



UNIVERSIDAD  
DE GRANADA



# Administración de Bases de Datos

## Grado en Ingeniería Informática

### Gestión y control de concurrencia



I. J. Blanco, A. G. López Herrera

Departamento de Ciencias de la  
Computación e Inteligencia Artificial  
<http://decsai.ugr.es>

- Problemas producidos por la concurrencia
- Ejecuciones concurrentes sin conflicto
- Algoritmos de control de concurrencia

- Ejecución concurrente de transacciones para optimizar la eficiencia.
- Controlador de concurrencia gestiona el uso de recursos.
- Hay que garantizar la consistencia de la BD.

- Anulación de transacción: cuando una transacción quiere acceder a un dato que está siendo modificado por otra o cuando es eliminado por otra.

T1	T2
lee( $X, x_i$ )	
	lee( $X, x_i$ )
	$x_i := x_i + 1$
	escribe( $X, x_i$ )
$x_i := x_i * 3$	
escribe( $X, x_i$ )	

- Estado inconsistente de la BD:
  - transacciones concurrentes que violan temporalmente las restricciones de la DB
  - Supongamos que B es clave externa a A, luego no deben tener valores distintos en ningún momento.

T1	T2
lee(A,x <sub>i</sub> )	
x <sub>i</sub> :=x <sub>i</sub> +1	
escribe(A,x <sub>i</sub> )	
	lee(A,z <sub>i</sub> )
	imprime(z <sub>i</sub> )
	lee(B,z <sub>j</sub> )
	imprime(z <sub>j</sub> )
lee(B,x <sub>j</sub> )	
x <sub>j</sub> :=x <sub>j</sub> +1	
escribe(B,x <sub>j</sub> )	

T1	T2
lee(A,xi)	
xi:=xi+1	
escribe(A,xi)	
	lee(A,zi)
	zi := zi * 2
	escribe(A,zi)
	lee(B,zj)
	zj := zj * 2
	escribe(B,zj)
lee(B,xj)	
xj:=xj+1	
escribe(B,xj)	

## Operaciones en conflicto:

- pertenecen a distintas transacciones,
- acceden al mismo dato, y
- alguna de ellas ejecuta la orden `escribe()`

- Un SGBD ejecuta transacciones concurrentes, formadas por sentencias.
- **Plan de ejecución:** secuencia de instrucciones pertenecientes a todas las transacciones concurrentes, ejecutadas en un orden preciso.



- **Átomo:** fragmento de información en la BD cuyo acceso concurrente debe controlarse.
- ¿Atributo en una tupla?, ¿tupla?, ¿columna?, ¿tabla?
- A menor tamaño, más difícil de controlar (hacen falta más recursos); a mayor tamaño, más transacciones tienen que esperar su turno con el átomo.

- Pensemos en las dos transacciones anteriores:

T1
lee(A,x <sub>i</sub> )
x <sub>i</sub> := x <sub>i</sub> + 1
escribe(A,x <sub>i</sub> )
lee(B,x <sub>j</sub> )
x <sub>j</sub> := x <sub>j</sub> + 1
escribe(B,x <sub>j</sub> )

T2
lee(A,z <sub>i</sub> )
z <sub>i</sub> := z <sub>i</sub> * 2
escribe(A,z <sub>i</sub> )
lee(B,z <sub>j</sub> )
z <sub>j</sub> := z <sub>j</sub> * 2
escribe(B,z <sub>j</sub> )

- Plan de ejecución 1:
  - ¿es correcto?
  - ¿Cuáles serán los valores finales en la BD para A y B?

T1	T2
lee(A,x <sub>i</sub> )	
x <sub>i</sub> := x <sub>i</sub> + 1	
escribe(A,x <sub>i</sub> )	
	lee(A,z <sub>i</sub> )
	z <sub>i</sub> := z <sub>i</sub> * 2
	escribe(A,z <sub>i</sub> )
lee(B,x <sub>j</sub> )	
x <sub>j</sub> := x <sub>j</sub> + 1	
escribe(B,x <sub>j</sub> )	
	lee(B,z <sub>j</sub> )
	z <sub>j</sub> := z <sub>j</sub> * 2
	escribe(B,z <sub>j</sub> )

- Plan de ejecución 1:
  - correcto
  - $A = (A+1) * 2$   
 $B = (B+1) * 2$
  - Si ejecutamos primero T1 completa y luego T2 completa, nos queda lo mismo: es serializable.

T1	T2
lee(A,x <sub>i</sub> )	
x <sub>i</sub> := x <sub>i</sub> + 1	
escribe(A,x <sub>i</sub> )	
	lee(A,z <sub>i</sub> )
	z <sub>i</sub> := z <sub>i</sub> * 2
	escribe(A,z <sub>i</sub> )
lee(B,x <sub>j</sub> )	
x <sub>j</sub> := x <sub>j</sub> + 1	
escribe(B,x <sub>j</sub> )	
	lee(B,z <sub>j</sub> )
	z <sub>j</sub> := z <sub>j</sub> * 2
	escribe(B,z <sub>j</sub> )

- Plan de ejecución 2:
  - ¿es correcto?
  - ¿Cuáles serán los valores finales en la BD para A y B?

T1	T2
	lee(A,z <sub>i</sub> )
	z <sub>i</sub> := z <sub>i</sub> * 2
lee(A,x <sub>i</sub> )	
x <sub>i</sub> := x <sub>i</sub> + 1	
	escribe(A,z <sub>i</sub> )
	lee(B,z <sub>j</sub> )
	z <sub>j</sub> := z <sub>j</sub> * 2
escribe(A,x <sub>i</sub> )	
lee(B,x <sub>j</sub> )	
x <sub>j</sub> := x <sub>j</sub> + 1	
escribe(B,x <sub>j</sub> )	
	escribe(B,z <sub>j</sub> )

- Plan de ejecución 2:
  - incorrecto
  - $A = A + 1$   
 $B = B * 2$
  - Ejecución no serializable.

T1	T2
	lee(A,z <sub>i</sub> )
	z <sub>i</sub> := z <sub>i</sub> * 2
lee(A,x <sub>i</sub> )	
x <sub>i</sub> := x <sub>i</sub> + 1	
	escribe(A,z <sub>i</sub> )
	lee(B,z <sub>j</sub> )
	z <sub>j</sub> := z <sub>j</sub> * 2
escribe(A,x <sub>i</sub> )	
lee(B,x <sub>j</sub> )	
x <sub>j</sub> := x <sub>j</sub> + 1	
escribe(B,x <sub>j</sub> )	
	escribe(B,z <sub>j</sub> )

- Si el resultado de la ejecución de un conjunto de transacciones concurrentes coincide con la ejecución secuencial de las transacciones, se dice que esa ejecución es **serializable**.
- El número de ejecuciones posibles crece con el número de transacciones y no se pueden explorar todas. Hay ciertas combinaciones que permiten determinar si la ejecución es correcta o no.

- Conjunto de sentencias que actúan sobre un átomo o variable concretos.
- Dos operaciones que no modifican el átomo y que pertenecen a dos transacciones distintas pueden ejecutarse simultáneamente. Es decir, las operaciones de “sólo lectura” pueden intercarse.



- $O_i$  y  $O_j$  son compatibles si toda ejecución simultánea de ambas da el mismo resultado que la ejecución secuencial de ambas en cualquier orden.

$O_1$
lee(A, $x_1$ )
$x_1 := x_1 + 1$
imprime( $x_1$ )

$O_2$
lee(A, $x_2$ )
$x_2 := x_2 * 2$
imprime( $x_2$ )

¿Compatibles o incompatibles?

$O_1$
lee(A, $x_1$ )
$x_1 := x_1 + 1$
imprime( $x_1$ )

$O_2$
lee(A, $x_2$ )
$x_2 := x_2 * 2$
imprime( $x_2$ )

Compatibles

$O_1$
lee(A, $x_1$ )
$x_1 := x_1 + 1$
escribe(A, $x_1$ )

$O_2$
lee(A, $x_2$ )
$x_2 := x_2 * 2$
escribe(A, $x_2$ )

¿Compatibles o incompatibles?

$O_1$
lee(A,x <sub>1</sub> )
x <sub>1</sub> := x <sub>1</sub> + 1
escribe(A,x <sub>1</sub> )

$O_2$
lee(A,x <sub>2</sub> )
x <sub>2</sub> := x <sub>2</sub> * 2
escribe(A,x <sub>2</sub> )

Incompatibles

$O_1$	$O_2$
lee(A,x <sub>1</sub> )	
	lee(A,x <sub>2</sub> )
x <sub>1</sub> := x <sub>1</sub> + 1	
	x <sub>2</sub> := x <sub>2</sub> * 2
escribe(A,x <sub>1</sub> )	
	escribe(A,x <sub>2</sub> )

- $O_i$  y  $O_j$  son permutables si la ejecución de  $O_j$  tras  $O_i$  da el mismo resultado que la ejecución de  $O_i$  tras  $O_j$ .

$O_1$
lee(A, $x_1$ )
$x_1 := x_1 + 1$
escribe(A, $x_1$ )

$O_2$
lee(A, $x_2$ )
$x_2 := x_2 + 10$
escribe(A, $x_2$ )

Son permutables

$O_1$
lee(A, $x_1$ )
$x_1 := x_1 + 1$
escribe(A, $x_1$ )

$O_2$
lee(A, $x_2$ )
$x_2 := x_2 + 10$
escribe(A, $x_2$ )

Son permutables

Pero ¿son compatibles?



$O_1$
lee(A,x <sub>1</sub> )
x <sub>1</sub> := x <sub>1</sub> + 1
escribe(A,x <sub>1</sub> )

$O_2$
lee(A,x <sub>2</sub> )
x <sub>2</sub> := x <sub>2</sub> + 10
escribe(A,x <sub>2</sub> )

$O_1$	$O_2$	A
lee(A,x <sub>1</sub> )		6
	lee(A,x <sub>2</sub> )	6
x <sub>1</sub> := x <sub>1</sub> + 1		7
	x <sub>2</sub> := x <sub>2</sub> + 10	16
escribe(A,x <sub>1</sub> )		7
	escribe(A,x <sub>2</sub> )	16

¡Incompatibles!

- Prueba con la permutabilidad de...

$T_1$
lee(A, $x_1$ )
lee(B, $x_2$ )
$x_1 := x_1 * 2 + x_2$
escribe(A, $x_1$ )

$T_2$
lee(A, $x_1$ )
$x_1 := x_1 * x_1$
escribe(B, $x_1$ )

- ¿Y la compatibilidad?

$T_1$
lee(A, $x_1$ )
lee(B, $x_2$ )
$x_1 := x_1 * 2 + x_2$
escribe(A, $x_1$ )

$T_2$
lee(A, $x_1$ )
$x_1 := x_1 * x_1$
escribe(B, $x_1$ )

- Considera la ejecución

$T_1$	$T_2$
lee( $A, x_1$ )	
lee( $B, x_2$ )	lee( $A, x_1$ )
$x_1 := x_1 * 2 + x_2$	$x_1 := x_1 * x_1$
escribe( $A, x_1$ )	escribe( $B, x_1$ )

$T_1$	$T_2$
lee( $A, x_1$ )	
	lee( $A, x_1$ )
lee( $B, x_2$ )	
	$x_1 := x_1 * x_1$
$x_1 := x_1 * 2 + x_2$	
escribe( $A, x_1$ )	
	escribe( $B, x_1$ )

- Transformaciones sobre una ejecución:
  - Separación de operaciones compatibles: dadas dos operaciones compatibles entrelazadas en transacciones distintas, se cambian por una secuencia de operaciones que den el mismo resultado.
  - Re-ordenación de operaciones permutables: se cambia el orden de ejecución de operaciones permutables.

- Teorema:
  - Una condición suficiente para una ejecución serializable es que pueda ser transformada por separación y permutación en una sucesión de transacciones.

- Supongamos:

$T_1$	$T_2$
lee( $A, x_1$ )	
$x_1 := x_1 * 10$	
escribe( $A, x_1$ )	
	lee( $A, x_3$ )
lee( $B, x_2$ )	
	$x_3 := x_3 * 5$
...	...

$T_1$	$T_2$
...	...
	escribe( $A, x_3$ )
$x_2 := x_2 * 3$	
escribe( $B, x_2$ )	
	lee( $B, x_4$ )
	$x_4 := x_4 * 3$
	escribe( $B, x_4$ )

- Separación de operaciones compatibles:

$T_1$	$T_2$
lee(A, $x_1$ )	
$x_1 := x_1 * 10$	
escribe(A, $x_1$ )	
	lee(A, $x_3$ )
	$x_3 := x_3 * 5$
	escribe(A, $x_3$ )
...	...

$T_1$	$T_2$
...	...
lee(B, $x_2$ )	
$x_2 := x_2 * 3$	
escribe(B, $x_2$ )	
	lee(B, $x_4$ )
	$x_4 := x_4 * 3$
	escribe(B, $x_4$ )



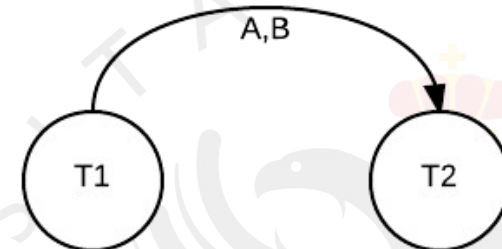
- Permutación de operaciones permutables:

$T_1$	$T_2$
lee(A,x <sub>1</sub> )	
$x_1 := x_1 * 10$	
escribe(A,x <sub>1</sub> )	
lee(B,x <sub>2</sub> )	
$x_2 := x_2 * 3$	
escribe(B,x <sub>2</sub> )	
...	...

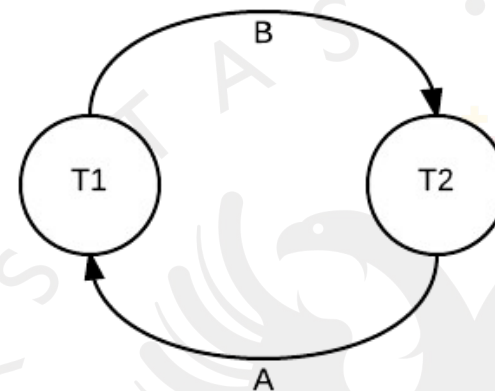
$T_1$	$T_2$
...	...
	lee(A,x <sub>3</sub> )
	$x_3 := x_3 * 5$
	escribe(A,x <sub>3</sub> )
	lee(B,x <sub>4</sub> )
	$x_4 := x_4 * 3$
	escribe(B,x <sub>4</sub> )

- Nodos: transacciones
- Arcos: restricciones en la ejecución
- $T_i$  precede a  $T_j$  si, y sólo si, existen dos operaciones no permutables sobre el mismo átomo en las dos transacciones y la operación de  $T_i$  es anterior a la de  $T_j$
- Habrá un arco entre dos transacciones si una precede a otra, y se marca el arco con el nombre del átomo implicado.

T1	T2
lee(A,x <sub>i</sub> )	
x <sub>i</sub> := x <sub>i</sub> + 1	
escribe(A,x <sub>i</sub> )	
	lee(A,z <sub>i</sub> )
	z <sub>i</sub> := z <sub>i</sub> * 2
	escribe(A,z <sub>i</sub> )
lee(B,x <sub>j</sub> )	
x <sub>j</sub> := x <sub>j</sub> + 1	
escribe(B,x <sub>j</sub> )	
	lee(B,z <sub>j</sub> )
	z <sub>j</sub> := z <sub>j</sub> * 2
	escribe(B,z <sub>j</sub> )



T1	T2
	lee(A,zi)
	zi := zi * 2
lee(A,xi)	
xi:=xi+1	
	escribe(A,zi)
escribe(A,xi)	
lee(B,xj)	
	lee(B,z <sub>j</sub> )
	zj := zj * 2
xj:=xj+1	
escribe(B,xj)	
	escribe(B,zj)



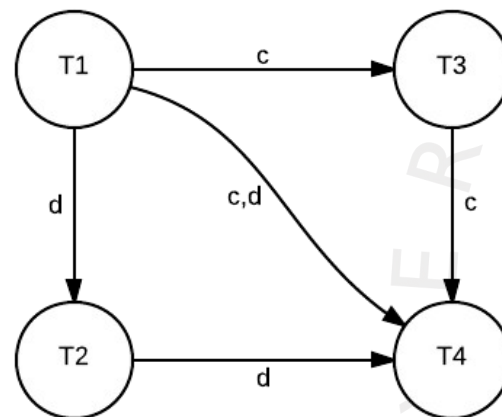
- Teorema:
  - Condición suficiente para que una ejecución sea serializable es que el grafo de dependencias no presente ciclos dirigidos.

- Un ejemplo:

lee( $T_1, c$ ), lee( $T_2, b$ ), lee( $T_1, d$ ), escribe( $T_1, c$ ),  
 lee( $T_3, c$ ), escribe( $T_3, c$ ), lee( $T_2, d$ ), lee( $T_3, a$ ),  
 lee( $T_4, c$ ), escribe( $T_2, d$ ), escribe( $T_2, b$ ), lee( $T_4, d$ ),  
 escribe( $T_4, d$ )

- Un ejemplo:

$\text{lee}(T_1, c)$ ,  $\text{lee}(T_2, b)$ ,  $\text{lee}(T_1, d)$ ,  $\text{escribe}(T_1, c)$ ,  $\text{lee}(T_3, c)$ ,  
 $\text{escribe}(T_3, c)$ ,  $\text{lee}(T_2, d)$ ,  $\text{lee}(T_3, a)$ ,  $\text{lee}(T_4, c)$ ,  $\text{escribe}(T_2, d)$ ,  
 $\text{escribe}(T_2, b)$ ,  $\text{lee}(T_4, d)$ ,  $\text{escribe}(T_4, d)$



- Técnicas para garantizar el aislamiento de las transacciones:
  - Permitir la ejecución y deshacer las que produzcan conflicto: *técnicas de ordenación por marcas de tiempo.*
  - Evitar ciclos mediante esperas: *técnicas de bloqueo.*



- Cada transacción recibe una marca de tiempo única cuando comienza.
- A cada átomo  $x$ , se asocia una referencia a la última transacción que opera sobre él:  $R(x)$
- Algunos algoritmos de este tipo son:
  - Ordenación total
  - Ordenación parcial
  - Ordenación parcial multi-versión
  - Control de concurrencia mediante validación

- Sean dos transacciones  $T_i$  y  $T_j$  con  $i$  y  $j$  marcas de tiempo tales que  $i < j$ , el algoritmo garantiza que  $T_i$  accede antes que  $T_j$ .
- Si una operación falla (ABORT), se deshace la transacción y se re-lanza más tarde, asignándole a la retrasada una referencia mayor que la de todas las actuales.
- **Problema:** las lecturas concurrentes también se ordenan, sin ser necesario por no ser conflictivas.

Procedimiento lee ( $T_i$ ,  $a$ );

INICIO

Si ( $R(a) \leq i$ ) entonces

{Ejecutar la lectura}

$R(a) := i$ ;

si-no

{ABORT: abortar la ejecucion}

Fin-si

FIN

Procedimiento escribe ( $T_i$ ,  $a$ );

INICIO

Si ( $R(a) \leq i$ ) entonces

{Ejecutar la escritura}

$R(a) := i$ ;

si-no

{ABORT: abortar la ejecucion}

Fin-si

FIN

- Sólo se ordenan las parejas de operaciones lee/escribe, escribe/lee y escribe/escribe.
- Cada átomo  $x$  tiene dos referencias: una para determinar la última transferencia que lo leyó  $RR(x)$  y otra para la que lo actualizó  $WR(x)$ .
- Si una operación falla (ABORT), se deshace la transacción y se re-lanza más tarde, asignándole a la retrasada una referencia mayor que la de todas las actuales.

Procedimiento lee ( $T_i, a$ ); { \* escribe/lee \* }

INICIO

Si ( $WR(a) \leq i$ ) entonces

{Ejecutar la lectura}

$RR(a) := \text{Max}(RR(a), i)$ ;

si-no

{ABORT: abortar la ejecucion}

Fin-si

FIN

Procedimiento escribe ( $T_i, a$ ); { \* lee/escribe y escribe/escribe \* }

INICIO

Si ( $RR(a) \leq i$ ) y ( $WR(a) \leq i$ ) entonces

{Ejecutar la escritura}

$WR(a) := i$ ;

si-no

{ABORT: abortar la ejecucion}

Fin-si

FIN

**Problema:** hace trabajo en vano cuando  $RR(a) \leq i < WR(a)$ , porque escribe información obsoleta.

**Alternativa:**

```

Procedimiento escribe (Ti, a); { * lee/escribe y escribe/escribe * }
  INICIO
    Si (RR(a) <= i) entonces
      Si (WR(a) <= i) entonces
        {Ejecutar la escritura}
        WR(a) := i;
      Fin-si
    si-no
      {ABORT: abortar la ejecucion}
    Fin-si
  FIN
  
```

Si la última que escribió es posterior, se ignora la operación de escritura.

- Persigue que las lecturas no aborten una transacción, pero implica que haya distintas versiones de cada átomo.
- Sólo habrá que buscar la última versión escrita con referencia menor que la de la transacción en curso.

Procedimiento lee ( $T_i$ ,  $a$ );

INICIO

$j := \{\text{ultima version de } a\};$

Repita mientras ( $WR_j(a) > i$ )

$j := j - 1;$

Fin-repita

{Ejecutar la lectura de la version  $j$ }

$RR_j(a) := \text{Max}(RR(a), i);$

FIN



**Procedimiento escribe ( $T_i$ ,  $a$ );**

**INICIO**

$j := \{\text{ultima version de } a\};$

Repite mientras ( $RR_j(a) > i$ )

$j := j - 1;$

Fin-repite

Si ( $WR_j(a) > i$ ) entonces --Solo si 0 en lectura

{ABORT: Abortar ejecucion}

si-no

{Ejecutar la escritura insertando una versión ( $j+1$ ) de  $a$ }

$WR_{j+1}(a) := i;$

Fin-si

**FIN**

- Las transacciones se dividen en tres fases:
  - *Lectura*: donde se leen átomos, realizan cálculos y actualizan variables,
  - *Validación*: se comprueba la validez de los datos,
  - *Escritura*: se vuelcan los átomos al buffer.
- Para cada transacción, se guardan las marcas de tiempo para inicio, validación y fin.

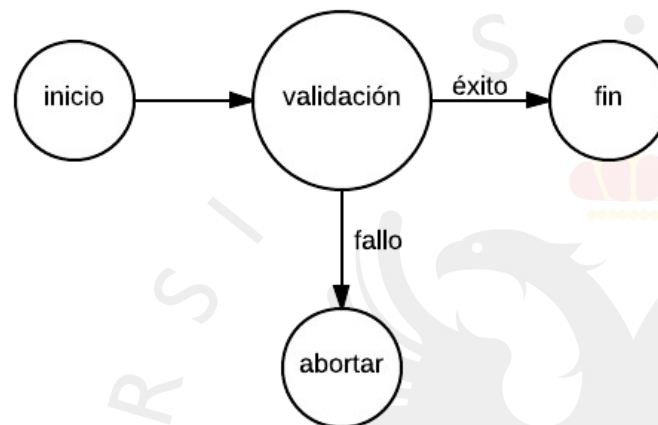
$T_1$
lee(A,x1)
lee(B,x2)
$x1 := x1 * 2 + x2$
escribe(A,x1)

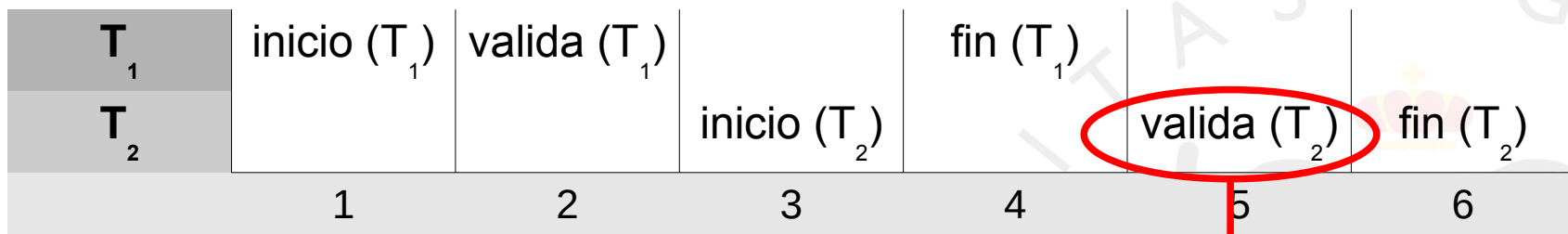
## Momentos

inicio ( $T_1$ )

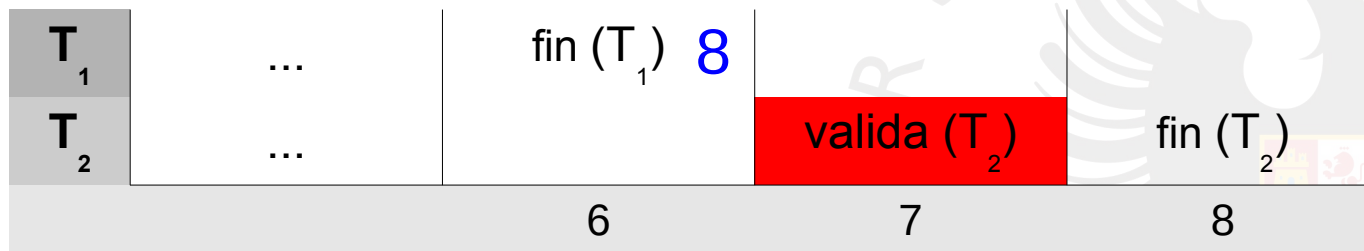
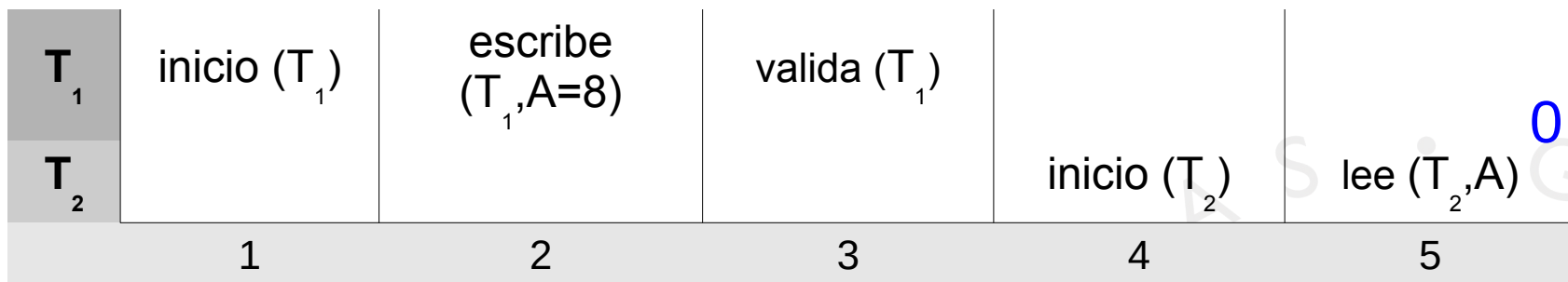
valida ( $T_1$ )

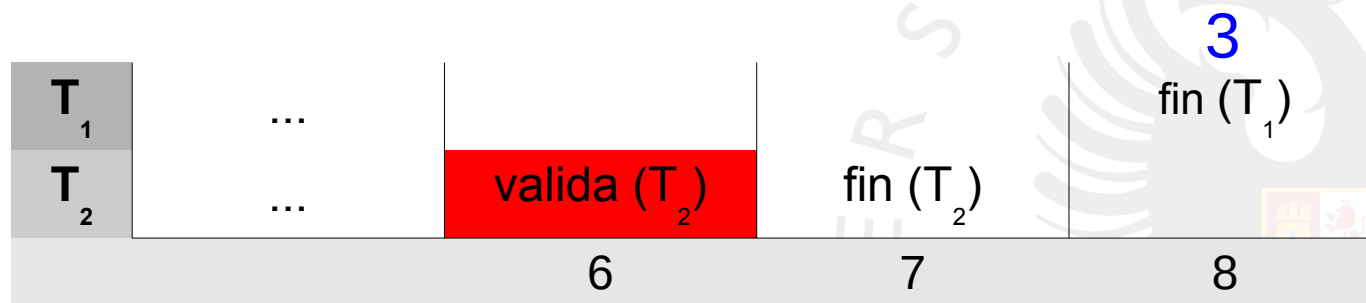
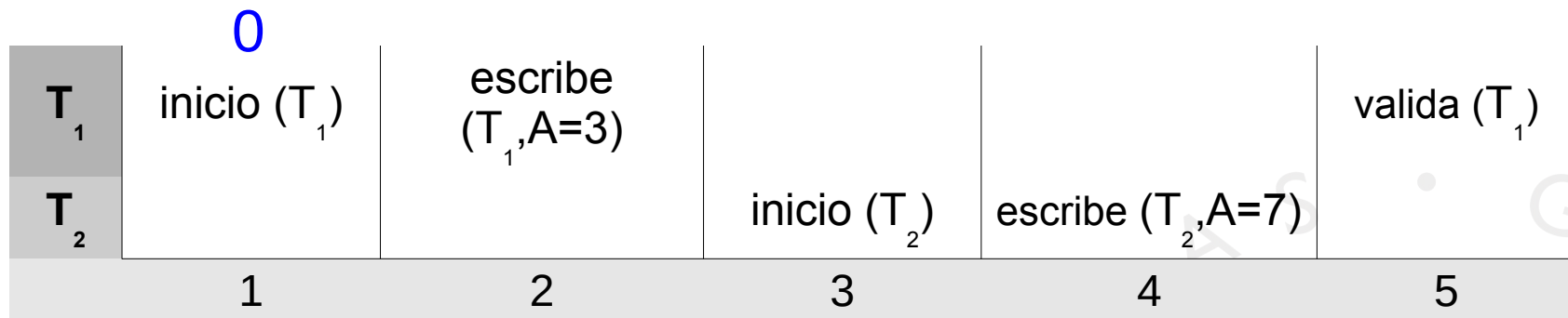
fin ( $T_1$ )





Se comprueba si los valores que usa  $T_2$  no han cambiado desde que se inició la transacción





Procedimiento COMPROBAR (i);

INICIO

Para cada j tal que (inicio(i) < fin(j))

/\* Tj activas con marcas inicio o válida /

Si (AR(i)  $\cap$  AW(j)  $\neq \emptyset$ ) entonces

{ABORT} /\* Abortar la ejecucion \*/

Fin-si

Fin-para

Para cada j tal que (fin(j) < valida(i))

/\* Tj activas con marca válida /

Si (AW(i)  $\cap$  AW(j)  $\neq \emptyset$ ) entonces

{ABORT} /\* Abortar la ejecucion de Ti \*/

Fin-si

Fin-para

FIN

Procedimiento COMPROBAR (i);

INICIO

Para cada j tal que (inicio(i) < fin(j))

/\* Tj activas con marcas inicio o válida /

Si (AR(i)  $\cap$  AW(j)  $\neq \emptyset$ ) entonces

{ABORT} /\* Abortar la ejecucion \*/

Fin-si

Fin-para

Para cada j tal que (fin(j) < valida(i))

/\* Tj activas con marca válida /

Si (AW(i)  $\cap$  AW(j)  $\neq \emptyset$ ) entonces

{ABORT} /\* Abortar la ejecucion de Ti \*/

Fin-si

Fin-para

FIN

Transacciones

Átomos leídos

Átomos escritos



Plan	Etapas
lee ( $T_1$ , A)	inicio ( $T_1$ )
escribe ( $T_1$ , A)	
lee ( $T_2$ , B)	inicio ( $T_2$ )
escribe ( $T_1$ , C)	
	valida ( $T_1$ )
	fin ( $T_1$ )
lee ( $T_3$ , B)	inicio ( $T_3$ )
escribe ( $T_3$ , A)	valida ( $T_2$ )
	valida ( $T_3$ )
	fin ( $T_2$ )



- Lo que hemos visto, planifica el retraso de transacciones que ejecutan accesos conflictivos y la repetición de operaciones ya ejecutadas.
- Estas técnicas detectan ejecuciones no serializables.

- Evitan ejecuciones incorrectas manteniendo en espera las transacciones con operaciones conflictivas sobre el mismo átomo.
- Permiten la ejecución simultánea de operaciones compatibles en base a su *modo*.
- **Modo de operación:**
  - Modos clásicos: lectura, actualización/escritura
  - Carácter: exclusivo, protegido
  - Propuestas CODASYL: *consulta no protegida M1 y consulta protegida M2, actualización no protegida M3 y actualización protegida M4, consulta exclusiva M5 y actualización exclusiva M6*

		Consulta		Actualización		Exclusiva	
		Prot.	No Prot.	Prot.	No Prot.	Consulta	Actualiz.
Consulta	Prot.	1	1	1	1	0	0
	No Prot.	1	1	0	0	0	0
Actualiz.	Prot.	1	0	1	0	0	0
	No Prot.	1	0	0	0	0	0
Exclus.	Consulta	0	0	0	0	0	0
	Actualiz.	0	0	0	0	0	0

- El controlador sólo permite la ejecución de operaciones compatibles.
- Los *protocolos de bloqueo* (mecanismos) se basan en dos operaciones:
  - LOCK ( $a, M$ ): bloqueo del átomo  $a$  en modo  $M$ , y
  - UNLOCK ( $a$ ): desbloqueo del átomo  $a$ .

- Se suele usar:
  - Un controlador por cada átomo
  - Un vector de 6 bits para cada átomo y transacción  $A(a,i)$  que almacena el acceso de la transacción  $i$  al átomo  $a$ .
  - Un vector de 6 bits por cada modo (seis vectores en total), con un 1 en la correspondiente posición.

- Una operación de bloqueo  $LOCK(a, M)$  es compatible con los modos actuales de ejecución sii

$$M \subset \neg(\neg C \times (\cup_{i \neq p} A(a, i)))$$

- Una operación de bloqueo  $LOCK(a, M)$  es compatible con los modos actuales de ejecución sii

Unión de los vectores actuales sobre el átomo  $a$

$$M \subset \neg(\neg C \times (\cup_{i \neq p} A(a, i)))$$

Matriz de **incompatibilidades**



- Una operación de bloqueo  $LOCK(a, M)$  es compatible con los modos actuales de ejecución sii

$$M \subset \neg(\neg C \times (\cup_{i \neq p} A(a, i)))$$

Vector de modos **incompatibles** con las  
con las ejecuciones actuales

- Una operación de bloqueo  $LOCK(a, M)$  es compatible con los modos actuales de ejecución sii

$$M \subset \neg(\neg C \times (\cup_{i \neq p} A(a, i)))$$

Vector de modos **compatibles** con las  
con las ejecuciones actuales

**Procedimiento LOCK (a, M);**

**INICIO**

**Si**  $M \subset \neg(\neg C \times (\cup_{i \neq p} A(a, i)))$  **entonces**

$A(a, p) := A(a, p) \cup M;$

**si-no**

**Insertar** (p, M) **en** Q(a);

**Bloquear** p;

**Fin-si**

**FIN**

Transacción

Cola de transacciones bloqueadas y modos por operación incompatible sobre el átomo a

Transacción y modo de acceso a un átomo

Procedimiento UNLOCK (a);

INICIO

$A(a, p) := 0;$

Para cada  $(q, M')$  de  $Q(a)$  hacer

Si  $M \subset \neg(\neg C \times (\cup_{i \neq p} A(a, i)))$  entonces

$A(a, q) := A(a, p) \cup M';$

Extraer  $(q, M')$  de  $Q(a);$

Desbloquear  $q;$

Fin-si

Fin-para

FIN

Transacción

Cola de transacciones bloqueadas y modos por operación incompatible sobre el átomo a

Transacción y modo de acceso a un átomo

- ¿Podría ejecutarse una operación con modo M5 si se están ejecutando una con modo M1 y otra con modo M3?

0	0	0	0	1	1
0	0	1	1	1	1
0	1	0	1	1	1
0	1	1	1	1	1
1	1	1	1	1	1
1	1	1	1	1	1

 $\times$ 

1
0
1
0
0
0

 $=$ 

0	1	0	1	1	1
---	---	---	---	---	---

- ¿Podría ejecutarse una operación con modo M5 si se están ejecutando una con modo M1 y otra con modo M3?

Modos  
compatibles

1	0	1	0	0	0
---	---	---	---	---	---

0	0	0	0	1	0
---	---	---	---	---	---

≠

1	0	1	0	0	0
---	---	---	---	---	---

¡No puede!

- Permiten la ejecución simultánea de operaciones compatibles sobre un átomo.
- Una transacción sólo se ejecuta si consigue el bloqueo de todos sus átomos.

- Antes de acceder a un átomo, hay que bloquearlo.
- **Transacción en dos fases:** aquella que no ejecuta LOCK despues de un UNLOCK (primero los LOCKs y luego los UNLOCKs).
- Cuando una transacción alcanza el *bloqueo máximo*, comienza la ejecución de sus operaciones.
- El orden de ejecución de transacciones lo establece el instante en el que alcanzan el estado de bloqueo máximo.



Toda ejecución completa de un conjunto de transacciones de dos fases es serializable.

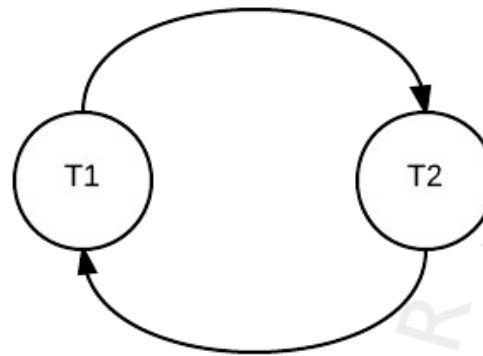
- Para que la ejecución concurrente sea serializable, toda transacción debe cumplir:
  - que las operaciones LOCK se hagan con el modo correcto y sobre el átomo necesario
  - que la operaciones UNLOCK sobre el átomo se realice una vez terminadas todas las operaciones sobre dicho átomo
  - que no se haga una operación LOCK después de ninguna operación UNLOCK

- Se da un bloqueo mortal (o *deadlock*) cuando:
  - un grupo de transacciones están a la espera de que otras desbloqueen un átomo
  - la ejecución de transacciones no bloqueadas no desbloquea ningún átomo requerido por ninguna de las transacciones bloqueadas
- Posibles tratamientos de esta situación:
  - Prevenir o
  - Detectar

- **Grafos de bloqueo:** grafos en los que los nodos son transacciones que acceden a átomos y los arcos determinan la relación de espera.

Se produce un bloqueo mortal si y sólo si el grafo de bloqueo contiene un ciclo

- Ejemplo de grafo de bloqueo:



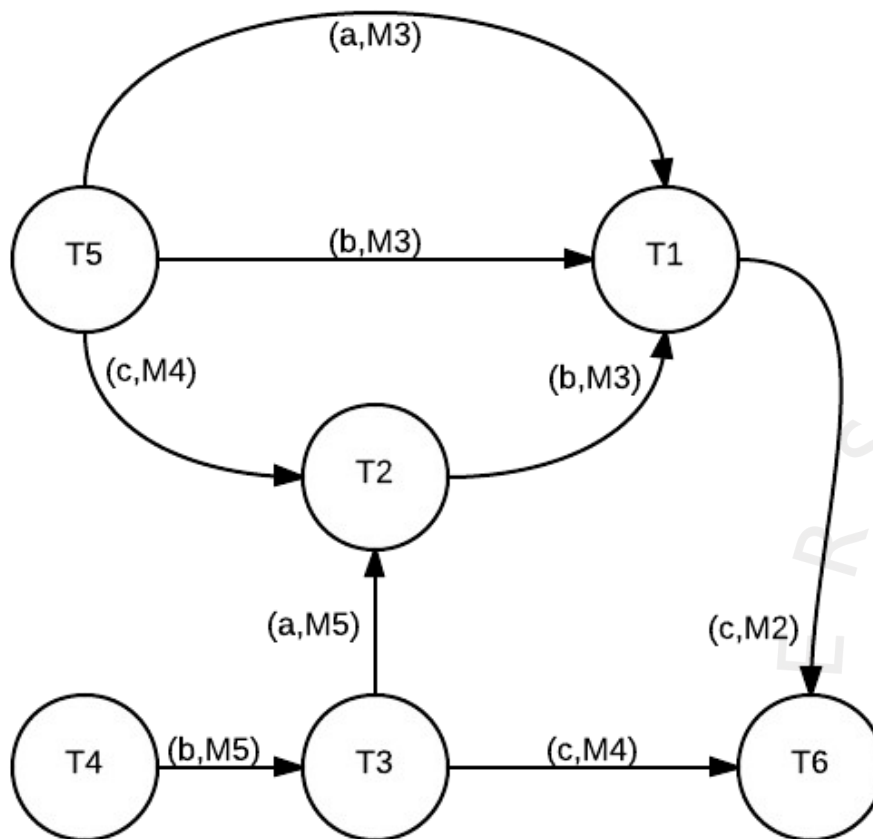
T1 espera a T2 (arco de T1 a T2) y T2 espera a T1 (arco de T2 a T1)

- Prevención:
  - Ordenar las transacciones, de modo que no puede haber una transacción *antigua* esperando por una *nueva*
    - *DIE-WAIT*: una transacción sólo espera a otra si esta otra es más joven; si no, se repite desde el principio.
    - *WOUND-WAIT*: una transacción espera a otra más vieja; si no, mata a una transacción más joven

Problema: aborta muchas transacciones no conflictivas realmente.

- Detección:
  - Explorar el grafo para encontrar ciclos.
  - Si existe un ciclo, matar una transacción del ciclo:
    - la que bloquea el átomo más solicitado, o
    - la que bloquea el mayor número de átomos.

Supongamos seis transacciones y tres átomos


 $N(1)=1$ 
 $N(2)=1$ 
 $N(3)=2$ 
 $N(4)=1$ 
 $N(5)=3$ 
 $N(6)=0$ 
 $Q(a)=\{(T5,M3),(T3,M5)\}$ 
 $Q(b)=\{(T4,M5),(T2,M3),$   
 $(T5,M3)\}$ 
 $Q(c)=\{(T5,M4),(T3,M4),$   
 $(T1,M2)\}$



- $N(k) = 0$  supone que el nodo está *resuelto*.
- Los nodos resueltos no provocan bloqueo y se eliminan, para re-evaluar los que quedan.
- Para aplicar este método, hay que:
  - re-marcar nodos después de borrar uno,
  - Tener una función *SLOCK* ( $a, k, j$ ) que determina si la operación  $j$  puede llevarse a cabo por la transacción  $k$  sobre el átomo  $a$  dado el estado actual del grafo.

Función DETECTAR;

INICIO

$T = \{\text{transacciones } j \text{ tales que } N(j) = 0\}$

$R = \{\text{lista de átomos de las transacciones } j \text{ incluidas en } T\}$

Repetir mientras aristas tachadas

Para cada átomo  $i$  de  $R$  hacer

Para cada par  $(T_k, M_j)$  en espera del átomo  $i$  hacer

Si  $(\text{SLOCK}(i, k, j))$  entonces

{Tachar arista  $(a_i, M_j)$  con origen en  $T_k$ };

$N(k) := N(k) - 1$ ;

Si  $(N(k) = 0)$  entonces

{Quitar  $T_k$  de  $T$ };

Fin-si;

Fin-si;

Fin-para;

Fin-para;

Fin-repetir;

Si  $(T = \emptyset)$  entonces

Detectar = falso;

si-no

Detectar = verdadero;

Fin-si;

FIN

