ii. Ídem ejercicio anterior, pero indicando al menos 3 alternativas para ejecutar la siguiente consulta, calculando el costo de cada una de ellas. Suponer los datos necesarios, índices, etc.

```
select e.nro_e, e.nom_e
from Trabaja_en t, Gerencia g, Empleado e
where t.nro_depto = g.nro_depto and e.nro_e = t.nro_e;
```

iii. Ídem anterior para la consulta:

```
select e.nro_e, e.nom_e
from Trabaja_en t, Gerencia g, Empleado e
where     t.nro_depto = g.nro_depto and e.nro_e = t.nro_e
      and e.edad < 30 and e.ciudad = "Rosario";
```

E6. Sean R y S dos relaciones tales que $n_R = 2.000.000$ y $n_S = 1.500.000$. En un bloque se pueden almacenar 40 tuplas de R o de S. $b_R = 50.000$ y $b_S = 37.500$.
Se pide seleccionar la mejor estrategia de junta, entre todas las que conozca, y estimar la cantidad de accesos al disco, para cada uno de los casos siguientes de disponibilidad de memoria volátil:

i. 2501 bloques.
ii. 501 bloques.
iii. 301 bloques.
iv. 101 bloques.

E7. Al describir la junta sobre *múltiples relaciones* aplicamos la estrategia de *pipeline* sobre el método de junta por iteración. Realizar las modificaciones necesarias para aplicar pipeline sobre el método *sort-merge*. ¿Es posible utilizar el método de junta hash con pipeline?

NOTAS BIBLIOGRÁFICAS

Un artículo de Goetz Graefe [110] pasa revista a diversos métodos de optimización de consultas y sirve de punto de entrada a la literatura. Un libro reciente es G. Graefe [111]. Una colección de artículos sobre el tema es la editada por W. Kim, D. S. Reiner, y D. S. Batory [138].

Entre los textos generales de bases de datos merece citarse el tratamiento que del procesamiento de consultas hacen P. O'Neil [167] y R. Ramakrishnan y J. Gehrke [174].

CAPÍTULO 8

CONTROL DEL PARALELISMO Y RECUPERACIÓN

8.1. INTRODUCCIÓN

Los problemas conceptuales encontrados en el diseño de sistemas operativos de multiprogramación demostraron tempranamente el cuidado que requiere el manejo de múltiples procesos operando en paralelo. La literatura de sistemas operativos contiene abundante información sobre técnicas para solucionar problemas tales como sincronización y exclusión mutua entre procesos. Sin embargo, los diseñadores de sistemas de bases de datos que permiten a múltiples procesos acceder a los datos en paralelo han debido construir un nuevo conjunto de técnicas ante los nuevos problemas planteados en este contexto.

Estos problemas se caracterizan por la generación de resultados inconsistentes que, como consecuencia de fallas que interrumpen el procesamiento, permiten el almacenamiento de resultados parciales; errores producidos por la interferencia entre procesos en paralelo o concurrentes, y la imposibilidad de conocer con certeza cuándo los datos que graba una aplicación se registran efectivamente en la base de datos en memoria estable.

Tanto el control del paralelismo como la recuperación serán analizados en un ambiente de base de datos centralizado. Expresamente no consideraremos el caso de una base de datos distribuida.

EJEMPLO 8.1: Supongamos que dos procesos, P_1 y P_2, desean modificar la base de datos. A los fines de nuestro ejemplo, supongamos que la base de datos solamente consiste de dos elementos, X e Y; por el momento es irrelevan-

te si X e Y son registros, tuplas, relaciones, etc. La función de P_1 es simplemente copiar el valor de X a Y, y la función de P_2 es copiar el valor de Y a X. Si ejecutamos primero P_1 y luego P_2, el resultado final es que tanto X como Y reciben el valor original de X, mientras que si ejecutamos primero P_2, el resultado final es que ambos reciben el valor original de Y. En los dos casos, X e Y tienen el mismo valor al finalizar la ejecución. Supongamos en cambio que, para agilizar el desempeño del sistema, permitimos que los pasos de P_1 y P_2 se ejecuten en forma *entrelazada*, en la forma indicada en la Figura 8.1. El efecto de esta ejecución de P_1 y P_2 es intercambiar los valores de X e Y; observamos que X e Y no necesariamente tienen el mismo valor

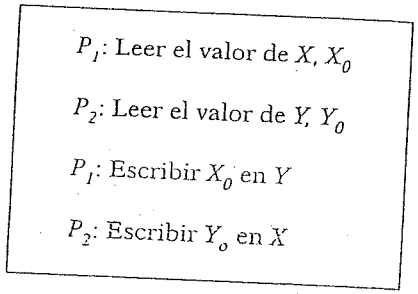


Figura 8.1
Transacciones Entrelazadas

después de esta ejecución. O sea, el haber entrelazado las ejecuciones de P_1 y P_2 hace posibles resultados incorrectos que ninguna ejecución secuencial puede producir.

A fines de discutir los problemas antes planteados, vamos a introducir la noción de *transacción*. Una transacción es una ejecución de un programa de aplicación que accede a la base de datos. El concepto análogo en el área de sistemas operativos es el de *proceso;* la principal distinción es que, a diferencia de los procesos, que suelen ser cíclicos y de duración prolongada, una transacción generalmente es de corta duración. Típicos ejemplos de transacciones serían una reserva o cancelación de pasaje en un vuelo, un depósito, retiro de fondos o transferencia en cuentas bancarias, la inscripción de un nuevo estudiante en una universidad, etc.

Una transacción consiste en una sucesión de *acciones*. Una acción es un paso elemental y atómico de una transacción, es decir, una operación tal que o bien se hace o no se hace, pero no puede hacerse parcialmente. Además, el

sistema sólo puede ejecutar una acción a la vez. O sea, el entrelazamiento de las transacciones se da a nivel de las acciones. Por el momento podemos considerar como acciones las operaciones de leer el valor de un ítem de la base de datos y actualizar el valor de un ítem. Las operaciones y cómputos que realice una transacción internamente, sin acceder a la base de datos, son irrelevantes para nuestro análisis. Por ejemplo, la transacción P_1 en el ejemplo 8.1 consiste de las dos acciones "Leer X" y "Escribir Y". No hemos definido qué es un *ítem* de la base de datos: esto dependerá de la implementación del controlador de paralelismo. Para fijar ideas, conviene pensar en un ítem como si fuera un registro o una tupla.

En general, una transacción contiene dos tipos de operaciones: operaciones de base de datos, tales como *Leer* y *Grabar*, y operaciones de transacción, conocidas como *Abortar* y *Commit*, término este último que no traduciremos con la finalidad de evitar distorsiones semánticas.

En SQL-92 no existe una sentencia especial para indicar el comienzo de una transacción, aunque ha sido incluida en el futuro estándar SQL:1999. En cambio sí existe un comando de terminación, tanto para la finalización normal como para la anormal. En el primer caso se indica con

```
Exec SQL Commit
```

mientras que en el segundo se escribe

```
Exec SQL Rollback.
```

En este segundo caso, la transacción debe abortar ante una situación anormal, por ejemplo la detección de un error irrecuperable, y, mediante el comando Rollback, solicita al sistema que deshaga los cambios a la base de datos realizados hasta el momento.

Una transacción es iniciada implícitamente al ejecutar una sentencia que requiere el contexto de una transacción. Consideraremos entonces una sentencia de *Comienzo de transacción*, sea ésta implícita o explícita. Cada vez que se inicia una transacción el sistema le asigna un identificador único. En adelante, es posible identificar cualquier acción de la transacción por medio de ese identificador.

Con *Commit* se indica que la transacción completó su ejecución exitosamente. En consecuencia, los cambios realizados a los datos resultan permanentes en la bd y visibles a otras transacciones que se ejecutan en paralelo.

El hecho de encapsular las operaciones de base de datos dentro de una

transacción, permite aprovechar las *garantías* que proporciona el sgbd. Esas garantías se conocen como *Propiedades ACID* de una transacción.

8.2. PROPIEDADES ACID

ACID es un acrónimo para las cuatro propiedades que un sgbd garantiza a las transacciones que se ejecutan bajo su control, a saber, Atomicidad, Consistencia, aIslamiento (la I proviene del inglés Isolation) y Durabilidad.

A continuación, examinaremos brevemente cada una de esas propiedades y analizaremos cómo se relacionan entre sí.

Atomicidad

Una transacción, como ya se dijo, consiste de un conjunto de acciones. Pero según esta propiedad, la transacción se ejecuta completamente o no se ejecuta en absoluto. En el primer caso la transacción ejecuta "commit", mientras que en el segundo *aborta*. El sgbd proporciona esta visión de *todo-o-nada* garantizando que si la transacción falla por cualquier motivo, deshará los efectos de las actualizaciones que la transacción haya realizado. Como ejemplo, tomemos el caso de una operación de transferencia entre dos cuentas bancarias. La transacción que implementa dicha operación pudo haberse ejecutado hasta la grabación del saldo de la cuenta *desde*, una vez descontado el monto a transferir; pero antes de grabar el saldo de la cuenta *hacia*, ya incrementado, se produce una falla y la transacción aborta. El resultado parcial producido deja la bd en un estado inconsistente. Afortunadamente, nuestro sgbd garantiza la *atomicidad*, por lo que para nosotros, los usuarios, no se ha realizado ningún cambio a la bd y ésta se encuentra exactamente en el estado en que se encontraba antes de iniciarse la transacción fallida.

Consistencia

De acuerdo con esta propiedad, una transacción transforma un estado consistente de la bd en otro estado, también consistente. Un estado consistente de la bd es aquel que satisface todas las restricciones de integridad. Luego, en principio, esta propiedad de las transacciones depende de cómo está realizado el programa de la transacción y esto es responsabilidad del programador de aplicaciones, independientemente de las otras propiedades. En nuestro ejemplo anterior, la condición de consistencia podría ser que la suma de los saldos, ante una operación de transferencia, se mantenga invariante.

AIslamiento

Por esta propiedad, el sgbd da la imagen de que cada transacción se ejecuta en aislamiento, sin interferencia de otras transacciones. Un conjunto de transacciones se dice *aislado* si el efecto de su ejecución en paralelo es exactamente el mismo que si las transacciones se ejecutaran una a una. La finalidad de la ejecución en paralelo es un mejor aprovechamiento de los recursos computacionales. Otro nombre para esta propiedad es *serializabilidad* y será explicada en detalle en la próxima sección.

Durabilidad

Por esta última propiedad, el sgbd garantiza que las actualizaciones a la bd realizadas por transacciones que ejecutaron *commit* serán durables y públicas. Es decir, los cambios serán registrados en la bd en memoria no volátil, típicamente un disco, y serán visibles a otras transacciones, independientemente de cualquier falla subsiguiente que sufra el sistema.

La propiedad de aislamiento es provista por el mecanismo de *control del paralelismo* de un sgbd y será objeto de análisis en las siguientes secciones. La propiedad de consistencia, como ya se dijo, se satisfará si el programa de aplicación es consistente y el sistema garantiza las demás propiedades. Las otras dos propiedades, atomicidad y durabilidad, son provistas por los componentes que manejan la *recuperación* y serán analizadas a partir de la sección 8.8.

En esta presentación nos limitaremos a analizar transacciones simples, por lo que no consideraremos transacciones con subtransacciones.

8.3. EJECUCIÓN DE TRANSACCIONES

Una *ejecución* de un conjunto de transacciones es una sucesión de las acciones *de* las transacciones con la propiedad de que las acciones de cada transacción aparecen en el mismo orden relativo en que aparecen en la transac-

ción. Por ejemplo, la Figura 8.1 muestra una ejecución de las transacciones P_1 y P_2. Intuitivamente, una ejecución es un entrelazamiento particular de las acciones de un conjunto de transacciones.

Nuestro primer objetivo va a ser caracterizar aquellas ejecuciones entrelazadas que son *correctas* para lo cual necesitamos primero definir nuestro criterio de correctitud. Nuestra suposición fundamental en este capítulo será que una transacción, tomada aisladamente, es correcta, o sea, que dada una base de datos que satisface todos nuestros criterios de consistencia antes de ejecutar la transacción, seguirá satisfaciéndolos después de ejecutada la transacción. Se sigue inmediatamente de esta suposición que la ejecución *serial*, o sea, no entrelazada, de una cantidad arbitraria de transacciones, también será correcta en el mismo sentido. Por lo tanto, vamos a definir como ejecución entrelazada correcta de un conjunto de transacciones a cualquier ejecución *equivalente* a alguna ejecución serial. Nos queda aún por definir cuándo dos ejecuciones de un mismo conjunto de transacciones son equivalentes. Sean E_1, E_2 dos ejecuciones de un conjunto de transacciones $T_1, T_2, \ldots, T_n$. Decimos que E_1 y E_2 son *equivalentes si:*

1. Dado el mismo estado inicial de la base de datos, E_1 y E_2 producen el mismo estado final de la misma.
2. Sea T_i una transacción que ejecuta un cierto paso de lectura de un ítem x en E_1 y E_2; el valor leído por este paso de lectura debe ser el mismo en ambas ejecuciones.

EJEMPLO 8.2: Consideremos dos transacciones T_1 y T_2 que transfieren fondos de una cuenta bancaria a otra. Supongamos que T_1 transfiere \$100 de la cuenta A a la cuenta B y T_2 transfiere \$200 de la cuenta B a la cuenta C. Para esto, cada transacción dispone de tres variables locales, S_A, S_B, S_C. La notación

$$T_1: S_A \leftarrow A$$

significa que la transacción T_1 lee el valor actual de la cuenta A a su variable privada S_A; esto no afecta el valor de las otras dos variables. En cambio, la notación

$$T_2: B \leftarrow S_B + 100$$

indica que la transacción T_2 escribe en la cuenta B el valor de su variable S_B más 100; esto sí afecta el valor de B que verán las otras transacciones.

La Figura 8.2 muestra tres ejecuciones distintas de estas transacciones. La ejecución E_1 es serial: no hay paralelismo. E_2 y E_3 no son seriales. Supongamos que tanto A como B tienen un saldo inicial de \$500 y C tiene un saldo inicial de \$0; entonces, después de ejecutar E_1, el saldo de A será de \$400, el de B será de \$400, y el de C será de \$200. Si en cambio ejecutamos E_2 con los mismos saldos iniciales, obtenemos los mismos saldos finales. Pero si ejecutamos E_3, el saldo final de la cuenta B no es de \$400, sino de \$600. ¿Qué ocurrió? T_1 y T_2 leyeron el valor original del saldo de la cuenta B, y la modificación de este saldo realizada por T_2 fue ignorada por T_1; esta última simplemente sumó \$100 al valor original del saldo. Esto es lo que se llama el problema de "actualización perdida".

E_1	E_2	E_3
$T_1: S_A \leftarrow A$	$T_1: S_A \leftarrow A$	$T_1: S_A \leftarrow A$
$T_1: A \leftarrow S_A - 100$	$T_2: S_B \leftarrow B$	$T_2: S_B \leftarrow B$
$T_1: S_B \leftarrow B$	$T_1: A \leftarrow S_A - 100$	$T_1: A \leftarrow S_A - 100$
$T_1: B \leftarrow S_B + 100$	$T_2: B \leftarrow S_B - 200$	$T_1: S_B \leftarrow B$
$T_2: S_B \leftarrow B$	$T_1: S_B \leftarrow B$	$T_2: B \leftarrow S_B - 200$
$T_2: B \leftarrow S_B - 200$	$T_2: S_C \leftarrow C$	$T_1: B \leftarrow S_B + 100$
$T_2: S_C \leftarrow C$	$T_2: C \leftarrow S_C + 200$	$T_2: S_C \leftarrow C$
$T_2: C \leftarrow S_C + 200$	$T_1: B \leftarrow S_B + 100$	$T_2: C \leftarrow S_C + 200$

Figura 8.2
Tres Ejecuciones

Cuando una ejecución E de un conjunto de transacciones es equivalente a alguna ejecución serial, decimos que E es *serializable*. Dada nuestra suposición de que las transacciones por sí solas preservan la consistencia de la base de datos, podemos concluir que toda ejecución serializable preserva la consistencia de la base de datos. El objetivo del módulo del sistema que controla la ejecución de transacciones en paralelo, a veces llamado el *planificador (scheduler)*, es asegurar que todas las ejecuciones sean serializables. Este control puede ejercerse a dos niveles distintos. Si se ejerce a nivel del sistema, los programadores que escriben transacciones se desentienden de los problemas causados por el paralelismo; si se ejerce a nivel de cada transacción, es la responsabilidad de los programadores garantizar que toda ejecución resultará serializable.

8.4: CANDADOS Y EXCLUSIÓN MUTUA

Los ejemplos vistos sugieren que necesitamos algún mecanismo que garantice a una transacción de acceso exclusivo a ciertos ítems de la base de datos. Por ejemplo, si la transacción T_1 en la Figura 8.2 pudiera evitar que T_2 actualizara la base de datos entre el momento en que T_1 lee el valor de la cuenta B y el momento en que T_1 actualiza este valor, la ejecución incorrecta E_3 no podría ocurrir. Tal mecanismo se conoce como *cierre (locking)*. El objeto de un pedido de cierre sobre un ítem por una transacción es la obtención de un *candado (lock)* sobre ese ítem. Si una transacción T_1 pide cerrar un ítem x, y x está *libre*, o sea, ninguna otra transacción ha adquirido un candado sobre x, entonces T_1 adquiere el candado sobre x y continúa ejecutando. Si otra transacción T_2 intenta cerrar el mismo ítem x antes de que T_1 *libere* x, T_2 queda bloqueada hasta tanto T_1 libere x.

Denotemos el pedido de una candado sobre un ítem x como LL(x) y la liberación de x como LI(x). Para garantizar exclusión mutua, toda transacción debe obedecer las dos reglas siguientes:

1. Una transacción no puede leer ni actualizar un ítem x hasta tanto no haya ejecutado la operación LL(x).
2. Una transacción que intenta obtener una candado sobre un ítem x que ha sido cerrado por otra transacción T debe esperar hasta que T ejecute la operación LI(x).

Volviendo al ejemplo 8.2, vemos que si T_2 ejecuta LL(B) antes de leer B, y no ejecuta *LI(B)* hasta después de haber actualizado B, T_1 se ve obligada a esperar y por lo tanto T_1 lee el valor de B ya actualizado por T_2. Esto implica que la ejecución E_3 no es posible si el sistema obedece las reglas 1 y 2 dadas arriba. Es natural preguntarse si las reglas 1 y 2 son suficientes para garantizar que toda ejecución será correcta, o sea, serializable. El ejemplo siguiente muestra que no es así.

EJEMPLO 8.3: Volvamos al ejemplo 8.1. Tenemos ahora dos transacciones T_1 y T_2 tales que T_1 copia el valor de x a y y T_2 copia el valor de y a x. La ejecución de la Figura 8.3 obedece las reglas 1 y 2 respecto a la adquisición y liberación de candados, pero no es serializable; es básicamente la misma ejecución que en la Figura 8.1.

T_1	T_2
LL(x)	
Leer x	
LI(x)	
—esperar—	LL(y)
—esperar—	Leer y
—esperar—	LI(y)
Ll(y)	
Escribir y	
LI(y)	
	LL(x)
	Escribir x
	LI(x)

Figura 8.3
Ejecución no serializable con candados

De ahora en más, vamos a suponer que toda ejecución de un conjunto de transacciones respeta las reglas 1 y 2. ¿Cómo podemos determinar si una ejecución es serializable o no? Antes de atacar este problema, vamos a refinar nuestro modelo de exclusión mutua. Es claro que si dos transacciones van a leer, pero no a modificar, un cierto ítem, no necesitan excluirse mutuamente. En el modelo que hemos introducido hasta el momento, las reglas 1 y 2 no permiten que una transacción lea un ítem mientras éste está cerrado por otra transacción. Para permitir esto, vamos a distinguir entre dos *modos* de cierre: *modo de lectura (read locking)* y *modo de escritura (write locking)*. La regla 1 no se modifica con este nuevo modelo. Para redefinir la regla 2, digamos que un modo de cierre *conflictúa* con otro si uno de ellos es de escritura. La nueva regla 2 es:

2. Una transacción que desea obtener una candado en modo M sobre un ítem x que ha sido cerrado por otra transacción T con un modo que conflictúa con M, debe esperar hasta que T libere el ítem x.

De ahora en más utilizaremos la notación LL para indicar un cierre de lectura y *LE* para indicar cierre de escritura.

EJEMPLO 8.4: La Figura 8.4 muestra una ejecución de tres transacciones, T_1, T_2 y T_3. Inicialmente, T_1 cierra el ítem A en modo de lectura. Cuando T_3 pide cerrar A en modo de lectura, puede hacerlo, ya que dos candados de lectura no conflictúan entre sí. T_2 en cambio no puede obtener una candado de lectura sobre C una vez que T_3 ha cerrado C en modo de escritura. El lector tal vez se pregunte si la ejecución de la Figura 8.4 es serializable; la respuesta es que sí, puesto que es equivalente a la ejecución serial T_1, T_3,T_2. En la próxima sección daremos un método para determinar la serializabilidad de una ejecución.

8.5. CARACTERIZACIÓN DE EJECUCIONES SERIALIZABLES

Nuestro objetivo en esta sección es presentar un algoritmo que permita decidir, dada una ejecución, si es serializable. ¿Para qué nos servirá este algoritmo? En la práctica, el controlador del paralelismo no va a permitir cualquier ejecución y a posteriori preocuparse de determinar si fue o no serializable. En cambio, el método más común es definir *protocolos de cierre (locking protocols)*. Un protocolo de cierre es un conjunto de reglas sobre el uso de candados tal que, si toda transacción se compromete a seguir estas reglas, el protocolo garantiza que toda ejecución posible es serializable.

T_1	T_2	T_3
LL(A)		
		LL(A)
LE(B)		
LI(A)		
LI(B)		
		LE(C)
		LI(C)
	LL(C)	
		LI(A)
		LE(B)
	LE(A)	
	LI(C)	
	LI(A)	
		LI(B)

Figura 8.4
Ejecución con Dos Modos de Cierre

La caracterización de ejecuciones serializables que vamos a presentar a continuación va a servir como herramienta teórica para demostrar que un protocolo tiene esta propiedad de garantizar serializabilidad.

Dada una ejecución E de un conjunto de transacciones $T_1, \ldots, T_n$, construimos un grafo llamado *grafo de precedencia*. G contiene un nodo por cada transacción T_i. Si T_i ejecuta una operación de cierre de un ítem x en modo M, y T_j ejecuta posteriormente una operación de cierre sobre el mismo ítem x en un modo que conflictúa con x, G contiene un arco que va de T_i a T_j. Intuitivamente, G representa las restricciones que debe satisfacer cualquier ejecución serial, si existe, equivalente a E. Por ejemplo, supongamos que en E la transacción T_i cierra x en modo de lectura, libera x, y luego la transacción T_j cierra x en modo de escritura. Si existe alguna ejecución serial S equivalente a E, T_i debe preceder a T_j en S. De lo contrario, el valor de x leído por T_i sería el valor ya modificado por T_j, mientras que en E era el valor antes de ser modificado por T_j, y por lo tanto E y S no podrían ser equivalentes.

EJEMPLO 8.5: La Figura 8.5 muestra el grafo de precedencia para la ejecución de la Figura 8.4. El grafo tiene un nodo por cada transacción T_1, T_2, T_3. Como T_1 cierra A en modo de lectura antes de que T_2 haga lo mismo en modo de escritura, existe un arco dirigido de T_1 a T_2. Además, T_1 cierra B en modo de escritura antes de que T_3 haga lo mismo; por lo tanto hay un arco de T_1 a T_3. Finalmente, dado que T_3 cierra C para escritura antes de que T_2 cierre C para lectura, hay un arco de T_3 a T_2.

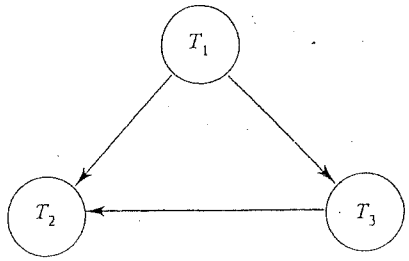


Figura 8.5
Grafo de Predecencia

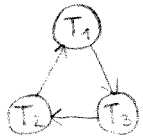
El siguiente teorema fundamental de la serializabilidad caracteriza las ejecuciones serializables en término de la presencia o ausencia de ciclos en el grafo de precedencia.

TEOREMA 8.1 Una ejecución es serializable si y sólo si su grafo de precedencia es acíclico.

DEMOSTRACIÓN: (Si) Si el grafo de precedencia es acíclico, podemos aplicar el método de ordenamiento topológico para listar los nodos en orden de manera que todos los arcos queden dirigidos de izquierda a derecha. El orden resultante representa una ejecución serial que satisface todos los requerimientos de precedencia impuestos por el grafo.

(Sólo si) Si el grafo de precedencia es cíclico, ninguna ejecución serial puede satisfacer todos los requerimientos de precedencia dados. Por lo tanto ninguna ejecución serial es equivalente a la ejecución dada, o sea, ésta no es serializable.

EJEMPLO 8.6: Ya que el grafo de la Figura 8.5 es acíclico, el teorema nos dice que la ejecución de la Figura 8.4 es serializable. En este caso, el único ordenamiento topológico posible del grafo es T_1; T_3; T_2 y ésta es la ejecución serial equivalente a la de la Figura 8.4. Consideremos en cambio la Figura 8.6, que representa otra posible ejecución del mismo conjunto de transacciones T_1, T_2 y T_3. El grafo correspondiente aparece en la Figura 8.7. Vemos que este grafo tiene un ciclo T_1-T_3-T_2 y por lo tanto la ejecución no es serializable. Intuitivamente, lo que ocurre es que, si existiera una ejecución serial equivalente a la dada, en dicha ejecución serial T_1 tendría que aparecer antes que T_3, para que el valor final de B fuera el mismo; T_3 tendría que aparecer antes que T_2, para que T_2 leyera el mismo valor de C y T_3 leyera el mismo valor de A; y T_2 tendría que aparecer antes que T_1, para que T_1 leyera el mismo valor de A. Esto es imposible, y por lo tanto no existe tal ejecución serial.



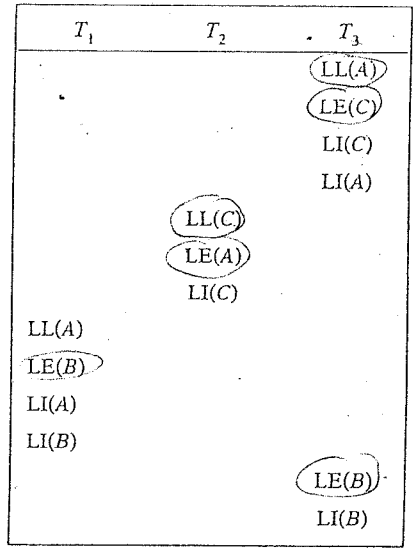
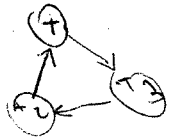


Figura 8.6
Ejecución no serializable



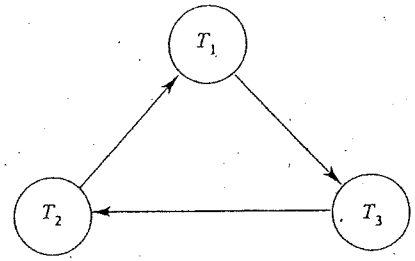


Figura 8.7
Grafo de precedencia cíclico

8.6. CIERRE DE DOS FASES

El protocolo de cierre llamado *de dos fases* consiste en que ninguna transacción puede hacer una operación de cierre una vez que ha liberado algún ítem que había cerrado. O sea, la evolución de una transacción consiste de una primera fase durante la cual la cantidad de ítems cerrados va creciendo a medida que la transacción adquiere nuevos candados, y una segunda fase durante la cual la cantidad de ítems cerrados va decreciendo a medida que la transacción va liberando ítems cerrados. La segunda fase comienza al realizar la transacción su primera operación LI; de allí en más, el protocolo indica que la transacción no puede volver a hacer ninguna operación LL ni LE sobre ningún ítem.

EJEMPLO 8.7: En la Figura 8.3, vemos que ninguna de las transacciones T_1 y T_2 obedece el protocolo de dos fases: T_1 ejecuta LL(y) después de LI(x), y T_2 ejecuta LL(x) después de LI(y). En la Figura 8.4, T_1 y T_2 obedecen el protocolo de dos fases, pero T_3 *cierra B* después de haber liberado *A y C*, violando el protocolo.

El siguiente teorema muestra que este protocolo posee la propiedad que deseamos: si todas las transacciones lo obedecen, tenemos garantizado que toda ejecución será serializable.

TEOREMA 8.2: Supongamos que todas las transacciones T_1, ..., T_n obedecen el protocolo de cierre de dos fases. Toda ejecución de este conjunto de transacciones es serializable.

DEMOSTRACIÓN: Sea E cualquier ejecución de T_1, ..., T_n. Si E no es serializable, sabemos por el Teorema 8.1 que su grafo de precedencia es cíclico. Sea T_{i1}, T_{i2}, ..., T_{ik}, T_{i1} un ciclo en el grafo de precedencia. Por construcción de este grafo, sabemos que T_{i1} ejecutó en E alguna operación de cierre en un modo que conflictuó con una operación de cierre posterior ejecutada por T_{i2}. Por lo tanto, T_{i1} tiene que haber liberado el ítem x antes de que T_{i2} lo haya cerrado.

Por un razonamiento similar, cada T_{ij} tiene que haber liberado algún ítem x_{ij} antes de que T_{ij+1} lo cierre, para $1 \leq j < k$, y T_{ik} tiene que haber liberado algún ítem z antes de que T_{i1} lo cierre. Pero la conjunción de todas estas aserciones implica que T_{i1} liberó el ítem x antes de cerrar el ítem z, por lo cual T_{i1} no obedece el protocolo de dos fases.

El Teorema 8.2 indica que obedecer el protocolo de dos fases es una condición *suficiente* para que toda ejecución de un conjunto de transacciones sea serializable. En cierto sentido, podemos demostrar que es también condición *necesaria*. Esto no significa que si un conjunto de transacciones no es de dos fases, necesariamente existe una ejecución no serializable de dichas transacciones. Por ejemplo, si todas las transacciones ejecutan solamente cierres de lectura, toda ejecución será serializable independientemente de si las transacciones siguen o no el protocolo de dos fases. Pero sí es posible demostrar que, dado cualquier conjunto de transacciones T_1, ..., T_n, si al menos una de ellas no es de dos fases, existe una transacción adicional T_{n+1} tal que es posible ejecutar el conjunto T_1, ..., T_{n+1} en forma no serializable.

Bloqueos por cierre

La introducción del cierre como método de control del paralelismo trae aparejados dos problemas clásicos ya conocidos en el área de sistemas operativos: *bloqueo (deadlock) y postergación indefinida (livelock* o *indefinite postponement)*. Consideremos por ejemplo las transacciones de la Figura 8.3. Supongamos que para evitar la ejecución no serializable de la Figura 8.3, T_1 y T_2 modifican su uso del cierre como lo indica la Figura 8.4. Supongamos que T_1 ejecuta la operación de cierre LL(x), e inmediatamente T_2 ejecuta LL(y). Como las operaciones afectan a dos ítems distintos, ambas transacciones siguen ejecutando. Cuando T_1 intenta ejecutar LL(y), debe esperar, porque y está cerrado por T_2. A su vez, T_2 debe esperar que T_1 libere x antes de poder cerrar x. Pero ninguna de las dos transacciones va a liberar el ítem que tiene cerrado hasta tanto la otra haga lo propio, y, por lo tanto, las dos transac-

ciones quedarán bloqueadas indefinidamente hasta tanto el sistema intervenga de alguna forma. Esto es lo que llamaremos *bloqueo*: una situación en la cual dos o más transacciones están condenadas a esperar indefinidamente la liberación de un conjunto de candados.

Para ver un ejemplo de postergación indefinida, supongamos que una transacción T está esperando que otra transacción T_1 libere un ítem x. Apenas T_1 libera x, y antes que T pueda obtener su candado sobre x, otra transacción T_2 se interpone y obtiene un candado sobre x. Si este fenómeno se repite indefinidamente, T nunca podrá continuar su ejecución.

El problema de postergación indefinida es relativamente fácil de evitar; por ejemplo, basta con poner en cola todos los pedidos de cierre del mismo ítem y despacharlos estrictamente en orden de llegada. El problema de bloqueo es algo más delicado. Existen dos tipos de métodos para solucionar el problema de bloqueo: *detección* y *prevención*. En el método de detección, el sistema examina periódicamente el estado de las transacciones en ejecución. Si comprueba la existencia de dos o más transacciones bloqueadas, el sistema *aborta* una de ellas, denominada *víctima*, la que debe recomenzar su ejecución. Todos los candados en posesión de la transacción abortada son liberados, permitiendo así que las demás transacciones puedan continuar ejecutando. Hay distintos grados de sofisticación en la comprobación del bloqueo: lo más simple es concluir que existe un bloqueo si hay dos o más transacciones que no han salido del estado de espera durante un tiempo superior a cierto umbral. Un método más cuidadoso consiste en mantener lo que se llama un *grafo de espera*. Este grafo tiene un nodo por cada transacción y un arco dirigido de T_i a T_j si T_i está esperando que T_j libere algún candado. No es difícil demostrar que existe una situación de bloqueo si y sólo si este grafo es cíclico.

8.7. NIVELES DE AISLAMIENTO

Los protocolos como el de Cierre de Dos Fases garantizan la ejecución serializable de un conjunto de transacciones imponiendo restricciones que degradan el desempeño de la ejecución. Es decir, el grado de concurrencia, medido como la cantidad de transacciones que se ejecutan en paralelo, puede mantenerse bajo a pesar de que recursos tales como la unidad central de proceso se encuentran en estado de espera mucho tiempo. Unas de las causas de esta situación la constituyen los cierres conflictivos a objetos de la bd. Esto ha llevado a imponer cierta flexibilidad a los planificadores (schedulers)

de los sistemas comerciales y se ha visto reflejado en el estándar SQL-92 a través de la definición de los llamados *niveles de aislamiento*.

Los niveles de aislamiento definidos en SQL-92 son los cuatro siguientes: 1) Lectura No Committed (Read UnCommitted), 2) Lectura Committed (Read Committed), 3) Lectura Repetible (Repeatable Read) y 4) Serializable. También se definieron tres subsecuencias de operaciones que pueden producir un comportamiento anómalo, por ejemplo no serializable, denominadas *fenómenos*: **Lectura Sucia** (Dirty Read), **Lectura No Repetible** (Non-repeatable Read) y **Fantasma** (Phantom).

Describiremos en primer término los tres fenómenos mencionados:

F1. Lectura Sucia: Una transacción T_1 realiza una "lectura sucia" si lee un ítem de dato que ha sido previamente actualizado por otra transacción T_2 y, al momento de la lectura por T_1, T_2 todavía no había llegado a ejecutar commit. Si más tarde T_2 aborta, T_1 ha utilizado un valor que nunca existió.

F2. Lectura No Repetible: Una transacción T_1 realiza una "lectura no repetible" si lee un ítem de dato y, cuando intenta leerlo nuevamente, éste ha sido modificado o aun eliminado por otra transacción T_2 la que, a su vez, ya realizó commit.

F3. Fantasma: Una transacción T_1 lee un conjunto de ítems de datos, por ejemplo un conjunto de filas de una tabla, que satisface una determinada condición de búsqueda. Luego, una transacción T_2 crea nuevos ítems, tales como nuevas filas en la tabla, que también satisfacen la condición de búsqueda, o modifica ítems con el mismo efecto y ejecuta commit. A continuación, si T_1 realiza exactamente la misma consulta, el resultado será distinto del obtenido en la primera oportunidad.

Una transacción que se ejecuta en el primer nivel de aislamiento (Lectura No Committed) exhibe los tres fenómenos descriptos y carece de aislamiento de otras transacciones que se ejecutan concurrentemente. En un sistema que utiliza candados para la sincronización, la transacción que lee no intenta obtener ningún candado y tampoco el sistema verifica que el dato a leer tenga un candado para otra transacción. Este nivel se utiliza en los casos en que las tablas o relaciones son estáticas, cuando la velocidad es más importante que un cien por ciento de exactitud, por ejemplo, en el caso de cálculos estadísticos.

En SQL-92 se declara para este caso

```
Set Transaction Read Only Isolation Level Read Uncommitted...
```

En el segundo nivel de aislamiento (Lectura Committed) una transacción exhibirá los fenómenos de Lectura No Repetible y Fantasma, pero no el de Lectura Sucia. Para una transacción en este nivel de aislamiento se declara

Set Transaction Read Only Isolation Level Read Committed...

En el tercer nivel de aislamiento (Repeatable Read) nos aseguramos que la transacción no exhibirá los dos primeros fenómenos, pero todavía puede exhibir el tercer fenómeno (Fantasma). En SQL-92, para una transacción en este nivel, se debe declarar

Set Transaction Read Only Isolation Level Repeatable Read...

Por último, en el cuarto nivel de aislamiento (Serializable), una transacción no exhibirá ninguno de los tres fenómenos. Luego, el sgbd garantizará que cualquier ejecución con transacciones en este nivel será totalmente serializable. En SQL-92 se declara, en este caso

Set Transaction Read Write Isolation Level Serializable...

Notemos que en los tres primeros casos, en los que no se garantiza la serializabilidad, sólo se admite declarar las transacciones como de lectura solamente o Read Only.

Cabe mencionar que los sgbd comerciales no necesariamente implementan los niveles de aislamiento propuestos en el estándar, sino que, en la mayoría de los casos han implementado su propia versión.

8.8. RECUPERACIÓN TRANSACCIONAL

La **recuperación** se define como la actividad de asegurar que las fallas de software y hardware no corrompan los datos persistentes. En un sgbd esta actividad es realizada por un componente denominado **Gestor de Recuperación** (Recovery Manager).

El Gestor de Recuperación(GR) es el responsable de dos de las propiedades ACID que un sgbd garantiza a toda transacción: *Atomicidad y Durabilidad*. Para esto, el GR asegura que la bd contendrá todos los cambios realizados por las transacciones que hicieron commit (Durabilidad) y ninguno de los cambios de las transacciones que abortaron (Atomicidad).

El GR se ocupa del procesamiento de las operaciones abortar (abort), commit y *reiniciar* (*restart*). La finalidad de reiniciar es restaurar la bd, ante una falla de sistema, a un estado consistente inmediatamente anterior a la falla.

Una transacción es abortada como consecuencia de una *falla*. El GR debe poder recuperar eficientemente a un estado consistente de la bd, ante cualquiera de los siguientes tres tipos de fallas: *de transacción, de sistema y de medio*. El primer tipo ocurre cuando una transacción aborta, por ejemplo, por ella misma al detectar una situación anormal, a causa de un error de programa o por el sgbd al ser seleccionada como víctima en una situación de bloqueo o deadlock.

Como sabemos, existen dos tipos de almacenamiento: i) la memoria volátil o primaria, la que almacena los programas durante su ejecución, así como los datos que éstos utilizan, y ii) la memoria no volátil o *estable*, normalmente un disco, la que almacena la base de datos persistente. Las fallas de sistema son las que afectan de manera impredecible el contenido de la memoria volátil, tales como la interrupción imprevista de la energía, un error de memoria o de procesador, un error del sistema operativo o del sgbd.

Las fallas de medio tienen que ver con la destrucción de alguna parte de la memoria no volátil, por ejemplo, por el "aterrizaje" de las cabezas lectograbadoras sobre la superficie magnetizada de un disco.

En este punto es necesario tener en cuenta lo dicho en el Capítulo 6 sobre manejo de buffers. El área de buffers se denomina también *memoria cache*. El *Gestor de Cache* (GC) o Cache Manager, soporta operaciones llamadas *Fetch(x)* y *Flush(x)*, mediante las cuales lee el bloque que contiene *x* desde el disco (y lo almacena en la cache) y graba al disco la página de cache que contiene *x*, respectivamente.

Consideremos la situación en que tenemos una página en memoria cache conteniendo una o más filas de una tabla. Esa página pudo haber sido actualizada por varias transacciones que ejecutaron *grabar* para filas almacenadas en esa página; pero el algoritmo de reemplazo de páginas definió que esa página permaneciera en memoria cache y no fuera a regrabarse al disco. Luego ambos contenidos son evidentemente diferentes. Más aún, las transacciones que actualizaron la página ejecutaron su commit. ¿Qué ocurre ahora si se produce una falla en la que debe reiniciarse el sistema? Sabemos que el contenido de la memoria volátil se pierde irremediablemente y el contenido de la bd en el disco no había sido actualizado. ¿Cómo hace el sgbd para cumplir con su compromiso de hacer que los cambios de transacciones que hicieron commit permanezcan en la bd? Una solución es utilizar un archivo auxiliar, denominado bitácora o *log*, en el que se registran los cambios que producen las transacciones a la bd.

Las técnicas que describiremos en las próximas secciones se aplican a los

dos primeros tipos de fallas mencionados. Las fallas de medio serán tratadas en la sección 8.13.

8.9. GESTIÓN DE *LOGS* O BITÁCORAS

El *log* es un archivo secuencial que contiene todos los datos relativos a las inserciones, eliminaciones y cambios efectuados a cualquier ítem de la bd. Cada vez que una transacción actualiza, por ejemplo, una fila de una tabla el sgbd genera una entrada en el log.

Con toda la información que se almacena habitualmente en el archivo log, su utilidad trasciende a la del proceso de recuperación. Otros usos significativos son la auditoría de procesos transaccionales, la evaluación del desempeño del sistema en lo relativo a capacidad de procesamiento y tiempos de respuesta, distribución de costos y facturación a los diversos usuarios, etc.

Un componente del sistema llamado *Gestor de Logs (GL)*, o Log Manager, se ocupa de las operaciones de grabación y lectura del log.

Cada registro del log cuenta con un *encabezado* y un *cuerpo*. El encabezado contiene los siguientes datos:

- Número de Secuencia del registro del Log (LSN); es la clave primaria de este archivo y está integrado por el número de archivo más la dirección relativa en bytes del registro en el archivo.
- LSN del registro previo en el log.
- Marca de Tiempo (Timestamp) con tiempo de creación del registro.
- Identificador de la transacción que grabó el registro (TID).
- Registro previo de esta transacción (LSN).

Esta información es de uso del GL. Los LSN's le permiten recorrer el log en ambos sentidos y recuperar los registros correspondientes a cada transacción de manera eficiente.

El cuerpo de un registro del log puede tener un tamaño variable y contiene toda la información necesaria para la recuperación. El registro completo puede ocupar desde unos pocos bytes a varias páginas.

Los archivos en los que reside el log son duplicados y, habitualmente, las copias son grabadas en distintos dispositivos, con el fin de minimizar la posibilidad de pérdida de datos ante cualquier tipo de falla.

Existen dos formas de acceder en modo lectura al log. Ya que éste es considerado una tabla, puede ser consultado por medio de SQL. La otra forma

de acceder es de bajo nivel y permite recuperar un registro por vez, con acceso directo a través del LSN. En el primer caso, por razones de seguridad, es posible que el acceso se realice usando el mecanismo de vistas de SQL.

Para grabar un registro en el log, el GL lo estructura armando el encabezamiento, mientras el cuerpo se recibe como parámetro. Luego le asigna espacio al final del log y lo graba físicamente. En general, los registros son conservados en buffers en la memoria volátil, por razones de eficiencia. Luego, estos registros pueden perderse si se produce una falla.

Normalmente, el contenido del buffer es grabado a la memoria volátil en el momento en que el buffer se completa; pero existen situaciones en que resulta necesario grabar al log inmediatamente, por ejemplo en el caso de un commit, para lo que usa un comando *Flush log*. Recordemos que debido a que el log se encuentra duplicado, las grabaciones físicas deben realizarse dos veces. Éste es un proceso costoso en términos de desempeño, por lo que se trata de optimizar la grabación. Para esto, se graban las páginas que se han completado y se trata de grabar tantas páginas como sea posible en una única operación de grabación física, independientemente de la estrategia adoptada (por ejemplo, LRU).

Un aspecto importante es el orden de grabación de los registros: el log debe reflejar el orden en que se realizaron las operaciones de las transacciones, en particular, las operaciones conflictivas. Estas últimas son las realizadas a un mismo ítem por dos transacciones concurrentes y donde una de ellas es grabar. El motivo de esto es que, ante una caída del sistema, el GR ejecutará un proceso de *rehacer* que permitirá recuperar los efectos de las operaciones, en el caso de las transacciones que llegaron al commit.

Como parte del log, el GR mantiene en memoria estable listas con identificadores de transacciones activas, transacciones que llegaron al commit y transacciones que abortaron.

Si bien muchos sistemas usan el mismo archivo de log para grabar las imágenes previas y las posteriores, es decir, los valores de los ítems previos y posteriores a la actualización, otros usan logs separados. En este último caso, y debido a que las imágenes previas no resultan necesarias después que la transacción ejecutó commit, ya que no se utilizan para la recuperación, pueden ser borradas en breve tiempo. La desventaja de esta estrategia son las grabaciones extras implicadas por el mantenimiento de los dos logs.

Dependiendo del nivel de actividad, un sgbd puede acumular en el log una gran cantidad de registros que deberán ser procesados en la eventualidad de una falla. Sin embargo, los registros antiguos posiblemente no tendrán incidencia en el estado de la bd, pero insumirán tiempo innecesario ante una recuperación. Para reducir el tamaño del log a procesar, el sgbd graba un registro en el log, llamado *punto de verificación o checkpoint*, coincidiendo con la copia a

memoria estable de las imágenes posteriores de las actualizaciones de transacciones que ejecutaron commit y las imágenes previas correspondientes a transacciones abortadas. Daremos más detalle sobre este tópico en la sección 8.12.

Veamos ahora un ejemplo de los datos que se registran en el log ante una ejecución de transacciones.

EJEMPLO 8.8: consideremos la ejecución E_2 de la Figura 8.2 y veamos en la Figura 8.8 qué registros se graban en el log, ante cada acción elemental de las transacciones. Obsérvese que Leer no genera ninguna entrada en el log; pero la primera operación de la transacción genera un registro de comienzo.

Operación	Entrada en el log	
T_1: Leer A, 700	(T_1, Comienzo)	No se registra la lectura, pero por ser la primer acción de T_1 se toma como Comienzo
T_2: Leer B, 850	(T_2, Comienzo)	Ídem anterior
T_1: Grabar A, 600	(T_1, A, 700, 600)	Este registro representa la actualización de A en el log. 700 y 600 son los valores de las imágenes previa y posterior, respectivamente.
T_2: Grabar B, 600	(T_2, B, 850, 600)	Ídem anterior para B. 850 y 600 son los valores de las imágenes previa y posterior.
T_1: Leer B, 600	no se genera entrada en el log	
T_2: Leer C, 500	no se genera entrada en el log	
T_1: Grabar C, 700	(T_1, C, 500, 700)	Actualización de C por T_1, con 500 y 700 los valores de las imágenes previa y posterior.
T_2: Abortar	(T_2, Abortar)	Este registro invoca la ejecución de **deshacer**.
T_1: Commit	(T_1, Commit)	Entrada correspondiente al commit de T_1. Graba el buffer del log en memoria estable.

Figura 8.8

8.10. EL GESTOR DE RECUPERACIÓN

Como se recordará, el Gestor de Recuperación (GR) es el responsable del procesamiento de las operaciones abortar, commit y reiniciar. A continuación describiremos los efectos de cada una de esas operaciones.

- Abortar transacción restaura los valores que tenía un ítem de dato previo a la ejecución de la transacción. Abortar es irrevocable, en el sentido de que una transacción abortada nunca podrá hacer commit.
- Commit transacción deja en firme las actualizaciones realizadas por la transacción en la bd. Commit también es irrevocable: una transacción que ejecutó commit no puede ser luego abortada.
- Reiniciar ejecuta abortar para las transacciones que estaban activas al momento de producirse la falla que motivó su ejecución; graba en la memoria estable las actualizaciones realizadas por transacciones que ejecutaron commit y deshace los cambios de las transacciones abortadas.

Asumiremos que las aplicaciones declaran transacciones encerrando las acciones elementales dentro de un par *Comienzo-Commit* o *Comienzo-Abortar*. Al ejecutarse Comienzo el sgbd le asigna a la transacción un identificador único T_i.

El GR debe respetar una serie de reglas o protocolos tales como **Grabar Antes en el Log** (WAL – Write Ahead Log) y **Forzar al Log en el Commit** (FLC – Force Log at Commit).

Durante el procesamiento normal, cualquier actualización a un ítem de dato genera dos operaciones: una grabación en la copia de la página del ítem que reside en la memoria cache y otra en el buffer del log. En algún momento, no predecible, el algoritmo de reemplazo de páginas puede decidir regrabar esa página en memoria cache a la bd en el disco. Si el GC grabara el contenido del buffer del log al log en memoria estable *después* de regrabar la bd, en caso de producirse una falla, la recuperación sería imposible ya que no habría manera de *deshacer* los cambios realizados por la transacción. Con el fin de evitar esta situación el GR debe respetar el protocolo WAL, por el cual los registros del log son grabados físicamente en memoria no volátil previamente a grabar la página afectada en la bd permanente.

El protocolo WAL se puede detallar por medio de las siguientes reglas:

1. Cada página en memoria cache tiene un LSN (Número de Secuencia del Log) correspondiente al registro del log de la última actualización realizada a esa página.
2. Cada actualización debe mantener el LSN de la página.
3. Cuando una página va a ser copiada (Flush) a memoria no volátil, el GC debe solicitar previamente al Gestor de Logs que copie todos los registros del log residentes en los buffers, incluyendo al correspondiente al LSN de la página, al archivo del log en memoria permanente.
4. Finalizada la grabación del log, la versión de memoria cache de la página puede regrabarse sobre la versión anterior de la página en la bd persistente.

También durante el procesamiento normal, podría ocurrir el siguiente caso. Una transacción llega a su punto de commit, o ejecuta commit, pero todas las páginas que modificó permanecen en memoria cache, del mismo modo que los registros del log en el buffer. Si en esta situación se produjera una falla, en el momento de la recuperación no habría ningún elemento que permitiera *rehacer* los efectos de la transacción. Para evitar esto, como parte de la ejecución del commit, el GR debe forzar la grabación al log no volátil de todos los registros del log que permanezcan en el buffer. Concretamente, el protocolo FLC establece que los registros del log de una transacción deben ser grabados al almacenamiento no volátil como parte del commit de la transacción.

Una transacción se considera que hizo commit en el momento en que su registro de commit se graba en el log, en la memoria no volátil.

Reiniciar

Durante la recuperación, después de una caída del sistema, el GR realiza la operación de reiniciar. Para esto, en primer término, lee el checkpoint más reciente y busca hacia el final del log para recuperarse a sí mismo, reconstruyendo las estructuras de datos necesarias para su funcionamiento eficiente. Luego inicia la tarea de recuperación propiamente dicha, para lo cual aplica alguno de los algoritmos que describiremos en la próxima sección. Por el momento, describiremos las operaciones básicas: **deshacer** (UNDO) y **rehacer** (REDO).

Deshacer

El GR requiere la operación *deshacer* si permite que las actualizacione de una transacción que no llegó al commit sean grabados en la bd persisten te. En la eventualidad de una falla, la bd contendrá esas actualizaciones in debidas. Éstas deberán ser *deshechas* en la operación reiniciar con la final dad de que en la bd sólo permanezcan los efectos de las transacciones qu hicieron commit.

La regla para *deshacer* se expresa por el protocolo WAL: si un ítem de da to en la bd persistente contiene el último valor actualizado por una transac ción que hizo commit, ese valor debe ser resguardado (en el log) antes de ser modificado en la bd persistente por una transacción que no hizo commit.

La operación *deshacer* puede reconstruir un estado anterior de un ítem d la bd a partir del estado actual de ese ítem y registros del log.

Rehacer

El GR requiere la operación *rehacer* si permite que las actualizaciones d una transacción que llegó al commit sean grabadas en la bd persistente después del commit. En la eventualidad de una falla, la bd no contendrá las actualizaciones de algunas transacciones que llegaron al commit. Éstas debe rán ser *rehechas* en la operación reiniciar para restablecer la bd a su estado correcto, es decir, incluir el efecto de todas las transacciones que llegaron a ejecutar commit.

La regla para rehacer expresa que, antes que una transacción pueda ha cer commit, el nuevo valor para el ítem de dato actualizado debe instalarse en el log.

La operación *rehacer* puede construir el nuevo estado de un ítem de la bd a partir de un estado anterior de ese ítem y registros del log.

Idempotencia de Reiniciar

Una falla puede interrumpir el procesamiento de cualquier operación, in cluso de *reiniciar*. Esto impone al GR la restricción que *reiniciar* sea **idempotente**: cualquier secuencia de ejecuciones incompletas seguidas por un ejecución completa de *reiniciar* debe tener el mismo efecto que una única eje cución completa de *reiniciar*.

8.11. ALGORITMOS DE RECUPERACIÓN

Existen diversos algoritmos de recuperación que utilizan el log. En esta sección describiremos uno de ellos, el **Deshacer/Rehacer.** No incluiremos en esta descripción los detalles de cada operación ni consideraremos aspectos de eficiencia.

Algoritmo Deshacer/Rehacer (Undo/Redo)

También denominado "Algoritmo de Recuperación con Actualización Inmediata de la Base de Datos", aplica las reglas enunciadas más arriba para **rehacer** y **deshacer**. Este algoritmo incorpora al log las tres siguientes listas, las que se graban como un archivo secuencial: transacciones activas, transacciones que hicieron commit y transacciones que abortaron.

A continuación describiremos los distintos tipos de registros del log que son utilizados por este algoritmo:

i. **Actualización**: documenta una operación de grabación de una transacción con el siguiente detalle:
   - Encabezamiento, ya descripto en la sección 8.9, incluyendo el LSN de este registro, el identificador de la transacción que ejecutó la actualización, el LSN del registro anterior de la misma transacción, etc.
   - Nombre o dirección en la bd persistente del ítem de dato actualizado.
   - Desplazamiento y longitud de la porción modificada del ítem de dato.
   - El valor anterior de la porción modificada del ítem de dato (imagen previa).
   - El valor nuevo de la porción modificada del ítem de dato (imagen posterior).

ii. **Commit**: expresa que la transacción llegó al commit. Contiene el encabezamiento y el tipo de registro.

iii. **Abortar**: expresa que la transacción abortó. Contiene el encabezamiento y el tipo de registro.

iv. **Checkpoint**: documenta la realización de un checkpoint con el siguiente detalle:

   - Lista de las transacciones activas al momento del checkpoint.
   - Lista de los ítems de datos que fueron modificados y permanecían en memoria cache.

La operación de **reiniciar** procesa el log en dos pasadas. La primera corresponde a **Deshacer** y procesa los registros del log desde final (el registro más reciente) hasta el primer checkpoint que encuentre o, en su defecto, el comienzo del archivo. La segunda pasada corresponde a **Rehacer** y procesa los registros desde el último checkpoint grabado, o el comienzo del archivo, hasta el registro más reciente.

Durante la búsqueda hacia atrás, correspondiente a **deshacer**, *reiniciar* mantiene una lista de transacciones que hicieron commit y una lista de transacciones que abortaron, las que denominaremos Lcom y Labo, respectivamente. **Deshacer** procede de la siguiente forma con cada registro del log:

- si es un registro de *commit*, agrega el identificador de transacción a Lcom;
- si es un registro de *abortar*, agrega el identificador de transacción a Labo;
- si es un registro de *actualización*:
  i. si la transacción está en Lcom, ignora el registro;
  ii. si la transacción no está en Lcom ni en Labo, luego estuvo activa al momento de la falla. Para considerarla abortada la incorpora a Labo;
  iii. si la transacción está ahora en Labo graba la imagen previa al ítem de dato en memoria cache. Si el ítem no está en cache, lo lee desde la bd persistente, por medio de un fetch. Si la transacción no tiene más registros de actualización para deshacer, es eliminada de Labo.

Rehacer realiza una búsqueda hacia delante en el log y, por cada registro de actualización de una transacción en Lcom, actualiza el ítem de dato correspondiente en memoria cache. Si la página que contiene ese ítem no está en cache, es leída con fetch desde la bd persistente. Cuando **rehacer** llega al final del log ha reinstalado todas las actualizaciones de transacciones que hicieron commit.

Al finalizar los dos procesos, el estado combinado de la bd persistente y la memoria cache contiene los efectos de todas las transacciones que llegaron al commit y ninguna actualización de las transacciones abortadas.

8.12. *CHECKPOINTS* O PUNTOS DE CONTROL

Durante el proceso completo de **reiniciar**, no es posible ejecutar ningún tipo de transacción, por lo que resulta importante reducir el tiempo de esta tarea. Una forma de optimización la proporcionan los checkpoints.

Cuanto más frecuentemente se ejecute un checkpoint, menor será la longitud del log a procesar por **reiniciar**, mejorando así los tiempos del proceso. Pero el checkpoint también insume tiempo, por lo que es necesario seleccionar una estrategia adecuada para realizarlo.

Una solución la brinda el **checkpoint borroso** o *fuzzy checkpoint*, denominado así porque se realiza mientras los objetos están siendo modificados. Para realizar un checkpoint borroso, el GR realiza las siguientes actividades:

i. No acepta ninguna nueva operación de actualización, commit o abortar.
ii. Construye una lista de todas las páginas en memoria cache que han sido modificadas (páginas sucias).
iii. Construye una lista de todas las transacciones activas con los LSN que apuntan al último registro grabado del log, para cada transacción.
iv. Graba un registro de checkpoint al log, el que incluye la lista de transacciones activas.
v. Acepta nuevamente las operaciones de actualización, commit y abortar.
vi. Mientras ejecuta las operaciones del paso v, realiza *flush* a la bd permanente de todas las páginas sucias de la lista construida en el paso ii. Esta tarea es realizada por el Gestor de Cache en la medida que disponga de tiempo libre.

El siguiente checkpoint sólo se puede realizar una vez que el GR terminó con el paso vi, es decir, después que las páginas sucias han sido regrabadas en el disco.

8.13. RECUPERACIÓN DE MEDIO

Hasta el momento hemos asumido que las fallas que pueden ocurrir sólo afectaban a la memoria volátil. Pero, como se mencionó en la sección 8.8, pueden ocurrir fallas de medio que las podemos definir como la pérdida de una parte de la memoria estable o permanente. En este caso, se debe recurrir a una copia de resguardo de la bd, generalmente una cinta o *cartridge*. Con esta copia y el log, normalmente sus copias de respaldo más el log actual, es posible recuperar la bd a un estado actualizado por medio de un proceso similar al que se usa para recuperar en los casos de falla de sistema.

Debido a que la solución recién descripta puede insumir demasiado tiempo para ciertos ambientes de procesamiento en línea, algunos sistemas usan discos espejados (mirrored disks). En este caso, se utilizan dos discos físicos para cada disco lógico, por lo que cada disco tiene una copia de respaldo actualizada al instante.

Una tecnología más general es RAID (Redundant Array of Inexpensive Disks), la que usa varios discos y admite que uno o más de ellos fallen. Esto es menos oneroso que la redundancia total, aunque un sistema RAID es más costoso que discos únicos de la misma capacidad. Existen diversas categorías de RAID, de las cuales mencionamos tres: RAID 0 es "disk stripping" o grabación de datos a través de varias unidades de disco. RAID 1 es almacenamiento cien por ciento redundante. RAID 5 distribuye los datos e información de paridad sobre varios discos, con redundancia suficiente como para alcanzar muy altos niveles de tolerancia a fallas. Se considera que para bases de datos muy grandes estas tecnologías pueden ser muy costosas, pero al menos podrían aplicarse al log.

Copias de Archivo

Como mencionamos antes, la recuperación en caso de falla de medio requiere mantener una copia de respaldo de la bd y una copia del log. Esta última debe contener todas las actualizaciones realizadas por transacciones que se ejecutaron después de realizar la copia de respaldo de la bd y que hicieron commit.

La creación de la copia de la bd consiste simplemente en copia la bd permanente, en su totalidad, a otro medio. Pero esto puede consumir demasiados recursos del sistema. Una alternativa es hacer copias diferenciales: se realiza una copia de la bd íntegra y luego, en lugar de hacer nuevas copias completas, sólo se archivan las páginas que han sido modificadas.