplica cuando se tiene procesamiento concurrente de transacciones.
- Lo más simple sería permitir solo ejecución secuencial, pero es mejor concurrente porque:
 - ✓ **Se aprovecha el procesador** en tanto se hacen los accesos a disco.
 - ✓ **Las transacciones no son del mismo tamaño:** Si una transacción corta debiese esperar hasta que termine una larga, bajaría el rendimiento.
 - ✓ Si se está operando sobre datos distintos en la BD, lo mejor es que **se ejecuten concurrentemente.**

- Principio general:

Una transacción que se está ejecutando debe ser capaz de terminar su ejecución sin interferencia de otras transacciones.

- La ejecución concurrente de transacciones puede ocasionar inconsistencia en los datos.
- Para evitar destruir la consistencia de la BD el SMBD hace una planeación de las transacciones aplicando esquemas para el **control de la concurrencia**.
- Una planeación (**schedule**) de la ejecución es una secuencia ordenada de acciones de varias transacciones:

T1.A, T2.A, T1.B, T1.C, T2.B, T1.B

Por ejemplo, tenemos dos transacciones **T1** y **T2** para transferir dinero entre cuentas:

1. **T1** transfiere **\$10,000** de una cuenta **A** a otra cuenta **B**:

T1

```
READ(A);  
A := A - 10000;  
WRITE(A);  
READ(B);  
B := B + 10000  
WRITE(B);
```

2. **T2** transfiere el **50%** de la cuenta **A** a la cuenta **B**:

T2

```
READ(A);  
desc := A * 0.5;  
A := A - desc;  
WRITE(A);  
READ(B);  
B := B + desc  
WRITE(B);
```

Planeación 1:

Tiempo	Transacciones		Valores	
	T1	T2	A	B
t1			\$20,000	\$10,000
t2	READ(A); A := A - 10000; WRITE(A); READ(B); B := B + 10000; WRITE(B);		\$10,000	\$20,000
t3		READ(A); desc := A * 0.5; A := A - desc; WRITE(A); READ(B); B := B + desc; WRITE(B);	\$5,000	\$25,000

Planeación 2:

Tiempo	Transacciones		Valores	
	T1	T2	A	B
t1			\$20,000	\$10,000
t2	READ(A); A := A - 10000; WRITE(A);		\$10,000	
t3		READ(A); desc := A * 0.5; A := A - desc; WRITE(A);	\$5,000	
t4	READ(B); B := B + 10000 WRITE(B);			\$20,000
t5		READ(B); B := B + desc WRITE(B);		\$25,000

...Procesamiento concurrente

Ahora, qué tal si la ejecución se realizara en el siguiente orden:

Tiempo	Transacciones		Valores	
	T1	T2	A	B
t1			\$20,000	\$10,000
t2	READ(A); A := A - 10000;			
t3		READ(A); desc := A * 0.5; A := A - desc; WRITE(A);	\$10,000	
t4	WRITE(A); READ(B); B := B + 10000 WRITE(B);			
t5		READ(B); B := B + desc; WRITE(B);		\$30,000

No todas las transacciones concurrentes llevan a estados consistentes.

- El SABD tiene un módulo llamado **control de concurrencia** para asegurar la consistencia de la BD.

- Hasta ahora se han visto transacciones que no consideran la posibilidad de fallas:
 - ✓ Si **Tx** falla por cualquier razón, es necesario **deshacer su efecto** para asegurar la **atomicidad** de la misma.
 - ✓ En el caso de que se tenga **concurrency** se debe deshacer toda transacción **Ti** que dependa de **Tx** (es decir, toda **Ti** que lea datos escritos en **Tx**).
- Para lograr esto es necesario **poner restricciones** al tipo de planeaciones permitidas en el sistema.
- Vemos el siguiente ejemplo...

Planeaciones recuperables

Tiempo	Transacciones		Valores	
	T1	T2	A	B
t1			\$20,000	\$10,000
t2	READ(A); A := A - 5000 WRITE(A);		\$15,000	
t3		READ(A);	\$15,000	
t4	READ(A);		\$20,000	

¿Qué pasa si se acepta **T2** después de su lectura y **T1** falla?

Una **planeación recuperable** es aquella en la que todo par de transacciones **T_i** y **T_j**, tales que **T_j** lee datos que **T_i** ha escrito, la **operación de comprometer** de **T_i** debe aparecer antes que la de **T_j**.

- Incluso si una planeación es recuperable, hay que retroceder varias transacciones para recuperar el estado previo a una falla en una transacción **T_i**.

...Planeaciones recuperables

Tiempo	Transacciones			Valores	
	T1	T2	T3	A	B
T1				\$20,000	\$10,000
T2	READ(A); A := A - 5000 READ(B); WRITE(A);			\$15,000	
t3		READ(A); READ(B); B := B + A; WRITE(B);			\$15,000
t4			READ(A); READ(B);	\$15,000	\$15,000

Qué pasaría si fallara **T3** justo cuando se está haciendo la lectura de **A** y **B**?

- Este fenómeno se conoce como **retroceso en cascada**.
- Se trata de un fenómeno indeseable porque aumenta significativamente el trabajo necesario para deshacer cálculos.

- Una forma de asegurar la secuencialidad es exigir que el acceso a los datos sea en exclusión mutua.
- De esta forma se asegura que mientras una transacción accede a un dato ninguna otra puede modificar tal dato.
- El mecanismo más simple consiste en asignar **candados**:



1. La idea es que cada dato tenga un **candado asociado**.
2. Antes que una transacción pueda acceder a un dato, el **planeador** primero examina su candado.
3. Si **ninguna transacción lo tiene** se le asigna a ella, en caso contrario, si **T1** solicitó un candado que ya tiene asignado **T2**, entonces **T1 tiene que esperar** hasta que **T2** lo libere.

...Control de concurrencia

- El candado se solicita en función del tipo de operación que se vaya a realizar y esperar hasta que éste sea asignado para realizar la operación.
- Existen dos tipos de candados:

- ✓ **Compartido.** Si la transacción **T_n** obtiene un candado **compartido (C)** sobre un elemento **D**, entonces **T_n solo puede leer** el dato D.
- ✓ **Exclusivo.** Si la transacción **T_n** obtiene un candado en modo **exclusivo (X)** sobre D, entonces **puede leer y/o escribir** el dato.

	C	X
C	true	false
X	false	true

...Control de concurrencia

- Se tiene una transacción **T1** que transfiere **\$10,000** de la cuenta **A** a la cuenta **B** (**A.saldo = \$20,000** y **B.saldo = \$10,000**)

T1

```
BLOQUEAR-X(A);
READ(A);
A := A - 10000;
WRITE(A);
DESBLOQUEAR(A);
BLOQUEAR-X(B)
READ(B);
B := B + 10000
WRITE(B);
DESBLOQUEAR(B);
```

- T2** es una transacción que lee el valor de **A** y el de **B** y muestra su suma.

T2

```
BLOQUEAR-C(B);
READ(B);
DESBLOQUEAR(B);
BLOQUEAR-C(A);
READ(A);
DESBLOQUEAR(A);
MOSTRAR(A + B);
```

- Si se tuviera la siguiente planeación:

Tiempo	Transacciones			Valores	
	T1	T2	T3	A	B
T1				\$20,000	\$10,000
T2	BLOQUEAR-X(A); READ(A); A := A - 10000; WRITE(A); DESBLOQUEAR(A);				
t3		BLOQUEAR-C(B); READ(B); DESBLOQUEAR(B); BLOQUEAR-C(A); READ(A); DESBLOQUEAR(A); MOSTRAR(A + B);			
t4	BLOQUEAR-X(B) READ(B); B := B + 10000 WRITE(B); DESBLOQUEAR(B);				\$20,000

Protocolo de bloqueo de dos fases

- Un bloqueo es un mecanismo que permite controlar el acceso concurrente a los datos.
- Las solicitudes de bloqueo se envían al **control de concurrencia**. La transacción puede realizar la operación sólo después de que se conceda la solicitud.
- A una transacción se le puede garantizar un bloqueo en un elemento si el bloqueo solicitado es compatible con los bloqueos que ya tengan otras transacciones sobre ese mismo elemento

	C	X
C	true	false
X	false	true

- Si no se puede garantizar un bloqueo, la transacción que lo solicita tiene que esperar hasta que los bloqueos incompatibles que tienen otras transacciones se hayan liberado. A continuación se autoriza el bloqueo.

...Protocolo de bloqueo de dos fases

T1

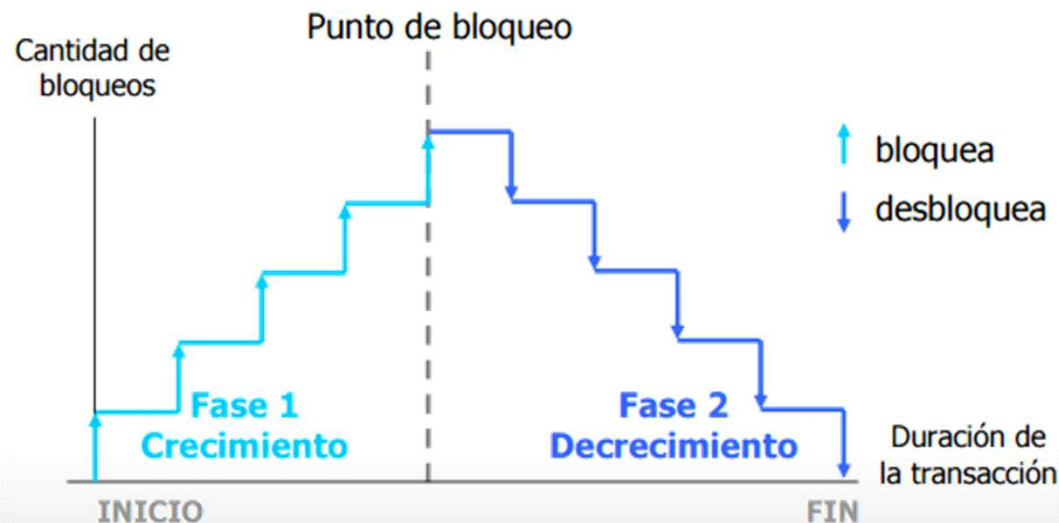
```

BLOQUEAR-C(A);
READ(A);
DESBLOQUEAR(A);
BLOQUEAR-C(B);
READ(B);
DESBLOQUEAR(B);
MOSTRAR(A+B)
    
```

- El bloqueo anterior no es suficiente para garantizar la secuencialidad, ***¿qué pasaría si se actualizaran A y B entre la lectura de A y B?***
- Un **protocolo de bloqueo** es un **conjunto de reglas** que siguen todas las transacciones cuando se solicitan o se liberan bloqueos.
- Los protocolos de bloqueo restringen el número de planificaciones posibles.

...Protocolo de bloqueo de dos fases

- Se utiliza para asegurar la secuencialidad de las transacciones.
- Consiste en bloquear y desbloquear en dos fases:
 - Fase de crecimiento.** Una transacción puede obtener candados pero no puede liberarlos. Inicialmente toda transacción está en ese estado.
 - Fase de decrecimiento.** Una transacción puede liberar candados pero no puede obtener nuevos. Una vez que la transacción libera un candado, entra en esta fase y no puede realizar más peticiones de candados.



...Protocolo de bloqueo de dos fases

El protocolo tiene dos variantes:

- **Protocolo de bloqueo riguroso de dos fases:**

Exige que se posean todos los candados hasta que se comprometa la transacción. Muchos SABD comerciales usan este protocolo.

- **Protocolo de bloqueo estricto de dos fases:**

Una transacción debe conservar todos los bloqueos exclusivos hasta que se comprometa la transacción.

Fallas en los protocolos de bloqueo

- En la mayoría de los protocolos de bloqueo se puede producir un **interbloqueo potencial**, por ejemplo, considere la siguiente transacción:

T1	T2
BLOQUEAR-X(B);	
READ(B);	
B := B - 50	
WRITE(B)	
	BLOQUEAR-C(A);
	READ(A);
	BLOQUEAR-C(B);
BLOQUEAR-X(A)	

- Ni **T1** ni **T2** pueden avanzar:

La ejecución de **BLOQUEAR-C(B)** ocasiona que **T2** espere a que **T1** libere su bloqueo sobre **B**, mientras que la ejecución de **BLOQUEAR-X(A)** ocasiona que **T1** espera a que **T2** libere su bloqueo sobre **A**. A esta situación se denomina **interbloqueo** (Para manejar un interbloqueo uno de los dos, *T1* o *T2*, debe retroceder y liberar sus bloqueos).

...Fallas en los protocolos de bloqueo

- ✓ Consistencia
- ✗ Interbloqueos
- ✗ Inanición

	C	X
C	✓	✗
X	✗	✗

- En la mayoría de los protocolos de bloqueo se puede producir un **interbloqueo potencial**.
- También es posible que se produzca la **inanición** si el **control de concurrencia** se ha diseñado defectuosamente, por ejemplo:

Una transacción está esperando un bloqueo exclusivo sobre un elemento determinado mientras que una secuencia de transacciones diferentes solicitan y obtienen la autorización de bloqueo compartido sobre el mismo elemento. La misma transacción retrocede repetidamente debido a los interbloqueos.

- El **control de concurrencia** se puede diseñar para impedir que se produzca la **inanición**.

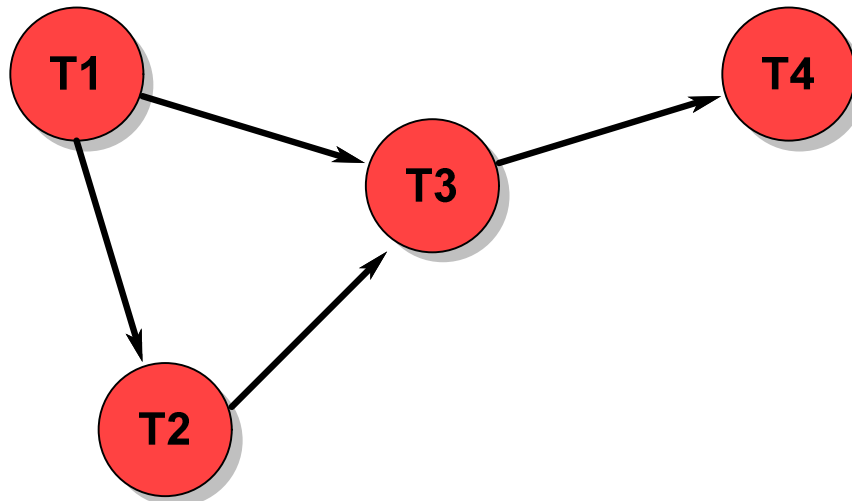
Interbloqueos (deadlocks)

- Es necesaria una estrategia para detectar deadlocks a fin de que ninguna transacción esté bloqueada eternamente.
- Una estrategia sencilla se basa en asignar límite en el tiempo de espera (**timeouts**): *una transacción que solicita un candado debe esperar hasta el tiempo límite para que se lo concedan y si no es así, debe abortar.*
- Difícil determinar un buen límite:
 - ✓ **Pequeño:** abortar transacciones que no estén en deadlock o que se haya llegado al límite y la transacción que tiene el dato no lo ha soltado.
 - ✓ **Grande:** se debe esperar mucho tiempo a que se aborten transacciones que están bloqueadas.

...Interbloqueos (deadlocks)

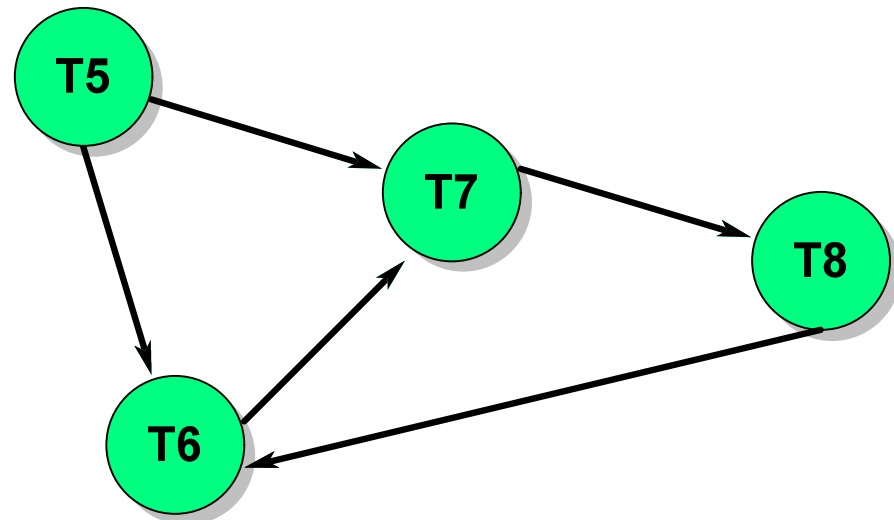
- Si no se puede asegurar la ausencia de deadlock, es necesario detectarlos:
 1. **Mantener información sobre la asignación de datos a las transacciones.**
 2. **Mantener información sobre las peticiones pendientes.**
 3. **Proporcionar un algoritmo que determine si existe un deadlock**
 4. **Recuperarse de ese deadlock.**
- Un algoritmo para detección de deadlocks consiste en crear una **gráfica (N,E)** de espera, donde **N** es el conjunto de transacciones en ejecución concurrente y **E** es el conjunto de enlaces **T_i → T_j** que indican que **T_i** está esperando a que **T_j** libere un candado que necesita.
- Hay **deadlock** si hay ciclos en la gráfica.

...Interbloqueos (deadlocks)



Sin interbloqueo

**Con interbloqueo
(existen ciclos)**



...Interbloqueos (deadlocks)

- Al detectarlo se debe abortar una transacción y eliminar su nodo asociado.
- Para elegir a la víctima se busca la que signifique un **costo mínimo**:

- ✓ Cuánto ha calculado la transacción y lo que le falta.
- ✓ Cuántos datos ha usado.
- ✓ Cuántos datos necesita.
- ✓ Cuántas transacciones se verán involucradas en el aborto.

Sean **T1** y **T2** dos transacciones que actualizan el saldo de una cuenta. El saldo inicial es de **\$1,500.00**:

T1.A:

```
saldo1 := (select saldo from cliente where cuenta = 'C-101');  
saldo1 := saldo1 + 500;
```

T1.B:

```
update cliente set saldo = :saldo1 where cuenta = 'C-101';
```

T2.A:

```
saldo2 := (select saldo from cliente where cuenta = 'C-101');  
saldo2 := saldo2 + 1000;
```

T2.B:

```
update cliente set saldo = :saldo2 where cuenta = 'C-101';
```

- Si la secuencia fuera:

T1.A, T2.A, T1.B, T2.B, T1.COMMIT, T2.COMMIT;

T1.A:

```
saldo1 := (select saldo from cliente where cuenta = 'C-101');  
saldo1 := saldo1 + 500;
```

T2.A:

```
saldo2 := (select saldo from cliente where cuenta = 'C-101');  
saldo2 := saldo2 + 1000;
```

T1.B:

```
update cliente set saldo = :saldo1 where cuenta = 'C-101';
```

T2.B:

```
update cliente set saldo = :saldo2 where cuenta = 'C-101';
```

Saldo de la cuenta **C-101** = **\$2,500**

- Con la secuencia:

T1.A, T1.B, T2.A, T1.ROLLBACK, T2.B, T2.COMMIT

T1.A:

```
saldo1 := (select saldo from cliente where cuenta = 'C-101');
saldo1 := saldo1 + 500;
```

T1.B:

```
update cliente set saldo = :saldo1 where cuenta = 'C-101';
```

T2.A:

```
saldo2 := (select saldo from cliente where cuenta = 'C-101');
saldo2 := saldo2 + 1000;
```

T2.B:

```
update cliente set saldo = :saldo2 where cuenta = 'C-101';
```

Saldo de la cuenta **C-101** = **\$3,000**

Problema de suma incorrecta

- Los saldos iniciales son **C-101 = \$2,000** y **C-102 = \$1,500**

T1.A:

```
saldo1 := (select saldo from cliente where cuenta = 'C-101');
saldo1 := saldo1 + 1000;
```

T1.B:

```
update cliente set saldo = :saldo1 where cuenta = 'C-101';
```

T1.C:

```
saldo1 := (select saldo from cliente where cuenta = 'C-102');
saldo1 := saldo1 - 1000;
```

T1.D:

```
update cliente set saldo = :saldo1 where cuenta = 'C-102';
```

T2.A:

```
total := (select sum(saldo) from cliente
          where cuenta = 'C-101' or cuenta = 'C-102');
```

... Problema de suma incorrecta

- Con la planeación:

T1.A, T1.B, T2.A, T2.COMMIT, T1.C, T1.D, T1.COMMIT

T1.A:

```
saldo1 := (select saldo from cliente where cuenta = 'C-101');
```

```
saldo1 := saldo1 + 1000;
```

T1.B:

```
update cliente set saldo = :saldo1 where cuenta = 'C-101';
```

T2.A:

```
total := (select sum(saldo) from cliente
          where cuenta = 'C-101' or cuenta = 'C-102');
```

T1.C:

```
saldo1 := (select saldo from cliente where cuenta = 'C-102');
```

```
saldo1 := saldo1 - 1000;
```

T1.D:

```
update cliente set saldo = :saldo1 where cuenta = 'C-102');
```

El valor de la suma es = **\$4,500**

Problema de lectura irrepetible

- Con la planeación:

T1.A, T2.A, T2.COMMIT, T1.B, T1.COMMIT

T1.A:

```
saldo1 := (select saldo from cliente where cuenta = 'C-101');
```

T2.A:

```
update cliente set saldo = 0.0 where cuenta = 'C-101';
```

T1.B:

```
saldo2 := (select saldo from cliente where cuenta = 'C-101');
```

El saldo de **C-101** = \$ 0.0

Problema de tuplas fantasma

- Esto ocurre cuando una operación de agregación se ejecuta en una transacción y da diferente resultado debido a la inserción de tuplas por otras transacciones.
- Con la planeación:

T1.A, T2.A, T1.B, T2.ROLLBACK, T1.COMMIT

T1.A:

```
totalA := (select sum(saldo) from cliente where cp = 14050);
```

T2.A:

```
insert into cliente values('C-105',10000,14050);
```

T1.B:

```
totalB := (select sum(saldo) from cliente where cp = 14050);
```

Transacciones de solo lectura

Al tener transacciones que no modifican la BD, es más fácil su ejecución en paralelo.

Ejemplo. Generación de una nómina

```
set transaction read only;
```

Debe incluirse antes de que empiece la transacción.

```
set transaction...
```

- Se usa para definir características de una transacción.
- No es inicio de transacción.

Sirven para determinar lo que una transacción tiene permitido ver.

1. **Total o serializable**. Este es por omisión.

2. **Lecturas sucias**

Cuando la transacción **T1** ejecuta una actualización sobre una tupla; **T2** trabaja con esa tupla y **T1** aborta.

Un ejemplo de lectura sucia

- Se busca un lugar disponible
Si lo hay se reserva, si no abortar
- Se pregunta al cliente si desea asiento.
Si acepta, terminar. Si no, liberar el asiento.

En este caso puede tener sentido permitir **lectura sucia**

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
```

3. Lecturas no repetibles.

Si **T1** recupera una tupla, luego **T2** la actualiza, luego **T1** recupera «**la misma**» tupla.

Si en el ejemplo anterior se tuviera:

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

4. Lecturas repetibles

Si **T1** recupera un conjunto de tuplas que satisfacen una condición, **T2** inserta una nueva tupla que satisface la condición. Luego **T1** vuelve a leer, aparecen «**nuevas tuplas**».

Se garantiza que al leer una tupla, las siguientes veces se tendrá, pero también se pueden recuperar tuplas fantasma (Inserciones a la BD mientras la transacción se está ejecutando).

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

Al ejecutar la transacción se verían tuplas desocupadas, incluyendo algunas recién liberadas (por ejemplo, por cancelaciones).

Nivel	Lectura sucia	No repetible	Tuplas fantasma
Serializable	NO	NO	NO
Repeatable read	NO	NO	SI
Read committed	NO	SI	SI
Read uncommitted	SI	SI	SI