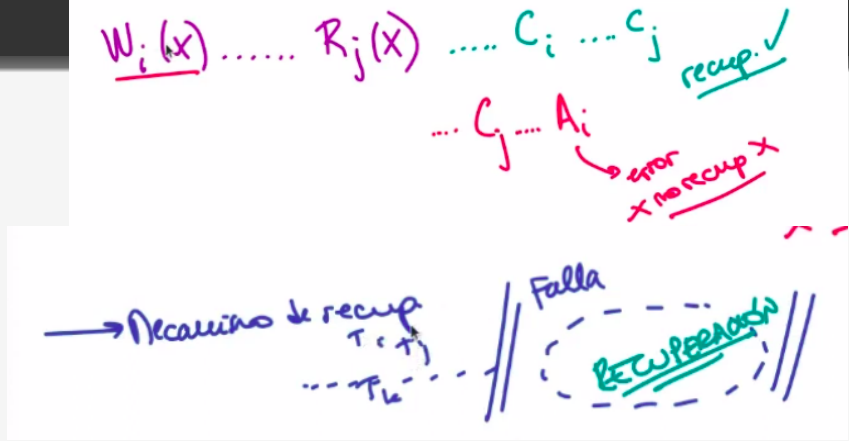
75.15 / 75.28 / 95.05 - Base de Datos



Recuperación

Mariano Beiró

Dpto. de Computación - Facultad de Ingeniería (UBA)

7 de junio de 2022

Topics

Fallas

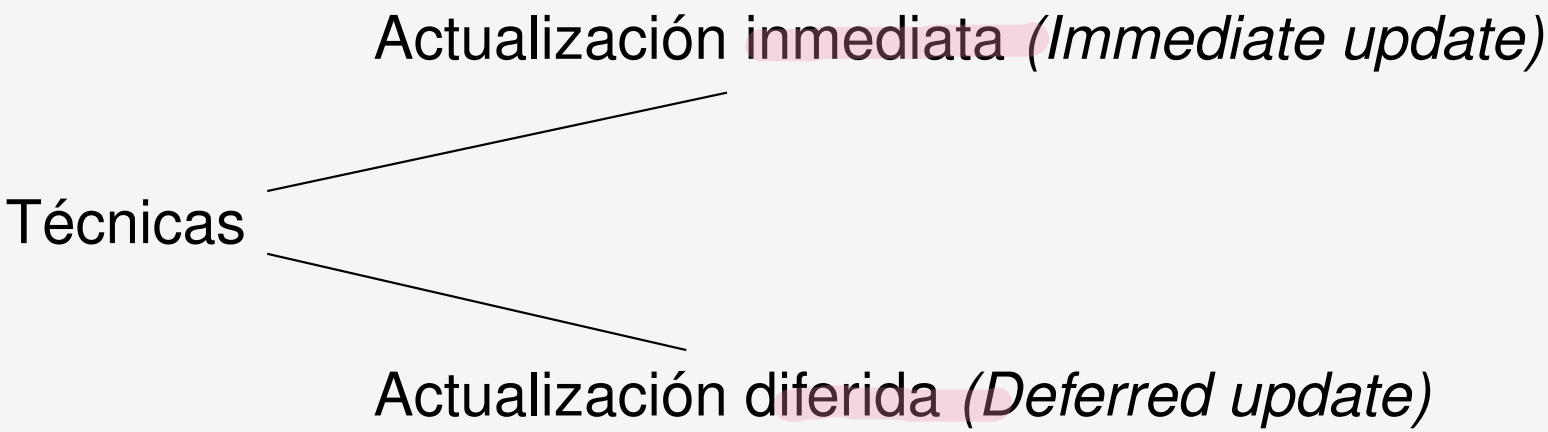
- Los sistemas reales sufren múltiples tipos de fallas:
  1. Fallas de sistema: Por errores de software ó hardware que detienen la ejecución de un programa: fallas de segmentación, división por cero, fallas de memoria.
  2. Fallas de aplicación: Aquellas que provienen desde la aplicación que utiliza la base de datos. Por ejemplo, la cancelación o vuelta atrás de una transacción.
  3. Fallas de dispositivos: Aquellas que provienen de un daño físico en dispositivos como discos rígidos o memoria.
  4. Fallas naturales externas: Son aquellas que provienen desde afuera del hardware en que se ejecuta nuestro SGBD. Ejemplos: caídas de tensión, terremotos, incendios, ...
- En situaciones catastróficas como 3 ó 4, es necesario contar con mecanismos de *backup* para recuperar la información.
- Aquí veremos cómo resolver las pérdidas de datos en memoria (*buffers* de datos, por ejemplo) de manera de garantizar las propiedades ACID, en situaciones no catastróficas como 1 y 2.

Introducción

- Comencemos por considerar que se produce una falla no catastrófica en el momento en que una transacción se está ejecutando.
- En algún momento el sistema se reinicia, y la base de datos deberá ser llevada al estado inmediato anterior al comienzo de la transacción.
- Para ello, es necesario mantener información en el *log* acerca de los cambios que la transacción fue realizando.
- Para cada instrucción de escritura que se ejecuta sobre un ítem:
  - $X \rightarrow$ Buffer en memoria $\rightarrow$ Disco
- Hasta ahora no hemos hablado de en qué momento un buffer en memoria se almacena en disco.

Introducción

Técnicas de volcado (*flush*) a disco



- En los métodos de actualización inmediata, los datos se guardan en disco lo antes posible, y necesariamente *antes del commit de la transacción*.
- En la actualización diferida, los datos se guardan en disco *después del commit de la transacción*.

Gestor de Recuperación

Estructura del *log*

- Recordemos que, para hacer posible la recuperación ante fallas, el gestor de recuperación del SGBD guarda una serie de información en un *log* (bitácora).
- El *log* almacena generalmente los siguientes registros:
  - (BEGIN, T_{id}): Indica que la transacción T_{id} comenzó.
  - (WRITE, T_{id}, X, x_{old}, x_{new}): Indica que la transacción T_{id} escribió el item X, cambiando su viejo valor x_{old} por un nuevo valor x_{new}.
  - (READ, T_{id}, X): Indica que la transacción T_{id} leyó el item X.
  - (COMMIT, T_{id}): Indica que la transacción T_{id} committeó.
  - (ABORT, T_{id}): Indica que la transacción T_{id} abortó.
- En realidad, según el algoritmo de recuperación que utilicemos, no siempre será necesario guardar los valores anteriores y actuales en el WRITE, y no siempre será necesario guardar los READ's.

Reglas WAL y FLC

- El gestor de *logs* se guía por dos reglas básicas:
  - WAL (Write Ahead Log)
  - FLC (Force Log at Commit)
- La regla WAL indica que antes de guardar un ítem modificado en disco, se debe escribir el registro de *log* correspondiente, en disco.
- La regla FLC indica que antes de realizar el *commit* el *log* debe ser volcado a disco.

Algoritmos de recuperación

Hipótesis

[ELM16 22.1.5; GM09 19.1.3]

- A continuación presentaremos 3 algoritmos distintos que permiten recuperar una base de datos ante fallas.
  - Deshacer (UNDO)
  - Rehacer (REDO)
  - Deshacer/Rehacer (UNDO/REDO)
- En los tres se asume que los solapamientos de transacciones son:
  - Recuperables
  - Evitan *rollbacks* en cascada
- Si esta hipótesis no se cumpliera los algoritmos deberán realizar algunos pasos adicionales que no describiremos. En particular, utilizar los registros de lectura $(READ, T_i, X, v)$ del *log*.

Algoritmo UNDO (Immediate update)

Regla de UNDO

- La regla de UNDO es:

> **Regla de UNDO**
>
> Antes de que una modificación sobre un ítem $X \leftarrow v_{new}$ por parte de una transacción no commiteada sea guardada en disco (*flushed*), se debe salvaguardar en el *log* en disco el último valor commiteado v_{old} de ese ítem.

Antes de guardar en disco un valor de una transacción que no commiteó, se salvaguarda en el log en disco el último valor commiteado del ítem.

Antes de escribir en disco un valor nuevo para un ítem de una transacción que no hizo commit, el gestor guarda en el log el valor viejo del ítem.

Algoritmo UNDO (Immediate update)

Procedimiento

[GM09 17.2.2]

- Para cumplir con la regla se utiliza el siguiente procedimiento:
  1. Cuando una transacción T_i modifica el ítem X remplazando un valor v_{old} por v, se escribe $(WRITE, T_i, X, v_{old})$ en el *log*, y se hace *flush* del *log* a disco.
  2. El registro $(WRITE, T_i, X, v_{old})$ debe ser escrito en el *log* en disco (*flushed*) antes de escribir (*flush*) el nuevo valor de X en disco (WAL).
  3. Todo ítem modificado debe ser guardado en disco antes de hacer *commit*.
  4. Cuando T_i hace *commit*, se escribe $(COMMIT, T_i)$ en el *log* y se hace *flush* del *log* a disco (FLC).

Algoritmo UNDO (Immediate update)

Observaciones

- Los tres primeros puntos aseguran que todas las modificaciones realizadas sean escritas a disco antes de que la transacción termine.
- De esta forma, una vez cumplimentado el paso 4, ya nunca será necesario hacer REDO. Si la transacción falla antes ó durante el punto 4, será deshecha (UNDO) al reiniciar.

> Se considera que la transacción commiteó cuando el registro (*COMMIT*, T_i) queda escrito en el *log*, en disco.

Algoritmo UNDO (Immediate update)

Reinicio

[GM09 17.2.3]

- Cuando el sistema reinicia se siguen los siguientes pasos:
  1. Se recorre el *log* de adelante hacia atrás, y por cada transacción de la que no se encuentra el COMMIT se aplica cada uno de los WRITE para restaurar el valor anterior a la misma *en disco*.
  2. Luego, por cada transacción de la que no se encontró el COMMIT se escribe $(ABORT, T)$ en el *log* y se hace *flush* del *log* a disco.
- Obsérvese que también podría ocurrir una falla durante el reinicio. Esto no es un problema porque el procedimiento de reinicio es idempotente: Si se ejecuta más de una vez, no cambiará el resultado.

Algoritmo UNDO (Immediate update)

Ejemplo

Ejemplo

Considere el siguiente solapamiento de transacciones.
Suponga que los valores iniciales de los ítems son $A = 60$, $B = 44$, $C = 38$.

Transacción T_1	Transacción T_2	Transacción T_3
begin		
leer_item(B)		
$B = B + 4$		
escribir_item(B)		
	begin	
	leer_item(A)	
	leer_item(C)	
	$A = A \div 2$	
	$C = C + 10$	
	escribir_item(A)	
	escribir_item(C)	
		begin
commit		
		leer_item(B)
		$B = B + 5$
		escribir_item(B)
	commit	
		leer_item(A)
		$A = A \times 1.10$
		escribir_item(A)
		commit

Algoritmo UNDO (Immediate update)

Ejemplo

Ejercicio 1

Escriba la secuencia de registros de un *log* UNDO (omita los registros de lectura).

Respuesta 1

01 (BEGIN, T1);
02 (WRITE, T1, B, 44);
03 (BEGIN, T2);
04 (WRITE, T2, A, 60);
05 (WRITE, T2, C, 38);
06 (BEGIN, T3);
07 (COMMIT, T1);
08 (WRITE, T3, B, 48);
09 (COMMIT, T2);
10 (WRITE, T3, A, 30);
11 (COMMIT, T3);

Algoritmo UNDO (Immediate update)

Ejercicio

Ejercicio 2

¿Hasta qué momento pueden guardarse los datos modificados por T1 en disco?

Respuesta 2

T1 sólo modifica B. B debe ser guardado en disco antes del *commit* de T1, es decir, antes de escribir ($COMMIT$, $T1$) en el *log* en disco.

Algoritmo UNDO (Immediate update)

Ejercicio

Ejercicio 3

¿Cómo reacciona el sistema ante una falla inmediatamente después del *commit* de T1?

Respuesta 3

Cuando el sistema reinicie, será necesario deshacer (UNDO) T2 y T3, que quedarán abortadas. Para ello se deberá escribir 38 en el ítem C y 60 en el ítem A en disco. Luego se escribe en el *log* (*ABORT*, *T*2) y (*ABORT*, *T*3) y se hace *flush* del *log* a disco.

Algoritmo REDO (Deferred update)

Regla

- La regla de REDO es:

> **Regla de REDO**
>
> Antes de realizar el *commit*, todo nuevo valor *v* asignado por la transacción debe ser salvaguardado en el *log*, en disco.

- ¿Ésto me obliga a guardar el ítem modificado en disco antes de commitear la transacción que lo modificó?
  - ✗ No, sólo el registro de *log*! De hecho, en el algoritmo REDO el ítem es actualizado en disco luego de commitear la transacción.

Algoritmo REDO (Deferred update)

Procedimiento

[GM09 17.3]

- Para cumplir con la regla se utiliza el siguiente procedimiento:
  1. Cuando una transacción T_i modifica el item X remplazando un valor v_{old} por v, se escribe $(WRITE, T_i, X, v)$ en el *log*.
  2. Cuando T_i hace *commit*, se escribe $(COMMIT, T_i)$ en el *log* y se hace *flush* del *log* a disco (FLC). Recién entonces se escribe el nuevo valor en disco.
- Atención: En el punto 1, ahora se escribe el valor nuevo en el log!
- Si la transacción falla antes del *commit*, no será necesario deshacer nada (al reiniciar se abortarán las transacciones no commiteadas). Si en cambio falla después de haber escrito el COMMIT en disco, la transacción será rehecha al iniciar.

> Nuevamente, se considera que la transacción commiteó cuando el registro $(COMMIT, T_i)$ queda escrito en el *log*, en disco.

Algoritmo REDO (Deferred update)

Observaciones

- Observaciones:
  - En el algoritmo REDO, una transacción puede commitear sin haber guardado en disco todos sus ítems modificados.
  - Ante una falla previa posterior al *commit*, entonces, será necesario reescribir (REDO) todos los valores que la transacción había asignado a los ítems.
  - Esto implicará recorrer todo el log de atrás para adelante aplicando cada uno de los WRITE.

Algoritmo REDO (Deferred update)

Reinicio

- Cuando el sistema reinicia se siguen los siguientes pasos:
  1. Se analiza cuáles son las transacciones de las que está registrado el COMMIT.
  2. Se recorre el *log* de atrás hacia adelante volviendo a aplicar cada uno de los WRITE de las transacciones que commitearon, para asegurar que quede actualizado el valor de cada ítem.
  3. Luego, por cada transacción de la que no se encontró el COMMIT se escribe $(ABORT, T)$ en el *log* y se hace *flush* del *log* a disco.

Algoritmo REDO (Deferred update)

Ejemplo

Ejercicio 1

Escriba la secuencia de registros de un *log* REDO para el mismo ejercicio considerado anteriormente (omita los registros de lectura).

Respuesta 1

01 (BEGIN, T1);
02 (WRITE, T1, B, 48);
03 (BEGIN, T2);
04 (WRITE, T2, A, 30);
05 (WRITE, T2, C, 48);
06 (BEGIN, T3);
07 (COMMIT, T1);
08 (WRITE, T3, B, 53);
09 (COMMIT, T2);
10 (WRITE, T3, A, 33);
11 (COMMIT, T3);

Algoritmo REDO (Deferred update)

Ejercicio

Ejercicio 2

¿Cómo reacciona el sistema ante una falla después del *commit* de T1?

Respuesta 2

Cuando el sistema reinicie, será necesario rehacer (REDO) T1. Para ello se deberá escribir 48 en el ítem B. Las transacciones T2 y T3 no tienen su COMMIT hecho, por lo tanto se escribe en el *log* (*ABORT*, *T*2) y (*ABORT*, *T*3) y se hace *flush* del *log* a disco.

Algoritmos de Recuperación

Algoritmo UNDO/REDO

[GM09 17.4.1]

- En el algoritmo UNDO/REDO es necesario cumplir con ambas reglas a la vez. El procedimiento es el siguiente:
  1. Cuando una transacción T_i modifica el item X remplazando un valor v_{old} por v, se escribe $(WRITE, T_i, X, v_{old}, v)$ en el *log*.
  2. El registro $(WRITE, T_i, X, v_{old}, v)$ debe ser escrito en el *log* en disco (*flushed*) antes de escribir (*flush*) el nuevo valor de X en disco.
  3. Cuando T_i hace *commit*, se escribe $(COMMIT, T_i)$ en el *log* y se hace *flush* del *log* a disco.
  4. Los ítems modificados pueden ser guardados en disco antes o después de hacer *commit*.

Algoritmo UNDO/REDO

Reinicio

[GM09 17.4.2]

- Cuando el sistema reinicia se siguen los siguientes pasos:
  1. Se recorre el *log* de adelante hacia atrás, y por cada transacción de la que no se encuentra el COMMIT se aplica cada uno de los WRITE para restaurar el valor anterior a la misma *en disco*.
  2. Luego se recorre de atrás hacia adelante volviendo a aplicar cada uno de los WRITE de las transacciones que commitearon, para asegurar que quede asignado el nuevo valor de cada ítem.
  3. Finalmente, por cada transacción de la que no se encontró el COMMIT se escribe $(ABORT, T)$ en el *log* y se hace *flush* del *log* a disco.

Algoritmo UNDO/REDO

Ejemplo

Ejercicio 1

Para la siguiente secuencia de registros de *log*, indique qué items deben/pueden haber cambiado su valor en disco. Luego aplique el algoritmo de recuperación UNDO/REDO e indique cómo queda el archivo de *log*.

01 (BEGIN, T1);
02 (WRITE, T1, A, 10, 15);
03 (BEGIN, T2);
04 (WRITE, T2, B, 30, 25);
05 (WRITE, T1, C, 35, 32);
06 (WRITE, T2, D, 14, 12);
07 (COMMIT, T2);

Algoritmo UNDO/REDO

Ejercicio

Respuesta

Todos los ítems pueden haber cambiado su valor en disco, pero no necesariamente deben haberlo cambiado.
Al aplicar UNDO/REDO deberemos abortar T_1. Para ello, en la fase de UNDO debemos reescribir el valor 35 en C y el valor 10 en A, en disco. Luego, en la fase de REDO debemos reescribir 25 en B y 12 en D. Por último, debemos agregar al *log* la línea $(ABORT, T1)$ y hacer *flush* del *log* a disco.

Puntos de control (checkpoints)

- Cuando reiniciamos el sistema no sabemos hasta donde tenemos que retroceder en el archivo de *log*. Aunque muchas transacciones antiguas ya commiteadas seguramente tendrán sus datos guardados ya en disco.
- Para evitar este retroceso hasta el inicio del sistema y el crecimiento ilimitado de los archivos de *log* se utilizan puntos de control (checkpoints).
- Un *punto de control (checkpoint)* es una registro especial en el archivo de *log* que indica que indica que todos los ítems modificados hasta ese punto han sido almacenados en disco.
- La presencia de un *checkpoint* en el *log* implica que todas las transacciones cuyo registro de *commit* aparece con anterioridad tienen todos sus ítems guardados en forma persistente, y por lo tanto ya no deberán ser deshechas ni rehechas.

Puntos de control (checkpoints)

Checkpoints activos vs. inactivos

- Los checkpoints inactivos (quiescent checkpoints) tienen un único tipo de registro: (CKPT).
- La creación de un *checkpoint inactivo* en el *log* implica la suspensión momentánea de todas las transacciones para hacer el volcado *(flush)* de todos los *buffers* en memoria al disco.
- Para aminorar la pérdida de tiempo de ejecución en el volcado a disco puede utilizarse una técnica conocida como checkpointing activo (non-quiescent o fuzzy checkpointing), que utiliza dos tipos de registros de *checkpoint*: *(BEGIN CKPT,* t_{act}*)* y *(END CKPT)*, en donde t_{act} es un listado de todas las transacciones que se encuentran activas (es decir, que aún no hicieron *commit*). El procedimiento varía según cada algoritmo de recuperación.

Algoritmo UNDO

Checkpointing inactivo

[GM09 17.2.4]

- En el algoritmo UNDO, el procedimiento de *checkpointing inactivo* se realiza de la siguiente manera:
  1. Dejar de aceptar nuevas transacciones.
  2. Esperar a que todas las transacciones hagan su *commit* (es decir, escriban su registro de COMMIT en el *log* y lo vuelquen a disco).
  3. Escribir (CKPT) en el *log* y volcarlo a disco.
- Si el sistema cae justo después de escribir (CKPT) en el *log*, ¿es posible que alguno de los ítems modificados por alguna transacción no hayan sido guardados a disco?
  - No, porque en el algoritmo UNDO la presencia del registro de COMMIT en el *log* implica que todos los ítems fueron ya salvaguardados en disco.
- Durante la recuperación, sólo debemos deshacer las transacciones que no hayan hecho *commit*, hasta el momento en que encontremos un registro de tipo (CKPT). De hecho, todo el archivo de *log* anterior al *checkpoint* podía ser eliminado.

Algoritmo UNDO

Checkpointing activo

[GM09 17.2.5]

- En la versión activa para el algoritmo UNDO, el procedimiento es el siguiente:
  1. Escribir un registro (BEGIN CKPT, t_{act}) con el listado de todas las transacciones activas hasta el momento.
  2. Esperar a que todas esas transacciones activas hagan su *commit* (sin dejar por eso de recibir nuevas transacciones)
  3. Escribir (END CKPT) en el *log* y volcarlo a disco.
- En la recuperación, al hacer el *rollback* se dan dos situaciones:
  - Que encontremos primero un registro (END CKPT). En ese caso, sólo debemos retroceder hasta el (BEGIN CKPT) durante el *rollback*, porque ninguna transacción incompleta puede haber comenzado antes).
  - Que encontremos primero un registro (BEGIN CKPT). Esto implica que el sistema cayó sin asegurar los *commits* del listado de transacciones. Deberemos volver hacia atrás, pero sólo hasta el inicio de la más antigua del listado.

Algoritmo UNDO

Checkpointing activo

Ejemplo

Considere la siguiente secuencia de registros de un *log* UNDO con *checkpointing* activo. El sistema falla después de loguear el último de ellos en disco.

01 (BEGIN, T1);
02 (WRITE, T1, X, 50);
03 (BEGIN, T2);
04 (WRITE, T1, Y, 15);
05 (WRITE, T2, X, 8);
06 (BEGIN, T3);
07 (WRITE, T3, Z, 3);
08 (COMMIT, T1);
09 (BEGIN CKPT, T2, T3);
10 (WRITE, T2, X, 7);
11 (WRITE, T3, Y, 4);

Algoritmo UNDO

Checkpointing activo

Ejercicio 1

¿Hasta qué línea será necesario volver atrás?

Respuesta 1

Hasta la línea 03.

Ejercicio 2

Indique cómo será el procedimiento de recuperación.

Respuesta 2

Es necesario deshacer las transacciones 2 y 3. Debemos escribir 3 en el ítem Z, 8 en el ítem X y 4 en el ítem Y, en disco. Finalmente agregamos (ABORT, T2) y (ABORT, T3) al *log*, y lo volcamos a disco.

Algoritmo REDO

Checkpointing activo

[GM09 17.3.3 17.3.4]

- En el algoritmo REDO con *checkpointing* activo se procede de la siguiente forma:
  1. Escribir un registro (BEGIN CKPT, t_{act}) con el listado de todas las transacciones activas hasta el momento y volcar el log a disco.
  2. Hacer el volcado a disco de todos los ítems que hayan sido modificados por transacciones que ya commitearon.
  3. Escribir (END CKPT) en el *log* y volcarlo a disco.
- En la recuperación hay nuevamente dos situaciones:
  - Que encontremos primero un registro (END CKPT).En ese caso, deberemos retroceder hasta el (BEGIN , T_x) más antiguo del listado que figure en el (BEGIN CKPT) para rehacer todas las transacciones que commitearon. Escribir (ABORT, T_y) para aquellas que no hayan commiteado.
  - Que encontremos primero un registro (BEGIN CKPT).Si el *checkpoint* llego sólo hasta este punto no nos sirve, y entonces deberemos ir a buscar un *checkpoint* anterior en el *log*.

Algoritmo REDO

Checkpointing activo

Ejemplo

Considere la siguiente secuencia de registros de un *log* REDO con *checkpointing* activo. El sistema falla después de loguear el último de ellos en disco.

```
01 (BEGIN, T1);
02 (WRITE, T1, A, 10);
03 (BEGIN, T2);
04 (WRITE, T2, B, 5);
05 (WRITE, T1, C, 7);
06 (BEGIN, T3);
07 (WRITE, T3, D, 8);
08 (COMMIT, T1);
09 (BEGIN CKPT, ....);
10 (BEGIN, T4);
11 (WRITE, T2, E, 5);
12 (COMMIT, T2);
13 (WRITE, T3, F, 7);
14 (WRITE, T4, G, 15);
15 (END CKPT);
16 (COMMIT, T3);
17 (BEGIN, T5);
18 (WRITE, T5, H, 20);
19 (BEGIN CKPT, ....);
20 (COMMIT, T5);
```

Algoritmo REDO

Ejercicio 1

Complete los listados de transacciones en los (BEGIN CKPT).

Respuesta 1

01 (BEGIN, T1);
02 (WRITE, T1, A, 10);
03 (BEGIN, T2);
04 (WRITE, T2, B, 5);
05 (WRITE, T1, C, 7);
06 (BEGIN, T3);
07 (WRITE, T3, D, 8);
08 (COMMIT, T1);
09 (BEGIN CKPT, T2, T3);
10 (BEGIN, T4);
11 (WRITE, T2, E, 5);
12 (COMMIT, T2);
13 (WRITE, T3, F, 7);
14 (WRITE, T4, G, 15);
15 (END CKPT);
16 (COMMIT, T3);
17 (BEGIN, T5);
18 (WRITE, T5, H, 20);
19 (BEGIN CKPT, T4, T5);
20 (COMMIT, T5);

Algoritmo REDO

Checkpointing activo

Ejercicio 2

¿Hasta qué línea será necesario volver atrás?

Respuesta 2

Hasta la línea 03, en que comienza la transacción 2.

Ejercicio 3

Indique cómo será el procedimiento de recuperación.

Respuesta 3

Es necesario rehacer las transacciones 2, 3 y 5 (que son las que commitearon) desde la línea 03. Entonces, asignamos $B = 5$, $D = 8$, $E = 5$, $F = 7$, $H = 20$. Finalmente agregamos (ABORT, T4) al *log*.

Algoritmo UNDO/REDO

Checkpointing activo

[GM09 17.4.3]

- En el algoritmo UNDO/REDO con *checkpointing* activo el procedimiento es:
  1. Escribir un registro (BEGIN CKPT, t_{act}) con el listado de todas las transacciones activas hasta el momento y volcar el log a disco.
  2. Hacer el volcado a disco de todos los ítems que hayan sido modificados antes del (BEGIN CKPT).
  3. Escribir (END CKPT) en el *log* y volcarlo a disco.
- En la recuperación es posible que debamos retroceder hasta el inicio de la transacción más antigua en el listado de transacciones, para deshacerla en caso de que no haya commiteado.

Algoritmo UNDO/REDO

Checkpointing activo

Ejemplo

Considere la siguiente secuencia de registros de un *log* UNDO/REDO con *checkpointing* activo.

01 (BEGIN, T1);
02 (WRITE, T1, A, 60, 61);
03 (COMMIT, T1);
04 (BEGIN, T2);
05 (WRITE, T2, A, 61, 62);
06 (BEGIN, T3);
07 (WRITE, T3, B, 20, 21);
08 (WRITE, T2, C, 30, 31);
09 (BEGIN, T4);
10 (WRITE, T3, D, 40, 41);
11 (WRITE, T4, F, 70, 71);
12 (COMMIT, T3);
13 (WRITE, T2, E, 50, 51);
14 (COMMIT, T2);
15 (WRITE, T4, B, 21, 22);
16 (COMMIT, T4);

Algoritmo UNDO/REDO

Checkpointing activo

Ejercicio 1

Suponga que se agrega un registro (BEGIN CKPT, T1) justo después de la línea 02. ¿En qué posición del listado podría escribirse el registro (END CKPT)?

Respuesta 1

El registro (END CKPT) podría escribirse en cualquier posición después del (BEGIN CKPT, T1), siempre que ya se hayan guardado a disco todos los ítems modificados con anterioridad al (BEGIN CKPT).

Algoritmo UNDO/REDO

Checkpointing activo

Ejercicio 2

Con ese *checkpoint* iniciado, hasta donde deberemos retroceder si se reinicia el sistema después de escribir en el *log* la línea (WRITE, T2, A, 61, 62)? Describa el procedimiento de reinicio.

Respuesta 2

Deberemos retroceder hasta el (BEGIN, T1). Hay que hacer el UNDO de la transacción $T2$ y el REDO de la transacción $T1$. Debemos entonces asignar $A = 61$. Luego debemos escribir (ABORT, T2).

Checkpointing activo

Síntesis

- En resumen:
  - En el algoritmo UNDO, escribimos el (END CKPT) cuando todas las transacciones del listado de transacciones activas hayan hecho *commit*.
  - Para el algoritmo REDO, escribimos (END CKPT) cuando todos los ítems modificados por transacciones que ya habían commiteado al momento del (BEGIN CKPT) hayan sido salvaguardados en disco.
  - En el UNDO/REDO escribimos (END CKPT) cuando todos los ítems modificados antes del (BEGIN CKPT) hayan sido guardados en disco.

Bibliografía

[GM09] Database Systems, The Complete Book, 2nd Edition.
H. García-Molina, J. Ullman, J. Widom, 2009.
Capítulo 17.
El contenido de clase (en particular el uso de *checkpoints*), está basado en este libro.

[ELM16] Fundamentals of Database Systems, 7th Edition.
R. Elmasri, S. Navathe, 2016.
Capítulo 22.

[SILB19] Database System Concepts, 7th Edition.
A. Silberschatz, H. Korth, S. Sudarshan, 2019.
Capítulo 19

[CONN15] Database Systems, a Practical Approach to Design, Implementation and Management, 6th Edition.
T. Connolly, C. Begg, 2015.
Capítulo 22 (Sección 3) Cubre muy brevemente el tema.