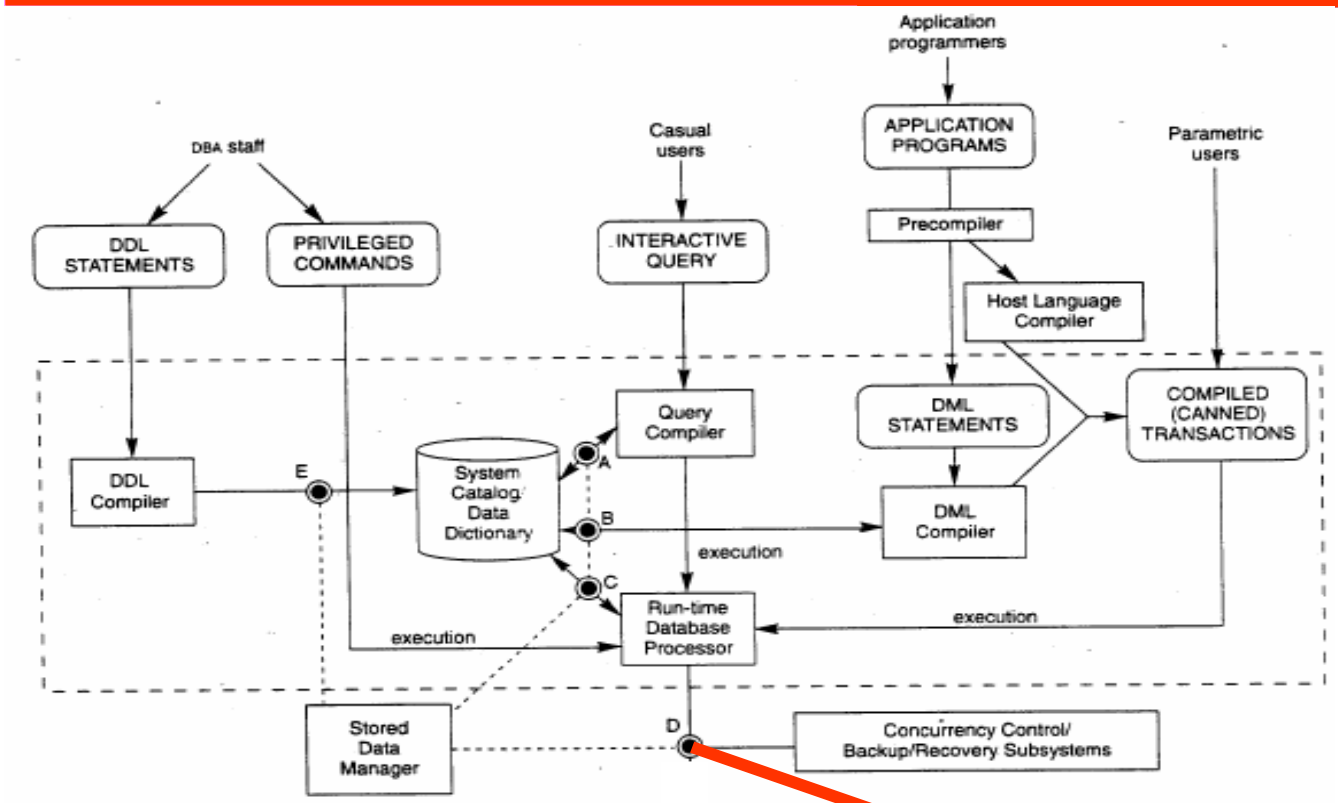
Recuperación de Fallas

Recuperación



Un sistema computarizado, como cualquier otro dispositivo mecánico o eléctrico, está sujeto a fallas. Existe una variedad de causas posibles para esas fallas.

En cada uno de estos casos, es muy probable que se pierda información de la base de datos. Es responsaliblidad del SGBD detectar estas fallas y restaurar la base de datos a un estado existente previo a la falla.

Esto se logra generalmente mediante procedimientos de respaldo (backup), bitácora (logging) y recuperación.

Tipos de Fallas

El sistema debe estar preparado para recuperarse no sólo de fallas puramente locales, como la aparición de una condición de desborde dentro de una transacción, sino también de fallas globales, como podría ser la interrupción del suministro eléctrico al CPU.

Las fallas locales son las que afectan sólo a la transacción en donde ocurrió.

Por el contrario las fallas globales, afectan a varias -y casi siempre a todas- las transacciones que se estaban efectuando en el momento de la falla, por lo cual tienen implicaciones importantes en el sistema.

Estas fallas pueden ser:

Tipos de Fallas

• Falla del Sistema: por ejemplo interrupción del servicio eléctrico, estas afectan a todas las transacciones que se estaban ejecutando pero no afectan a la base de datos.

Las fallas de sistema se conocen también como caídas (crash) suaves. El problema aquí es que se pierda el contenido de memoria principal, en particular, las áreas de almacenamiento temporal o *buffers*.

Si esto ocurre, no se conocerá el estado preciso de la transacción que se estaba ejecutando en el momento de la falla, esta transacción jamás se podrá completar con éxito por lo que será preciso anularla cuando se reinicie el sistema.

Además, puede ocurrir que sea necesario volver a ejecutar algunas transacciones que sí se realizaron con éxito antes de la falla pero cuyas modificaciones no lograron efectuarse sobre la base de datos porque no lograron ser transferidas de los *buffers* de la base de datos a la base de datos física (en disco).

Tipos de Fallas

• Fallas en los medios de almacenamiento: por ejemplo una falla en el controlador de disco o un aterrizaje de cabeza en el disco, estas fallas sí causan daños a la base de datos o a una porción de ella y afecta, al menos, a las transacciones que están haciendo uso de esa porción.

Las fallas de los medios de almacenamiento se llaman caídas duras. La recuperación de una falla semejante implica, en esencia, cargar de nuevo la base de datos a partir de una copia de respaldo (*database backup*) y después utilizar la bitácora, o *log file*, para realizar de nuevo todas las transacciones terminadas desde que se hizo esa copia para respaldo.

No hay necesidad de anular todas las transacciones inconclusas en el momento de la falla, porque por definición esas transacciones ya se anularon (se destruyeron) de todas maneras.

Tipos de Fallas

• Fallas por catástrofes: por ejemplo terremotos, incendios, inundaciones, etc. Su tratamiento es similar al de fallas de los medios. La principal técnica para manejar este tipo de fallas es la del *database backup* . Como se mencionó anteriormente, este es un respaldo periódico que se hace de la base de datos. Después de una caída de esta índole el sistema se restaura recargando la base de datos con la copia del último respaldo y recreando la base de datos mediante la bitácora o *log file*.

• Errores del Sistema: como realizar operaciones que causen un *overflow* de un entero o la división por cero, así mismo puede ocurrir que se pasen valores erróneos a algún parámetro o que se detecte un error en la lógica de un programa, o que sencillamente no se encuentren los datos del programa. Además, en algunos ambientes de desarrollo el usuario puede explícitamente interrumpir una transacción durante su ejecución.

• Aplicación del Control de Concurrencia: que ocurre por ejemplo cuando una transacción viola las reglas de serialización o cae en interbloqueo.

TRANSACCIONES

El cuerpo de una transacción puede verse como:

OPERACIONES EN UNA TRANSACCIÓN:

Begin transaction : inicio de la transacción

Read a : lectura del objeto a

Write a : escritura del objeto a

Rollback : anulación de la transacción

Commit : fin de la transacción

Begin transaction T
O1
O2
.
.
.
On
Commit T.

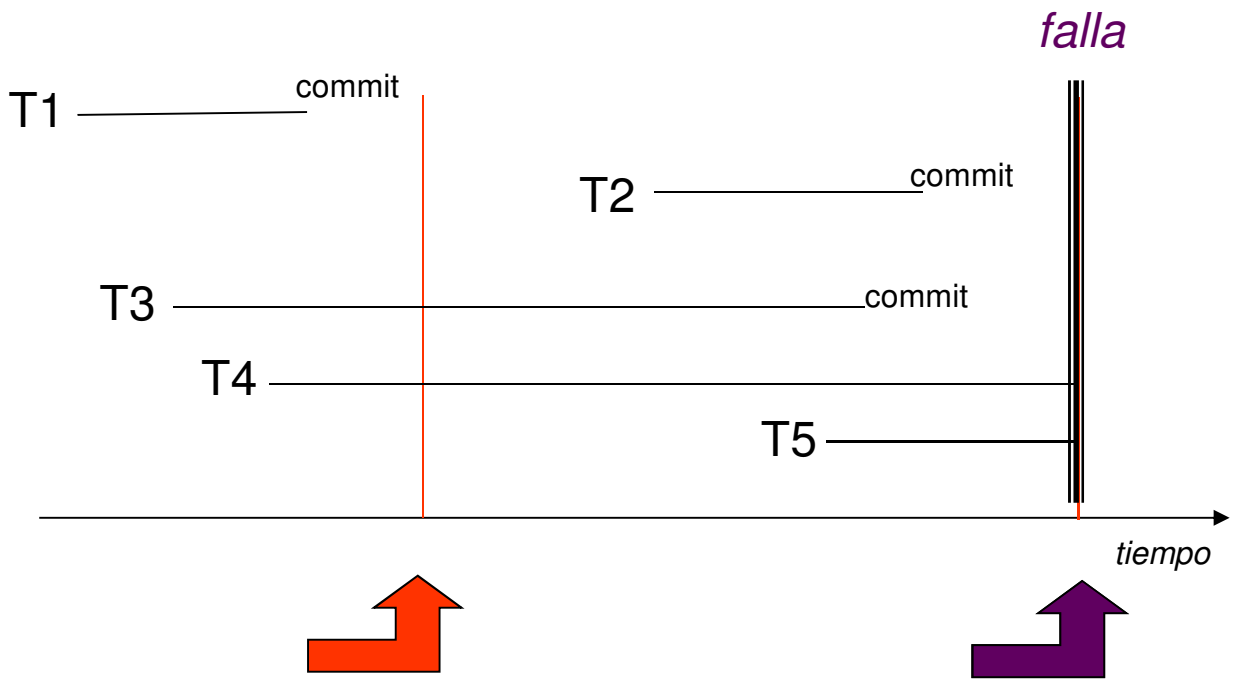
Begin transaction T
O1
O2
.
.
.
On
Rollback.

PROPIEDADES DE LAS TRANSACCIONES

- Atomicidad: una transacción T es una unidad atómica de ejecución, es decir, o se ejecuta toda o no se ejecuta nada. Dicho de otra manera, los efectos de la T son reflejados en la BD o son completamente ignorados.
- Consistencia: los efectos de una transacción T trasladan la BD de un estado consistente a otro estado consistente. Una BD es consistente si satisface todas las consideraciones iniciales de estructura e integridad.
- Independencia: una transacción T se realiza como si fuera la única en hacerlo.
- Durabilidad: los efectos de una transacción T que ha alcanzado su punto de validación (ha llegado al ´COMMIT´) deben ser permanentes en la BD.

Recuperación de Fallas

Se refiere al tratamiento de las distintas fallas que pueden ocurrir en un SGBD y cómo es posible restaurar el sistema después de dicha fallas.



Recuperación de Fallas

Así como en el control de concurrencia, la recuperación de fallas está estrechamente ligada a el procesamiento de las transacciones.

Recordemos que una transacción tiene la cualidad de ser atómica a pesar de que puede estar compuesta de varias operaciones, la atomicidad se controla con la llegada al COMMIT.

Si una transacción no sufrió ninguna problema y se pudo ejecutar completa, entonces el SGBD debe "comprometerse" a hacer permanentes los cambios que la transacción hizo sobre la base de datos ya que esta debió quedar en un estado consistente.

ATOMICIDAD

```
Begin transaction T
O1
O2
.
.
.
On
Commit T.
```

```
Begin transaction T
O1
O2
.
.
.
On
Rollback.
```

La atomicidad también ocurre en caso de que la transacción sufra algún problema que le impida ejecutarse. En este caso, la operación **ROLLBACK** señala el término no exitoso de la transacción, indicando al SGBD que "algo" debió salir mal, advierte que la base de datos puede estar en un estado inconsistente y que todas las modificaciones, que la transacción haya hecho hasta el momento, deben "retroceder" o anularse

¿Cómo puede anularse una transacción?

El sistema mantiene una *bitácora* o *log file* en respaldo, por lo general en un disco, en donde se detallan todas las operaciones de actualización y los valores iniciales y finales de cada objeto.

Por lo tanto, si resulta necesario anular alguna modificación específica, el sistema puede utilizar la entrada correspondiente de la bitácora para restaurar el valor original del objeto modificado.

TRES CONCEPTOS

1. Bitácora (*Log file*).
2. *Checkpoint.*
3. *Rollback (UNDO)*

1.- *Log file*

Es un registro que mantiene el curso de todas las transacciones que afectan los *data items* de la base de datos.

Esta información es mantenida en disco (memoria secundaria) de manera que sólo puede estar afectada, eventualmente, por fallas en medios de almacenamiento.

1.- *Log file*

Los *Logs* tienen una serie de entradas que se detallan a continuación. Las *T* se refieren a un identificador único de cada transacción:

[start-transaction,T]: registra que la transacción ha comenzado.

[write-item,T,X,old,new-value]: registra la transacción T ha cambiado el valor del data item X del valor viejo *old* al valor nuevo *new-value*.

[read-item,T,X]: registra que la transacción T ha leído el valor del data item X de la base de datos.

[commit,T]: registra que la transacción T ha terminado exitosamente y que los efectos pueden ser grabados permanentemente (committed, comprometidos) en la base de datos.

[abort,T]: registra que la transacción T ha sido abortada.

2.- *Checkpoint*

Es otro tipo de entrada en el *Log*. Un registro *checkpoint* se graba periódicamente en el *Log* justo en el punto en donde se guardó, en la base de datos en el disco, el efecto de todas las operaciones de escritura hechas por las transacciones que terminaron exitosamente (llegaron al commit).

Si hay una falla, las transacciones que se completaron antes del *checkpoint* no deben ser rehechas en el *recovery* (no debe aplicarse un REDO).

El SGBD debe decidir cada cuando hace un *checkpoint*. Esto puede ser cada *m* minutos o cada *n* transacciones completadas exitosamente, estos son parámetros del sistema.

2.- *Checkpoint*

HACER UN CHECKPOINT IMPLICA:

1. Suspender temporalmente la ejecución de todas las transacciones.

2. Forzar la escritura de todas las actualizaciones de las operaciones de las transacciones que llegaron al commit que están en los buffers de memoria principal al disco.

3. Escribir un registro *checkpoint* en el *Log* y forzar la escritura del *Log* en el disco.

4. Reasumir la ejecución de las transacciones.

2.- *Checkpoint*

Un registro *checkpoint* puede incluir una lista de las transacciones activas (aquellas que no se han terminado de ejecutar).

3.- *Rollback (UNDO)*

Si la transacción falla, por alguna razón, después de la actualización de la base de datos es necesario "anularla" (RollBack o UNDO).

Cualquier valor del data item que haya sido cambiado por la transacción debe volver a su valor previo.

Si una transacción T es anulada (*rollback*), cualquier transacción S, que en el interim, haya leído un valor de un data item X modificado por T, entonces S también debe anularse (rollback).

3.- *Rollback (UNDO)*

De igual forma, si una transacción R ha leído un valor de un data item Y que ha sido modificado por S, entonces R también se debe anular y así sucesivamente. Este fenómeno se conoce como *rollback* en cascada.

El *rollback* en cascada puede ser muy costoso y es por ello que los mecanismos de *recovery* han catalogado a este fenómeno como indeseable o nunca requerido.

Técnicas de Recuperación de Fallas no Catastróficas

La recuperación de las fallas de transacciones significa que la base de datos se debe restaurar desde algún estado considerado correcto del pasado - lo más cercano posible al momento de la falla. Para lograr esto el sistema recurre al *log file.*

Vamos a considerar las dos principales técnicas de *recovery* debido a fallas no catastróficas. A saber:

1. Actualización Diferida (*deferred update*) conocida también como NO UNDO/REDO.

2. Actualización Inmediata (*inmediate update*) conocida también como UNDO/NO REDO

ACTUALIZACION DIFERIDA – NO UNDO/REDO

La idea que manejan estas técnicas consiste en postergar o diferir cualquier actualización a la base de datos hasta que la transacción complete su ejecución exitosamente y llegue a su COMMIT.

Durante la ejecución de la transacción las actualizaciones son grabadas en el *log* y en el área de trabajo de la transacción. Cuando la transacción llega a su COMMIT, el *log* es forzado a escribirse en disco y luego las actualizaciones se reflejan en la base de datos. Si la transacción falla antes de llegar a su COMMIT no es necesario aplicar el UNDO a ninguna operación pues la transacción no ha afectado a la base de datos de ninguna manera.

Protocolo de ACTUALIZACION DIFERIDA

El protocolo típico de la actualización diferida se define como sigue:

1. Una transacción no puede hacer cambios a la base de datos hasta que llegue a su COMMIT.

2. Una transacción no llega a su COMMIT hasta que todas las operaciones de actualización hayan sido registradas en el *log* y el *log* haya sido forzado a escribirse en disco.

Si la transacción falla después de llegar a su COMMIT pero antes de que los cambios sean reflejados en la base de datos, basta con aplicar el algoritmo REDO según las operaciones y los valores de los data items que aparecen en el **log**.

Por eso a esta técnica, a veces, se le conoce como **NO UNDO/REDO.**

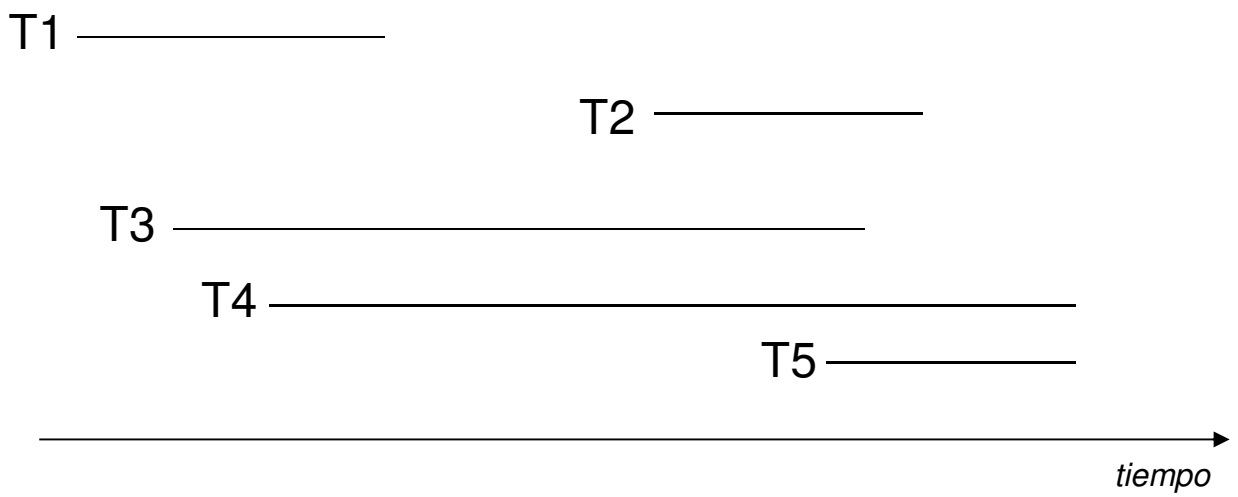
ACTUALIZACIÓN DIFERIDA EN AMBIENTES MULTIUSUARIOS

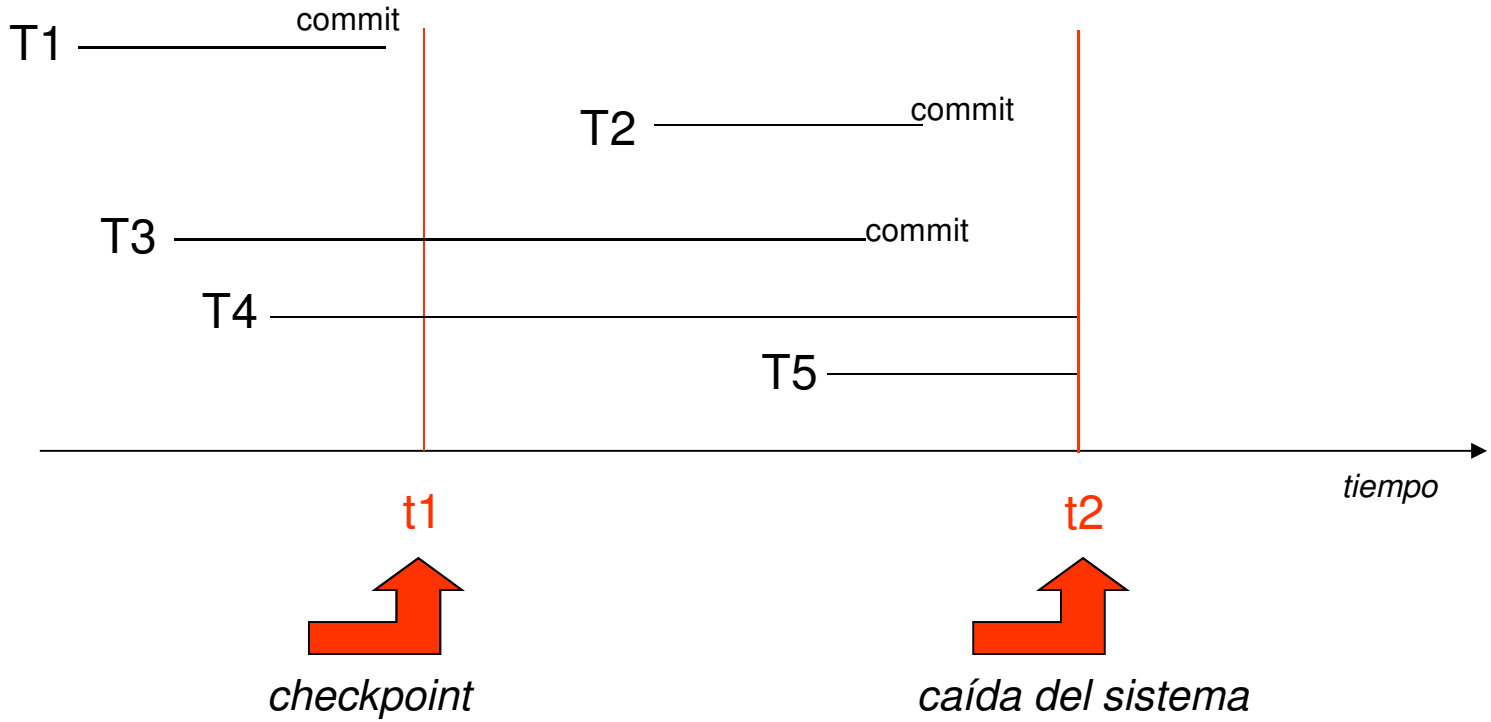
Para ambientes multiusuarios con control de concurrencia, el *recovery* puede hacerse más complejo dependiendo del protocolo de control de concurrencia usado. En general, a mayor grado de concurrencia que se desea llegar más difícil se vuelve el proceso de *recovery*.

Supongamos que el control de concurrencia se logra con el protocolo de dos fases asignando cerrojos (*locks*) a los data item requeridos por una transacción antes de que ella comience a ejecutarse. Para combinar el método de actualización diferida para *recovery* con la técnica para el control de la concurrencia, es necesario mantener todos los cerrojos sobre los items activos hasta que la transacción llegue a su COMMIT.

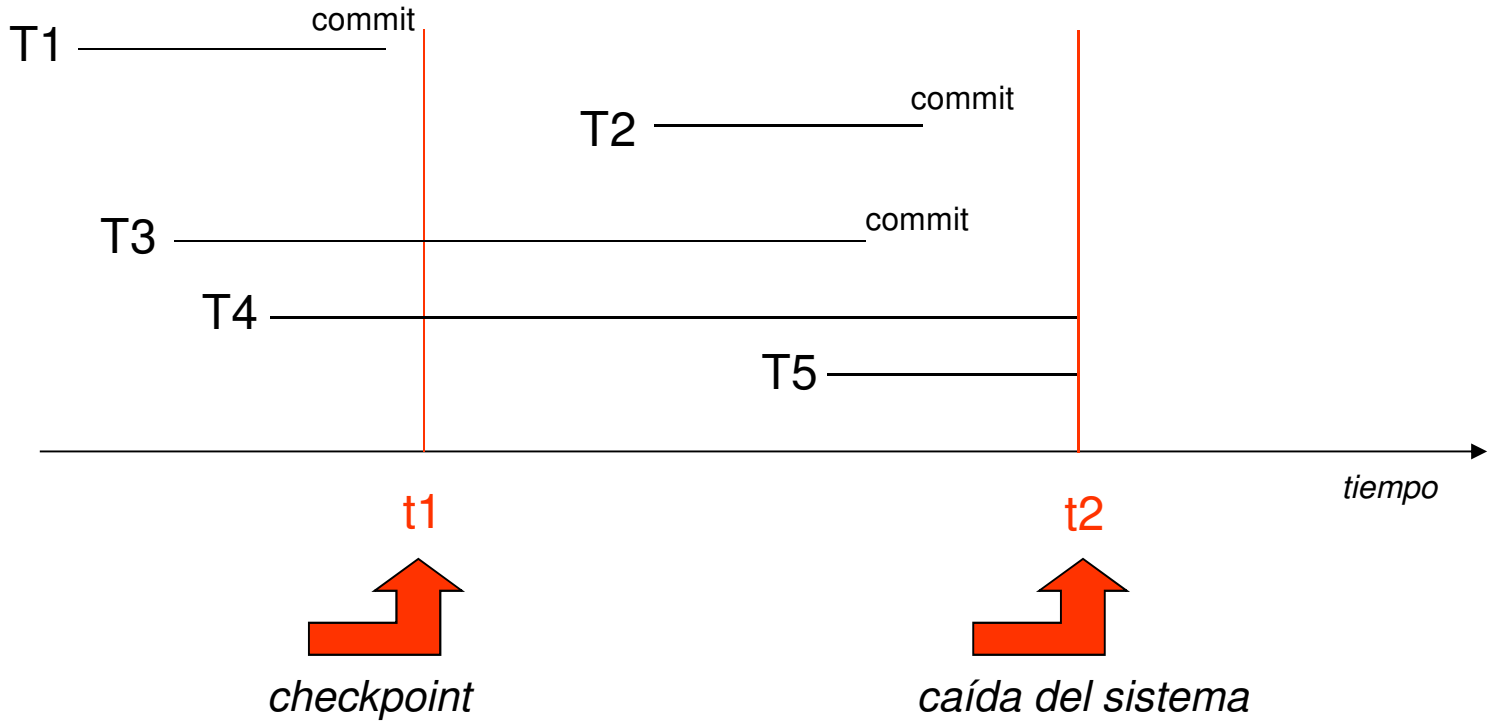
Después de ello, los cerrojos se remueven. A continuación se da el algoritmo de *recovery* llamado RDU_M (recovery deferred update in a multiuser environment) asumiendo que las entradas del *checkpoint* se encuentran en el *log file*.

PROCEDURE RDU_M: usa dos listas de transacciones mantenidas por el sistema: las transacciones que llegaron a su COMMIT (T) desde el último *checkpoint* y las transacciones activas (T´). Aplica el algoritmo REDO a todas las operaciones que han escrito (write-item) sobre los items, de las transacciones que han llegado a su COMMIT según el orden en el que ellas aparecen en el log. Las transacciones que están activas y que no han llegado a su COMMIT deben ser canceladas y sometidas nuevamente a ejecución.





En este ejemplo, cuando el *checkpoint* ocurre en el tiempo t1, T1 ha llegado a su COMMIT mientras que T3 y T4 no. T2 y T3 llegan a su COMMIT antes de que en t2 ocurra la caída del sistema, sin embargo ni T4 ni T5 logran terminar.



Según el método de RDU_M no es necesario hacer REDO a la transacción T1 pues esto ocurrió antes del último *checkpoint*. Las operaciones de escritura de T2 y T3 deben ser rehechas, debido a que estas llegaron a su COMMIT después del último *checkpoint*. Tanto T4 como T5
no se toman en cuenta debido a que ninguna de sus operaciones de escritura afectaron a la base de datos bajo el protocolo de actualización diferida.

T1	T2	T3	T4
read_item(A)	read_item(B)	read_item(A)	read_item(B)
read_item(D)	write_item(B)	write_item(A)	write_item(B)
write_item(D)	read_item(D)	read_item(C)	read_item(A)
	write_item(D)	write_item(C)	write_item(A)

Log file

[start-transaction,T1]
[write_item,T1,D,20]
[commit, T1]
[checkpoint]
[start-transaction,T4]
[write_item,T4,B,15]
[write_item,T4,A,20]
[commit,T4]
[start-transaction,T2]
[write_item,T2,B,12]
[start-transaction,T3]
[write_item,T3,A,30]
[write_item,T2,D,25] ← Caída del Sistema

Log file

[start-transaction,T1]
[write_item,T1,D,20]
[commit, T1]
[checkpoint]
[start-transaction,T4]
[write_item,T4,B,15]
[write_item,T4,A,20]
[commit,T4]
[start-transaction,T2]
[write_item,T2,B,12]
[start-transaction,T3]
[write_item,T3,A,30]
[write_item,T2,D,25] ← Caída del Sistema

T2 y T3 son ignorados debido a que ellos no llegaron a su COMMIT. T4 es rehecho debido a que ella llegó a su COMMIT después de que se hiciera el *checkpoint*.

Operaciones que no afectan a la base de datos

En general, no todas las operaciones de las transacciones afectan a la base de datos: generar o imprimir mensajes o reportes son ejemplos de ello.

Si una transacción falla antes de completarse, el reporte no debe ser tomado en cuenta, esto debe ocurrir sólo después de que la transacción haya llegado a su COMMIT. Esto suele controlarse dejando este tipo de acciones en una cola de trabajos en lotes (*batch jobs)*.

Los trabajos en lotes son ejecutados sólo si la transacción es exitosa si no lo es se cancelan.

ACTUALIZACIÓN INMEDIATA UNDO/NO REDO

Con esta estrategia el SGBD actualiza la base de datos inmediatamente después de ejecutar las operaciones en las transacciones sin esperar llegar al COMMIT.

PROCEDURE RIU_M: usa dos listas de transacciones: las transacciones que han llegado a su COMMIT desde el último *checkpoint* y las transacciones activas.

Aplica el algoritmo UNDO a todas las operaciones que han escrito (write-item) sobre los items, de las transacciones activas que aparecen en el *log*. Las operaciones deben ser deshechas en orden inverso a como aparecen en el *log*.

No se debe aplicar el algoritmo REDO a las transacciones que han llegado a su COMMIT, ya que ellas ya actualizaron la base de datos.

ACTUALIZACIÓN INMEDIATA EN AMBIENTES MULTIUSUARIOS

El UNDO se define como:

UNDO(WRITE_OP)

Deshacer una operación de escritura sobre un data item WRITE_OP consiste en examinar sus entradas al log

[write_item,T,X,old_value,new_value]

y colocar como valor del data item X el valor que aparece en old_value, considerado el BFIM (valor del data item antes de la actualización, BFter Image).

Deshacer un número de operaciones de escritura de una o más transacciones desde el *log* implica proceder en orden inverso a como estas operaciones fueron grabadas en el *log*.

Fuentes consultadas:

[1] Prof. Elsa Liliana Tovar.
Notas de clase compiladas.

[2] Navathe, Elsamri,
“Fundamentals of DataBase System”.

[3] http://queens.db.toronto.edu/~koudas/courses/cscd43/Lecture9.pdf