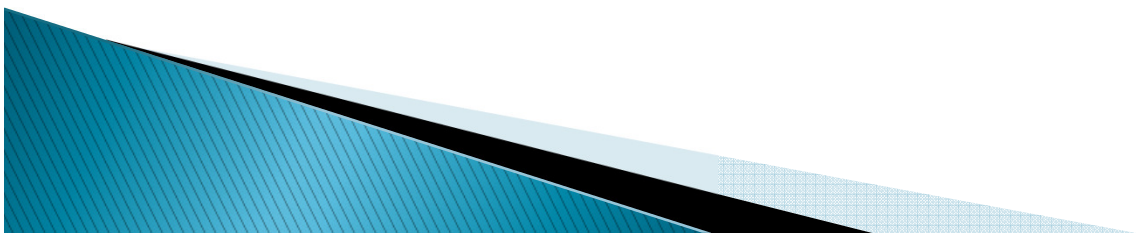
CONTROL DE CONCURRENCIA Y RECUPERACIÓN EN BASES DE DATOS

MATERIA: BASE DE DATOS

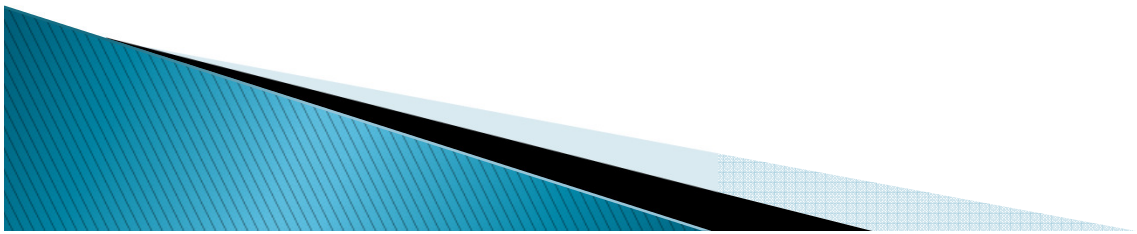
CUATRIMESTRE: 1C2015

DOCENTE: ALEJANDRO EIDELSZTEIN



CONTROL DE CONCURRENCIA Y RECUPERACIÓN

- **TRANSACCIONES**
- **PROBLEMAS DE CONCURRENCIA Y RECUPERACIÓN**
- **PROPIEDADES ACID**
- **SERIALIZABILIDAD**
- MÉTODOS PESIMISTAS:
    - LOCKING
- MÉTODOS OPTIMISTAS:
    - TIMESTAMPING
    - TIMESTAMPING MULTIVERSIÓN
    - VALIDACIÓN
- **RECUPERABILIDAD**



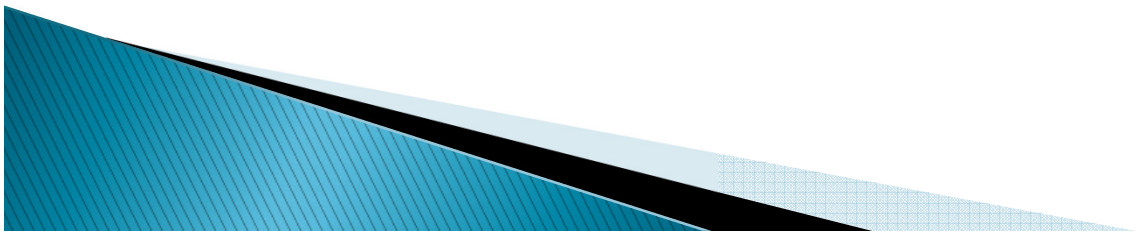
ITEMS:

Construiremos un modelo para estudiar los problemas de concurrencia en BD.

En este modelo veremos a la BD como un conjunto de *ítems*.

Un ítem puede ser un atributo, una tupla o una relación entera.

Los denominaremos con letras: A, B, X, Y, etc.

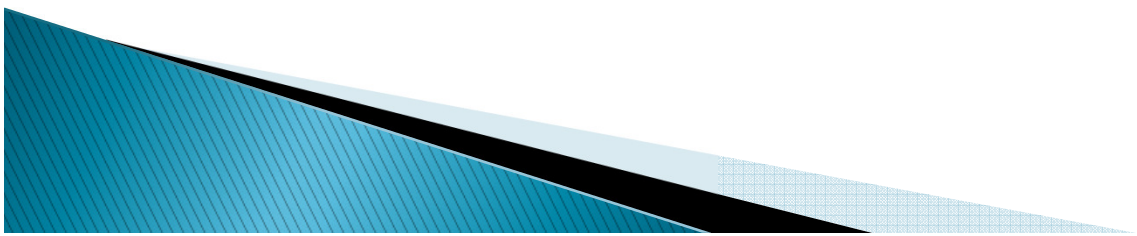


ESPACIOS DE MEMORIA:

LOCAL DE LA TRANSACCION:
Es privado de la transacción. No es visto por las otras transacciones. Es en memoria principal.

POOL DE BUFFERS:
Es público. Es en memoria principal.
Por ahora haremos abstracción de este espacio de memoria

DISCO:
Bloques de disco. Es público. Es en memoria secundaria.



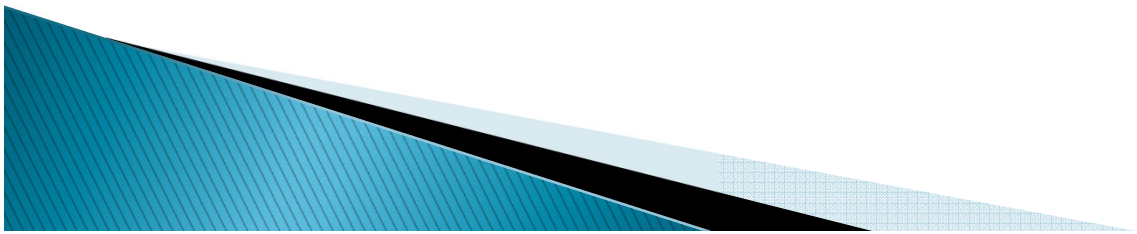
TRANSACCIONES

DEFINICIÓN DE TRANSACCIÓN:

Una *transacción T es una ejecución de un programa P que accede a la BD.*

Un mismo programa P puede ejecutarse varias veces. Cada ejecución de P es una transacción T_i.

Una transacción es una sucesión de *acciones (u operaciones)*



DEFINICIÓN DE TRANSACCIÓN (CONT.):

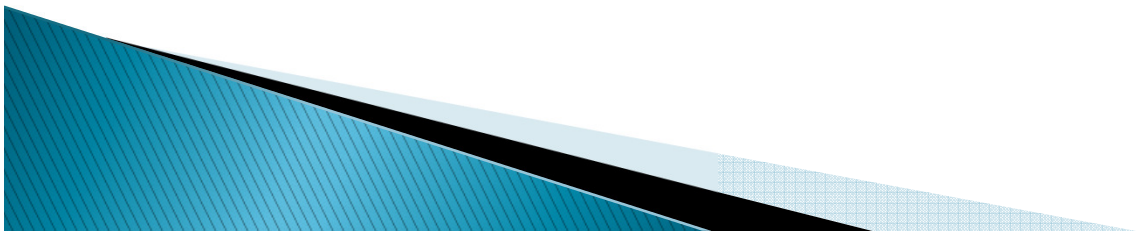
Una acción es un paso atómico. Estos pueden ser:

Leer un ítem X:	$r_i[X]$
Escribir un ítem X:	$w_i[X]$
Abort de T_i :	a_i
Commit de T_i :	c_i

$r_i[X]$ copia el valor del ítem X en disco a la variable local de la transacción.

$w_i[X]$ copia el valor de la variable local de la transacción al ítem X en disco.

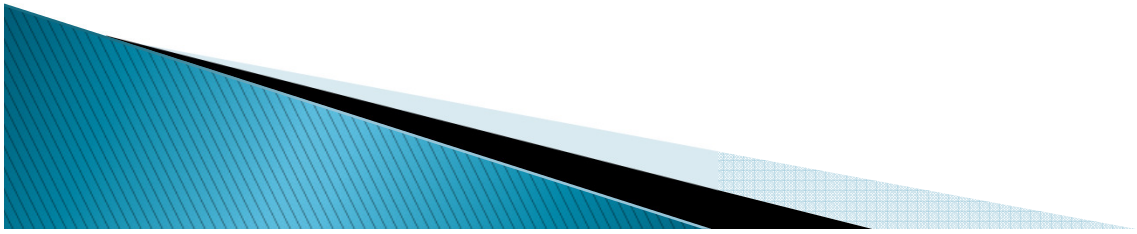
Haremos abstracción del resto de las operaciones que no sean las 4 mencionadas más arriba.



SEMÁNTICA DE LAS OPERACIONES DE FINALIZACIÓN:

ABORT:
Si la transacción finaliza con un *abort* se deben deshacer todos los cambios que ésta hizo en los items. Es un *rollback* de todas sus operaciones como si la transacción nunca hubiera existido.

COMMIT:
Si la transacción finaliza con un *commit* se debe garantizar que todos los cambios hechos por ésta en los items queden grabados en forma permanente en la base.

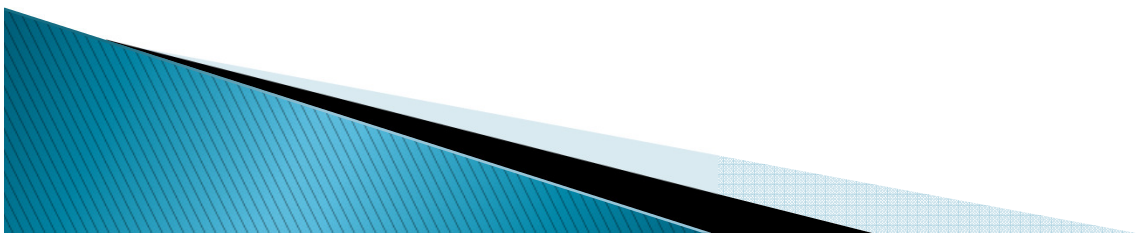


DEFINICIÓN DE TRANSACCIÓN (CONT.):

Ahora podemos definir una transacción como:

$$T_i \subseteq \{r_i[X], w_i[X] \,/\, X \text{ es un ítem}\} \cup \{a_i, c_i\}$$

A este modelo de escritura lo llamaremos *modelo read/write* o *modelo sin locking*



EJEMPLO DE TRANSACCIÓN:

Si tenemos un programa:

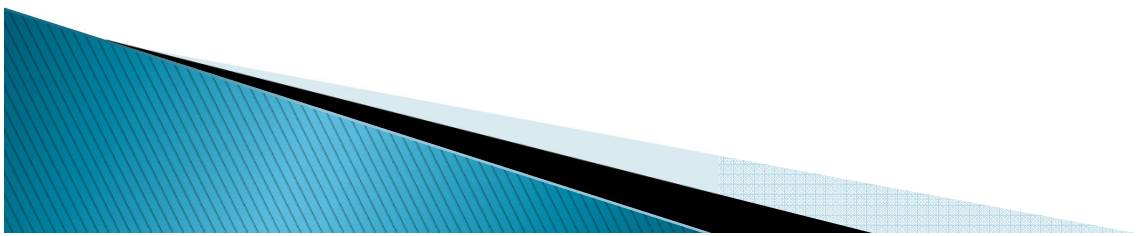
P= Read(A); A:=A+1; Write(A);

Dos transacciones T_1 y T_2 correspondientes cada una a una ejecución de P podrían ser:

$T_1 = r_1[A]\ w_1[A]\ c_1$

$T_2 = r_2[A]\ w_2[A]\ a_2$

Se puede observar que hacemos abstracción de la operación A:=A+1



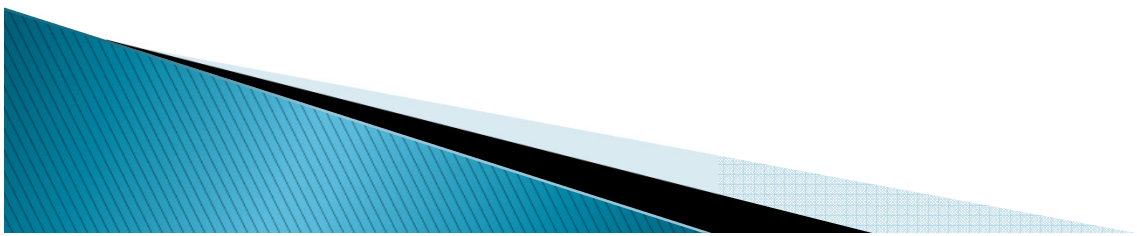
PROBLEMAS DE CONCURRENCIA Y RECUPERACIÓN:

Las Ti se ejecutan en forma concurrente (entrelazada) y esto genera el
problema de interferencia.

Asimismo, pueden ocurrir fallas en medio de la ejecución y esto genera el *problema de recuperación.*

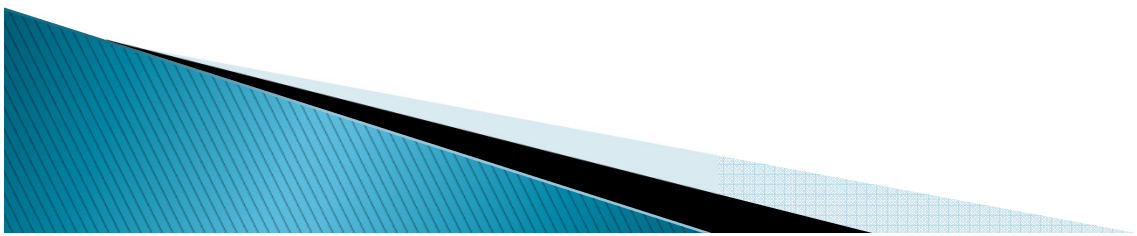
Los problemas clásicos que se pueden presentar son:

- lost update
- dirty read
- non-repeatable read
- phantom read



LOST UPDATE:

El *lost update o actualización perdida* ocurre cuando se pierde la actualización hecha por una transacción T_1 por la acción de otra transacción T_2 sobre el mismo ítem (ambas transacciones leen el mismo valor anterior del item y luego lo actualizan en forma sucesiva)

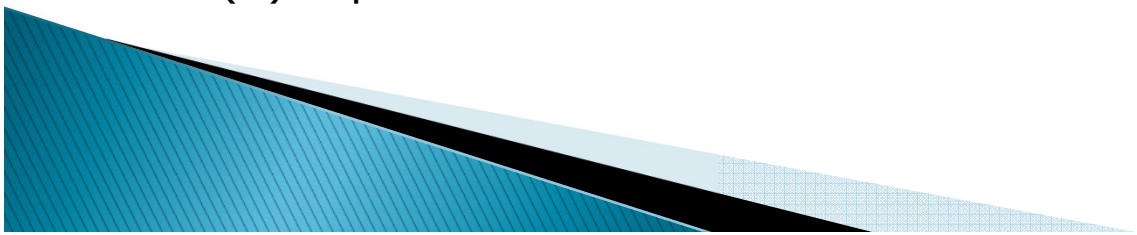


Ejemplo 1:

Supongamos el programa P= Read(A); A:=A+1; Write(A);
y dos ejecuciones de P, T_1 y T_2 sobre el ítem A, con el siguiente entrelazamiento:

T_1	T_2	A	< -- (valor del ítem A en disco, inicialmente igual a 5)
Read(A)		5	
	Read(A)	5	
A:=A+1		5	
	A:=A+1	5	
	Write(A)	6	
Write(A)		6	< -- (valor incorrecto de A, se perdió la actualización hecha por T_2)

Nota:
A:=A+1 se hace sobre la variable local de la transacción.
Read(A) copia el valor del ítem A en disco a la variable local de la transacción.
Write(A) copia el valor de la variable local de la transacción al ítem A en disco.

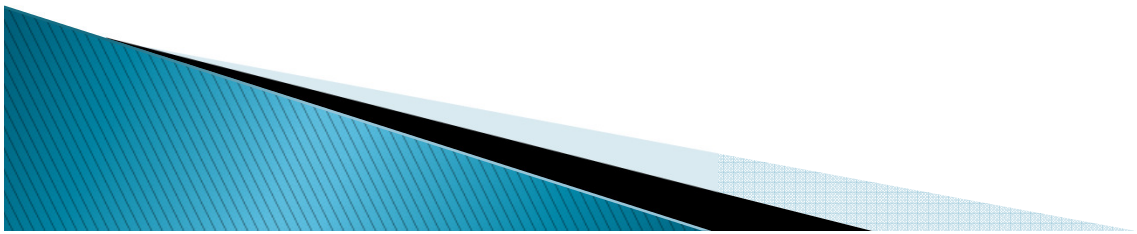


DIRTY READ:

El *dirty read o lectura sucia* ocurre cuando una transacción T_2 lee un valor de un ítem dejado por otra transacción T_1 que no hizo commit antes de que T_2 leyera el item.

Observemos que si T_1 aborta, T_2 se quedó con un valor sucio que será deshecho por el rollback de T_1.

Esto además podría producir un fenómeno no deseado como es el de *aborts en cascada* (si T_2 leyó de T_1 que abortó, deberíamos abortar T_2, luego si T_3 leyó de T_2 deberíamos abortar a su vez T_3 y así sucesivamente)



Ejemplo 2:
Supongamos los programas :

P_1= Read(A); A:=A-1; Write(A); Read(B); B:=B/A; Write(B);

P_2= Read(A); A:=A*2; Write(A);

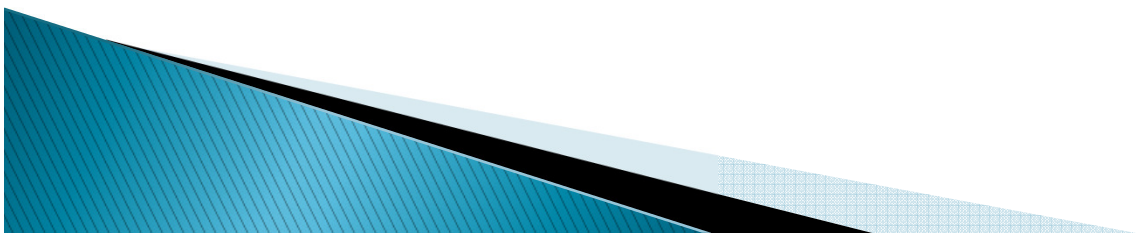
y dos ejecuciones: una T_1 de P_1 sobre los ítems A y B, y una T_2 de P_2 sobre el ítem A, con el siguiente entrelazamiento:

T_1	T_2	A	B	< -- (valor de los ítems A y B en disco, inicialmente A=1 y B=2)
Read(A)		1	2	
A:=A-1		1	2	
Write(A)		0	2	
	Read(A)	0	2	
	A:=A*2	0	2	
Read(B)		0	2	
	Write(A)	0	2	
B:=B/A		0	2	< -- (T_1 falla por la división por cero y aborta volviendo A al valor anterior, pero T_2 ya leyó A)

NON-REPEATABLE READ:

También llamado *unrepeatable read*, el non-*repeatable read* o lectura no repetible ocurre cuando una transacción T_1 lee un ítem, otra transacción T_2 lo modifica y luego T_1 vuelve a leer ese item y ve que ahora tiene otro valor.

Dicho de otra forma, es cuando durante el avance de una transacción un ítem es leído dos veces y los valores difieren en las dos lecturas.



NON-REPEATABLE READ:

Ejemplo 2.1:
Supongamos los programas:

P_1= Read(A); Read(A);
P_2= Read(A); A:=A+1; Write(A);

y dos ejecuciones T_1 de P_1 y T_2 de P_2 sobre el ítem A, con el siguiente entrelazamiento:

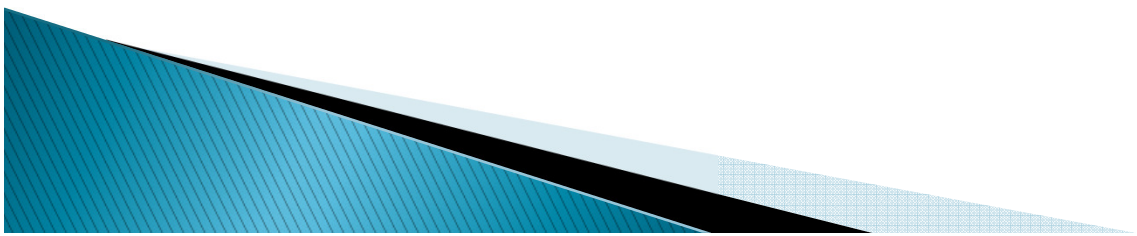
T_1	T_2	A	< -- (valor del ítem A en disco, inicialmente=5)
Read(A)		5	
	Read(A)	5	
	A:=A+1	5	
	Write(A)	6	
Read(A)		6	< -- (T_1 vuelve a leer A pero ve que ahora tiene otro valor)

PHANTOM READ:

El *phantom read* o lectura fantasma ocurre cuando una transacción T_1 ejecuta un query y lee un conjunto de ítems (generalmente tuplas), otra transacción T_2 inserta nuevos items y luego T_1 vuelve a ejecutar el mismo query y ahora el conjunto de ítems ha cambiado.

Dicho de otra forma, es cuando durante el avance de una transacción dos queries indénticos se ejecutan y el conjunto de tuplas resultante del segundo difiere del primero.

El *phantom read* podría verse como un caso particular de *non-repeatable read*



PROPIEDADES ACID:
La idea es que dada una BD en un estado consistente, luego de ejecutarse las transacciones la BD quede también en un estado consistente.

El SGBD debe garantizar esto último haciendo que las transacciones cumplan con las *propiedades* ***ACID***.

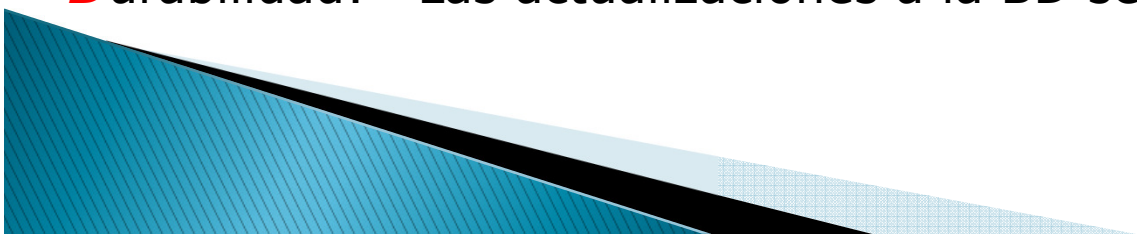
Estas propiedades son:

Atomicidad: T se ejecuta completamente o no se ejecuta por completo (todo o nada)

Consistencia: T transforma un estado consistente de la BD en otro estado consistente (los programas deben ser correctos)

A***I***slamiento: Las Ti se ejecutan sin interferencias.

Durabilidad: Las actualizaciones a la BD serán durables y públicas.



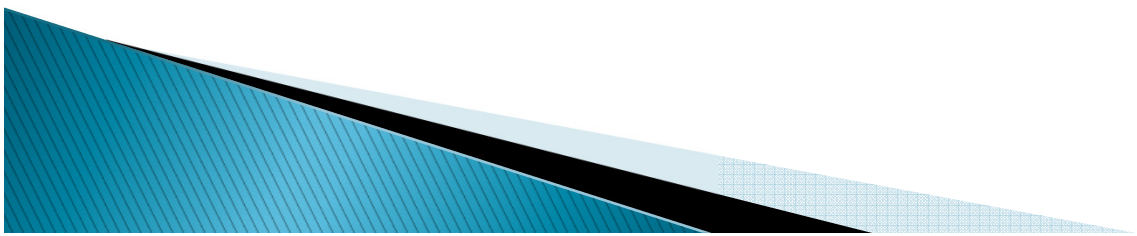
SERIALIZABILIDAD

HISTORIAS (SCHEDULES):

Si T= $\{T_1, T_2, ..., T_n\}$ es un conjunto de transacciones, entonces una *historia* (o *schedule*) H sobre T es:

$$H = U_{i=1,n}\ T_i$$

Donde H respeta el orden de las acciones de cada T_i.



Ejemplo 3:

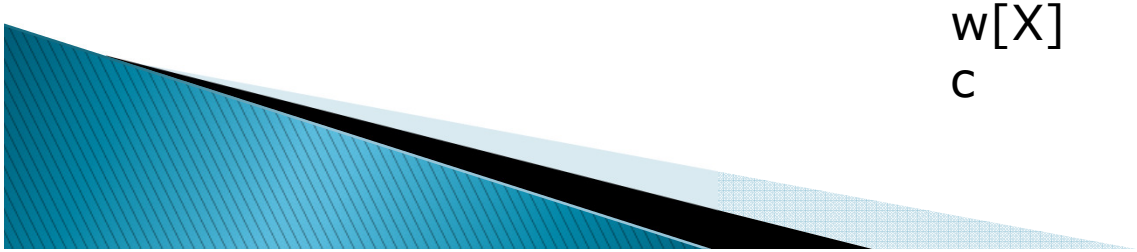
Dados $T_1 = r_1[X]\ w_1[X]\ c_1$,
$T_3 = r_3[X]\ w_3[Y]\ w_3[X]\ c_3$,

una historia H_1 sobre el conjunto de transacciones $\{T_1, T_3\}$ y el conjunto de ítems {X,Y} podría ser:

$$H_1 = r_1[X]\ r_3[X]\ w_1[X]\ c_1\ w_3[Y]\ w_3[X]\ c_3$$

También podemos expresar H1 en forma tabular:

T_1	T_3
r[X]	
	r[X]
w[X]	
c	
	w[Y]
	w[X]
	c



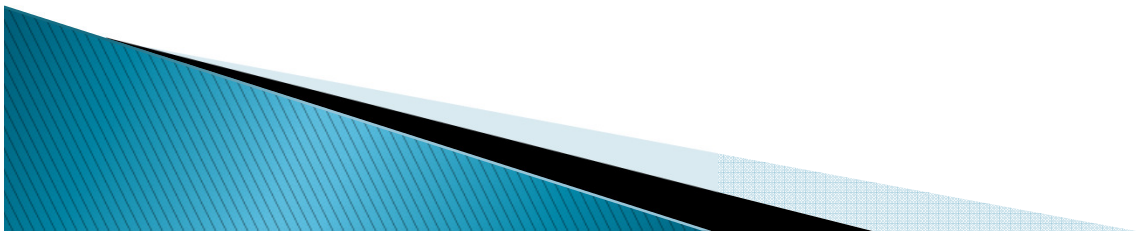
Ejemplo 3.1:

En este ejemplo todas las operaciones de T_1 preceden a las de T_2 (H_1) o viceversa (H_2)
Diremos que H_1 y H_2 son *Seriales*.

$T_1 = r_1[X]\ w_1[X]\ c_1,$
$T_3 = r_3[X]\ w_3[Y]\ w_3[X]\ c_3,$

$H_1 = r_1[X]\ w_1[X]\ c_1\ r_3[X]\ w_3[Y]\ w_3[X]\ c_3$

$H_2 = r_3[X]\ w_3[Y]\ w_3[X]\ c_3\ r_1[X]\ w_1[X]\ c_1$



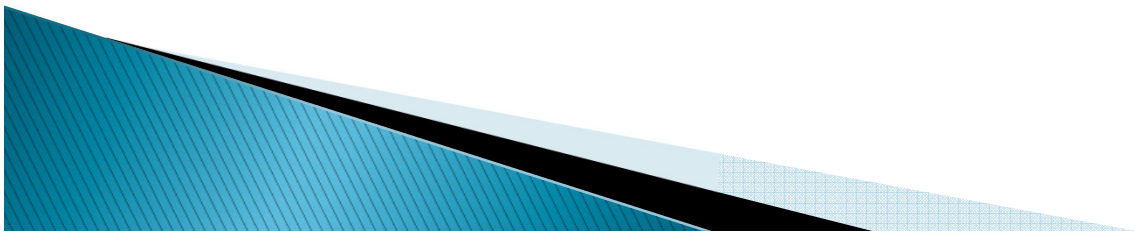
EQUIVALENCIA DE HISTORIAS:

Dos historias H y H' son *equivalentes (H≡H') si:*

1) Si están definidas sobre el mismo conjunto de transacciones.

2) Las operaciones *conflictivas tienen el mismo orden.*

Dos operaciones de T_i y T_j (i≠j) son *conflictivas si operan sobre el mismo ítem y al menos alguna de las dos es un write.*



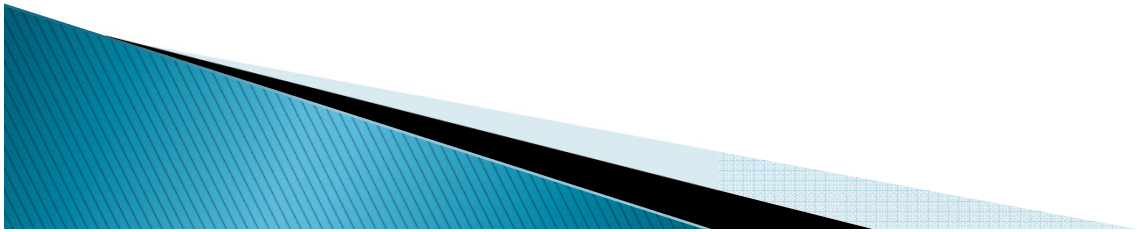
HISTORIAS SERIALES:

H es *serial* *(H_s) si para todo par de transacciones T_i y T_j en H, todas las operaciones de T_i preceden a las de T_j o viceversa.*

Las historias seriales (y las equivalentes a estas) son las que consideraremos como *correctas*.

HISTORIAS SERIALIZABLES:

H es *serializable* *(SR) si es equivalente a una historia serial (H_s)*



GRAFO DE PRECEDENCIA:

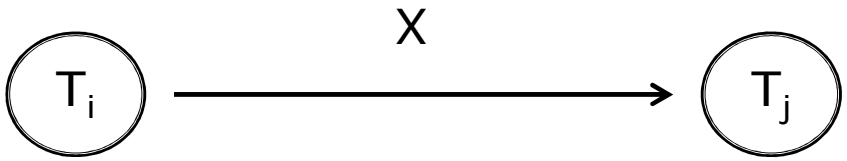
Dado H sobre T= $\{T_1, T_2, ..., T_n\}$, un SG para H, *SG(H)*, es un grafo dirigido cuyos nodos son los T_i y cuyos arcos $T_i \rightarrow T_j$ ($i \neq j$) son tal que alguna operación de T_i precede y conflictúa con alguna operación de T_j en H.

CONSTRUCCIÓN DEL GRAFO DE PRECEDENCIA (MODELO READ/WRITE):

Algoritmo:

1) Hacer un nodo por cada T_i y

2) Si alguna operación de T_i precede y conflictúa con alguna operación de T_j ($i \neq j$) en H, luego hacer un arco $T_i \rightarrow T_j$



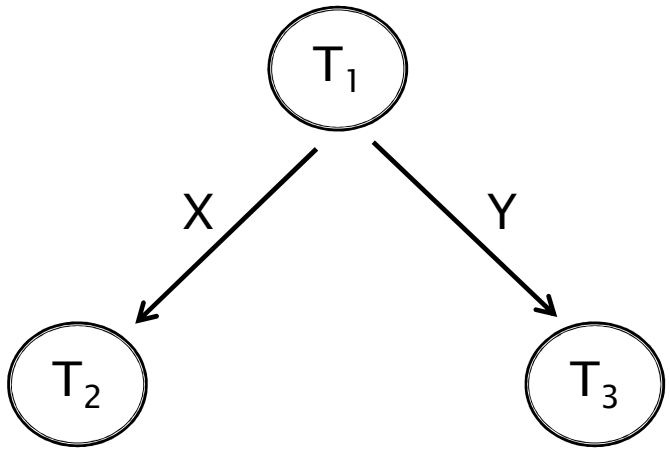
Ejemplo 4:

Dados :

T_1= w[X] w[Y] c
T_2= r[X] w[X] c
T_3= r[Y] w[Y] c

H= $w_1[X]$ $w_1[Y]$ c_1 $r_2[X]$ $r_3[Y]$ $w_2[X]$ c_2 $w_3[Y]$ c_3

SG(H):

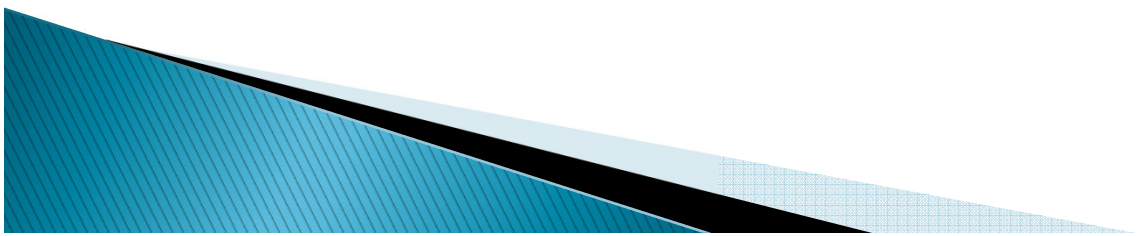


Luego para determinar si es Serializable observamos el siguiente teorema:

TEOREMA 1 DE SERIALIZABILIDAD:

H es SR si y solo si SG(H) es *acíclico*.

H es equivalente a cualquier Hs serial que sea un *ordenamiento topológico de SG(H)*



Theorem 2.1: (The Serializability Theorem) A history H is serializable iff SG(H) is acyclic.

Proof: (if) Suppose H is a history over $T = \{T_1, T_2, \ldots, T_n\}$. Without loss of generality, assume $T_1, T_2, \ldots, T_m$ ($m \leq n$) are all of the transactions in T that are committed in H. Thus $T_1, T_2, \ldots, T_m$ are the nodes of SG(H). Since SG(H) is acyclic it may be topologically sorted.[8] Let $i_1, \ldots, i_m$ be a permutation of $1, 2, \ldots, m$ such that $T_{i_1}, T_{i_2}, \ldots, T_{i_m}$ is a topological sort of SG(H). Let H_s be the serial history $T_{i_1} T_{i_2} \ldots T_{i_m}$. We claim that C($H$) $\equiv H_s$. To see this, let $p_i \in T_i$ and $q_j \in T_j$, where T_i, T_j are committed in H. Suppose p_i, q_j conflict and $p_i <_H q_j$. By definition of SG(H), $T_i \rightarrow T_j$ is an edge in SG(H). Therefore in any topological sort of SG(H), T_i must appear before T_j. Thus in H_s all operations of T_i appear before any operation of T_j, and in particular, $p_i <_{H_s} q_j$. We have proved that any two conflicting operations are ordered in C(H) in the same way as H_s. Thus C(H) $\equiv H_s$ and, because H_s is serial by construction, H is SR as was to be proved.

(only if) Suppose history H is SR. Let H_s be a serial history equivalent to C(H). Consider an edge $T_i \rightarrow T_j$ in SG(H). Thus there are two conflicting operations p_i, q_j of T_i, T_j (respectively), such that $p_i <_H q_j$. Because C(H) $\equiv H_s$, $p_i <_{H_s} q_j$. Because H_s is serial and p_i in T_i precedes q_j in T_j, it follows that T_i appears before T_j in H_s. Thus, we've shown that if $T_i \rightarrow T_j$ is in SG(H) then T_i appears before T_j in H_s. Now suppose there is a cycle in SG(H), and without loss of generality let that cycle be $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_k \rightarrow T_1$. These edges imply that in H_s, T_1 appears before T_2 which appears before T_3 which appears ... before T_k which appears before T_1. Thus, the existence of the cycle implies that each of $T_1, T_2, \ldots, T_k$ appears before itself in the serial history H_s, an absurdity. So no cycle can exist in SG(H). That is, SG(H) is an acyclic directed graph, as was to be proved. □

Extraído del libro:
Concurrency Control and Recovery in Database Systems; de Philip A. Bernstein, Vassos Hadzilacos y Nathan Goodman;1987

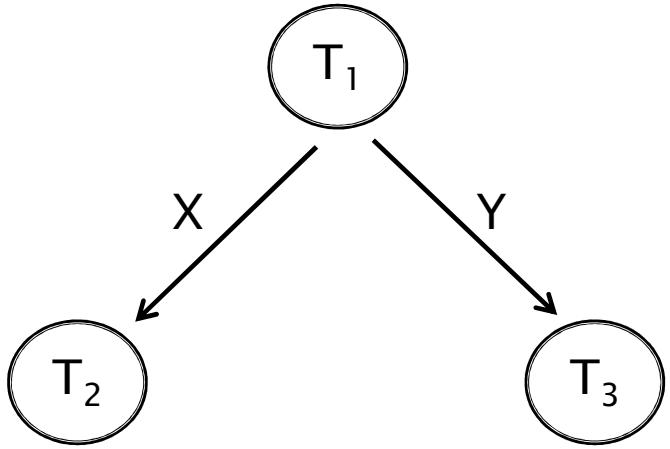
Ejemplo 4:

Dados :

T_1= w[X] w[Y] c
T_2= r[X] w[X] c
T_3= r[Y] w[Y] c

H= $w_1[X]$ $w_1[Y]$ c_1 $r_2[X]$ $r_3[Y]$ $w_2[X]$ c_2 $w_3[Y]$ c_3

SG(H):



Vemos que H es SR y es equivalente a las historias seriales:

H´ = T_1 T_2 T_3
H" = T_1 T_3 T_2

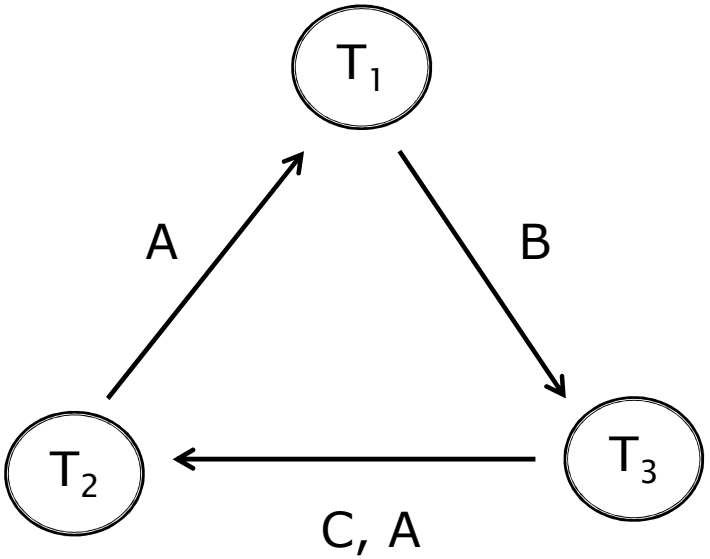
Ejemplo 5:

Dados:

T_1= r[A] w[B]
T_2= r[C] w[A]
T_3= r[A] w[C] w[B]

H= r_3[A] w_3[C] r_2[C] w_2[A] r_1[A] w_1[B] w_3[B]

SG(H):



Vemos que SG(H) tiene un ciclo, por lo tanto H no es SR.

LOCKING

DEFINICIÓN DE LOCK:

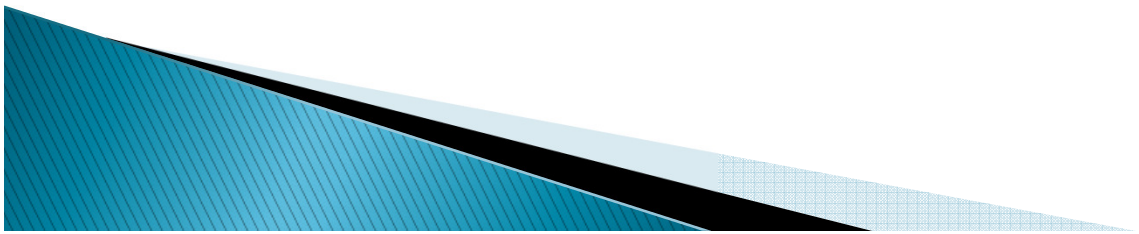
El *lock* es un privilegio de acceso a ítem de la BD.

El motor es el encargado de obtener y liberar los locks a pedido de las transacciones cuando éstas desean hacer un read o un write.

Al usar locking aparecen dos problemas:

Livelocks

Deadlocks



LOCKING BINARIO (Exclusive locks):

Este modelo de locking tiene 2 estados o valores:

Locked:	Lock(X)	$l_i[X]$
Unlocked:	Unlock(X)	$u_i[X]$

El lock binario fuerza exclusión mutua sobre un ítem X.

Ejemplo:
Si ahora reescribimos el programa P (Ejemplo 1) que producía lost update como:

P= Lock(A); Read(A); A:=A+1; Write(A); Unlock(A);

Nos queda:
T1= L(A) r(A) w(A) U(A)
T2= L(A) r(A) w(A) U(A)
H= $L_1(A)$ $r_1(A)$ $w_1(A)$ $U_1(A)$ $L_2(A)$ $r_2(A)$ $w_2(A)$ $U_2(A)$

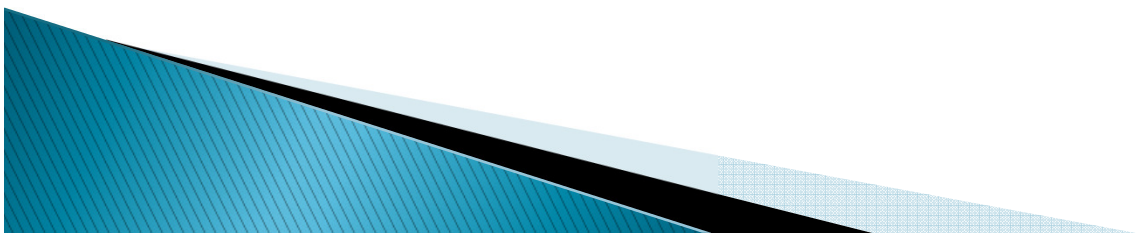
y vemos que el lost update no se puede producir, ¿por qué?

LOCKING TERNARIO (Shared/Exclusive locks):

Este modelo permite mayor concurrencia que el binario.
Tiene 3 estados o valores:

Read locked:	RLock(X)	$rl_i[X]$	(lock compartido)
Write locked:	WLock(X)	$wl_i[X]$	(lock exclusivo)
Unlocked:	ULock(X)	$ul_i[X]$ o $u_i[X]$	

NOTA: Puede ocurrir en algunos casos que una T_i requiera un *upgrade* de un RLock(X) a un WLock(X). A esto lo llamaremos *lock conversion*.



MODELO BASADO EN LOCKING:

En este modelo una transacción T es vista como una secuencia de locks y unlocks

Haremos abstracción de las operaciones de read y write que veníamos usando en el modelo anterior *sin locking*

Por ejemplo,

en el modelo de *locking binario* escribiremos las transacciones y las historias de esta forma:

T1= L[B] U[B]
T2= L[A] U[A] L[B] U[B]
T3= L[A] U[A]
H= $L_2[A]$ $U_2[A]$ $L_3[A]$ $U_3[A]$ $L_1[B]$ $U_1[B]$ $L_2[B]$ $U_2[B]$

y en el modelo de *locking ternario*:

T1= RL[A] WL[B] WL[A] UL[A] UL[B]
T2= RL[A] UL[A] RL[B] UL[B]
H= $RL_1[A]$ $RL_2[A]$ $WL_1[B]$ $UL_2[A]$ $WL_1[A]$ $UL_1[A]$ $UL_1[B]$ $RL_2[B]$ $UL_2[B]$

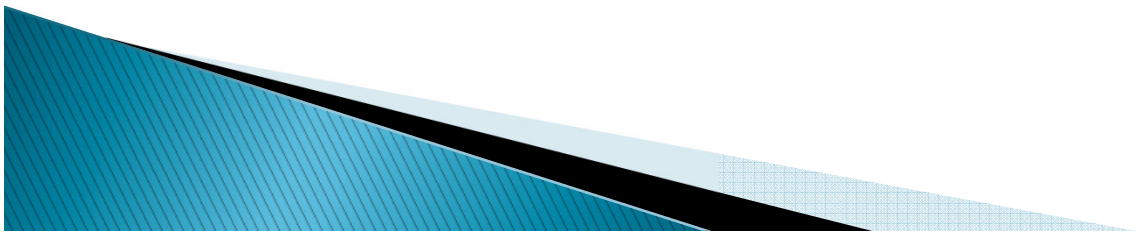
Nota: Observar el lock conversion en T_1

REGLAS DE LEGALIDAD DE LOCKING:

H es *legal* si:

1) Una T_i no puede leer ni escribir un ítem X hasta tanto no haya hecho un lock de X.

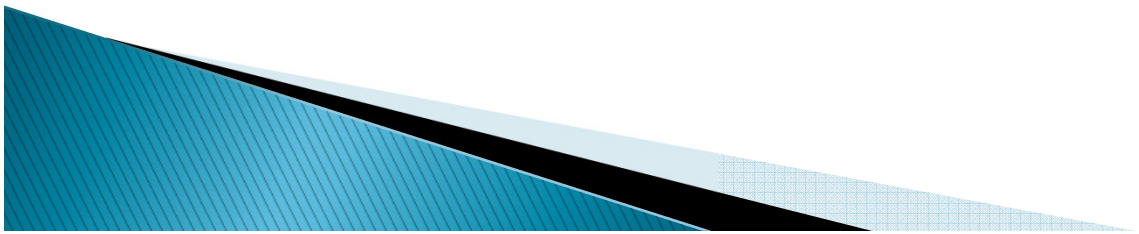
2) Una T_i que desea obtener un lock sobre X que ha sido lockeado por T_j en un modo que conflictúa, debe esperar hasta que T_j haga unlock de X.



MATRIZ DE COMPATIBILIDAD DE LOCKING:

Lock sostenido por T_j:

Lock pedido Por T_i:		**RLOCK**	**WLOCK**
	RLOCK	Y	N
	WLOCK	N	N



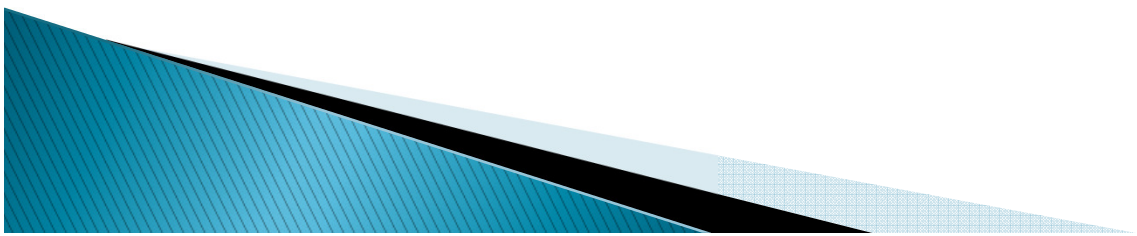
CONSTRUCCIÓN DEL GRAFO DE PRECEDENCIA (MODELO LOCKING BINARIO: LOCK/UNLOCK):

Algoritmo:

1) Hacer un nodo por cada T_i

2) Si T_i hace Lock de X y luego T_j ($i \neq j$) hace Lock de X en H, hacer un arco T_i -> T_j

NOTA: Para aplicar este algoritmo asumimos que H es legal.

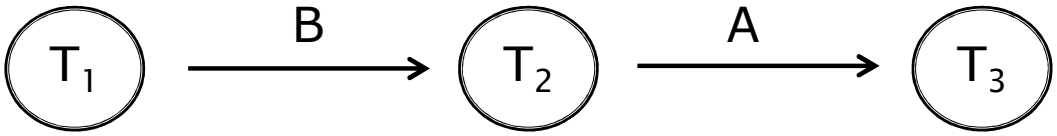


Ejemplo 6:

Dada:
H= $L_2[A]$ $U_2[A]$ $L_3[A]$ $U_3[A]$ $L_1[B]$ $U_1[B]$ $L_2[B]$ $U_2[B]$

Vemos que H es legal

Para ver si es SR hacemos el SG(H):



H es SR y es equivalente a T_1 T_2 T_3.

CONSTRUCCIÓN DEL GRAFO DE PRECEDENCIA (MODELO DE LOCKING TERNARIO: RLOCK/WLOCK/UNLOCK):

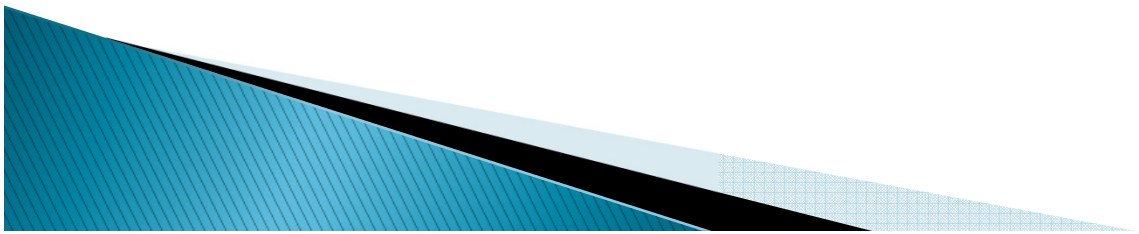
Algoritmo:

1) Hacer un nodo por cada T_i

2) Si T_i hace RLock o WLock de X, y luego T_j hace WLock de X en H ($i \neq j$), hacer un arco T_i -> T_j

3) Si T_i hace WLock de X y luego T_j ($i \neq j$) hace RLock de X en H, hacer un arco T_i -> T_j

NOTA: Para aplicar este algoritmo asumimos que H es legal.

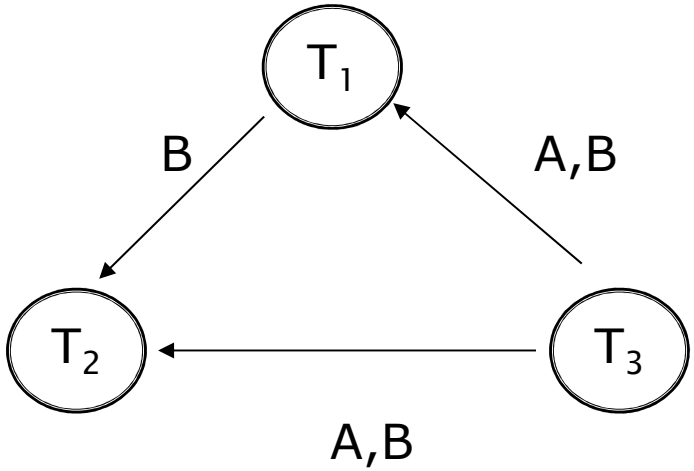


Ejemplo 7:

Dada:
H= $wl_3[A]$ $ul_3[A]$ $rl_1[A]$ $wl_3[B]$ $rl_2[A]$ $ul_3[B]$ $wl_1[B]$ $ul_2[A]$ $ul_1[A]$
$ul_1[B]$ $rl_2[B]$ $ul_2[B]$

Vemos que H es legal

Para ver si es SR hacemos el SG(H):



H es SR y equivalente a: T_3 T_1 T_2

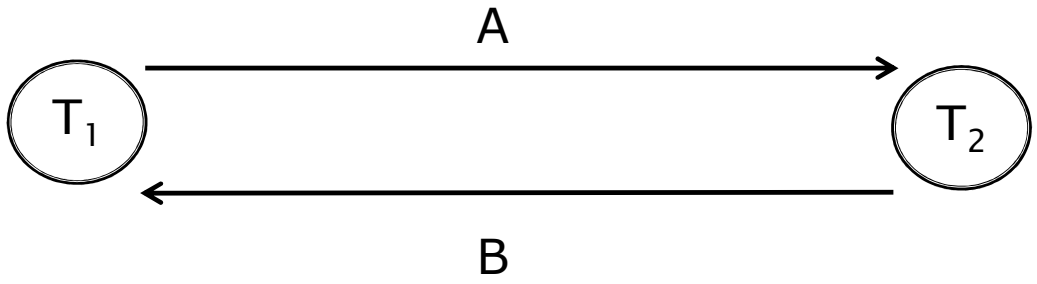
LOCKING Y SERIALIZABILIDAD:
Ahora nos podríamos preguntar si al usar locking (y H es legal) obtendremos siempre historias serializables.

Veamos un ejemplo:
Ejemplo 8:

T_1= l[A] u[A] l[B] u[B]
T_2= l[A] l[B] u[A] u[B]

H= l_1[A] u_1[A] l_2[A] l_2[B] u_2[A] u_2[B] l_1[B] u_1[B]

SG(H):



Vemos que H es legal pero no es serializable.

Ejemplo 9:

Dados:

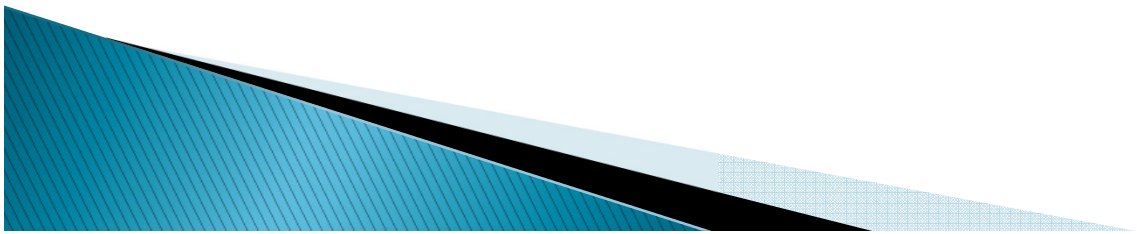
T_1= rl[A] wl[B] ul[A] ul[B]
T_2= rl[A] ul[A] rl[B] ul[B]
T_3= wl[A] ul[A] wl[B] ul[B]
T_4= rl[B] ul[B] wl[A] ul[A]

H= $wl_3[A]$ $rl_4[B]$ $ul_3[A]$ $rl_1[A]$ $ul_4[B]$ $wl_3[B]$ $rl_2[A]$ $ul_3[B]$ $wl_1[B]$

$ul_2[A]$ $ul_1[A]$ $wl_4[A]$ $ul_1[B]$ $rl_2[B]$ $ul_4[A]$ $ul_2[B]$

Vemos que H es legal

Si hacemos el SG(H) veremos que tiene ciclos y por lo tanto no es SR (por ejemplo T_3 precede a T_4 por A y al revés por B)



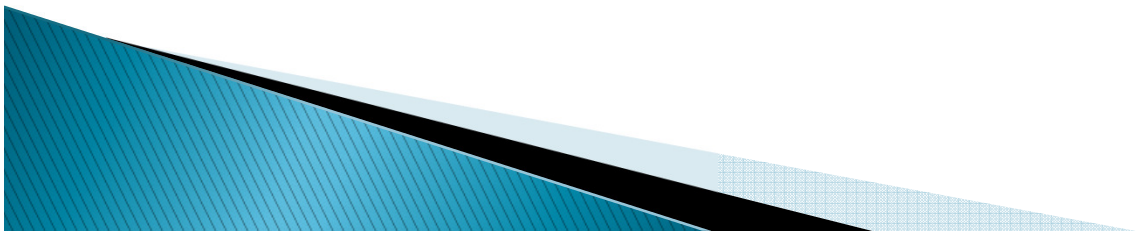
LOCKING Y SERIALIZABILIDAD (CONT.):

Observamos que el mecanismo de locking por si solo no garantiza serializabilidad. Se necesita agregar un *protocolo* para posicionar los locks y unlocks.

La idea es usar un protocolo de *dos fases* en cada transacción:

Una *primera fase* de crecimiento donde la transacción va tomando todos los ítems (locks) y luego

una *segunda fase* de decrecimiento donde los va liberando (unlocks)

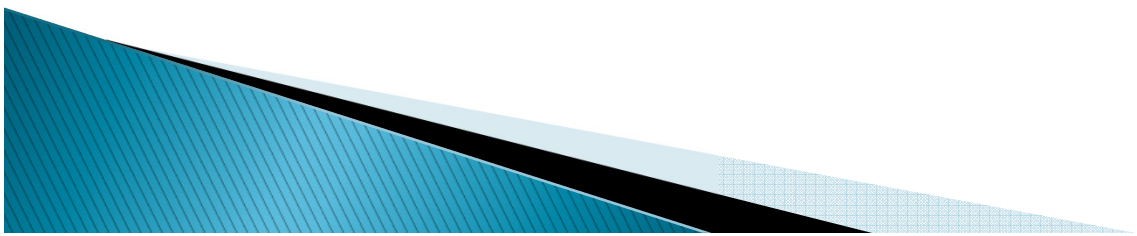


PROTOCOLO 2PL (Two Phase Locking):

T es 2PL si todos los locks preceden al primer unlock.

TEOREMA 2 DE SERIALIZABILIDAD:

Dado $T=\{T_1, T_2, \ldots, T_n\}$, si toda T_i en T es 2PL, entonces todo H legal sobre T es SR.



Ejemplo 10:

Si volvemos a considerar el Ejemplo 9 –de locking ternario- donde H no es SR veremos que T_2, T_3 y T_4 no son 2PL:

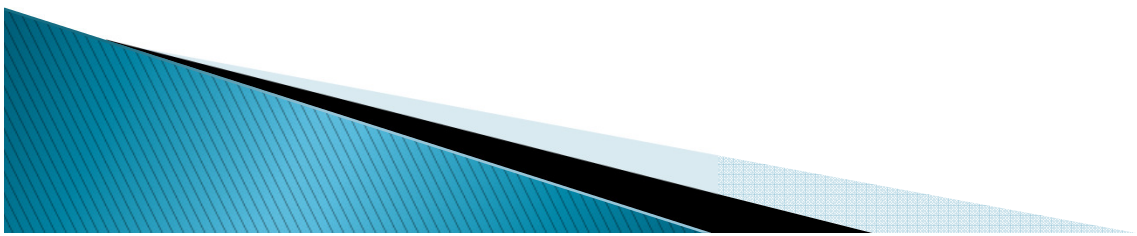
T_1= rl[A] wl[B] ul[A] ul[B]

T_2= rl[A] ul[A] rl[B] ul[B]

T_3= wl[A] ul[A] wl[B] ul[B]

T_4= rl[B] ul[B] wl[A] ul[A]

De la misma forma, si volvemos al Ejemplo 8 –de locking binario- veremos que T1 no es 2PL.



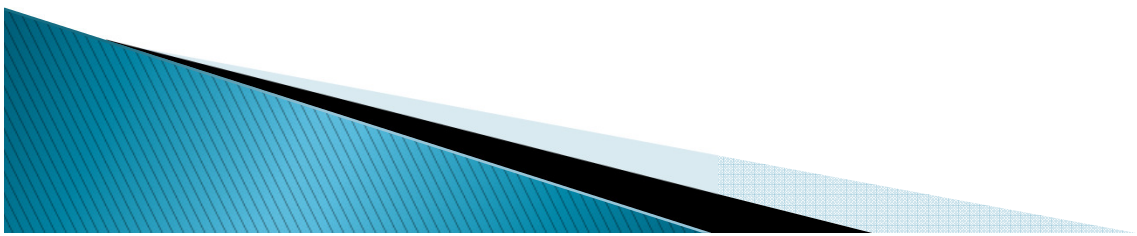
RECUPERABILIDAD EN BASES DE DATOS

EL PROBLEMA DE LA RECUPERABILIDAD

CLASIFICACION DE HISTORIAS SEGÚN RECUPERABILIDAD

RECUPERABILIDAD Y SERIALIZABILIDAD

LOCKING Y RECUPERABILIDAD



LECTURA ENTRE TRANSACCIONES (Ti READ FROM Tj):

Decimos que *T_i lee de T_j* en H si T_j es la transacción que última escribió sobre X (y no abortó) al tiempo que T_i lee X.

O dicho en forma más rigurosa, si:

1. $W_j(X) < R_i(X)$ (1)
2. $A_j \not< R_i(X)$ (2)
3. Si hay algún $W_k(X)$ tal que $W_j(X) < W_k(X) < R_i(X)$, entonces $A_k < R_i(X)$

(1) <, significa precede

(2) /<, significa no precede

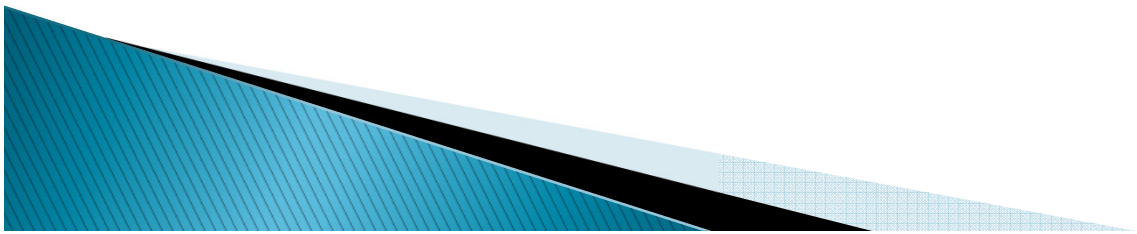
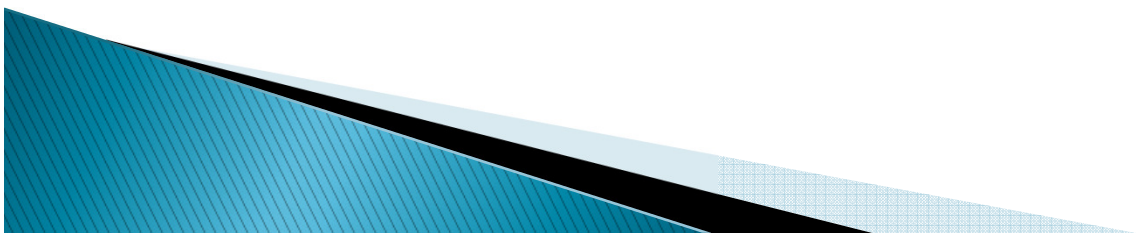


IMAGEN ANTERIOR DE UN WRITE (BEFORE IMAGE):

La *imagen anterior* de una operación Write(X,val) es el valor que tenía X justo antes de esta operación.

Podemos asumir que el SGBD implementa el *abort* restaurando las imágenes anteriores de todos los writes de una transacción.



EL PROBLEMA DE LA RECUPERABILIDAD:

Ejemplo 1:

H_1= Write$_1$(X,2); Read$_2$(X); Write$_2$(Y,3); Commit$_2$;

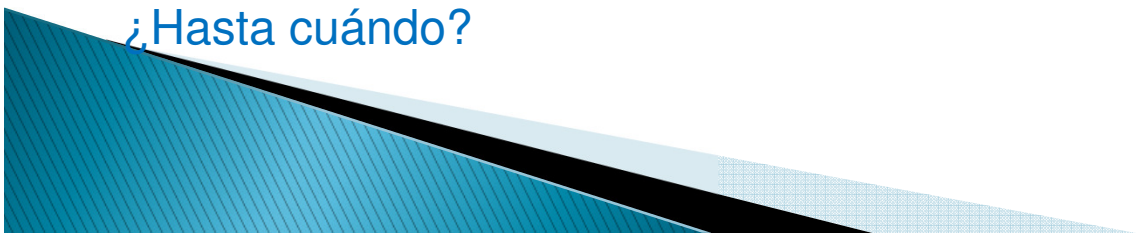
Supongamos que inicialmente los item X e Y tienen un valor igual a 1.

Ahora supongamos que T_1 aborta –y por lo tanto debería hacer un rollback y volver X al valor anterior- y entonces luego T_2 debería también abortar y hacer un rollback –porque leyó un valor sucio que le dejó T_1- pero si lo hacemos estaríamos violando la semántica del commit y esto trae confusión.

Llegamos a una situación en el que estado consistente anterior de la BD es irrecuperable.

Para evitar esta situación deberíamos demorar el commit de T_2.

¿Hasta cuándo?



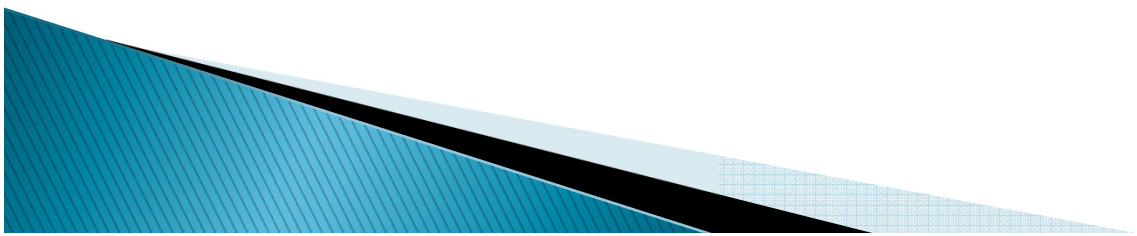
Ejemplo 2:

H_2= Write$_1$(X,2); Read$_2$(X); Write$_2$(Y,3); Abort$_1$;

Supongamos un caso similar al anterior pero donde T_1 abortó y por lo tanto todavía podemos recuperar el estado consistente anterior abortando T_2.

Pero sin embargo esto nos puede llevar a la situación no deseada de *aborts en cascada*.

Para evitar esta situación deberíamos demorar cada Read(X) hasta que los correspondientes Ti que previamente hicieron un Write(X,val) hayan hecho abort o commit.



Ejemplo 3:

H3= $Write_1(X,2)$; $Write_2(X,3)$; $Abort_1$; $Abort_2$;

Supongamos que inicialmente el item X tiene un valor igual a 1.

Aquí vemos que la imagen anterior de $Write_2(X,3)$ es 2, escrito por T_1.

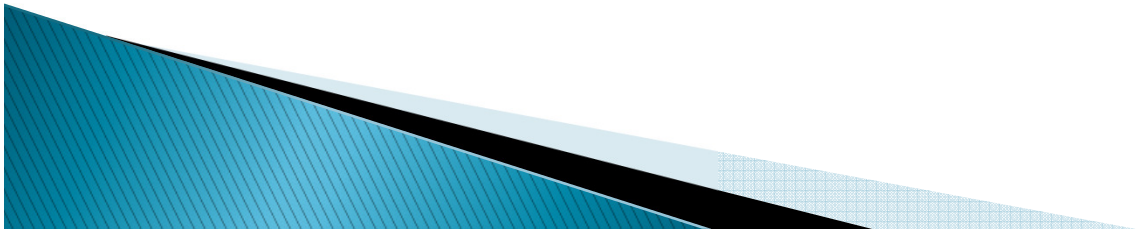
El valor de X, después de que $Write_2(X,3)$ es deshecho, debería ser 1 que es el valor inicial de X, dado que ambos updates de X fueron abortados (como si no se hubieran ejecutado ninguna de las dos transacciones)

Aunque el estado anterior todavía podría recuperarse, dado que no hubo commit y todo puede deshacerse, igualmente hemos llegado a una situación de confusión.

El problema es que dejó de funcionar la implementación del abort (como restauración de las imágenes anteriores de los writes de una transacción)

Podemos evitar este problema pidiendo que la ejecución de un Write(X,val) sea demorado hasta que la transacción que previamente escribió X haya hecho commit o abort.

Si pedimos lo mismo con respecto al Read(X) decimos que tenemos una *ejecución estricta*.



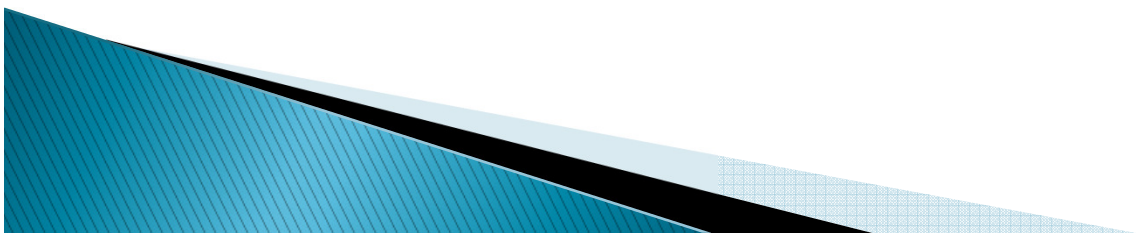
CLASIFICACIÓN DE HISTORIAS SEGÚN RECUPERABILIDAD:

No Recuperables (No RC)

Recuperables (RC)

Evitan Aborts en Cascada (ACA)

Estrictas (ST)

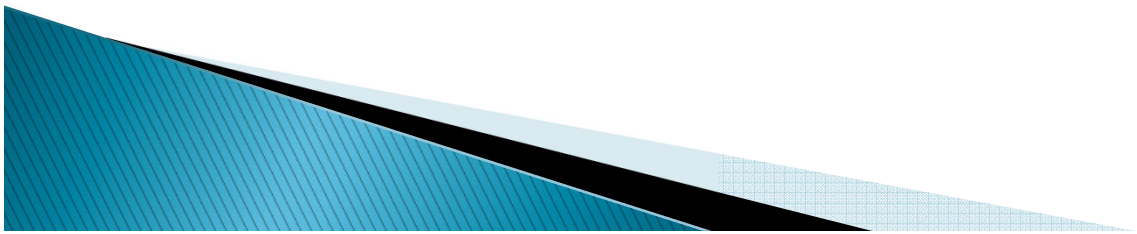


HISTORIAS RECUPERABLES:

Decimos que H es *recuperable* (RC), si cada transacción hace su commit después de que hayan hecho commit todas las transacciones (otras de si misma) de las cuales lee.

O en forma equivalente:

Siempre que T_i lee de T_j ($i \neq j$) en H y $C_i \in H$ y $C_j < C_i$

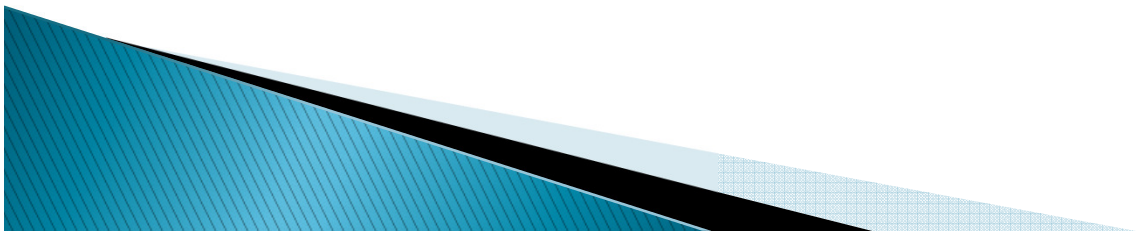


HISTORIAS QUE EVITAN ABORTS EN CASCADA:

Decimos que H *evita aborts en cascada* (avoids cascading aborts) (ACA), si cada transacción puede leer solamente aquellos valores que fueron escritos por transacciones que ya hicieron commit (o por si misma)

O en forma equivalente:

Siempre que T_i lee X de T_j ($i \neq j$) en H y $C_i \in H$ y $C_j < R_i(X)$

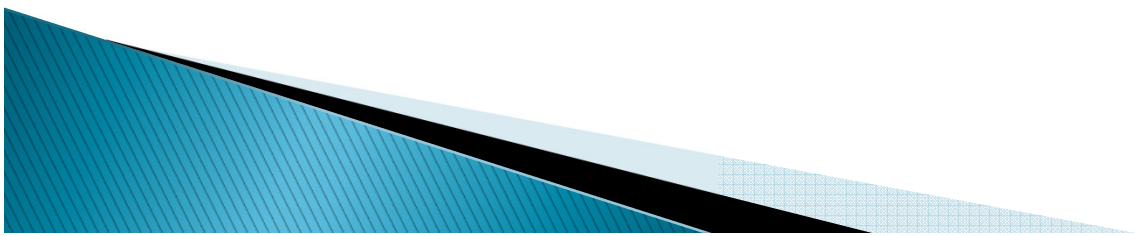


HISTORIAS ESTRICTAS:

Decimos que H es *estricta* (ST), si ningún item X puede ser leído o sobrescrito hasta que la transacción que previamente escribió X haya finalizado haciendo commit o abort.

O en forma equivalente:

Siempre que $W_j(X) < O_i(X)$ $(i \neq j)$, o $A_j < O_i(X)$ o $C_j < O_i(X)$, donde $O_i(X)$ es $R_i(X)$ o $W_i(X)$



Ejemplo 4:

T_1= W(X) W(Y) W(Z) C

T_2= R(U) W(X) R(Y) W(Y) C

H_7 = $W_1(X)$ $W_1(Y)$ $R_2(U)$ $W_2(X)$ $R_2(Y)$ $W_2(Y)$ **C_2** W1(Z) **C_1**

H_8 = $W_1(X)$ $W_1(Y)$ $R_2(U)$ $W_2(X)$ $R_2(Y)$ $W_2(Y)$ $W_1(Z)$ **C_1** **C_2**

H_9 = $W_1(X)$ $W_1(Y)$ $R_2(U)$ $W_2(X)$ $W_1(Z)$ **C_1** $R_2(Y)$ $W_2(Y)$ **C_2**

H_{10}= $W_1(X)$ $W_1(Y)$ $R_2(U)$ $W_1(Z)$ **C_1** $W_2(X)$ $R_2(Y)$ $W_2(Y)$ **C_2**

Vemos que:

H_7 no es RC porque T_2 lee Y de T_1 pero $C_2 < C_1$.

H_8 es RC pero no es ACA porque T_2 lee Y de T_1 antes que T_1 haya hecho commit.

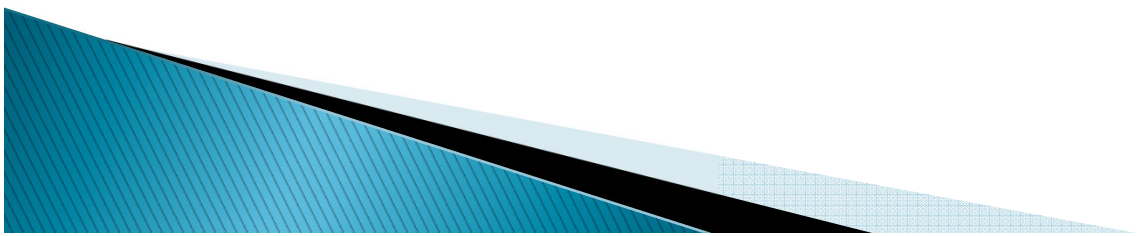
H_9 es ACA pero no es ST porque T_2 sobreescribe el valor de X escrito por T_1 antes que T_1 termine.

H_{10} es ST.

TEOREMA DE RECUPERABILIDAD:

$$ST \subset ACA \subset RC.$$

Este teorema nos dice que las propiedades de ST son más restrictivas que las de ACA y que las de éstas son a su vez más restrictivas que las de RC.

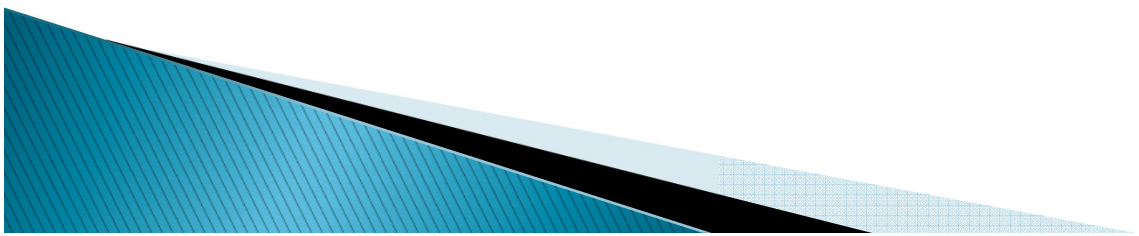


RECUPERABILIDAD y SERIALIZABILIDAD:

El concepto de recuperabilidad es ortogonal al concepto de serializabilidad, o sea que una historia H puede ser no RC, RC, ACA o ST y a la vez ser SR o no SR.

El conjunto SR intersecta los conjuntos RC, ACA y ST pero no es comparable a cada uno de ellos.

Las historias seriales son ST y SR.



Ejemplo 5:

Dadas las mismas historias del ejemplo anterior:

$H_7 = W_1(X)\ W_1(Y)\ R_2(U)\ W_2(X)\ R_2(Y)\ W_2(Y)\ C_2\ W_1(Z)\ C_1$

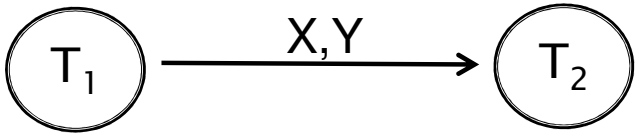
$H_8 = W_1(X)\ W_1(Y)\ R_2(U)\ W_2(X)\ R_2(Y)\ W_2(Y)\ W_1(Z)\ C_1\ C_2$

$H_9 = W_1(X)\ W_1(Y)\ R_2(U)\ W_2(X)\ W_1(Z)\ C_1\ R_2(Y)\ W_2(Y)\ C_2$

$H_{10} = W_1(X)\ W_1(Y)\ R_2(U)\ W_1(Z)\ C_1\ W_2(X)\ R_2(Y)\ W_2(Y)\ C_2$

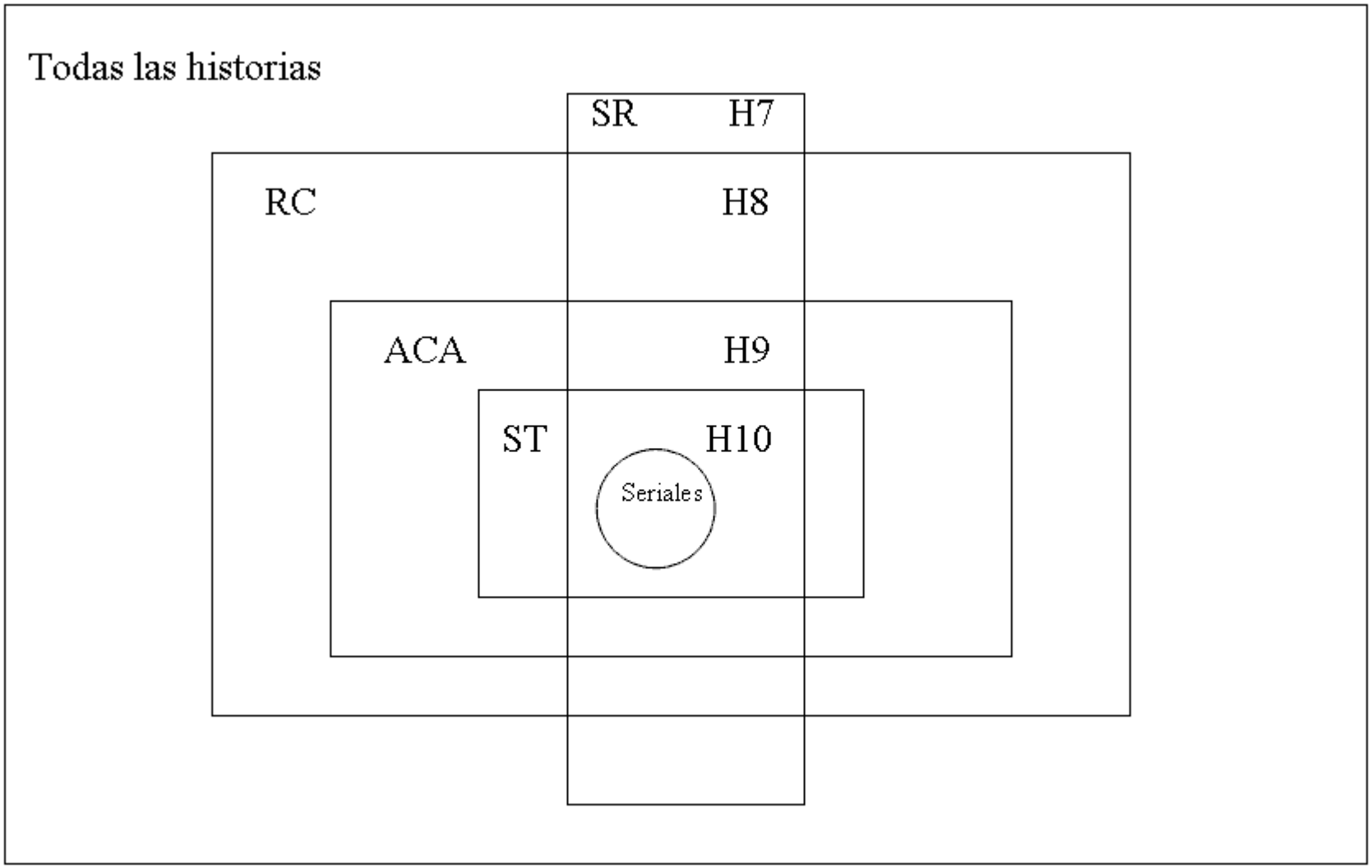
Vemos que:

$SG(H_7) = SG(H_8) = SG(H_9) = SG(H_{10}) =$



No hay ciclos y por lo tanto son todas SR.

Diagrama de Venn:



¿LAS HISTORIAS ESTRICTAS GARANTIZAN SERIALIZABILIDAD?

Ejemplo 6:

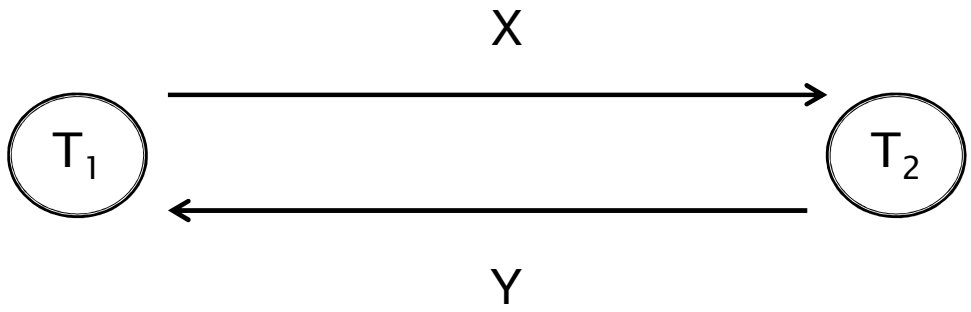
T_1= R(X) W(Y) C

T_2= W(Y) W(X) C

H= R_1(X) W_2(Y) W_2(X) C_2 W_1(Y) C_1

Vemos que H es estricta.

Hacemos el SG(H) para ver si es SR:

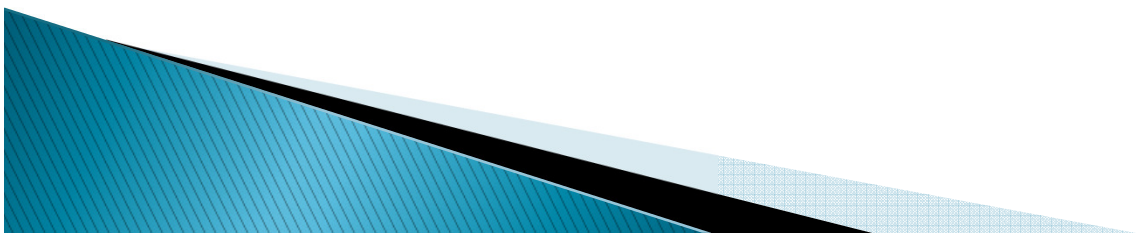


y vemos que H no es SR aunque es ST.

LOCKING Y RECUPERABILIDAD:

Hay una variante del protocolo 2PL que además de serializabilidad garantiza recuperabilidad:

Es el protocolo 2PL Estricto (2PLE)



PROTOCOLO 2PL ESTRICTO (STRICT 2PL):

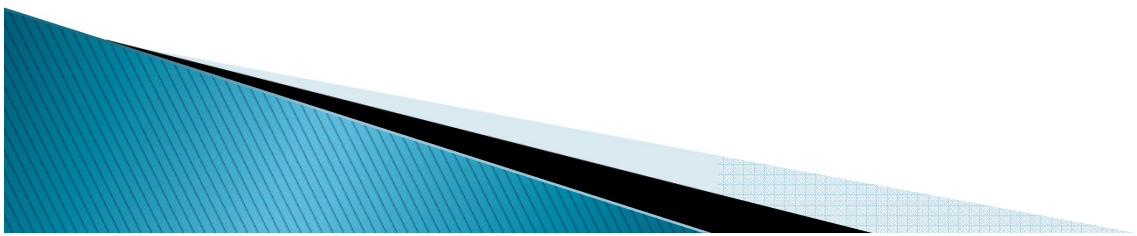
T cumple con *2PL estricto* (2PLE) si cumple con 2PL y además no libera ninguno de sus locks exclusivos (WriteLocks) hasta después de haber hecho commit o abort. (1)

Observamos que:

Este protocolo garantiza historias *estrictas (ST)* con respecto a *recuperabilidad*, y *serializables (SR)* con respecto a *serializabilidad*, o sea que todo H que cumpla con 2PL estricto (todas sus Ti son 2PL estrictas) es ST y SR.

Sin embargo, 2PL estricto no garantiza que estemos libres de deadlocks

(1) Observar que en el modelo de locking (binario o ternario) los unlocks pueden ejecutarse después del correspondiente commit o abort de la transacción (esto no afecta la legalidad).



PROTOCOLO 2PLE (Cont.)

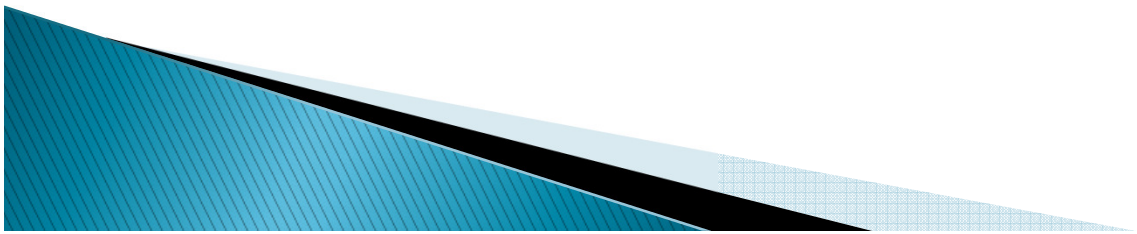
Por ejemplo, si usamos el *protocolo 2PLE*,

en el modelo de *locking binario* debemos escribir las transacciones y las historias de esta forma:

T_1= L[B] C U[B]
T_2= L[A] L[B] C U[A] U[B]
T_3= L[A] C U[A]
H= $L_2[A]\ L_1[B]\ C_1\ U_1[B]\ L_2[B]\ C_2\ U_2[A]\ L_3[A]\ U_2[B]\ C_3\ U_3[A]$

y en el modelo de *locking ternario*:

T_1= RL[A] WL[B] UL[A] C UL[B]
T_2= RL[A] RL[B] UL[A] UL[B] C
H= $RL_1[A]\ RL_2[A]\ WL_1[B]\ UL_1[A]\ C_1\ UL_1[B]\ RL_2[B]\ UL_2[A]\ UL_2[B]\ C_2$



¿PREGUNTAS?