# PARTE 5

# Conceptos del procesamiento de transacciones

CAPÍTULO 17

# Introducción a los conceptos y la teoría sobre el procesamiento de transacciones

El concepto de transacción proporciona un mecanismo para definir las unidades lógicas del procesamiento de una base de datos. Los **sistemas de procesamiento de transacciones** son sistemas con grandes bases de datos y cientos de usuarios concurrentes ejecutando transacciones de bases de datos. Entre estos sistemas podemos citar los de reservas en aerolíneas, banca, procesamiento de tarjetas de crédito, mercado de acciones, etcétera. Estos sistemas requieren una alta disponibilidad y una respuesta rápida para cientos de usuarios simultáneos. En este capítulo presentamos los conceptos relacionados con los sistemas de procesamiento de transacciones. Definimos el concepto de transacción, que se utiliza para representar una unidad lógica de procesamiento de base de datos que debe completarse en su totalidad para garantizar su exactitud. También explicamos el problema de controlar la concurrencia, que surge cuando varias transacciones enviadas por varios usuarios interfieren entre sí de un modo que produce unos resultados incorrectos. Asimismo, explicamos cómo recuperarnos de los fallos en las transacciones.

La Sección 17.1 explica por qué el control de la concurrencia y la recuperación son necesarios en un sistema de bases de datos. La Sección 17.2 introduce el concepto de transacción y explica otros conceptos relacionados con el procesamiento de transacciones. La Sección 17.3 presenta los conceptos de atomicidad, conservación de la consistencia (o de la coherencia), aislamiento, y durabilidad o permanencia (denominados propiedades ACID), que se consideran deseables en las transacciones. La Sección 17.4 introduce el concepto de la planificación (*schedule*) de la ejecución de transacciones y tipifica la recuperabilidad de las planificaciones. La Sección 17.5 explica el concepto de serialización (o seriabilidad) de las ejecuciones de transacciones concurrentes, que también se puede utilizar para definir las secuencias de ejecución correctas (o planificaciones) de las transacciones concurrentes. La Sección 17.6 presenta los servicios que soportan el concepto de transacción en SQL.

Los dos capítulos siguientes continúan con más detalles sobre las técnicas utilizadas para el procesamiento de transacciones. El Capítulo 18 describe las técnicas básicas de control de la concurrencia, y el Capítulo 19 presenta una panorámica de las técnicas de recuperación.

## 17.1 Introducción al procesamiento de transacciones

En esta sección introducimos, informalmente, los conceptos de ejecución concurrente de transacciones y recuperación ante el fallo de una transacción. La Sección 17.1.1 compara los sistemas de bases de datos mono-

usuario y multiusuario, y muestra cómo la ejecución concurrente de transacciones puede tener lugar en los sistemas multiusuario. La Sección 17.1.2 define el concepto de transacción y presenta un modelo sencillo de ejecución de transacciones (basado en operaciones de lectura y escritura en bases de datos) que se utiliza para formalizar los conceptos de control de la concurrencia y recuperación. La Sección 17.1.3 muestra, mediante ejemplos informales, por qué son necesarias las técnicas de control de la concurrencia en los sistemas multiusuario. Por último, la Sección 17.1.4 explica por qué se necesitan técnicas que permitan la recuperación ante un fallo, y lo hace explicando las distintas maneras en que las transacciones pueden fallar mientras se ejecutan.

### 17.1.1 Sistema monousuario frente a sistema multiusuario

Un criterio que sirve para clasificar un sistema de bases de datos es el número de usuarios que lo pueden utilizar **al mismo tiempo**. Un DBMS es **monousuario** si sólo lo puede utilizar un usuario a la vez, y es **multiusuario** si varios usuarios pueden utilizar el sistema (y, por tanto, acceder a la base de datos) simultáneamente. Los DBMSs monousuario están principalmente restringidos a los sistemas de computación personal; la mayoría de los demás DBMSs son multiusuario. Por ejemplo, un sistema de reservas en aerolíneas lo utilizan simultáneamente cientos de agentes de viajes. Los sistemas de bancos, aseguradoras, supermercados, etcétera, también operan con muchos usuarios que envían transacciones simultáneamente al sistema.

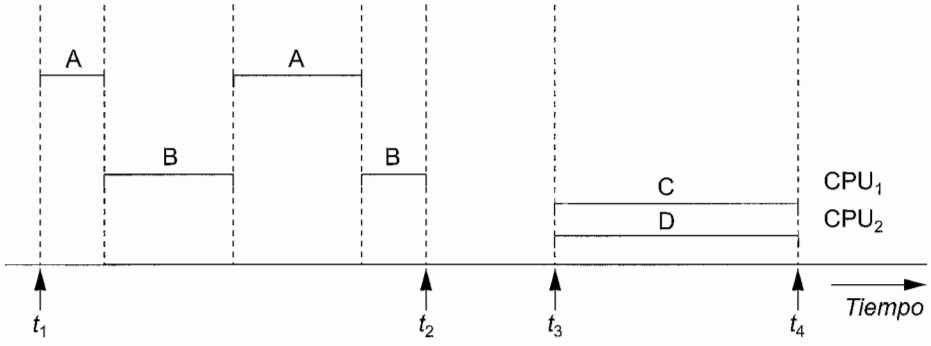
A la base de datos pueden acceder simultáneamente múltiples usuarios (y utilizar los computadores) debido al concepto de **multiprogramación**, que permite al computador ejecutar varios programas (o **procesos**) al mismo tiempo. Si sólo hay una unidad central de procesamiento (CPU), realmente sólo se ejecuta un proceso a la vez. Sin embargo, los **sistemas operativos multiprogramación** ejecutan algunos comandos de un proceso, después suspenden ese proceso y ejecutan algunos comandos del siguiente proceso, y así sucesivamente. El proceso se reanuda en el mismo punto en que se suspendió, siempre que consiga turno para utilizar de nuevo la CPU. Por tanto, la ejecución concurrente de procesos realmente es **interpolada**, como se ilustra en la Figura 17.1, que muestra dos procesos A y B ejecutándose concurrentemente en modo interpolado. La interpolación mantiene ocupada la CPU cuando un proceso requiere una operación de entrada o salida (E/S), como la lectura de un bloque del disco. La CPU pasa a ejecutar otro proceso en lugar de permanecer ociosa durante el tiempo de E/S. La interpolación también evita que un proceso largo retrase otros procesos.

Si el computador tiene varios procesadores (CPUs), es posible el **procesamiento paralelo** de varios procesos, como se ilustra en la Figura 17.1 con los procesos C y D. La mayoría de la teoría relativa al control de la concurrencia en las bases de datos está desarrollada en términos de **concurrencia interpolada**, por lo que en el resto de este capítulo asumimos este modelo. En un DBMS multiusuario, los elementos de datos almacenados son los recursos principales a los que pueden acceder los usuarios interactivos o programas de aplicación (que están recuperando y modificando constantemente información de la base de datos).

### 17.1.2 Transacciones, operaciones de lectura y escritura y búferes DBMS

Una **transacción** es un programa en ejecución que constituye una unidad lógica del procesamiento de una base de datos. Una transacción incluye una o más operaciones de acceso a la base de datos (operaciones de inserción, eliminación, modificación o recuperación). Las operaciones con bases de datos que forman una transacción pueden estar incrustadas dentro de una aplicación o pueden especificarse interactivamente mediante un lenguaje de consulta de alto nivel como SQL. Una forma de definir los límites de una transacción es especificando explícitamente las sentencias **begin transaction** y **end transaction** en un programa de aplicación; en este caso, todas las operaciones de acceso a la base de datos que se encuentran entre las dos sentencias son consideradas como una transacción. Un programa de aplicación puede contener más de una transacción si contiene varios límites de transacción. Si las operaciones de bases de datos de una transacción no actualizan la base de datos, sino que únicamente recuperan datos, se dice que la **transacción es de sólo lectura**.

**Figura 17.1.** Procesamiento interpolado frente a procesamiento paralelo de transacciones concurrentes.




El modelo de una base de datos que se utiliza para explicar los conceptos de procesamiento de transacciones es muy simplificado. Una **bases de datos** está representada básicamente como una colección de **elementos de datos con nombre**. El tamaño de un elemento de datos se conoce como **granularidad**. Puede ser un campo de algún registro de la base de datos, o puede ser una unidad más grande, como un registro o incluso un bloque de disco entero, pero los conceptos que explicamos son independientes de la granularidad del elemento de datos. Con este modelo de base de datos simplificado, las operaciones básicas de acceso a la base de datos que una transacción puede incluir son las siguientes:

- **read_item(*X*)** (leer elemento). Lee un elemento de base de datos denominado *X* y lo almacena en una variable de programa. Para simplificar nuestra notación, asumimos que *la variable de programa también se llama X.*
- **write_item(*X*)** (escribir elemento). Escribe el valor de la variable de programa *X* en un elemento de base de datos denominado *X*.

Como explicamos en el Capítulo 13, la unidad básica de transferencia de datos desde el disco a la memoria principal es un bloque. La ejecución de un comando read_item(*X*) incluye estos pasos:

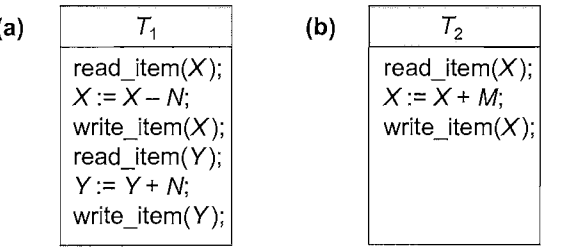
1. Encontrar la dirección del bloque de disco que contiene el elemento *X*.
2. Copiar ese bloque de disco en un búfer de la memoria principal (si dicho bloque no se encuentra ya en algún búfer de la memoria principal).
3. Copiar el elemento *X* desde el búfer a la variable de programa *X*.

La ejecución de un comando write_item(*X*) abarca estos pasos:

1. Encontrar la dirección del bloque de disco que contiene el elemento *X*.
2. Copiar ese bloque de disco en un búfer de la memoria principal (si dicho bloque no se encuentra ya en algún búfer de la memoria principal).
3. Copiar el elemento *X* desde la variable de programa *X* a su ubicación correcta en el búfer.
4. Almacenar el bloque actualizado desde el búfer al disco (puede ser inmediatamente o en un momento posterior en el tiempo).

El paso 4 realmente actualiza la base de datos en el disco. En algunos casos, el búfer no se almacena inmediatamente en el disco, en caso de haber introducido cambios adicionales en el búfer. Normalmente, la decisión sobre cuándo almacenar un bloque de disco modificado que se encuentra en un búfer de la memoria principal la toma el gestor de recuperación del DBMS en cooperación con el sistema operativo subyacente. El DBMS generalmente mantendrá varios **búferes** de la memoria principal para albergar los bloques de disco de la base de datos que contienen los elementos de datos que se están procesando. Cuando todos estos búferes están ocupados, y es preciso copiar otros bloques de la base de datos en la memoria, se utiliza alguna polí-

**Figura 17.2.** Dos ejemplos de transacciones. (a) Transacción $T_1$. (b) Transacción $T_2$.




tica de sustitución de búferes para elegir cuál de ellos sustituir. Si el búfer elegido se ha modificado, debe escribirse en el disco antes de su reutilización.[1]

Una transacción incluye operaciones read_item y write_item para acceder y actualizar la base de datos. La Figura 17.2 muestra ejemplos de dos transacciones muy sencillas. El **conjunto de lectura** (*read-set*) de una transacción es el conjunto de todos los elementos que la transacción lee, y el **conjunto de escritura** (*write-set*) es el conjunto de todos los elementos que la transacción escribe. Por ejemplo, el conjunto de lectura de $T_1$ en la Figura 17.2 es $\{X, Y\}$ y su conjunto de escritura también es $\{X, Y\}$.

Los mecanismos de control de la concurrencia y de recuperación están principalmente ligados a los comandos de acceso a la base de datos en una transacción. Las transacciones emitidas por varios usuarios pueden ejecutarse concurrentemente y pueden acceder y actualizar los mismos elementos de la base de datos. Si esta ejecución concurrente no está controlada, pueden surgir problemas, como una base de datos inconsistente (incoherente). En la siguiente sección introducimos algunos de los problemas que pueden surgir.

### 17.1.3 Por qué es necesario controlar la concurrencia

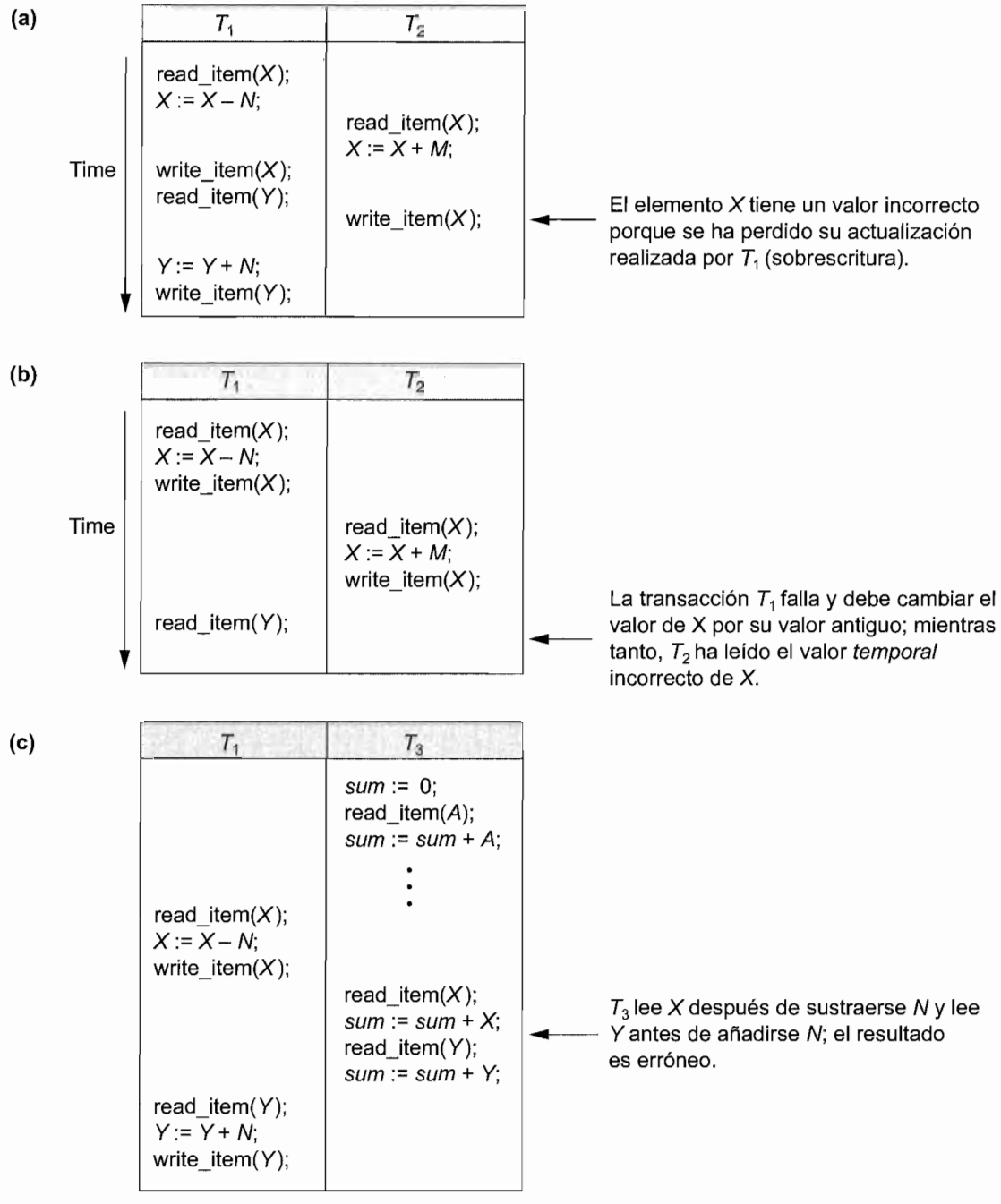
Cuando las transacciones se ejecutan de manera incontrolada pueden surgir algunos problemas. Vamos a ilustrarlos haciendo referencia a una base de datos de reservas en aerolíneas simplificada en la que se almacena un registro por cada vuelo de una aerolínea. Cada registro incluye el número de plazas reservadas en ese vuelo como un *elemento de datos con nombre*, junto con otra información. La Figura 17.2(a) muestra una transacción $T_1$ que *transfiere N* reservas de un vuelo cuyo número de plazas reservadas está almacenado en el elemento de base de datos $X$ a otro vuelo cuyo número de plazas reservadas se almacena en el elemento de base de datos $Y$. La Figura 17.2(b) muestra una transacción $T_2$ más sencilla que *reserva M* plazas en el primer vuelo ($X$) al que se hace referencia en la transacción $T_1$.[2] Para simplificar nuestro ejemplo, no mostramos ciertas porciones de las transacciones, como la comprobación de si un vuelo tiene suficientes plazas disponibles antes de efectuar la reserva de plazas adicionales.

Cuando se escribe un programa de acceso a una base de datos, tiene como parámetros los números de vuelo, sus fechas y el número de plazas que se pueden reservar; por tanto, el mismo programa puede utilizarse para ejecutar muchas transacciones, cada una con diferentes vuelos y cantidades de plazas a reservar. A fin de controlar la concurrencia, una transacción es una *ejecución particular* de un programa en una fecha, vuelo y número de plazas dadas. En la Figura 17.2(a) y (b), las transacciones $T_1$ y $T_2$ son *ejecuciones específicas* de los programas que se refieren a vuelos específicos cuyos números de plazas se almacenan en los elementos

[1] No explicaremos las políticas de sustitución de búferes porque normalmente se explican en los libros dedicados a los sistemas operativos.

[2] Un ejemplo parecido y muy utilizado es el de la base de datos de un banco, con una transacción haciendo una transferencia de fondos de la cuenta $X$ a la cuenta $Y$, y otra transacción haciendo un depósito en la cuenta $X$.

**Figura 17.3.** Algunos problemas que surgen cuando no se controla la concurrencia. (a) Problema por pérdida de actualización. (b) Problema de la actualización temporal. (c) Problema de la suma incorrecta.




de datos $X$ e $Y$ de la base de datos. A continuación explicamos los tipos de problemas que nos podemos encontrar con estas transacciones si se ejecutan simultáneamente.

**Problema por pérdida de actualización.** Este problema surge cuando dos transacciones que acceden a los mismos elementos de la base de datos tienen sus operaciones interpoladas de un modo que hace que el

valor de algunos elementos de la base de datos sean incorrectos. Suponga que las transacciones $T_1$ y $T_2$ se envían aproximadamente al mismo tiempo, y suponga que sus operaciones están interpoladas como se aprecia en la Figura 17.3(a); el valor final del elemento $X$ es incorrecto porque $T_2$ lee el valor de $X$ *antes* de que $T_1$ lo cambie en la base de datos; por tanto, se pierde el valor actualizado resultante de $T_1$. Por ejemplo, si $X = 80$ al principio (originalmente, el vuelo tiene 80 reservas), $N = 5$ ($T_1$ transfiere 5 reservas de plaza del vuelo correspondiente a $X$ al vuelo correspondiente a $Y$), y $M = 4$ ($T2$ reserva 4 plazas en $X$), el resultado final debe ser $X = 79$; pero en la interpolación de operaciones de la Figura 17.3(a), el resultado es $X = 84$ porque se ha perdido la actualización de $T_1$ que eliminaba las cinco plazas de $X$.

**Problema de la actualización temporal (o lectura sucia).** Este problema ocurre cuando una transacción actualiza un elemento de la base de datos y, después, falla por alguna razón (consulte la Sección 17.1.4). El elemento actualizado es accedido por otra transacción antes de cambiar a su valor original. La Figura 17.3(b) muestra un ejemplo donde $T_1$ actualiza el elemento $X$ y después falla antes de concluir, por lo que el sistema debe devolver $X$ a su valor original. Sin embargo, antes de poder hacerlo, la transacción $T_2$ lee el valor *temporal* de $X$, que no se grabará permanentemente en la base de datos debido al fallo de $T_1$. El valor del elemento $X$ que es leído por $T_2$ se denomina *dato sucio* porque lo ha creado una transacción que no se ha completado y confirmado todavía; por tanto, este problema también se conoce como *problema de la lectura sucia.*

**El problema de la suma incorrecta.** Si una transacción está calculando una función de suma agregada sobre varios registros, mientras otras transacciones están actualizando algunos de esos registros, la función agregada puede calcular algunos valores antes de que sean actualizados y otros después de ser actualizados. Por ejemplo, suponga que una transacción $T_3$ está calculando el número total de reservas en todos los vuelos; mientras tanto, se está ejecutando la transacción $T_1$. Si se produce la interpolación de operaciones de la Figura 17.3(c), el resultado de $T_3$ estará desfasado una cantidad $N$ porque $T_3$ lee el valor de $X$ *después* de que se hayan sustraído $N$ plazas de ella, pero lee el valor de $Y$ *antes* de que esas $N$ plazas se hayan añadido a él.

Otro problema que puede surgir es la denominada **lectura irrepetible**, en la que una transacción $T$ lee un elemento dos veces y el elemento es modificado por otra transacción $T'$ entre las dos lecturas. Por tanto, $T$ recibe *valores diferentes* en sus dos lecturas del mismo elemento. Esto sucede, por ejemplo, si durante una tran-sacción de reserva en una aerolínea, un cliente busca información sobre la disponibilidad de plaza en varios vuelos. Cuando el cliente opta por un vuelo en particular, la transacción lee el número de plazas de ese vuelo una segunda vez antes de completar la reserva.

## 17.1.4 Por qué es necesaria la recuperación

Siempre que se envía una transacción a un DBMS para su ejecución, el sistema es responsable de garantizar que todas las operaciones de la transacción se completen satisfactoriamente y que su efecto se grabe permanentemente en la base de datos, o de que la transacción no afecte a la base de datos o a cualquier otra transacción. El DBMS no debe permitir que algunas operaciones de una transacción $T$ se apliquen a la base de datos mientras otras no. Esto puede ocurrir si una transacción **falla** después de ejecutar algunas de sus operaciones, pero antes de ejecutar todas ellas.

**Tipos de fallos.** Los fallos se clasifican generalmente como fallos de transacción del sistema y del medio. Hay varias razones posibles por las que una transacción puede fallar en medio de su ejecución:

1. **Un fallo del computador (caída del sistema).** Durante la ejecución de una transacción se produce un error del hardware, del software o de la red. Las caídas del hardware normalmente se deben a fallos en los medios (por ejemplo, un fallo de la memoria principal).
2. **Un error de la transacción o del sistema.** Alguna operación de la transacción puede provocar que falle, como un desbordamiento de entero o una división por cero. El fallo de una transacción también

se puede deber a unos valores erróneos de los parámetros o debido a un error lógico de programación.[3] Además, el usuario puede interrumpir la transacción durante su ejecución.

3. **Errores locales o condiciones de excepción detectados por la transacción.** Durante la ejecución de una transacción, se pueden dar ciertas condiciones que necesitan cancelar la transacción. Por ejemplo, puede que no se encuentren los datos para la transacción. Una condición de excepción,[4] como un saldo de cuenta insuficiente en una base de datos bancaria, puede provocar que una transacción, como la retirada de fondos, sea cancelada. Esta excepción debe programarse en la propia transacción, en cuyo caso, no sería considerada un fallo.
4. **Control de la concurrencia.** El método de control de la concurrencia (consulte el Capítulo 18) puede optar por abortar la transacción, para restablecerla más tarde, porque viola la serialización (consulte la Sección 17.5) o porque varias transacciones se encuentran en estado de bloqueo.
5. **Fallo del disco.** Algunos bloques del disco pueden perder sus datos debido a un mal funcionamiento de la lectura o la escritura o porque se ha caído la cabeza de lectura/escritura del disco. Esto puede ocurrir durante una operación de lectura o escritura de la transacción.
6. **Problemas físicos y catástrofes.** Se refiere a una lista interminable de problemas que incluye fallos de alimentación o aire acondicionado, fuego, robo, sabotaje, sobrescritura de discos y cintas por error, y montaje de la cinta errónea por parte del operador.

Los fallos de los tipos 1, 2, 3 y 4 son más comunes que los de los tipos 5 o 6. Siempre que se produce un fallo de tipo 1 a 4, el sistema debe guardar información suficiente para recuperarse del fallo. El fallo de un disco u otros fallos catastróficos de tipo 5 o 6 no se dan con frecuencia; si ocurren, la recuperación es la tarea principal. En el Capítulo 19 explicamos cómo recuperarnos de un fallo.

El concepto de transacción es fundamental para muchas técnicas de control de la concurrencia y de recuperación ante fallos.

# 17.2 Conceptos de transacción y sistema

En esta sección explicamos conceptos adicionales relativos al procesamiento de transacciones. La Sección 17.2.1 describe los distintos estados de una transacción y explica las operaciones pertinentes adicionales necesarias en su procesamiento. La Sección 17.2.2 explica el registro del sistema, que guarda información necesaria para la recuperación. La Sección 17.2.3 describe los puntos de confirmación (*commit points*) de las transacciones, y por qué son importantes en el procesamiento de las mismas.

## 17.2.1 Estados de una transacción y operaciones adicionales

Una transacción es una unidad atómica de trabajo que se completa en su totalidad o no se lleva a cabo en absoluto. Para fines de recuperación, el sistema debe hacer el seguimiento de cuándo se inicia, termina y confirma o aborta una transacción (consulte la Sección 17.2.3). Por consiguiente, el gestor de recuperación hace un seguimiento de las siguientes operaciones:

- BEGIN_TRANSACTION. Marca el inicio de la ejecución de una transacción.
- READ o WRITE. Especifican operaciones de lectura o escritura en los elementos de la base de datos que se ejecutan como parte de una transacción.

[3] En general, una transacción debe probarse cuidadosamente para asegurarnos de que no tiene errores (errores de programación lógicos).

[4] Las condiciones de excepción, si se programan correctamente, no constituyen fallos de transacción.

- END_TRANSACTION. Especifica que las operaciones READ y WRITE de la transacción han terminado y marca el final de la ejecución de la transacción. Sin embargo, en este punto puede ser necesario comprobar si los cambios introducidos por la transacción pueden aplicarse de forma permanente a la base de datos (confirmados) o si la transacción se ha cancelado porque viola la serialización (consulte la Sección 17.5) o por alguna otra razón.
- COMMIT_TRANSACTION. Señala una *finalización satisfactoria* de la transacción, por lo que los cambios (actualizaciones) ejecutados por la transacción se pueden **enviar** con seguridad a la base de datos y no se desharán.
- ROLLBACK (o ABORT). Señala que la transacción *no ha terminado satisfactoriamente*, por lo que deben *deshacerse* los cambios o efectos que la transacción pudiera haber aplicado a la base de datos.

La Figura 17.4 muestra un diagrama que describe los estados de ejecución de una transacción. Una transacción entra en **estado activo** inmediatamente después de iniciarse su ejecución; en este estado puede emitir operaciones READ y WRITE. Cuando la transacción termina, pasa al **estado de parcialmente confirmada**. En este punto, se necesitan algunos protocolos de recuperación para garantizar que un fallo del sistema no supondrá la imposibilidad de registrar los cambios de la transacción de forma permanente (normalmente, grabando los cambios en el registro del sistema, que explicamos en la siguiente sección).[5] Una vez que esta comprobación es satisfactoria, se dice que la transacción ha alcanzado su punto de confirmación y entra en el **estado de confirmada**. Los puntos de confirmación se explican en detalle en la Sección 17.2.3. Una vez confirmada la transacción, ha concluido satisfactoriamente su ejecución y todos sus cambios deben grabarse permanentemente en la base de datos.

No obstante, una transacción puede entrar en el **estado de fallo** si falla una de las comprobaciones o si la transacción es cancelada durante su estado activo. Entonces es posible anular la transacción para deshacer el efecto de sus operaciones de escritura en la base de datos. El **estado terminado** se alcanza cuando la transacción abandona el sistema. La información relativa a la transacción que se guarda en tablas del sistema mientras se está ejecutando, se elimina cuando la transacción termina. Las transacciones fallidas o canceladas pueden *restablecerse* más tarde (automáticamente o después de haber sido reenviadas por el usuario) como transacciones completamente nuevas.

## 17.2.2 El registro del sistema (*log*)

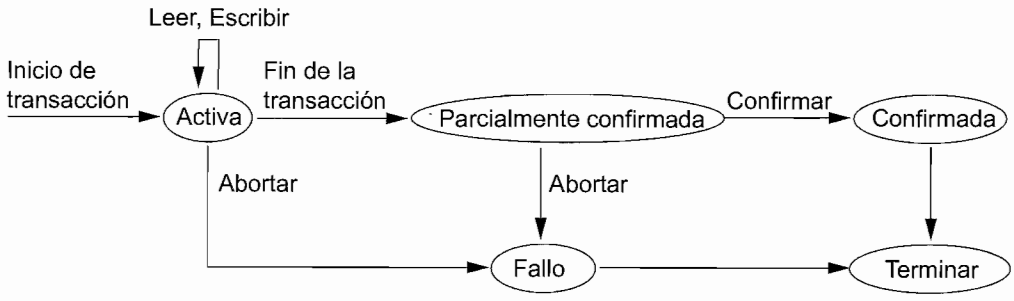
Para poder recuperarse de los fallos que afectan a las transacciones, el sistema mantiene un **registro**[6] para hacer un seguimiento de todas las operaciones de las transacciones que afectan a los valores de los elementos de una base de datos. Esta información puede ser necesaria para recuperarse ante un fallo. Este registro se guarda en el disco, por lo que no se ve afectado por ningún tipo de fallo, excepto los de disco o catastróficos. Además, periódicamente se realiza una copia de seguridad del registro para protegerse contra dichos fallos catastróficos. La siguiente lista muestra los tipos de entradas (denominadas **entradas del registro**) que se escriben en el registro y la acción que cada una de ellas lleva a cabo. En estas entradas, $T$ se refiere a un **id de transacción** único que el sistema genera automáticamente y que utiliza para identificar las transacciones:

1. [**start_transaction**, $T$]. Indica que se ha iniciado la ejecución de la transacción $T$.
2. [**write_item**, $T$, $X$, *valor_antiguo*, *valor_nuevo*]. Indica que la transacción $T$ ha cambiado el valor del elemento $X$ de *valor_antiguo* a *valor_nuevo*.

[5] Un control optimista de la concurrencia (consulte la Sección 18.4) también requiere que en este punto se realicen ciertas comprobaciones que garanticen que la transacción no interfiere con otras transacciones en ejecución.

[6] El registro se ha denominado a veces diario del DBMS.

**Figura 17.4.** Diagrama de estado de una transacción ilustrando los estados de su ejecución.




3. [**read_item**, *T*, *X*]. Indica que la transacción *T* ha leído el valor del elemento *X* de la base de datos.
4. [**commit**, *T*]. Indica que la transacción *T* se ha completado satisfactoriamente, y afirma que su efecto puede confirmarse (grabarse permanentemente) en la base de datos.
5. [**abort**, *T*]. Indica que la transacción *T* se abortó.

Los protocolos para recuperación que evitan las anulaciones en cascada (consulte la Sección 17.4.2), entre los que podemos citar casi todos los protocolos prácticos, no requieren que las operaciones de lectura se escriban en el registro del sistema. Sin embargo, si el registro también se utiliza con otros fines (como la auditoría, que hace un seguimiento de todas las operaciones de bases de datos), entonces pueden incluirse esas entradas. Además, algunos protocolos de recuperación requieren entradas WRITE más sencillas que no incluyen *valor_nuevo* (consulte la Sección 17.4.2).

Observe que estamos asumiendo que todos los cambios permanentes en la base de datos ocurren dentro de las transacciones, por lo que la noción de recuperarse ante el fallo de una transacción equivale a deshacer o rehacer individualmente las operaciones de esa transacción desde el registro. Si el sistema se cae, podemos recuperar un estado consistente de la base de datos examinando el registro y utilizando una de las técnicas que se describen en el Capítulo 19. Como el registro del sistema contiene un registro o entrada por cada operación de escritura que cambia el valor de algún elemento de la base de datos, es posible **deshacer** el efecto de esas operaciones de escritura de una transacción *T* rastreando hacia atrás el registro y restableciendo todos los elementos modificados por una operación de escritura de *T* a sus valores antiguos (*valor_antiguo*). También es posible que tengamos que **rehacer** las operaciones de una transacción si todas sus actualizaciones se grabaron en el registro pero se produjo un fallo antes de poder garantizarse que todos los *valores_nuevos* se grabaron permanentemente en la base de datos.[7] La tarea de rehacer las operaciones de la transacción *T* se consigue moviéndonos hacia delante en el registro y configurando todos los elementos modificados por una operación de escritura de *T* a sus *valores_nuevos*.

## 17.2.3 Punto de confirmación de una transacción

Una transacción *T* alcanza su **punto de confirmación** cuando todas sus operaciones que acceden a la base de datos se han ejecutado satisfactoriamente y el efecto de todas ellas se ha grabado en el registro. Más allá del punto de confirmación, se dice que la transacción se ha **confirmado**, y se asume que su efecto se ha *registrado permanentemente* en la base de datos. La transacción escribe entonces un registro de envío [commit, *T*] en el registro del sistema. Si se produce un fallo del sistema, se buscan hacia atrás en el registro del sistema todas las transacciones *T* que han escrito un registro [start_transaction, *T*] pero que no han escrito todavía su registro [commit, *T*]; es posible que haya que anular estas transacciones para deshacer su efecto en la base de datos durante el proceso de recuperación. Las transacciones que han escrito su registro *commit* en el registro del

[7] Las tareas de rehacer y deshacer se explican más en profundidad en el Capítulo 19.

sistema también deben haber registrado todas sus operaciones de escritura, para que su efecto en la base de datos pueda *rehacerse* a partir de las entradas del registro.

Observe que el fichero del registro debe guardarse en el disco. Como explicamos en el Capítulo 13, la actualización de un fichero en disco supone copiar el bloque adecuado del fichero a un búfer de la memoria principal, actualizar el búfer en la memoria principal y copiar el búfer en el disco. Es común mantener uno o más bloques del fichero del registro del sistema en los búferes de la memoria principal hasta que se llenan para después escribirlos de una sola vez en el disco, en lugar de escribir en disco cada vez que se añade una entrada al registro. Esto ahorra el coste que suponen varias escrituras en disco del mismo bloque del fichero del registro. Cuando el sistema se cae, el proceso de recuperación sólo tiene en cuenta las entradas del registro que se *escribieron en el disco*, porque el contenido de la memoria principal se puede haber perdido. Por tanto, *antes* de que una transacción alcance su punto de confirmación, cualquier porción del registro que no se haya escrito todavía en el disco, debe escribirse ahora. Este proceso se denomina **escritura forzosa** del fichero del registro antes de la confirmación de una transacción.

## 17.3 Propiedades deseables de las transacciones

Las transacciones deben poseer varias propiedades, a menudo denominadas propiedades **ACID**, que deben ser implementadas por el control de la concurrencia y los métodos de recuperación del DBMS. Las propiedades ACID son las siguientes:

- **Atomicidad.** Una transacción es una unidad atómica de procesamiento; o se ejecuta en su totalidad o no se ejecuta en absoluto.
- **Conservación de la consistencia.** Una transacción está conservando la consistencia si su ejecución completa lleva a la base de datos de un estado consistente a otro.
- **Aislamiento.** Una transacción debe aparecer como si estuviera ejecutándose de forma aislada a las demás. Es decir, la ejecución de una transacción no debe interferir con la ejecución de ninguna otra transacción simultánea.
- **Durabilidad.** Los cambios aplicados a la base de datos por una transacción confirmada deben persistir en la base de datos. Estos cambios no deben perderse por culpa de un fallo.

La propiedad de la atomicidad requiere que la transacción se ejecute hasta su finalización. El subsistema de recuperación de transacciones del DBMS tiene la responsabilidad de garantizar la atomicidad. Si una transacción falla y no se completa por alguna razón (por ejemplo, por una caída del sistema en medio de su ejecución), la técnica de recuperación debe deshacer sus efectos en la base de datos.

Se considera que la conservación de la consistencia es responsabilidad de los programadores que escriben los programas de bases de datos o del módulo DBMS que implementa las restricciones de integridad. Recuerde que un **estado de la base de datos** es una colección de todos los elementos de datos (valores) almacenados en la base de datos en un momento dado. Un **estado consistente** de la base de datos satisface las restricciones especificadas en el esquema, así como cualquier otra restricción que deba cumplirse en la base de datos. Un programa de base de datos debe escribirse de forma que garantice que, si la base de datos se encuentra en un estado consistente antes de ejecutar la transacción, estará en un estado consistente después de haberse completado la ejecución de la transacción, asumiendo que *no interfiere con ninguna otra transacción*.

El aislamiento es tarea del subsistema de control de la concurrencia del DBMS.[8] Si cada transacción no hace visibles sus actualizaciones a otras transacciones hasta que se confirma, se implementa una forma de aislamiento que soluciona el problema de la actualización temporal y elimina las anulaciones en cascada (consulte el Capítulo 19). Ha habido intentos de definir el **nivel de aislamiento** de una transacción. Una transacción

[8] En el Capítulo 18 veremos los protocolos de control de la concurrencia.

tiene un nivel 0 (cero) de aislamiento si no sobrescribe las lecturas sucias de las transacciones de nivel más alto. El aislamiento de nivel 1 (uno) no pierde actualizaciones; y el aislamiento de nivel 2 no pierde actualizaciones y no hace lecturas sucias. Por último, el aislamiento de nivel 3 (también conocido como *aislamiento verdadero*) tiene, además de las propiedades del grado 2, lecturas que se pueden repetir.

Por último, la propiedad de la durabilidad es responsabilidad del subsistema de recuperación del DBMS. En el Capítulo 19 veremos cómo los protocolos de recuperación implementan la durabilidad y la atomicidad.

# 17.4 Clasificación de las planificaciones en base a la recuperabilidad

Cuando las transacciones se están ejecutando concurrentemente en modo interpolado, el orden de ejecución de las operaciones de las distintas transacciones se conoce como **planificación**. En esta sección, primero definimos el concepto de planificación, y después veremos los tipos de planificaciones que posibilitan la recuperación cuando se produce un fallo. En la Sección 17.5 clasificamos las planificaciones en términos de interferencia de las transacciones participantes, lo que nos llevará a los conceptos de serialización y planificaciones serializables.

## 17.4.1 Planificaciones de transacciones

Una **planificación** $S$ de $n$ transacciones $T_1$, $T_2$, ..., $T_n$ es una ordenación de las operaciones de esas transacciones sujeta a la restricción de que, por cada transacción $T_i$ que participa en $S$, las operaciones de $T_i$ en $S$ deben aparecer en el mismo orden en el que tienen lugar en $T_i$. No obstante, observe que las operaciones de otras transacciones $T_j$ pueden interpolarse con las operaciones de $T_i$ en $S$. Por ahora, considere que el orden de las operaciones en $S$ responde a una *ordenación total*, aunque es teóricamente posible tratar con planificaciones cuyas operaciones forman *ordenaciones parciales* (como veremos más tarde).

En lo referente a la recuperación y el control de la concurrencia, nuestro interés se centra en las operaciones read_item y write_item de las transacciones, así como en las operaciones commit y abort. Una notación abreviada para describir una planificación utiliza los símbolos $r$, $w$, $c$ y $a$ para las operaciones read_item, write_item, commit y abort, respectivamente, y añade como subíndice el id de la transacción (número de la transacción) a cada operación de la planificación. En esta notación, el elemento $X$ de la base de datos que se lee o escribe se escribe entre paréntesis a continuación de las operaciones $r$ y $w$. Por ejemplo, la planificación de la Figura 17.3(a), que llamaremos $S_a$, puede escribirse de este modo según esta notación:

$$S_a: r_1(X); r_2(X); w_1(X); r_1(Y); w_2(X); w_1(Y);$$

De forma parecida, la planificación para la Figura 17.3(b), que llamaremos $S_b$, puede escribirse de este modo, si asumimos que la transacción $T_1$ abortó después de su operación read_item($Y$):

$$S_b: r_1(X); w_1(X); r_2(X); w_2(X); r_1(Y); a_1;$$

Dos operaciones de una planificación entran en **conflicto** si satisfacen estas tres condiciones: (1) pertenecen a transacciones diferentes; (2) acceden al mismo elemento $X$; y (3) al menos una de las operaciones es write_item($X$). Por ejemplo, en la planificación $S_a$, las operaciones $r_1(X)$ y $w_2(X)$ están en conflicto, al igual que $r_2(X)$ y $w_1(X)$, y $w_1(X)$ y $w_2(X)$. Sin embargo, las operaciones $r_1(X)$ y $r_2(X)$ no están en conflicto, puesto que se trata de dos operaciones de lectura; las operaciones $w_2(X)$ y $w_1(Y)$ no entran en conflicto porque operan sobre elementos de datos distintos, $X$ e $Y$; y las operaciones $r_1(X)$ y $w_1(X)$ tampoco entran en conflicto porque pertenecen a la misma transacción.

Se dice que una planificación $S$ de $n$ transacciones $T_1$, $T_2$, ..., $T_n$ es una **planificación completa** si se dan las siguientes condiciones:

1. Las operaciones en $S$ son exactamente las operaciones en $T_1$, $T_2$, ..., $T_n$, incluyendo una operación commit o abort como última operación de cada transacción de la planificación.
2. Para cualquier par de operaciones de la misma transacción $T_i$, su orden de aparición en $S$ es el mismo que su orden de aparición en $T_i$.
3. Para cualesquiera dos operaciones en conflicto, una de las dos debe producirse antes que la otra en la planificación.[9]

La condición 3 permite que dos *operaciones no conflictivas* ocurran en la planificación sin tener que definir cuál de ellas se produce primero, lo que nos lleva a la definición de una planificación como una **ordenación parcial** de las operaciones en las $n$ transacciones.[10] Sin embargo, en la planificación debe especificarse un orden total para cualquier par de operaciones conflictivas (condición 3) y para cualquier par de operaciones de la misma transacción (condición 2). La condición 1 simplemente dice que todas las operaciones de las transacciones deben aparecer en la planificación completa. Como cada transacción se confirma o anula, una planificación completa no contendrá ninguna transacción activa al final.

En general, es difícil encontrar planificaciones completas en un sistema de procesamiento de transacciones porque se envían transacciones nuevas continuamente al sistema. Por tanto, es útil definir el concepto de **proyección confirmada** $C(S)$ de una planificación $S$, que sólo incluye las operaciones de $S$ que pertenecen a las transacciones confirmadas (es decir, las transacciones $T_i$ cuya operación commit $c_i$ está en $S$).

### 17.4.2 Clasificación de las planificaciones en base a la recuperabilidad

Con algunas planificaciones es fácil la recuperación ante fallos, mientras que no lo es tanto con otras. Por tanto, es importante distinguir los tipos de planificaciones para los que la recuperación es posible, así como aquellas para las que la recuperación es relativamente simple. Estas clasificaciones no proporcionan realmente el algoritmo de recuperación; sólo intentan clasificar teóricamente los distintos tipos de planificaciones.

En primer lugar, nos gustaría asegurar que, una vez confirmada una transacción $T$, *nunca* debe ser necesario anularla. Las planificaciones que teóricamente cumplen este criterio, se denominan planificaciones recuperables, y aquellas que no lo cumplen, **irrecuperables** (y, por tanto, no deben permitirse). Una planificación $S$ es recuperable si ninguna transacción $T$ en $S$ se confirma hasta que todas las transacciones $T'$ que han escrito un elemento que $T$ lee se han confirmado. Una transacción $T$ **lee** de la transacción $T'$ en una planificación $S$ si algún elemento $X$ es escrito en primer lugar por $T'$ y más tarde leído por $T$. Además, $T'$ no debe cancelarse antes de que $T$ lea el elemento $X$, y no puede haber transacciones que escriban $X$ después de que $T'$ lo escriba y antes de que $T$ lo lea (a menos que esas transacciones, si las hay, se hayan cancelado antes de que $T$ lea $X$).

Como veremos, las planificaciones recuperables requieren un proceso de recuperación más complejo, pero si se guarda la información suficiente (en el registro), podemos idear un algoritmo de recuperación. Las planificaciones (parciales) $S_a$ y $S_b$ de la sección anterior son recuperables, ya que satisfacen la definición anterior. Considere la siguiente planificación, $S_a'$, que es idéntica a $S_a$ excepto que se han añadido dos operaciones commit a $S_a$:

$$S_a'\text{: } r_1(X);\ r_2(X);\ w_1(X);\ r_1(Y);\ w_2(X);\ c_2;\ w_1(Y);\ c_1;$$

[9] En teoría, no es necesario determinar un orden entre pares de operaciones *no conflictivas*.

[10] En la práctica, la mayoría de las planificaciones tienen un orden total de las operaciones. Si se emplea el procesamiento paralelo, es teóricamente posible tener planificaciones con operaciones no conflictivas parcialmente ordenadas.

$S_a'$ es recuperable, aunque padece el problema de la pérdida de actualización. Sin embargo, considere estas dos planificaciones (parciales), $S_c$ y $S_d$:

$S_c$: $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; $c_2$; $a_1$;

$S_d$: $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$; $c_1$; $c_2$;

$S_e$: $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$; $a_1$; $a_2$;

$S_c$ no es recuperable porque $T_2$ lee el elemento $X$ de $T_1$, y después se confirma $T_2$ antes que $T_1$. Si $T_1$ se cancela después de la operación $c_2$ en $S_c$, entonces el valor de $X$ que $T_2$ lee ya no es válido y $T_2$ debe cancelarse *después* de ser confirmada, lo que lleva a una planificación que no es recuperable. Para que la planificación sea recuperable, la operación $c_2$ en $S_c$ debe posponerse hasta después de confirmarse $T_1$, como se muestra en $S_d$; si $T_1$ se cancela en lugar de confirmarse, entonces $T_2$ también debe cancelarse como se muestra en $S_e$, porque el valor de $X$ que leyó ya no es válido.

En una planificación recuperable, ninguna transacción confirmada debe ser anulada. Sin embargo, es posible que se produzca un fenómeno conocido como **anulación en cascada**, en el que una transacción no *confirmada* tiene que ser anulada porque lee un elemento de una transacción que falló. Es lo que se ilustra en la planificación $S_e$, donde la transacción $T_2$ tiene que ser anulada porque lee el elemento $X$ de $T_1$, y $T_1$ se canceló entonces.

Como la anulación en cascada puede consumir mucho tiempo (puesto que puede darse sobre muchas transacciones [consulte el Capítulo 19]), es importante distinguir las transacciones en las que está garantizado que este fenómeno no ocurra. Una transacción es ***cascadeless***, o se dice que **evita la anulación en cascada**, si cada transacción de la planificación sólo lee elementos escritos por las transacciones confirmadas. En este caso, no se descartarán todos los elementos leídos, para que no se produzca una anulación en cascada. Para satisfacer este criterio, el comando $r_2(X)$ de las planificaciones $S_d$ y $S_e$ debe posponerse hasta después de haberse confirmado (o cancelado) $T_1$; en consecuencia, $T_2$ se retrasa pero se garantiza la ausencia de la anulación en cascada si $T_1$ se cancela.

Por último, hay un tercer tipo, más restrictivo, de planificación, denominado **planificación estricta**, en la que las transacciones *no pueden leer ni escribir* un elemento $X$ hasta haberse confirmado (o cancelado) la última transacción que escribió $X$. Las planificaciones estrictas simplifican el proceso de recuperación. En una planificación estricta, el proceso de deshacer una operación write_item($X$) de una transacción cancelada es simplemente recuperar la **imagen "antes"** (*valor_antiguo* o BFIM) del elemento de datos $X$. Este sencillo procedimiento siempre funciona correctamente para las planificaciones estrictas, pero puede que no funcione para las planificaciones recuperables o *cascadeless*. Por ejemplo, considere la planificación $S_f$:

$S_f$: $w_1(X, 5)$; $w_2(X, 8)$; $a_1$;

Suponga que el valor de $X$ era originalmente 9, que es la imagen "antes" almacenada en el registro del sistema junto con la operación $w_1(X, 5)$. Si $T_1$ se cancela, como en $S_f$, el procedimiento de recuperación que almacena la imagen "antes" de una operación de escritura cancelada restaurará el valor de $X$ a 9, aunque la transacción $T_2$ ya lo cambió por 8, lo que nos lleva a unos resultados potencialmente incorrectos. Aunque la planificación $S_f$ evita la anulación en cascada, no es una planificación estricta, puesto que permite a $T_2$ escribir el elemento $X$ aunque la transacción $T_1$ que escribió $X$ en último lugar todavía no se haya confirmado (o cancelado). Una planificación estricta no tiene este problema.

Hemos clasificado las planificaciones de acuerdo con los siguientes términos: (1) recuperabilidad, (2) la posibilidad de evitar la anulación en cascada, y (3) la rigurosidad. Después vimos que estas propiedades de las planificaciones son condiciones sucesivamente más estrictas. Por tanto, la condición (2) implica la condición (1), y la condición (3) implica las condiciones (2) y (1). Así, todas las planificaciones estrictas evitan la anulación en cascada, y todas las planificaciones que evitan la anulación en cascada son recuperables.

# 17.5 Clasificación de las planificaciones basándose en la serialización

En la sección anterior clasificamos las planificaciones en base a sus propiedades de recuperabilidad. Ahora, clasificamos los tipos de planificaciones que se consideran correctos cuando varias transacciones se están ejecutando simultáneamente. Suponga que dos usuarios (agentes de viajes reservando en una aerolínea) remiten al DBMS, aproximadamente al mismo tiempo, las transacciones $T_1$ y $T_2$ de la Figura 17.2. Si no está permitida la interpolación de operaciones, sólo hay dos posibles resultados:

1. Ejecutar todas las operaciones de la transacción $T_1$ (en secuencia) seguidas por todas las operaciones de la transacción $T_2$ (en secuencia).
2. Ejecutar todas las operaciones de la transacción $T_2$ (en secuencia) seguidas por todas las operaciones de la transacción $T_1$ (en secuencia).

Estas alternativas se muestran en la Figura 17.5(a) y (b), respectivamente. Si está permitida la interpolación de operaciones, el sistema puede ejecutar las operaciones individuales de las transacciones obedeciendo a diversas ordenaciones. En la Figura 17.5(c) se muestran dos planificaciones posibles. El concepto de **serialización de planificaciones** se utiliza para identificar las planificaciones que son correctas cuando las ejecuciones de transacción tienen interpoladas sus operaciones en las planificaciones. Esta sección define la serialización y explica cómo puede utilizarse en la práctica.

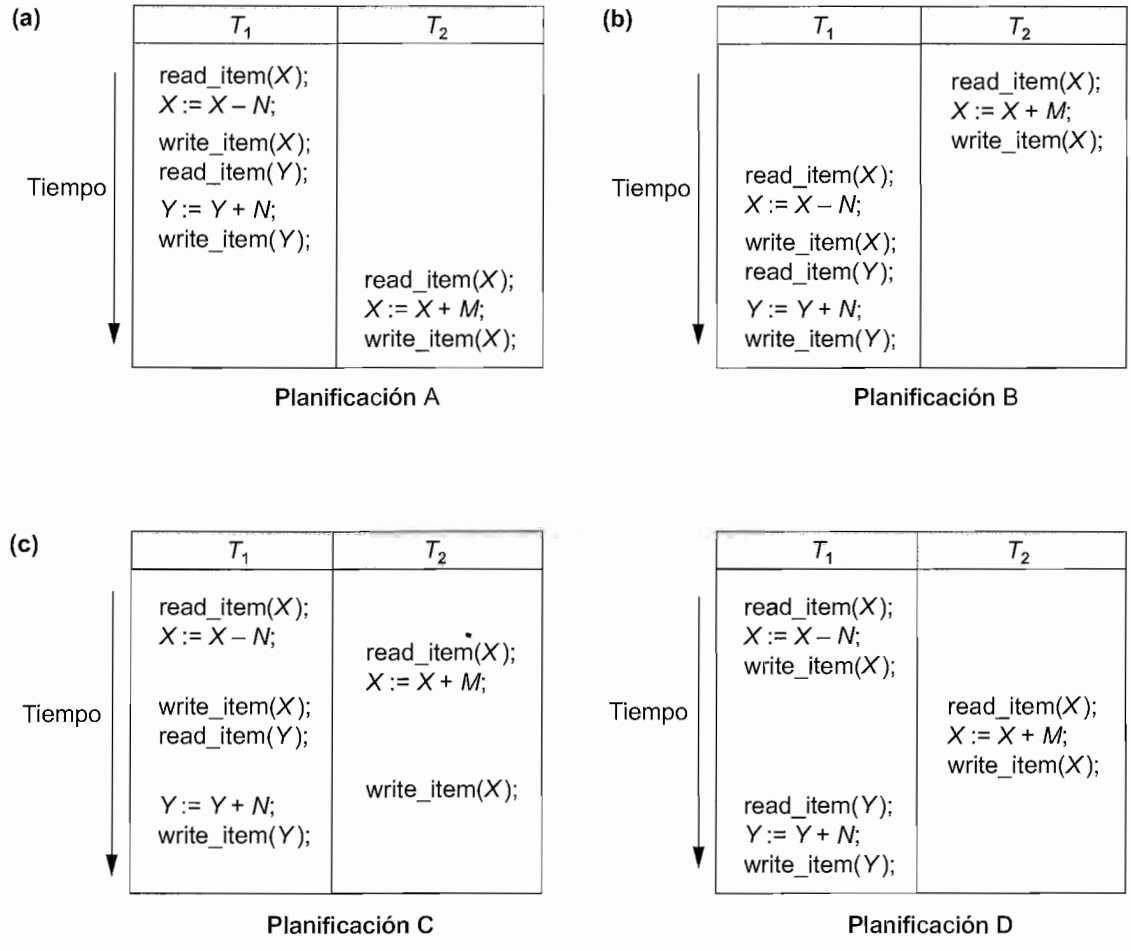
## 17.5.1 Planificaciones en serie, no serie y serializables por conflicto

Las planificaciones A y B de la Figura 17.5(a) y (b) están en *serie* porque las operaciones de cada transacción se ejecutan consecutivamente, sin que se interpolen las operaciones de otra transacción. En una planificación en serie, las transacciones se ejecutan enteras y en serie: $T_1$ y después $T_2$ en la Figura 17.5(a), y $T_2$ y después $T_1$ en la Figura 17.5(b). Las planificaciones C y D de la Figura 17.5(c) se denominan *no serie* porque cada secuencia interpola las operaciones de las dos transacciones.

Formalmente, una planificación $S$ es **serie** si, por cada transacción $T$ que participa en la planificación, todas las operaciones de $T$ se ejecutan consecutivamente en la planificación; en caso contrario, se dice que la planificación **no es serie**. Por consiguiente, en una planificación en serie, sólo hay una transacción activa al mismo tiempo (la confirmación, o la cancelación, de la transacción activa inicia la ejecución de la siguiente transacción). En una planificación en serie no se produce la interpolación. Una suposición razonable que podemos hacer, si consideramos que las transacciones son *independientes*, es considerar que cada planificación en serie es correcta. Podemos asumir esto porque se asume que una transacción es correcta si se ejecuta sola (según la propiedad de la conservación de la consistencia de la Sección 17.3). Por tanto, no importa qué transacción se ejecuta primero. Siempre y cuando cada transacción se ejecute desde el principio hasta el final sin que interfieran las operaciones de otras transacciones, conseguiremos un resultado correcto en la base de datos. El problema con las planificaciones serie es que limitan la concurrencia o la interpolación de operaciones. En una planificación en serie, si una transacción está esperando a que una operación de E/S se complete, no podemos hacer que el procesador de la CPU cambie a otra transacción, derrochándose de este modo un valioso tiempo de procesamiento de CPU. Además, si alguna transacción $T$ es bastante larga, las otras transacciones deben esperar a que $T$ complete todas sus operaciones antes de comenzar. Por tanto, y en la práctica, las planificaciones serie están generalmente consideradas inaceptables.

Para ilustrar nuestra explicación, considere las planificaciones de la Figura 17.5, y asuma que los valores iniciales de los elementos de la base de datos son $X$=90 e $Y$=90, y que $N$=3 y $M$=2. Después de ejecutar las transacciones $T_1$ y $T_2$, sería de esperar que los valores de la base de datos fueran $X$=89 e $Y$=93, de acuerdo con el significado de las transacciones. Sin duda, la ejecución de cualquiera de las planificaciones A o B ofrece los

**Figura 17.5.** Ejemplos de planificaciones serie y no serie en las que se ven implicadas las transacciones $T_1$ y $T_2$. (a) Planificación en serie A: $T_1$ seguida por $T_2$. (b) Planificación en serie B: $T_2$ seguida por $T_1$. (c) Dos planificaciones no serie C y D con interpolación de operaciones.

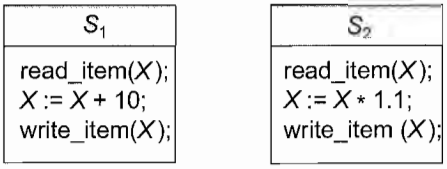



resultados correctos. Ahora, considere las planificaciones no serie C y D. La planificación C (que es la misma que la de la Figura 17.3[a]) ofrece los resultados $X$=92 e $Y$=93, siendo el valor de $X$ erróneo, mientras que la planificación D ofrece los resultados correctos.

La planificación C ofrece un resultado incorrecto debido al problema de la pérdida de actualización explicado en la Sección 17.1.3; la transacción $T_2$ lee el valor de $X$ antes de que sea cambiado por la transacción $T_1$, de modo que en la base de datos sólo se refleja el efecto de $T_2$ sobre $X$. El efecto de $T_1$ sobre $X$ se *pierde*, al ser sobrescrito por $T_2$, lo que nos lleva a un resultado incorrecto para el elemento $X$. Sin embargo, algunas planificaciones no serie ofrecen el resultado correcto, como la planificación $D$. Nos gustaría determinar cuáles de las planificaciones no serie *siempre* ofrecen un resultado correcto y cuáles pueden proporcionar resultados erróneos. El concepto utilizado para clasificar las planificaciones de este modo es la serialización de una planificación.

Una planificación $S$ de $n$ transacciones es **serializable** si es *equivalente a alguna planificación en serie* de las mismas $n$ transacciones. Definiremos brevemente el concepto de equivalencia de planificaciones. A partir de $n$ transacciones, podemos obtener $n!$ planificaciones en serie posibles y muchas más planificaciones no serie. Podemos formar dos grupos disjuntos con las planificaciones no serie: aquellas que son equivalentes a una (o

**Figura 17.6.** Dos planificaciones equivalentes en cuanto a resultado para el valor inicial de $X = 100$, pero que en general no son de resultado equivalente.




más) de las planificaciones en serie y, por tanto, serializables; y aquellas que no son equivalentes a ninguna planificación en serie y, por tanto, no son serializables.

Decir que una planificación no serie $S$ es serializable es equivalente a decir que es correcta, porque es equivalente a una planificación en serie, que se considera correcta. La cuestión es: ¿Cuándo se considera que dos planificaciones son *equivalentes*? Hay varias formas de definir la equivalencia de planificación. La definición más sencilla, pero menos satisfactoria, implica comparar los efectos de las planificaciones sobre la base de datos. Dos planificaciones son de **resultado equivalente** si producen el mismo estado final de la base de datos. Sin embargo, dos planificaciones diferentes pueden producir accidentalmente el mismo estado final. Por ejemplo, en la Figura 17.6, las planificaciones $S_1$ y $S_2$ producirán el mismo estado final de la base de datos si se ejecutan en una base de datos con un valor inicial de $X$=100; pero para otros valores iniciales de $X$, las planificaciones no son de resultado equivalente. Además, estas planificaciones ejecutan transacciones diferentes, por lo que no deben considerarse definitivamente equivalentes. Por tanto, la equivalencia de resultado por sí sola no puede utilizarse para definir la equivalencia de planificaciones. La metodología más segura y general para definir la equivalencia de planificaciones es no hacer ninguna asunción sobre los tipos de operaciones incluidos en las transacciones. Para que dos planificaciones sean equivalentes, las operaciones aplicadas a cada elemento de datos afectado por las planificaciones deben aplicarse a ese elemento en ambas planificaciones *en el mismo orden*. Generalmente se utilizan dos definiciones de equivalencia de planificaciones: *equivalencia por conflicto* y *equivalencia por vista*. A continuación explicamos la equivalencia por conflicto, que es la definición más utilizada.

Dos planificaciones son **equivalentes por conflicto** si el orden de cualquier par de *operaciones en conflicto* es el mismo en las dos planificaciones. Como recordará de la Sección 17.4.1, dos operaciones de una planificación entran en conflicto si pertenecen a transacciones diferentes, acceden al mismo elemento de la base de datos, y al menos una de las dos operaciones es una operación write_item. Si dos operaciones conflictivas se aplican con un orden diferente en dos planificaciones, el efecto puede ser diferente en la base de datos o en otras transacciones de la planificación y, por tanto, las planificaciones no son equivalentes por conflicto. Por ejemplo, si en la planificación $S_1$ se produce una operación de lectura y escritura en el orden $r_1(X)$, $w_2(X)$, y en el orden inverso, $w_2(X)$, $r_1(X)$, en la planificación $S_2$, el valor leído por $r_1(X)$ puede ser diferente en las dos planificaciones. De forma parecida, si dos operaciones de escritura tienen lugar en el orden $w_1(X)$, $w_2(X)$ en $S_1$, y en el orden inverso, $w_2(X)$, $w_1(X)$, en $S_2$, es probable que la siguiente operación $r(X)$ en las dos planificaciones lean valores diferentes; o si son las últimas operaciones que escriben el elemento $X$ en las planificaciones, el valor final del elemento $X$ en la base de datos será diferente.

Utilizando el concepto de equivalencia por conflicto, decimos que una planificación $S$ es **serializable por conflicto**[11] si es equivalente (por conflicto) con alguna planificación en serie $S'$. En tal caso, podemos reordenar las operaciones *no conflictivas* de $S$ para formar la planificación serie equivalente $S'$. De acuerdo con esta definición, la planificación $D$ de la Figura 17.5(c) es equivalente a la planificación en serie $A$ de la Figura 17.5(a). En las dos planificaciones, la operación read_item($X$) de $T_2$ lee el valor de $X$ escrito por $T_1$, mientras

[11] Utilizaremos *serializable* para dar a entender serializable por conflicto. Otra definición de serializable que se utiliza en la práctica (consulte la Sección 17.6) es tener lecturas repetitivas, no lecturas sucias, y sin registros fantasma (consulte la Sección 18.7.1 si desea una explicación de fantasma).

que las otras operaciones read_item leen los valores correspondientes al estado inicial de la base de datos. Además, $T_1$ es la última transacción que escribe $Y$, y $T_2$ es la última transacción que escribe $X$ en ambas planificaciones. Como A es una planificación en serie y la planificación D es equivalente a A, D es una planificación serializable. Las operaciones $r_1(Y)$ y $w_1(Y)$ de la planificación D no entran en conflicto con las operaciones $r_2(X)$ y $w_2(X)$, puesto que acceden a elementos de datos diferentes. Por consiguiente, podemos mover $r_1(Y)$, $w_1(Y)$ antes de $r_2(X)$, $w_2(X)$, lo que nos lleva a la planificación serie equivalente de $T_1$, $T_2$.

La planificación C de la Figura 17.5(c) no es equivalente a ninguna de las dos posibles planificaciones en serie, A y B, y, por tanto, no es *serializable*. El intento de reordenar las operaciones de la planificación C para encontrar una planificación en serie equivalente falla porque $r_2(X)$ y $w_1(X)$ entran en conflicto, lo que significa que no podemos mover $r_2(X)$ hacia abajo para obtener la planificación en serie equivalente de $T_1$, $T_2$. De forma parecida, como $w_1(X)$ y $w_2(X)$ entran en conflicto, no podemos mover $w_1(X)$ hacia abajo para obtener la planificación serie equivalente de $T_2$, $T_1$.

En la Sección 17.5.4 explicamos otra definición más compleja de equivalencia (denominada equivalencia por vista, que conduce al concepto de serialización por vista).

### 17.5.2 Comprobación de la serialización por conflicto de una planificación

Hay un algoritmo sencillo para determinar la serialización por conflicto de una planificación. La mayoría de los métodos de control de la concurrencia *no* comprueban realmente la serialización. Más bien, se desarrollan protocolos, o reglas, que garantizan que una planificación será serializable. Aquí explicamos el algoritmo de comprobación de la serialización por conflicto de las planificaciones para entender mejor esos protocolos de control de la concurrencia, que explicamos en el Capítulo 18.

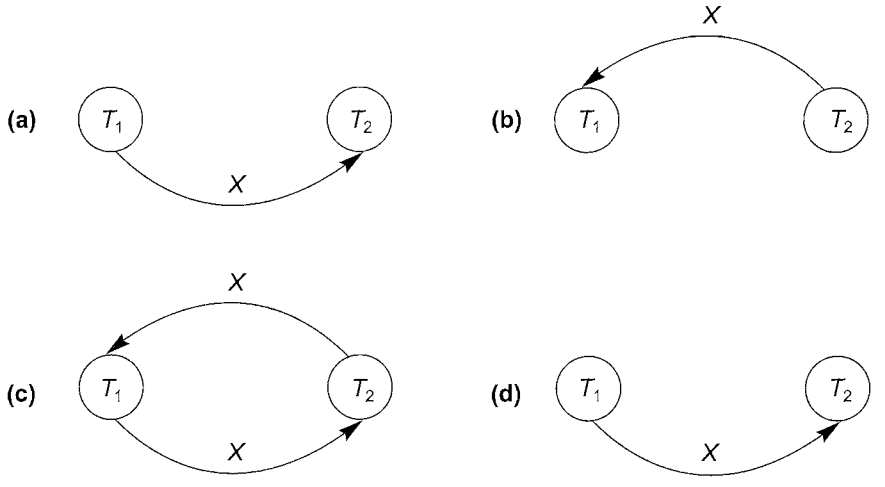
El Algoritmo 17.1 puede utilizarse para comprobar la serialización por conflicto de una planificación. El algoritmo sólo examina las operaciones read_item y write_item de una planificación para construir un **gráfico de precedencia** (o **gráfico de serialización**), que es un **gráfico tendente** $G = (N, E)$ consistente en un conjunto de nodos $N = \{T_1, T_2, ..., T_n\}$ y un conjunto de bordes o arcos tendentes $E = \{e_1, e_2, ..., e_m\}$. En el gráfico hay un nodo por cada transacción $T_i$ de la planificación. Cada arco $e_i$ del gráfico tiene la forma $(T_j \rightarrow T_k)$, 1 ? $j$ = ?, 1 ? $k = n$, donde $T_j$ es el **nodo inicial** de $e_i$ y $T_k$ es el **nodo final** de $e_i$. Se crea un arco de estas características si una de las operaciones de $T_j$ aparece en la planificación antes que alguna *operación en conflicto* de $T_k$.

**Algoritmo 17.1. Comprobación de la serialización por conflicto de una planificación *S*.**

1. Por cada transacción $T_i$ participante en la planificación $S$, crear un nodo etiquetado como $T_i$ en el gráfico de precedencia.
2. Por cada caso de $S$ donde $T_j$ ejecute una operación read_item($X$) después de que $T_i$ ejecute write_item($X$), crear un arco $(T_i \rightarrow T_j)$ en el gráfico de precedencia.
3. Por cada caso de S donde $T_j$ ejecute una operación write_item($X$) después de que $T_i$ ejecute read_item($X$), crear un arco $(T_i \rightarrow T_j)$ en el gráfico de precedencia.
4. Por cada caso de S donde $T_j$ ejecute una operación write_item($X$) después de que $T_i$ ejecute write_item($X$), crear un arco $(T_i \rightarrow T_j)$ en el gráfico de precedencia.
5. La planificación $S$ es serializable si, y sólo si, el gráfico de precedencia no tiene ciclos.

El gráfico de precedencia se construye como se describe en el Algoritmo 17.1. Si hay un ciclo en el gráfico de precedencia, la planificación $S$ no es serializable (por conflicto); si no hay ningún ciclo, $S$ es serializable. Un **ciclo** en un gráfico tendente es una **secuencia de arcos** $C = ((T_j \rightarrow T_k), (T_k \rightarrow T_p), ..., (T_i \rightarrow T_j))$ con la propiedad de que el nodo inicial de cada arco (excepto en el primer arco) es el mismo que el nodo final del

**Figura 17.7.** Construcción de los gráficos de precedencia para las planificaciones A a D de la Figura 17.5 para probar la serialización por conflicto. (a) Gráfico de precedencia para la planificación en serie A. (b) Gráfico de precedencia para la planificación en serie B. (c) Gráfico de precedencia para la planificación C (no serializable). (d) Gráfico de precedencia para la planificación D (serializable, equivalente a la planificación A).




arco anterior, y el nodo inicial del primer arco es el mismo que el nodo final del último arco (la secuencia empieza y termina en el mismo nodo).

En el gráfico de precedencia, un arco de $T_i$ a $T_j$ significa que la transacción $T_i$ debe ir antes que la transacción $T_j$ en cualquier planificación en serie que sea equivalente a $S$, porque aparecen dos operaciones en conflicto en ese orden en la planificación. Si no hay ningún ciclo en el gráfico de precedencia, podemos crear una **planificación en serie equivalente** $S'$ que sea equivalente a $S$, ordenando de este modo las transacciones que participan en $S$: siempre que exista un arco en el gráfico de precedencia de $T_i$ a $T_j$, $T_i$ debe aparecer antes que $T_j$ en la planificación en serie equivalente $S'$.[12] Los arcos ($T_i \rightarrow T_j$) en un gráfico de precedencia pueden etiquetarse opcionalmente con el(los) nombre(s) del(de los) elemento(s) que llevan a crear el arco. La Figura 17.7 muestra dichas etiquetas en los arcos.

En general, varias planificaciones en serie pueden ser equivalentes a $S$ si el gráfico de precedencia para $S$ no tiene ciclos. Sin embargo, si el gráfico de precedencia tiene un ciclo, es fácil mostrar que no podemos crear ninguna planificación en serie equivalente, por lo que $S$ no es serializable. Los gráficos de precedencia creados para las planificaciones A a D, respectivamente, de la Figura 17.5 aparecen en la Figura 17.7(a) a (d). El gráfico para la planificación C tiene un ciclo, por lo que no es serializable. El gráfico para la planificación D no tiene ciclos, por lo que es serializable, y la planificación serie equivalente es $T_1$ seguida de $T_2$. Los gráficos para las planificaciones A y B no tienen ciclo, como era de esperar, porque las planificaciones son serie y, por tanto, serializables.

La Figura 17.8 muestra otro ejemplo, en el que participan tres transacciones. La Figura 17.8(a) muestra las operaciones read_item y write_item de cada transacción. En la Figura 17.8(b) y (c) se muestran, respectivamente, las dos planificaciones, $E$ y $F$, para esas transacciones, mientras que en las partes (d) y (e) se muestran los gráficos de precedencia para dichas planificaciones. La planificación $E$ no es serializable porque el gráfico de precedencia correspondiente tiene ciclos. La planificación $F$ es serializable; su planificación en serie

[12] Este proceso de ordenación de los nodos de un gráfico se conoce como ordenación topológica.

**Figura 17.8.** Otro ejemplo de comprobación de la serialización. (a) Operaciones de lectura y escritura de las tres transacciones, $T_1$, $T_2$ y $T_3$. (b) Planificación *E*. (c) Planificación *F*. (d) Gráfica de precedencia para la planificación *E*.

(a)

| Transacción $T_1$ |
|---|
| read_item($X$); |
| write_item($X$); |
| read_item($Y$); |
| write_item($Y$); |

| Transacción $T_2$ |
|---|
| read_item($Z$); |
| read_item($Y$); |
| write_item($Y$); |
| read_item($X$); |
| write_item($X$); |

| Transacción $T_3$ |
|---|
| read_item($Y$); |
| read_item($Z$); |
| write_item($Y$); |
| write_item($Z$); |

(b)

Tiempo

| Transacción $T_1$ | Transacción $T_2$ | Transacción $T_3$ |
|---|---|---|
| | read_item($Z$); | |
| | read_item($Y$); | |
| | write_item($Y$); | |
| | | read_item($Y$); |
| | | read_item($Z$); |
| read_item($X$); | | |
| write_item($X$); | | |
| | | write_item($Y$); |
| | | write_item($Z$); |
| | read_item($X$); | |
| read_item($Y$); | | |
| write_item($Y$); | | |
| | write_item($X$); | |

Planificación E

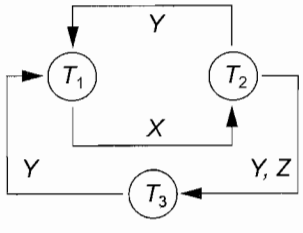
(c)

Tiempo

| Transacción $T_1$ | Transacción $T_2$ | Transacción $T_3$ |
|---|---|---|
| | | read_item($Y$); |
| | | read_item($Z$); |
| read_item($X$); | | |
| write_item($X$); | | |
| | | write_item($Y$); |
| | | write_item($Z$); |
| | read_item($Z$); | |
| read_item($Y$); | | |
| write_item($Y$); | | |
| | read_item($Y$); | |
| | write_item($Y$); | |
| | read_item($X$); | |
| | write_item($X$); | |

Planificación F

(d)




Planificaciones en serie equivalentes

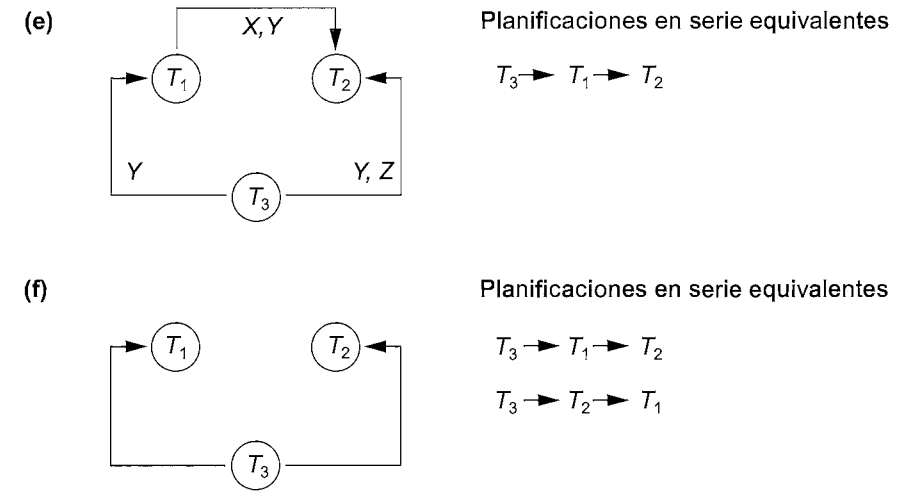
Ninguna

Razón

*Ciclo* $X(T_1 \rightarrow T_2), Y(T_2 \rightarrow T_1)$

*Ciclo* $X(T_1 \rightarrow T_2), YZ(T_2 \rightarrow T_3), Y(T_3 \rightarrow T_1)$

**Figura 17.8.** Otro ejemplo de comprobación de la serialización. (e) Gráfica de precedencia para la planificación *F*. (f) Gráfica de precedencia con dos planificaciones serie equivalentes.




equivalente se muestra en la Figura 17.8(e). Aunque para *F* sólo existe una planificación en serie equivalente, en general puede haber más de una planificación en serie equivalente para una planificación serializable. La Figura 17.8(f) muestra un gráfico de precedencia que representa una planificación que tiene dos planificaciones en serie equivalentes.

## 17.5.3 Usos de la serialización

Como explicamos anteriormente, decir que una planificación *S* es serializable (por conflicto) (es decir, *S* es equivalente [por conflico] a una planificación en serie), es como decir que *S* es correcta. Sin embargo, ser *serializable* es distinto a estar *en serie*. Una planificación en serie representa un procesamiento ineficaz, porque no está permitida la interpolación de las operaciones de transacciones diferentes. Esto puede llevar a una baja utilización de la CPU mientras una transacción espera por una E/S de disco, o porque se está esperando a que otra transacción termine; esto ralentiza considerablemente el procesamiento. Una planificación serializable proporciona los beneficios de la ejecución concurrente sin ninguna corrección. En la práctica, es muy difícil probar la serialización de una planificación. Por regla general, la interpolación de operaciones de las transacciones concurrentes (que normalmente las ejecuta el sistema operativo como procesos) es determinada por el planificador del sistema operativo, que asigna recursos a todos los procesos. Factores como la carga del sistema, el tiempo de envío de una transacción y las prioridades de los procesos contribuyen a la ordenación de las operaciones en una planificación. Por tanto, es difícil determinar de antemano cómo se interpolarán las operaciones de una planificación para garantizar la serialización.

Si las transacciones se ejecutan a voluntad y, después, se comprueba la serialización de la planificación resultante, debemos cancelar el efecto de la planificación si resulta no ser serializable. Esto es un serio problema, por lo que esta metodología no es práctica. Por tanto, la metodología que se toma en la mayoría de los sistemas prácticos es determinar los métodos que garantizan la serialización, sin tener que probar las propias planificaciones. La metodología tomada en la mayoría de los DBMSs comerciales es diseñar **protocolos** (conjuntos de reglas) que (seguidos por *cada* transacción individual o implementados por un subsistema de control de la concurrencia de un DBMS) garantizarán la serialización de *todas las planificaciones en las que las transacciones participan.*

Aquí aparece otro problema: cuando se envían transacciones continuamente al sistema, es difícil determinar cuándo empieza una planificación y cuándo termina. La teoría de la serialización puede adaptarse para tratar con este problema considerando únicamente la proyección confirmada de una planificación $S$. Como recordará de la Sección 17.4.1, la *proyección confirmada* $C(S)$ de una planificación $S$ sólo incluye las operaciones de $S$ que pertenecen a las transacciones confirmadas. Podemos definir teóricamente una planificación $S$ como serializable si su proyección confirmada $C(S)$ es equivalente a alguna planificación en serie, puesto que el DBMS sólo garantiza las transacciones confirmadas.

En el Capítulo 18 veremos diferentes protocolos de control de la concurrencia que garantizan la serialización. La técnica más común, denominada *bloqueo de dos fases*, está basada en el bloqueo de los elementos de datos para evitar que las transacciones concurrentes interfieran entre sí, y la implementación de una condición adicional que garantice la serialización. Es la técnica que se utiliza en la mayoría de los DBMSs comerciales. También se han propuesto otros protocolos;[13] *ordenación por marca de tiempo*, con la que a cada transacción se le asigna una marca de tiempo única y el protocolo garantiza que cualquier operación conflictiva se ejecuta en el orden de las marcas de tiempo de las transacciones; *protocolos multiversión*, que están basados en el mantenimiento de varias versiones de elementos de datos; y *protocolos optimistas* (también denominados *certificación* o *validación*), que comprueban las posibles violaciones de la serialización una vez terminadas las transacciones, pero antes de que esté permitida su confirmación.

### 17.5.4 Equivalencia por vista y serialización por vista

En la Sección 17.5.1 definimos los conceptos de equivalencia por conflicto de las planificaciones y la serialización por conflicto. Otra definición menos restrictiva de equivalencia de las planificaciones se conoce como *equivalencia por vista*. Esto nos lleva a otra definición de serialización denominada *serialización por vista*. Se dice que dos planificaciones $S$ y $S'$ son **equivalentes en vista** si se dan estas tres condiciones:

1. El mismo conjunto de transacciones participa en $S$ y $S'$, y $S$ y $S'$ incluyen las mismas operaciones de esas transacciones.
2. Para cualquier operación $r_i(X)$ de $T_i$ en $S$, si el valor de $X$ leído por la operación ha sido escrito por una operación $w_j(X)$ de $T_j$ (o si es el valor original de $X$ antes de iniciarse la planificación), la misma condición debe mantenerse para el valor de $X$ leído por la operación $r_i(X)$ de $T_i$ en $S'$.
3. Si la operación $w_k(Y)$ de $T_k$ es la última operación en escribir el elemento $Y$ en $S$, entonces $w_k(Y)$ de $T_k$ también debe ser la última operación para escribir el elemento $Y$ en $S'$.

La idea tras la equivalencia por vista es que, siempre y cuando cada operación de lectura de una transacción lea el resultado de la misma operación de escritura en las dos transacciones, las operaciones de escritura de cada transacción deben producir los mismos resultados. Se dice, por tanto, que las operaciones de lectura *ven la misma vista* en las dos planificaciones. La condición 3 garantiza que la operación de escritura final en cada elemento de datos es la misma en las dos planificaciones, por lo que el estado de la base de datos debe ser el mismo al final de ambas planificaciones. Una planificación $S$ se dice que es **serializable por vista** si es la vista equivalente a una planificación en serie.

Las definiciones de serialización por conflicto y serialización por vista son similares si se cumple una condición conocida como **suposición de escritura restringida** en todas las transacciones de la planificación. Esta condición establece que cualquier operación de escritura $w_i(X)$ en $T_i$ va precedida por una operación $r_i(X)$ en $T_i$ y que el valor escrito por $w_i(X)$ en $T_i$ sólo depende del valor de $X$ leído por $r_i(X)$. Esto supone que el cálculo del valor nuevo de $X$ es una función $f(X)$ basada en el valor antiguo de $X$ leído de la base de datos. No obstante, la definición de serialización por vista es menos restrictiva que la de serialización por conflicto bajo

[13] Estos otros protocolos no se han utilizado mucho en la práctica hasta ahora; la mayoría de los sistemas utilizan alguna variante del protocolo de bloqueo de dos fases.

la **suposición de escritura restringida**, donde el valor escrito por una operación $w_i(X)$ en $T_i$ puede ser independiente de su valor antiguo de la base de datos. Es lo que se conoce como **escritura ciega**, y se ilustra con la siguiente planificación $S_g$ de tres transacciones, $T_1$: $r_1(X)$; $w_1(X)$; $T_2$: $w_2(X)$; y $T_3$: $w_3(X)$:

$$S_g: r_1(X); w_2(X); w_1(X); w_3(X); c_1; c_2; c_3;$$

En $S_g$ las operaciones $w_2(X)$ y $w_3(X)$ son escrituras ciegas, porque $T_2$ y $T_3$ no leen el valor de $X$. La planificación $S_g$ es serializable por vista, puesto que es el equivalente por vista de la planificación en serie $T_1$, $T_2$, $T_3$. Sin embargo, $S_g$ no es serializable por conflicto, puesto que no es equivalente por conflicto a ninguna planificación en serie. Se ha mostrado que cualquier planificación serializable por conflicto también es serializable por vista, pero no a la inversa, como mostraba el ejercicio anterior. Hay un algoritmo para probar si una planificación $S$ es o no serializable por vista. Sin embargo, el problema de probar la serialización por vista es difícil, lo que significa que encontrar un algoritmo de tiempo polinomial eficaz para este problema es altamente improbable.

### 17.5.5 Otros tipos de equivalencia de planificaciones

La serialización de las planificaciones es a veces considerada demasiado restrictiva como condición para garantizar la exactitud de las ejecuciones concurrentes. Algunas aplicaciones pueden producir planificaciones que son correctas satisfaciendo las condiciones menos severas que la serialización por conflicto o la serialización por vista. Un ejemplo es el tipo de transacciones conocidas como **transacciones de débito-crédito** (por ejemplo, las que aplican depósitos y retiradas de fondos a un elemento de datos cuyo valor es el saldo actual de una cuenta bancaria). La semántica de las operaciones de débito-crédito es que actualizan el valor de un elemento de datos $X$ sustrayendo o añadiendo al valor de ese elemento de datos. Como las operaciones de adición y sustracción son conmutativas (es decir, pueden aplicarse en cualquier orden), es posible generar planificaciones correctas que no sean serializables. Por ejemplo, considere las siguientes transacciones, cada una de las cuales puede utilizarse para transferir una cantidad de dinero entre dos cuentas bancarias:

$$T_1: r_1(X); X := X - 10; w_1(X); r_1(Y); Y := Y + 10; w_1(Y);$$
$$T_2: r_2(Y); Y := Y - 20; w_2(Y); r_2(X); X := X + 20; w_2(X);$$

Considere la siguiente planificación no serializable, $S_h$, para las dos transacciones:

$$S_h: r_1(X); w_1(X); r_2(Y); w_2(Y); r_1(Y); w_1(Y); r_2(X); w_2(X);$$

Con el conocimiento, o **semántica**, adicional de que las operaciones entre cada $r_i(I)$ y $w_i(I)$ son conmutativas, sabemos que el orden de ejecución de las secuencias consistentes en (lectura, actualización, escritura) no es importante siempre y cuando cada secuencia (lectura, actualización, escritura) por una transacción particular $T_i$ en un elemento $I$ concreto no sea interrumpida por operaciones en conflicto. Por tanto, la planificación $S_h$ es considerada correcta aunque no sea serializable. Los investigadores han estado trabajando para extender la teoría del control de la concurrencia al tratamiento de casos en los que se considera que la serialización es demasiado restrictiva como condición para medir la exactitud de las planificaciones.

## 17.6 Soporte de transacciones en SQL

La definición de transacción SQL es parecida a nuestro concepto de transacción ya definido. Es decir, es una unidad lógica de trabajo cuya atomicidad está garantizada. Una simple sentencia SQL se considera que siempre es atómica (tanto si completa la ejecución sin errores, como si falla y deja la base de datos sin cambios).

Con SQL, no hay ninguna sentencia de inicio de transacción explícita. El inicio de una transacción se hace implícitamente cuando se encuentran sentencias SQL particulares. Sin embargo, cada transacción debe tener una sentencia explícita de terminación, que puede ser COMMIT o ROLLBACK. Una transacción tiene ciertas

características atribuibles a ella, que se especifican en SQL con una sentencia SET TRANSACTION. Las características son el *modo de acceso*, el *tamaño del área de diagnóstico* y el *nivel de aislamiento*.

El **modo de acceso** puede especificarse como READ ONLY o READ WRITE. Lo predeterminado es READ WRITE, a menos que especifiquemos el nivel de aislamiento READ UNCOMMITTED, en cuyo caso se asume READ ONLY. El modo READ WRITE permite seleccionar, actualizar, insertar, eliminar y crear comandos para su ejecución. El modo READ ONLY, como su nombre indica, es simplemente para la recuperación de datos.

La opción del **tamaño del área de diagnóstico**, DIAGNOSTIC SIZE *n*, especifica un valor entero *n*, que indica el número de condiciones que se pueden mantener simultáneamente en el área de diagnóstico. Estas condiciones proporcionan retroalimentación (errores o excepciones) al usuario o al programa en la sentencia SQL recién ejecutada.

La opción de **nivel de aislamiento** se especifica con la sentencia ISOLATION LEVEL <aislamiento>, donde el valor para <aislamiento> puede ser READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ o SERIALIZABLE.[14] El nivel de aislamiento predeterminado es SERIALIZABLE, aunque algunos sistemas utilizan READ COMMITTED como su valor predeterminado. El uso del término SERIALIZABLE aquí está basado en no permitir violaciones que provoquen lecturas sucias, irrepetibles y fantasmas,[15] y, por tanto, no es idéntico a como se definió la serialización anteriormente en la Sección 17.5. Si una transacción se ejecuta a un nivel de aislamiento más bajo que SERIALIZABLE, entonces se pueden producir una o más de estas violaciones:

1. **Lectura sucia.** Una transacción $T_1$ puede leer la actualización de una transacción $T_2$, que todavía no se ha confirmado. Si $T_2$ falla y es cancelada, entonces $T_1$ habría leído un valor que no existe y es incorrecto.
2. **Lectura irrepetible.** Una transacción $T_1$ puede leer un valor dado de una tabla. Si otra transacción $T_2$ actualiza más tarde ese valor y $T_1$ lee de nuevo el valor, $T_1$ verá un valor diferente.
3. **Fantasmas.** Una transacción $T_1$ puede leer un conjunto de filas de una tabla, quizá basándose en alguna condición especificada en la cláusula WHERE de SQL. Ahora, suponga que una transacción $T_2$ inserta una fila nueva que también satisface la condición de la cláusula WHERE utilizada en $T_1$, en la tabla utilizada por $T_1$. Si $T_1$ se repite, entonces $T_1$ verá un fantasma, una fila que anteriormente no existía.

La Tabla 17.1 resume las posibles violaciones para los distintos niveles de aislamiento. Una entrada con la palabra "Sí" indica que es posible una violación, y una entrada con "No" indica que no es posible. READ UNCOMMITTED es la opción más clemente, mientras que SERIALIZABLE es la más restrictiva al evitar los tres problemas mencionados anteriormente.

**Tabla 17.1.** Posibles violaciones según el nivel de aislamiento.

| | Tipo de violación | | |
|---|---|---|---|
| **Nivel de aislamiento** | **Lectura sucia** | **Lectura irrepetible** | **Fantasma** |
| READ UNCOMMITTED | Sí | Sí | Sí |
| READ COMMITTED | No | Sí | Sí |
| REPEATABLE READ | No | No | Sí |
| SERIALIZABLE | No | No | No |

Una transacción SQL podría parecerse a lo siguiente:

[14] Son similares a los *niveles de aislamiento* que explicamos brevemente al final de la Sección 17.3.

[15] Los problemas de lectura sucia y de lectura irrepetible se explicaron en la Sección 17.1.3. Los fantasmas se explican en la Sección 18.6.1.

```
EXEC SQL WHENEVER SQLERROR GOTO UNDO;
EXEC SQL SET TRANSACTION
   READ WRITE
   DIAGNOSTIC SIZE 5
   ISOLATION LEVEL SERIALIZABLE;
EXEC SQL INSERT INTO EMPLEADO (Nombre, Apellidos, Dni, Dno, Sueldo)
   VALUES ('Luis', 'Campos', '991004321', 2, 35000);
EXEC SQL UPDATE EMPLEADO
   SET Sueldo = Sueldo * 1.1 WHERE Dno = 2;
EXEC SQL COMMIT;
GOTO THE_END;
UNDO: EXEC SQL ROLLBACK;
THE_END: ... ;
```

La transacción anterior primero inserta una nueva fila en la tabla EMPLEADO y, después, actualiza el sueldo de todos los empleados del departamento 2. Si se produce un error en cualquiera de las sentencias SQL, la transacción entera es anulada. Esto implica que los sueldos actualizados (por esta transacción) deberían restaurarse a su valor antiguo y que la fila recién insertada debería eliminarse.

Como hemos visto, SQL proporciona algunas características orientadas a las transacciones. El DBA o los programadores de la base de datos pueden beneficiarse de esas opciones para intentar mejorar el rendimiento de la transacción relajando la serialización, siempre y cuando sea aceptable para sus aplicaciones.

# 17.7 Resumen

En este capítulo hemos explicado los conceptos DBMS relacionados con el procesamiento de las transacciones. Hemos introducido el concepto de transacción, así como las operaciones relacionadas con el procesamiento de las mismas. Asimismo, hemos comparado los sistemas monousuario con los sistemas multiusuario, y hemos presentado algunos ejemplos de cómo una ejecución incontrolada de transacciones concurrentes en un sistema multiusuario puede llevar a resultados y valores incorrectos en la base de datos. También hemos explicado los distintos tipos de fallos que pueden surgir durante la ejecución de una transacción.

A continuación, vimos los estados típicos por los que pasa una transacción durante su ejecución, y explicamos diversos conceptos que se utilizan en los métodos de recuperación y control de la concurrencia. El registro del sistema hace un seguimiento de los accesos a la base de datos, y el sistema utiliza esa información para recuperarse de los fallos. Una transacción puede tener éxito y alcanzar su punto de confirmación, o puede fallar y tener que anularse. Una transacción confirmada implica que sus cambios se han grabado permanentemente en la base de datos. Ofrecimos una visión general de las propiedades deseables de las transacciones (atomicidad, conservación de la consistencia, aislamiento y durabilidad), que con frecuencia reciben el nombre de propiedades ACID (por sus iniciales en inglés).

Después definimos una planificación como una secuencia de ejecución de las operaciones de varias transacciones, con una posible interpolación. Clasificamos las planificaciones según su recuperabilidad. Las planificaciones recuperables garantizan que, una vez confirmada una transacción, nunca habrá necesidad de que haya que deshacerla. Las planificaciones que evitan la anulación en cascada añaden una condición adicional para garantizar que ninguna transacción cancelada requiera la cancelación en cascada de otras transacciones. Las planificaciones estrictas proporcionan una condición aún más fuerte que permite un esquema de recuperación sencillo consistente en restaurar los valores antiguos de los elementos modificados por una transacción cancelada.

Hemos definido la equivalencia de planificaciones y visto que una planificación serializable es equivalente a alguna planificación en serie. Hemos definido los conceptos de equivalencia por conflicto y de equivalencia por vista, que conducen a las definiciones de serialización por conflicto y serialización por vista. Una planificación serializable se considera correcta. Presentamos los algoritmos para probar la serialización (por conflicto) de una planificación. Explicamos por qué probar la serialización no es práctico en un sistema real, aun-

que puede utilizarse para definir y verificar los protocolos de control de la concurrencia, y mencionamos brevemente definiciones menos restrictivas de la equivalencia de planificaciones. Por último, ofrecimos una breve panorámica de cómo se utilizan los conceptos de transacción en la práctica dentro de SQL.

En el Capítulo 18 veremos los protocolos de control de la concurrencia, y en el Capítulo 19 los protocolos de recuperación.

## Preguntas de repaso

**17.1.** ¿Qué se entiende por ejecución concurrente de las transacciones de una base de datos en un sistema multiusuario? Explique por qué es necesario controlar la concurrencia, y ofrezca algunos ejemplos informales.

**17.2.** Explique los diferentes tipos de fallos. ¿A qué hace referencia un fallo catastrófico?

**17.3.** Explique las acciones llevadas a cabo por las operaciones read_item y write_item en una base de datos.

**17.4.** Dibuje un diagrama de estado y explique los estados típicos por los que pasa una transacción durante su ejecución.

**17.5.** ¿Para qué se utiliza el registro del sistema? ¿Cuáles son las clases típicas de entradas en un registro del sistema? ¿Qué son los puntos de confirmación de una transacción, y por qué son importantes?

**17.6.** Explique las propiedades de atomicidad, durabilidad, aislamiento y conservación de la consistencia de una transacción de base de datos.

**17.7.** ¿Qué es una planificación? Defina los conceptos de planificaciones recuperables, planificaciones que evitan la anulación en cascada y planificaciones estrictas, y compárelas según su recuperabilidad.

**17.8.** Explique las diferentes medidas de la equivalencia de transacciones. ¿Cuál es la diferencia entre equivalencia por conflicto y equivalencia por vista?

**17.9.** ¿Qué es una planificación en serie? ¿Qué es una planificación serializable? ¿Por qué se considera que una planificación en serie es correcta? ¿Por qué se considera que una planificación serializable es correcta?

**17.10.** ¿Cuál es la diferencia entre las suposiciones de escritura restringida y escritura no restringida? ¿Cuál es más realista?

**17.11.** Explique cómo se utiliza la serialización para implementar el control de la concurrencia en un sistema de base de datos. ¿Por qué a veces se considera que la serialización es demasiado restrictiva como medida de exactitud para las planificaciones?

**17.12.** Describa los cuatro niveles de aislamiento en SQL.

**17.13.** Defina las violaciones causadas por cada una de estas circunstancias: lectura sucia, lectura irrepetible y fantasmas.

## Ejercicios

**17.14.** Cambie la transacción $T_2$ de la Figura 17.2(b) para que se lea:

```
read_item(X);
X := X + M;
if X > 90 then exit
else write_item(X);
```

Explique el resultado final de las diferentes planificaciones de la Figura 17.3(a) y (b), donde $M = 2$ y $N = 2$, respecto a las siguientes cuestiones. ¿La adición de la condición anterior cambia

el resultado final? ¿El resultado obedece la regla de consistencia implícita (que la capacidad de $X$ es 90)?

**17.15.** Repita el Ejercicio 17.14, añadiendo una comprobación en $T_1$ para que $Y$ no exceda de 90.

**17.16.** Añada la operación commit al final de cada una de las transacciones, $T_1$ y $T_2$, de la Figura 17.2; después, liste todas las posibles planificaciones para las transacciones modificadas. Determine cuál de las planificaciones es recuperable, cuál evita la anulación en cascada, y cuál es estricta.

**17.17.** Liste todas las planificaciones posibles para las transacciones $T_1$ y $T_2$ de la Figura 17.2, y determine cuáles son serializables por conflicto (correctas) y cuáles no.

**17.18.** ¿Cuántas planificaciones *en serie* existen para las tres transacciones de la Figura 17.8(a)? ¿Cuáles son? ¿Cuál es la cantidad total de planificaciones posibles?

**17.19.** Escriba un programa para crear todas las planificaciones posibles para las tres transacciones de la Figura 17.8(a), y para determinar cuáles de ellas son serializables por conflicto y cuáles no. Por cada planificación serializable por conflicto, su programa debe imprimir la planificación y listar todas las planificaciones en serie equivalentes.

**17.20.** ¿Por qué en SQL se necesita una sentencia explícita de final de transacción, pero no una sentencia explícita de inicio?

**17.21.** Describa situaciones en las que cada uno de los distintos niveles de aislamiento sea de utilidad para el procesamiento de transacciones.

**17.22.** ¿Cuáles de las siguientes planificaciones son serializables (por conflicto)? Por cada planificación serializable, determine las planificaciones en serie equivalentes.

**a.** $r_1(X); r_3(X); w_1(X); r_2(X); w_3(X);$

**b.** $r_1(X); r_3(X); w_3(X); w_1(X); r_2(X);$

**c.** $r_3(X); r_2(X); w_3(X); r_1(X); w_1(X);$

**d.** $r_3(X); r_2(X); r_1(X); w_3(X); w_1(X);$

**17.23.** Considere las transacciones $T_1$, $T_2$ y $T_3$, y las planificaciones $S_1$ y $S_2$ que se ofrecen a continuación. Dibuje los gráficos de serialización (precedencia) para $S_1$ y $S_2$, y explique si cada planificación es o no es serializable. Si una planificación es serializable, escriba la(s) planificación(es) en serie equivalente(s).

$T_1$: $r_1(X); r_1(Z); w_1(X);$

$T_2$: $r_2(Z); r_2(Y); w_2(Z); w_2(Y);$

$T_3$: $r_3(X); r_3(Y); w_3(Y);$

$S_1$: $r_1(X); r_2(Z); r_1(Z); r_3(X); r_3(Y); w_1(X); w_3(Y); r_2(Y); w_2(Z); w_2(Y);$

$S_2$: $r_1(X); r_2(Z); r_3(X); r_1(Z); r_2(Y); r_3(Y); w_1(X); w_2(Z); w_3(Y); w_2(Y);$

**17.24.** Considere las planificaciones *S3*, *S4* y *S5*. Determine si cada planificación es estricta, que evita la anulación en cascada, recuperable o no recuperable. (Determine la condición de recuperabilidad más estricta que cada planificación satisfaga).

$S_3$: $r_1(X); r_2(Z); r_1(Z); r_3(X); r_3(Y); w_1(X); c_1; w_3(Y); c_3; r_2(Y); w_2(Z); w_2(Y); c_2;$

$S_4$: $r_1(X); r_2(Z); r_1(Z); r_3(X); r_3(Y); w_1(X); w_3(Y); r_2(Y); w_2(Z); w_2(Y); c_1; c_2; c_3;$

$S_5$: $r_1(X); r_2(Z); r_3(X); r_1(Z); r_2(Y); r_3(Y); w_1(X); c_1; w_2(Z); w_3(Y); w_2(Y); c_3; c_2;$

## Bibliografía seleccionada

Los conceptos de serialización se introdujeron en Gray y otros (1975). El concepto de transacción de base de datos se explicó por primera vez en Gray (1981). Gray ganó el codiciado ACM Turing Award en 1998 por su

trabajo sobre las transacciones en las bases de datos y la implementación de transacciones en los DBMSs relacionales. Bernstein, Hadzilacos, y Goodman (1987) se centra en las técnicas de control de la concurrencia y de recuperación en los sistemas de bases de datos centralizados y distribuidos; es una referencia excelente. Papadimitriou (1986) ofrece una perspectiva más teórica. Un gran libro de referencia de más de mil páginas es Gray y Reuter (1993), que ofrece una perspectiva más práctica de los conceptos y las técnicas del procesamiento de transacciones. Elmagarmid (1992) y Bhargava (1989) ofrecen colecciones de artículos de investigación sobre el procesamiento de transacciones. El soporte de transacciones en SQL se describe en Date y Darwen (1993). La serialización por vista se define en Yannakakis (1984). La recuperación de las planificaciones se explica en Hadzilacos (1983, 1988).

CAPÍTULO 18

# Técnicas de control de la concurrencia

En este capítulo explicamos algunas técnicas de control de la concurrencia que se utilizan para garantizar la ausencia de interferencias o la propiedad de aislamiento de las transacciones que se ejecutan simultáneamente. La mayoría de estas técnicas garantizan la serialización de las planificaciones (consulte la Sección 17.5), utilizando **protocolos** (conjuntos de reglas) que garantizan esa serialización. Un importante conjunto de protocolos emplea la técnica del **bloqueo** de los elementos de datos para evitar que varias transacciones accedan concurrentemente a los elementos; en la Sección 18.1 se describen algunos protocolos de bloqueo, que se utilizan en casi todos los DBMSs comerciales. Otro conjunto de protocolos de control de la concurrencia utilizan las **marcas de tiempo**. Una marca de tiempo es un identificador único generado por el sistema para cada transacción. Los protocolos de control de la concurrencia que utilizan la ordenación de marcas de tiempo para garantizar la serialización se describen en la Sección 18.2. En la Sección 18.3 explicamos los protocolos de control de la concurrencia **multiversión**, que utilizan varias versiones de un elemento de datos. En la Sección 18.4 presentamos un protocolo basado en el concepto de **validación** o **certificación** de una transacción después de que haya ejecutado sus operaciones; estos protocolos se denominan a veces **protocolos optimistas**.

Otro factor que afecta al control de la concurrencia es la **granularidad** de los elementos de datos (es decir, qué porción de la base de datos es representada por un elemento de datos). Un elemento puede ser tan pequeño como el valor de un atributo (campo) o tan grande como un bloque de disco, o incluso un fichero entero o la base de datos entera. En la Sección 18.5 explicamos la granularidad de los elementos. En la Sección 18.6 explicamos los problemas que pueden surgir con el control de la concurrencia cuando se utilizan los índices para procesar las transacciones. Por último, en la Sección 18.7 explicamos algunos problemas relacionados con el control de la concurrencia.

Son suficientes las Secciones 18.1, 18.5, 18.6 y 18.7, y posiblemente la 18.3.2, si el interés principal es una introducción a las técnicas de control de la concurrencia que más a menudo se utilizan en la práctica. Las demás técnicas tienen un interés principalmente teórico.

## 18.1 Técnicas de bloqueo en dos fases para controlar la concurrencia

Algunas de las principales técnicas que se utilizan para controlar la ejecución concurrente de transacciones están basadas en el concepto de bloqueo de elementos de datos. Un **bloqueo** es una variable asociada a un ele-

mento de datos que describe el estado de ese elemento respecto a las posibles operaciones que se le puedan aplicar. Generalmente, hay un bloqueo por cada elemento de datos de la base de datos. Los bloqueos se utilizan como un medio para sincronizar el acceso de las transacciones concurrentes a los elementos de la base de datos. En la Sección 18.1.1 explicamos la naturaleza y los tipos de bloqueos. Después, en la Sección 18.1.2 presentamos los protocolos que utilizan el bloqueo para garantizar la serialización de las planificaciones de transacciones. Por último, en la Sección 18.1.3 explicamos dos problemas asociados con el uso de bloqueos (interbloqueo e inanición), así como su manipulación.

## 18.1.1 Tipos de bloqueos y tablas de bloqueo del sistema

En el control de la concurrencia se utilizan varios tipos de bloqueos. A fin de introducir gradualmente los conceptos de bloqueo, primero explicamos los bloqueos binarios, que son sencillos pero restrictivos, por lo que no se utilizan en la práctica. Después explicamos los bloqueos compartidos/exclusivos, que ofrecen unas capacidades de bloqueo más generales y que se utilizan en la práctica en los esquemas de bloqueo de bases de datos. En la Sección 18.3.2 describimos un bloqueo de certificación y mostramos cómo puede utilizarse para mejorar el rendimiento de los protocolos de bloqueo.

**Bloqueos binarios.** Un **bloqueo binario** puede tener dos **estados** o **valores:** bloqueado y desbloqueado (o 1 y 0, para simplificar). Cada elemento $X$ de la base de datos tiene un bloqueo distinto. Si el valor del bloqueo sobre $X$ es 1, el elemento $X$ *no podrá ser accedido* por una operación de base de datos que solicite el elemento. Si el valor del bloqueo sobre $X$ es 0, es posible acceder al elemento cuando es solicitado. Nos referiremos al valor (o estado) actual del bloqueo asociado con el elemento $X$ como **bloquear($X$)**.

Con el bloqueo binario se utilizan dos operaciones, bloquear_elemento y desbloquear_elemento. Una transacción solicita acceso a un elemento $X$ emitiendo primero una operación **bloquear_elemento($X$)**. Si BLOQUEAR($X$)=1, la transacción está obligada a esperar. Si BLOQUEAR($X$)=0, se establece a 1 (la transacción **bloquea** el elemento) y la transacción tiene permiso para acceder al elemento $X$. Cuando la transacción ha terminado de utilizar el elemento, emite una operación **desbloquear_elemento($X$)**, que asigna 0 a BLOQUEAR($X$) **(desbloquea** el elemento) por lo que otras transacciones pueden acceder a $X$. Por tanto, un bloqueo binario impone la **exclusión mutua** en el elemento de datos. En la Figura 18.1 se muestra una descripción de las operaciones bloquear_elemento($X$) y desbloquear_elemento($X$).

Las operaciones bloquear_elemento y desbloquear_elemento deben implementarse como unidades indivisibles (conocidas como **secciones críticas** en los sistemas operativos); es decir, no debe permitirse la interpolación una vez iniciada una operación de bloqueo o desbloqueo hasta que la operación termina o la transacción espera. En la Figura 18.1, el comando esperar dentro de la operación bloquear_elemento($X$) se implementa normalmente colocando la transacción en una cola de espera para el elemento $X$ hasta que se desbloquea éste y la transacción obtiene acceso a él. Las demás transacciones que también quieren acceder a $X$ se colocan en la misma cola. Por tanto, se considera que el comando esperar está fuera de la operación bloquear_elemento.

Es muy fácil implementar un bloqueo binario; basta con una variable de tipo binario, BLOQUEAR, asociada a cada elemento de datos $X$ de la base de datos. En su forma más sencilla, un bloqueo puede ser un registro con tres campos: <Nombre_elemento_datos, BLOQUEAR, Transacción_de_bloqueo>, más una cola para las transacciones que están esperando a acceder al elemento. El sistema sólo necesita guardar registros de este tipo para los elementos que actualmente están bloqueados, y lo hace en una **tabla de bloqueo**, que puede organizarse como un fichero de dispersión. Los elementos que no figuran en esta tabla están desbloqueados. El DBMS tiene un **subsistema gestor de bloqueos** que rastrea y controla el acceso a los bloqueos.

En caso de utilizar este sencillo esquema de bloqueo binario, cada transacción debe cumplir las siguientes reglas:

1. Una transacción $T$ debe emitir la operación bloquear_elemento($X$) antes de que se ejecute cualquier operación leer_elemento($X$) o escribir_elemento($X$) en $T$.

**Figura 18.1.** Operaciones de bloqueo y desbloqueo para lo bloqueos binarios.

```
bloquear_elemento(X):
B: Si BLOQUEAR(X) = 0 (* elemento desbloqueado *)
       then BLOQUEAR(X) ← 1 (* bloquear el elemento *)
   entonces
       inicio
       esperar (hasta BLOQUEAR(X) = 0
           y el gestor de bloqueo retoma la transacción);
       ir a B
       fin;

desbloquear_elemento(X):
   BLOQUEAR(X) ← 0; (* desbloquear el elemento *)
   Si hay transacciones esperando
       entonces retomar una de las transacciones que espera;
```

2. Una transacción $T$ debe emitir la operación desbloquear_elemento($X$) después de haberse completado todas las operaciones leer_elemento($X$) y escribir_elemento($X$) de $T$.
3. Una transacción $T$ no emitirá una operación bloquear_elemento($X$) si ya posee el bloqueo del elemento $X$.[1]
4. Una transacción $T$ no emitirá una operación desbloquear_elemento($X$) a menos que ya posea el bloqueo del elemento $X$.

Estas reglas puede implementarlas el módulo gestor de bloqueos del DBMS. Entre las operaciones bloquear_elemento($X$) y desbloquear_elemento($X$) de la transacción $T$, se dice que $T$ **posee el bloqueo** del elemento $X$. A lo sumo, una transacción puede poseer el bloqueo de un elemento en particular. De este modo, dos transacciones no pueden acceder simultáneamente al mismo elemento.

**Bloqueos compartidos/exclusivos (o lectura/escritura).** El esquema de bloqueo binario anterior es demasiado restrictivo para los elementos de base de datos porque, como máximo, sólo una transacción puede poseer un bloqueo sobre un elemento dado. Debemos permitir que varias transacciones tengan acceso al mismo elemento $X$ si todas ellas acceden a $X$ sólo para leer. Sin embargo, si una transacción va a escribir un elemento $X$, debe tener acceso exclusivo a $X$. Con este fin, se utiliza un tipo de bloqueo diferente denominado **bloqueo de modo múltiple**. En este esquema (denominado **bloqueos compartido/exclusivo** o **de lectura/escritura**) hay tres operaciones de bloqueo: bloquear_lectura($X$), bloquear_escritura($X$) y desbloquear($X$). Un bloqueo asociado con un elemento $X$, BLOQUEAR($X$), tiene ahora tres posibles estados: *bloqueado para lectura*, *bloqueado para escritura* o *desbloqueado*. Un **elemento bloqueado para lectura** también se denomina de **lectura compartida** porque otras transacciones pueden leer el elemento, mientras que un **elemento bloqueado para escritura** se denomina de **escritura exclusiva** porque una sola transacción posee en exclusiva el bloqueo de un elemento.

Un método para implementar las operaciones anteriores en un bloqueo de lectura/escritura es hacer un seguimiento del número de transacciones que poseen un bloqueo compartido (lectura) sobre un elemento de la tabla de bloqueo. Cada registro de dicha tabla tendrá cuatro campos: <Nombre_elemento_datos, BLOQUEAR, Número_de_lecturas, Transacción(es)_bloqueo(s)>. Una vez más, para ahorrar espacio, el sistema debe mantener registros de bloqueo sólo para los elementos bloqueados de la tabla de bloqueo. El valor (estado) de BLOQUEAR puede ser bloqueado para lectura o bloqueado para escritura, adecuadamente codificado (si

[1] Esta regla se puede eliminar si modificamos la operación bloquear_elemento($X$) de la Figura 18.1 para que, si el elemento está actualmente bloqueado *por la transacción solicitante*, el bloqueo sea concedido.

asumimos que no se guardan registros en la tabla de bloqueo para los elementos desbloqueados). Si BLOQUEAR($X$)=bloqueado para escritura, el valor de Transacción(es)_bloqueo(s) es una sola transacción que almacena el bloqueo exclusivo (escritura) de $X$. Si BLOQUEAR($X$)=bloqueado para lectura, el valor de Transacción(es)_bloqueo(s) es una lista de una o más transacciones que poseen el bloqueo compartido (lectura) de $X$. En la Figura 18.2 se describen las tres operaciones, bloquear_lectura($X$), bloquear_escritura($X$) y desbloquear($X$).[2] Como antes, cada una de las tres operaciones debe considerarse como indivisible; no debe permitirse la interpolación una vez iniciada una de las operaciones, hasta que la operación termina concediendo el bloqueo, o la transacción se coloca en una cola de espera para el elemento.

Cuando utilizamos el esquema de bloqueo compartido/exclusivo, el sistema debe implementar las siguientes reglas:

1. Una transacción $T$ debe emitir la operación bloquear_lectura($X$) o bloquear_escritura($X$) antes de que se ejecute cualquier operación leer_elemento($X$) de $T$.
2. Una transacción $T$ debe emitir la operación bloquear_escritura($X$) antes de que se ejecute cualquier operación escribir_elemento($X$) de $T$.
3. Una transacción $T$ debe emitir la operación desbloquear($X$) una vez que se hayan completado todas las operaciones leer_elemento($X$) y escribir_elemento($X$) de $T$.[3]
4. Una transacción $T$ no emitirá una operación bloquear_lectura($X$) si ya posee un bloqueo de lectura (compartido) o de escritura (exclusivo) para el elemento $X$. Esta regla se puede hacer menos estricta, como veremos en breve.
5. Una transacción $T$ no emitirá una operación bloquear_escritura($X$) si ya posee un bloqueo de lectura (compartido) o de escritura (exclusivo) para el elemento $X$. Esta regla se puede hacer menos estricta, como veremos en breve.
6. Una transacción $T$ no emitirá una operación desbloquear($X$) a menos que ya posea un bloqueo de lectura (compartido) o de escritura (exclusivo) sobre el elemento $X$.

**Conversión de bloqueos.** En ocasiones, es deseable hacer menos estrictas las condiciones 4 y 5 del listado anterior para permitir una **conversión del bloqueo**; es decir, una transacción que ya posee un bloqueo sobre el elemento $X$ tiene permitido, bajo ciertas condiciones, **convertir** el bloqueo de un estado a otro. Por ejemplo, es posible para una transacción $T$ emitir una operación bloquear_lectura($X$) y más tarde **promocionar** el bloqueo emitiendo una operación bloquear_escritura($X$). Si $T$ es la única transacción que posee un bloqueo de lectura sobre $X$ en el momento de emitir la operación bloquear_escritura($X$), el bloqueo puede promocionarse; en caso contrario, la transacción debe esperar. También es posible para una transacción $T$ emitir una operación bloquear_escritura($X$) y más tarde **degradar** el bloqueo emitiendo una operación bloquear_lectura($X$). Cuando se utiliza la promoción y la degradación de bloqueos, la tabla de bloqueo debe incluir los identificadores de transacción en la estructura de registro de cada bloqueo [en el campo Transacción(es)_bloqueo(s)] para almacenar la información sobre las transacciones que poseen bloqueos sobre el elemento. Las descripciones de las operaciones bloquear_lectura($X$) y bloquear_escritura($X$) de la Figura 18.2 deben modificarse adecuadamente. Lo dejamos como ejercicio para el lector.

Con los bloqueos binarios o bloqueos de lectura/escritura en las transacciones, como describimos anteriormente, no está garantizada la serialización de las planificaciones. La Figura 18.3 muestra un ejemplo en el que se han seguido las reglas de bloqueo anteriores pero el resultado puede ser una planificación no serializable. Esto se debe a que en la Figura 18.3(a) los elementos $Y$ de $T_1$ y $X$ de $T_2$ se desbloquearon demasiado pronto. Esto permite que se produzca una planificación como la de la Figura 18.3(c), que no es una planificación

[2] Estos algoritmos no permiten la *actualización* o la *degradación* de los bloqueos, como se describe posteriormente en esta sección. El lector puede ampliar los algoritmos para que permitan estas operaciones adicionales.

[3] Esta regla se puede relajar para que una transacción pueda desbloquear un elemento, y bloquearlo de nuevo más tarde.

**Figura 18.2.** Bloqueo y desbloqueo de operaciones para bloqueos de dos modos (lectura/escritura o compartido/exclusivo).

```
bloquear_lectura(X):
B: Si BLOQUEAR(X) 5 "desbloqueado"
   entonces inicio BLOQUEAR(X) ← "boqueado para lectura";
      Número_de_lecturas(X) ←1
      fin
sino Si BLOQUEAR(X) 5 "bloqueado para lectura"
   entonces Número_de_lecturas(X) ← Número_de_lecturas(X) 1 1
sino  inicio
   esperar (hasta BLOQUEAR(X) 5 "desbloqueado"
      y el gestor de bloqueos retoma la transacción);
   ir a B
   fin;

bloquear_escritura(X):
B: Si BLOQUEAR(X) 5 "desbloqueado"
      entonces BLOQUEAR(X) ← "bloqueado para escritura"
   sino  inicio
         esperar (hasta BLOQUEAR(X) 5 "desbloqueado"
            y el gestor de bloqueos retoma la transacción);
         ir a B
         fin;

desbloquear(X):
Si BLOQUEAR(X) 5 "bloqueado para escritura"
   entonces inicio BLOQUEAR(X) ← "desbloqueado";
            retomar una de las transacciones en espera, si las hay
            fin
sino Si BLOQUEAR(X) 5 "bloqueado para lectura"
   entonces inicio
            Número_de_lecturas(X) ← Número_de_lecturas(X) – 1;
            Si Número_de_lecturas(X) 5 0
               entonces inicio BLOQUEAR(X) 5 "desbloqueado";
                        retomar una de las transacciones en espera, si las hay
                        fin
            fin;
```

serializable y, por tanto, ofrece unos resultados incorrectos. Para garantizar la serialización, debemos seguir *un protocolo adicional* relativo al posicionamiento de las operaciones de bloqueo y desbloqueo en cada transacción. El protocolo mejor conocido, el bloqueo en dos fases, se describe en la siguiente sección.

## 18.1.2 Garantía de la serialización por el bloqueo en dos fases

Una transacción obedece el **protocolo de bloqueo en dos fases** si *todas* las operaciones de bloqueo (bloquear_lectura, bloquear_escritura) preceden a la *primera* operación de desbloqueo de la transacción.[4] Una

[4] No tiene relación con el protocolo de confirmación en dos fases para la recuperación en las bases de datos distribuidas (consulte el Capítulo 25).

**Figura 18.3.** Transacciones que no obedecen el bloqueo en dos fases. (a) Dos transacciones $T_1$ y $T_2$. (b) Resultado de posibles planificaciones en serie de $T_1$ y $T_2$. (c) Una planificación no serializable $S$ que utiliza bloqueos.

**(a)**

| $T_1$ | $T_2$ |
|---|---|
| bloquear_lectura($Y$);<br>leer_elemento($Y$);<br>desbloquear($Y$);<br>bloquear_escritura($X$);<br>leer_elemento($X$);<br>$X := X + Y$;<br>escribir_elemento($X$);<br>desbloquear($X$); | bloquear_lectura($X$);<br>leer_elemento($X$);<br>desbloquear($X$);<br>bloquear_escritura($Y$);<br>leer_elemento($Y$);<br>$Y := X + Y$;<br>escribir_elemento($Y$);<br>desbloquear($Y$); |

**(b)** Valores iniciales: $X$=20, $Y$=30

Resultado de la planificación en serie $T_1$ seguida de $T_2$: $X$=50, $Y$=80

Resultado de la planificación en serie $T_2$ seguida de $T_1$: $X$=70, $Y$=50

**(c)**

Tiempo

| $T_1$ | $T_2$ |
|---|---|
| bloquear_lectura($Y$);<br>leer_elemento($Y$);<br>desbloquear($Y$); | |
| | bloquear_lectura($X$);<br>leer_elemento($X$);<br>desbloquear($X$);<br>bloquear_escritura($Y$);<br>leer_elemento($Y$);<br>$Y := X + Y$;<br>escribir_elemento($Y$);<br>desbloquear($Y$); |
| bloquear_escritura($X$);<br>leer_elemento($X$);<br>$X := X + Y$;<br>escribir_elemento($X$);<br>desbloquear($X$); | |

Resultado de la planificación $S$:
$X$=50, $Y$=50
(no serializable)

transacción de este tipo puede dividirse en dos fases: una **primera fase de expansión** o **crecimiento**, durante la cual pueden adquirirse bloqueos nuevos sobre los elementos, pero no pueden liberarse; y una **segunda fase de reducción**, en la que los bloqueos existentes se pueden liberar pero no se pueden adquirir bloqueos nuevos. Si está permitida la conversión de bloqueos, la promoción de los bloqueos (de bloqueado para lectura a bloqueado para escritura) debe realizarse durante la fase de expansión, y la degradación de los bloqueos (de bloqueado para escritura a bloqueado para lectura) debe realizarse en la segunda fase. Por tanto, una operación bloquear_lectura($X$) que degrada un bloqueo de escritura que se posee sobre $X$ sólo puede aparecer en la segunda fase.

Las transacciones $T_1$ y $T_2$ de la Figura 18.3(a) no obedecen el protocolo de bloqueo en dos fases porque la operación bloquear_escritura($X$) sigue a la operación desbloquear($Y$) de $T_1$, y, de forma parecida, la operación bloquear_escritura($Y$) sigue a la operación desbloquear($X$) de $T_2$. Si implementamos el bloqueo en dos fases, las transacciones pueden reescribirse como $T_1$' y $T_2$' (véase la Figura 18.4). Ahora, la planificación de la Figura 18.3(c) no está permitida para $T_1$' y $T_2$' (con el orden modificado de sus operaciones de bloqueo y desbloqueo) bajo las reglas de bloqueo descritas en la Sección 18.1.1 porque $T_1$' emitirá su operación bloquear_escritura($X$)

**Figura 18.4.** Transacciones $T_1'$ y $T_2'$, que son iguales a las transacciones $T_1$ y $T_2$ de la Figura 18.3, pero obedecen el protocolo de bloqueo en dos fases. Observe que pueden provocar un interbloqueo.

| $T_1'$ |
|---|
| bloquear_lectura($Y$);<br>leer_elemento($Y$);<br>bloquear_escritura($X$);<br>desbloquear($Y$)<br>leer_elemento($X$);<br>$X := X + Y$;<br>escribir_elemento($X$);<br>desbloquear($X$); |

| $T_2'$ |
|---|
| bloquear_lectura($X$);<br>leer_elemento($X$);<br>bloquear_escritura($Y$);<br>desbloquear($X$)<br>leer_elemento($Y$);<br>$Y := X + Y$;<br>escribir_elemento($Y$);<br>desbloquear($Y$); |

*antes* de desbloquear el elemento $Y$; en consecuencia, cuando $T_2'$ emite su operación bloquear_lectura($X$), se ve obligada a esperar hasta que $T_1'$ libera el bloqueo emitiendo una operación desbloquear($X$) en la planificación.

Puede demostrarse que, si *cada* transacción de una planificación obedece el protocolo de bloqueo en dos fases, *la planificación es serializable*, obviando la necesidad de comprobar la serialización de las planificaciones. El mecanismo de bloqueo, por la implementación de las reglas de bloqueo en dos fases, también implementa la serialización.

El bloqueo en dos fases puede limitar la cantidad de concurrencia que puede darse en una planificación, porque una transacción $T$ no puede liberar un elemento $X$ después de haberlo usado si $T$ debe bloquear más tarde un elemento adicional $Y$; o, por el contrario, $T$ debe bloquear el elemento adicional $Y$ antes de que lo necesite para poder liberar $X$. Por tanto, $X$ debe permanecer bloqueado por $T$ hasta que todos los elementos que la transacción tiene que leer o escribir hayan sido bloqueados; sólo entonces podrá $T$ liberar $X$. Mientras tanto, otra transacción que pretenda acceder a $X$ puede verse obligada a esperar, aunque $T$ haya terminado con $X$; por el contrario, si $Y$ es bloqueado antes de que se necesite, otra transacción que pretende acceder a $Y$ está obligada a esperar aunque $T$ ya no esté utilizando $Y$. Éste es el precio por garantizar la serialización de todas las planificaciones sin tener que comprobar las propias planificaciones.

**Bloqueo en dos fases básico, conservador, estricto y riguroso.** Son variaciones del bloqueo en dos fases (2PL). La técnica que acabamos de describir se conoce como **2PL básico**. Una variación conocida como **2PL conservador** (o **2PL estático**) requiere una transacción para bloquear todos los elementos a los que tendrá acceso *antes de comenzar a ejecutarse*, mediante la declaración previa de los conjuntos de lectura y escritura. Como recordará de la Sección 17.1.2, el **conjunto de lectura** de una transacción es el conjunto de todos los elementos que la transacción lee, y el **conjunto de escritura** es el conjunto de todos los elementos que escribe. Si no es posible bloquear cualquiera de los elementos predeclarados necesarios, la transacción no bloqueará ningún elemento; en cambio, esperará a que puedan bloquearse todos los elementos. El 2PL conservador es un protocolo libre de interbloqueos, como veremos en la Sección 18.1.3 cuando expliquemos el problema del interbloqueo. Sin embargo, es difícil de utilizar en la práctica debido a la necesidad de predeclarar los conjuntos de lectura y escritura, algo que no es posible en la mayoría de las situaciones.

En la práctica la variación más popular de 2PL es 2PL estricto, que asegura las planificaciones estrictas (consulte la Sección 17.4). En esta variación, una transacción $T$ no libera ninguno de sus bloqueos exclusivos (escritura) hasta *después* de confirmarse o abortar. Por tanto, ninguna otra transacción puede leer o escribir un elemento que es escrito por $T$ a menos que ésta se confirme, lo que lleva a una planificación estricta en cuanto a recuperabilidad. 2PL estricto no está libre de los interbloqueos. Una variación más restrictiva de 2PL estricto es **2PL riguroso**, que también garantiza planificaciones estrictas. En esta variación, una transacción

$T$ no libera ninguno de sus bloqueos (exclusivos o compartidos) hasta después de su confirmación o cancelación. Es más fácil de implementar que 2PL estricto. La diferencia entre 2PL conservador y riguroso es que el primero debe bloquear todos sus elementos *antes de iniciarse*, de modo que una vez que la transacción empieza se encuentra en la segunda fase; el riguroso no desbloquea ninguno de sus elementos hasta *después de terminar* (por confirmación o por cancelación), de modo que la transacción se encuentra en la primera fase hasta que termina.

En muchos casos, el **subsistema de control de la concurrencia** es el responsable de generar las solicitudes de bloqueo para lectura y bloqueo para escritura. Por ejemplo, suponga que el sistema está preparado para el protocolo 2PL estricto. Después, siempre que la transacción $T$ emita una operación leer_elemento($X$), el sistema llamará a una operación bloquear_lectura($X$) en nombre de $T$. Si el estado de BLOQUEAR($X$) es bloqueado para escritura por alguna otra transacción $T'$, el sistema coloca $T$ en la cola de espera para el elemento $X$; en caso contrario, concede la solicitud bloquear_lectura($X$) y permite la ejecución de la operación leer_elemento($X$) de $T$. Por otro lado, si la transacción $T$ emite una operación escribir_elemento($X$), el sistema llama a la operación bloquear_escritura($X$) en nombre de $T$. Si el estado de BLOQUEAR($X$) es bloqueado para escritura o bloqueado para lectura por alguna otra transacción $T'$, el sistema coloca $T$ en la cola de espera para el elemento $X$; si el estado de BLOQUEAR($X$) es bloqueado para lectura y $T$ es la única transacción que posee el bloqueo de lectura sobre $X$, el sistema promociona el bloqueo a bloqueado para escritura y permite la operación escribir_elemento($X$) por $T$; por último, si el estado de BLOQUEAR($X$) es desbloqueado, el sistema concede la solicitud bloquear_escritura($X$) y permite la ejecución de la operación escribir_elemento($X$). Después de cada acción, el sistema debe actualizar en consecuencia su tabla de bloqueo.

Aunque el protocolo de boqueo en dos fases garantiza la serialización (es decir, cada planificación que se permite es serializable), no permite *todas las posibles* planificaciones serializables (es decir, algunas planificaciones serializables serán prohibidas por el protocolo). Además, el uso de bloqueos puede provocar dos problemas adicionales: interbloqueo e inanición. En la siguiente sección explicamos estos problemas y sus soluciones.

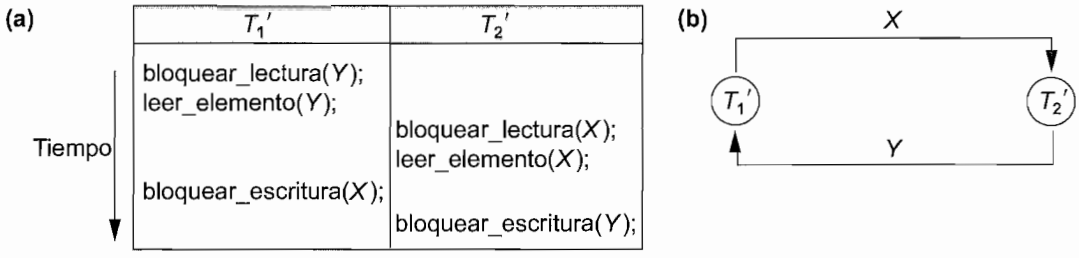
### 18.1.3 Interbloqueos e inanición

El **interbloqueo** se produce cuando *cada* transacción $T$ en un conjunto de *dos o más transacciones* está esperando a algún elemento que está bloqueado por alguna otra transacción $T'$ de dicho conjunto. Por tanto, cada transacción del conjunto está parada en espera a que una de las demás transacciones del conjunto libere el bloqueo sobre un elemento. En la Figura 18.5(a) se muestra un ejemplo sencillo, donde las transacciones $T_1'$ y $T_2'$ están interbloqueadas en una planificación parcial; $T_1'$ está esperando a $X$, que está bloqueado por $T_2'$, mientras que $T_2'$ está parada en espera de $Y$, que está bloqueado por $T_1'$. Mientras tanto, ni $T_1'$ ni $T_2'$ ni cualquier otra transacción puede acceder a los elementos $X$ e $Y$.

**Protocolos de prevención de interbloqueos.** Una forma de evitar el interbloqueo es utilizando un **protocolo de prevención del interbloqueo**,[5] que se utiliza en el bloqueo en dos fases conservador. Requiere que cada transacción bloquee *con antelación todos los elementos que necesita* (algo que normalmente no es un supuesto práctico); si alguno de los elementos no puede obtenerse, ninguno de los elementos se bloquea. En cambio, la transacción espera y después intenta de nuevo bloquear todos los elementos que necesita. Obviamente, esta solución limita la concurrencia. Un segundo protocolo, que también limita la concurrencia, consiste en *ordenar todos los elementos* de la base de datos y asegurarse de que una transacción que necesite varios elementos los bloqueará según ese orden. Esto obliga al programador (o al sistema) a conocer el orden de los elementos, que tampoco resulta práctico en el contexto de la base de datos.

[5] Estos protocolos no se utilizan en la práctica, ya sea debido a suposiciones poco realistas o una posible sobrecarga del sistema. La detección del interbloqueo y los tiempos limitados (siguiente sección) son más prácticos.

**Figura 18.5.** Ilustración del problema del interbloqueo. (a) Planificación parcial de $T_1'$ y $T_2'$ que está en estado de interbloqueo. (b) Un gráfico de espera para la planificación parcial de (a).




Se han propuesto otros esquemas de prevención del interbloqueo, que toman una decisión acerca de qué hacer con una transacción involucrada en una posible situación de interbloqueo: ¿debe bloquearse y hacerla esperar, o debe cancelarse, o debe apropiarse y abortar otra transacción? Estas técnicas utilizan el concepto de **marca de tiempo de transacción** TS($T$), que es un identificador único asignado a cada transacción. Normalmente, la marca de tiempo está basada en el orden en que las transacciones han sido iniciadas; por tanto, si la transacción $T_1$ empieza antes que la transacción $T_2$, entonces TS($T_1$) < TS($T_2$). La transacción *más antigua* tiene el valor de marca de tiempo *más pequeño*. Hay dos esquemas de prevención del interbloqueo, denominados esperar-morir y herir-esperar. Suponga que la transacción $T_i$ intenta bloquear un elemento $X$ pero no es capaz porque $X$ está bloqueado por alguna otra transacción $T_j$ con un bloqueo conflictivo. Las reglas que estos esquemas obedecen son las siguientes:

- **Esperar-morir.** Si TS($T_i$) < TS($T_j$), entonces ($T_i$ es más antigua que $T_j$) $T_i$ tiene permitido esperar; en caso contrario ($T_i$ es más nueva que $T_j$), aborta $T_i$ ($T_i$ *muere*) y se reinicia más tarde *con la misma marca de tiempo.*
- **Herir-esperar.** Si TS($T_i$) < TS($T_j$), entonces ($T_i$ es más antigua que $T_j$) aborta $T_j$ ($T_i$ *hiere a* $T_j$) y se reinicia más tarde *con la misma marca de tiempo;* en caso contrario ($T_i$ es más nueva que $T_j$), $T_i$ tiene que esperar.

En esperar-morir, una transacción más antigua tiene que esperar a una transacción más nueva, mientras que una transacción más nueva que solicita un elemento que posee una transacción más antigua es abortada y reiniciada. La metodología herir-esperar hace lo contrario: una transacción más nueva tiene que esperar a una transacción más antigua, mientras que una transacción más antigua que solicita un elemento que posee una transacción más nueva se *apropia* de la transacción más nueva abortándola. Los dos esquemas terminan con la cancelación de la transacción más nueva de las dos que pueden estar implicadas en un interbloqueo. Puede demostrarse que estas dos técnicas están *libres de interbloqueos*, puesto que en esperar-morir, las transacciones sólo esperan a las transacciones más nuevas, por lo que no se crean ciclos. De forma parecida, en herir-esperar, las transacciones sólo esperan a las transacciones más antiguas para que no se creen ciclos. Sin embargo, las dos técnicas pueden provocar que algunas transacciones aborten y reinicien innecesariamente, aunque estas transacciones *realmente nunca pueden provocar un interbloqueo.*

Otro grupo de protocolos que evitan el interbloqueo no requiere las marcas de tiempo; son los algoritmos de no espera y espera cautelosa. En el algoritmo de **no espera**, si una transacción no puede lograr un bloqueo, aborta y se reinicia tras un lapso de tiempo, sin comprobar si se ha producido o no un interbloqueo. Como este esquema puede provocar que las transacciones aborten y reinicien innecesariamente, se propuso el algoritmo de **espera cautelosa** para intentar reducir el número de abortos/reinicios innecesarios. Suponga que la transacción $T_i$ intenta bloquear un elemento $X$ pero no puede hacerlo porque $X$ está bloqueado por alguna otra transacción $T_j$ con un bloqueo por conflicto. Las reglas de la espera cautelosa son las siguientes:

- **Espera cautelosa.** Si $T_j$ no está bloqueada (no está esperando por algún otro elemento bloqueado), entonces $T_i$ es bloqueada y puede esperar; en caso contrario, $T_i$ aborta.

Se puede ver que la espera cautelosa está libre de interbloqueos, considerando el momento $b(T)$ en que cada transacción bloqueada $T$ fue bloqueada. Si las dos transacciones, $T_i$ y $T_j$, son bloqueadas, y $T_i$ está esperando a $T_j$, entonces $b(T_i) < b(T_j)$, puesto que $T_i$ sólo puede esperar a $T_j$ a la vez cuando $T_j$ no está bloqueada. Por tanto, los tiempos de bloqueo forman una ordenación total de todas las transacciones bloqueadas, por lo que no puede producirse ningún ciclo que provoque el interbloqueo.

**Detección del interbloqueo.** Esta metodología es más práctica para tratar con el interbloqueo; según ella, el sistema comprueba si realmente existe un estado de interbloqueo. Esta solución resulta atractiva si esperamos que haya poca interferencia entre las transacciones (es decir, si diferentes transacciones accederán raramente a los mismos elementos y al mismo tiempo). Esto puede suceder si las transacciones son cortas y cada una sólo bloquea unos cuantos elementos, o si la carga de transacciones es ligera. Por el contrario, si las transacciones son largas y cada una utiliza muchos elementos, o si la carga de la transacción es demasiado grande, puede ser beneficioso utilizar un esquema de prevención del interbloqueo.

Una forma sencilla de detectar un estado de interbloqueo es mediante un **gráfico de espera**. En este gráfico se crea un nodo por cada transacción que actualmente se está ejecutando. Siempre que una transacción $T_i$ está esperando a bloquear un elemento $X$ que actualmente está bloqueado por una transacción $T_j$, en el gráfico se crea un arco tendente ($T_i \rightarrow T_j$). Cuando $T_j$ libera el o los bloqueos sobre los elementos por los que $T_i$ está esperando, el arco tendente desaparece del gráfico de espera. Tenemos un estado de interbloqueo si, y sólo si, el gráfico de espera tiene un ciclo. Un problema de esta metodología es la cuestión de determinar *cuándo* el sistema debe comprobar si hay un interbloqueo. Pueden utilizarse criterios como el número de transacciones actualmente en ejecución o el lapso de tiempo que varias transacciones han estado esperando por los elementos bloqueados. La Figura 18.5(b) muestra el gráfico de espera para la planificación (parcial) de la Figura 18.5(a). Si el sistema se encuentra en estado de interbloqueo, algunas de las transacciones que lo provocan deben abortarse. La elección de cuáles hay que abortar se conoce como **selección de la víctima**. El algoritmo de selección de la víctima debe evitar generalmente la selección de transacciones que han estado ejecutándose durante mucho tiempo y que han realizado muchas actualizaciones, por lo que debe elegir transacciones que no han introducido muchos cambios.

**Tiempos limitados.** Otra solución más sencilla es el uso de tiempos limitados, un método práctico por su baja sobrecarga y simplicidad. En este método, si una transacción lleva en espera un tiempo superior a un tiempo definido por el sistema, éste asume que la transacción puede estar bloqueada y procede a su eliminación (independientemente de que exista o no un interbloqueo).

**Inanición (espera indefinida).** Otro problema que puede surgir cuando se utiliza el bloqueo es la inanición, que tiene lugar cuando una transacción no puede continuar durante un periodo de tiempo indeterminado mientras otras transacciones del sistema continúan con normalidad. Esto puede ocurrir si el esquema de espera para los elementos bloqueados es injusto, dando prioridad a algunas transacciones sobre otras. Una solución es tener un esquema de espera justo, como el uso de una cola del tipo **primero en llegar, primero en ser servido**; las transacciones pueden bloquear un elemento en el orden en que solicitaron originalmente el bloqueo. Otro esquema permite que algunas transacciones tengan prioridad sobre otras, pero aumenta la prioridad de una transacción que más tiempo lleva esperando, hasta que finalmente obtiene la prioridad más alta y procede. La inanición también puede darse debido a la selección de víctima, si el algoritmo selecciona repetidamente la misma transacción como víctima, lo que provoca su aborto y que nunca termine la ejecución. El algoritmo puede utilizar prioridades más altas para las transacciones que se han abortado varias veces para evitar este problema. Los esquemas esperar-morir y herir-esperar explicados anteriormente evitan la inanición.

# 18.2 Control de la concurrencia basado en la ordenación de marcas de tiempo

El uso de bloqueos, en combinación con el protocolo 2PL, garantiza la serialización de las planificaciones. Las planificaciones serializables producidas por 2PL tienen sus planificaciones en serie equivalentes basadas en el orden en que las transacciones en ejecución bloquean los elementos que adquieren. Si una transacción necesita un elemento que ya está bloqueado, puede verse obligada a esperar hasta que el elemento sea liberado. Una metodología diferente que garantiza la serialización implica el uso de marcas de tiempo para ordenar la ejecución de las transacciones para una planificación en serie equivalente. En la Sección 18.2.1 explicamos las marcas de tiempo y en la Sección 18.2.2 veremos cómo se implementa la serialización ordenando las transacciones en base a sus marcas de tiempo.

## 18.2.1 Marcas de tiempo

Una **marca de tiempo** es un identificador único creado por el DBMS para identificar una transacción. Normalmente, los valores de las marcas de tiempo son asignados en el orden en el que las transacciones son enviadas al sistema, por lo que una marca de tiempo puede verse como el *tiempo de inicio de la transacción*. Nos referiremos a la marca de tiempo de la transacción $T$ como **TS($T$)**. Las técnicas de control de la concurrencia basadas en la ordenación de las marcas de tiempo no utilizan bloqueos; por tanto, *no se pueden producir interbloqueos*.

Las marcas de tiempo se pueden generar de varias formas. Una posibilidad es utilizar un contador que se incremente cada vez que su valor es asignado a una transacción. Las marcas de tiempo se numeran como 1, 2, 3, ... en este esquema. Un contador tiene un valor máximo finito, por lo que el sistema debe reiniciar periódicamente el contador a cero cuando no se ejecutan transacciones durante un corto periodo de tiempo. Otra forma de implementar las marcas de tiempo es utilizar el valor de fecha/hora actual del reloj del sistema y asegurarse de que no se han generado dos valores de marca de tiempo durante el mismo tictac del reloj.

## 18.2.2 Algoritmo de ordenación de marcas de tiempo

La idea para este esquema es ordenar las transacciones según sus marcas de tiempo. Una planificación en la que participan las marcas de tiempo es serializable, y la planificación en serie equivalente tendrá las transacciones ordenadas según sus marcas de tiempo. Es lo que se conoce como **ordenación por marcas de tiempo** (**TO**, *timestamp ordering*). Esto difiere de 2PL, en el que una planificación es serializable siendo equivalente a alguna planificación en serie permitida por los protocolos de bloqueo. No obstante, en la ordenación por marcas de tiempo, la planificación es equivalente a un *orden en serie particular* correspondiente al orden de las marcas de tiempo de las transacciones. El algoritmo debe garantizar que, para cada elemento accedido por *operaciones en conflicto* de la planificación, el orden con el que se accede a ese elemento no viola el orden de serialización. Para ello, el algoritmo asocia a cada elemento $X$ de la base de datos dos valores de marca de tiempo (**TS**):

1. **marca_lectura($X$).** Es la **marca de tiempo de lectura** del elemento $X$; es la marca de tiempo más grande entre todas las marcas de tiempo de las transacciones que han leído satisfactoriamente el elemento $X$ (es decir, marca_lectura($X$) = TS($T$), donde $T$ es la transacción *más nueva* que ha leído $X$ con éxito).
2. **marca_escritura($X$).** Es la **marca de tiempo de escritura** del elemento $X$; es la marca de tiempo más grande entre todas las marcas de tiempo de las transacciones que han escrito el elemento $X$ satisfactoriamente (es decir, marca_escritura($X$) = TS($T$), donde $T$ es la transacción *más nueva* que ha escrito satisfactoriamente el elemento $X$).

**Ordenación de marcas de tiempo (TO, *Timestamp Ordering*) básica.** Siempre que alguna transacción $T$ intenta emitir una operación leer_elemento($X$) o escribir_elemento($X$), el algoritmo **TO básico** compara la marca de tiempo de $T$ con marca_lectura($X$) y marca_escritura($X$) para garantizar que no se viola el orden de las marcas de tiempo de la transacción en ejecución. Si se viola este orden, la transacción $T$ se cancela y se vuelve a enviar al sistema como una transacción nueva con una *marca de tiempo nueva*. Si $T$ es abortada y anulada, también debe anularse cualquier transacción $T_1$ que pueda haber utilizado un valor escrito por $T$. De forma parecida, cualquier transacción $T_2$ que pueda haber utilizado un valor escrito por $T_1$ también debe anularse, y así sucesivamente. Este efecto se conoce como **anulación en cascada** y es uno de los problemas asociados con la TO básica, ya que no se garantiza que las planificaciones producidas sean recuperables. Debe implementarse un *protocolo adicional* para garantizar que las planificaciones son recuperables, evitan la anulación en cascada o son estrictas. Primero vamos a describir el algoritmo de la TO básica. El algoritmo de control de la concurrencia debe comprobar si las operaciones en conflicto violan la ordenación de las marcas de tiempo en estos dos casos:

1. Cuando la transaction $T$ emite una operación escribir_elemento($X$):
   a. Si marca_lectura($X$) > TS($T$) o si marca_escritura($X$) > TS($T$), abortar y anular $T$ y rechazar la operación. Debe hacerse esto porque alguna transacción *más nueva* con una marca de tiempo mayor que TS($T$) (y, por tanto, *posterior* a $T$ en el orden de las marcas de tiempo) ya ha leído o escrito el valor del elemento $X$ antes de que $T$ tuviera ocasión de escribir $X$, violándose así la ordenación de las marcas de tiempo.
   b. Si la condición del apartado (a) no se cumple, entonces se ejecuta la operación escribir_elemento($X$) de $T$ y se establece marca_escritura($X$) a TS($T$).
2. Cuando la transacción $T$ emite una operación leer_elemento($X$):
   a. Si marca_escritura($X$) > TS($T$), entonces se aborta y anula la transacción $T$ y se rechaza la operación. Debe hacerse esto porque alguna transacción *más nueva* con una marca de tiempo mayor que TS($T$) (y, por tanto, *posterior* a $T$ en el orden de las marcas de tiempo) ya ha escrito el valor del elemento $X$ antes de que $T$ tuviera la oportunidad de leer $X$.
   b. Si marca_escritura($X$) = TS($T$), entonces se ejecuta la operación leer_elemento($X$) de $T$ y se asigna a marca_lectura($X$) el valor *mayor* entre TS($T$) y la marca marca_lectura($X$) actual.

Por tanto, siempre que el algoritmo de TO básica detecta dos *operaciones en conflicto* que suceden en el orden incorrecto, rechaza la más tardía de las dos abortando la transacción que la emitió. Las planificaciones producidas por la TO básica son, por tanto, *serializables por conflicto*, como el protocolo 2PL. Sin embargo, algunas planificaciones que son posibles bajo un protocolo, no están permitidas bajo el otro. Por tanto, *ningún* protocolo permite *todas las posibles* planificaciones serializables. Como mencionamos anteriormente, con la ordenación de las marcas de tiempo no se produce el interbloqueo. Sin embargo, puede darse un reinicio cíclico (y, por tanto, la inanición) si una transacción es continuamente abortada y reiniciada.

**Ordenación de marcas de tiempo (TO, *Timestamp Ordering*) estricta.** Es una variante de la TO básica que garantiza que las planificaciones son tanto **estrictas** (para una recuperabilidad fácil) como serializables (por conflicto). En esta variación, una transacción $T$ que emite una operación leer_elemento($X$) o escribir_elemento($X$) de modo que TS($T$) > marca_escritura(X) ve *retardada* su operación de lectura o escritura hasta que la transacción $T'$ que *escribió* el valor de $X$ [por tanto, TS($T'$) = marca_escritura($X$)] es confirmada o abortada. Para implementar este algoritmo, es necesario simular el bloqueo de un elemento $X$ que ha sido escrito por la transacción $T'$ hasta que $T'$ es confirmada o abortada. Este algoritmo *no provoca el interbloqueo*, puesto que $T$ espera por $T'$ sólo si TS($T$) > TS($T'$).

**Regla de escritura de Thomas.** Es una modificación del algoritmo de TO básica, que no implementa la serialización por conflicto; pero rechaza menos operaciones de escritura, al modificar las comprobaciones de la operación escribir_elemento($X$) de este modo:

1. Si marca_lectura($X$) > TS($T$), entonces se aborta y anula $T$ y se rechaza la operación.
2. Si marca_escritura($X$) > TS($T$), entonces no se ejecuta la operación de escritura pero continúa el procesamiento. Esto es debido a que alguna transacción con una marca de tiempo mayor que TS($T$) (y, por tanto, posterior a $T$ en la ordenación de las marcas de tiempo) ya ha escrito el valor de $X$. Por consiguiente, debemos ignorar la operación escribir_elemento($X$) de $T$ porque ya está anticuada y obsoleta. Cualquier conflicto que surja de esta situación sería detectado por el caso (1).
3. Si no se cumple ninguna de las condiciones anteriores, se ejecuta la operación escribir_elemento($X$) de $T$ y se asigna a marca_escritura($X$) el valor de TS($T$).

# 18.3 Técnicas multiversión para controlar la concurrencia

Otros protocolos de control de la concurrencia se basan en conservar los valores antiguos de un elemento de datos cuando es actualizado. Es lo que se conoce como control multiversión de la concurrencia, porque se conservan varias versiones (valores) de un elemento. Cuando una transacción necesita acceder a un elemento, se elige una versión *adecuada* para mantener la serialización de la planificación actualmente en ejecución, si es posible. La idea es que algunas operaciones de lectura que serían rechazadas por otras técnicas, pueden ser aceptadas si *leen una versión más antigua* del elemento para mantener la serialización. Cuando una transacción escribe un elemento, escribe una *versión nueva* y se conserva la versión antigua. Algunos algoritmos de control de la concurrencia multiversión utilizan la serialización por vista, en lugar de la serialización por conflicto.

Un inconveniente obvio de las técnicas multiversión es que se necesita más almacenamiento para conservar las distintas versiones de los elementos de la base de datos. No obstante, es posible tener que conservar de todos modos las versiones más antiguas (por ejemplo, con fines de recuperación). Además, algunas aplicaciones de bases de datos requieren versiones más antiguas a fin de conservar un histórico de la evolución de los valores de los elementos de datos. El caso extremo es una *base de datos temporal* (consulte el Capítulo 24), que hace un seguimiento de todos los cambios y de las horas en las que se produjeron. En estos casos, no hay ninguna penalización por almacenamiento adicional para las técnicas multiversión, puesto que las versiones antiguas ya están guardadas.

Se han propuesto algunos esquemas de control multiversión de la concurrencia. Aquí vamos a ver dos esquemas, uno basado en la ordenación de las marcas de tiempo y otro basado en 2PL.

## 18.3.1 Técnica multiversión basada en la ordenación de las marcas de tiempo

En este método, el sistema guarda varias versiones $X_1$, $X_2$, ..., $X_k$ de cada elemento de datos $X$. Por *cada versión*, se conservan el valor de versión $X_i$ y estas dos marcas de tiempo:

1. marca_lectura($X_i$). La marca de tiempo de lectura de $X_i$ es el valor más grande de las marcas de tiempo de todas las transacciones que han leído satisfactoriamente la versión $X_i$.
2. marca_escritura($X_i$). La marca de tiempo de escritura de $X_i$ es la marca de tiempo de la transacción que escribió el valor de la versión $X_i$.

Siempre que una transacción $T$ tiene permitido ejecutar una operación escribir_elemento($X$), se crea una nueva versión, $X_{k+1}$, del elemento $X$, con los valores de marca_escritura($X_{k+1}$) y marca_lectura($X_{k+1}$) establecidos a TS($T$). Según el caso, cuando una transacción $T$ tiene permitido leer el valor de la versión $X_i$, a marca_lectura($X_i$) se le asigna el valor más grande entre la marca_lectura($X_i$) actual y TS($T$).

Para garantizar la serialización, se utilizan estas reglas:

1. Si la transacción $T$ emite una operación escribir_elemento($X$), y la versión $i$ de $X$ tiene el valor marca_escritura($X_i$) más alto de todas las versiones de $X$ que también es *menor o igual que* TS($T$), y marca_lectura($X_i$) > TS($T$), entonces se aborta y anula la transacción $T$; en caso contrario, se crea una nueva versión, $X_j$, de $X$ con marca_lectura($X_j$) = marca_escritura($X_j$) = TS($T$).
2. Si la transacción $T$ emite una operación leer_elemento($X$), se busca la versión $i$ de $X$ que tenga la marca_escritura($X_i$) más grande de todas las versiones de $X$ pero que sea *menor o igual que* TS($T$); entonces, se devuelve el valor de $X_i$ a la transacción $T$, y se establece el valor de marca_lectura($X_i$) al valor más grande entre TS($T$) y la marca_lectura($X_i$) actual.

Como vimos en el caso 2, una operación leer_elemento($X$) siempre es satisfactoria, ya que encuentra la versión apropiada $X_i$ que tiene que leer basándose en el valor de escribir_TS de las distintas versiones existentes de $X$. En el caso 1, no obstante, la transacción $T$ puede abortarse y anularse. Esto sucede si $T$ intenta escribir una versión de $X$ que podría haber leído otra transacción $T'$ cuya marca de tiempo es marca_lectura($X_i$); sin embargo, $T'$ ya ha leído la versión $X_i$, que fue escrita por la transacción con una marca de tiempo igual a marca_escritura($X_i$). Si surge este conflicto, se anula $T$; en caso contrario, se crea una nueva versión de $X$, escrita por la transacción $T$. Si se anula $T$, puede producirse una anulación en cascada. Por tanto, para garantizar la recuperabilidad, una transacción $T$ no debe poder confirmarse hasta que se hayan confirmado todas las transacciones que hayan escrito alguna versión leída por $T$.

### 18.3.2 Bloqueo en dos fases multiversión utilizando bloqueos de certificación

En este esquema de bloqueo multimodo hay *tres modos de bloqueo* para un elemento: lectura, escritura y certificación, en lugar de los dos modos (lectura y escritura) explicados anteriormente. Por tanto, el estado de BLOQUEAR($X$) para un elemento $X$ puede ser: bloqueado para lectura, bloqueado para escritura, bloqueado para certificación o desbloqueado. En el esquema de bloqueo estándar con los bloqueos de lectura y escritura (consulte la Sección 18.1.1), un bloqueo de escritura es un bloqueo exclusivo. Podemos describir la relación entre los bloqueos de lectura y escritura del esquema estándar mediante la **tabla de compatibilidad de bloqueos** de la Figura 18.6(a). Una entrada "Sí" significa que si una transacción $T$ posee el tipo de bloqueo especificado en la cabecera de la columna del elemento $X$ y si la transacción $T'$ solicita el tipo de bloqueo especificado en la cabecera de fila del mismo elemento $X$, entonces $T'$ *puede obtener el bloqueo* porque los modos de bloqueo son compatibles. Por el contrario, una entrada "No" en la tabla indica que los bloqueos son incompatibles, por lo que $T'$ *debe esperar* hasta que $T$ *libere* el bloqueo.

En el esquema de bloqueo estándar, una vez que una transacción obtiene un bloqueo de escritura sobre un elemento, ninguna otra transacción puede acceder a ese elemento. La idea tras un 2PL multiversión es permitir que otras transacciones $T'$ puedan leer un elemento $X$ mientras una transacción $T$ posee un bloqueo de escritura sobre $X$. Esto se consigue manteniendo *dos versiones* de cada elemento $X$; una de ellas deberá haber sido escrita por alguna transacción confirmada. La segunda versión $X'$ se crea cuando una transacción $T$ adquiere un bloqueo de escritura sobre el elemento. Otras transacciones pueden continuar leyendo la *versión confirmada* de $X$ mientras $T$ conserva el bloqueo de escritura. La transacción $T$ puede escribir el valor de $X'$ cuando lo necesite, sin que ello afecte al valor de la versión confirmada de $X$. Sin embargo, un vez que $T$ ya está confirmada, debe obtener un **bloqueo de certificación** sobre todos los elementos sobre los que actualmente tiene bloqueos de escritura antes de poder confirmarse. El bloqueo de certificación no es compatible con los bloqueos de lectura, por lo que la transacción puede que tenga que retrasar su confirmación hasta que todos sus elementos bloqueados para escritura sean liberados por cualquier transacción de lectura a fin de obtener los bloqueos de certificación. Una vez adquiridos estos bloqueos (que son bloqueos exclusivos), la versión confirmada de $X$ del elemento de datos se establece al valor de la versión $X'$, la versión $X'$ es descartada y los bloqueos de certificación son liberados. La tabla de compatibilidad de bloqueos para este esquema se muestra en la Figura 18.6(b).

**Figura 18.6.** Tablas de compatibilidad de bloqueos. (a) Tabla de compatibilidad para el esquema de bloqueo de lectura/escritura. (b) Tabla de compatibilidad para el esquema de bloqueo de lectura/escritura/certificación.

**(a)**

| | Lectura | Escritura |
|---|---|---|
| Lectura | Sí | No |
| Escritura | No | No |

**(b)**

| | Lectura | Escritura | Certificación |
|---|---|---|---|
| Lectura | Sí | Sí | No |
| Escritura | Sí | No | No |
| Certificación | No | No | No |

En este esquema 2PL multiversión, se pueden realizar varias operaciones de lectura simultáneamente a una operación de escritura (algo que los esquemas 2PL convencionales no permiten). Sin embargo, una transacción deberá esperar a confirmarse hasta obtener los bloqueos de certificación exclusivos de *todos los elementos* que ha modificado. Este esquema evita los abortos en cascada, ya que las transacciones sólo leen la versión de $X$ que fue escrita por una transacción confirmada. Sin embargo, si permitimos que un bloqueo de lectura se convierta en un bloqueo de escritura, se puede producir un interbloqueo, que debería tratarse con las variaciones de las técnicas que vimos en la Sección 18.1.3.

## 18.4 Técnicas de control de la concurrencia optimistas (validación)

En todas las técnicas de control de la concurrencia que hemos visto hasta ahora, se lleva a cabo cierto grado de comprobación *antes* de que pueda ejecutarse una operación de base de datos. Por ejemplo, en el bloqueo se realiza una comprobación para determinar si el elemento al que se va a acceder está bloqueado. En la ordenación de las marcas de tiempo, se compara la marca de tiempo de la transacción con las marcas de tiempo de lectura y escritura del elemento. Dicha comprobación representa un coste durante la ejecución de la transacción, que supone ralentizar las transacciones.

En las **técnicas de control de la concurrencia optimistas**, también conocidas como **técnicas de validación** o **certificación**, no se realiza comprobación alguna mientras la transacción se está ejecutando. Varios de los métodos de control de la concurrencia utilizan la técnica de la validación. Sólo describiremos un esquema, en el que las actualizaciones en la transacción no se aplican directamente a los elementos de la base de datos hasta que la transacción alcanza su final. Durante la ejecución de la transacción, todas las actualizaciones se aplican a *copias locales* de los elementos de datos que se conservan para la transacción.[6] Al término de la ejecución de la transacción, una **fase de validación** comprueba si alguna de las actualizaciones de la transacción viola la serialización. El sistema debe conservar cierta información necesaria para la fase de validación. Si la serialización no se viola, la transacción es confirmada y se actualiza la base de datos a partir de las copias locales; en caso contrario, la transacción es abortada y reiniciada posteriormente.

[6] ¡Esto puede asemejarse al almacenamiento de varias versiones de los elementos!

Hay tres fases para este protocolo de control de la concurrencia:

1. **Fase de lectura.** Una transacción puede leer valores de los elementos de datos confirmados de la base de datos. Sin embargo, las actualizaciones sólo se aplican a las copias locales (versiones) de los elementos de datos conservadas en el espacio de trabajo de la transacción.
2. **Fase de validación.** La comprobación se realiza para garantizar que la serialización no será violada si las actualizaciones de la transacción se aplican a la base de datos.
3. **Fase de escritura.** Si la fase de validación es satisfactoria, las actualizaciones de la transacción son aplicadas a la base de datos; en caso contrario, las actualizaciones se descartan y se reinicia la transacción.

La idea tras el control optimista de la concurrencia es hacer todas las comprobaciones de inmediato; por tanto, la ejecución de la transacción procede con un coste mínimo hasta alcanzar la fase de validación. Si hay una ligera interferencia entre las transacciones, la mayoría serán validadas satisfactoriamente. Sin embargo, si la interferencia es demasiada, muchas transacciones que se ejecutan hasta su terminación verán descartados sus resultados y deberán reiniciarse más tarde. Bajo estas circunstancias, las técnicas optimistas no funcionan bien. Se llaman así porque asumen que se producirá una pequeña interferencia y, por tanto, que no hay necesidad de hacer una comprobación durante la ejecución de la transacción.

El protocolo optimista que describimos utiliza las marcas de tiempo de las transacciones y requiere que el sistema mantenga el control de los conjuntos de escritura y lectura de las transacciones. Además, de cada transacción hay que guardar las horas de *inicio* y *terminación* de alguna de las tres fases. Recuerde que el conjunto de escritura de una transacción es el conjunto de elementos que escribe, y el conjunto de lectura es el conjunto de elementos que lee. En la fase de validación de la transacción $T_i$, el protocolo comprueba que $T_i$ no interfiere con ninguna transacción confirmada ni con otras transacciones que actualmente estén en su fase de validación. La fase de validación para $T_i$ comprueba que, para *cada* transacción $T_j$ que se confirma o está en su fase de validación, se cumple *una* de las siguientes condiciones:

1. La transacción $T_j$ completa su fase de escritura antes de que $T_i$ inicie su fase de lectura.
2. $T_i$ inicia su fase de escritura después de que $T_j$ complete su fase de escritura, y el conjunto de lectura de $T_i$ no tiene elementos en común con el conjunto de escritura de $T_j$.
3. Ni el conjunto de lectura ni el conjunto de escritura de $T_i$ tienen elementos en común con el conjunto de escritura de $T_j$, y $T_j$ completa su fase de lectura antes de que $T_i$ complete su fase de lectura.

Al validar la transacción $T_i$, primero se comprueba la primera condición para cada transacción $T_j$, puesto que la condición (1) es la más sencilla de comprobar. Sólo si la condición (1) es falsa se comprueba la condición (2), y sólo si la condición (2) es falsa se comprueba la condición (3) (la más compleja de evaluar). Si se cumple cualquiera de estas tres condiciones, no hay interferencia y $T_i$ se valida satisfactoriamente. Si *ninguna* de estas tres condiciones se cumple, la validación de la transacción $T_i$ falla y es abortada y reiniciada más tarde debido a que puede haberse producido alguna interferencia.

## 18.5 Granularidad de los elementos de datos y bloqueo de la granularidad múltiple

Todas las técnicas de control de la concurrencia asumen que la base de datos está formada por un cierto número de elementos de datos. Un elemento de base de datos puede ser alguna de estas cosas:

- Un registro de la base de datos.
- El valor de un campo de un registro de la base de datos.
- Un bloque de disco.

- Un fichero entero.
- La base de datos entera.

La granularidad puede afectar al rendimiento del control de la concurrencia y de la recuperación. En la Sección 18.5.1 explicamos algunas de las contrapartidas con respecto a elegir el nivel de granularidad utilizado para el bloqueo, mientras que en la Sección 18.5.2 explicamos un esquema de bloqueo de granularidad múltiple, en el que es posible cambiar dinámicamente el nivel de granularidad (tamaño del elemento de datos).

### 18.5.1 Consideraciones sobre el nivel de granularidad para el bloqueo

El tamaño de los elementos de datos se denomina con frecuencia **granularidad del elemento de datos**. La *granularidad fina* se refiere a los tamaños de elemento pequeños, mientras que la *granularidad gruesa* se refiere a los tamaños de elemento más grandes. Al elegir el tamaño del elemento de datos hay que tener en consideración varias contrapartidas. Explicaremos el tamaño del elemento de datos en el contexto del bloqueo, aunque pueden desarrollarse unos argumentos parecidos para otras técnicas de control de la concurrencia.

En primer lugar, observe que cuanto mayor es el tamaño del elemento de datos, más bajo es el grado de concurrencia permitido. Por ejemplo, si el tamaño del elemento de datos es un bloque de disco, una transacción $T$ que necesita bloquear un registro $B$ debe bloquear el bloque de disco entero $X$ que contiene $B$ porque un bloqueo está asociado con el elemento de datos entero (bloque). Ahora, si otra transacción $S$ quiere bloquear un registro $C$ diferente que resulta residir en el mismo bloque $X$ en un modo de bloqueo en conflicto, se ve obligada a esperar. Si el tamaño del elemento de datos fuera un registro, la transacción $S$ podría proceder, porque estaría bloqueando un elemento de datos diferente (registro).

Por otro lado, cuanto más pequeño es el tamaño del elemento de datos, mayor número de elementos hay en la base de datos. Como cada elemento está asociado con un bloqueo, el sistema tendrá una cantidad mayor de bloqueos activos que el gestor de bloqueos deberá controlar. Se realizarán más operaciones de bloqueo y desbloqueo, lo que provocará una sobrecarga más alta. Además, se necesitará más espacio de almacenamiento para la tabla de bloqueo. En el caso de las marcas de tiempo, se necesita almacenamiento para las marcas de lectura y de escritura de cada elemento de datos, y habrá una sobrecarga parecida para manipular una gran cantidad de elementos.

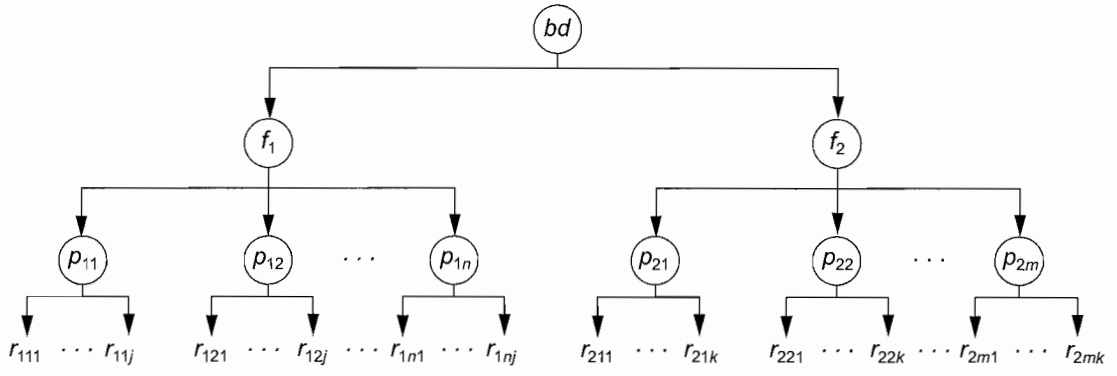
Dado lo anterior, una cuestión obvia es la siguiente: ¿cuál es el mejor tamaño de elemento? La respuesta es que *depende de los tipos de transacciones implicadas*. Si una transacción típica accede a pocos registros, lo mejor es tener una granularidad equivalente a un registro. Por otro lado, si una transacción accede normalmente a muchos registros del mismo fichero, lo mejor es tener una granularidad equivalente a un bloque o un fichero, de modo que la transacción considerará todos los registros como uno (o unos cuantos) elementos de datos.

### 18.5.2 Bloqueo por nivel de granularidad múltiple

Ya que el mejor tamaño de granularidad depende de la transacción dada, parece adecuado que un sistema de bases de datos deba soportar varios niveles de granularidad, donde el nivel de granularidad puede ser diferente para distintas combinaciones de transacciones. La Figura 18.7 muestra una sencilla jerarquía de granularidad con una base de datos que contiene dos ficheros, cada uno de ellos con varias páginas de disco, y cada página con varios registros. Esto puede utilizarse para ilustrar un protocolo 2PL de **nivel de granularidad múltiple**, donde es posible que se solicite un bloqueo a cualquier nivel. Sin embargo, se necesitarán tipos de bloqueos adicionales para soportar eficazmente un protocolo semejante.

Considere el siguiente escenario, únicamente con los tipos de bloqueo compartido y exclusivo, que se refiere al ejemplo de la Figura 18.7. Suponga que la transacción $T_1$ quiere actualizar *todos los registros* del fichero

**Figura 18.7.** Jerarquía de granularidad para ilustrar el bloqueo por nivel de granularidad múltiple.




$f_1$, y $T_1$ solicita y se le concede un bloqueo exclusivo para $f_1$. Después, todas las páginas de $f_1$ ($p_{11}$ a $p_{1n}$) (y los registros contenidos en esas páginas) son bloqueadas en modo exclusivo. Esto es beneficioso para $T_1$ porque la configuración de un solo bloqueo a nivel de fichero es más eficaz que la configuración de $n$ bloqueos a nivel de página o tener que bloquear independientemente cada registro. Ahora, piense en otra transacción $T_2$ que sólo quiere leer el registro $r_{1nj}$ de la página $p_{1n}$ del fichero $f_1$; entonces $T_2$ solicitaría un bloqueo a nivel de registro compartido sobre $r_{1nj}$. Sin embargo, el sistema de bases de datos (es decir, el gestor de transacciones o, más concretamente, el gestor de bloqueos) debe verificar la compatibilidad del bloqueo solicitado con los bloqueos ya existentes. Una forma de verificar esto es atravesando el árbol desde la hoja $r_{1nj}$ hasta $p_{1n}$ hasta $f_1$ hasta *bd*. Si en cualquier momento surge un bloqueo conflictivo en cualquiera de estos elementos, entonces la solicitud de bloqueo para $r_{1nj}$ es denegada y $T_2$ es bloqueada y deberá esperar. Esta travesía sería bastante eficaz.

Sin embargo, ¿qué pasaría si la solicitud de la transacción $T_2$ llegara *antes* que la solicitud de la transacción $T_1$? En este caso, el bloqueo de registro compartido es concedido a $T_2$ para $r_{1nj}$, pero cuando es solicitado el bloqueo a nivel de fichero de $T_1$, al gestor de bloqueos le resulta muy difícil comprobar todos los nodos (páginas y registros) que son descendientes del nodo $f_1$ en busca de un conflicto por bloqueo. Esto sería muy ineficaz y acabaría con el propósito de tener bloqueos de nivel de granularidad múltiple. Para que resulte práctico el bloqueo por nivel de granularidad múltiple, son necesarios otros tipos de bloqueos, denominados **bloqueos de intención**. La idea es que una transacción indique, junto con la ruta desde la raíz hasta el nodo deseado, el tipo de bloqueo (compartido o exclusivo) que requerirá de uno de los descendientes del nodo. Hay tres tipos de bloqueos de intención:

1. Bloqueo de intención compartida (IS) indica que se solicitará un(os) bloqueo(s) compartido(s) en algún(os) (de los) nodo(s) descendiente(s).
2. Bloqueo de intención exclusiva (IX) indica que se solicitará un(os) bloqueo(s) exclusivo(s) en algún(os) (de los) nodo(s) descendiente(s).
3. Bloqueo compartido con intención exclusiva (SIX) indica que el nodo actual está bloqueado en modo compartido pero se solicitará un(os) bloqueo(s) exclusivo(s) en algún(os) (de los) nodo(s) descendiente(s).

La tabla de compatibilidad de los tres bloqueos de intención, y los bloqueos compartido y exclusivo, se muestran en la Figura 18.8. Además de la introducción de los tres tipos de bloqueos de intención, debe utilizarse un protocolo de bloqueo adecuado. El protocolo de **bloqueo por nivel de granularidad múltiple** (**MGL**, ***multiple granularity locking***) consiste en las siguientes reglas:

1. Debe adherirse a la compatibilidad de bloqueo (basándose en la Figura 18.8).
2. La raíz del árbol debe bloquearse primero, en cualquier modo.

**Figura 18.8.** Matriz de compatibilidad de bloqueos para el bloqueo de granularidad múltiple.

| | IS | IX | S | SIX | X |
|---|---|---|---|---|---|
| IS | Sí | Sí | Sí | Sí | No |
| IX | Sí | Sí | No | No | No |
| S | Sí | No | Sí | No | No |
| SIX | Sí | No | No | No | No |
| X | No | No | No | No | No |

3. Un nodo $N$ puede ser bloqueado por una transacción $T$ en modo S o IS sólo si el nodo padre de $N$ ya está bloqueado por la transacción $T$ en modo IS o IX.
4. Un nodo $N$ puede ser bloqueado por una transacción $T$ en modo X, IX o SIX sólo si el padre del nodo $N$ ya está bloqueado por la transacción $T$ en modo IX o SIX.
5. Una transacción $T$ puede bloquear un nodo sólo si no ha desbloqueado ningún nodo (para implementar el protocolo 2PL).
6. Una transacción $T$ puede desbloquear un nodo, $N$, sólo si ninguno de los hijos del nodo $N$ está actualmente bloqueado por $T$.

La regla 1 simplemente dice que no pueden concederse los bloqueos en conflicto. Las reglas 2, 3 y 4 determinan las condiciones para que una transacción pueda bloquear un nodo dado en cualquiera de los modos de bloqueo. Las reglas 5 y 6 del protocolo MGL implementan las reglas 2PL para producir planificaciones serializables. Para ilustrar el protocolo MGL con la jerarquía de base de datos de la Figura 18.7, considere estas tres transacciones:

1. $T_1$ quiere actualizar los registros $r_{111}$ y $r_{211}$.
2. $T_2$ quiere actualizar todos los registros de la página $p_{12}$.
3. $T_3$ quiere leer el registro $r_{11j}$ y el fichero $f_2$ entero.

La Figura 18.9 muestra una posible planificación serializable para estas tres transacciones. Sólo mostramos las operaciones de bloqueo y desbloqueo. Utilizamos la notación <tipo_bloqueo>(<elemento>) para mostrar las operaciones de bloqueo de la transacción.

El protocolo de nivel de granularidad múltiple es especialmente apropiado para procesar una mezcla de transacciones que incluya lo siguiente: (1) transacciones cortas que sólo acceden a unos pocos elementos (registros o campos) y (2) transacciones largas que acceden a ficheros enteros. En este entorno, con un protocolo semejante se incurre en menos bloqueos de transacciones y menos sobrecarga por bloqueo, en comparación con un método de bloqueo de granularidad de un solo nivel.

## 18.6 Uso de bloqueos para controlar la concurrencia en los índices

El bloqueo en dos fases también se puede aplicar a los índices (consulte el Capítulo 14), donde los nodos de un índice son equiparables a las páginas de disco. Sin embargo, la posesión de bloqueos en las páginas de índice hasta la fase de reducción de 2PL puede provocar una cantidad indebida de bloqueos de transacciones

**Figura 18.9.** Operaciones de bloqueo para ilustrar una planificación serializable.

| $T_1$ | $T_2$ | $T_3$ |
|---|---|---|
| IX(*bd*) | | |
| IX($f_1$) | | |
| | IX(*bd*) | |
| | | IS(*bd*) |
| | | IS($f_1$) |
| | | IS($p_{11}$) |
| IX($p_{11}$) | | |
| X($r_{111}$) | | |
| | IX($f_1$) | |
| | X($p_{12}$) | |
| | | S($r_{11j}$) |
| IX($f_2$) | | |
| IX($p_{21}$) | | |
| X($p_{211}$) | | |
| desbloquear($r_{211}$) | | |
| desbloquear($p_{21}$) | | |
| desbloquear($f_2$) | | |
| | | S($f_2$) |
| | desbloquear($p_{12}$) | |
| | desbloquear($f_1$) | |
| | desbloquear(*bd*) | |
| desbloquear($r_{111}$) | | |
| desbloquear($p_{11}$) | | |
| desbloquear($f_1$) | | |
| desbloquear(*bd*) | | |
| | | desbloquear($r_{11j}$) |
| | | desbloquear($p_{11}$) |
| | | desbloquear($f_1$) |
| | | desbloquear($f_2$) |
| | | desbloquear(*bd*) |

porque la búsqueda en un índice siempre *empieza* por la *raíz*. Por consiguiente, si una transacción quiere insertar un registro (operación de escritura), se bloqueará la raíz en modo exclusivo, para que todas las demás solicitudes de bloqueo en conflicto para el índice deban esperar hasta que la transacción entre en la fase de reducción. Esto bloquea el acceso al índice a todas las demás transacciones, por lo que en la práctica debemos utilizar otras metodologías para bloquear un índice.

La estructura de árbol del índice puede beneficiarse de cuando se desarrolla un esquema de control de la concurrencia. Por ejemplo, cuando se está ejecutando una búsqueda en un índice (operación de lectura), se recorre el árbol desde la raíz hasta una hoja. Una vez que se ha accedido a un nodo del nivel inferior, los nodos del nivel superior de esa ruta no se utilizarán de nuevo. Así, una vez obtenido un bloqueo de lectura sobre un nodo hijo, se puede liberar el bloqueo sobre el padre. Cuando se está aplicando una inserción a un nodo hoja (es decir, cuando se insertan una clave y un puntero), entonces debe bloquearse un nodo hoja específico en modo exclusivo. No obstante, si ese nodo no está lleno, la inserción no provocará cambios en los nodos de índice del nivel superior, lo que implica que no deben bloquearse exclusivamente.

Un método más conservador para las inserciones sería bloquear el nodo raíz en modo exclusivo y después acceder al nodo hijo apropiado de la raíz. Si el nodo hijo no está lleno, entonces el bloqueo sobre el nodo raíz

puede liberarse. Este método puede aplicarse hasta llegar a la hoja, lo que normalmente suponen tres o cuatro niveles desde la raíz. Aunque se posean los bloqueos exclusivos, se liberan pronto. Una alternativa, el **método más optimista**, sería solicitar y mantener *bloqueos compartidos* sobre los nodos que llevan hasta el nodo hoja, con un *bloqueo exclusivo* sobre esa hoja. Si la inserción provoca la división de la hoja, la inserción se propagará al o los nodos del nivel superior. Después, los bloqueos sobre el o los nodos del nivel superior pueden actualizarse al modo exclusivo.

Otro método para bloquear un índice es utilizar una variante del árbol $B^+$, denominada **árbol de enlace B**. En este tipo de árbol, los nodos hermanos del mismo nivel se enlazan en cada nivel. Esto permite el uso de bloqueos compartidos al solicitar una página, y requiere que el bloqueo sea liberado antes de acceder al nodo hijo. En una operación de inserción, el bloqueo compartido sobre un nodo se actualizaría al modo exclusivo. Si se produce una división, hay que volver a bloquear el nodo padre en modo exclusivo. Hay una complicación en las operaciones de búsqueda ejecutadas concurrentemente con la actualización. Suponga que una operación de actualización concurrente sigue la misma ruta que la búsqueda, e inserta una entrada nueva en el nodo hoja. Además, suponga que la inserción provoca la división del nodo hoja. Cuando la inserción se ha realizado, se reanuda el proceso de búsqueda, siguiendo el puntero hasta la hoja deseada, sólo para ver que la clave que se está buscando no está presente porque la división la ha movido a un nodo hoja nuevo, que sería el *hermano derecho* del nodo hoja original. Sin embargo, el proceso de búsqueda todavía puede tener éxito si sigue el puntero (enlace) desde el nodo hoja original hasta su hermano derecho, adonde se habrá movido la clave deseada.

La manipulación de la eliminación, donde se combinan dos o más nodos del árbol de índice, también forma parte del protocolo de concurrencia de árbol de enlace B. En este caso, se mantienen los bloqueos sobre los nodos que se van a combinar, así como un bloqueo sobre el padre de los dos nodos a combinar.

# 18.7 Otros problemas del control de la concurrencia

En esta sección veremos algunos problemas más relacionados con el control de la concurrencia. En la Sección 18.7.1 explicamos los problemas asociados con la inserción y la eliminación de registros y lo que se conoce como *problema del fantasma*, que puede surgir cuando se insertan registros. Este problema se describió en la Sección 17.6 como un problema potencial que requiere una medida de control de la concurrencia. En la Sección 18.7.2 explicamos los problemas que se pueden dar cuando una transacción visualiza algunos datos en el monitor antes de confirmarse, y resulta que después se cancela.

## 18.7.1 Inserción, eliminación y registros fantasma

Cuando se **inserta** un nuevo elemento de datos en la base de datos, es obvio que no es posible acceder a él hasta haberse creado el elemento y completado la operación de inserción. En un entorno de bloqueo, se puede crear un bloqueo para el elemento y establecer el modo exclusivo (escritura); el bloqueo puede liberarse al mismo tiempo que se liberan otros bloqueos de escritura, en base al protocolo de control de la concurrencia que se esté utilizando. En el caso de un protocolo basado en marcas de tiempo, las marcas de tiempo de lectura y escritura del elemento nuevo se establecen a la marca de tiempo de la transacción que se está creando.

A continuación, piense en la **operación de eliminación** aplicada a un elemento de datos existente. En los protocolos de bloqueo, de nuevo hay que obtener un bloqueo exclusivo (escritura) antes de que la transacción pueda eliminar el elemento. Para la ordenación de las marcas de tiempo, el protocolo debe garantizar que ninguna transacción posterior ha leído o escrito el elemento antes de permitir la eliminación del elemento.

La situación conocida como **problema del fantasma** se da cuando un registro nuevo que una transacción $T$ está insertando satisface una condición que un conjunto de registros accedido por otra transacción $T'$ debe

satisfacer. Por ejemplo, suponga que la transacción $T$ está insertando un registro EMPLEADO nuevo cuyo Dno = 5, mientras que la transacción $T'$ está accediendo a todos los registros EMPLEADO cuyo Dno = 5 (por ejemplo, para sumar todos los sueldos a fin de calcular el presupuesto en personal del departamento 5). Si el orden en serie equivalente es $T$ seguida por $T'$, entonces $T'$ debe leer el registro EMPLEADO nuevo e incluir su Sueldo en la suma. Si el orden es $T'$ seguida por $T$, no se incluiría el sueldo nuevo. Aunque las transacciones entran lógicamente en conflicto, en el último caso no hay realmente ningún registro (elemento de datos) en común entre las dos transacciones, puesto que $T'$ puede haber bloqueado todos los registros con Dno = 5 *antes* de que $T$ inserte el registro nuevo. Esto es porque el registro que provoca el conflicto es un **registro fantasma** que ha aparecido de repente en la base de datos en la que se ha insertado. Si otras operaciones con las dos transacciones entran en conflicto, es posible que el protocolo de control de la concurrencia no reconozca el conflicto debido al registro fantasma.

Una solución a este problema es utilizar el **bloqueo de índice**, que explicamos en la Sección 18.6. Como recordará del Capítulo 14, un índice incluye entradas que tienen el valor de un atributo, más un conjunto de punteros a todos los registros del fichero que tienen ese valor. Por ejemplo, un índice con el campo Dno de EMPLEADO incluiría una entrada para cada valor de Dno distinto, más un conjunto de punteros a todos los registros de EMPLEADO que tienen ese valor. Si la entrada de índice se bloquea antes de que el registro pueda ser accedido, entonces el conflicto con el registro fantasma puede detectarse, porque la transacción $T'$ solicitaría un bloqueo de lectura sobre la entrada de índice correspondiente a Dno = 5, y $T$ solicitaría un bloqueo de escritura sobre la misma entrada *antes* de que ellas colocaran los bloqueos sobre los registros actuales. Como los bloqueos de índice entrarían en conflicto, se detectaría el conflicto del fantasma.

Una técnica más genérica, denominada **bloqueo de predicado**, bloquearía de un modo parecido el acceso a todos los registros que satisfacen un *predicado* (condición) arbitrario; sin embargo, los bloqueos de predicado han demostrado ser difíciles de implementar con eficacia.

### 18.7.2 Transacciones interactivas

Otro problema surge cuando las transacciones interactivas leen la entrada y escriben la salida en un dispositivo interactivo, como el monitor, antes de ser confirmadas. El problema es que un usuario puede introducir un valor para un elemento de datos en una transacción $T$ basándose en el valor escrito en pantalla por la transacción $T'$, que puede que no esté confirmada. Esta dependencia entre $T$ y $T'$ no puede ser modelada por el método de control de la concurrencia del sistema, ya que sólo está basado en el usuario que interactúa con dos transacciones.

Una forma de tratar este problema consiste en posponer la salida de las transacciones a pantalla hasta que han sido confirmadas.

### 18.7.3 Cerrojos

Los bloqueos que se mantienen durante poco tiempo se conocen normalmente como **cerrojos**. Los cerrojos no obedecen el protocolo de control de la concurrencia usual, como el bloqueo en dos fases. Por ejemplo, puede utilizarse un cerrojo para garantizar la integridad física de una página cuando se está escribiendo ésta desde el búfer al disco; se adquiriría un cerrojo para la página, se escribiría la página en el disco y, por último, se liberaría el cerrojo.

## 18.8 Resumen

En este capítulo hemos explicado las técnicas DBMS para controlar la concurrencia. Empezamos con los protocolos basados en los bloqueos, que son, de lejos, los más utilizados en la práctica. Explicamos el protocolo de bloqueo en dos fases (2PL) y algunas de sus variantes: 2PL básico, 2PL estricto, 2PL conservador y 2PL

riguroso. Las variantes estricta y rigurosa son más comunes debido a sus mejores propiedades de recuperabilidad. Hicimos una introducción de los conceptos de bloqueos compartidos (lectura) y exclusivos (escritura), y vimos cómo el bloqueo puede garantizar la serialización al utilizarse en combinación con la regla de bloqueo en dos fases. También presentamos varias técnicas para tratar con el problema del interbloqueo, que puede surgir con los bloqueos. En la práctica, es normal utilizar la detección del interbloqueo (gráficos de espera) y tiempos limitados.

Presentamos otros protocolos de control de la concurrencia que no se utilizan tan a menudo en la práctica, pero que son importantes como alternativas teóricas que muestran una solución para este problema. Entre ellos encontramos el protocolo de ordenación de marcas de tiempo, que garantiza la serialización basada en la ordenación de las marcas de tiempo de las transacciones. Las marcas de tiempo son identificadores de transacción únicos generados por el sistema. Explicamos la regla de escritura de Thomas, que mejora el rendimiento pero no garantiza la serialización por conflicto. También hemos visto el protocolo de ordenación de marcas de tiempo estricto. Explicamos dos protocolos multiversión, que asumen que en la base de datos se pueden guardar las versiones más antiguas de los elementos de datos. Una técnica, denominada bloqueo en dos fases multiversión (que se ha utilizado en la práctica), asume que pueden existir dos versiones de un elemento e intenta aumentar la concurrencia haciendo compatibles los bloqueos de escritura y lectura (a costa de introducir un modo de bloqueo de certificación adicional). También hemos presentado un protocolo multiversión basado en la ordenación de las marcas de tiempo, y un ejemplo de protocolo optimista, que también se conoce como protocolo de certificación o validación.

Después, centramos nuestra atención en la granularidad de los elementos de datos. Describimos un protocolo de bloqueo multigranularidad que permite modificar la granularidad (tamaño del elemento) en base a la mezcla de transacciones del momento, con el objetivo de mejorar el rendimiento del control de la concurrencia. Asimismo, hemos hablado del problema práctico que supone desarrollar protocolos de boqueo para los índices de modo que éstos no se conviertan en un estorbo para el acceso concurrente. Por último, hablamos del problema del fantasma y de los problemas relacionados con las transacciones interactivas, y describimos brevemente el concepto de cerrojos y de sus diferencias con los bloqueos.

El siguiente capítulo está dedicado a las técnicas de recuperación.

## Preguntas de repaso

**18.1.** ¿Qué es el protocolo de bloqueo en dos fases? ¿Cómo garantiza la serialización?

**18.2.** Detalle algunas de las variaciones del protocolo de bloqueo en dos fases. ¿Por qué es preferible utilizar casi siempre el bloqueo en dos fases estricto o riguroso?

**18.3.** Explique los problemas del interbloqueo y la espera indefinida, y los diferentes métodos de tratar con ellos.

**18.4.** Compare los bloqueos binarios con los bloqueos exclusivos/compartidos. ¿Por qué son preferibles estos últimos?

**18.5.** Describa los protocolos esperar-morir y herir-esperar para prevenir el interbloqueo.

**18.6.** Describa los protocolos de espera cautelosa, no espera y tiempo limitado para la prevención del interbloqueo.

**18.7.** ¿Qué es una marca de tiempo? ¿Cómo genera el sistema las marcas de tiempo?

**18.8.** Explique el protocolo de ordenación de marcas de tiempo para controlar la concurrencia. ¿En qué se diferencia la ordenación estricta de marcas de tiempo de la ordenación básica de marcas de tiempo?

**18.9.** Explique dos técnicas multiversión para controlar la concurrencia.

**18.10.** ¿Qué es un bloqueo de certificación? ¿Cuáles son las ventajas y los inconvenientes de utilizar los bloqueos de certificación?

**18.11.** ¿En qué se diferencian las técnicas de control de la concurrencia optimistas de otras técnicas de control de la concurrencia? ¿Por qué se llaman también técnicas de validación o certificación? Explique las fases típicas de un método optimista de control de la concurrencia.

**18.12.** ¿Cómo afecta la granularidad de los elementos de datos al rendimiento del control de la concurrencia? ¿Qué factores afectan a la selección del tamaño de granularidad para los elementos de datos?

**18.13.** ¿Qué tipo de bloqueos son necesarios para las operaciones de inserción y eliminación?

**18.14.** ¿Qué es el bloqueo de granularidad múltiple? ¿Bajo qué circunstancias se utiliza?

**18.15.** ¿Qué son los bloqueos de intención?

**18.16.** ¿Cuándo se utilizan los cerrojos?

**18.17.** ¿Qué es un registro fantasma? Explique el problema que un registro fantasma puede provocar en el control de la concurrencia.

**18.18.** ¿Cómo resuelve el bloqueo de índice el problema del fantasma?

**18.19.** ¿Qué es un bloqueo de predicado?

## Ejercicios

**18.20.** Demuestre que el protocolo de bloqueo en dos fases básico garantiza la serialización por conflicto de las planificaciones. (*Sugerencia:* Demuestre que, si un gráfico de serialización para una planificación tiene un ciclo, entonces al menos una de las transacciones participantes en la planificación no obedece el protocolo de bloqueo en dos fases).

**18.21.** Modifique las estructuras de datos para los bloqueos multimodo y los algoritmos para bloquear_lectura($X$), bloquear_escritura($X$) y desbloquear($X$) de modo que sean posibles la degradación y la promoción de los bloqueos. (*Sugerencia:* El bloqueo necesita comprobar el o los id de transacción que conservan el bloqueo, si los hay).

**18.22.** Demuestre que el bloqueo en dos fases estricto garantiza las planificaciones estrictas.

**18.23.** Demuestre que los protocolos esperar-morir y herir-esperar evitan el interbloqueo y la inanición.

**18.24.** Demuestre que la espera cautelosa evita el interbloqueo.

**18.25.** Aplique el algoritmo de ordenación de marcas de tiempo a las planificaciones de la Figura 17.8(b) y (c), y determine si el algoritmo permitirá la ejecución de las planificaciones.

**18.26.** Repita el Ejercicio 18.25, pero utilice el método de ordenación de marcas de tiempo multiversión.

**18.27.** ¿Por qué el bloqueo en dos fases no se utiliza como método para controlar la concurrencia para índices como los árboles $B^+$?

**18.28.** La matriz de compatibilidad de la Figura 18.8 muestra que los bloqueos IS e IX son compatibles. Explique por qué es así.

**18.29.** El protocolo MGL dice que una transacción $T$ puede desbloquear un nodo $N$, sólo si ninguno de los hijos del nodo $N$ está bloqueado por la transacción $T$. Demuestra que, sin esta condición, el protocolo MGL sería incorrecto.

## Bibliografía seleccionada

El protocolo de bloqueo en dos fases y el concepto de bloqueos de predicado se propusieron por vez primera en Eswaran y otros (1976). Bernstein y otros (1987), Gray y Reuter (1993), y Papadimitriou (1986) se centran en el control de la concurrencia y la recuperabilidad. Kumar (1996) se centra en el rendimiento y los métodos de control de la concurrencia. El bloqueo se explica en Gray y otros (1975), Lien y Weinberger (1978), Kedem y Silbershatz (1980), y Korth (1983). Los interbloqueos y los gráficos de espera se formalizaron en Holt (1972), y los esquemas esperar-morir y herir-esperar se presentaron en Rosenkrantz y otros

(1978). La espera cautelosa se explicó en Hsu y otros (1992). Helal y otros (1993) compara varios métodos de bloqueo. Las técnicas de control de la concurrencia basadas en las marcas de tiempo se explican en Bernstein y Goodman (1980) y Reed (1983). El control optimista de la concurrencia se explica en Kung y Robinson (1981) y Bassiouni (1988). Papadimitriou y Kanellakis (1979) y Bernstein y Goodman (1983) explican las técnicas multiversión. La ordenación de marcas de tiempo multiversión se propuso en Reed (1978, 1983), y el bloqueo en dos fases multiversión se explica en Lai y Wilkinson (1984). En Gray y otros (1975) se propuso un método para el bloqueo con granularidades múltiples, y el efecto de bloquear las granularidades se analiza en Ries y Stonebraker (1977). Bhargava y Reidl (1988) presenta un método para seleccionar dinámicamente entre distintos métodos de control de la concurrencia y de recuperabilidad. Los métodos de control de la concurrencia para los índices se presentan en Lehman y Yao (1981) y en Shasha y Goodman (1988). En Srinivasan y Carey (1991) se presenta un estudio del rendimiento de varios algoritmos de control de la concurrencia con árboles $B^+$.

Otro trabajo reciente sobre el control de la concurrencia incluye el control de la misma basándose en la semántica (Badrinath y Ramamritham, 1992), los modelos de transacción para actividades de ejecución larga (Dayal y otros, 1991), y la administración de transacciones multinivel (Hasse y Weikum, 1991).