Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación
Cátedra de Base de Datos

Autor: Alejandro Eidelsztein

CONTROL DE CONCURRENCIA Y RECUPERACIÓN EN BASES DE DATOS

1. TRANSACCIONES Y SERIALIZABILIDAD[1] :

1.1. 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.

1.2. 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 Ti.
Una transacción es una sucesión de *acciones* (u operaciones)
Una acción es un paso atómico. Estos pueden ser:

- *leer* un ítem X: ri[X]
- *escribir* un ítem X: wi[X]
- *abort* de Ti : ai
- *commit* de Ti : ci

Ahora podemos definir y escribir una transacción como:

$Ti \subseteq \{ri[X], wi[X] / X \text{ es un ítem}\} \cup \{ai, ci\}$

A esta forma de describir una transacción lo llamaremos *modelo read/write* o *modelo sin locking*.

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* que analizaremos en la próxima clase.

Dos problemas clásicos que se pueden presentar son el *lost update* y el *dirty read*.

1.3. LOST UPDATE:

El lost update o actualización perdida ocurre cuando se pierde la actualización hecha por una transacción T1 por la acción de otra transacción T2 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, T1 y T2 sobre el ítem A, con el siguiente entrelazamiento:

T1	T2	A < -- (valor del ítem A en disco, inicialmente=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 T2)

[1] En este apunte cuando indiquemos "serializabilidad" (serializability en inglés) nos estaremos refiriendo en todos los casos a "conflict-serializability". De igual forma cuando indiquemos "equivalente" (equivalent) y "serializable" nos estaremos refiriendo a conflict-equivalent y conflict-serializable.

Nota:
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.
A:=A+1 se hace sobre la variable local de la transacción.

1.4. DIRTY READ:
El dirty read o lectura sucia ocurre cuando una transacción T2 lee un valor de un ítem dejado por otra transacción T1 que no hizo commit antes de que T2 leyera el ítem.
Dicho de otra forma, es cuando T1 lee valores no "comiteados" dejados por T2.
Observemos que si T1 aborta, T2 se quedó con un valor sucio que será deshecho por el rollback de T1.
Esto además podría producir un fenómeno no deseado como es el de *aborts en cascada* (si T2 leyó de T1 que abortó, deberíamos abortar T2, luego si T3 leyó de T2 deberíamos abortar a su vez T3 y así sucesivamente)

Ejemplo 2:
Supongamos los programas
P1= Read(A); A:=A-1; Write(A); Read(B); B:=B/A; Write(B); y
P2= Read(A); A:=A*2; Write(A);
y dos ejecuciones: una T1 de P1 sobre los ítems A y B, y una T2 de P2 sobre el ítem A, con el siguiente entrelazamiento:

T1	T2	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	< -- (T1 falla por la división por cero y aborta volviendo A al valor anterior, pero T2 ya leyó A)

1.5. NON-REPEATABLE READ[1]:
El non-repeatable read o lectura no repetible ocurre cuando una transacción T1 lee un ítem, otra transacción T2 lo modifica y luego T1 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.

Ejemplo 2.1:
Supongamos los programas
P1= Read(A); Read(A);
P2= Read(A); Write(A);

y dos ejecuciones T1 de P1 y T2 de P2 sobre el ítem A, con el siguiente entrelazamiento:

T1	T2	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	< -- (T1 vuelve a leer A pero ve que ahora tiene otro Valor)

1.6. PHANTOM READ:
El phantom read o lectura fantasma ocurre cuando una transacción T1 ejecuta un query y lee un conjunto de ítems (generalmente tuplas), otra transacción T2 inserta nuevos items y luego T1 vuelve a ejecutar el mismo query y ahora el conjunto de ítems ha cambiado.

[1] También llamado Unrepeteable Read.

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.

1.5. 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:

- **A**tomicidad: T se ejecuta completamente o no se ejecuta por completo (todo o nada)
- **C**onsistencia: 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.
- **D**urabilidad: Las actualizaciones a la BD serán durables y públicas.

1.6. HISTORIAS (SCHEDULES)

Por ejemplo, si P= Read(X); X:=X+1; Write(X); Commit;
entonces dos ejecuciones distintas de P, T1 y T2 las escribiremos como:
T1= r1[X] w1[X] c1
T2= r2[X] w2[X] c2

Si T= {T1,T2, ...,Tn} es un conjunto de transacciones, entonces una *historia* (o *schedule*) H sobre T es:
H= Ui=1,n Ti
Donde H respeta el orden de las acciones de cada Ti.

Ejemplo 3:
Si T1= r1[X w1[X] c1 y T3= r3[X] w3[Y] w3[X] c3, una historia H1 sobre el conjunto de transacciones {T1,T3} y el conjunto de ítems {X,Y} podría ser:

H1= r1[X] r3[X] w1[X] c1 w3[Y] w3[X] c3

También podemos expresar H1 en forma tabular:

T1	T3
r[X]	
	r[X]
w[X]	
c	
	w[Y]
	w[X]
	c

1.7. 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 Ti y Tj (i≠j) son *conflictivas* si operan sobre el mismo ítem y al menos alguna de las dos es un write.

1.8. HISTORIAS SERIALES:

H es *serial* (Hs) si para todo par de transacciones Ti y Tj en H, todas las operaciones de Ti preceden a las de Tj o viceversa.
Las historias seriales (y las equivalentes a estas) son las que consideraremos como correctas.

1.9. HISTORIAS SERIALIZABLES:
H es *serializable* (SR) si es equivalente a una historia serial (Hs)

1.10. GRAFO DE PRECEDENCIA:
Dado H sobre T= {T1,T2, ...,Tn}, un SG para H, *SG(H)*, es un grafo dirigido cuyos nodos son los Ti y cuyos arcos Ti → Tj (i≠j), tal que alguna operación de Ti precede y conflictua con alguna operación de Tj en H.

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

1. Hacer un nodo por cada Ti y
2. Si alguna operación de Ti precede y conflictua con alguna operación de Tj (i<>j) en H, luego hacer un arco Ti → Tj

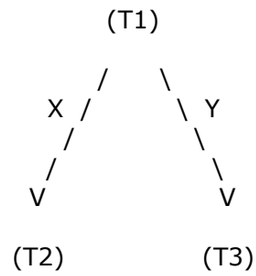
1.12. TEOREMA 1 DE SERIALIZABILIDAD:
H es SR si y solo si SG(H) es acíclico.

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

Ejemplo 4:
Dados :
T1= w[X] w[Y] c
T2= r[X] w[X] c
T3= r[Y] w[Y] c

H= w1[X] w1[Y] c1 r2[X] r3[Y] w2[X] c2 w3[Y] c3

SG(H):

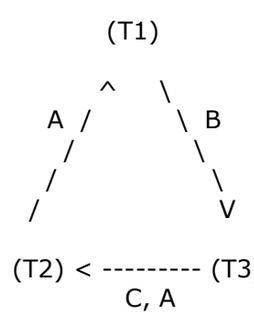


Vemos que H es SR (serializable) y es equivalente a las historias seriales:
H´= T1 T2 T3
H" = T1 T3 T2

Ejemplo 5:
Dados:
T1= r[A] w[B]
T2= r[C] w[A]
T3= r[A] w[C] w[B]

H= r3[A] w3[C] r2[C] w2[A] r1[A] w1[B] w3[B]

SG(H):



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

Veamos ahora un ejercicio típico:

Dada la siguiente historia H, sobre el conjunto de transacciones {T1, T2, T3, T4} y el conjunto de items {A, B, C, D, E}:

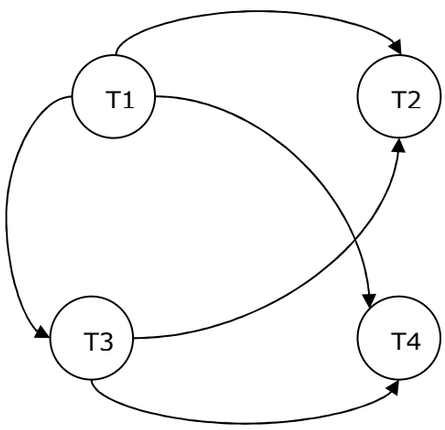
H= r2[E] w1[A] r2[A] r1[B] r3[A] w3[D] r3[C] r4[A] r3[B] w2[C] r4[D] r1[E]

Hacer el SG(H), indicar si es SR, y en caso afirmativo, indicar todas las historias seriales equivalentes.

Para resolver el ejercicio en primer lugar es conveniente reescribir H en forma tabular:

	T1	T2	T3	T4
1		R(E)		
2	W(A)			
3		R(A)		
4	R(B)			
5			R(A)	
6			W(D)	
7			R(C)	
8				R(A)
9			R(B)	
10		W(C)		
11				R(D)
12	R(E)			

Luego dibujamos el grafo de precedencia SG(H):
Para esto aplicamos el algoritmo 1.11: dibujamos un nodo para cada transacción y un arco de Ti a Tj (i<>j) si alguna operación de Ti precede y conflictua con alguna operación de Tj.



SG(H) es acíclico por lo tanto H es serializable (SR).
Esto es por aplicación del teorema de serializabilidad: H es SR, sii SG(H) es acíclico.

Por último para obtener las historias seriales equivalentes a H listamos los ordenamientos topológicos de SG(H).
Para esto aplicamos el siguiente algoritmo de ordenamiento topológico: Buscamos el nodo al cual no llegan arcos, lo listamos y lo quitamos del grafo junto con sus arcos, y así sucesivamente volvemos a aplicar el procedimiento hasta que hayamos quitado todos los nodos.
En este caso nos queda:

T1, T3, T2, T4
T1, T3, T4, T2

La ejecución de H será equivalente a cualquiera de las dos ejecuciones seriales indicadas arriba.

2. LOCKING:

2.1. DEFINICIÓN DE LOCK:
El *lock* es un privilegio de acceso a ítem de la BD.

Decimos que una Ti setea u obtiene un lock sobre el item X.

Al usar locking aparecen dos problemas que consideraremos en la próxima clase: *Livelocks* y *Deadlocks.*

2.2. LOCKING BINARIO (Exclusive locks):
Este modelo de locking tiene 2 estados o valores:

Locked:	Lock(X)	li[X]
Unlocked:	Unlock(X)	ui[X]

El lock binario fuerza exclusión mutua sobre un ítem X.

Ejemplo:
Si reescribimos el programa P que producía lost update como:
P= Lock(A); Read(A); A:=A+1; Write(A); Unlock(A);
y hacemos la historia H con T1 y T2 veremos que el lost update no se produce.

2.3. LOCKING TERNARIO (Shared/Exclusive locks):
Este modelo permite mayor concurrencia que el binario.
Tiene 3 estados o valores:

Read locked:	RLock(X)	rli[X]	(lock compartido)
Write locked:	WLock(X)	wli[X]	(lock exclusivo)
Unlocked:	ULock(X)	uli[X] o ui[X]	

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

2.4. 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 las operaciones de read y write que veníamos usando en el modelo anterior sin locking.

2.5. REGLAS DE LEGALIDAD DE LOCKING:
H es legal si:
1. Una Ti no puede leer ni escribir un ítem X hasta tanto no haya hecho un lock de X.
2. Una Ti que desea obtener un lock sobre X que ha sido lockeado por Tj en un modo que conflictua, debe esperar hasta que Tj haga unlock de X.

2.6. MATRIZ DE COMPATIBILIDAD DE LOCKING (CONFLICTOS):

		Lock sostenido por Tj:	
		RLOCK	**WLOCK**
Lock pedido por Ti:	**RLOCK**	Y	N
	WLOCK	N	N

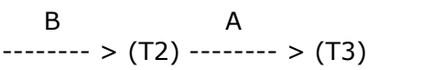
2.7. CONSTRUCCIÓN DEL GRAFO DE PRECEDENCIA (MODELO LOCKING BINARIO: LOCK/UNLOCK)[1]:
Algoritmo:
1. Hacer un nodo por cada Ti
2. Si Ti hace Lock de X y luego Tj (i<>j) hace Lock de X en H, luego hacer un arco Ti -> Tj

[1] Para aplicar este algoritmo asumimos que la historia H es legal.

Ejemplo 6:
Dada:
H= l2[A] u2[A] l3[A] u3[A] l1[B] u1[B] l2[B] u2[B]

Vemos que H es legal

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



H es SR y es equivalente a T1 T2 T3.

2.8. CONSTRUCCIÓN DEL GRAFO DE PRECEDENCIA (MODELO LOCKING TERNARIO: RLOCK/WLOCK/UNLOCK)[2]:

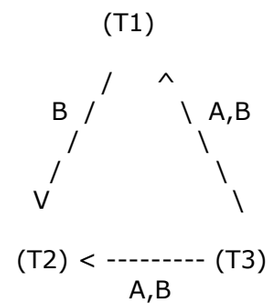
Algoritmo:

1. Hacer un nodo por cada Ti
2. Si Ti hace RLock o WLock de X, y luego Tj (i<>j) hace WLock de X en H, luego hacer un arco Ti -> Tj
3. Si Ti hace WLock de X y Tj (i<>j) hace RLock de X en H, luego hacer un arco Ti -> Tj

Ejemplo 7:
Dados:
T1= rl[A] wl[B] ul[A] ul[B]
T2= rl[A] ul[A] rl[B] ul[B]
T3= wl[A] ul[A] wl[B] ul[B]

H= wl3[A] ul3[A] rl1[A] wl3[B] rl2[A] ul3[B] wl1[B] ul2[A] ul1[A] ul1[B] rl2[B] ul2[B]

Vemos que H es legal
Hacemos el SG(H):



Vemos que H es SR y equivalente a T3,T1,T2

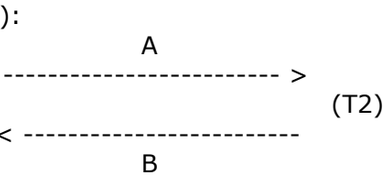
2.9. LOCKING Y SERIALIZABILIDAD:

Ahora nos podríamos preguntar si al usar locking (y H es legal) obtendremos siempre historias serializables. Veamos un ejemplo.

Ejemplo 8:
T1= l[A] u[A] l[B] u[B]
T2= l[A] l[B] u[A] u[B]

H= l1[A] u1[A] l2[A] l2[B] u2[A] u2[B] l1[B] u1[B]

SG(H):



Vemos que H es legal pero no es serializable.

Ejemplo 9:
Dados:
T1= rl[A] wl[B] ul[A] ul[B]

[2] Para aplicar este algoritmo asumimos que la historia H es legal.

T2= rl[A] ul[A] rl[B] ul[B]
T3= wl[A] ul[A] wl[B] ul[B]
T4= rl[B] ul[B] wl[A] ul[A]

H= wl3[A] rl4[B] ul3[A] rl1[A] ul4[B] wl3[B] rl2[A] ul3[B] wl1[B] ul2[A] ul1[A] wl4[A] ul1[B] rl2[B] ul4[A] ul2[B]

Vemos que H es legal
Si hacemos el SG(H) veremos que tiene ciclos y por lo tanto no es SR.

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)

2.10. PROTOCOLO 2PL (Two Phase Locking):

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

2.11. TEOREMA 2 DE SERIALIZABILIDAD:

Dado T={T1, T2, …, Tn}, si toda Ti en T es 2PL, entonces todo H legal sobre T es SR.

Si volvemos a considerar el Ejemplo 8 -de locking binario- donde H no es SR veremos que T1 no es 2PL.
Idem para el Ejemplo 9 –de locking ternario- donde H no es SR veremos que T2, T3 y T4 no son 2PL.

3. RECUPERABILIDAD:

3.1. LECTURA ENTRE TRANSACCIONES (*Ti READ FROM Tj*):

Decimos que *Ti lee de Tj* en H si Tj es la transacción que última escribió sobre X, pero no abortó, al tiempo que Ti lee X.

O dicho en otra forma más rigurosa, si:

1. Wj(X) < Ri(X)[3]
2. Aj /< Ri(X)[4]
3. Si hay algún Wk(X) tal que Wj(X) < Wk(X) < Ri(X), entonces Ak < Ri(X)

3.2. 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.

3.3. EL PROBLEMA DE LA RECUPERABILIDAD:

Veamos algunos ejemplos:

Ejemplo 1:

Sea la siguiente historia:

H1= Write1(X,2); Read2(X); Write2(Y,3); Commit2.

Supongamos que inicialmente los item X e Y tienen un valor igual a 1.
Ahora supongamos que T1 aborta –y por lo tanto debería hacer un rollback y volver X al valor anterior- y entonces luego T2 debería también abortar y hacer un rollback –porque leyó un valor

[3] p < q denota que la operación p precede a q en el orden de ejecución.

[4] p /<q denota que la operación p no precede a q en el orden de ejecución.

sucio que le dejó T1- 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 T2.

Ejemplo 2:

Sea H2= Write1(X,2); Read2(X); Write2(Y,3); Abort1.

Supongamos un caso similar al anterior pero donde T1 abortó y por lo tanto todavía podemos recuperar el estado consistente anterior abortando T2.
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 Ti que previamente hicieron un Write(X,val) hayan hecho un abort o un commit.

Ejemplo 3:

Sea H3= Write1(X,2); Write2(X,3); Abort1; Abort2

Supongamos que inicialmente el item X tiene un valor igual a 1.
Aquí vemos que la imagen anterior de Write2(X,3) es 2, escrito por T1.
Sin embargo el valor de X, después de que Write2(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-
Si embargo 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 un commit o un abort.
Si pedimos lo mismo con respecto al Read(X) decimos que tenemos una *ejecución estricta*.

3.4. CLASIFICACIÓN DE HISTORIAS SEGÚN RECUPERABILIDAD:

Ahora veamos las definiciones:

3.4.1. Historias Recuperables:

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

O en forma equivalente:

Siempre que Ti lee de Tj ($i \neq j$) en H y $Ci \in H$ y $Cj < Ci$

3.4.2. 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 Ti lee X de Tj ($i \neq j$) en H y $Cj < Ri(X)$

3.4.3. 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 $Wj(X) < Oi(X)$ ($i \neq j$), o $Aj < Oi(X)$ o $Cj < Oi(X)$, donde Oi(X) es Ri(X) o Wi(X)

Ejemplo 4:

Dados:

T1= W(X) W(Y) W(Z) C
T2= R(U) W(X) R(Y) W(Y) C

H7 = W1(X) W1(Y) R2(U) W2(X) R2(Y) W2(Y) **C2** W1(Z) **C1**
H8 = W1(X) W1(Y) R2(U) W2(X) R2(Y) W2(Y) W1(Z) **C1** **C2**
H9 = W1(X) W1(Y) R2(U) W2(X) W1(Z) **C1** R2(Y) W2(Y) **C2**
H10= W1(X) W1(Y) R2(U) W1(Z) **C1** W2(X) R2(Y) W2(Y) **C2**

Vemos que:

H7 no es RC porque T2 lee Y de T1 pero C2 < C1.

H8 es RC pero no es ACA porque T2 lee Y de T1 antes que T1 haga commit.

H9 es ACA pero no es ST porque T2 sobreescribe el valor de X escrito por T1 antes que T1 termine.

H10 es ST.

3.5. 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 ésta son a su vez más restrictivas que las de RC.

3.6. 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, vemos que:

SG(H7) = SG(H8) = SG(H9) = SG(H10) = (T1) ---------- X,Y > (T2)

No hay ciclos y por lo tanto son todas SR.

En la siguiente figura podemos ver el diagrama de Venn correspondiente:

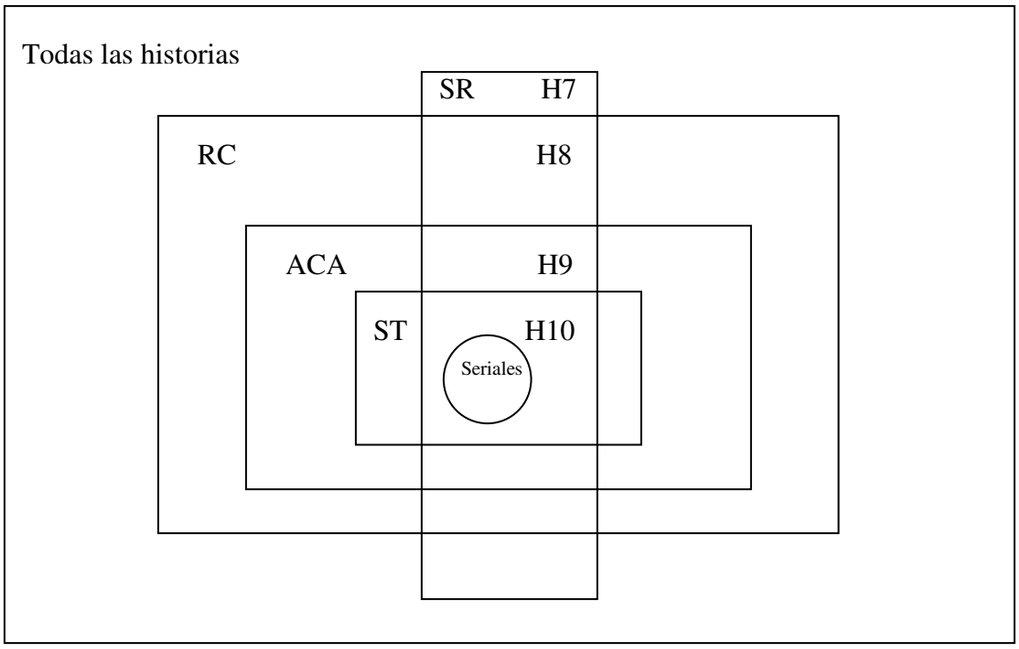


Figura 1.

Ejemplo 6:

Dados:

T1= R(X) W(Y) C
T2= W(Y) W(X) C

H= R1(X) W2(Y) W2(X) C2 W1(Y) C1

X

Hacemos el SG(H) : (T1) ---------- > (T2)
< ----------
Y

y vemos que H no es SR aunque es ST.

3.7. LOCKING Y RECUPERABILIDAD:

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

3.7.1 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.

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*.

BIBLIOGRAFIA:

Algunos de los ejemplos de este apunte fueron tomados de los siguientes libros:

- "Concurrency Control and Recovery in Database Systems", de Philip A. Bernstein, Vassos Hadzilacos y Nathan Goodman, 1987.

- "Introducción a las Bases de Datos Relacionales", de Alberto Mendelzon y Juan Ale, 2000.

- "Principles of Database and Knowledge-Base Systems", Volume I, de Jeffrey Ullman, 1988.