Dpto. Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur

Elementos de Bases de Datos

Segundo Cuatrimestre 2015

Clase 13:

Gestión de Transacciones
Control de Concurrencia

Mg. María Mercedes Vitturini
[mvitturi@uns.edu.ar]

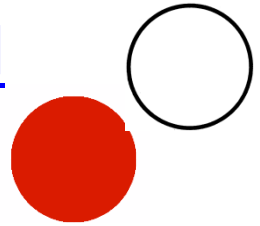
Transacciones

Transacciones

- Definición
- Características
- Read (X) / Write (X)
- Input (B_X)/Output (B_X)
- Estados

Propiedades

- Atomicidad



- Consistencia
- Aislamiento

- Durabilidad

Transacción

Definición: se denomina *transacción* a una colección de operaciones sobre la base de datos que conforman una unidad lógica de trabajo y que posiblemente acceden y actualizan varios ítems de datos

Ejemplos

- La *transferencia* de una cuenta a otra, consiste en la actualización de ambas cuentas.
- La *venta de un pasaje* que incluye las tareas de generar un ticket y ocupar un asiento.
- El *intercambio de comisiones de cursados* entre dos alumnos.

Ejemplo

Supongamos la siguiente transacción: *"Se desea realizar una transferencia de 50$ de la cuenta A a la cuenta B"*

```
T: 1. Read(A);
   2. A:=A-50;
   3. Write(A);
   4. Read(B);
   5. B:=B+50;
   6. Write(B)
```

Donde:

read(x) – transfiere el ítem de dato X de la base de datos al espacio de datos local a la transacción que ejecuta el read.

write(x) – transfiere el ítem de dato X desde el espacio de datos local a la transacción que ejecuta el write a base de datos

Principales Operaciones

Read(X): transfiere el ítem de dato X desde el buffer de datos compartido a al área local de la transacción.

Write(X): que transfiere el ítem de dato X desde el área local de la transacción que ejecutó el write al buffer de datos compartido.

Area Local

Buffer de Datos

Input (B): transfiere bloque de datos B desde almacenamiento secundario al buffer de datos compartido.

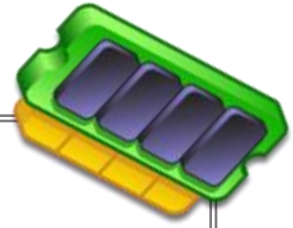
Output (B): transfiere bloque de datos B desde el buffer de datos compartido al almacenamiento secundario.

Buffer de Datos

Almacen. Secundario

Cada operación WRITE *no necesariamente* resulta en una escritura inmediata del ítem de dato al disco (OUTPUT).

Espacios de Datos



Almacenamien
to secundario

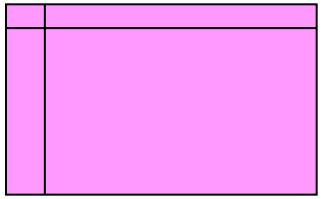
Memoria RAM

Buffer de
Datos
Compartido

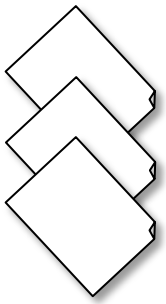
Input/
Ouput

read/
write

Buffer local
transacción T_1



Buffer local
transacción T_2



Unidad de
comunicación
“bloque”

Unidad de
comunicación
“dato”

Transacciones - Características

Características de una transacción:

- Eventualmente accede y/o actualiza varios ítems de datos.
- Accede y deja a la BD en un estado consistente.
- El programador es quien define los límites de la transacción.
- Múltiples transacciones se ejecutan en "paralelo".
- El DBMS debe asegurar la ejecución adecuada de cada transacción, a pesar de la existencia de fallos:
  - Fallas de hardware, caídas del sistema, etc.
  - Ejecución concurrente de varias transacciones.

Transacciones – Propiedades

El DBMS debe asegurar las siguientes propiedades que preservan integridad en la base de datos:

Atomicidad

- Las operaciones de una transacción se ejecutan *todas* o *ninguna*.

Consistencia

- La ejecución de una transacción preserva la consistencia de la BD.

Durabilidad

- Después de que una transacción se ejecutó con éxito, los cambios en la BD persisten, más allá de las fallas del sistema.

Transacciones – Propiedades

Aislamiento

- Aunque varias transacciones se ejecuten de manera concurrente, el resultado debe ser equivalente a alguna secuencia de ejecuciones en serie.
  - Esto es, dadas T_i y T_j, concurrentes.
  - Para T_i el resultado debe ser equivalente a T_j, ejecutándose antes que T_i se inicie, o T_j iniciando su ejecución después que T_i finalizó.

Propiedades ACAD (ACID)

ATOMICIDAD (*atomicity*), CONSISTENCIA (*consistency*), AISLACIÓN (*isolation*) y DURABILIDAD (*durability*).

Propiedades: Atomicidad

T transfiere $ 50 de la cuenta *A* a la cuenta *B*.

```
T: 1.Read(A);
   2.A:=A-50;
   3.Write(A);
   4.Read(B);
   ---------->Falla !!
   5.B:=B+50;
   6.Write(B);
```

- Supongamos que, antes de la ejecución de T, *A* tenía $1000 y *B* tenía $2000.
- Es de esperar que luego de T, *A* tenga $950 y *B* tenga $2050.
- Sin embargo, por fallos, por ejemplo en el sistema, T podría abortarse antes de actualizar *B* y el resultado sería: *A* con $950 y *B* con $2000.
  - ¡Perdimos $50 pesos sin hacer nada!
  - Estado Inconsistente.

Propiedades: Consistencia

T: transfiere $ 50 de la cuenta *A* a la cuenta *B*.

```
T:  1.Read(A);
    2.A:=A-50;
    3.Write(A);
    4.Read(B);
    5.B:=B+50;
    6.Write(B);
```

- El requerimiento de consistencia aquí es que la suma de *A* y *B* se preserve después de la ejecución de T.
- Sin el requerimiento de consistencia, se podría generar o perder dinero mediante la transacción.
- Asegurar consistencia para una transacción individual es plena responsabilidad del programador.

Propiedades: Durabilidad

- Una vez que una transacción se terminó exitosamente (ie, se confirmó), *los cambios realizados en la base de datos persisten*, aunque se produzcan luego fallas en el sistema.
  - Para garantizar durabilidad se debe asegurar que:
    - Las modificaciones se escriben en disco antes de que la transacción finalice, ó
    - La información sobre los cambios efectuados por una transacción es suficiente para repetirla en caso de fallo de sistema.
- La responsabilidad de durabilidad es del **Gestor de Recuperación.**

Propiedades: Aislación

T1 transfiere $ 50 de la cuenta *A* a la cuenta *B*.
T2 consulta e imprime los saldos

```
T1:1.Read(A);
   2.A:=A-50;
   3.Write(A);
            T2:4.Read (A);
               5.Read (B);
               6.Print(A,B,A+B);
    7.Read(B);
    8.B:=B+50;
    9.Write(B);
```

- Si varias transacciones se ejecutan concurrentemente, sus operaciones pueden llegar a entrelazarse de manera incorrecta.
- En el ejemplo, T_2 lee los datos *A* y *B* después de que T_1 modificó *A* pero antes de que modifique *B* y así accede a datos incorrectos. Este tema será ampliamente abordado cuando estudiemos *serializabilidad*.

Estados de una Transacción

Activa – el estado inicial. La transacción permanece en ese estado mientras se está ejecutando.

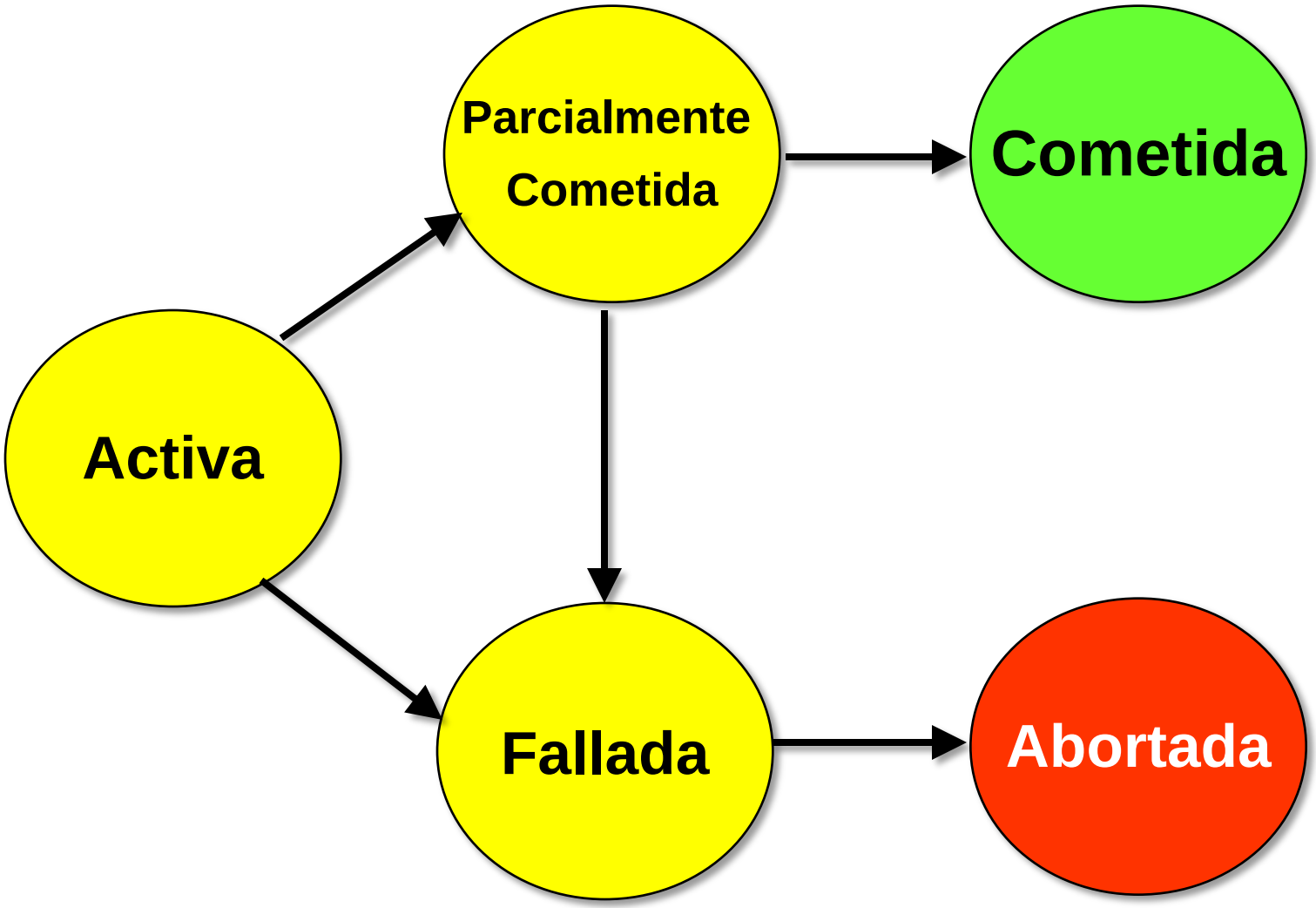
Parcialmente Cometida – después de que se ejecutó la última instrucción de la transacción.

Fallada – después de descubrir que la ejecución normal no puede continuar.

Abortada – después de que la transacción "retrocedió" **(rolled back)** y la base de datos fue restablecida al estado anterior al comienzo de la transacción.

Cometida – después de una finalización exitosa.

Estados de una Transacción



Transacciones Abortadas

Después de que una transacción es abortada es posible:

- **Reiniciar la Transacción**: solamente si la transacción fue abortada como consecuencia de alguna *falla de hardware* o algún *error de software externo* (esto es, que no fue creado mediante la lógica interna de la transacción). Una transacción reiniciada se considera una nueva transacción.
- **Matar la Transacción:** generalmente cuando ocurre un *error lógico interno* que debe ser corregido reescribiendo el programa de aplicación, o cuando la entrada fue incorrecta, o cuando los datos deseados no fueron encontrados.

Control de Concurrencia

Técnicas para evaluar y garantizar *aislamiento*

Concurrencia

- Ejecuciones concurrentes y Aislamiento
- Planificaciones
  - Planificación en Serie.
  - Planificación concurrente.
  - Planificación concurrente serializable.
- Test de Serializabilidad
  - Instrucciones conflictivas.
  - Test de Serializabilidad en Conflictos
  - Grafo de Control Conflictos

Ejecuciones Concurrentes

- En entornos multiprogramados, es posible ejecutar varias transacciones en forma concurrente.

Entre los beneficios del procesamiento concurrente:

- Mejora la utilización de disco y procesador, aumentando la productividad (*troughput*).
- Mejora los tiempos de respuesta, transacciones cortas no necesitan esperar por transacciones largas.

Control de Concurrencia

- Cuando se están ejecutando varias transacciones de manera concurrente, se puede *perder la consistencia de la base de datos* aún cuando cada una de las transacciones individuales sea correcta.
- Los **esquemas de control de concurrencia** son los mecanismos para lograr **aislamiento**.
- Asumimos que cada transacción utiliza las operaciones read(X) / write(X), donde X es el dato a leer/escribir.

Definición

Planificación – es la secuencia cronológica en el cual se ejecutan en el sistema las instrucciones de transacciones concurrentes:

$$P_k = I_{i,j}^{*} \quad (\text{con } I_{i,j} \text{ instrucción j-ésima de } T_i)$$

- La planificación de un conjunto de transacciones debe considerar todas las instrucciones de las mismas.
- Toda planificación debe respetar el orden en el que se presentan las instrucciones de cada transacción individual.

Planificaciones

- Decimos que una planificación es una **planificación en serie** si las instrucciones pertenecientes a cada una de las transacciones aparecen todas juntas (consecutivas).
  - Dadas *n* transacciones existen *n!* planificaciones en serie.
- Una planificación *concurrente* **es serializable** cuando el resultado de su ejecución equivale a alguna ejecución en serie.

Ejemplo de transacciones

```
T0: 1.Read(A);
    2.A:= A-50;
    3.Write(A);
    4.Read(B);
    5.B := B+50;
    6.Write(B).
```

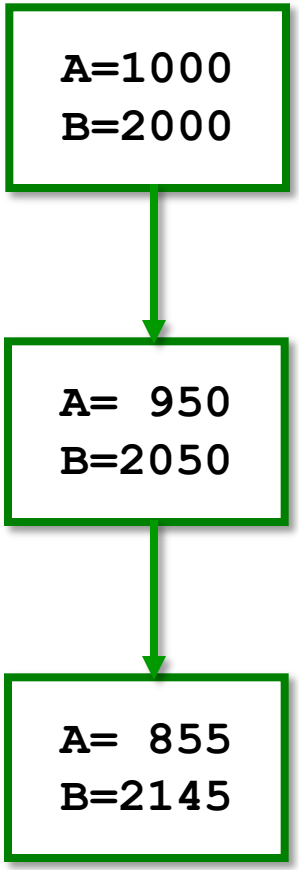
Transfiere 50 pesos de la cuenta A a la cuenta B.

```
T1: 1.Read(A);
    2.Temp := A*0.1;
    3.A:= A-Temp;
    4.Write(A);
    5.Read(B);
    6.B := B+Temp;
    7.Write(B).
```

Transfiere el 10% de la cuenta A a la cuenta B.

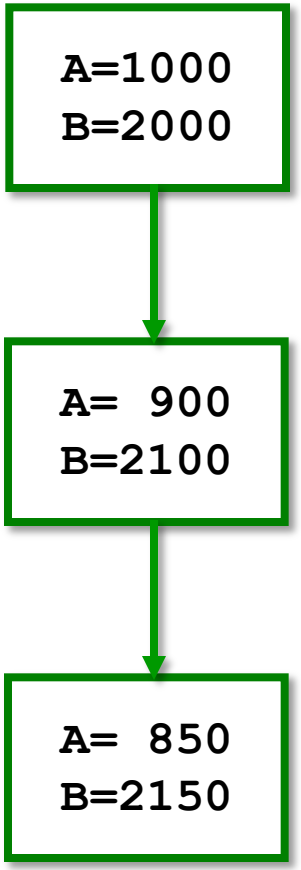
Planificación #1 en serie: “T0 ; T1”

T_0	T_1
Read(A); A:= A-50; Write(A); Read(B); B := B+50; Write(B).	Read(A); Temp := A*0.1; A := A-Temp; Write(A); Read(B); B := B+Temp; Write(B).



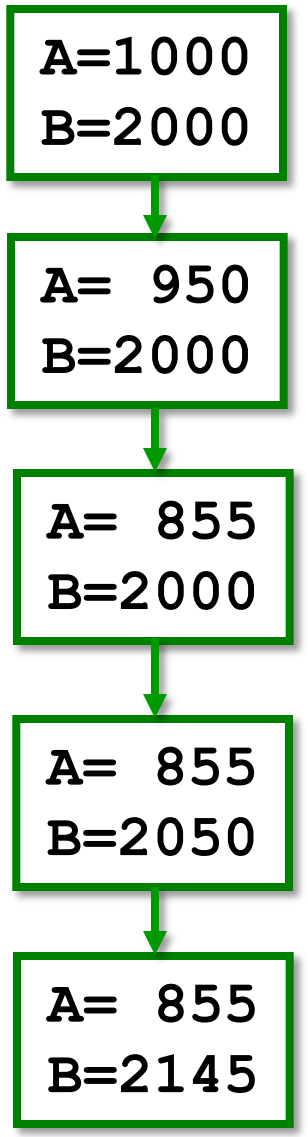
Planificación #2 en serie: “T1; T0”

T_0	T_1
Read(A); A:= A-50; Write(A); Read(B); B := B+50; Write(B).	Read(A); Temp := A*0.1; A:= A-Temp; Write(A); Read(B); B := B+Temp; Write(B).



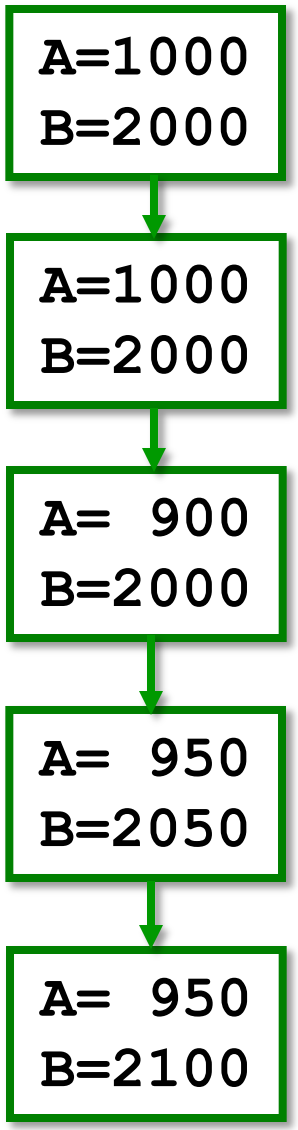
Planificación #3: concurrente serializable

T_0	T_1
1. Read(A);	
2. A:= A-50;	
3. Write(A);	
	4. Read(A);
	5. Temp := A*0.1;
	6. A:= A-Temp;
	7. Write(A);
8. Read(B);	
9. B := B+50;	
10.Write(B).	
	11.Read(B);
	12.B := B + Temp;
	13.Write(B).



Planificación #4: concurrente no serializable

T_0	T_1
1. Read(A);	
2. A:= A-50;	
	3. Read(A);
	4. Temp := A*0.1;
	5. A:= A-Temp;
	6. Write(A);
	7. Read(B);
8. Write(A);	
9. Read(B);	
10.B := B+50;	
11.Write(B).	
	12.B := B+Temp;
	13.Write(B).



Equivalencia entre Planificaciones

- Por definición, asumimos que cada transacción preserva la consistencia de la base de datos (responsabilidad humana). Así, la ejecución en serie de un conjunto de transacciones también preserva la consistencia de la base de datos.
- Una **planificación concurrente**, es **serializable** si *es equivalente a alguna planificación en serie*.
- Vamos a ver dos formas de equivalencia y serializabilidad entre planificaciones:
  1. Seriabilidad en **conflictos.**
  2. Seriabilidad en **vistas.**

Transacciones

- En lo que sigue, se asume la una vista simplificada de las transacciones:
  - En general, se ignoran las operaciones que no se corresponden a una instrucción **read** o **write.**
  - Se asume que una transacción puede realizar un número arbitrario de cálculos sobre los datos en sus buffers locales entre instrucciones **reads** y **writes**.
  - Las planificaciones simplificadas consisten básicamente de instrucciones **read** y **write.**

Conflicto en Planificaciones

Sea una planificación **S** en las que existen dos *instrucciones consecutivas* $\mathbf{I_i}$ *e* $\mathbf{I_j}$ de las transacciones $\mathbf{T_i}$ y $\mathbf{T_j}$ respectivamente.

- Si $\mathbf{I_i}$ e $\mathbf{I_j}$ se refieren a datos diferentes no hay diferencias si cambian de orden.
- Si $\mathbf{I_i}$ **= Read(Q)** y $\mathbf{I_j}$ **= Read(Q)** el orden no importa ya que ambas transacciones leen el mismo valor de Q.
- El **conflicto** se produce cuando ambas refieren al mismo dato Q y *al menos una de las instrucciones* $\mathbf{I_i}$ o $\mathbf{I_j}$ es un **Write(Q).**

Conflicto en Planificaciones

El **conflicto** con instrucciones consecutivas se produce si al menos una de las instrucciones I_i e I_j es una instrucción Write(Q):

- Si **I_i = Read(Q)** y **I_j = Write(Q).** Si I_i viene antes que I_j entonces T_i no lee el valor de Q que escribe T_j. Si I_j viene antes que I_i entonces T_i lee el valor que escribe T_j. El orden si importa.
- Si **I_i = Write(Q)** y **I_j = Read(Q)** el caso es análogo al anterior.
- Si **I_i = Write(Q)** y **I_j = Write(Q)**. El orden de las instrucciones afecta a las próximas sentencias Read de la planificación S.

Conflictos en Planificaciones

Definición:

Dos instrucciones $\mathbf{I_i}$ e $\mathbf{I_j}$ referentes a dos transacciones $\mathbf{T_i}$ y $\mathbf{T_j}$ respectivamente. Se dice que tales instrucciones están en **conflicto** *cuando son instrucciones de transacciones distintas sobre el mismo dato Q y al menos una de ellas es una instrucción Write.*

Test de Seriabilidad en Conflictos

- Sean I_i e I_j instrucciones consecutivas en una planificación S.
- Si I_i e I_j son instrucciones de diferentes transacciones que no se están en conflicto, entonces pueden intercambiarse para obtener una planificación S' equivalente a S.
- S y S' son planificaciones equivalentes ya que todas las instrucciones aparecen en el mismo orden salvo para las instrucciones I_i e I_j, cuyo orden no importa.

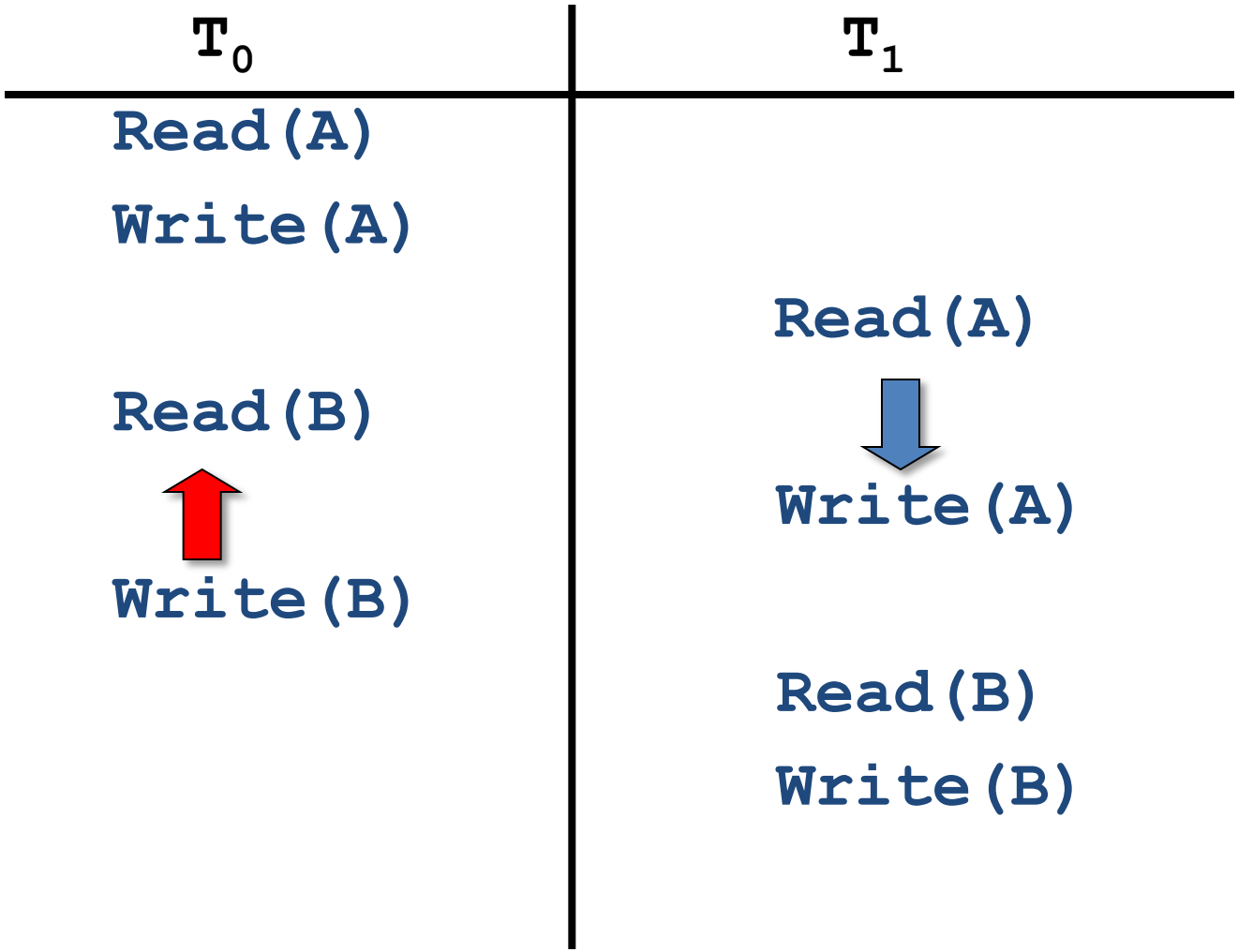
Test de Seriabilidad en Conflictos

- Si S puede transformarse en S' intercambiando instrucciones no conflictivas se dice que S y S' son equivalentes en cuanto a conflictos.
- Una planificación S es serializable en cuanto a conflictos si es equivalente en conflictos a una planificación en serie.
- La equivalencia en cuanto a conflictos es una definición muy rigurosa (restrictiva) pero que se puede probar de manera eficiente.

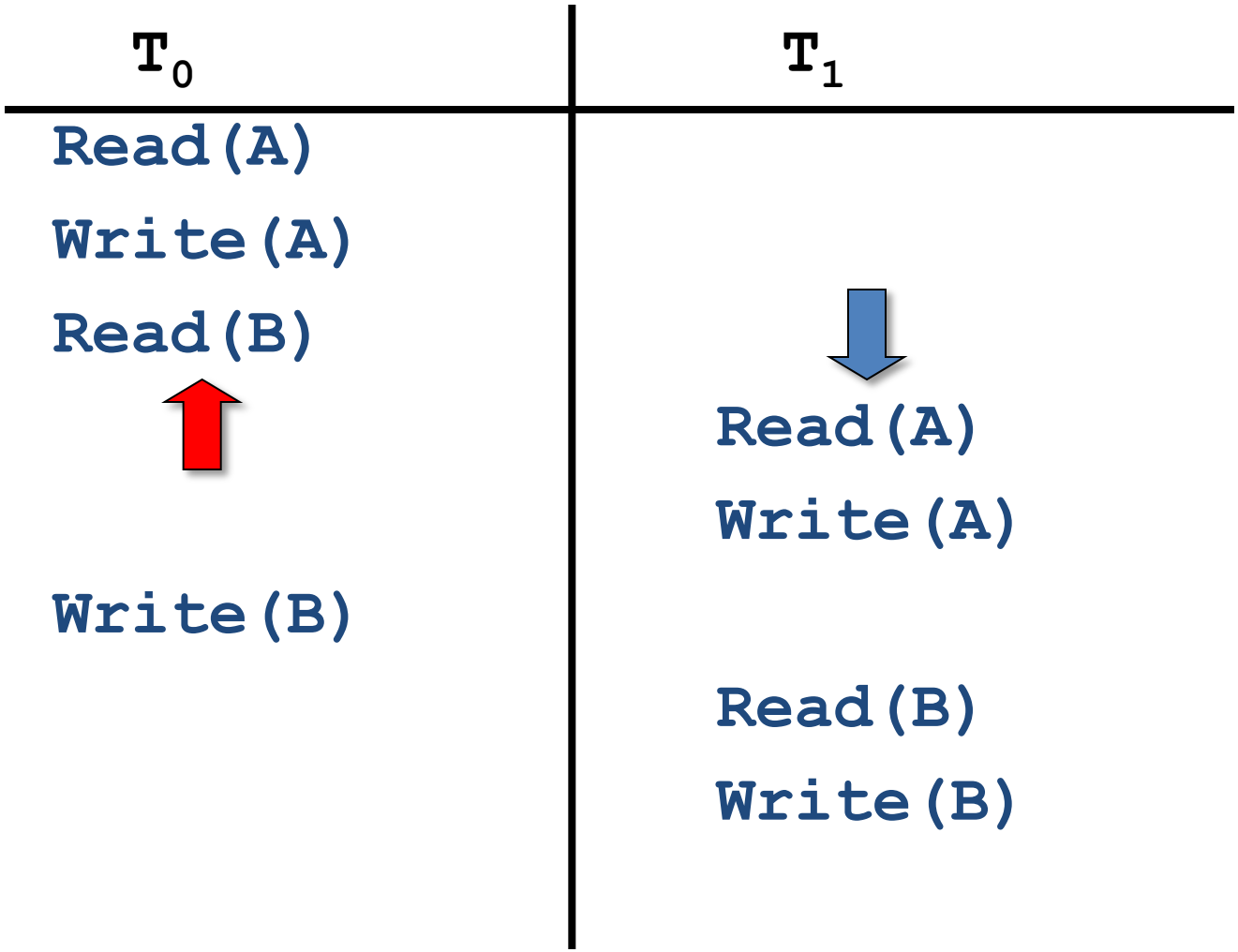
Planificación serializable en conflictos

T_0	T_1
Read(A)	
Write(A)	
	Read(A)
	Write(A)
Read(B)	
Write(B)	
	Read(B)
	Write(B)

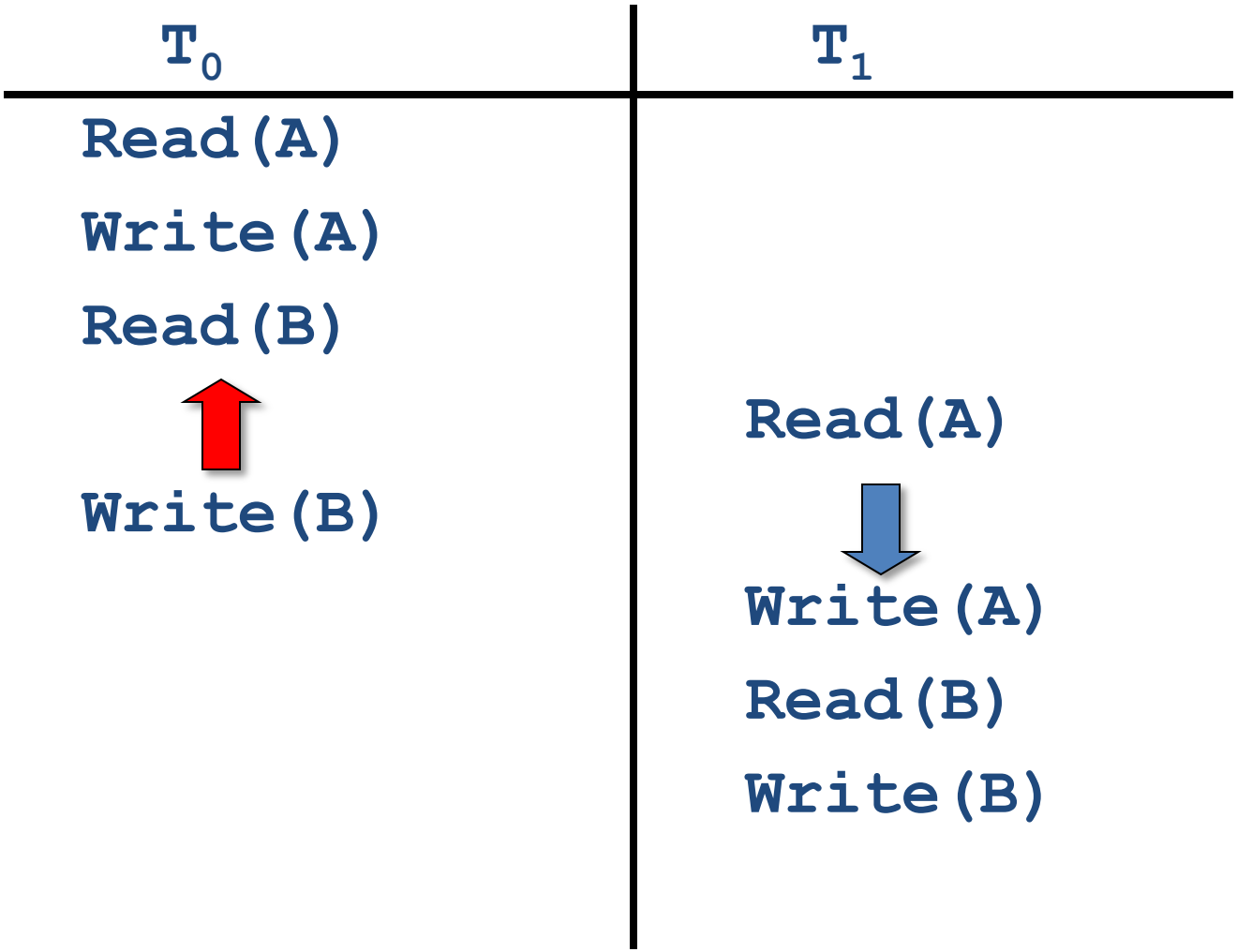
Planificación serializable en conflictos



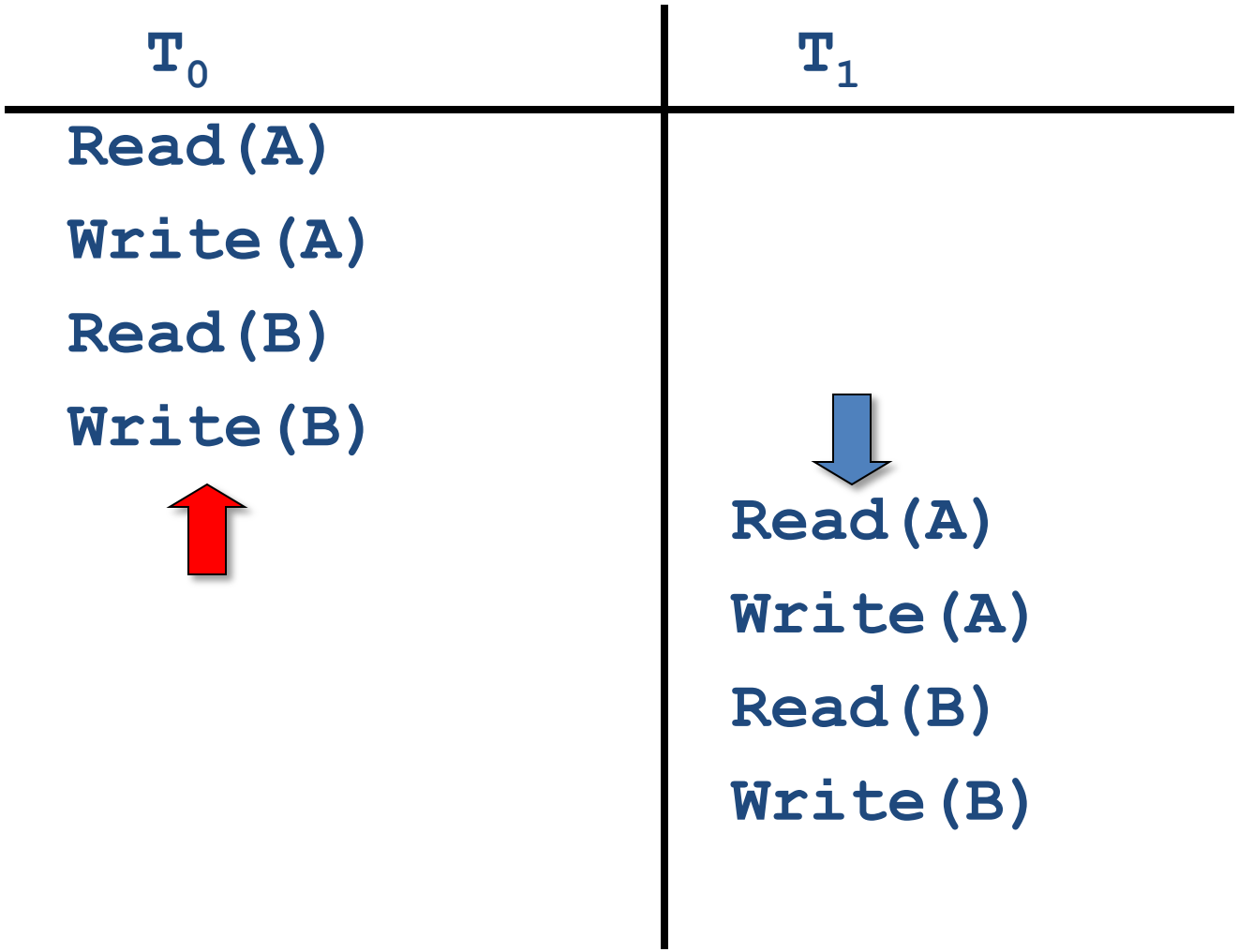
Planificación serializable en conflictos



Planificación serializable en conflictos



Planificación serializable en conflictos



Planificación serializable en Conflicto

- Una planificación S es *serializable en cuanto a conflictos* si es equivalente en conflictos a una planificación en serie.
- Sin embargo, es posible tener dos planificaciones que produzcan la misma salida pero que no sean serializables en cuanto a conflictos.

Serializable en Conflictos ⇨ Serializable

Serializable ⇦̸ Serializable en Conflictos

Planificación no serializable en conflictos

T_0	T_1
Read(A); A := A-50; Write(A)	
	Read(B); B := B-10; Write(B);
Read(B); B := B+50; Write(B).	
	Read(A) A := A+10; Write(A).

Equivalente a la planificación en serie $<T_0,T_1>$.

No serializable por conflicto entre instruc-ciones.

Pruebas de serializabilidad

- Una planificación es serializable en conflictos si se puede transformar intercambiando instrucciones consecutivas en una planificación en serie.
- Sin embargo, no es eficiente intentar alcanzar una planificación en serie “probando” este tipo de intercambios.
- Un algoritmo más eficientes que determina si una planificación S es serializable **construye un grafo de precedencia** para S y evalúa las propiedades del grafo.

Grafo de Precedencia

Consideremos alguna posible planificación para un conjunto de transacciones $T_1, T_2, ..., T_n$

Grafo de Precedencia — **G = (V, A)** un grafo dirigido donde cada vértice V_i se corresponde con cada transacción T_j.

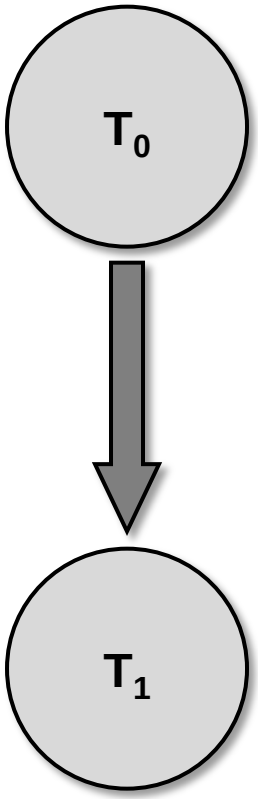
- Se añade una arco desde T_i **a** T_j **(**$T_i \rightarrow T_j$**)** cuando se da alguna de las siguientes condiciones:
  - T_i ejecuta write(Q) antes que T_j ejecute read(Q).
  - T_i ejecuta read(Q) antes que T_j ejecute write(Q).
  - T_i ejecuta write(Q) antes que T_j ejecute write(Q).

Análisis del Grafo

- Si existe una arista $T_i \rightarrow T_j$ entonces se indica que en cualquier planificación en serie equivalente T_i debe preceder a T_j.
- Si el grafo de precedencia de S tiene un ciclo entonces la planificación S *no es serializable en conflictos*.
- Inversamente, si el grafo de precedencia de S no tiene ciclos entonces *es serializable en conflictos*.
- Si el grafo tiene *n* vértices, necesitamos un algoritmo del $O(n^2)$ para detectar ciclos en el grafo.

Planificación 1: grafo de precedencia

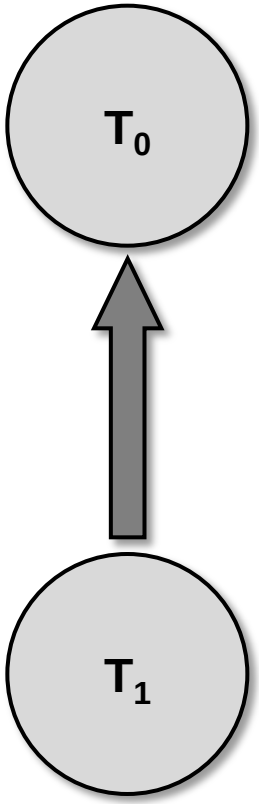
T_0	T_1
Read(A); A := A-50; Write(A); Read(B); B := B+50; Write(B).	Read(A); Temp := A*0.1; A := A-Temp; Write(A); Read(B); B := B+Temp; Write(B).



El arco $T_0 \rightarrow T_1$ existe pues las escrituras de T_0 deben preceder a las lecturas y escrituras de T_1.

Planificación 2: grafo de precedencia

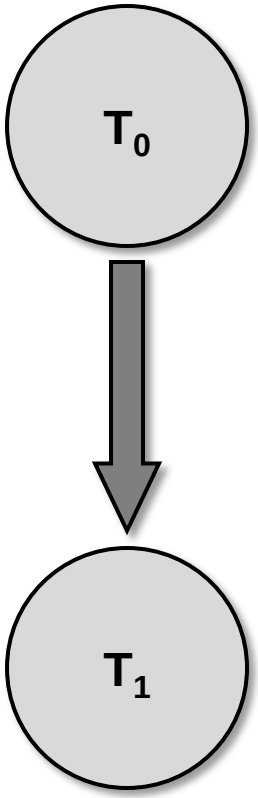
T_0	T_1
Read(A); A := A-50; Write(A); Read(B); B := B+50; Write(B).	Read(A); Temp := A*0.1; A := A-Temp; Write(A); Read(B); B := B+Temp; Write(B).



El arco $T_1 \rightarrow T_0$ existe pues las escrituras de T_1 deben preceder a las lecturas y escrituras de T_0.

Planificación 3: grafo de precedencia

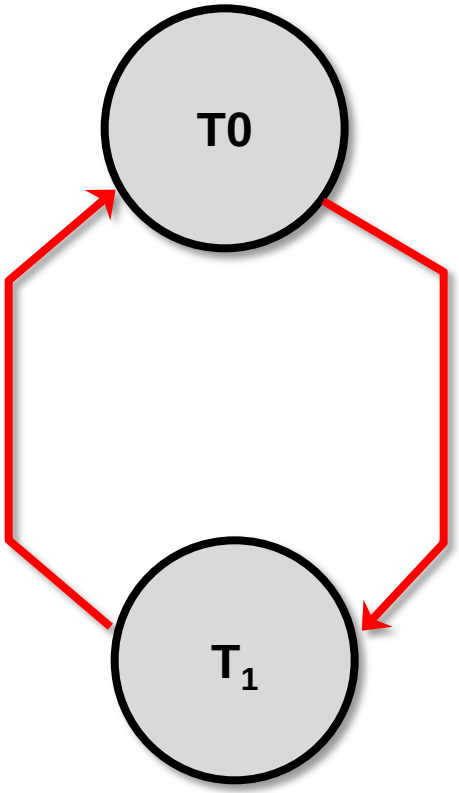
T_0	T_1
Read(A); A:= A-50; Write(A);	
	Read(A); Temp := A*0.1; A:= A-Temp; Write(A);
Read(B); B := B+50; Write(B).	
	Read(B); B := B+Temp; Write(B).



Las dos lecturas de T_1 deben ser posteriores a las dos escrituras de T_0 respectivamente.

Planificación 4: grafo de precedencia

T_0	T_1
Read(A); (1)	
A := A-50;	
	Read(A);
	Temp := A*0.1;
	A := A-Temp;
	Write(A); (2)
	Read(B); (3)
Write(A);	
Read(B);	
B := B+50;	
Write(B); (4)	
	B := B+Temp;
	Write(B).



El arco $T_0 \rightarrow T_1$ existe pues (1) debe preceder a (2) y el arco $T_1 \rightarrow T_0$ existe pues (3) debe preceder a (4).

Temas de la clase de hoy

- **Transacciones**
  - Propiedades de las transacciones: ACID.
  - Estados de una transacción.
- **Ejecuciones concurrentes**
  - Serializabilidad.
    - Serializabilidad en cuanto a conflictos.
    - Pruebas de serializabilidad en cuanto a conflictos.
- **Bibliografía**
  - *Database System Concepts*– A. Silberschatz. Capítulo 15.
  - *DataBase System – The Complete Book* – H. Molina,