Parcialito 4 - Concurrencia y recuperación - SIN CORRECCIÓN RÁPIDA

● Graded

Student

Juana Rehl

Total Points

9 / 10 pts

Question 1

I.1 **3** / 3 pts

✔ **– 0 pts** Correcto

– 1 pt Hace mal el grafo

– 1 pt lista mal los conflictos

– 1 pt Justifica mal porque es serializable

– 2 pts Se pedia una forma de entrega.

– 1 pt Si es recuperable. RT3(X) leyó lo escrito por WT2(X), entonces T3 depende de T2 y conlleva a que T3 debe commitear después de T2.

– 1 pt Si es recuperable. el solapamiento es recuperable se garantiza que T2 haga COMMIT antes de COMMIT de T3
cT1; cT2; cT3; Ó cT2; cT3; cT1;
RT3(X) leyó lo escrito por WT2(X), entonces T3 depende de T2 y conlleva a que T3 debe commitear después de T2.

– 0.5 pts es recuperable no hay que hacer modificaciones

– 0.5 pts porque es recuperable? justificacion

Question 2

I.2 **3** / 3 pts

✔ **– 0 pts** Correcto

– 0.5 pts Pone mal los recursos en el grafo

– 0.5 pts Mal la cantidad de conflictos es 6

– 0.5 pts Mal el listado de conflictos
T1 → T2: WT1 (X), RT2 (X)
T1 → T2: RT1 (X), WT2 (X)
T1 → T2: WT1 (X), WT2 (X)
T2 → T3: WT2 (Y), RT3 (Y)
T2 → T3: WT2 (Y), WT3 (Y)
T2 → T3: RT2 (Y), WT3 (Y)

– 1 pt Falta justificacion T2 commitea antes que T1 (debe hacerlo despues)
y T3 commitea antes que T2 (debe hacerlo despues).
El orden de commits deberia ser
CT1 ; CT2 ; CT3

– 0.5 pts Falta PARTE de la justificacion T2 commitea antes que T1 (debe hacerlo despues)
y T3 commitea antes que T2 (debe hacerlo despues).
El orden de commits deberia ser
CT1 ; CT2 ; CT3

– 2 pts No respeta formato de entrega

– 0.5 pts Mala justificación es Serializable?
Si, no hay ciclos en el grafo de precedencias

Question 3

I.3 2 / 3 pts

– 0 pts Correcto

✔ **– 1 pt** Vuelve a la linea equivocada.

2 El algoritmo te dice que todo lo que está antes del begin está en disco, y que tenés que volver a la transacción más antigua del begin en caso de tener que deshacer cualquiera que no haya commiteado.

– 1 pt Escribe el abort en la mitad del alogritmo

– 3 pts Hace primero REDO y después UNDO

– 2 pts Resuelve pero no sigue el algoritmo

– 1 pt Le da importancia al BEGIN sin el END

– 3 pts No sigue el algoritmo

– 0.5 pts Escribe mal el log

– 0.5 pts No escribe ABORT(T3)

– 1 pt Asume que todo lo que está antes del END está en disco

– 3 pts Incompleto

1 Si, pero igual tenés que volver a la linea 01.

3 Por último?

4 Si primero se realiza el undo (bien) por qué primero describís el REDO?

Question 4

Por entregar **1** / 1 pt

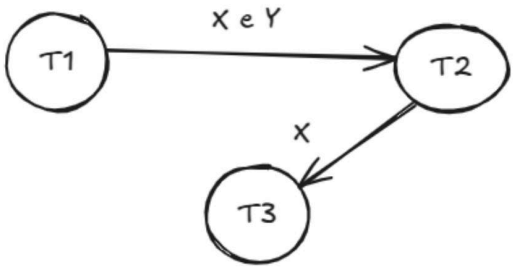
✔ **– 0 pts** Correcto

Question assigned to the following page: 1

Juana Rehl
112185
Parcialito 3 - Diseño relacional

Ejercicio 1

1. bT1; bT2; bT3; RT1(X); WT2(X); RT3(X); WT1(Y); RT2(Y); WT3(Z); cT1; cT2; cT3;

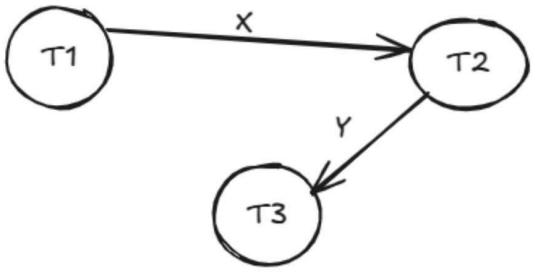
Ejer I.1	
A .Grafo:	
B. Cantidad total de conflictos	3
C. Es Serializable?	Es serializable ya que en el grafo de precedencia no hay presencia de un ciclo, por lo que se pueden serializar las transacciones.
D. Es Recuperable?	Es recuperable porque ninguna transacción lee datos de otra transacción que aún no haya hecho commit.
E. Modificación para hacerlo recuperable	

Lista de conflictos:

{RT1(X);WT2(X)}
{WT2(X);RT3(X)}
{WT1(Y);RT2(Y)}

Question assigned to the following page: 2

2. bT1; bT2; bT3; RT1(X); WT1(X); RT2(X); WT2(X); RT2(Y); WT2(Y); RT3(Y); WT3(Y); cT3; cT2; cT1;

Ejer I.2	
A .Grafo:	
B. Cantidad total de conflictos	6
C. Es Serializable?	Es serializable ya que en el grafo de precedencia no hay presencia de un ciclo, por lo que se pueden serializar las transacciones.
D. Es Recuperable?	No es recuperable ya que T2 y T3 leen datos de transacciones que todavía no han hecho commit.
E. Modificación para hacerlo recuperable	Para que sea recuperable, se debe cambiar el orden de los commits. El nuevo orden sería: cT1;cT2;cT3

Lista de conflictos:
{ RT1(X);WT2(X)}
{WT1(X);RT2(X)}
{RT2(Y);WT3(Y)}
{WT2(Y); RT3(Y)}
{WT1(X);WT2(X)}
{WT2(Y);WT3(Y)}

Question assigned to the following page: 3

Ejercicio 2

Primero se realiza la parte del UNDO, leyendo el log de abajo hacia arriba. En esta etapa hay que encontrar un bloque de checkpoint completo (con su **BEGIN CKPT** y **END CKPT**). Si aparece un **BEGIN CKPT** sin su cierre, se ignora porque no aporta información útil.

En este caso, el bloque válido del checkpoint empieza en la línea 03 y termina en la línea 08, ya que el **BEGIN CKPT** de la línea 10 no tiene su **END CKPT**.

A partir de ese checkpoint, solo se revisan las operaciones posteriores al **BEGIN CKPT** de la línea 03, porque lo anterior ya está guardado en disco. Como las transacciones T1 y T2 ya están commiteadas (líneas 04 y 07), no hace falta deshacerlas.

Luego, en la parte del REDO, se vuelve a ejecutar la línea 06, ya que solo se deben rehacer las operaciones de las transacciones que hicieron commit. En el caso de T1, no es necesario porque sus cambios fueron antes del checkpoint y ya están en disco.

Por último, la transacción T3 nunca llega a commitear, así que hay que deshacer sus modificaciones. Se revierten las operaciones de las líneas 11 y 12, dejando el recurso B con valor 6 y A con valor 4. Esto se logra con el proceso de UNDO, que revisa el log de abajo hacia arriba para detectar las transacciones sin commit y revertir sus efectos. Finalmente, se agrega un ABORT de T3, ya que, como no llegó a hacer commit, debe marcarse como abortada.

Log final:

Nro línea | Log
01 (BEGIN, T1);
02 (WRITE T1, A, 1, 2);
03 (BEGIN CKPT, T1);
04 (COMMIT, T1);
05 (BEGIN, T2);
06 (WRITE T2, A, 2, 4);
07 (COMMIT, T2);
08 (END CKPT);
09 (BEGIN, T3);
10 (BEGIN CKPT, T3);
11 (WRITE T3, A, 4, 3);
12 (WRITE T3, B, 6, 7);
13 (ABORT, T3)