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

Elementos de Bases de Datos

Segundo Cuatrimestre 2015

Clase 14:

Gestión de Transacciones
Control de Concurrencia

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

Transacciones

Transacciones - Repaso

Se denomina **transacción** a una colección de operaciones sobre la base de datos que forman una *unidad lógica de trabajo*.

Características

- Eventualmente puede acceder y/o actualizar varios ítems de datos.
- Accede y deja a la BD en un estado consistente.
- Puedes suceder que múltiples transacciones se ejecuten concurrentemente.
- Es el programador quién define los límites de una transacción.

Propiedades ACID

- ✓ **Atomicidad** *(atomicity)*. — DBMS
- ✓ **Consistencia** *(consistency)*. — Programador
- ✓ **Aislación** *(isolation)*. — DBMS
- ✓ **Durabilidad** *(durability)*. — DBMS

Planificaciones

PLANIFICACIÓN se denomina así a la secuencia cronológica en el cual se ejecutan en el sistema las instrucciones de transacciones concurrentes.

$$P_k = \langle I_{i,j} \rangle \quad (\text{con } I_{i,j} \text{ instrucción j-ésima de } T_i)^1$$

[1]- Framework centralizado

- Se dice de una planificación que es una **planificación es en serie** si las instrucciones pertenecientes a cada transacción aparecen juntas.
- Una planificación *no en serie* que es una "**planificación serializable"** si el resultado de su ejecución *equivale a alguna* planificación en serie. ☺

Planificaciones Serializables

Estrategias de Testeo

Equivalencia

Dos planificaciones **P_i** y **P_j** son **planificaciones equivalentes**, si a pesar de *no tener la misma secuencia de instrucciones* ***I_{ij}***, *generan el mismo resultado* final.

Tests de Equivalencia

- Existen distintas mecanismos para testear o verificar la calidad de serializabilidad de una planificación:
  1. Test de **serializabilidad en conflictos** (test con *grafo de conflictos*) ✓
  2. **Test de serializabilidad en vistas.**

Conflictos

Sean las 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 y al menos una de ellas es una instrucción Write.*

- Si I_i e I_j no están en conflicto, pueden intercambiarse para obtener una planificación S' equivalente a S.
- La forma práctica de verificarlo es construir el grafo de precedencia para conflictos.

Instrucciones Conflictivas

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.

Grafo de Precedencia en Conflicto

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

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).

Dada una planificación P, se construye el grafo para determinar si es o no serializable en cuanto a conflictos.

Ejemplos

¿Es serializable?

Serializable en Conflictos

P1	T1	T2
1	Read (A)	
2	Write (A)	
3		Read (A)
4		Write (A)
5	Read (B)	
6	Write (B)	
7		Read (B)
8		Write (B)

No serializable en Conflictos

P2	T1	T2
1	Read (A)	
2	Write (A)	
3		Read (A)
4		Write (A)
5		Read(B)
6	Read (B)	
7	Write (B)	
8		Write (B)

Equivalencia en Vistas

Dos planificaciones **S y S' son equivalentes en cuanto a vistas** si cumplen *todas* las siguientes condiciones para cada ítem de dato Q:

1. Si T_i ejecuta Read(Q) y lee el valor inicial de Q en **S**, entonces T_i debe leer el valor inicial de Q en **S'.**
2. Si T_i ejecuta Read(Q) en **S** y ese valor fue producido por T_j (si existe), entonces T_i debe leer en **S'** el valor producido por T_j.
3. La transacción (si existe) que ejecuta Write(Q) final en la planificación **S** debe ejecutar la operación final Write(Q) en la planificación **S´**

- **1 y 2** aseguran que cada transacción lee los mismos valores en ambas planificaciones y, por lo tanto, realiza el mismo cálculo.
- **3**, junto con **1 y 2**, asegura que ambas planificaciones resultan en el mismo estado final.

Serializabilidad de Vistas

T_1	T_2	T_3
Read(Q);	Write(Q).	
Write(Q).		Write(Q).

- Es serializable en vistas y es equivalente a la planificación en serie **<T_1,T_2,T_3>.**
- No es serializable en conflictos *¿grafo?*

- Las transacciones T_2 y T_3 realizan escrituras sobre Q sin haberlo leído: *escrituras ciegas.* Las planificaciones serializables en vistas (no en conflictos) admiten escrituras ciegas.

Serializabilidad de Vistas

- Una planificación es **serializable en vistas** si es equivalente en vistas a alguna planificación en serie.
- La equivalencia en vistas es menos rigurosa que la equivalencia en conflictos.
- Toda planificación serializable en conflictos también es serializable en vistas

  Serializabilidad en conflictos ⇨ **Serializabilidad en vistas**

- Sin embargo, no es cierto que toda planificación serializable en vistas también lo es en conflictos.

  Serializabilidad en vistas ⇏ **Serializabilidad en conflictos**

Planificaciones

Serializabilidad y Equivalencias

- Las siguiente planificación produce una salida equivalente a la planificación en serie <T_1,T_5>. Sin embargo, *no es equivalente en cuanto a vistas* (ie ni en cuanto a conflictos)

T_1	T_5
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)

Vistas vs. Conflictos

- Los módulos para testear serializabilidad en vistas tienen un costo exponencial relativo al tamaño del grafo de precedencia *(ver anexo)*.

- El problema de chequear si una planificación es serializable en vistas cae en la clase de problemas NP-completos. Así la existencia de un algoritmo eficiente es extremadamente poco probable.

Control de Concurrencia

Transacciones y acceso a recursos compartidos

Implementación de Aislamiento

Vamos a estudiar alternativas en **protocolos de control de concurrencia** como mecanismos para **garantizar aislamiento**:

- Proporcionar concurrencia.
- Controlar el acceso a los datos compartidos.
- Generando únicamente planificaciones serializables.

RECORDAMOS

- Los entornos concurrentes proveen las ventajas:
  - Mejor la utilización de disco y procesador, aumentando la productividad (*troughput*).
  - Mejoran el tiempos de respuesta.

Control de Concurrencia

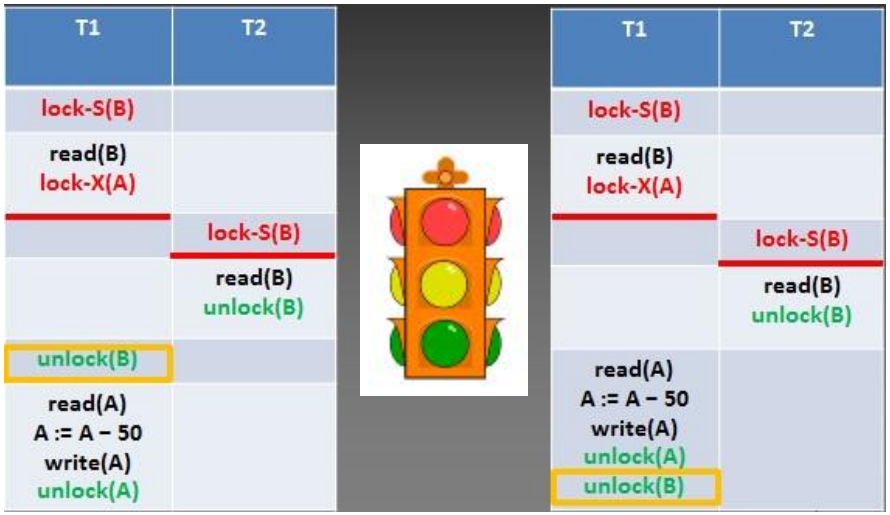
- Un **protocolo de control de concurrencia** garantiza que se generen planificaciones *concurrentes*, pero asegurando *serializabilidad*.
  - Los protocolos que vamos a estudiar **no** examinan el grafo de precedencia.
  - Imponen reglas que evitan planificaciones no serializables (pesimistas).
  - Los distintos protocolos varían en cuanto al paradigma, al nivel de concurrencia y el overhead.
- Los tests de serializabilidad ayudan a entender por qué un protocolo es correcto.

Control de concurrencia

- Los protocolos de control de concurrencia sirven para garantizar *aislamiento*.
- Existen dos estrategias básicas para control de concurrencia:
  - Basadas en bloqueo de recursos.
  - Basadas en estampillas de tiempo de las transacciones.

➔ Distintas implementaciones usan una u otra estrategia o una combinación de ellas.

Control de Concurrencia: Protocolos Basados en Bloqueos



Estrategia: bloqueos de recursos

Protocolos Basados en Bloqueos (PBB)

- Un **bloqueo (*Lock*)** es un mecanismo para controlar el acceso concurrente a un recurso (ítem de dato).

Modos de bloqueo:

Compartido *(lock-S):* si una transacción T solicita y obtiene un bloqueo compartido sobre un dato Q entonces T puede leer el dato pero no escribir Q.

Exclusivo *(lock-X)*: si una transacción T solicita y obtiene un bloqueo exclusivo sobre un dato Q entonces T puede leer y escribir Q.

- Las transacciones envían sus pedidos de bloqueo al *Gestor de Control de Concurrencia*.

Gestor de Control de Concurrencia

1. Una transacción que requiere un bloqueo (compartido o exclusivo) sobre un ítem de dato Q lo solicita al gestor.
2. El gestor de control de concurrencia se lo concederá únicamente si su *solicitud es compatible* con los bloqueos ya asignados a otras transacciones sobre el mismo dato Q. Caso contrario T deberá esperar.

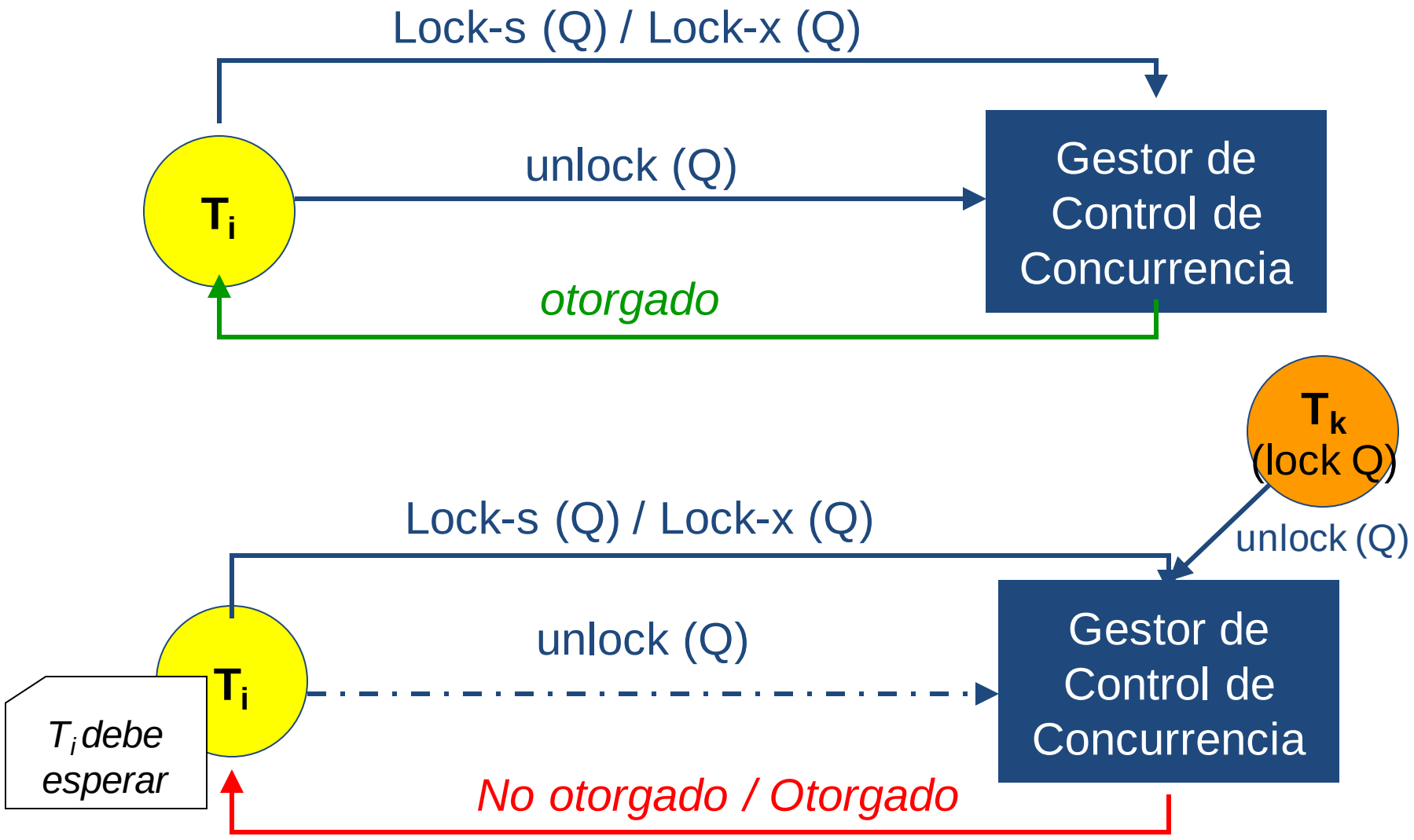
Tabla de compatibilidad de bloqueos:

Bloqueos previos

Compatibilidad	Lock-S	Lock-X
Lock-S	**Si**	**No**
Lock-X	**No**	**No**

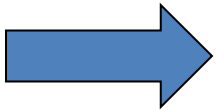
Pedido de bloqueo sobre Q

Otorgamiento de bloqueos



Protocolo Basado en Bloqueos (PBB)

```
T:    read(A);
      write(A);
      read(B);
      write(B);
      read (C);
```



```
T: lock-x(A);
   read(A);
   write(A);
   unlock(A);
   lock-x(B);
   read(B);
   write(B);
   unlock(B);
   lock-s(C);
   read(C);
   unlock(C);
```

Protocolos – Propiedades

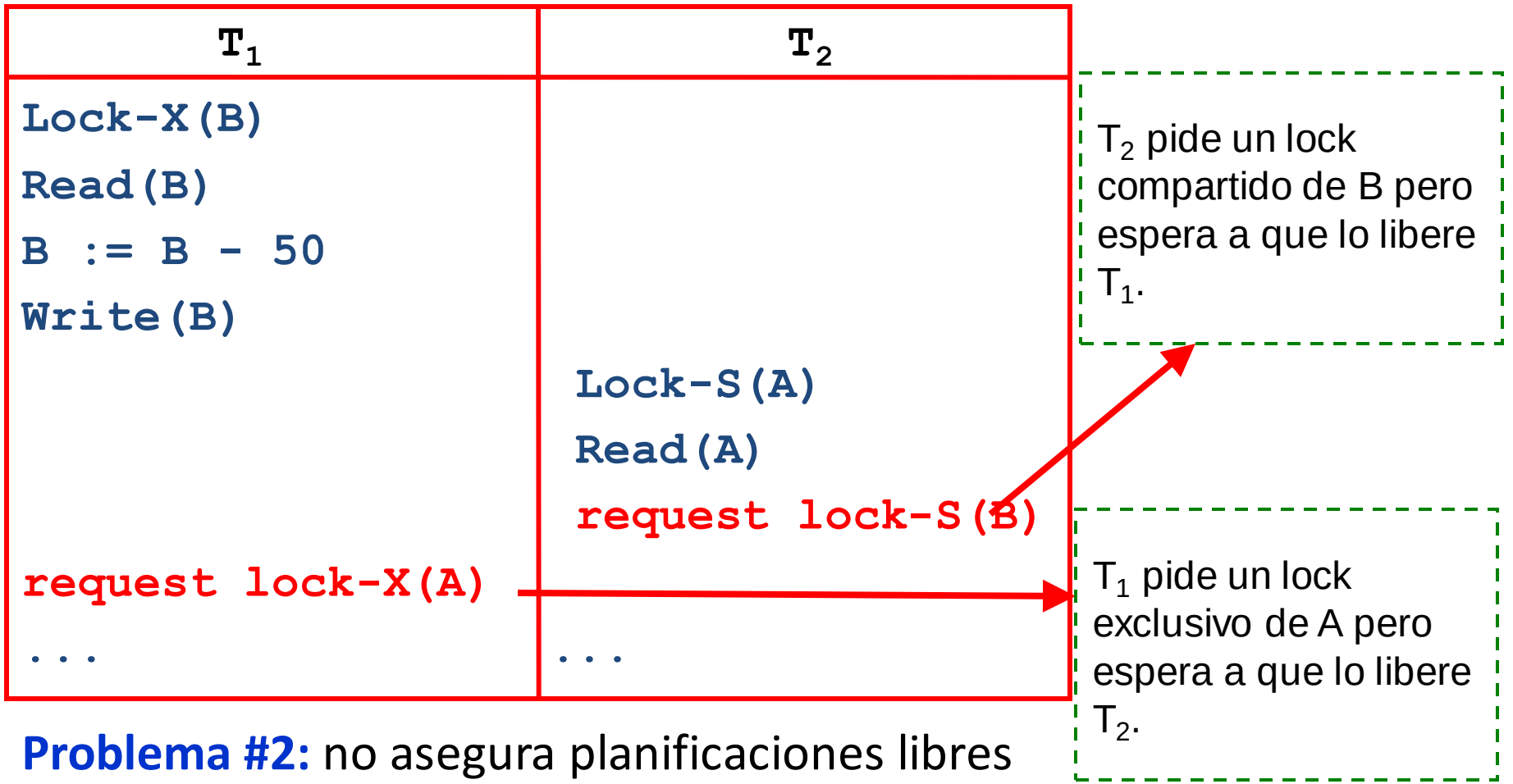
- **SERIALIZABILIDAD** ¿el protocolo asegura generar sólo planificaciones serializables?
- **DEADLOCK** ¿el protocolo genera situaciones en donde un grupo de transacciones no avanza esperando infinitamente recursos que tiene otra?
- **INANICIÓN** ¿el protocolo genera situaciones donde una transacción espera infinitamente por algún recurso?
- **RETROCESO EN CASCADA** ¿genera situaciones en donde una transacción que aborta arrastra a otra/s transacción/es?
- **PLANIFICACIONES NO RECUPERABLES** ¿genera situaciones en donde una transacción que aborta arrastraría a otra/s transacciones cometidas?

Protocolo Basado en Bloqueos

- **Problema #1**: El mecanismo de solicitud de bloqueos a libertad no garantiza solo planificaciones serializables.

	T_1	T_2
1	**Lock-x (A)**	
2	Read (A)	
3	A = A - 50	
4	Write (A)	
5	**Unlock (A)**	
6		**Lock-S(B)**
7		Read (B)
8		tt = B*0.1
9		**Unlock (B)**
10		**Lock-X(A)**
11		Read (A)
12		A = A + tt
13		Write (A)
14		**Unlock (A)**
15	**Lock-x(B)**	
16	Read (B)	
17	B = B + 50	
18	Write (B)	
19	**Unlock (B)**	

PBB – Deadlock



Problema #2: no asegura planificaciones libres de deadlock

El sistema deberá **retroceder** alguna de las transacciones

Problemas del PBB

☹ El PBB no está libre de *deadlock*.

- **Solución**: incluir alguna política de administración de deadlock (detección o prevención).

☹ PBB también puede causar problemas de *inanición* (*starvation*), esto es, transacciones que nunca alcanzan a disponer de los recursos que necesitan, porque son otorgados antes a otras transacciones.

- **Solución**: establecer políticas de otorgamiento de bloqueos con prioridades.

☹ PBB genera planificaciones no serializables, esto es significa, estados inconsistentes de la BD.

- No es un protocolo de control de concurrencia que garantice serilizabilidad.

Protocolo de Bloqueo de 2 Fases PB2F

Solución: requerir que cada transacción haga sus solicitudes de bloqueos y desbloqueos en dos fases:

- **Fase de Crecimiento**: Una transacción puede obtener nuevos bloqueos pero *no puede liberar ningún bloqueo.*
- **Fase de Encogimiento**: Una transacción puede liberar bloqueos pero *no puede obtener ningún bloqueo nuevo.*

- El **PB2F** garantiza planificaciones **serializables en conflictos**.

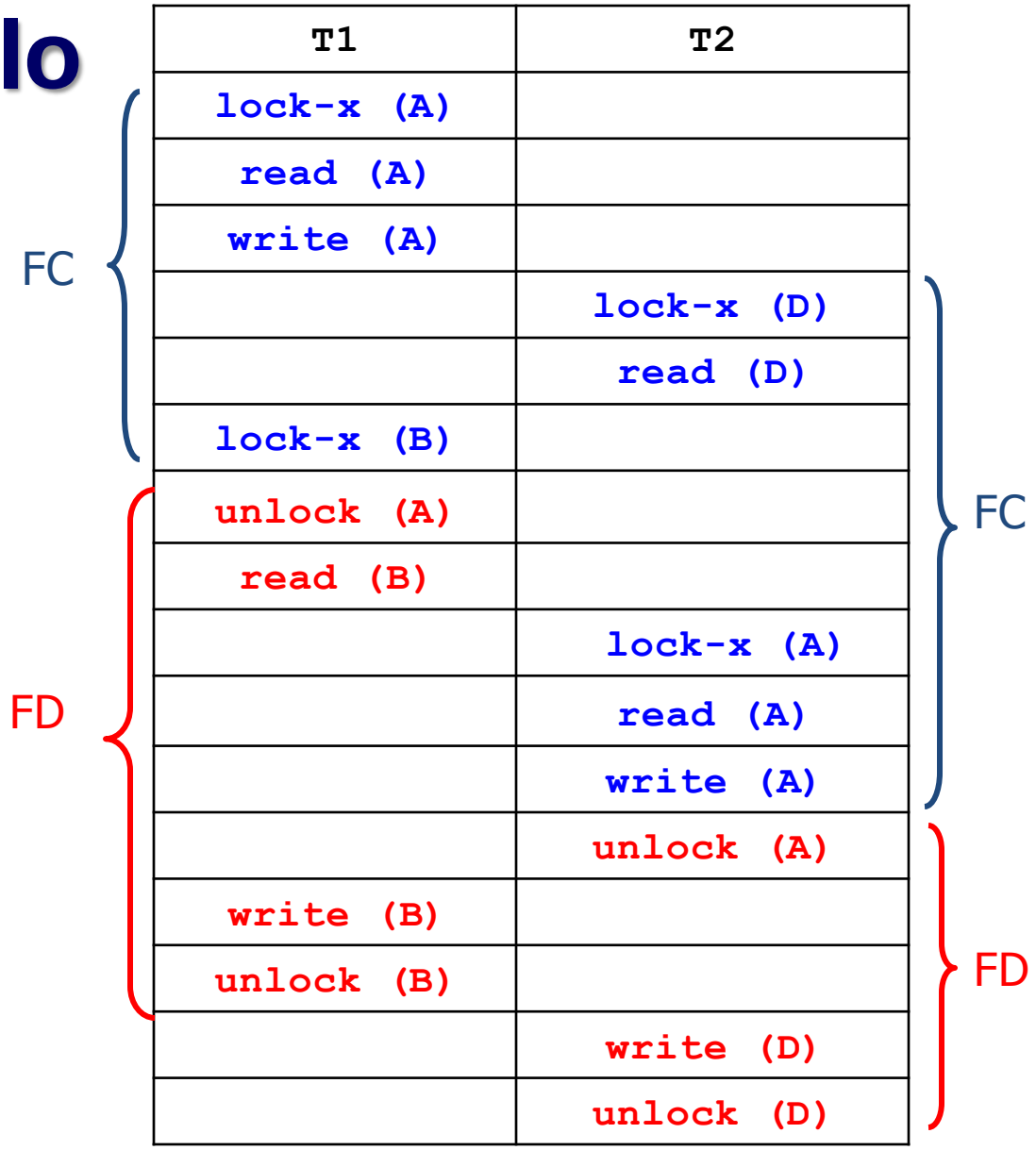
Ejemplo

T_1 = read(A); write(A); read(B); write(B);

T_2= read(D); read(A); write(A); write(D);

Serie equivalente:

$<T_1, T_2>$



Protocolo de Bloqueo de 2 Fases

Características de PB2F

- El PB2F asegura serializabilidad en conflictos.
- La serie equivalente queda definida según cada *punto de lock* (último lock obtenido).

- No está libre de deadlock.

Definiciones

- Si una transacción falla durante su ejecución, para asegurar *atomicidad*, se deben deshacer sus efectos.
- Así, en un sistema concurrente, si existe una transacción T_j *que depende de* T_i (T_j leyó un dato escrito por T_i) T_j también deberá ser retrocedida.

Planificación no recuperable: se dice que una planificación es no recuperable si existe T_j *que depende de* T_i, T_j comete antes que T_i termine y posteriormente ocurre una falla de T_i

Retrocesos en cascada: se dice que una planificación tiene retrocesos en cascada si existe T_k *que depende de* T_j, T_j *que ... que depende de* T_i Si falla T_i, deberán retroceder en cascada T_i , T_j, T_k

Retroceso en Cascada

T_1= read(A); write(A);
read(B); write(B);

T_2= read(A);
read(B); C=A+B; write(C)

⇨ **La falla de T_1 arrastra a T_2**

	T_1	T_2
1	Lock-x (A)	
2	read (A)	
3	write (A)	
4	Lock-x (B)	
5	unlock(A)	
6		Lock-x (A)
7		read (A)
8	read (B)	
9	*falla T1*	
	…	…

Retrocesos en Cascada

T1	T2	T3	T4
read (A)			
write (A)			
	read (A)		
	write (B)		
		read(B)	
			read(A)
			write (A)
falla T1			
	…	…	. .

T2, T3 y T4 Retroceden en cascada

Planificación no recuperable

T_1= read(A); write(A); read(B); write(B);

T_2= read(A); read(C); C=A; write(C)

⇨ **La falla de T_1 debería arrastrar a T_2 que cometió.**

	T_1	T_2
1	Lock-x(A)	
2	read(A)	
3	write(A)	
4	Lock-x(B)	
5	unlock(A)	
6		Lock-s(A)
7		read(A)
8		Lock-x(C)
9		unlock(A)
		read (C)
		C:= A
10		write(C)
11		unlock(C)
12		*Commit T_2*
13	read(B)	
14	*Falle T1*	
15		

Problemas del PB2F

- El PB2F tiene las siguientes desventajas:
  - ☹ Posibilidad de que alguna planificación caiga en *deadlock*.
  - ☹ Puede generar *retrocesos en cascada* (*cascading rollback*), esto es, ante la falla de una transacción, se genera que fallen otras transacciones.
  - ☹ Puede generar *planificaciones no recuperables*, como resultado de un retroceso en cascada.

PB2F Estricto

Variantes a PB2F: **Protocolo de Bloqueo de 2 Fases Estricto**. Similar al PB2F, donde

- Cada transacción debe conservar todos los bloqueos exclusivos que solicitó hasta alcanzar el estado cometida.
- De esta manera se garantiza:
  - Que el dato "escrito" por una transacción $\mathbf{T_i}$ no pueda ser "leído" (ni escrito) por otra $\mathbf{T_j}$ hasta que $\mathbf{T_i}$ cometa.
  - Solución a los problemas de retrocesos en cascada y planificaciones recuperables.

Ejemplo

Datas las transacciones:

- T_1= read(A); read(B); write(A); write(B);
- T_2= read(D); read(A); write(A);
- Esta planificación es equivalente a la serie **<T_1, T_2>**

T1	T2
lock-x (A)	
read (A)	
lock-x (B)	
read (B)	
	lock-s (D)
	read (D)
write (A)	
write (B)	
unlock (A,B)	
	lock-x (A)
	unlock (D)
	read (A)
	write (A)
	unlock (A)

Variante: PB2F Riguroso

- El **Protocolo de Bloqueo de 2 Fases Riguroso** exige que toda transacción mantenga todos sus bloqueos (sean exclusivos o compartidos) hasta que la transacción alcance el estado cometido.
- El orden de serializabilidad es el orden en que se comprometen las transacciones.

✍ Para analizar

- Los protocolos de PB2F Estricto y PB2F Rigurosos ¿solucionan el problema de deadlock?
- ¿Retrocesos en cascada?
- ¿Planificaciones no recuperables?

- ¿Se le ocurre algún mecanismo de bloqueo para evitar deadlock?

Variante: PB2F Refinado

- Supongamos una transacción T_i:

  T_i = read(A_1); read(A_2); ... read(A_n); write(A_1);

- El **PB2F Refinado** permite *conversiones de bloqueos*:
  - De **Lock-S** a **Lock-X**.
    - **Upgrade**: de modo compartido a exclusivo.
    - Se incluyen en la *fase de crecimiento*.
  - De **Lock-X** a **Lock-S**.
    - **Downgrade**: de modo exclusivo a compartido.
    - Se incluyen en la *fase de decrecimiento*.

PB2F Refinado: Ejemplo

Dadas las transacciones:

- T_1= read(A); read(B); write(A); write(B);

- T_2= read(D); read(A); write(A); write(D);

T1	T2
lock-s (A)	
read (A)	
lock-s (B)	
read (B)	
	lock-s (D)
	read (D)
upgrade (A)	
write (A)	
upgrade (B)	
write (B)	
unlock (A, B)	
	lock-s (A)
	read (A)
	upgrade (A)
	write (A)
	upgrade (D)
	write(D)
	unlock (A,D)

Implementación de bloqueos transparente

PB2F Refinado+ Riguroso

- Cuando una transacción T realiza una operación **Read(Q)** se ejecuta:
  - **Lock-S(Q); Read(Q)**
- Cuando una transacción T realiza una operación **Write(Q)** se ejecuta:
  - Si T tiene un acceso compartido entonces ejecuta:
    - **Upgrade(Q); Write(Q)**
  - De lo contrario, T ejecuta:
    - **Lock-X(Q); Write(Q)**
- Todos los bloqueos que tenga una transacción los conserva hasta que dicha transacción se comprometa o aborte.

Temas de la clase de hoy

- Serializabilidad en Vistas
- Control de concurrencia:
  - Protocolos de bloqueos de dos fases: 2 Fases, 2 Fases Estricto, 2 Fases Riguroso, 2 Fases Refinado

- **Bibliografía**
  - *Database System Concepts*– A. Silberschatz. Capítulo 15.
  - *DataBase System – The Complete Book* – H. Molina, J. Ullman. Capítulo 18.

Test de Serializabilidad de Vistas

- Para determinar la serializabilidad de vistas en una planificación, se construye un grafo de precedencia etiquetado.
- Sea S una planificación que involucra a las transacciones $\{T_1, T_2, \ldots, T_n\}$.
- Se agregan dos transacciones ficticias T_b y T_f.
  - T_b (transacción inicial) escribe cada dato leído por las transacciones de S.
  - T_f (transacción final) lee cada dato escrito por las transacciones de S.

Pruebas de Serializabilidad de Vistas

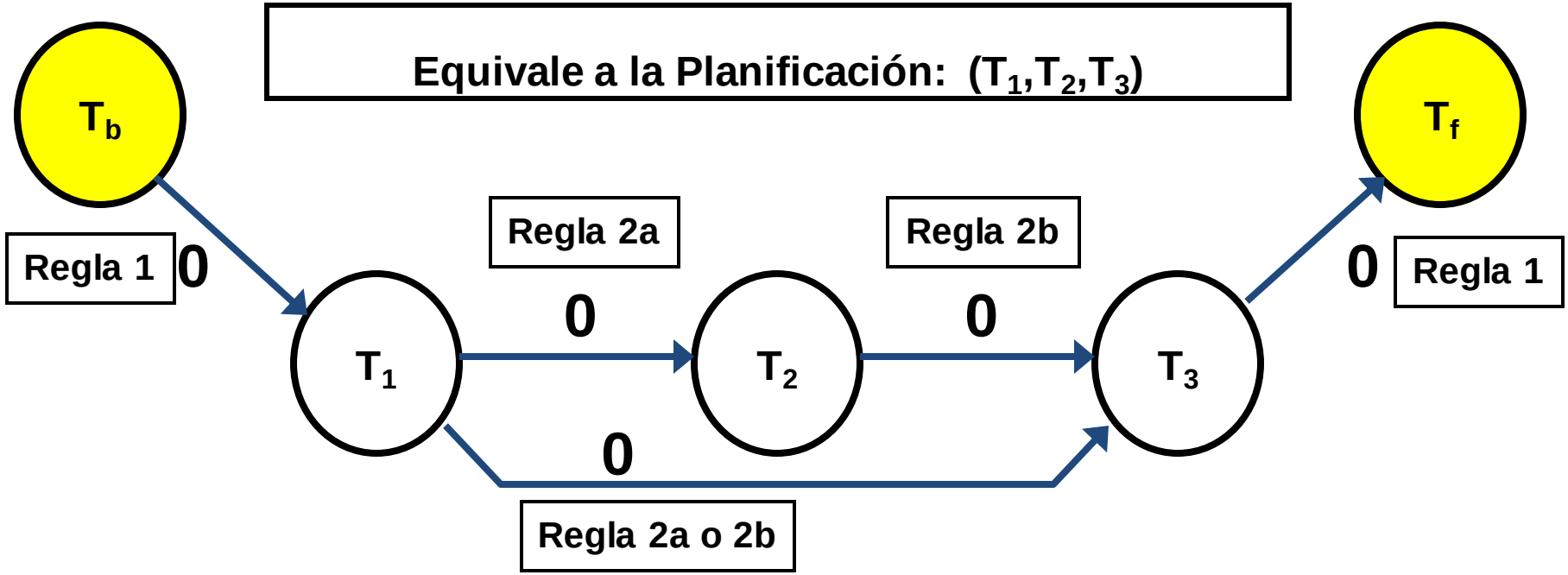
1. Se añade una arista $T_i \xrightarrow{0} T_j$ si la transacción T_j lee el dato Q escrito por T_i.
2. Por cada dato Q tal que T_j lee el valor de Q escrito por T_i, y T_k ejecuta Write(Q) tal que $T_k \neq T_b$ se hace lo siguiente:
   a. Si $T_i = T_b$ y $T_j \neq T_f$ entonces se inserta en el grafo la arista $T_j \xrightarrow{0} T_k$.
   b. Si $T_i \neq T_b$ y $T_j = T_f$ entonces se inserta en el grafo la arista $T_k \xrightarrow{0} T_i$.
   c. Si $T_i \neq T_b$ y $T_j \neq T_f$ entonces se insertan en el grafo las aristas $T_k \xrightarrow{p} T_i$ **y** $T_j \xrightarrow{p} T_k$, donde **p** es un número de etiqueta no usada en el grafo etiquetado.

Pruebas de Serializabilidad de Vistas

- Las reglas *a* y *b* son casos especiales que resultan de que T_b y T_f representan la primera y última transacción respectivamente.
- La regla *c* dice que si T_i escribe un dato que lee T_j y T_k escribe el mismo dato entonces T_k debe aparecer antes de T_i o bien después de T_j.
- Al aplicar la regla *c* no se requiere que T_k esté antes de T_i **y** después de T_j. Se exige que preceda a T_i, **o** bien suceda a T_j.
- Si el grafo de precedencia no contiene ciclos entonces la planificación es serializable en vistas.
- No obstante, la aparición de un ciclo (cuando existen etiquetas distintas de 0) no implica que la planificación no sea serializable en vistas.

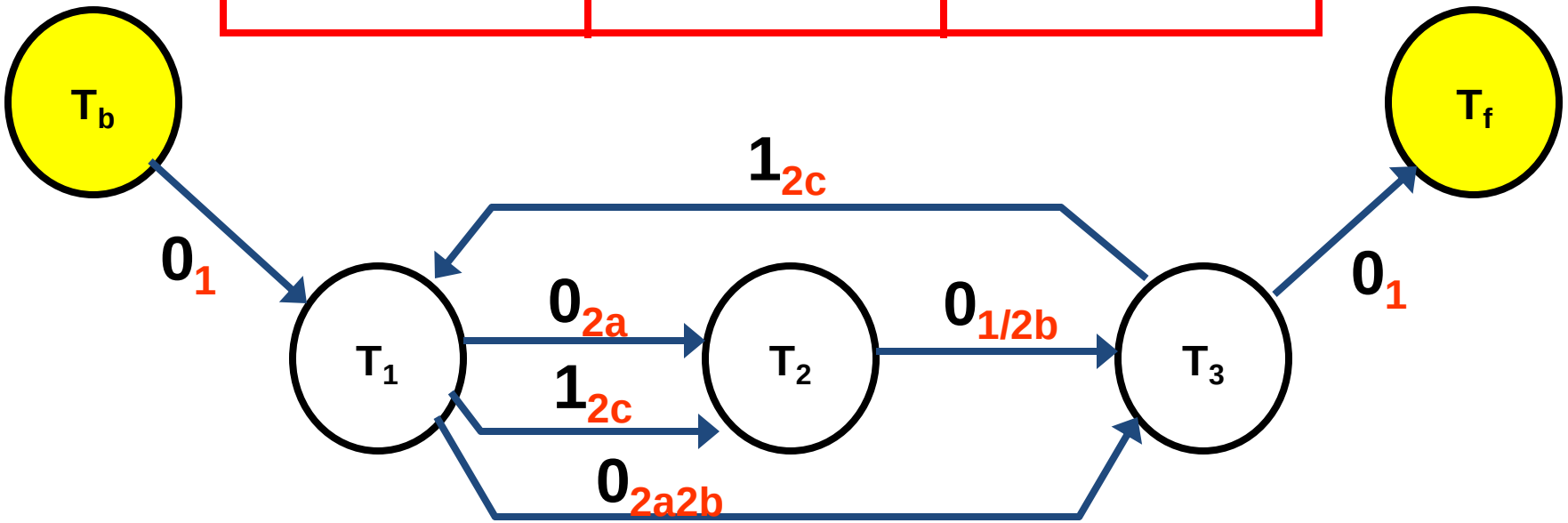
Planificación 1: serializable en vistas

T_1	T_2	T_3
Read(Q);		
	Write(Q).	
Write(Q).		
		Write(Q).



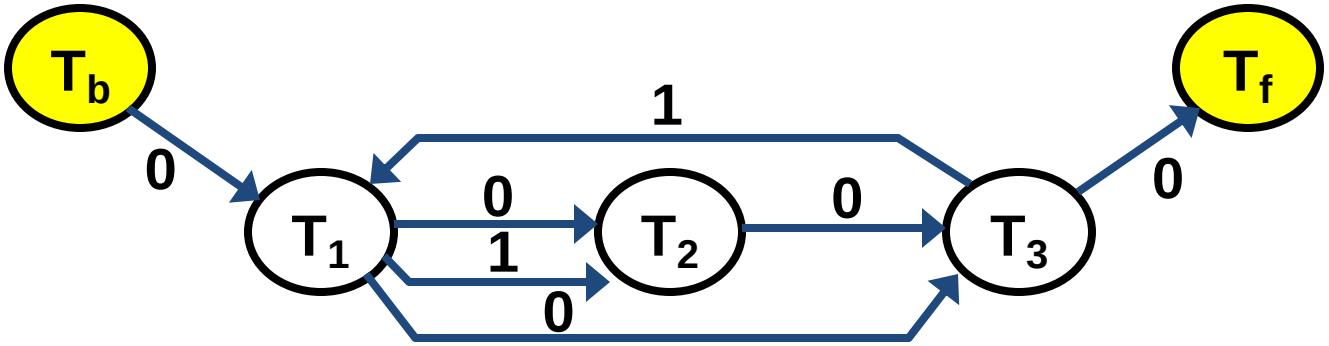
Planificación 2: serializable en vistas

T_1	T_2	T_3
Read(Q);		
	Write(Q).	
		Read(Q);
Write(Q).		
		Write(Q).

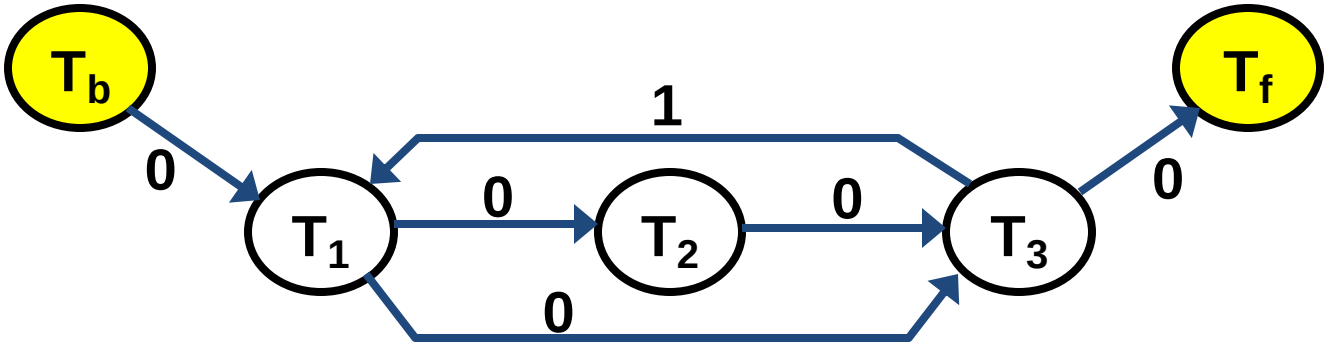


Los subíndices en las etiquetas indican las reglas que generan el arco.

Planificación 3: serializable en vistas

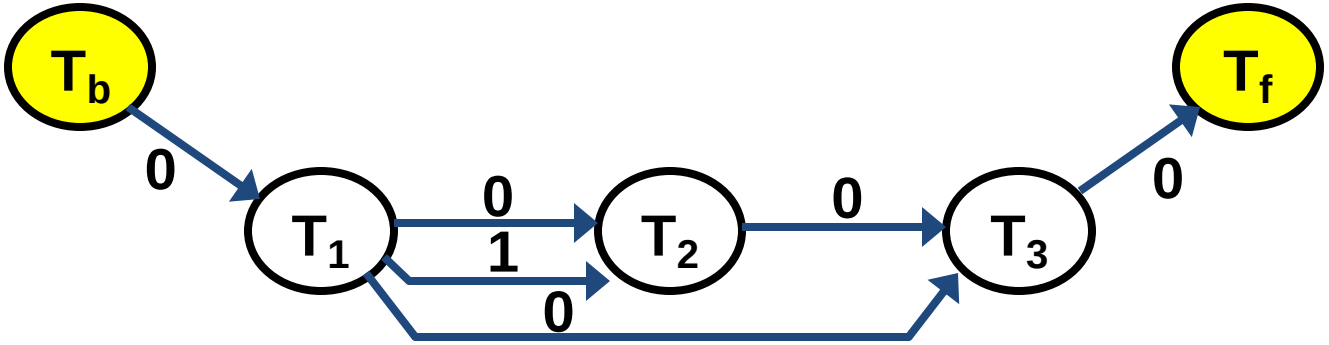


Los arcos con etiqueta 1 representan 2 opciones distintas.



1ra Opción:

Grafo Cíclico. ☹



2da Opción:

Grafo Acíclico. ☺

Equivale a la Planificación:

(T_1, T_2, T_3)