RECUPERACION

RECUPERABILIDAD

¿Cómo garantizamos la durabilidad de ACID? Podríamos escribir en disco cada commit. Ahora, que asa si en el medio de esa escritura a disco (ponele q tenia que escribir 4 cosas), escribí dos y se me cortó la luz. Ahí perdemos la atomicidad; la trx se ejecutó por la mitad.

La serializabilidad nos asegura el *aislamiento.* Ahora queremos asegurar que una vez que una transacción commiteó, la misma no deba ser deshecha. Esto nos ayudará a implementar la propiedad de ***durabilidad.***

Un solapamiento es recuperable si y solo si ninguna transacción T realiza el commit hasta tanto todas las transacciones que escribieron datos antes de que T los leyera hayan commiteado.

Que pasa si hay una T1 que escribe un dato. La T2 lee ese dato, modifica algo en base a esa lectura, y comittea. Si esta trx hace rollback, el dato que había leído la que comitteo, no es correcto, y yo también debería deshacer ese dato, pero ya committee; ya le dije al que hizo la transacción que estaba todo Ok y no es así.

Por eso queremos que cuando se haga el commit, si leyó datos de otra transacción, esa otra transacción también haya commiteado.

Voy a ver, ante cada commit, si las cosas que leyó cada transacción, que alguna otra escribió, ya haya commiteado. Tiene que comitear antes del commit de la que lee. No es que tiene que commitear antes de la lectura.

- Ejemplos:
  - $b_{T_1}; r_{T_1}(X); b_{T_2}; r_{T_2}(X); w_{T_1}(X); r_{T_1}(Y); w_{T_2}(X); c_{T_2}; w_{T_1}(Y); C_{T_1};$
  - ¿Es recuperable? → ✓
  - ¿Es serializable? → ✗

Para ver si es serializable, vamos a ver todas las escrituras, y en base a todas las escrituras de cada item, una transacción debe venir antes que otra. Armamos el grafo.

Por el Wt1, T2 debe venir antes que T1, y por el Wt2, T1 debe venir antes que T2. Tengo un grafo ciclico, así que no es serializable.

- $b_{T_1}; r_{T_1}(X); b_{T_2}; w_{T_1}(X); r_{T_2}(X); r_{T_1}(Y); w_{T_2}(X); ...$
- Si T_2 commitea ahora, ¿el solapamiento es recuperable? → ✗
- (Si T_1 luego aborta, T_2 va a haber escrito en un forma persistente datos inválidos.)

Si el T2 commitea al final. ¿Es recuperable? Tengo que ver si leyó algo que alguna otra escribió y esa otra no commiteo.

En esta segunda secuencia el solapamiento no es recuperable porque T2 esta leyendo X, que previamente fue escrita por T1, y T1 no commiteo.

Recuperable no implica serializable, y serializable no implica recuperable.

- ¿Recuperable => Serializable? **Falso**
- ¿Serializable => Recuperable? **Falso.** Un solapamiento serializable podría tener un conflicto WR

Gestor de recuperación

Dado un solapamiento recuperable, puede ser necesario **abortar** una transacción antes de que sea commiteada. Para eso el SGBD necesita que su *gestor de recuperación* guarde data en el log.

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 ítem 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 ítem X[2].
- (COMMIT, T_{id}): Indica que la transacción T_{id} committeó.
- (ABORT, T_{id}): Indica que la transacción T_{id} abortó.

Los valores viejos de cada ítem almacenados en los registros WRITE del *log* son los que permitirán deshacer los efectos de la transacción en el momento de hacer el ***rollback***

Rollback

Un SGBD no debería jamás permitir la ejecución de un solapamiento que **no sea recuperable.**

Transacción T_1	Transacción T_2
begin	
	begin
leer_item(A)	
$A = A + 100$	
escribir_item(A)	
	leer_item(A)
	$A = A + 200$
	escribir_item(A)
	commit
abort	

Acá cuando abortamos, estamos poniendo el valor original de A, estamos perdiendo la actualización de la 2da transacción, pero yo le dije que había salido todo bien.

A pesar de que el solapamiento es serializable, el SGBD no podrá deshacer los efectos de la transacción T1 sin eliminar también los efectos de T2, que ya commiteó. **Este tipo de solapamientos no debe producirse**

Tenemos que lograr deshacer los efectos de una transacción Tj que hay que abortar sin afectar la serializabilidad de las transacciones restantes

- Si las modificaciones hechas por Tj no fueron leídas por nadie, entonces basta con procesar el log de Tj en forma inversa para deshacer sus efectos (rollback).
- Si una transacción Ti leyó un dato modificado por Tj, entonces será necesario hacer el rollback de Ti para volverla a ejecutar. Sería conveniente que Ti no hubiera commiteado aún

Si un solapamiento de transacciones es recuperable, entonces nunca será necesario deshacer transacciones que ya hayan commiteado.

Aun así puede ser necesario deshacer transacciones que no aún no han commiteado, y puede que esto produzca una ***casacada de rollbacks.***

Casacada de rollbacks

Que un solapamiento sea recuperable, no implica que no sea necesario tener que hacer *rollbacks* en cascada de transacciones que aún no commitearon

- Ejemplo:
  - $b_{T_1}; r_{T_1}(X); b_{T_2}; w_{T_1}(X); r_{T_2}(X); r_{T_1}(Y); w_{T_2}(X); w_{T_1}(Y); C_{T_1}; c_{T_2};$
  - ¿Es recuperable? → ✓
  - ¿Es serializable? → ✓
  - Pero, ¿qué sucede si T_1 aborta después de $w_{T_2}(X)$?

Transacción T_1	Transacción T_2
begin	
leer_item(X)	
	begin
escribir_item(X)	
	leer_item(X)
leer_item(Y)	
	escribir_item(X)
escribir_item(Y)	
commit	
	commit

  - Como T_2 lee un ítem que T_1 había modificado, la lectura que hace T_2 ya no es válida. Entonces, T_2 deberá ser abortada en cascada.

Si aborta T1 ahí, habría que abortar T2 también, porque aborto T1, que había escrito el Item X que lo leyó T2. Tenemos un rollback en cascada. **La serialización no evita los rollbacks en cascada**

Para evitar los rollbacks en cascada es necesario que una transacción no lea valores que aún no fueron commiteados. Esto es más fuerte que la condición de recuperabilidad.

Esta definición implica que quedan prohibidos los conflictos de la forma sin que en el medio exista un commit c_{Ti}.

Se evita entonces la anomalía de la lectura sucia

¿Evita *rollbacks* en cascada ⇒ Recuperable?
- ✓ Verdadero

¿Evita *rollbacks* en cascada ⇒ Serializable?
- ✗ Falso

Esta definición no cubre la ***anomalía de actualización perdida***

Ejemplo

Transacción T_1	Transacción T_2
begin	
leer_item(X)	
	begin
	leer_item(X)
escribir_item(X)	
leer_item(Y)	
	escribir_item(X)
abort	

Es recuperable: SI, Evita rollbacks en cascada: SI, Es serializable: NO

Protocolo de lock de dos fases estricto (S2PL)

Los locks también pueden ayudar a asegurar la recuperabilidad:

Protocolo de lock de dos fases estricto (S2PL)

Una transacción no puede adquirir un *lock* luego de haber liberado un *lock* que había adquirido, y además los *locks* de escritura sólo pueden ser liberados después de haber commiteado la transacción.

Si solo puedo liberar los locks de escritura luego de commitear, cuando alguien lea algo que otra escribió, esa que escribió ya commiteo, entonces evito el problema de la recuperabilidad.

En caso de no diferenciar tipos de *locks,* se convierte en ***riguroso:***

Protocolo de lock de dos fases riguroso (R2PL)

Los *locks* sólo pueden ser liberados después del commit.

S2PL y R2PL garantizan que todo solapamiento sea no solo serializable, sino también recuperable, y que no se producirán cascadas de rollbacks al deshacer una transacción.

Control de concurrencia basado en timestamps

En el método de control de concurrencia basado en timestamps, cuando se aborta una transacción Ti, cualquier transacción que haya usado datos que Ti modificó debe ser abortada en cascada

- Para garantizar la recuperabilidad se puede escoger entre varias opciones:
  - (a) No hacer el *commit* de una transacción hasta que todas aquellas transacciones que modificaron datos que ella leyó hayan hecho su *commit*. Esto garantiza recuperabilidad.
  - (b) Bloquear a la transacción lectora hasta tanto la escritora haya hecho su *commit*. Esto evita rollbacks en cascada.
  - (c) Hacer todas las escrituras durante el commit, manteniendo una copia paralela de cada ítem para cada transacción. Para esto, la escritura de los ítems en el *commit* deberá estar centralizada y ser atómica.

RECUPERACION

¿Qué pasa si tengo que abortar una transacción de varias etapas por la mitad y yo había escrito a disco? Para este punto asumimos que ya estamos trabajando con un motor que garantiza la recuperabilidad, y el aislamiento

Puedo tener varios tipos de falla: falla de sistema (se detiene la ejecución del programa), fallas de aplicaciones (vienen desde la aplicación que usa la bbd, como por ej la cancelación de una transacción), falla de dispositivo (daño físico en disco rígido), o fallas naturales (vienen de afuera, como un corte de luz)

Para las últimas dos fallas en general se emplean técnicas de backup y restorear lo perdido. Vamos a trabajar más con las 1ras dos fallas, y que pasa con datos que fueron grabados parcialmente.

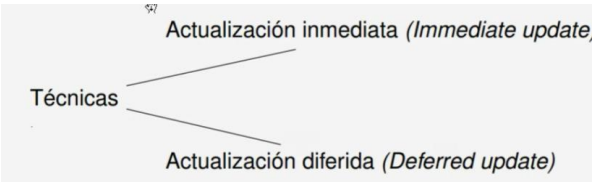
Fallas no catastróficas

En algún momento el sistema se reinicia, y la base de datos deberá ser llevada al estado inmediato anterior al comienzo de la transacción.

Yo tengo que saber que cambios hizo la transacción, porque voy a tener que volver atrás. Para ello en general se tiene un ***log,*** que es básicamente un archivo con cierto formato donde grabamos los datos.

Para cada instrucción de escritura que se ejecuta sobre un item: X -> buffer en memoria -> Disco. Lo guardamos rápido en memo y después en disco porque es costoso guardar en disco.

Tenemos que fijarnos bien en ***qué momento el dato que está en el buffer de memoria se guarda en disco.*** Hay dos técnicas de volcado en disco.



Para la 1ra, los datos se guardan en disco lo antes posible, y necesariamente ***antes del commit de la transacción.*** Osea voy a estar guardando constantemente cada vez que se hace un write.

Para la 2da: los datos se guardan en disco después del commit de la transacción.

Gestor de recuperación: Reglas WAL y FLC

Tenemos dos reglas importantes para manejar.

Yo tengo en el log un registro de cada operación que se da en la base por cada transacción. El log almacena todas las acciones posibles:

- 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.

Reglas WAL y FLC

REGLA WAL (write ahead log): indica que antes de guardar un ítem modificado en disco se debe escribir el registro de log correspondiente, EN DISCO.

REGLA FLC (force log at commit): indica que antes de realizar el commit el log debe ser volcado a disco.

ALGORITMOS DE RECUPERACION

En los 3 algoritmos que vamos a ver **se asume que los solapamientos de transacciones son:**

- Recuperables
- Evitan rollbacks en cascada

Si estos no se cumplieran deberíamos empezar a usar los registros de lectura del log (READ) y eso nos complejiza un poco. **Cuando escribamos el log no vamos a tener en cuenta los READ entonces,** para simplificar las cosas.

Además, según el algoritmo que usemos, no siempre es necesario guardar en el WRITE los valores viejos y los nuevos de los ítems. A veces es uno, a veces el otro y a veces los dos.

UNDO (immediate update)

> 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.

Si una trx me guarda un valor nuevo, debo grabar un Write de la transacción, con el valor viejo. **Siempre lo hacemos sobre transacciones no commiteadas.**

Para garantizar esta regla, usamos este 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).

El paso dos sirve para hacer el UNDO en caso de falla, así puedo volver para atrás y tener registro del valor anterior.

Los 3 primeros pasos aseguran que todas las modificaciones realizadas sean escritas a disco antes de que la transacción termine.

Así, una vez cumplimentado el paso 4, ya nunca será necesario hacer REDO. Osea, **si committe**, ***no hago redo.*** Si la transacción falla antes o durante el punto 4, será deshecha (UNDO) al reiniciar porque no fue committeada.

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

Reinicio ante una falla

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.

Voy yendo de lo último hasta el principio, buscando las transacciones que no hayan comitteado, y voy haciendo undo de todos los cambios que hicieron esas transacciones. SOLO analizo las que no commitearon; agarro el write que hicieron y los piso.

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. Para registrar que estas transacciones abortaron

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.

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

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

```
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);
```

el leer item B lo salteo porque dije que no miro los Read. Sí marco los begin de las transacciones, y los WRITE

OJO: cuando escribo en la T3, el valor de B ya fue modificado. Arranco siendo 44, le sume 4 y committe antes de escribir el item B en la T3, asi que arranca siendo 48.

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

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.

3. ¿Como reacciona el sistema ante una falla inmediata después del commit de T1?

Miro del COMMIT de T1 para arriba. Lo de abajo no existe. Hasta acá, el sistema sabe que comitteo T1 y que T2 y T3 no. ¿Como lo sabe? Leyendo desde la última entrada hasta la 1ra.

Entonces, como T1 sí hizo COMMIT, solo tengo que deshacer los cambios de T2 y T3, y luego marcar el ABORT de estas dos.

T3 solo tiene un begin; no hizo nada asi q no hace falta cambiar nada. T2 hizo cambios y NO comitteo, tengo que deshacer lo que hizo. Por T2, piso C con 38, y A con 60. El Write de T1 NO tengo que volverlo atrás porque si comitte.

Luego se escribe en el log (ABORT,T2) y (ABORT,T3) y se hace flush del log a disco.

REDO (Deferred Update)

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

Acá UNICAMENTE guardo el valor nuevo. Lo tengo que grabar antes del commit, y debo actualizar el disco con el valor nuevo después del commit. **El log, antes del commit, el disco después**.

Esto NO me obliga a guardar el ítem modificado en disco antes de commitear la transacción que lo modificó. ¡SOLO el registro de log! De hecho, en el algoritmo REDO el ítem es actualizado en disco luego de commitear la transacción.

Si una transacción committeo, quizás fue a disco o quizás no

- 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!

Con $v = v_{new}$

Si la transacción falla antes del commit, no será necesario deshacer nada, porque lo que falló nunca se mandó a disco (al reiniciar se abortarán las transacciones no commiteadas). Entonces si NO commitearon no realizo cambios, solo aborto **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.

- 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.

Cuando el sistema reinicia se siguen los siguientes pasos:

1. Se analiza cuáles son las transacciones de las que está registrado el COMMIT. **Las que SI commitearon.** Esto implica leer todo el log. La lectura puede ser de atrás hacia adelante o al revés.
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.

Ejemplo

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

```
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);
```

Básicamente los begin, write y commit son lo mismo, con el mismo orden pero lo que cambia es lo que guardo en el valor de WRITE. Ahora me quedo con el valor actualizado, no con el anterior.

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

De abajo para arriba, separo las Transacciones que comittearon de las que no. Acá solo committeo T1. Ahora, tengo que hacer el redo de estas que sí commitearon.

Cuando el sistema reinicie, será necesario rehacer (REDO) de 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,T2) y (ABORT,T3) y se hace flush del log a disco.

UNDO/REDO

- 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*.

Osea debo registrar tanto el valor viejo como el valor nuevo y puedo pasar los datos a disco **en cualquier momento**.

Esto me da la flexibilidad de hacer ambas operaciones. Tanto UNDO como REDO.

Cuando me encuentro con una transacción que NO committeo, no se si se llegó a grabar a disco o no, porque eso (punto 4), lo puedo hacer en cualquier momento. Entonces, por las dudas, la deshago (UNDO).

Cuando me encuentro con una transacción que SI committeo, tampoco se si llego a grabar sus datos o no, entonces por las dudas las rehago (REDO)

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.

1. UNDO de las que tienen commit
2. REDO de las que si commitearon
3. ABORT de las transacciones de las que no se encontró commit

Ejemplo

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);
```

Todos los ítems ***pueden*** haber cambiado su valor en disco, porque no me limita cuando puedo grabar, pero no necesariamente deben haberlo cambiado.

Primero aplico en UNDO *(de abajo para arriba)* ***y después el REDO*** (de arriba para abajo)

Voy a leer de la más vieja a la más nueva. Para las que no comittearon, pongo el valor nuevo.

Para las que si commitearon leo de arriba para abajo y actualizo los valores. Finalmente agrego los ABORT

Al aplicar UNDO/REDO deberemos abortar T1. 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

¿Qué pasa cuando una BDD esta activa por mucho tiempo? Yo no voy a estar pudiendo leer un Log gigante, de muchas gigas.

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 un registro especial en el archivo de log 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.**

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 ni abort). Entre el begin y el end tengo todas las transacciones que están activas. Todo el resto ya se grabaron

El procedimiento varía según cada algoritmo de recuperación.

Checkpoint inactivo para UNDO

- 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.

Checkpoint activo para UNDO

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.

Cuando todas las activas hicieron commit, hago el log.

En la recuperación, al hacer el rollback se dan dos situaciones:

a. Que encontremos primero un registro (END CKPT). En ese caso, sólo debemos retroceder hasta el primer (BEGIN CKPT) durante el rollback, porque ninguna transacción incompleta puede haber comenzado antes). Todas las activas, si o si nacieron después de begin checkopoint. **Solo hago UNDO de las que están hacia atrás del BEGIN CKPT que no comittearon (osea leo para arriba), y también después del END CKPT.**

b. 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. Es decir, voy hasta el inicio de la transacción más antigua. Osea, el begin de la transacción más antigua que esta entre las activas. Luego se hace el UNDO normal

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);
```

1. ¿Hasta qué línea es necesario volver atrás? Hasta la 3

Encontramos un BEGIN CKPT, asi que tengo el 2do caso, solo voy a volver hasta el inicio de las mas antiguas entre T2 y T3 (las activas, xq están en el BEGIN). La mas antigua es la T2, asi que solo vuelvo hasta la linea 3.

2. Indique como será el proceso de recuperación

Es necesario deshacer las transacciones 2 y 3. Debemos escribir 3 en el ítem Z, 8 en el ítem X (me tengo que quedar con el valor mas viejo de todos) y 4 en el ítem Y, en disco. Finalmente agregamos (ABORT, T2) y (ABORT, T3) al log, y lo volcamos a disco.

Checkpoint activo para REDO

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.

Para el REDO, **el END CKPT se hace cuando se flushearon a disco todas las Tx que commitearon ANTES del BEGIN CKPT**

En la recuperación hay nuevamente dos situaciones:

a. Que encontremos primero un registro (END CKPT). En ese caso, deberemos retroceder hasta el (BEGIN , Tx) más antiguo del listado que figure en el (BEGIN CKPT) para rehacer todas las transacciones que commitearon. Escribir (ABORT, Ty) para aquellas que no hayan commiteado. **EL REDO se hace sobre las transacciones activas en el BEGIN CKPT que SI commitearon, y solo rehago las operaciones que están después del BEGIN CKPT en adelante. El commit puede estar después del END CKPT**
b. 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. Es decir, tengo que seguir subiendo hasta que encuentre un END.

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);
```

1. Complete los listados de transacciones en los BEGIN CKPT

Las que están activas hasta ese Checkpoint en la línea 9, son las T2 y T3

```
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);
```

2. ¿hasta qué linea hay que volver atrás?

Primeero encuentro el BEGIN CKPT de la línea 19, no me sirve, así que sigo subiendo. Encuentro un END en la línea 15 asi q lo matcheo con su BEGIN CKPT de la línea 9, que marca como transacciones activas a T2 y T3

Voy a tener que volver hasta la línea 3, en donde comienza la transacción mas antigua, la T2

3. Indique como será el procedimiento de recuperación

Voy a rehacer las operaciones que SI commitearon **mirando todo el log desde el BEGIN CKPT hasta el final del log,** osea la T2, T3 y T5. Las rehago desde arriba hacia abajo. Luego, aborto las que no committearon (T4). Si no se entiendó bien esto, ver el EJ 3 e la practica

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.

Checkpoint activo para UNDO/REDO

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.

Guardo los **ITEMS** NO las transacciones

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, o para rehacer sus operaciones posteriores al BEGIN CKPT, en caso de que haya commiteado.

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);
```

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)?

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). En este caso es solo A.

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.

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).

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.