



FCEyN UBA

BASES DE DATOS

Recuperación utilizando Logging



Índice

1. Introducción	2
1.1. Espacios de memoria	3
1.2. Generalidades	4
2. Descripción General de los mecanismos de recuperación	4
2.1. Undo Logging	4
2.1.1.	4
2.1.2.	5
2.2. Redo Logging	5
2.2.1.	5
2.2.2.	6
2.3. Consideraciones generales	6
2.4. Undo/Redo Logging	7
2.4.1.	7
2.4.2.	7
3. Checkpoints	7
3.1. Undo/Checkpoint Quiescent	8
3.2. Undo/Checkpoint No-Quiescent	8
3.3. Redo/Checkpoint No-Quiescent	8
3.4. Undo-Redo/Checkpoint No-Quiescent	9
4. Ejemplos	10
4.1. Ejemplo Undo Checkpoint Quiescent	10
4.2. Ejemplo Redo Checkpoint No-Quiescent	11
4.2.1. Situación 1	11
4.2.2. Situación 2	12
4.3. Ejemplo Undo/Redo Checkpoint No-Quiescent	12
4.3.1. Situación 1	12
4.3.2. Situación 2	13

1. Introducción

El presente apunte está orientado a dar una visión general del proceso de recuperación utilizando la técnica de Logging. De ninguna manera pretende ser un estudio exhaustivo del tema ni reemplazar la bibliografía recomendada. Veremos tres estrategias de recuperación: *Undo Logging*, *Redo Logging* y *Undo/Redo Logging*. En primera instancia describiremos cada una de las estrategias en forma general y posteriormente con la utilización de puntos de control o *checkpoints* ya sean estos *quiescentes* o *no-quiescentes*.

En resumen tenemos un objetivo:



Objetivo

Llevar la base de datos a un estado consistente, luego de la ocurrencia de una falla del sistema ("crash").

Para lo cual utilizamos técnicas:



Técnicas

Basadas en el uso del Log, el cual registra la historia de los cambios sobre la base de datos.

El Log es un archivo *append-only* al que se le van agregando registros que dan cuenta de eventos importantes. Los posibles tipos de registros son:

- **Start record:** $\langle \text{START } T_i \rangle$ La transacción T_i ha empezado.
- **Commit record:** $\langle \text{COMMIT } T_i \rangle$ La transacción T_i ha completado exitosamente.
- **Abort record:** $\langle \text{ABORT } T_i \rangle$ La transacción T_i abortó.
- **Update record:** Existe uno **distinto** por cada método de logging. Indica que hubo un cambio en un ítem de la BD.
 - **Undo Logging:** $\langle T, X, v \rangle$
 - **Redo Logging:** $\langle T, X, w \rangle$
 - **Undo/Redo Logging:** $\langle T, X, v, w \rangle$
- **Start y End Checkpoints record:** Estos registros dependerán del tipo de checkpoint utilizado.

Las transacciones se ejecutan concurrentemente, grabando los *Logs Records* en forma intercalada en el Log. El Log es manejado por el Log Manager. Los bloques del Log son inicialmente creados en memoria principal y reservados por el Buffer Manager, al igual que cualquier otro bloque que necesita la BD.

1.1. Espacios de memoria

Utilizaremos la abstracción de ítem para hablar de la unidad mínima de lectura y escritura. Cada ítem tendría un valor y las transacciones leen y escriben ítems (se asume que los ítems no exceden un bloque).

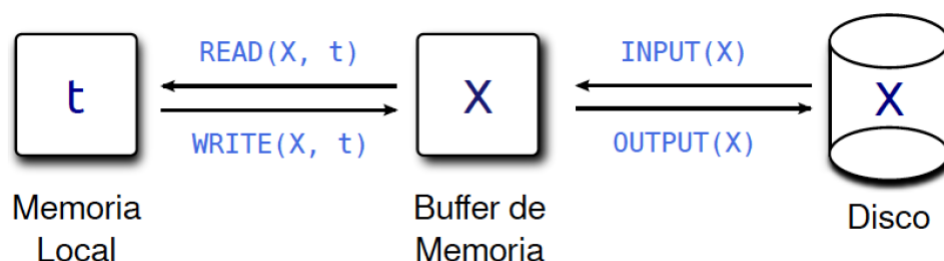


Figura 1: Espacios de memoria

Las operaciones primitivas de una transacción son:

- **INPUT(X):** Copia el bloque de disco que contiene el ítem X a un buffer de memoria.
- **READ(X, t):** Copia el ítem X del buffer de memoria a la variable temporal t (memoria local de la transacción). Si el bloque que contiene el ítem no está en memoria, se ejecuta primero un INPUT(X).
- **WRITE(X, t):** Copia el valor de la variable local t al ítem X en el pool buffers de memoria. Si el bloque que contiene el ítem no está en memoria se ejecuta primero un INPUT(X).
- **OUTPUT(X):** Copia el bloque que contiene el ítem X del buffer de memoria al disco.
- **FLUSH LOG:** Comando emitido por el Log Manager para que el Buffer Manager fuerce la escritura de los registros de Log al disco.



Atención

Los **READS** y los **WRITES** son emitidos por las transacciones.
 Los **INPUTS** y los **OUTPUTS** son emitidos por el buffer manager.

Es importante tomar en cuenta que los registros de Log son primero escritos a memoria y no a se escriben a disco en cada acción



Idempotencia

Los pasos de recuperación son idempotentes, es decir, ejecutarlos muchas veces tiene exactamente el mismo efecto que hacerlo sólo una vez.

1.2. Generalidades

A su vez las transacciones las podemos dividir en *completas* e *incompletas*

- **Transacciones completas:** son las transacciones comiteadas (terminaron exitosamente con COMMIT) y las transacciones abortadas (terminaron con ABORT).
- **Transacciones incompletas:** son las transacciones con registro de START, y sin registro de COMMIT ni ABORT

2. Descripción General de los mecanismos de recuperación

2.1. Undo Logging



Undo Logging Recovery

Deshace las transacciones incompletas.

Para cada transacción incompleta T_i , escribe un registro $\langle \text{ABORT } T_i \rangle$ al log y

Realiza un *flush* del log

El registro de log tiene la siguiente estructura: $\langle T, X, v \rangle$, indica que la transacción T cambió el valor de X y que el valor anterior era v .

2.1.1. Reglas

- **U1:** Los registros del tipo $\langle T, X, v \rangle$ se escriben a disco **antes** que el nuevo valor de X se escriba a disco en la base de datos.
- **U2:** El registro $\langle \text{COMMIT } T_i \rangle$ se escribe a disco **después** que todos los elementos modificados por T se hayan escrito en disco (y tan rápido como sea posible).

Resumiendo, se escribe a disco en el siguiente orden:

- (a) Los registros de log de los ítems que se modificaron,
- (b) Los ítems que se modificaron y, finalmente,
- (c) El registro COMMIT.

El orden de (a) y (b) se aplica a cada ítem en forma individual

2.1.2. Recovery


Los pasos a seguir son:

- Paso 1: Dividir las transacciones entre Completas (comiteadas o abortadas) e Incompletas
- Paso 2: Deshacer los cambios de las T incompletas. El Recovery Manager recorre el log *desde el final hacia arriba*, recordando las T_i de las cuales ha visto un record $\langle \text{COMMIT } T_i \rangle$ o un record $\langle \text{ABORT } T_i \rangle$. A medida que va subiendo si encuentra un record $\langle T_i, X, v \rangle$ entonces:
 - a) Si T_i es una transacción cuyo $\langle \text{COMMIT } T_i \rangle$ (o $\langle \text{ABORT } T_i \rangle$) ha sido visto, entonces no hacer nada.
 - b) Si no, T_i es una transacción incompleta. El Recovery Manager debe cambiar el valor de X a v (X ha sido alterado justo antes del *Crash*).
- Paso 3: Finalmente, el Recovery Manager debe grabar en el Log un $\langle \text{ABORT } T_i \rangle$ por cada T_i incompleta y luego hacer un flush del log

El Undo Logging tiene las siguientes desventajas:

- No podemos “*commitear*” una transacción sin antes grabar todos sus cambios a disco.
- Requiere que los ítems sean grabados inmediatamente después de que la transacción termino incrementando, tal vez, el número de I/O.

2.2. Redo Logging



Redo Logging Recovery

Rehace las transacciones que hicieron **commit**.

Para cada transacción incompleta T_i , escribe un registro $\langle \text{ABORT } T_i \rangle$ al log y

Realiza un **flush** del log

El registro de log tiene la siguiente estructura: $\langle T_i, X, w \rangle$, indica que la transacción T_i cambió el valor de X y que el **valor nuevo** es w .

2.2.1. Reglas

La regla en este caso es una sola llamada: *write ahead logging rule*, que establece que los registros del log se escriben a disco ANTES que los elementos. De esta regla se desprenden dos casos:

- **R1 a:** Los registros del tipo $\langle T, X, w \rangle$ se escriben a disco **antes** que el nuevo valor de X se escriba a disco en la base de datos.
- **R1 b:** El registro $\langle \text{COMMIT } T_i \rangle$ se escribe a disco **antes** que cualquier elemento modificado por T_i se haya escrito en disco.

Resumiendo, se escribe a disco en el siguiente orden:

- (a) Los registros de log de los ítems que se modificaron,
- (b) El registro COMMIT
- (c) Los ítems que se modificaron con sus valores nuevos

2.2.2. Recovery

Para recuperar la BD a partir del Log los pasos a seguir son:

- Paso 1: Dividir las transacciones entre Completas (comiteadas o abortadas) e Incompletas
- Paso 2: El Recovery Manager recorre el log *desde el comienzo*, y para cada registro $\langle T_i, X, v \rangle$ encontrado
 - a) Si T_i es una transacción no *commiteada* no hacer nada.
 - b) Si no, T_i es una transacción *commiteada*. El Recovery Manager debe escribir el valor de w para X en la Base de Datos.
- Paso 3: Finalmente, el Recovery Manager debe grabar en el Log un $\langle \text{ABORT } T_i \rangle$ por cada T_i incompleta y luego hacer un flush del log

2.3. Consideraciones generales

- El método **Undo** requiere que los items sean grabados inmediatamente después de que la transacción terminó, quizás incrementando el número de I/O.
- El método **Redo** requiere mantener todos los bloques modificados en el Buffer hasta que la transacción realice un *commit* y los registros de log sean efectivamente escritos a disco (se haga un flush de ellos), esto puede incrementar el promedio de bloques en *buffer* requeridos por las transacciones.
- El tercer método de Logging, llamado Undo/Redo y que veremos a continuación, provee mayor flexibilidad para ordenar las acciones pero a expensas de mantener más información en el Log.

2.4. Undo/Redo Logging



Redo Logging Recovery

Deshace todas las transacciones incompletas a partir de la mas reciente
Rehace las transacciones que hicieron **commit** a partir de la mas antigua.

Para cada transacción incompleta T_i , escribe un registro $\langle \text{ABORT } T_i \rangle$ al log y
 Realiza un **flush** del log

El registro de log tiene la siguiente estructura: $\langle T_i, X, v, w \rangle$, indica que la transacción T_i cambió el valor de X y que el valor viejo es v y el valor nuevo es w .

2.4.1. Reglas

Los registros del tipo $\langle T_i, X, v, w \rangle$ se escriben a disco **antes** que el nuevo valor de X se escriba a disco en la base de datos.

2.4.2. Recovery

Para recuperar la BD a partir del Log hay que hacer lo siguiente:

- ★ Aplicar la estrategia **Redo** a todas las transacciones *commiteadas* en el orden de las primeras a las últimas.
- ★ Aplicar **Undo** a todas las transacciones incompletas en el orden de las últimas a las primeras.

Es necesario aplicar ambas políticas. Esto es porque el registro $\langle \text{COMMIT } T_i \rangle$ puede preceder o seguir los cambios de los ítems de la BD en disco. Por lo tanto, podemos tener una transacción que hizo *commit* con alguno o ninguno de sus cambios en disco, o una transacción sin *commit* con alguno o todos sus cambios en disco.

3. Checkpoints

Los *checkpoints* se realizan periódicamente y son utilizados para evitar tener que recorrer todo el archivo de log durante la recuperación. Hay dos tipos de estrategias de checkpoints:

- **Checkpoint Quiescent:** No aceptan nuevas transacciones (el sistema queda “inactivo” durante el checkpoint).
- **Checkpoint No-Quiescent:** Aceptan nuevas transacciones durante el checkpoint.

3.1. Undo/Checkpoint Quiescente

Se realizan las siguientes etapas:

1. Dejar de aceptar nuevas transacciones.
2. Esperar a que todas las transacciones activas (aquellas con $\langle \text{START } T_i \rangle$ y sin $\langle \text{COMMIT } T_i \rangle$ ni $\langle \text{ABORT } T_i \rangle$) realicen un *commit* o un *abort*.
3. Escribir un $\langle \text{CKPT} \rangle$ en el log y luego efectuar un flush.
4. Aceptar nuevas transacciones.

Recovery: Se lee el Log a partir del último registro y hasta el punto de *checkpoint*. Sabemos que hasta ahí todas las transacciones terminaron y sus cambios fueron guardados en disco. Aplicar la política del **UNDO Logging**.

3.2. Undo/Checkpoint No-Quiescente

Esta estrategia es mas compleja porque se permite comenzar nuevas transacciones durante el proceso del *checkpoint*.

Pasos:

- 1 Escribir $\langle \text{Start CKPT}(T_1, T_2 \dots T_k) \rangle$ en el log, y efectuar un flush. $T_1, T_2 \dots T_k$ son las transacciones **activas** al momento de introducir el *checkpoint*.
- 2 Esperar a que todas las transacciones $T_1, T_2 \dots T_k$ terminen (ya sea con abort o commit), pero sin prohibir que empiecen otras transacciones.
- 3 Escribir $\langle \text{End CKPT} \rangle$ en el log y efectuar un flush del mismo.

Recovery: Leer el Log desde el último registro, aplicando la política **UNDO**. Para ello hay que recorrer según los siguientes casos:

- Si se encuentra un $\langle \text{End CKPT} \rangle$, no se debe leer más allá del $\langle \text{Start CKPT} \rangle$.
- Si se encuentra un $\langle \text{Start CKPT} \rangle$. (no hay $\langle \text{End CKPT} \rangle$ debido a un crash). Se debe leer el Log hasta el $\langle \text{START } T_i \rangle$, ($1 \leq i \leq k$), más antiguo de las transacciones de la lista del $\langle \text{Start CKPT}(T_1, T_2 \dots T_k) \rangle$ que quedaron incompletas

3.3. Redo/Checkpoint No-Quiescente

La estrategia de Redo Logging presenta un problema con los checkpoint que no existía en el Undo Logging. Como los cambios a la base de datos realizados por transacciones que hicieron *commit* pueden eventualmente ser copiados al disco mucho después que el momento en el cual se la transacción realiza el *commit* no podemos limitarnos a las

transacciones activas en el momento en que se decide hacer un *checkpoint*. Es necesario escribir a disco todas las modificaciones realizadas por las transacciones que hicieron *commit* antes de finalizar el *checkpoint*. Por lo tanto trataremos solo con *checkpoint no-quiescente*. Pasos:

- 1 Escribir $\langle \text{Start CKPT}(T_1, T_2 \dots T_k) \rangle$ en el log, y efectuar un flush. $T_1, T_2 \dots T_k$ son las transacciones **activas** al momento de introducir el *checkpoint*.
- 2 Forzar a disco los ítems que fueron escritos en el pool de buffers por transacción que hicieron *commit* al momento del $\langle \text{Start CKPT} \rangle$. pero que aún no fueron guardados en disco.
- 3 Escribir $\langle \text{End CKPT} \rangle$ en el log y efectuar un flush del mismo.

Recovery: Se debe aplicar la política **REDO**: recorrer el log identificando las transacciones que hicieron *commit* y rehacerlas. Para decidir lo que hay que recorrer (empezando desde el ultimo registro hacia los mas antiguos):

- Si se encuentra un $\langle \text{End CKPT} \rangle$, se debe rehacer las T_i que tienen *commit* y estaban activas al momento del checkpoint y las que comenzaron después del mismo. Tener en cuenta que el $\langle \text{End CKPT} \rangle$ indica que las transacciones que hicieron *commit* antes del $\langle \text{Start CKPT} \rangle$ tienen sus cambios en disco.
- Si se encuentra un $\langle \text{Start CKPT} \rangle$. (no hay $\langle \text{End CKPT} \rangle$ debido a un crash). Ubicar el $\langle \text{End CKPT} \rangle$ **anterior** y su correspondiente $\langle \text{Start CKPT}(T_1, T_2 \dots T_k) \rangle$ y rehacer todas las transacciones comiteadas que comenzaron a partir de ahí o están entre las T_i , desde el $\langle \text{START } T_i \rangle$ más antiguo de dichas transacciones.

3.4. Undo-Redo/Checkpoint No-Quiescente

Pasos:

- 1 Escribir $\langle \text{Start CKPT}(T_1, T_2 \dots T_k) \rangle$ en el log, y efectuar un flush. $T_1, T_2 \dots T_k$ son las transacciones **activas** al momento de introducir el *checkpoint*.
- 2 Escribir a disco todos los buffers “sucios” o *dirty buffers* (contienen elementos cambiados que aún no fueron escritos a disco) al momento del $\langle \text{Start CKPT} \rangle$. Escribir a disco **todos** los buffers sucios, no sólo los que hayan sido modificados por transacciones comiteadas al momento del $\langle \text{Start CKPT} \rangle$.
- 3 Escribir $\langle \text{End CKPT} \rangle$ en el log y efectuar un flush del mismo.

**Nota**

Notar que no es necesario esperar que las transacciones activas terminen para introducir el $\langle \text{End CKPT} \rangle$

Recovery: Se debe aplicar la política **Undo-Redo** para deshacer las incompletas y rehacer las que hicieron *commit*. Para decidir lo que hay que recorrer (empezando desde el ultimo registro hacia los mas antiguos):

- Transacciones incompletas: para deshacerlas se debe retroceder hasta el start más antiguo de ellas. Agregar registro $\langle \text{ABORT } T_i \rangle$ al log para cada transacción T_i incompleta, y hacer flush del log
- Transacciones con *commit*: Si leyendo desde el último registro se encuentra un $\langle \text{End CKPT} \rangle$ sólo será necesario rehacer las acciones efectuadas desde el correspondiente registro $\langle \text{Start CKPT}(T_1, T_2 \dots T_k) \rangle$ en adelante. Si no encontramos el $\langle \text{End CKPT} \rangle$, para saber desde dónde comenzar utilizamos el mecanismo usado para el mismo caso en la estrategia Redo

4. Ejemplos

4.1. Ejemplo Undo Checkpoint Quiescente

Supongamos el siguiente archivo de log:

1. $\langle \text{START } T_1 \rangle$
2. $\langle T_1, A, 5 \rangle$
3. $\langle \text{START } T_2 \rangle$
4. $\langle T_2, B, 10 \rangle$
5. $\langle T_2, C, 15 \rangle$
6. $\langle T_1, D, 20 \rangle$
7. $\langle \text{COMMIT } T_1 \rangle$
8. $\langle \text{ABORT } T_2 \rangle$
9. $\langle \text{CKPT} \rangle$
10. $\langle \text{START } T_3 \rangle$
11. $\langle T_3, E, 25 \rangle$

Supongamos ahora que ocurre un crash luego del paso 11.

Recovery:

T_2 es ignorada debido a que se encuentra un $\langle \text{ABORT } T_2 \rangle$.

Se identifica T_3 como la única transacción a deshacer.

Recorremos desde el final hasta el último $\langle \text{CKPT} \rangle$, en el paso 9.

Transacciones a deshacer: T_3 .

Cambios en la base de datos:

- $F := 30$ (deshaciendo el paso 12)
- $E := 25$ (deshaciendo el paso 11)

Cambios en el log: Se agrega el registro $\langle \text{ABORT } T_3 \rangle$ y luego Flush Log.

4.2. Ejemplo Redo Checkpoint No-Quiescente

Supongamos el siguiente archivo de log:

1. $\langle \text{START } T_1 \rangle$
2. $\langle T_1, A, 5 \rangle$
3. $\langle \text{START } T_2 \rangle$
4. $\langle \text{COMMIT } T_1 \rangle$
5. $\langle T_2, B, 10 \rangle$
6. $\langle \text{START CKPT } (T_2) \rangle$
7. $\langle T_2, C, 15 \rangle$
8. $\langle \text{START } T_3 \rangle$
9. $\langle T_3, D, 20 \rangle$
10. $\langle \text{End CKPT} \rangle$
11. $\langle \text{COMMIT } T_2 \rangle$
12. $\langle \text{COMMIT } T_3 \rangle$

4.2.1. Situación 1

Sucede un Crash y el último registro es $\langle \text{COMMIT } T_2 \rangle$ (Paso 11).

Las transacciones a rehacer son T_1 y T_2 . Sin embargo, puedo ignorar T_1 ya que encuentro un $\langle \text{End CKPT} \rangle$ (paso 10). Debo empezar a recorrer entonces desde el start de T_2 en el paso 3.

Transacciones a rehacer: T_2 (ignoro T_1).

Cambios en la base de datos:

- $B := 10$ (rehaciendo el paso 5)
- $C := 15$ (rehaciendo el paso 7)

Cambios en el log: Se agrega el registro $\langle \text{ABORT } T_3 \rangle$ y luego Flush Log

4.2.2. Situación 2

Sucede un Crash y el último registro es $\langle \text{COMMIT } T_3 \rangle$ (Paso 12).

Nuevamente en este caso se encuentra un $\langle \text{End CKPT} \rangle$, por lo que solo hay que concentrarse en las transacciones activas al momento del $\langle \text{START CKPT } (T_2) \rangle$ y en las que comenzaron después. En este caso, se identifica a T_2 y T_3 como las transacciones a rehacer. Se debe empezar desde el start más antiguo (el de T_2 en el paso 3).

Transacciones a rehacer: T_2 y T_3 (ignoro T_1).

Cambios en la base de datos:

- $B := 10$ (rehaciendo el paso 5)
- $C := 15$ (rehaciendo el paso 7)
- $D := 20$ (rehaciendo el paso 9)

Cambios en el log: ninguno.

4.3. Ejemplo Undo/Redo Checkpoint No-Quiescente

Suponga el siguiente log:

1. $\langle \text{START } T_1 \rangle$
2. $\langle T_1, A, 4, 5 \rangle$
3. $\langle \text{START } T_2 \rangle$
4. $\langle \text{START } T_9 \rangle$
5. $\langle T_9, X, 9, 90 \rangle$
6. $\langle \text{ABORT } T_9 \rangle$
7. $\langle \text{COMMIT } T_1 \rangle$
8. $\langle T_2, B, 9, 10 \rangle$
9. $\langle \text{START CKPT } (T_2) \rangle$
10. $\langle T_2, C, 14, 15 \rangle$
11. $\langle \text{START } T_3 \rangle$
12. $\langle T_3, D, 19, 20 \rangle$
13. $\langle \text{END CKPT} \rangle$
14. $\langle \text{COMMIT } T_2 \rangle$
15. $\langle \text{COMMIT } T_3 \rangle$

4.3.1. Situación 1

Sucede un Crash y el último registro es $\langle \text{COMMIT } T_3 \rangle$ (Paso 15):

Al encontrar un registro $\langle \text{ABORT } T_9 \rangle$, se descarta T_9 .

Por otro lado, puede identificarse a T_1 , T_2 y T_3 como transacciones a rehacer, y ninguna transacción para deshacer.

Sin embargo, al encontrar un $\langle \text{END CKPT} \rangle$ (paso 13), se sabe que todas las modificaciones anteriores ya fueron escritas a disco (por las reglas del mecanismo). Por lo tanto se puede ignorar a T_1 , y para T_2 y T_3 sólo deben rehacer las acciones del paso 9 en adelante, que es cuando se introdujo el $\langle \text{START CKPT} \rangle$.

Transacciones a rehacer: T_2 y T_3 (se ignora T_1).

Transacciones a deshacer: ninguna.

Cambios en la base de datos:

- $C := 15$ (rehaciendo el paso 10)
- $D := 20$ (rehaciendo el paso 12)

Cambios en el log: ninguno.

4.3.2. Situación 2

Sucede un Crash y el último registro es $\langle \text{COMMIT } T_2 \rangle$ (Paso 14):

En primer paso, se ignora a T_9 por el registro $\langle \text{ABORT } T_9 \rangle$. Asimismo, se identifica a T_1 y T_2 como transacciones a rehacer y a T_3 como una transacción a deshacer.

Para deshacer T_3 , se debe retroceder hasta su $\langle \text{START } T_3 \rangle$ en el paso 11.

Para rehacer T_1 y T_2 , como encontramos un registro $\langle \text{END CKPT} \rangle$ podemos ignorar T_1 , y para T_2 nos alcanza con empezar a rehacer desde el paso 9 (cuando comenzó el $\langle \text{START CKPT} \rangle$).

Transacciones a rehacer: T_2 (se ignora T_1).

Transacciones a deshacer: T_3 .

Cambios en la base de datos:

- $D := 19$ (deshaciendo el paso 12)
- $C := 15$ (rehaciendo el paso 10)

Cambios en el log: Se agrega el registro $\langle \text{ABORT } T_3 \rangle$ y luego Flush Log..

Referencias

- [1] *Database Systems: The Complete Book (2 ed.)*. Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. 2008. Prentice Hall Press, Upper Saddle River, NJ, USA.