



Benemérita Universidad Autónoma de Puebla

Facultad de Ciencias de la Computación

*“ADMINISTRACIÓN DE TRANSACCIONES
EN BASES DE DATOS,
CASO DE ESTUDIO: MySQL”*

TESIS PROFESIONAL

PARA OBTENER EL TÍTULO DE:

**LICENCIADO EN CIENCIAS
DE LA COMPUTACIÓN**

PRESENTA:

JUAN ANGEL TABOADA MONDRAGÓN

ASESOR:

MARIA DEL ROCÍO BOONE ROJAS

OCTUBRE 2010

AGRADECIMIENTO

A mis padres, por su apoyo y sacrificio incondicional durante mis estudios y mi vida personal, por sus sabios consejos, desvelos, amor, cariño, comprensión, paciencia y dedicación; espero se sientan orgullosos de la persona que han formado y que Dios me los conserve por mucho tiempo. Uds. son mi ejemplo a seguir y una parte muy importante en mis éxitos.

A mi esposa Mariana y a mis hijos Angel Emiliano y Zoe Amairany, que llegaron a darle un impulso y sentido importante a mi vida, a ser mi motor y el de mis metas, por Uds. seguiré siempre adelante y juntos llegaremos a cumplir nuestros más grandes anhelos. Confío en Dios en que nos ayude en tener la inteligencia, fortaleza, dedicación y coraje para que podamos salir siempre adelante.

A mis hermanas y hermanos Pepe †, Coco, Toña, Eric, Mary y Rafa, gracias por su valioso e incondicional apoyo, por estar siempre conmigo, tanto en las buenas, como en las malas, por sus consejos, sus palabras de aliento, ustedes también son parte muy importante de lo que soy.

A mis sobrinas y sobrinos Ary, Edna, Lauris, Andy, Claus, Miry, Yessi, Ana, Erick, Emanuel, Rafa, Ayla, Uriel por el haberme hecho participe en su bella infancia, por esos gratos momentos que pasamos en familia. Me siento muy orgulloso de ustedes.

A todos mis amigos, por su gran apoyo incondicional, confianza, lealtad y gratos momentos de sano esparcimiento.

A mis Maestros, por su disposición, entrega, enseñanzas, consejos y apoyo; el concluir con esta etapa de mi vida. Uds. han forjado mi carrera profesional con buenos cimientos, espero no defraudarlos y contribuir con mi granito de arena para que este país crezca y lleguemos a ser una mejor y gran nación.

ÍNDICE GENERAL

PAGINA

INTRODUCCIÓN.....	4
1. CONCEPTOS FUNDAMENTALES DE LA ADMINISTRACIÓN DE TRANSACCIONES.....	6
1.1. Definición de Transacción.....	7
1.2. Sistema de uno y de varios usuarios.....	8
1.3. Estados de las transacciones y operaciones adicionales.....	9
1.4. Operaciones de lectura y escritura en una transacción.....	11
1.5. Bitácora de sistema.....	12
1.6. Punto de confirmación de una transacción.....	13
1.7. Puntos de control en la bitácora del sistema.....	14
1.8. Propiedades deseables en las transacciones.....	14
1.9. Tipos de Transacciones.....	15
1.9.1. Transacciones de larga duración.....	15
1.9.2. Transacciones anidadas.....	17
1.9.3. Transacciones de compensación.....	18
1.9.4. Implementación.....	19
1.9.5. Restricciones temporales.....	20
2. RECUPERACIÓN DE TRANSACCIONES EN BASES DE DATOS.....	21
2.1. ¿Por qué es necesaria la recuperación?.....	22
2.2. Tipo de Fallos.....	24
2.3. Recuperación del sistema y de los medios de almacenamiento.....	27
2.4. Falla del sistema.....	28
2.5. Falla de los medios de almacenamiento.....	30
2.6. Confirmación de dos fases.....	30
3. CONTROL DE CONCURRENCIA.....	32
3.1. ¿Por qué es necesario el control de concurrencia?.....	33
3.2. Planificaciones en serie y en paralelo.....	33
3.3. Protocolos basados en bloqueo.....	37
3.4. El protocolo de bloqueo de dos fases.....	41
3.5. Protocolos basados en grafos.....	43
3.6. Protocolos basados en hora de entrada.....	45
3.7. Protocolo de ordenación por hora de entrada.....	46
3.8. Protocolo Basado en Validación.....	48
3.9. Protocolo en Árbol.....	50
3.10. Esquema multiversión.....	53
3.11. Las operaciones insertar y suprimir.....	54
3.12. Bloqueo Mortal o Deadlock.....	55

	PAGINA
4. CASO DE ESTUDIO: TRANSACCIONES EN MySQL.....	60
4.1. Características de MySQL.....	61
4.2. Tipos de tablas usadas por MySQL.....	61
4.3. Que es un Stored Procedure?.....	63
4.3.1. Stored Procedures: Ventajas.....	63
4.3.2. Creando un Stored Procedure en MySQL.....	64
4.3.3. Cursores.....	64
4.3.4. Que es un trigger?.....	64
4.4. Transacciones.....	65
4.5. Transacciones en MySQL.....	67
4.5.1. Transacciones en MySQL: bloqueos.....	68
4.5.2. Transacciones en MySQL: ejemplos.....	69
ANEXO A.....	74
CONCLUSIONES.....	77
BIBLIOGRAFÍA.....	78

INTRODUCCIÓN.

Antecedentes.

Un sistema de bases de datos completo ha de ofrecer por fuerza varios recursos adicionales, aparte de las funciones básicas de acceso a datos, y un conocimiento general de la tecnología de bases de datos no sería completo si no se entendieran hasta cierto punto esos recursos adicionales. Los recursos en cuestión, un conjunto un tanto heterogéneo, se puede dividir en dos categorías amplias: los proporcionados por el DBMS en sí y los proporcionados por algún subsistema de sección frontal o por una aplicación integrada u otro componente auxiliar.

Antes que nada, la información en una base de datos esta sujeta a varios posibles peligros (tanto deliberados como accidentales). Por ejemplo, un disco podría sufrir un daño físico o destruirse, o ciertos daños confidenciales, podrían quedar expuestos a un usuario no autorizado. Por tanto, el sistema, o en términos más precisos el DBMS base, deberá ofrecer un conjunto apropiado de controles para proteger la base de datos contra ese tipo de riesgos. Entre los controles necesarios para desarrollar nuestro tema están los de recuperación y concurrencia. Los problemas de recuperación y concurrencia en un sistema de base de datos están muy vinculados a la noción de procesamiento de información.

Nota: Suponemos en la mayor parte de esta tesis que se está en un ambiente “grande” o de macro computador. Los DBMS “pequeños” suelen ofrecer poco o ningún apoyo para la recuperación y la concurrencia; la cuestión concurrencia sencillamente no es pertinente y la recuperación se considera como problema del usuario (el usuario debe preparar copias de respaldo de la base de datos y rehacer su trabajo en forma manual si se presenta una falla).

Existen diversos modelos de transacciones propuestos para actividades de larga duración, interactivas y cooperativas en el contexto del desarrollo de un software para múltiples usuarios y ambientes CAD/CAM (Computer Aided Design, “Diseño Asistido por Computadora”)/(Computed Aided Manufacturin, “Manufactura Asistida por Computadora”). Muchas de esas son variantes del modelo **checkout**¹, el cual direcciona la larga duración y la naturaleza interactiva de las actividades soportadas por estos ambientes, pero todavía aísla a los usuarios del ambiente, puesto que su fabricación hace difícil su interacción mientras algunas actividades se están ejecutando. Sin embargo pocos modelos de transacciones cooperativas se han propuesto para facilitar la colaboración, mientras mantengan algunas garantías de consistencia.

Las transacciones de las bases de datos convencionales garantizan la atomicidad en dos sentidos: atomicidad de concurrencia, para accesos concurrentes consistentes, y la atomicidad del fracaso, persistencia consistente. Desafortunadamente, estas propiedades de atomicidad limitan severamente la

¹ El modelo básico **checkout**, es apoyado por numerosas herramientas fabricadas comercialmente para el desarrollo de software (ejemplo, **Adele** [Estublier en 1984], **DFEE** [Leblang y Chase en 1987], **SMS** [Schwanke en 1989]). Katz en 1990 da una apreciación comprensiva de la versión y configuración de sistemas orientados hacia ambientes CAD/CAM.

aplicación del concepto de transacción con respecto a modernas aplicaciones en bases de datos tal como el desarrollo de software y ambientes CAD/CAM, aunque todavía estas aplicaciones requieren de un acceso concurrente constante y persistente.

Desde una perspectiva de las transacciones, los ambientes multiusuario se caracterizan por:

- **Su larga duración** – actividades que consisten en secuencias de accesos a la base de datos que pueden durar de minutos a meses. El rollback de transacciones largas, funciona en uno de dos sentidos, para el control de concurrencia o la recuperación de fallas, el cual es generalmente inaceptable debido al costo económico y mortal del trabajo perdido.
- **El control interactivo** – los usuarios escogen las acciones dentro de sus actividades cuando las estén realizando. Es usualmente inconveniente el planear la programación a prioridad de transacciones, y automáticamente rehacer actividades que son a menudo no factibles.
- **La cooperación entre usuarios** – los usuarios comparten resultados parciales de sus actividades mientras se encuentran en proceso. La serializabilidad es a veces incompatible con la consistencia, cuando el esfuerzo es conjunto, se necesita tomar la base de datos desde un estado consistente semánticamente a otro.

Objetivo General.

Identificar, analizar y realizar una experimentación básica de mecanismos y problemas que plantean la recuperación y control de concurrencia de bases de datos basada en transacciones.

ESTRUCTURA DEL DOCUMENTO

El presente trabajo de tesis se encuentra organizado en 4 capítulos que son:

Capítulo 1. En este capítulo, se incluyen los conceptos fundamentales de la administración de transacciones y tipos de transacciones.

Capítulo 2. Se define la necesidad por la cual la recuperación de transacciones es indispensable, detallando algunos conceptos, los cuales son de suma importancia para el éxito de una transacción.

Capítulo 3. Se explican los principales conceptos y la necesidad del control de concurrencia para el manejo de transacciones en un ambiente de múltiples usuarios.

Capítulo 4. Se propone un caso de estudio para analizar las transacciones mediante MySQL, en donde se ejemplifican diferentes tipos de transacciones que son ejecutadas por usuarios, a la vez de caracterizar cada una de las mismas.

Al final se encuentran las conclusiones y perspectivas del trabajo así como la bibliografía utilizada.

Capítulo 1. CONCEPTOS FUNDAMENTALES DE LA ADMINISTRACIÓN DE TRANSACCIONES.

En el capítulo uno, se definirán los conceptos fundamentales de la administración de transacciones y tipos de transacciones, para comprender el significado de Transacción y sus características.

1. CONCEPTOS FUNDAMENTALES DE LA ADMINISTRACIÓN DE TRANSACCIONES.

Las aplicaciones de bases de datos a gran escala, con bases de datos de gran tamaño y con cientos de usuarios concurrentes, como los sistemas de reservas, los bancos, la bolsa, etc., requieren una alta disponibilidad, tiempo de respuesta mínimo para cada uno de los usuarios concurrentes que esté utilizando el sistema.

Las aplicaciones de bases de datos suelen ser sistemas a los que acceden concurrentemente varios usuarios por lo que es posible que varios usuarios estén intentando acceder simultáneamente a los mismos datos. Esta situación es especialmente delicada cuando alguno o varios de estos usuarios han intentado modificar los datos que están siendo utilizados por los otros usuarios, de forma que hay que controlar la gestión y el acceso a los datos en este tipo de situaciones, esto se consigue mediante el uso de **transacciones**.

1.1. Definición de Transacción.

Una transacción, de acuerdo Elmasri [10 y 11], es una unidad lógica de trabajo o procesamiento: ejecución de un programa que incluye operaciones de acceso a la base de datos.

Una transacción es una secuencia de operaciones que llevan la base de datos desde un estado de consistencia¹ a otro estado de consistencia, por esto suele decirse también que la transacción es una **unidad lógica de integridad**.

Cuando múltiples transacciones son introducidas en el sistema por varios usuarios, es necesario evitar que interfieran entre ellas de forma tal que provoquen que la BD que den en un estado no consistente; desde este punto de vista, podemos ver una transacción como una **unidad lógica de concurrencia**.

Si ocurre un fallo que provoca la caída del sistema de bases de datos, cuando había varias transacciones en curso de ejecución, muy probablemente los datos en la BD quedaran erróneos (estado inconsistente debido a un fallo); en estas circunstancias, se debe garantizar que la BD pueda ser recuperada a un estado en el cual su contenido sea consistente, observando las transacciones que terminaron su ejecución y las que quedaron a medio; por esto una transacción es considerada también una **unidad lógica de recuperación**.

La idea clave es que una transacción debe ser **atómica**, es decir, las operaciones que la componen deben ser ejecutadas en su totalidad o no ser ejecutadas en absoluto.

Una sentencia de definición o manipulación de datos ejecutada de forma interactiva (por ejemplo utilizar el SQL*Plus de Oracle para realizar una consulta) puede suponer el inicio de una transacción. Asimismo, la ejecución de una sentencia SAL por parte de un programa que no tiene ya una transacción en progreso, supone la iniciación de una transacción.

¹ El conjunto de valores almacenados en la base de datos cumple toda restricción de integridad especificada en el esquema, así como cualesquiera otras restricciones que deban cumplirse en la B. D.

Toda transacción finaliza con una operación de *commit* (confirmar) o bien con una operación de *rollback* (anular, abortar o revertir).

Tanto una operación como la otra puede ser de tipo explícito (si la propia transacción (su código) contiene una sentencia *COMMIT* o *ROLLBACK*) o implícito (si dicha operación es realizada por el sistema de forma automática, por ejemplo tras detectar una terminación normal (éxito) o normal (fallo) de la transacción).

Por defecto, una vez finalizada una transacción, si todas sus operaciones se han realizado con éxito, se realiza un *COMMIT* implícito de dicha transacción; y si alguna de ellas tuvo problemas, se lleva a cabo un *ROLLBACK* implícito de la transacción (es decir, se deshacen todas las operaciones que había realizado hasta el momento del fallo).

1.2. Sistema de uno y de varios usuarios.

Un criterio para clasificar los sistemas de bases de datos, es por el número de usuarios que pueden utilizarlos de manera concurrente, es decir, al mismo tiempo. Un DBMS es monousuario si un sólo usuario a la vez puede utilizarlo y es multiusuario si muchos usuarios pueden utilizarlo al mismo tiempo. Los DBMS monousuario suelen estar restringidos a algunos sistemas de microcomputador y casi todos los demás DBMS son multiusuarios.

El que muchos usuarios puedan utilizar los sistemas de computador al mismo tiempo se debe al concepto de multiprogramación, que permite al computador procesar simultáneamente varios programas (o transacciones). Si sólo hay una unidad central de procesamiento (UCP), en realidad sólo se puede procesar un programa a la vez. Sin embargo, los **sistemas operativos de multiprogramación** ejecutan algunas órdenes de un programa, luego suspenden ese programa y ejecutan algunas órdenes del siguiente programa y así sucesivamente. Cuando a un programa le llega el turno de usar la UCP otra vez, se reanuda su ejecución en el punto en el que se suspendió. Así pues, la ejecución concurrente de los programas es en realidad **intercalada**, como se ilustra en la *figura 1.1*, donde se muestra la ejecución concurrente de dos programas *A* y *B* de manera intercalada. La intercalación mantiene ocupada a la UCP cuando un programa en ejecución requiere una operación de entrada o de salida (E/S), como leer un bloque de un disco. La UCP se conmuta para ejecutar otro programa en vez de permanecer ociosa durante el tiempo de E/S.

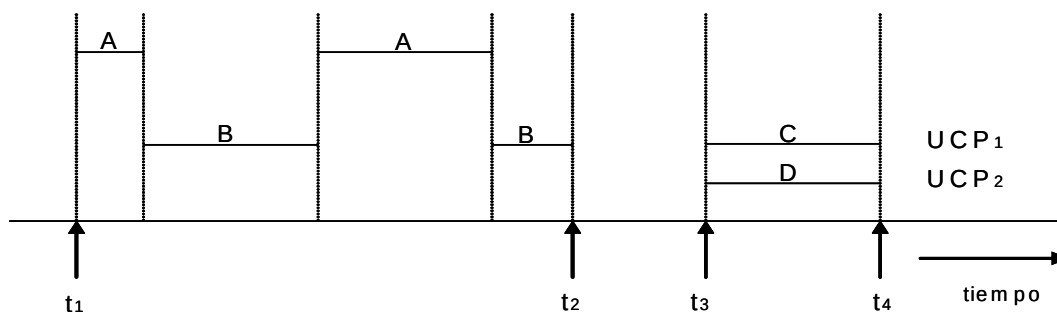


Figura 1.1 Concurrency intercalada vs. simultánea.

Si el computador cuenta con múltiples procesadores (UCP) en hardware, es posible el procesamiento **simultáneo** de varios programas, dando lugar a una concurrencia simultánea, no intercalada, como lo ilustran los programas **C** y **D** de la figura 1.1. Casi toda la teoría sobre el control de concurrencia en las bases de datos, se desarrolló en términos de la concurrencia intercalada, aunque se puede adaptar a la concurrencia simultánea.

En un DBMS multiusuario, los elementos de información almacenados son los recursos primarios a los que pueden tener acceso concurrente los programas de usuarios, que constantemente obtienen información de la base de datos y la modifican. En un sentido amplio, *a la ejecución de un programa que lee o modifica el contenido de la base de datos es a lo que llamamos una **transacción**.*

Como se dijo antes, *la ejecución de un programa* que incluye operaciones de acceso a la base de datos se denomina **transacción de base de datos** o simplemente **transacción** ¹.

1.3. Estados de las transacciones y operaciones adicionales.

Una transacción, de acuerdo Elmasri [10 y 11], es una unidad atómica de trabajo que se realiza por completo o, bien, no se efectúa en absoluto. Para fines de recuperación, el sistema necesita mantenerse al tanto de cuando la transacción se inicia, termina y se confirma o aborta. Así pues, el gestor de recuperación se mantiene al tanto de las siguientes operaciones:

- **INICIO_DE_TRANSACCIÓN:** Esta marca el principio de la ejecución de la transacción.
- **LEER o ESCRIBIR:** Estas especifican operaciones de lectura o escritura de elementos de la base de datos que se efectúan como parte de una transacción.
- **FIN_DE_TRANSACCIÓN:** Esta especifica que las operaciones de LEER y ESCRIBIR de la transacción han terminado y marca el límite de la ejecución de la transacción. Sin embargo, en este punto puede ser necesario verificar si los cambios introducidos por la transacción se pueden aplicar permanentemente a la base de datos (confirmar) o si la transacción debe abortarse porque viola el control de concurrencia o por alguna otra razón.
- **COMMIT_TRANSACCIÓN:** Este señala que la transacción terminó con éxito y que cualesquier cambios (actualizaciones) ejecutados por ella se pueden confirmar sin peligro en la base de datos y que no se cancelaran.
- **ROLLBACK (o ABORTAR):** Esta indica que la transacción terminó sin éxito y que cualesquier cambios o efectos que pueda haber aplicado a la base de datos se deben cancelar.

¹ Si las operaciones de base de datos en una transacción no actualizan datos, sino que sólo los leen, se habla de una **transacción de sólo lectura**.

Además de las anteriores, algunas técnicas de recuperación requieren operaciones adicionales como las siguientes:

- DESHACER: Similar a *rollback*, pero se aplica a una sola operación y no a una transacción completa.
- REHACER: Esta especifica que ciertas operaciones de transacción se deben repetir (rehacer) para asegurar que todas las operaciones de una transacción confirmada se hayan aplicado con éxito a la base de datos.

La *figura 1.2* muestra un diagrama de transición de estados que describe el paso de una transacción por sus estados de ejecución. Una transacción entra en un estado **activo** inmediatamente después de que inicia su ejecución y ahí puede emitir operaciones LEER y ESCRIBIR. Cuando la transacción termina, pasa al estado **parcialmente confirmado**. En este punto, algunas técnicas de control de concurrencia requieren que se efectúen ciertas verificaciones para garantizar que la transacción no interfiera otras transacciones en ejecución. Por añadidura, algunos protocolos de recuperación deben asegurar que un fallo del sistema no provocará una incapacidad para registrar permanentemente los cambios hechos por la transacción (usualmente asentando los cambios en una bitácora del sistema, de la cual hablaremos en la siguiente subsección). Una vez realizadas con éxito ambas verificaciones, se dice que la transacción ha llegado a su punto de confirmación y pasa al **estado confirmado**. Los puntos de confirmación se tratarán con mayor detalle en la sección 1.5. Una vez que una transacción está en el estado confirmado, ha concluido con éxito su ejecución.

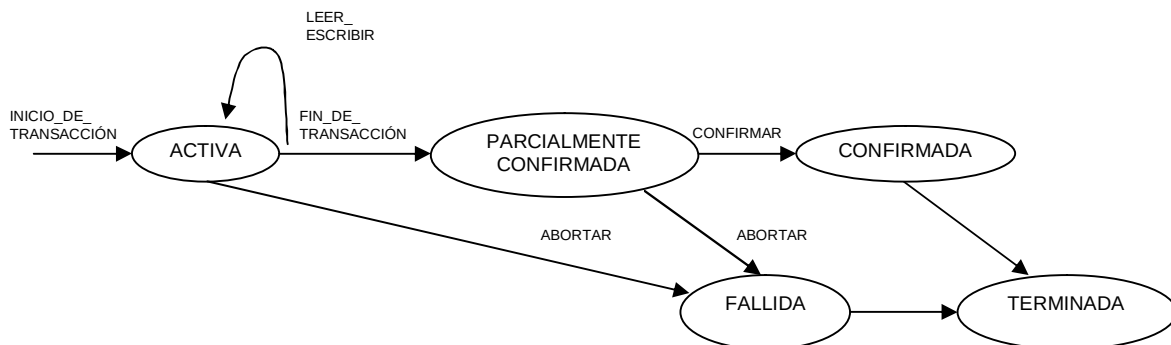


Figura 1.2. Diagrama de transición de estados para la ejecución de transacciones.

Sin embargo, una transacción puede pasar al estado **fallido** si una de las verificaciones falla o si la transacción se aborta mientras está en el estado activo. En tal caso, es posible que la transacción deba revertirse para anular los efectos de sus operaciones ESCRIBIR sobre la base de datos. El estado **terminado** corresponde al abandono del sistema por parte de la transacción. Las transacciones fallidas o abortadas pueden reiniciarse posteriormente, ya sea en forma automática o después de reintroducirse como transacciones nuevas.

1.4. Operaciones de lectura y escritura en una transacción.

Al hablar de las técnicas de control de concurrencia y de recuperación nos ocuparemos de las transacciones en el nivel de elemento de información y bloques de discos [10 y 11]. En este nivel, las operaciones de acceso a la base de datos que una transacción puede incluir son:

- **Leer_elemento(X):** Lee un elemento de la base de datos llamado X y lo coloca en una variable de programa. Para simplificar nuestra notación, supondremos que la variable de un programa también se llama X .
- **Escribir_elemento(X):** Escribe el valor de la variable de programa X en el elemento de la base de datos llamado X .

Un **bloque** es la unidad básica de transferencia de datos del disco a la memoria principal del computador. En general, un elemento de información será un campo de algún registro de la base de datos, aunque puede ser una unidad mayor, como un registro o incluso un bloque completo.

La ejecución de una orden **leer_elemento(X)** incluye los siguientes pasos:

1. Encontrar la dirección del bloque de disco que contiene el elemento X .
2. Copiar ese bloque de disco en un almacenamiento intermedio (buffer) dentro de la memoria principal (si ese bloque no está ya en almacenamiento intermedio).
3. Copiar el elemento X del almacenamiento intermedio a la variable de programa llamada X .

La ejecución de una orden **escribir_elemento(X)** incluye los siguientes pasos:

1. Encontrar la dirección del bloque de disco que contiene el elemento X .
2. Copiar ese bloque de disco en un almacenamiento intermedio dentro de la memoria principal (si ese bloque no está ya en almacenamiento intermedio).
3. Copiar el elemento X desde la variable de programa llamada X al lugar correcto dentro del almacenamiento intermedio.
4. Transferir el bloque actualizado desde el almacenamiento intermedio al disco (ya sea de inmediato o en algún momento posterior).

(a) leer_elemento(X); $X := X - N$; escribir_elemento(X); leer_elemento(Y); $Y := Y + N$; escribir_elemento(Y);	(b) leer_elemento(X); $X := X + M$; escribir_elemento(X);
--	--

Figura 1.3 Dos transacciones simples. (a) La transacción T_1 . (b) La transacción T_2 .

El paso 4 es el que de hecho actualiza la base de datos en disco. En algunos casos el almacenamiento intermedio no se transfiere de inmediato al disco, por si fuera necesario hacer cambios adicionales en el buffer. Por lo regular, la decisión

sobre cuando almacenar en disco un bloque modificado que esté en un almacenamiento intermedio corresponde al gestor de recuperación o el sistema operativo.

Para tener acceso a la base de datos, una transacción deberá incluir operaciones leer_elemento y escribir_elemento. La *figura 1.3* muestra ejemplos de dos transacciones muy simples. Los mecanismos de control de concurrencia y de recuperación se ocupan principalmente de las órdenes de acceso a la base de datos incluidas en una transacción.

Las transacciones introducidas por los diversos usuarios se podrían ejecutar de manera concurrente y podrían leer y actualizar los mismos elementos de la base de datos. Si esta ejecución concurrente no se controla, puede provocar que la base de datos no sea consistente. Mas adelante presentaremos de manera informal tres de los problemas que pueden presentarse cuando se ejecutan sin control las transacciones concurrentes.

1.5. Bitácora del sistema.

Para poderse recuperar de los fallos de transacciones [10 y 11], el sistema mantiene una **bitácora**¹. La bitácora se mantiene en disco, de modo que no la afecta ningún tipo de fallo, más que los de disco o catastróficos. Además la bitácora se respalda periódicamente en cinta para protegerse contra fallos catastróficos. A continuación mencionaremos los tipos de entradas que se describen en la bitácora y la acción que realiza cada una de ellas. En estas entradas, *T* se refiere a un **identificador de transacción** único que el sistema genera automáticamente y que sirve para identificar cada transacción:

1. (inicio_de_transacción, *T*): Asienta que se ha iniciado la ejecución de la transacción *T*.
2. (escribir_elemento, *T*, *X*, valor_anterior, nuevo_valor): Asienta que la transacción *T* cambió el valor del elemento de base de datos *X* de valor_anterior a nuevo_valor.
3. (leer_elemento, *T*, *X*): Asienta que la transacción *T* leyó el valor del elemento de base de datos *X*.
4. (commit, *T*): Asienta que la transacción *T* terminó con éxito y establece que su efecto se puede confirmar (asentar permanentemente) en la base de datos.
5. (rollback, *T*): Asienta que se abortó la transacción *T*.

Los protocolos de recuperación que evitan las reversiones en cascada no requieren que las operaciones LEER se asienten en la bitácora del sistema, pero otros protocolos sí necesitan estas entradas para la recuperación. En el primer caso, el costo extra de asentar las operaciones en la bitácora se reduce, porque en ella se registran menos operaciones, sólo las de ESCRIBIR. Además, los protocolos estrictos requieren entradas ESCRIBIR más simples que no incluyen nuevo_valor.

¹ (A veces llamada diario) La **bitácora** sigue la pista a todas las operaciones de transacciones que afectan los valores de elementos de la base de datos.

Cabe señalar que aquí estamos suponiendo que las transacciones no se pueden anidar. También suponemos que todos los cambios permanentes de la base de datos ocurren dentro de las transacciones, así que la noción de recuperarse de una transacción equivale a deshacer o rehacer las operaciones recuperables individualmente a partir de la bitácora. Si el sistema se cae, podemos recuperar la base de datos a un estado consistente examinando la bitácora. Dado que la bitácora contiene un registro de cada operación ESCRIBIR que altera el valor de algún elemento de la base de datos, es posible **deshacer** (cancelar) el efecto de estas operaciones ESCRIBIR de una transacción T , rastreando hacia atrás en la bitácora y reestableciendo todos los elementos alterados con una operación ESCRIBIR de T a su valor_anterior. También podemos **rehacer** el efecto de las operaciones ESCRIBIR de una transacción T rastreando hacia delante en la bitácora y cambiando todos los elementos modificados por una operación ESCRIBIR de T a su nuevo_valor. Puede ser necesario rehacer las operaciones de una transacción si todas sus actualizaciones están asentadas en la bitácora, pero ocurre un fallo antes de que estemos seguros de que todos los nuevos valores se han escrito permanentemente en la base de datos real.

1.6. Punto de confirmación de una transacción.

Una transacción T llega a su **punto de confirmación**¹ cuando todas sus operaciones que tienen acceso a la base de datos se han ejecutado con éxito y el efecto de todas estas operaciones se ha asentado en la bitácora [10 y 11]. Más allá del punto de confirmación, se dice que la transacción está **confirmada** y se supone que su efecto se *asentó permanentemente* en la base de datos. En ese momento la transacción escribe una entrada (*commit*, T) en la bitácora. Si ocurre un fallo del sistema, buscamos en la bitácora todas las transacciones T que han escrito una entrada (*inicio_de_transacción*, T) en la bitácora pero no han escrito todavía su entrada (*commit*, T); es posible que durante el proceso de recuperación estas transacciones tengan que revertirse para cancelar su efecto sobre la base de datos. Las transacciones que ya escribieron su entrada de confirmación en la bitácora también deberán haber asentado en ella todas sus operaciones ESCRIBIR, pues de lo contrario no estarían confirmadas; su efecto sobre la base de datos puede rehacerse a partir de las entradas de la bitácora.

Observe que el archivo de la bitácora debe mantenerse en disco. La actualización de un archivo en disco implica copiar el bloque apropiado del archivo del disco a un almacenamiento intermedio en la memoria principal, actualizar este almacenamiento intermedio y copiarlo de la memoria principal al disco. Con frecuencia se mantiene un bloque del archivo de bitácora en memoria principal hasta que se llena de entradas y luego se escribe una sola vez en el disco, en vez de escribirlo cada vez que se añade una entrada. Esto ahorra el gasto extra de escribir varias veces en disco el mismo bloque de archivo. Cuando se cae el sistema, sólo las entradas de bitácora que se han escrito en disco se considera que están en el proceso de recuperación, porque puede haberse perdido el contenido de la memoria principal. Por ello, antes de que una transacción llegue a su punto de confirmación, cualquier porción de la bitácora que no se haya escrito

¹ También se conoce como **punto de sincronización**, especialmente en productos comerciales.

en el disco todavía se deberá grabar. Este proceso se denomina forzar la escritura del archivo de bitácora (también llamado *Protocolo de bitácora de escritura adelantada*) antes de confirmar una transacción.

1.7. Puntos de control en la bitácora del sistema.

Otro tipo de entradas en la bitácora es el llamado punto de control (punto de revisión). Un registro (punto_de_control) se escribe en la bitácora periódicamente cada vez que el sistema escribe en la base de datos en disco, el efecto de todas las operaciones ESCRIBIR de las transacciones confirmadas [10 y 11]. Así pues, todas las transacciones que tengan sus entradas (*commit,T*) en la bitácora antes de una entrada (punto_de_control) no necesitarán la repetición de sus operaciones ESCRIBIR en caso de una caída del sistema.

El gestor de recuperación de un DBMS debe decidir con qué intervalos asentar un punto de control. El intervalo puede medirse en el tiempo, digamos, cada *m* minutos, o por el número *t* de transacciones confirmadas desde el último punto de control, donde los valores de *m* o de *t* son parámetros del sistema. El asentamiento de un punto de control consiste en las siguientes acciones:

1. Suspender temporalmente la ejecución de las transacciones.
2. Forzar la escritura de todas las operaciones de actualización pertenecientes a transacciones confirmadas del almacenamiento intermedio al disco.
3. Escribir un registro (punto_de_control) en la bitácora y forzar la escritura de la bitácora en el disco.
4. Reanudar la ejecución de transacciones.

Un registro de punto de control en la bitácora también puede incluir información adicional, como una lista de identificadores de las transacciones activas o las ubicaciones (direcciones) del primer registro y del más reciente (último) en la bitácora para cada transacción activa. Esto puede facilitar la anulación de operaciones en caso de ser necesaria la reversión de una transacción.

1.8. Propiedades deseables en las transacciones.

Las transacciones atómicas deben poseer varias propiedades [10 y 11]. Estas se conocen como ACID (Atomicity, Consistency, Isolation, Durability) y deben ser impuestas por los métodos de control de concurrencia y de recuperación de los DBMS. Las propiedades ACID son:

1. **Atomicidad**: Una transacción es una unidad atómica de procesamiento; o bien se realiza por completo o no se realiza en absoluto.
2. **Conservación de la consistencia**: Una ejecución correcta de la transacción debe llevar a la base de datos de un estado consistente a otro.
3. **Aislamiento**: Una transacción no debe dejar que otras transacciones puedan ver sus actualizaciones antes de que sea confirmada; esta propiedad, cuando se hace cumplir estrictamente, resuelve el problema de la actualización temporal y hace innecesarias las reconversiones en cascada de las transacciones.

4. **Durabilidad o permanencia**: Una vez que una transacción ha modificado la base de datos y las modificaciones se han confirmado, éstas nunca deben perderse por un fallo subsecuente.

La propiedad de atomicidad requiere que ejecutemos una transacción hasta completarla. Es obligación del método de recuperación garantizar la atomicidad. Si por alguna razón una transacción no puede completarse, como por una caída del sistema a la mitad de su ejecución, el método de recuperación debe cancelar todos los efectos de la transacción sobre la base de datos.

En general, se considera que la propiedad de conservación de la consistencia es responsabilidad de los programadores que escriben los programas de base de datos o del módulo del DBMS que impone las restricciones de integridad. Recuerde que un **estado de la base de datos** es una colección de todos los elementos de información (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, además de cualesquier otras restricciones que deban cumplirse en la base de datos. Los programas de base de datos deben elaborarse a modo de garantizar que, si la base de datos está en un estado consistente antes de ejecutar la transacción, estará en un estado consistente después de la ejecución completa de la transacción, suponiendo que no hay interferencias con otras transacciones.

El aislamiento lo impone el método de control de concurrencia. Se ha intentado definir el nivel o grado de aislamiento de una transacción. Se dice que una transacción tiene aislamiento de grado 0 (cero) si no sobrescribe las lecturas sucias de transacciones de nivel más alto; una transacción con nivel de aislamiento 1 (uno) no tiene actualizaciones perdidas y el aislamiento grado 2 no tiene actualizaciones perdidas ni lecturas sucias. Por último, el aislamiento grado 3 (también llamado aislamiento verdadero) tiene, además de las propiedades del grado 2, lecturas repetibles. La propiedad de durabilidad es responsabilidad del método de recuperación.

1.9. Tipos de Transacciones.

1.9.1. Transacciones de larga duración.

El concepto de transacción, según Date [7], fue desarrollado inicialmente en el contexto de aplicaciones de procesamiento de datos en el que la mayor parte de las transacciones no son interactivas y son de corta duración. Cuando este concepto es aplicado a sistemas de bases de datos que implican interacción humana, surgen serios problemas. Estas interacciones tienen las siguientes propiedades clave:

- **Larga duración.** Cuando hay una interacción humana con una transacción activa, esa transacción llega a ser de larga duración desde la perspectiva del computador, puesto que el tiempo de respuesta humana es lento comparado con la velocidad del computador. Además, en aplicaciones de diseño, la actividad humana que se está modelando puede implicar horas,

días o incluso un largo periodo. Así pues, las transacciones pueden ser de larga duración en términos humanos y también en términos de la máquina.

- **Exposición de datos no ejecutados.** Los datos que una transacción de larga duración genera y presenta al usuario no están ejecutados, puesto que la transacción puede abortar. Así, los usuarios, y como resultado otras transacciones, pueden ser forzados a leer datos no ejecutados. Si varios usuarios cooperan en un diseño, puede que las transacciones de usuarios necesiten intercambiar datos antes de que la transacción se ejecute.
- **Subtareas.** Una transacción interactiva puede constar de un conjunto de subtareas iniciadas por el usuario. El usuario puede que desee abortar una subtarea sin causar necesariamente que aborte toda la transacción.
- **Recuperabilidad.** No es aceptable abortar una transacción interactiva de larga duración por una caída del sistema. La transacción activa debe recuperarse a un estado que existía poco antes de la caída de forma que se pierda relativamente poco trabajo humano.
- **Rendimiento.** El buen rendimiento en un sistema de transacciones interactivas se define como tiempo de respuesta rápido; a diferencia de los sistemas no interactivos, en los que el objetivo es el rendimiento alto (número de transacciones por segundo). Los sistemas de alto rendimiento hacen que el uso de los recursos del sistema sea eficiente. Sin embargo, en el caso de transacciones interactivas, el <<recurso>> más costoso es el usuario. Para optimizar la eficiencia y satisfacción del usuario, el tiempo de respuesta debe ser rápido (desde una perspectiva humana). En los casos en los que una tarea tarde mucho tiempo, el tiempo de respuesta debe ser previsible (es decir, la variación en los tiempos de respuesta debe ser baja), de forma que los usuarios puedan manejar bien su tiempo.

Efectos adversos en transacciones de larga duración.

Cada uno de los protocolos convencionales de control de concurrencia, tiene efectos adversos en las transacciones de larga duración:

- **Bloqueo de dos fases.** Cuando un bloqueo no puede garantizarse, se fuerza a la transacción que solicita el bloqueo a que espere que se desbloquee el dato en cuestión. La duración de esta espera es proporcional a la duración de la transacción que tiene el bloqueo. Si el dato está bloqueado por una transacción de corta duración, esperamos que el tiempo de espera sea corta (excepto en el caso de bloqueo total o de sobrecargo del sistema). Sin embargo, si el dato está bloqueado por una transacción de larga duración, la espera será de larga duración. Los tiempos de espera largos conllevan un tiempo de espera más largo y un aumento de la probabilidad de bloqueo total.
- **Protocolos basados en grafos.** Los protocolos basados en grafos permiten que los bloqueos se liberen antes que bajo los protocolos de bloqueo de dos fases, y previene el bloqueo total. Sin embargo, imponen una ordenación en los datos. Las transacciones deben bloquear los datos de una manera consistente con esa ordenación. Como resultado, una transacción puede que tenga que bloquear más datos de los que necesite.

Además, una transacción debe tener un bloqueo hasta que no haya ninguna probabilidad de que este se vuelva a necesitar. Así es probable que el bloqueo de larga duración espere.

- **Protocolos basados en hora de entrada.** Los protocolos basados en hora de entrada nunca requieren que espere una transacción. Sin embargo, requieren que las transacciones aborten bajo determinadas circunstancias. Si se aborta una transacción de larga duración, se pierde una cantidad importante de trabajo. Para transacciones no interactivas, este trabajo perdido repercute en el rendimiento. Para transacciones interactivas, repercute también en la satisfacción del usuario. No es deseable en absoluto que un usuario se de cuenta de que se han perdido varias horas de trabajo.
- **Validación.** Al igual que los protocolos basados en hora de entrada, los protocolos de validación hacen que se cumpla la serializabilidad abortando las transacciones.

Así, parece que el hacer que se cumpla la serializabilidad resulta en esperas de larga duración, aborto de transacciones de larga duración o ambos. Al hacer que se cumpla la serializabilidad surgen problemas adicionales cuando consideramos la recuperación. Anteriormente tratamos el problema del retroceso en cascada, en el que el aborto de una transacción puede conducir al aborto de otras transacciones. Este fenómeno es indeseable, particularmente para las transacciones de larga duración. Si se utiliza bloqueo, los bloqueos exclusivos deben mantenerse hasta el final de la transacción si se va a evitar el retroceso en cascada. Sin embargo, esto aumenta la duración del tiempo de espera ¹.

1.9.2. Transacciones anidadas.

Una transacción de larga duración, según Date [7], puede ser vista como una colección de subtareas o subtransacciones relacionadas. Estructurando una transacción como un conjunto de subtransacciones, podemos aumentar el paralelismo, puesto que puede ser posible ejecutar varias subtransacciones en paralelo. Además, es posible tratar los fallos de una subtransacción (debidos a abortos, caída del sistema, etc.), sin tener que retroceder la transacción de larga duración entera.

Una transacción anidada T se representa mediante un conjunto $T'=\{t_1, t_2, \dots, t_n\}$ de subtransacciones y un orden parcial de P en T . Una subtransacción t_i de T puede abortar sin obligar a T a que aborte. En lugar de ello, T puede reiniciar a t_i o simplemente decidir que t_i no se ejecute. Si t_i se ejecuta, esto no hace a t_i permanente. En cambio, t_i ejecuta a T y todavía puede abortar si T aborta. Una ejecución de T no debe violar el orden parcial P , es decir, si en el grafo de precedencia aparece una arista $t_i \rightarrow t_j$ entonces $t_j \rightarrow t_i$ no debe estar en el cierre transitivo de P .

¹ Así, parece que el cumplimiento de la atomicidad de la transacción debe conducir a un aumento de la probabilidad de esperas de larga duración o a crear una posibilidad de retroceso en cascada.

La anidación puede tener varios niveles de profundidad, representando una subdivisión de una transacción en subtareas, subsubtarear y así sucesivamente. En el nivel más bajo de la anidación tenemos las operaciones estándar de la base de datos **read** y **write** que hemos utilizado anteriormente.

Aunque el principal valor práctico de las transacciones anidadas surge en las transacciones de larga duración complejas, usaremos el ejemplo sencillo de la figura para mostrar como la anidación puede crear operaciones de más alto nivel que aumentan la concurrencia. Volvemos a escribir la transacción *T15* usando las subtransacciones *tu* y *tb* que realizan operaciones de incremento o decremento.

- *T15* consta de:
 - o *ta* resta 50 de A.
 - o *tb* suma 50 a B.

De manera similar, volvemos a escribir la transacción *T16* usando las subtransacciones *tc* y *td* que realizan operaciones de incremento o decremento:

- *T16* consta de:
 - o *tc* resta 10 de B.
 - o *td* suma 10 a A.

No se especifica ningún orden en *ta*, *tb*, *tc* y *td*. Cualquier ejecución de estas subtransacciones generará un resultado correcto. La planificación de la figura 1.9 corresponde a la planificación $\langle ta, tc, tb, td \rangle$.

<i>T15</i>	<i>T16</i>
read(A) <i>A:=A-50</i> write(A)	
	read(B) <i>B:=B-10</i> write(B)
read(B) <i>B:=B+50</i> write(B)	
	read(A) <i>A:=A+10</i> write(A)

Figura 1.9 Una Planificación no serializable en conflictos.

1.9.3. Transacciones de compensación.

Anteriormente utilizamos la operación **undo** para eliminar los cambios en la base de datos que realizó una transacción abortada. Ello garantiza la atomicidad de las transacciones [7].

El acercamiento a este enfoque tiene efectos adversos en las transacciones de larga duración. Para reducir la frecuencia de espera de larga duración, es

necesario que las transacciones que no han terminado se expongan a otras transacciones que se ejecutan de manera concurrente. Sin embargo, la exposición de los datos no ejecutados puede crear retrocesos en cascada.

El concepto de *transacciones de compensación* trata de este problema proporcionando una alternativa a la operación **undo** para recuperación en fallos de transacciones. En vez de deshacer estrictamente todos los cambios que produjo la transacción que ha fallado, se presenta especial atención a la “compensación” del fallo.

1.9.4. Implementación.

Los conceptos de transacción tratados en esta sección, crean serias dificultades de implementación. Aquí presentamos algunos de estos problemas y tratamos como se pueden resolver ¹ [7].

Las transacciones de larga duración deben sobrevivir a las caídas del sistema. Esto puede lograrse realizando un **redo** en las subtransacciones terminadas y realizando un **undo** o una compensación para todas las subtransacciones de corta duración que estuvieran activas en el momento de la caída. Sin embargo, esto sólo es parte del problema. En los sistemas de base de datos típicos, estos datos internos del sistema como tablas de bloqueo y horas de entrada de transacciones se guardan en almacenamiento estable. Para que una transacción de larga duración se reanude después de una caída, estos datos se deben restaurar. Por tanto, no sólo es necesario registrar los cambios en la base de datos, sino también los cambios en los datos internos del sistema pertenecientes a las transacciones de larga duración.

El registro de actualizaciones se hace más complejo según los tipos de datos que pueda haber en la base de datos. Un dato puede estar en un diseño CAD, en el texto de un documento o en otra forma de diseño compuesto. Estos datos son grandes físicamente. Por tanto, no es deseable que se almacenen en un registro de bitácora ambos valores del dato: el antiguo y el nuevo.

Existen dos enfoques para reducir el gasto extra de garantizar la recuperación de los datos grandes:

- **Registro de operación.** Sólo se almacenan en el registro la operación realizada en el dato y el nombre del dato. Para cada operación, debe existir una operación inversa. **Undo** se realiza utilizando la operación inversa y **redo** se utiliza usando la propia operación. La recuperación mediante la operación de registro es más difícil, ya que **redo** y **undo** no son idempotentes.
- **Registro y doble paginación.** El registro se utiliza para modificaciones en datos pequeños, mientras que los datos grandes se recuperan utilizando una técnica de doble paginación. Usando la doble paginación, solo se necesita almacenar en duplicado las páginas que realmente se modifiquen.

¹ **Nota:** Sea cual sea la técnica utilizada, las complejidades que introducen las transacciones de larga duración y los datos complican el proceso de recuperación. Por tanto, es deseable permitir que ciertos datos que no sean críticos, estén exentos de registro y confiar en las copias de seguridad fuera de línea y en la intervención humana.

1.9.5. Restricciones temporales.

Las transacciones que hemos considerado hasta ahora pertenecen a los valores almacenados en la base de datos. En determinadas aplicaciones, las restricciones incluyen una fecha/hora tope en la que una tarea debe terminarse. Ejemplos de estas aplicaciones incluyen gestión de planta, control de tráfico y planificación. Cuando se incluyen las fechas/horas tope, el que una ejecución sea correcta ya no es únicamente un favor de consistencia de la base de datos. Más bien nos interesa cuantas fecha/horas tope faltan y por cuanto tiempo faltan [7]. Las fechas/horas tope se caracterizan como:

- **Fijas (Hard).** La tarea tiene valor cero si termina antes de la fecha/hora tope.
- **Temporales (Soft).** La tarea ha disminuido el valor si termina después de la fecha/hora tope, con el valor próximo a cero según aumenta el grado de latencia. Los sistemas con fechas/horas tope se denominan *sistemas en tiempo real*.

La gestión de transacciones en sistemas en tiempo real debe tener en cuenta fechas/horas tope. Si el protocolo de concurrencia determina que una transacción T debe esperar, puede causar que T pierda su fecha/hora tope. En tales casos, puede ser preferible evitar que se ejecute la transacción que tiene el bloqueo y permitir que T continúe. Sin embargo, la prevención (el evitar la ejecución) debe utilizarse con cuidado, porque el tiempo que ha perdido la transacción de la que se previene (debido al retroceso y al reinicio) puede causar que pierda su fecha/hora tope.

Desafortunadamente, es difícil determinar si es preferible el retroceso o la espera en una situación dada. El problema surge de la variación en el tiempo de ejecución de las transacciones. En el mejor de los casos, todos los accesos a datos se refieren a datos del buffer de la base de datos. En el peor de los casos, cada acceso hace que se escriba una página del buffer en el disco (precedido de los registros de bitácora necesarios) seguido de la lectura del disco de la página que contienen los datos que va a ser accedidos. Debido a que los dos o más accesos a disco que se necesitan, en el peor de los casos, tardan varios órdenes de magnitud más de tiempo que el que requiere las referencias a memoria principal en el mejor caso, el tiempo de ejecución de transacción puede estimarse muy vagamente.

Los problemas de las bases de datos en tiempo real se encuentran en proceso investigación. Las notas bibliográficas hacen referencia a algunas investigaciones iniciales en este campo.

Capitulo 2. RECUPERACIÓN DE TRANSACCIONES EN BASES DE DATOS.

En este capítulo, se define la necesidad por la cual la recuperación de transacciones es indispensable, detallando algunos conceptos, los cuales son de suma importancia para el éxito de una transacción.

2. RECUPERACIÓN DE TRANSACCIONES EN BASES DE DATOS.

2.1. ¿Por qué es necesaria la recuperación?

Una transacción comienza con la ejecución satisfactoria de una instrucción BEGIN TRANSACTION y termina con la ejecución satisfactoria de una instrucción COMMIT o ROLLBACK. COMMIT establece lo que es conocido, entre muchas otras acepciones, como **punto de confirmación** (también se conoce como **punto de sincronización**, especialmente en productos comerciales) [7]. Un punto de confirmación corresponde entonces al final de una unidad de trabajo lógica y por lo tanto, a un punto en el cual la base de datos está o debería estar en un estado consistente. Por el contrario, ROLLBACK regresa la base de datos al estado en que estaba antes de BEGIN TRANSACTION, lo que en efecto significa regresar al punto de confirmación anterior. (La frase "punto de confirmación anterior" sigue siendo adecuada, aun en el caso de que sea la primera transacción en el programa; si coincidimos en que la primera BEGIN TRANSACTION del programa está estableciendo tácitamente un "punto de confirmación" inicial.)

Nota: A lo largo de este capítulo, el término "base de datos" significa en realidad sólo aquella parte de la base de datos a la que la transacción en consideración está teniendo acceso. Es posible que otras transacciones estén en ejecución paralela y hagan cambios a sus propias partes, por lo que también es posible que la "base de datos total" no esté en un estado completamente consistente en un punto de confirmación.

Al establecer un punto de confirmación:

1. Todas las actualizaciones hechas por el programa en ejecución desde el punto de confirmación anterior, son confirmadas; es decir, **se vuelven permanentes**. Antes del punto de confirmación, todas estas actualizaciones deben ser vistas como *meramente tentativas*, en el sentido de que pueden ser deshechas posteriormente. Es un hecho que una vez que una transacción ha sido confirmada nunca podrá ser deshecha (ésta es la definición de "confirmada").
2. Todo el posicionamiento de la base de datos se pierde y todos los bloqueos de tuplas son liberados. El "posicionamiento de la base de datos" se refiere en este caso a la idea de que en cualquier momento, un programa en ejecución tendrá una direccionalidad hacia ciertas tuplas (por ejemplo, mediante determinados *cursores* en el caso de SQL); esta direccionalidad se pierde en un punto de confirmación ¹.

El párrafo 2 anterior, salvo el comentario acerca de la posible retención de alguna direccionalidad, y por tanto, la posible conservación de determinados bloqueos de tuplas, también se aplica cuando la transacción termina con ROLLBACK en vez de COMMIT. El párrafo 1, por supuesto, no presenta el mismo caso.

¹ *Nota:* Algunos sistemas proporcionan una opción por la cual el programa puede conservar la direccionalidad hacia determinadas tuplas (y por lo tanto, conservar determinados bloqueos de tuplas) entre una transacción y la siguiente.

Observe cuidadosamente que COMMIT y ROLLBACK terminan la *transacción* y no el programa. Por lo general, una sola ejecución del programa consta de una *secuencia* de varias transacciones ejecutándose una tras otra, como ilustra la figura 2.1.

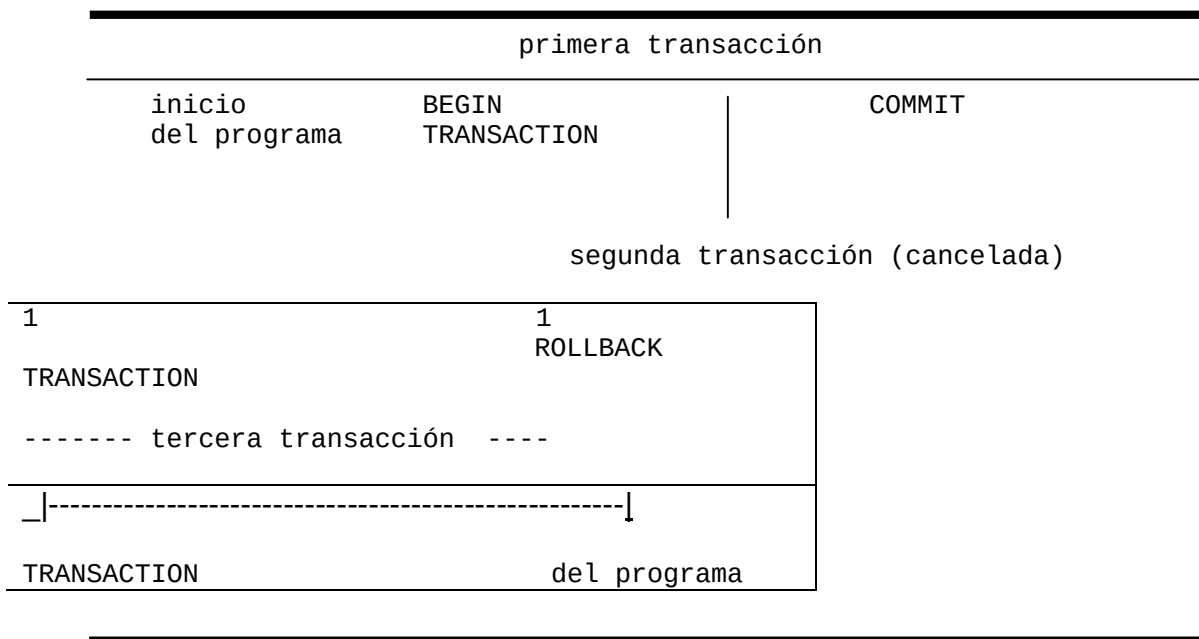


Figura 2.1 La ejecución del programa es una secuencia de transacciones.

En ese ejemplo incluimos pruebas explícitas para errores y emitimos una instrucción ROLLBACK explícita cuando fue detectado algún error. Pero el sistema por supuesto no puede dar por hecho que los programas de aplicación siempre incluirán pruebas explícitas para todos los errores posibles. Por lo tanto, el sistema emitirá una instrucción ROLLBACK *implícita* para cualquier transacción que falle por cualquier razón, para llegar a su terminación planeada (donde "terminación planeada" significa una instrucción COMMIT o ROLLBACK explícitas).

Ahora podemos ver que las transacciones no sólo representan la unidad de trabajo sino también la unidad de **recuperación**. Si una transacción es confirmada satisfactoriamente, el sistema garantizará que sus actualizaciones queden grabadas permanentemente en la base de datos, aunque el sistema llegue a fallar en el siguiente momento. Por ejemplo, es probable que el sistema falle después de haber recibido a COMMIT, pero antes de escribir físicamente las actualizaciones en la base de datos; éstas podrían seguir esperando en un búfer de memoria principal y por lo tanto, estar perdidas en el momento de la falla. Aunque esto suceda, el procedimiento de reinicio del sistema grabará esas actualizaciones en la base de datos. Éste es capaz de descubrir los valores que hay que escribir, examinando las entradas relevantes de la bitácora.

(De aquí se deduce que la bitácora debe ser escrita físicamente antes de que termine el procesamiento de COMMIT; lo que se conoce como **regla de escritura anticipada de la bitácora**). El procedimiento de reinicio recuperará cualquier transacción que haya terminado satisfactoriamente pero que no haya podido lograr

que sus actualizaciones fueran escritas físicamente antes de la caída. Por lo tanto, como dije anteriormente, las transacciones son la unidad de recuperación. *Nota:* En el siguiente capítulo veremos que también son la unidad de concurrencia. Además, puesto que supuestamente transforman un estado consistente hacia otro estado consistente de la base de datos, también pueden ser vistas como una unidad de *integridad*.

2.2. Tipos de fallos.

Hay varias razones por las que una transacción puede fallar mientras se está ejecutando [10 y 11]:

1. Un fallo del computador (caída): Un error de hardware o software ocurre en el sistema de cómputo durante la ejecución de una transacción. Si el equipo falla, es posible que se pierda el contenido de la memoria interna del computador.
2. Un error de la transacción o del sistema: Alguna operación de una transacción puede hacer que ésta falle, por ejemplo, un desbordamiento de enteros o una división entre ceros. También puede haber un fallo de transacción debido a valores erróneos de los parámetros o a un error lógico de programación. Por añadidura, puede suceder que el usuario interrumpa a propósito la transacción durante su ejecución; por ejemplo, al emitir control-C en un entorno VAX/VMS o UNIX.
3. Errores locales o condiciones de excepción detectadas por la transacción: Durante la ejecución de transacciones pueden presentarse ciertas condiciones que requieran la cancelación de la transacción. Por ejemplo, es posible que no se encuentren los datos para la transacción. Una condición, como un saldo insuficiente en una cuenta de una base de datos bancaria, puede hacer que se cancele una transacción, como un retiro de fondos de esa cuenta. Esto puede hacerse por una instrucción ABORTAR programada en la transacción misma.
4. Imposición del control de concurrencia: El método de control de concurrencia puede decidir que se aborte la transacción para reiniciarla después, porque viola la serializabilidad o porque varias transacciones se encuentran en un estado de bloqueo mortal.
5. Fallos del disco: Algunos bloques de disco pueden perder sus datos por un mal funcionamiento de lectura o de escritura o por un aterrizaje de una cabeza de lectura/escritura. Esto puede suceder durante una operación de lectura o de escritura de la transacción.
6. Problemas y catástrofes físicos: Esto se refiere a una interminable lista de problemas que incluyen interrupción del suministro de energía, fallo del acondicionamiento de aire, incendio, robo, sabotaje, sobre escritura en discos o cintas por error, y que el operador haya montado la cinta equivocada.

Los fallos 1, 2, 3, y 4 son más comunes que los fallos 5 y 6. Siempre que ocurre un fallo tipo del 1 al 4, el sistema debe mantener suficiente información para

recuperarse del fallo. Los fallos de disco o los catastróficos (tipos 5 ó 6) no ocurren con tanta frecuencia; si se dan, la recuperación es una tarea bastante complicada. El concepto de transacción atómica es fundamental para muchas técnicas de control de concurrencia y de recuperación de fallos.

Consideremos el siguiente ejemplo [7]. Para efectos de la ilustración, tenemos tres tablas: la tabla *S*, que representa a los proveedores; la tabla *P*, que representa a las partes y la tabla *SP*, que representa los envíos de partes hechos por los proveedores (figura 2.2)

S	S#	SNOMBRE	SITUACIÓN	CIUDAD	SP	S#	P#		
	CANT								
	S1	Salazar	20	Londres		S1	P1	300	
	S2	Jaimes	10	Paris		S1	P2	200	
	S3	Bernal	30	Paris		S1	P3	400	
	S4	Corona	20	Londres		S1	P4	200	
	S5	Aldana	30	Atenas		S1	P5	100	
						S1	P6	100	
						S2	P1	300	
P	P#	PNOMBRE	COLOR	PESO	CIUDAD		S2	P2	400
							S3	P3	200
	P1	Tuerca	Rojo	12	Londres		S4	P2	200
	P2	Perno	Verde	17	Paris		S4	P4	300
	P3	Birlo	Azul	17	Roma		S4	P5	400
	P4	Birlo	Rojo	14	Londres				
	P5	Leva	Azul	12	Paris				
	P6	Engrane	Rojo	19	Londres				

CANTTOTAL

CAMPO ADICIONAL

Figura 2.2 La base de datos de proveedores y partes.

La tabla *P*, la de partes contiene un campo adicional CANTTOTAL que representa la cantidad total enviada de la parte en cuestión; en otras palabras, el valor CANTTOTAL para una parte dada es igual a la suma de todos los valores SP.CANT de todos los registros *SP* correspondientes a esa parte. Consideremos ahora la siguiente secuencia de operaciones, cuya intención es añadir un nuevo envío (S5,P1,1000) a la base de datos:

```

EXEC SQL WHENEVER SQLERROR GO TO ANULAR ;
EXEC SQL INSERT
      INTO      SP ( S#, P#, CANT )
      VALUES ( 'S5', 'P1', 1000 ) ;
EXEC SQL UPDATE P
      SET CANTTOTAL = CANTTOTAL + 1000
      WHERE P# = 'P1' ;
EXEC SQL COMMIT ;
GO TO TERMINAR;

ANULAR:

EXEC SQL ROLLBACK ;

TERMINAR: RETURN ;

```

La proposición INSERT (insertar) agrega el nuevo envío a la tabla **SP**, la proposición UPDATE (actualizar) pone al día en forma apropiada el campo CANTTOTAL de la parte **P1**.

El objetivo de este ejemplo es mostrar que una operación supuestamente individual y atómica, “añadir un nuevo envío”, implica en realidad dos modificaciones de la base de datos. Por añadidura, la base de datos ni siquiera es consistente entre esas dos modificaciones; viola en forma temporal el requerimiento según el cual el valor de CANTTOTAL para la parte **P1** debe ser igual a la suma de todos los valores SP.CANT correspondiente a esa parte. Así pues, una unidad lógica de trabajo (o sea, una transacción) no es por fuerza una sola operación en la base de datos; más bien, es en general una secuencia de varias de esas operaciones mediante la cual un estado consistente de la base de datos se transforma en otro estado consistente, sin conservar por fuerza la consistencia en todos los puntos intermedios.

Ahora bien, es evidente que lo que no puede permitirse en el ejemplo es la ejecución de una de las dos modificaciones y no de la otra (porque en ese caso la base de datos quedaría en un estado inconsciente). En una situación ideal, desde luego, nos gustaría tener una garantía absoluta de que se realizaran las dos modificaciones. Por desgracia, es imposible ofrecer tal garantía; siempre existe la posibilidad de una falla y, además de una falla en el peor momento posible. Por ejemplo, el sistema podría caerse justo después de la primera modificación, o podría haber un desborde aritmético en la segunda, etcétera. Pero el sistema maneja el *procesamiento de transacciones*, podrá ofrecer lo más cercano a esa garantía. En términos específicos, garantizará que si la transacción ejecuta algunas modificaciones y después se presenta una falla (por cualquier razón) antes de que llegue el término normal de la transacción, se *anularán esas modificaciones*. Así, o bien la transacción se lleva a cabo en su totalidad (es decir, se hace como si jamás se hubiera ejecutado). De esta manera, puede lograrse que una secuencia de operaciones, la cual en esencia no es atómica, aparente serlo desde un punto de vista externo.

El componente del sistema encargado de lograr esta atomicidad (o apariencia de atomicidad) se conoce como *manejador de transacciones*, y las operaciones COMMIT (comprometer) y ROLLBACK (retroceder) son la clave de su funcionamiento:

- La operación COMMIT señala el término exitoso de la transacción: le dice al manejador de transacciones que se ha finalizado con éxito una unidad lógica de trabajo, que la base de datos está (o debería estar) de nuevo en un estado consistente, y que se puede “comprometer”, o hacer permanentes, todas las modificaciones efectuadas por esa unidad de trabajo.
- La operación ROLLBACK, en cambio, señala el término no exitoso de la transacción: dice al manejador de transacciones que algo salió mal, que la base de datos podría estar en un estado inconsistente, y que todas las modificaciones efectuadas hasta el momento por la unidad lógica de trabajo deben “retroceder” o anularse.

En el ejemplo, entonces emitiremos un COMMIT si realizamos con éxito las dos modificaciones, con lo cual se comprometerían las alteraciones en la base de datos y se volverán permanentes. En cambio, si algo sale mal, o sea, si cualquiera de las dos proposiciones de modificación produce la condición SQLERROR, emitiremos un ROLLBACK, a fin de anular todas las modificaciones efectuadas hasta ese momento.

Nota 1: Hemos mostrado de manera explícita las operaciones COMMIT y ROLLBACK, para efectos del ejemplo. Sin embargo, en el caso específico del DB2 emitirá automáticamente una instrucción COMMIT cuando termine en forma normal cualquier programa, y emitirá también automáticamente una instrucción ROLLBACK cuando cualquier programa no termine en forma normal (sea cual sea la razón; en particular, si un programa termina de manera anormal debido a una falla del *sistema*, emitirá un ROLLBACK por su cuenta cuando se reinicie el sistema). Por ejemplo, por tanto, podríamos haber omitido la instrucción COMMIT explícita, pero no la instrucción ROLLBACK explícita.

Nota 2: Un programa de explicación real no sólo deberá modificar la base de datos (o intentar modificarla) sino además enviar algún tipo de mensaje de vuelta al usuario final indicando lo sucedido. En el ejemplo, podríamos enviar el mensaje “Envío agregado” si se llega a la instrucción COMMIT, o el mensaje “Error: no se agregó el envío” en caso contrario. El manejo de mensajes, a su vez, tiene implicaciones adicionales en cuanto a la recuperación.

En este punto, probablemente se pregunte cómo es posible anular una modificación. La respuesta es que el sistema mantiene una *bitácora* o diario en cinta o (más comúnmente) en disco, en los cuales se registran los detalles de todas las operaciones de actualización, en particular, los valores inicial y final del objeto modificado. Por tanto, si resulta necesario anular alguna modificación específica, el sistema puede utilizar la entrada correspondiente de la bitácora para restaurar el valor original del objeto modificado.

Un punto adicional: en un sistema relacional, desde luego, las proposiciones de manipulación de datos se encuentran en el nivel de conjuntos y por lo regular trabajan con varios registros a la vez. ¿Qué sucedería si hay alguna falla durante la ejecución de una proposición de éstas? Por ejemplo, ¿es posible que una actualización (UPDATE) de múltiples registros ponga al día varios de sus registros objetivos y después falle antes de poner al día el resto? La respuesta, al menos en DB2, es que no es posible; las proposiciones individuales de SQL deben ser por fuerza atómicas en lo concerniente a su efecto sobre la base de datos. Si se llega a presentar un error durante la ejecución de estas proposiciones, la base de datos no sufrirá alteración alguna.

2.3. Recuperación del sistema y los medios de almacenamiento.

El sistema debe estar preparado para recuperarse no sólo de fallas puramente locales, como la aparición de una condición de desborde dentro de una sola transacción, sino también de fallas “globales” como podría ser una interrupción del suministro eléctrico en la UCP [7]. Una falla, por definición, afecta sólo la transacción en la cual se presentó esa falla. Una falla global, en cambio, afecta a

varias y con mucha probabilidad a la totalidad, de las transacciones que se estaban efectuando en el momento de la falla, por lo cual tendrá implicaciones para todo el sistema.

Tales fallas se dividen en dos categorías amplias:

- **Fallas del sistema** (por ejemplo, interrupción del suministro de electricidad), las cuales afectan a todas las transacciones que se están realizando pero no dañan físicamente a la base de datos. Las fallas del sistema se conocen también como *caídas suaves*.
- **Fallas de los medios de almacenamiento** (por ejemplo, un aterrizaje de cabezas en el disco), las cuales si causan daños a la base de datos, o a una porción de ella, y afectan al menos a las transacciones que están utilizando esa porción. Las fallas de los medios de almacenamiento se denominan a veces *caídas duras*.

2.4. Falla del sistema.

El punto crítico en lo tocante a una falla del sistema es que se pierde *el contenido de la memoria principal*, en particular, se pierden las áreas de almacenamiento temporal (buffers) de datos [7]. Por tanto, ya no se conocerá el estado preciso de la transacción que se estuviera realizando en el momento de la falla; esa transacción jamás se podrá completar con éxito, por lo cual será preciso *anularla* (retroceder) cuando se reinicie el sistema. Por añadidura, podría ser necesario en el momento de reiniciar *volver a realizar* ciertas transacciones cuya ejecución si terminó con éxito antes de la caída pero cuyas modificaciones no lograron ser transferidas de los buffers de la base de datos a la base de datos física.

Surge la pregunta ¿Cómo sabe el sistema en el momento de reiniciar cuales transacciones debe anular y cuales debe realizar otra vez? La respuesta es la siguiente. Cada cierto intervalo previamente establecido, por lo regular cuando se ha grabado un número especificado de entradas en la bitácora, el sistema “establece un punto de revisión” de manera automática. El establecimiento de un punto de revisión implica: *a)* grabar físicamente el contenido de los buffers de datos en la base de datos física y *b)* grabar físicamente un *registro de punto de revisión* especial en la bitácora física. El registro de punto de revisión incluye una lista de todas las transacciones que se estaban realizando en el momento de establecerse el punto de revisión. Para comprender la forma como se utiliza esta información, examínese la *figura 2.3*, la cual deberá leerse de la siguiente manera:

- Se presentó una falla de sistema en el momento *tf*.
- El punto de verificación más reciente antes de *tf* se tomó en el momento *tv*.
- Las transacciones del tipo *T1* se completaron antes del tiempo *tv*.
- Las transacciones del tipo *T2* se iniciaron antes del tiempo *tv* y se completaron después del tiempo *tv* y antes del tiempo *tf*.
- Las transacciones del tipo *T3* también se iniciaron antes del tiempo *tv* pero no se completaron antes del tiempo *tf*.
- Las transacciones del tipo *T4* se iniciaron después del tiempo *tv* y terminaron (satisfactoriamente) antes del tiempo *tf*.

- Por último, las transacciones del tipo T_5 también se iniciaron después del tiempo t_v pero no se completaron antes del tiempo t_f .

Deberá ser evidente que al reiniciarse el sistema deberán anularse las transacciones de los tipos T_3 y T_5 y deberán realizarse de nuevo las transacciones de los tipos T_2 y T_4 . Adviértase, empero, que las transacciones del tipo T_1 no entran en absoluto en el proceso de reinicio, porque sus modificaciones se grabaron físicamente en la base de datos en el momento t_v como parte del proceso de punto de revisión. Por tanto, en el momento del reinicio el sistema efectúa el siguiente procedimiento a fin de identificar las transacciones de los tipos T_2 - T_5 :

1. Comenzar con dos listas de transacciones, la lista ANULAR y la lista REPETIR. Igualar la lista ANULAR a la lista de todas las transacciones incluidas en el registro de punto de revisión. Dejar vacía la lista REPETIR.
2. Examinar la bitácora hacia delante a partir del registro de punto de revisión.
3. Si se encuentra una entrada de bitácora de “iniciar transacción” para la transacción T , añadir T a la lista ANULAR.

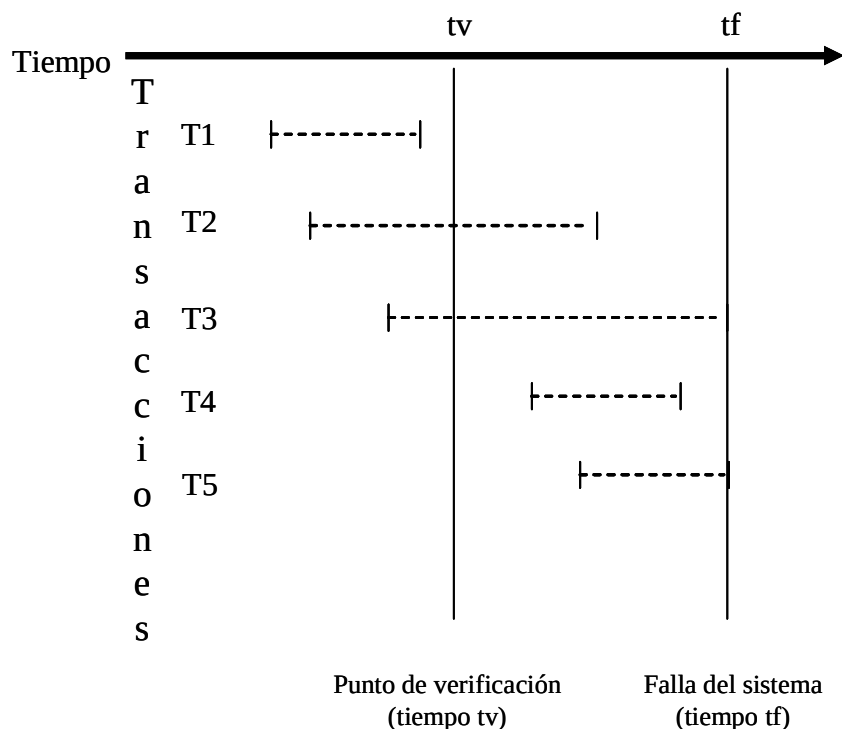


Figura 2.3. Cinco categorías de transacciones.

4. Si se encuentra una entrada de bitácora de “comprometer” para la transacción T , pasar esa transacción de la lista ANULAR a la lista REPETIR.
5. Cuando se llegue al final de la bitácora, las listas ANULAR y REPETIR identificarán, respectivamente, las transacciones de los tipos T_3 y T_5 y de los tipos T_2 y T_4 .

Enseguida el sistema revisará la bitácora hacia atrás, anulando todas las transacciones de la lista ANULAR. A continuación la revisará otra vez hacia delante, realizando de nuevo todas las transacciones en la lista REPETIR. Por último, una vez terminada toda esta actividad de recuperación (y sólo entonces), el sistema estará listo para aceptar trabajos nuevos ¹.

2.5. Falla de los medios de almacenamiento.

Una falla de los medios de almacenamiento es un percance (como por ejemplo un aterrizaje de las cabezas del disco o una falla del controlador del disco) en el cual se destruye físicamente alguna porción de la base de datos [7]. La recuperación de una falla semejante implica en esencia cargar de nuevo (*o restaurar*) la base de datos a partir de una copia de respaldo (*o vaciado*) y utilizar después la bitácora tanto la porción activa como la de archivo, en general para realizar de nuevo todas las transacciones terminadas desde que se hizo esa copia de respaldo. No hay necesidad de anular las transacciones inconclusas en el momento de la falla, porque por definición todas las modificaciones de esas transacciones ya se “anularon” (destruyeron) de todas maneras.

La necesidad de poder realizar una recuperación de medios de almacenamiento implica la necesidad de una utilería de *vaciado/restauración* (o de *descarga/recarga*). La porción de vaciado de una utilería como ésta sirve para crear copias de respaldo de la base de datos cuando sea necesario. (Estas copias pueden guardarse en cinta u otro medio de almacenamiento de archivo; no es necesario mantenerlas en medios de acceso directo). La parte de restauración de la utilería servirá entonces para recrear la base de datos después de una falla de los medios de almacenamiento a partir de una copia de respaldo especificada.

2.6. Confirmación de dos fases.

En esta sección analizaremos de manera breve una derivación muy importante del concepto básico de compromiso/retroceso, llamada *compromiso en dos fases* [7]. El compromiso en dos fases es importante cuando una transacción dada puede interactuar con varios “administradores de recursos” independientes, cada uno de los cuales administra su propio conjunto de recursos recuperables. Por ejemplo, consideremos una transacción que pone al día tanto la base de datos IMS como una base de datos DB2, si la transacción se completa con éxito, todas sus modificaciones se deberán anular (dicho de otro modo, no deberá ser posible que se comprometan las modificaciones en IMS y se anulen las modificaciones en DB2, o viceversa, pues en ese caso la transacción ya no sería atómica).

En consecuencia, no tendrá sentido que la transacción emita, digamos, una instrucción COMMIT a IMS y una instrucción ROLLBACK a DB2; y aún cuando emitiera la misma instrucción a los dos, existiría la posibilidad de una caída del sistema entre las dos, con resultados lamentables. Así pues, en vez de ello, la transacción emite una sola instrucción COMMIT (o ROLLBACK) *para todo el* -

¹ Notará que nuestra descripción del proceso de recuperación del sistema está muy simplificada. En particular, muestra al sistema realizando primero operaciones de “deshacer” y luego operaciones de “rehacer”. Los primeros sistemas funcionaban de esa forma, pero por razones de eficiencia los sistemas modernos generalmente hacen las cosas al revés.

sistema. Esa instrucción es manejada por un componente del sistema llamado *coordinador*, cuya tarea es garantizar que los dos administradores de recursos (o sea, IMS y DB2, en nuestro ejemplo) comprometan o anulen al *unísono* las modificaciones de las cuales son responsables, y además hacer valerosa garantía *aún si el sistema falla durante el proceso*. Y es el protocolo de compromiso en dos fases el que le permite ofrecer esa garantía.

Veamos cómo funciona. Vamos a suponer por sencillez que la transacción termina con éxito, de modo que la operación para todo el sistema es COMMIT, no ROLLBACK.

Al recibir esa solicitud de comprometer, el coordinador realiza el siguiente proceso de dos fases:

1. Primero, ordena a todos los administradores de recursos que se preparen para cualquiera de las dos posibilidades con respecto a la transacción. En la práctica, esto significa que cada uno de los administradores de recursos debe forzar la grabación de todas las entradas de la bitácora correspondientes a los recursos locales empleados por la transacción en su propia bitácora física (es decir, en almacenamiento no volátil; suceda lo que suceda después, el manejador de recursos tendrá ya un *registro permanente* de lo que hizo con respecto a la transacción, y así podrá comprometer o anular sus modificaciones, según sea necesario). Si la grabación forzada tiene éxito, el manejador de recursos contestará “todo bien” al coordinador. En caso contrario, contestará “no está bien”.
2. Cuando el coordinador haya recibido respuesta de todos los manejadores de recursos, forzará la grabación de una entrada en su *propia* bitácora física, para registrar su decisión en cuanto a la transacción. Si todas las respuestas fueron “todo bien”, la decisión será “comprometer”; si cualquiera de las respuestas fue “no está bien”, la decisión será “retroceder”. En cualquiera de los dos casos, el coordinador informará de su decisión a todos los administradores de recursos, y *cada administrador de recursos deberá comprometer o anular la transacción localmente, según se le haya ordenado*. Adviértase que cada administrador de recursos debe hacer lo que ordena el coordinador en la fase 2; ése es el protocolo. (Obsérvese también, por cierto, que es la aparición del registro de decisión en la bitácora física del coordinador lo que marca la transacción de la fase 1 a la fase 2).

Ahora bien, si el sistema falla en algún punto durante el proceso en general, el procedimiento se reinicia buscará el registro de decisión en la bitácora del coordinador. Si lo encuentra, el proceso de compromiso en dos fases podrá continuar donde fue interrumpido. Si no lo encuentra, supondrá que la decisión fue “retroceder”, con lo cual el proceso se podrá completar en la forma apropiada.

Cabe señalar que el administrador de comunicación de datos (administrador DC) se puede considerar como un administrador de recursos en el sentido que se le acaba de dar. Es decir, los mensajes pueden considerarse como un recurso recuperable, igual que los registros de base de datos, y el administrador DC necesita ser capaz de participar en el proceso de compromiso en dos fases.

Capitulo 3. CONTROL DE CONCURRENCIA.

En el capítulo, se exponen los principales conceptos para el control de concurrencia, así como también la necesidad que existe del mismo, para un adecuado manejo de transacciones en un ambiente de múltiples usuarios.

3. CONTROL DE CONCURRENCIA.

3.1 ¿Por qué es necesario el control de concurrencia?

Uno de los conceptos más importantes de los sistemas modernos es, sin duda, la *multiprogramación* [7]. Teniendo varias transacciones ejecutándose al mismo tiempo, puede compartirse el procesador entre ellas. Los beneficios de la multiprogramación son un mejor aprovechamiento del procesador y una productividad total de transacciones más alta, es decir, la cantidad de trabajo que se realiza en un intervalo de tiempo dado. Además, en el caso de transacciones interactivas en las que el usuario espera el resultado, el tiempo de respuesta debe ser lo más corto posible.

Así pues, en un entorno de multiprogramación es posible ejecutar varias transacciones de manera concurrente. Como se verá más adelante, es necesario que el sistema controle la interacción entre las transacciones concurrentes para evitar que destruyan la consistencia de la base de datos. El control se logra por medio de varios mecanismos a los que nos referiremos como esquema de **control de concurrencia**.

Cuando se ejecutan varias transacciones de manera concurrente, la consistencia de la base de datos puede destruirse aún cuando cada una de las transacciones individuales sea correcta. En esta sección presentamos el concepto de *serializabilidad* para ayudar a identificar las ejecuciones que se garantiza que aseguran la consistencia.

Supondremos que la variable local temporal tiene el mismo nombre que la del dato al que se está accediendo. Así en el resto de esta sección, usaremos $\text{read}(Q)$ y $\text{write}(Q)$ para $\text{read}(Q,q)$ y $\text{write}(Q,q)$

3.2 Planificaciones en serie y en paralelo.

Considérese el sistema bancario simplificado con varias cuentas y un conjunto de transacciones que acceden y actualizan esas cuentas [7]. Sean T_0 y T_1 dos cuentas que transfieren fondos de una cuenta a otra. La transacción T_0 transfiere 50 dólares de la cuenta A a la cuenta B y se define así:

```
 $T_0$ :  read(A);  
      A:= A -50;  
      write(A);  
      read(B);  
      B:=B+50;  
      write(B).
```

La transacción T_1 transfiere el 10 por 100 del saldo de la cuenta A a la cuenta B y se define así:

```

T1:  read(A);
      temp:=A*0.1;
      A:=A-temp;
      write(A);
      read(B);
      B:=B+temp;
      write(B).

```

Sean los valores actuales de las cuentas **A** y **B**, 1000 dólares y 2000 dólares, respectivamente. Supongamos que las dos transacciones se ejecutan de una en una en el orden **T0** seguida de **T1**. Esta secuencia de ejecución está representada en la figura 3.2. En la figura, la secuencia de pasos de instrucción está en orden cronológico de arriba abajo, apareciendo las instrucciones de **T0** en la columna izquierda y las instrucciones de **T1** en la columna derecha. Los valores finales de las cuentas **A** y **B** después de la ejecución de la figura 3.2, son 855 dólares y 2,145 dólares, respectivamente. Así la cantidad total de dinero en la cuentas **A** y **B**, es decir, la suma de **A+B**, se conserva después de la ejecución de ambas transacciones.

T0	T1
read(A)	
A:= A -50	
write(A)	
read(B)	
B:= B +50	
write(B)	
	read(A)
	temp:= A *0.1
	A:= A -temp
	write(A)
	read(B)
	B:= B +temp
	write(B)

Figura 3.2 Planificación 1, planificación en serie en la que **T0** va seguida de **T1**.

De manera similar, si las transacciones se ejecutan de una en una en el orden **T1** seguida de **T0**, entonces la secuencia de ejecución correspondiente es la de la figura 3.2.1. Una vez más, como era de esperar, se conserva la suma **A+B**, y los valores finales de las cuentas **A** y **B** son 850 dólares y 2,150 dólares, respectivamente.

Las secuencias de ejecución descritas anteriormente se llaman *planificaciones* y representan el orden cronológico en que se ejecutan las instrucciones en el sistema. Es evidente que una planificación para un conjunto de transacciones debe constar de todas las instrucciones de esas transacciones y debe conservar el orden en que aparecen las instrucciones en cada transacción individual. Por ejemplo, en la transacción **T0**, la instrucción **write(A)** debe aparecer antes de la instrucción **read(B)** en cualquier planificación válida. En el siguiente estudio llamaremos a la primera secuencia de ejecución (**T0** seguida de **T1**) planificación 1 y a la segunda secuencia de ejecución (**T1** seguida de **T0**) planificación 2.

<i>T0</i>	<i>T1</i>
read(<i>A</i>) $A := A - 50$ write(<i>A</i>) read(<i>B</i>) $B := B + 50$ write(<i>B</i>)	read(<i>A</i>) $temp := A * 0.1$ $A := A - temp$ write(<i>A</i>) read(<i>B</i>) $B := B + temp$ write(<i>B</i>)

Figura 3.2.1 Planificación 2, planificación en serie en la que *T1* va seguida de *T0*.

Las planificaciones descritas arriba se denominan *planificaciones en serie*. Cada una de ellas consta de una secuencia de instrucciones de varias transacciones en las que las instrucciones pertenecientes a una única instrucción aparecen juntas en esa planificación. Así, para un conjunto de *n* transacciones existen *n!* planificaciones en serie diferentes.

Cuando se ejecutan varias transacciones concurrentemente, no es preciso que la planificación correspondiente esté en serie. Así, el número de planificaciones posibles para un conjunto de *n* transacciones es mucho mayor que *n!* Volviendo al ejemplo anterior, supóngase que las dos transacciones se ejecutan de manera concurrente. Existen varias secuencias de ejecución posibles, ya que ahora pueden irse cambiando varias instrucciones de ambas transacciones. Una de las posibles planificaciones se muestra en la *figura 3.2.2*. Después de llevarse a cabo esta ejecución, llegamos al mismo estado que al que se llega en el caso en que las transacciones se ejecuten en serie en el orden *T0* seguida de *T1*. La suma *A+B* sí se conserva.

<i>T0</i>	<i>T1</i>
read(<i>A</i>) $A := A - 50$ write(<i>A</i>)	read(<i>A</i>) $temp := A * 0.1$ $A := A - temp$ write(<i>A</i>)
read(<i>B</i>) $B := B + 50$ write(<i>B</i>)	read(<i>B</i>) $B := B + temp$ write(<i>B</i>)

Figura 3.2.2 Planificación 3, planificación serializable concurrente.

No todas las ejecuciones concurrentes resultan en un estado incorrecto. Para ilustrarlo considérese la planificación en paralelo de la *figura 3.2.3*. Después de la ejecución de esta planificación llegamos a un estado en el que los valores finales de las cuentas *A* y *B* son 950 dólares y 2,100 dólares, respectivamente. Este estado final es un estado *inconsistente*, ya que hemos ganado 50 dólares en el proceso de la ejecución concurrente. De hecho, la ejecución de las dos transacciones no conserva la suma $A+B$.

Es necesario que una transacción sea un programa que conserve la consistencia. Es decir, cada transacción, al ejecutarse sola, transfiere el sistema de un estado consistente a un nuevo estado consistente. Sin embargo, durante la ejecución de una transacción puede que el sistema entre temporalmente en un estado inconsistente. La inconsistencia temporal es la que causa la inconsistencia en las planificaciones en paralelo.

Está claro que una planificación después de su ejecución debe dejar la base de datos en un estado consistente. Además, el efecto de una planificación pudiera haber ocurrido sin ninguna ejecución concurrente. Es decir, la planificación debe ser, de alguna forma, equivalente a una planificación en serie. Para precisar más este concepto necesitamos definir la equivalencia de planificaciones.

<i>T0</i>	<i>T1</i>
read(<i>A</i>) $A:=A-50$	read(<i>A</i>) $temp:=A*0.1$ $A:=A-temp$ write(<i>A</i>) read(<i>B</i>)
write(<i>A</i>) read(<i>B</i>) $B:=B+50$ write(<i>B</i>)	 $B:=B+temp$ write(<i>B</i>)

Figura 3.2.3 Planificación 4, planificación no serializable.

Consideramos solamente dos operaciones **read** y **write**. Así, suponemos que entre una instrucción **read(Q)** y una instrucción **write(Q)** sobre un dato *Q*, una transacción puede realizar una secuencia arbitraria de operaciones sobre el valor de *Q*. Así, las únicas operaciones significativas de una transacción, desde el punto de vista de una planificación, son las instrucciones **read** y **write**. Debido a esto, normalmente sólo mostraremos instrucciones **read** y **write** en las planificaciones, como en la representación de la planificación 5 que se muestra en la *figura 3.2.4*.

T_0	T_1
read(A) write(A)	
	read(A) write(A)
read(B) write(B)	
	read(B) write(B)

Figura 3.2.4 Planificación 5, mostrando solamente las instrucciones read y write.

3.3 Protocolos basados en bloqueo.

Una forma de asegurar la serializabilidad es exigir que el acceso a los datos se haga de manera mutuamente excluyente [7]; es decir, que mientras una transacción accede a un dato, ninguna otra puede modificarlo. El método más común que se utiliza para implementar esto, es permitir que una transacción acceda a un dato sólo si tiene actualmente un bloqueo en ese dato.

Bloqueo.

Existen varios modos de bloquear un dato. En esta sección restringiremos la atención a dos modos:

- **Compartido.** Si una transacción T_i , ha obtenido un bloqueo de modo compartido (representado por 5) en el dato Q , entonces T_i puede leer este dato, pero no puede escribir Q .
- **Exclusivo.** Si una transacción T_i , ha obtenido un bloqueo de modo exclusivo (representado por X) en el dato Q , entonces T_i puede leer y escribir Q .

Exigimos que todas las transacciones pidan un bloqueo en el modo adecuado en el dato Q dependiendo del tipo de operaciones que van a realizar sobre Q .

Dado un conjunto de modos de bloqueo, podemos definir una *función de compatibilidad* en ellos de la siguiente manera. Sean A y B modos de bloqueo arbitrarios. Supóngase que una transacción T_i solicita un bloqueo de modo A en el dato Q en el cual la transacción $T_j (T_i \neq T_j)$ tiene actualmente un bloqueo de modo B . Si puede concedérsele a la transacción T_i un bloqueo en Q inmediatamente, a pesar de la presencia del bloqueo de modo B , entonces decimos que el modo A es compatible con el modo B . Este tipo de funciones puede representarse de forma conveniente por medio de una matriz. La relación de compatibilidad entre los dos modos de bloqueo que se utilizan en esta sección esta dada por la matriz **comp** de la figura 3.3. Un elemento **comp**(A, B) de la matriz tiene el valor verdadero si, y sólo si, el modo A es compatible con el modo B ¹.

¹ Nótese que el modo compartido es compatible con el modo compartido, pero no con el modo exclusivo.

En un momento dado, transacciones diferentes pueden tener en forma simultánea varios bloqueos de modo compartido en un dato determinado. Una posterior solicitud de bloqueo de modo exclusivo tiene que esperar a que se liberen los bloqueos de modo compartido que existen.

	S	X
S	Verdadero	Falso
X	Falso	Falso

Figura 3.3 Matriz **comp** de compatibilidad de bloqueo.

Una transacción solicita un bloqueo compartido en el dato Q ejecutando la instrucción **lock-S(Q)**. De manera similar, un bloqueo exclusivo se solicita por medio de la instrucción **lock-N(Q)**. Un dato (Q) puede desbloquearse por medio de la instrucción **unlock(Q)**.

Para acceder a un dato, la transacción T_i primero debe bloquearlo. Si un dato ya está bloqueado por otra transacción en un modo incompatible, entonces T_i debe esperar hasta que se liberen todos los bloqueos incompatibles que tengan otras transacciones. La transacción T_i puede desbloquear un dato al que había bloqueado con anterioridad.

Cabe hacer notar que una transacción debe tener un bloqueo en un dato mientras esté accediendo a él. Además, no siempre es deseable que una transacción desbloquee un dato inmediatamente después de su último acceso a ese dato, ya que existe la posibilidad de que no puede garantizarse la serializabilidad.

Para ilustrarlo, considérese de nuevo el sistema bancario simplificado. Sean A y B dos cuentas a las que acceden las transacciones T_7 y T_8 . La transacción T_7 transfiere 50 dólares de la cuenta B a la cuenta A , y se define así:

```

T7:  lock-X(B);
      read(B);
      B:=B-50;
      write(B);
      unlock(B);
      lock-X(A)
      read(A);
      A:=A+50;
      write(A);
      unlock(A).

```

La transacción T_8 presenta la cantidad total de dinero en las cuentas A y B , es decir, la suma $A+B$, y se define así:

```

T8:  lock-S(A);
      read(A);
      unlock(A);
      lock-S(B)
      read(B);
      unlock(B);
      display(A+B).

```

Supóngase que los valores de las cuentas **A** y **B** son 100 dólares y 200 dólares, respectivamente. Si estas dos transacciones se ejecutan en serie, ya sea en el orden **T7**, **T8** o **T8**, **T7**, entonces la transacción **T8** presentará el valor 300 dólares. Sin embargo, si estas transacciones se ejecutan de manera concurrente, entonces es posible tener la planificación 11 que se muestra en la *figura 3.3.1*. En este caso, la transacción **T8** presenta 250 dólares que es incorrecta. La razón de que esto sea así es porque la transacción **T7** desbloqueó el dato **B** demasiado pronto, por lo que **T8** vio un estado inconsistente.

Supóngase ahora que se retrasa el desbloqueo hasta el final de la transacción. La transacción **T9** de abajo corresponde a **T7** con el desbloqueo retrasado y se define así:

T9: lock-**X**(**B**);
 read(**B**);
B:=**B**-50;
 write(**B**);
 lock-**X**(**A**)
 read(**A**);
A:=**A**+50;
 write(**A**);
 unlock(**B**);
 unlock(**A**).

T7	T8
lock- X (B) read(B) B := B -50 write(B) unlock(B)	
	lock- D (A) read(A) unlock(A) write(B) lock- S (B) read(B) unlock(B) Display(A + B)
lock- X (A) read(A) A := A -50 write(A) unlock(A)	

Figura 3.3.1 Planificación 11

La transacción T_{10} de abajo corresponde a T_8 con el desbloqueo retrasado y se define así:

T_{10} : lock-S(A);
 read(A);
 lock-S(B)
 read(B);
 display($A+B$);
 unlock(A);
 unlock(B).

Considérese la planificación parcial de la *figura 3.3.2* para T_9 y T_{10} . Puesto que T_9 tiene un bloqueo de modo exclusivo en B y T_{10} pide un bloqueo de modo compartido en B , T_{10} espera a que T_9 desbloquee B . De manera similar, puesto que T tiene un bloqueo de modo compartido en A y T_9 pide un bloqueo de modo exclusivo en A , T_9 espera a que T_{10} desbloquee a A . Así, hemos llegado a un estado en el que ninguna de estas transacciones puede seguir su ejecución normal. Esta situación se denomina *bloqueo*. Cuando ocurre un bloqueo, el sistema debe retroceder una de las dos transacciones. Una vez que retrocede una de las dos transacciones, los datos que tenía bloqueados esa transacción se liberan. La otra transacción puede entonces disponer de esos datos y continuar su ejecución.

Lo que hemos señalado aquí es que el bloqueo debe utilizarse con cuidado. Si intentamos lograr un máximo de concurrencia desbloqueando datos tan pronto como sea posible, podemos obtener estados inconsistentes. Si no desbloqueamos un dato antes de solicitar un bloqueo en otro dato, pueden ocurrir situaciones de bloqueo.

T_9	T_{10}
lock-X(B)	
read(B)	
$B:=B-50$	
write(B)	
	lock-S(A)
	read(A)
	Lock-S(B)
lock-X(A)	

Figura 3.3.2 Planificación 12

Exigiremos que todas las transacciones del sistema sigan un conjunto de reglas, llamado *protocolo de bloqueo* ¹. Los protocolos de bloqueo restringen el número de planificaciones en un subconjunto propio de todas las planificaciones serializables.

¹ Que indica cuando una transacción puede bloquear y desbloquear cada uno de los datos.

Antes de hacerlo necesitamos introducir algunas definiciones:

Sea $\{T_9, T_i, \dots, T_j\}$ un conjunto de transacciones que participan en una planificación S . Decimos que T_i precede a T_j en S , escrito $T_i \rightarrow T_j$, si existe un dato Q tal que T_i haya tenido un modo de bloqueo A en Q y T_j haya tenido posteriormente un modo de bloqueo B en Q , y $\text{comp}(A, B) = \text{falso}$. Si $T_i \rightarrow T_j$, entonces esto implica que en cualquier planificación en serie equivalente, T_i debe aparecer antes de T_j . Los conflictos entre instrucciones corresponden a modos de bloqueo incompatibles.

Decimos que una planificación S es legal bajo un protocolo de bloqueo dado si s es una planificación posible para un conjunto de transacciones que sigue las reglas del protocolo del bloqueo. Decimos que un protocolo de bloqueo garantiza la serializabilidad en conflictos sólo si para todas las planificaciones legales la relación asociada es acíclica.

3.4 El protocolo de bloqueo de dos fases.

Un protocolo que garantiza la serializabilidad es el *protocolo de bloqueo de dos fases*¹ [7]. Este protocolo requiere que todas las transacciones hagan sus solicitudes de bloqueo y desbloqueo en dos fases:

- *Fase de crecimiento*. Una transacción puede obtener bloqueos pero no puede liberar ningún bloqueo.
- *Fase de encogimiento*. Una transacción puede liberar bloqueos pero no puede obtener ningún bloqueo nuevo.

Inicialmente, una transacción está en la fase de crecimiento. La transacción adquiere los bloqueos que necesita. Una vez que la transacción libere un bloqueo, entra en la fase de encogimiento y no podrá solicitar más bloqueos.

Por ejemplo, las transacciones T_9 y T_{10} son de dos fases. Por otra parte, las transacciones T_7 y T_8 no son de dos fases.

El protocolo de bloqueo de dos fases garantiza la serializabilidad en conflictos. Sin embargo, no garantiza que no se van a presentar situaciones de bloqueo. Para demostrar esto, obsérvese que las transacciones T_9 y T_{10} son de dos fases, pero en la planificación 12 (figura 2.4.2) están bloqueadas.

Si T_i no es una transacción de dos fases, siempre es posible encontrar otra transacción T_j que sea de dos fases, tal que exista una posible planificación no serializable en conflictos T_i y T_j .

Considérense las dos transacciones siguientes para las cuales hemos mostrado sólo algunas de las operaciones significativas de **read** y **write**. La transacción T_{11} se define así:

T_{11} : read($a1$);
 read($a2$);
 read(an);
 write($a1$).

¹ El **bloqueo de dos fases** no tiene nada que ver con la confirmación de dos fases, simplemente tienen nombres similares.

Mientras que la transacción T_{12} se define así:

T_{12} : read(a_1);
 read(a_2);
 display ($a_1 + a_2$).

Si empleamos el protocolo de bloqueo de dos fases, entonces T_{11} debe bloquear a a_1 en modo exclusivo. Por tanto, cualquier ejecución concurrente de ambas transacciones viene a ser una ejecución en serie. Sin embargo, obsérvese que T_{11} necesita un bloqueo exclusivo en a_1 sólo al final de su ejecución, cuando escribe a_1 . Así si T_{11} pudiera bloquear inicialmente a a_1 en modo compartido y después cambiar el bloqueo por uno de modo exclusivo, podríamos obtener mayor concurrencia, ya que T_{11} y T_{12} podrían acceder a a_1 y a_2 , simultáneamente.

Esta observación nos lleva a un refinamiento del protocolo de bloqueo básico de dos fases, en el que se permiten *conversiones*, de bloqueo. Proporcionaremos un mecanismo para elevar un bloqueo compartido a un bloqueo exclusivo y para bajar un bloqueo exclusivo a un bloqueo compartido. Denotamos la conversión de modos compartidos a exclusivos por **upgrade** y de exclusivos a compartidos por **downgrade**. La conversión de bloqueos no puede permitirse que ocurra de manera arbitraria. La elevación (compartido a exclusivo) sólo puede tener lugar en la fase de crecimiento, mientras que la conversión de modo exclusivo a compartido sólo puede tener lugar en la fase de encogimiento.

Volviendo al ejemplo, las transacciones T_{11} y T_{12} pueden ejecutarse concurrentemente bajo el protocolo de bloqueo de dos fases refinado, como se muestra en la planificación incompleta de la *figura 3.4*, donde sólo se muestran algunas de las instrucciones de bloqueo.

Obsérvese que una transacción que intenta elevar un bloqueo en el dato Q , puede verse obligada a esperar. Esto ocurre si otra transacción bloquea a Q en modo compartido.

T_{11}	T_{12}
lock- $S(a_1)$	
	lock- $S(a_1)$
lock- $S(a_2)$	
	lock- $S(a_2)$
lock- $S(a_3)$	
lock- $S(a_4)$	
	unlock(a_1)
	unlock(a_2)
lock- $S(a_n)$	
upgrade(a_1)	

Figura 3.4 Planificación incompleta con una conversión de bloqueo.

Ahora describimos un esquema sencillo que puede utilizarse para las instrucciones de bloqueo apropiadas para una transacción. Cuando una transacción T_1 ejecuta una operación **read(Q)**, el sistema ejecuta una instrucción **lock-S(Q)** seguida de la instrucción **read(Q)**.

Cuando T_1 ejecuta una operación **write(Q)**, el sistema comprueba si T_1 ya tiene un bloqueo compartido en Q . En ese caso, el sistema ejecuta una instrucción **upgrade(Q)** seguida de la instrucción **write(Q)**. En caso contrario, el sistema ejecuta una instrucción **lock-X(Q)** seguida de la instrucción **write(Q)**.

Si eso es posible, para un conjunto de transacciones existen planificaciones serializables en conflictos que no pueden obtenerse mediante el protocolo de bloqueo de dos fases. Sin embargo, para seguir un procedimiento mejor que el bloqueo de dos fases, necesitamos tener información adicional acerca de las transacciones, o imponer alguna estructura u ordenación sobre el conjunto de datos de la base de datos.

3.5 Protocolos basados en grafos.

Como se indicó anteriormente, en ausencia de información referente a la forma en que se accede a los datos, el protocolo de bloqueo de dos fases es necesario y suficiente para garantizar la serializabilidad. Por tanto, si deseamos desarrollar protocolos que no sean de dos fases, necesitamos tener información adicional acerca de como accederá cada transacción a la base de datos. Existen varios modelos que difieren en la cantidad de información proporcionada. El modelo más sencillo requiere que tengamos conocimiento previo del orden en que se va a acceder a los datos de la base de datos. Dada esa información, es posible construir protocolos de bloqueo que no sean de dos fases pero que de todos modos garanticen la serializabilidad.

Para adquirir ese conocimiento previo imponemos un ordenación parcial $\rightarrow$ en el conjunto $D=[d_1, d_2, \dots, d_h]$ de todos los datos. Si $d_i \rightarrow d_j$ entonces cualquier transacción que acceda tanto a d_i como a d_j debe acceder a d_i antes de acceder a d_j .

Esta ordenación parcial puede ser el resultado de la organización lógica o física de los datos, o puede imponerse únicamente con el propósito de controlar la concurrencia.

La ordenación parcial implica que el conjunto D puede verse ahora como un grafo acíclico dirigido, llamado *grafo de base de datos*. En esta sección, por sencillez, restringiremos la atención sólo a los grafos que sean árboles con raíz.

Presentaremos un protocolo simple, llamado protocolo de árbol, que está restringido al empleo sólo de bloqueos exclusivos.

En el protocolo de árbol, la única instrucción de bloqueo permitida es **lock-X**. Cada transacción T_i puede bloquear un dato a lo más una vez y debe observar las siguientes reglas:

1. El primer bloqueo de T_i puede ser en cualquier dato.
2. A partir de ese momento, T_i puede bloquear un dato Q sólo si T_i bloquea actualmente al padre de Q .
3. Los datos pueden bloquearse en cualquier momento.
4. Un dato que T_i haya bloqueado y desbloqueado, T_i no puede volver a bloquearlo posteriormente.

Como dijimos anteriormente, todas las planificaciones que sean legales bajo el protocolo de árbol son serializables en conflictos. Para ilustrar este protocolo, considérese el grafo de base de datos en la *figura 3.6*. Las cuatro transacciones siguientes siguen el protocolo de árbol de ese grafo. Hemos mostrado sólo las instrucciones de bloqueo y desbloqueo.

T_{13} : lock-X(**B**); lock-X(**E**); unlock(**E**); lock-X(**D**);
 unlock(**B**); lock-X(**G**); unlock(**D**); unlock(**G**).
 T_{14} : lock-X(**D**); lock-X(**H**); unlock(**D**); lock-X(**J**); unlock(**J**);
 unlock(**H**).
 T_{15} : lock-X(**B**); lock-X(**E**); unlock(**E**); unlock(**B**).
 T_{16} : lock-X(**D**); lock-X(**H**); unlock(**D**); unlock(**H**).

Una planificación posible en la que participan estas cuatro transacciones está representada en la *figura 3.5.1*. Obsérvese que durante su ejecución la transacción T_{13} tiene bloqueos en dos árboles *disjuntos*.

Obsérvese que la planificación de la *figura 3.6.1* es serializable ¹ en conflictos. Puede demostrarse que el protocolo de árbol garantiza no sólo la serializabilidad, sino también la libertad de desbloqueo.

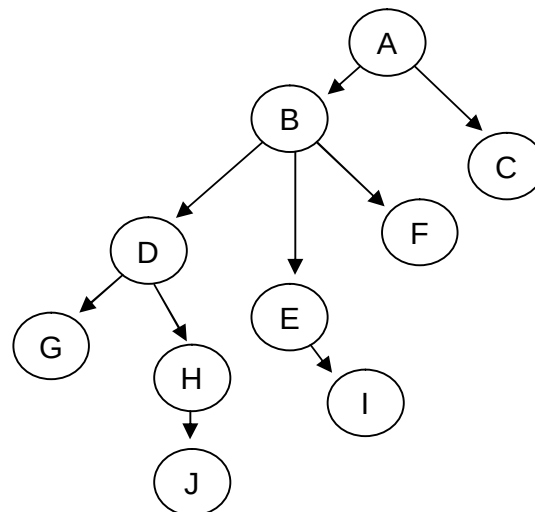


Figura 3.5 Grafo de base de datos con estructura de árbol.

¹ SERIALIZABLE no resulta una buena palabra reservada en este caso, ya que se supone que los planes son los serializables y no las transacciones. Un mejor término podría ser simplemente TWO PHASE, que significa que la transacción obedecerá (o será obligada a obedecer) el protocolo de bloqueo de dos fases

<i>T13</i>	<i>T14</i>	<i>T15</i>	<i>T16</i>
lock- <i>X</i> (<i>B</i>)			
	lock- <i>X</i> (<i>D</i>) lock- <i>X</i> (<i>H</i>) unlock(<i>D</i>)		
lock- <i>X</i> (<i>E</i>) unlock(<i>E</i>) lock- <i>X</i> (<i>D</i>) unlock(<i>B</i>)			
	unlock(<i>H</i>)	lock- <i>X</i> (<i>B</i>) lock- <i>X</i> (<i>E</i>)	
lock- <i>X</i> (<i>G</i>) unlock(<i>D</i>)			lock- <i>X</i> (<i>D</i>) lock- <i>X</i> (<i>H</i>) Unlock(<i>D</i>) unlock(<i>H</i>)
		unlock(<i>E</i>) unlock(<i>B</i>)	
unlock(<i>G</i>)			

Figura 3.5.1 Planificación serializable.

El protocolo de bloqueo de árbol tiene la ventaja, con respecto al protocolo de bloqueo de dos fases de que el desbloqueo puede ocurrir antes. Esto puede resultar en menos tiempo de espera y en un aumento de la concurrencia. Además, puesto que hay libertad de bloqueo, no se requieren retrocesos. Sin embargo, el protocolo tiene la desventaja de que en algunos casos es necesario que una transacción bloquee datos a los que no accede. Por ejemplo, una transacción que necesite acceder a los datos *A* y *J* del grafo de la *figura 3.5* debe bloquear no sólo *A* y *J* sino también *B*, *D* y *H*. Esto resulta en un aumento del tiempo extra que se requiere para el bloqueo, la posibilidad de tiempo de espera adicional y una disminución potencial de la concurrencia.

Es posible que, para un conjunto de transacciones, existan planificaciones serializables en conflictos que no puedan obtenerse mediante el protocolo de bloqueo de árbol. De hecho existen posibles planificaciones bajo el protocolo de bloqueo de dos fases que no son posibles bajo el protocolo de árbol y viceversa. En los ejercicios se presentan ejemplos de estas planificaciones.

3.6 Protocolos basados en hora de entrada.

En los protocolos de bloqueo descrito arriba, el orden entre cada par de transacciones que están en conflicto, es determinado en el tiempo de ejecución por el primer bloqueo que piden ambas transacciones que implica modos incompatibles [7]. Otro método para determinar el orden de serializabilidad es seleccionar por adelantado una orden entre transacciones. El método más común de hacerlo es utilizar un esquema de *ordenación por hora de entrada*.

Hora de entrada

A cada transacción T_i del sistema le asociamos una hora de entrada fija única, representada por $TS(T_i)$. El sistema de base de datos asigna esta hora de entrada antes de que la transacción T_i empiece su ejecución. Si se asignó la hora de entrada $TS(T_i)$ a la transacción T_i y entra una nueva transacción T_j al sistema, entonces $TS(T_i) < TS(T_j)$. Existen dos métodos sencillos para implementar este esquema:

- Utilizar el valor del reloj del sistema como hora de entrada; es decir, la hora de entrada de una transacción es igual al valor del reloj cuando la transacción entra en el sistema.
- Utilizar el contador lógico que se incrementa después de asignar una nueva hora de entrada; es decir, la hora de entrada de una transacción es igual al valor del contador cuando la transacción entra en el sistema.

Las horas de entrada de las transacciones determinan el orden de serializabilidad. Así, si $TS(T_i) < TS(T_j)$, entonces el sistema debe asegurarse de que la planificación producida es equivalente a una planificación en serie en la que la transacción T_i aparece antes de la transacción T_j .

Para implementar este esquema asociamos dos valores de hora de entrada a cada dato Q :

- **W-hora-de-entrada(Q)**, que representa la mayor hora de entrada de cualquier transacción que ejecutó con éxito $\text{write}(Q)$.
- **R-hora-de-entrada(Q)**, que representa la mayor hora de entrada de cualquier transacción que ejecutó con éxito $\text{read}(Q)$.

Estas horas de entrada se actualizan cada vez que se ejecuta una nueva instrucción $\text{read}(Q)$ o $\text{write}(Q)$.

3.7. Protocolo de ordenación por hora de entrada.

El protocolo de ordenación por hora de entrada garantiza que todas las operaciones **read** y **write** que pudieran entrar en conflicto se ejecuten en el orden de hora de entrada. Este protocolo opera de la siguiente manera:

- Supóngase que la transacción T_i solicita $\text{read}(Q)$.
 - o Si $TS(T_i) < \text{W-hora-de-entrada}(Q)$, entonces esto implica que T_i necesita leer un valor de Q que ya fue sobrescrito. Por tanto, la operación read se rechaza y T_i retrocede.
 - o Si $TS(T_i) \geq \text{W-hora-de-entrada}(Q)$, entonces la operación read se ejecuta, y $\text{R-hora-de-entrada}(Q)$ se pone al valor máximo de $\text{R-hora-de-entrada}(Q)$ y $TS(T_i)$.

- Supóngase que la transacción T_i solicita $\text{write}(Q)$.
 - o Si $TS(T_i) < R\text{-hora-de-entrada}(Q)$, entonces esto implica que el valor de Q en T_i esta produciendo, se necesitó con anterioridad y se supuso que nunca se produciría. Por tanto, la operación write se rechaza y T_i retrocede.
 - o Si $TS(T_i) < W\text{-hora-de-entrada}(Q)$, entonces esto implica que T_i esta intentando escribir un valor obsoleto de Q . Por tanto, esta operación write se rechaza y T_i retrocede.
 - o En los demás casos, la operación write se ejecuta y $W\text{-hora-de-entrada}(Q)$ se pone al valor máximo de $W\text{-hora-de-entrada}(Q)$ y $TS(T_i)$.

A una transacción T_i que retrocede el esquema de control de concurrencia como resultado de solicitar una operación **read** o **write**, se le asigna una nueva hora de entrada y se reinicia.

Para ilustrar este protocolo considérense las transacciones T_{17} y T_{18} definidas a continuación. La transacción T_{17} presenta los contenidos de las cuentas A y B y se define así:

T_{17} : $\text{read}(B)$;
 $\text{read}(A)$;
 $\text{display}(A+B)$.

La transacción T_{18} transfiere 50 dólares de la cuenta A a la cuenta B y después presenta los contenidos de ambas:

T_{18} : $\text{read}(B)$;
 $B := B - 50$;
 $\text{write}(B)$
 $\text{read}(A)$;
 $A := A + 50$;
 $\text{write}(A)$
 $\text{display}(A+B)$.

Al presentar las planificaciones bajo el protocolo de hora de entrada, supondremos que se asigna una hora de entrada a una transacción inmediatamente antes de su primera instrucción. Así, en la planificación 13 de la *figura 3.7.2* $TS(T_{17})$ $TS(T_{18})$, y es posible la planificación bajo el protocolo de hora de entrada.

Observamos que la ejecución de arriba también puede ser producida por el protocolo de bloqueo de dos fases. Sin embargo, existen planificaciones que son posibles bajo el protocolo de bloqueo de dos fases pero no son posibles bajo el protocolo de hora de entrada, y viceversa.

El protocolo de ordenación por hora de entrada garantiza la serializabilidad en conflictos. Esto se sigue del hecho de que las operaciones que pudieran entrar en conflicto se procesan en el orden de la hora de entrada. El protocolo asegura la libertad de bloqueo, ya que las transacciones nunca esperan.

<i>T17</i>	<i>T18</i>
read(<i>B</i>)	read(<i>B</i>) <i>B</i> := <i>B</i> -50 write(<i>B</i>)
read(<i>A</i>)	read(<i>A</i>)
display(<i>A</i> + <i>B</i>)	<i>A</i> := <i>A</i> +50 write(<i>A</i>) display(<i>A</i> + <i>B</i>)

Figura 3.7.2 Planificación 13

3.8. Protocolo Basado en Validación.

En los casos en los que la mayor parte de las transacciones son transacciones sólo de lecturas, la frecuencia con que se presentan conflictos entre transacciones puede ser muy baja. Así muchas de esas transacciones, si se ejecutaran sin la supervisión de una planificación de control de concurrencia, dejarían de todos modos al sistema en un estado consistente. Una planificación de control de concurrencia supone cierto coste extra en la ejecución de código y posible retraso de las transacciones. Pudiera ser conveniente utilizar un esquema alternativo que implique un coste extra menor. Una dificultad al reducir el coste extra es que no sabemos por adelantado cuales son las transacciones que van a entrar en conflicto. Para ello necesitamos tener un esquema de supervisión del sistema.

Suponemos que cada transacción *Ti* se ejecuta en dos o tres fases diferentes durante su tiempo de vida, dependiendo de si se trata de una transacción sólo de lectura o de actualización.

1. **Fase de lectura.** Durante esta fase tiene lugar la ejecución de la transacción *Ti*. Se leen los valores de los diferentes datos y se almacenan en variables locales con respecto a *Ti*. Todas las operaciones write se realizan sobre variables locales, sin actualizar la base de datos real.
2. **Fase de validación.** La transacción *Ti* realiza una prueba de validación para determinar si puede copiar a la base de datos las variables locales temporales que tienen los resultados de las operaciones write sin causar una violación de la serializabilidad.
3. **Fase de escritura.** Si la transacción *Ti* logra la validación (paso 2), entonces se aplican las actualizaciones reales a la base de datos, en caso contrario, *Ti* retrocede.

Las tres fases de transacciones que se ejecutan en forma concurrente pueden intercambiarse. Las fases de lectura y escritura no requieren explicación. La única fase que conviene examinar con más detalle es la fase de validación. Para realizar la prueba de validación necesitamos saber cuando tuvieron lugar las distintas fases de la transacción *Ti*. Por tanto, asociaremos tres horas de entrada diferentes a la transacción *Ti*:

- **Inicio (T_i)**, la hora en que T_i comenzó a ejecutarse.
- **Validación (T_i)**, la hora en que T_i terminó su fase de lectura y comenzó su fase de validación.
- **Término (T_i)**, la hora en que T_i terminó su fase de escritura.

El orden de serializabilidad se determina mediante la técnica de ordenación por hora de entrada, utilizando el valor de la hora de entrada Validación (T_i). Así, el valor $TS(T_i) = \text{Validación}(T_i)$ y si $TS(T_j) < TS(T_k)$, entonces cualquier planificación que se produzca debe ser equivalente a una planificación en serie en la que la transacción T_j aparece antes de la T_k . La razón de que hayamos elegido Validación (T_i) en vez de Inicio (T_i) como hora de entrada de la transacción T_i es que podemos esperar un tiempo de respuesta mejor siempre que la frecuencia de conflictos entre transacciones sea baja.

La prueba de validación para la transacción T_j requiere que para todas las transacciones T_i con $TS(T_i) < TS(T_j)$, se cumpla una de las dos condiciones siguientes:

1. Término (T_i) < Inicio (T_j). Puesto que T_i termina su ejecución antes de que comience T_j el orden de serializabilidad se mantiene.
2. El conjunto de datos que T_i escribe no tiene intersección con el conjunto de datos que T_j lee, y T_i termina su fase escritura antes de que T_j empiece su fase de Validación (Inicio(T_j) < Término (T_i) < Validación(T_i)). Esta condición garantiza que las escrituras de T_i y T_j no se traslapen. Puesto que las escrituras de T_i no afectan a la lectura de T_j y puesto que T_j no puede afectar a la lectura de T_i el orden de serializabilidad se mantiene.

Para ilustrar lo anterior, considérense de nuevo las transacciones T_{17} y T_{18} . Supóngase que $TS(T_{17}) < TS(T_{18})$. Entonces la fase de validación tiene éxito en la producción de la planificación 14, que está representada en la *figura 3.8*. Obsérvese que esta planificación es serializable, pero no puede producirse por el protocolo de bloqueo de dos fases no por el esquema de ordenación por hora de entrada.

El esquema de validación evita automáticamente los retrocesos en cascada, ya que las escrituras reales tienen lugar sólo después de que la transacción que solicita la escritura se haya confirmado.

T_{17}	T_{18}
read(B)	read(B) $B := B - 50$ write(B) read(A) $A := A + 50$ write(A)
read(A) display($A + B$)	display ($A + B$)

Figura 3.8 Planificación 14, planificación producida utilizando validación.

3.9. Protocolo en Árbol.

En los esquemas de control de concurrencia que se han descrito hasta ahora, hemos usado cada dato individual como la unidad en la que se realiza la sincronización [7].

Sin embargo, se dan circunstancias en las que sería conveniente agrupar varios datos y tratarlos como una unidad de sincronización individual. Por ejemplo, si una transacción T_i necesita acceder a toda la base de datos y se utiliza un protocolo de bloqueo, entonces T_i debe bloquear todos los elementos de la base de datos. Claramente, esto implica un consumo de tiempo. Sería mejor si T_i pudiera solicitar un sólo bloqueo para bloquear toda la base de datos. Por otra parte, si la transacción T_i necesita acceder sólo a unos cuantos datos, no debe ser necesario bloquear toda la base de datos, pues se perdería la concurrencia.

Lo que necesita es un mecanismo que permita al sistema definir múltiples niveles de *granularidad*. Esto puede lograrse permitiendo que los datos tengan distintos tamaños y se defina una jerarquía de granularidades de los datos, en la que las granularidades pequeñas se anidan en las más grandes. Una jerarquía así puede representarse gráficamente como un árbol. Nótese que el árbol que describimos aquí es bastante diferente al que utiliza el protocolo de árbol. Un nodo que no es hoja de árbol de granularidad múltiple representa los datos asociados a sus descendientes. En el protocolo de árbol, cada nodo es un dato independiente.

Para ilustrarlo, considérese el árbol de la *figura 3.9*, que consta de cuatro niveles de nodos. El nivel más alto representa toda la base de datos. Debajo de él se encuentran nodos del tipo *área*. Cada *área*, a su vez, tiene nodos del tipo *archivo* como descendientes. Por último, cada *archivo* tiene nodos del tipo *registro*.

Cada nodo de árbol puede bloquearse individualmente. Como en el protocolo de bloqueo de dos fases, utilizaremos los modos de bloqueo *compartido* y *exclusivo*. Cuando una transacción bloquee un nodo, ya sea en modo *compartido* o *exclusivo*, esto implica que la transacción también habrá bloqueado todos los descendientes de ese nodo en el mismo modo de bloqueo. Por ejemplo, si la transacción T_i bloquea *explícitamente* el archivo F_c de la *figura 3.9*, en modo *exclusivo*, entonces ha bloqueado *implícitamente* en modo *exclusivo* todos los registros pertenecientes a ese archivo. No necesita bloquear *explícitamente* los registros individuales de F_c .

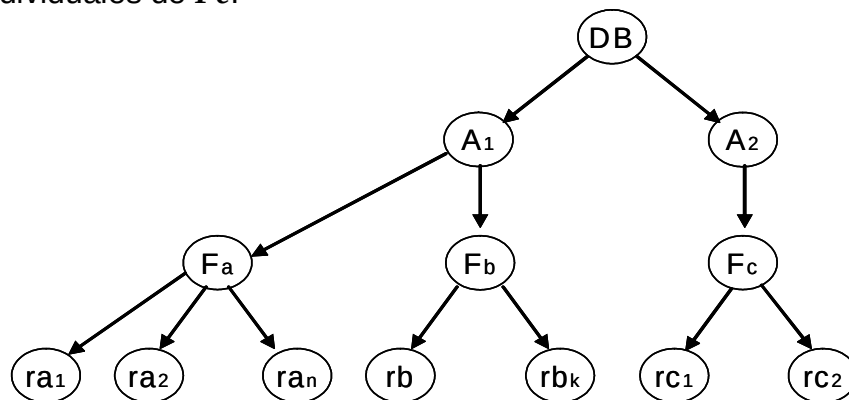


Figura 3.9 Jerarquía de granularidad.

Supóngase que la transacción **Tj** desea bloquear el registro **rb6** del archivo **Fb**. Puesto que **Ti** bloqueó **Fb** explícitamente, se sigue que **rb6** también se bloqueará (implícitamente). ¡Pero cuando **Ti** solicite una petición de bloqueo para **rb6**, **rb6** no se bloqueará explícitamente! ¿Cómo determina el sistema si **Ti** puede bloquear **rb6** o no? Para hacerlo, **Tj** debe recorrer el árbol desde la raíz hasta el registro **rb6**. Si cualquier nodo en ese camino está bloqueado en un modo incompatible, entonces **Ti** debe retrasarse.

Supóngase ahora que la transacción **Ti** desea bloquear toda la base de datos. Para hacerlo, debe bloquear simplemente la raíz de la jerarquía. Sin embargo, obsérvese que **Tk** no debe tener éxito en el bloqueo del nodo raíz, ya que **Ti** tiene actualmente un bloqueo en parte del árbol (específicamente el archivo **Fb**). ¿Cómo determina esto el sistema? Una posibilidad es recorrer todo el árbol. Sin embargo, se anularía todo el propósito de la planificación de bloqueo de granularidad múltiple. Una forma más eficiente de lograr esto es introducir una nueva clase de modos de bloqueo, llamados *modos de bloqueo de intención*. Si se bloquea un nodo de un modo de intención, implica que se está haciendo un bloqueo explícito en un nivel más bajo del árbol (es decir, en una granularidad más fina). Los bloqueos de intención se ponen en todos los antepasados de un nodo antes de que ese nodo se bloquee explícitamente. Así pues, una transacción no necesita recorrer todo el árbol para determinar si puede bloquear un nodo con éxito o no. para lograrlo, una transacción que desee bloquear un nodo, por ejemplo **Q**, debe recorrer un camino del árbol desde la raíz hasta **Q**. Mientras hace el recorrido, la transacción va bloqueando los distintos nodos en modo de intención ¹.

Si un nodo se bloquea en modo compartido de intención (representado por **IS**), esto implica que se está haciendo bloqueo explícito en un nivel más bajo del árbol pero sólo con bloqueos de modo compartido. De manera similar, si un nodo se bloquea en modo *exclusivo de intención* (representado por **IX**), entonces se está haciendo un bloqueo explícito en un nivel más bajo con bloqueos de modo exclusivo o de modo compartido. Por último, si un nodo se bloquea en un modo *compartido y exclusivo de intención* (representado por **SIX**), esto implica que el subárbol del cual ese nodo es raíz tiene un bloqueo explícito en modo compartido y que se está haciendo bloqueo explícito en un nivel más bajo con bloqueos de modo exclusivo. La función de compatibilidad de estos modos de bloqueo se presenta en la *figura 3.9.1*.

El protocolo de bloqueo de granularidad múltiple que se presenta a continuación garantiza la serializabilidad. Cada transacción **Ti** puede bloquear un nodo **Q** utilizando las reglas siguientes:

1. Debe respetarse la función de compatibilidad de bloqueo de la *figura 3.9.1*.
2. Primero debe bloquearse la raíz del árbol y puede bloquearse en cualquier modo.
3. **Ti** puede bloquear al nodo **Q** en el modo **S** o **IS** sólo si **Ti** bloqueó al padre de **Q** en modo **IX** o **SIX**.

¹ Existe un modo de intención asociado al modo compartido y otro asociado al modo exclusivo.

4. T_i puede bloquear al nodo Q en el modo X , SIX o IX sólo si T_i bloqueó al padre de Q en modo IX o SIX .
5. T_i puede bloquear un nodo sólo si no desbloqueó anteriormente ningún nodo (es decir, T_i es de dos fases).
6. T_i puede desbloquear un nodo Q sólo si ninguno de los hijos de Q está bloqueado por T_i .

	IS	IX	S	SIX	X
IS	Verdadero	Verdadero	Verdadero	Verdadero	Falso
IX	Verdadero	Verdadero	Falso	Falso	Falso
S	Verdadero	Falso	Verdadero	Falso	Falso
SIX	Verdadero	Falso	Falso	Falso	Falso
X	Falso	Falso	Falso	Falso	Falso

Figura 3.9.1. Matriz de compatibilidad.

Obsérvese que el protocolo de granularidad múltiple requiere que los bloqueos se adquieran en orden de arriba-abajo (de la raíz a las hojas) y se liberen en orden de abajo-arriba (de las hojas a la raíz).

Para ilustrar el protocolo, considérese el árbol de la *figura 3.9* y las siguientes transacciones:

- Sea T_{19} la transacción que lee el registro ra_2 del archivo Fa . T_{19} necesita bloquear la base de datos, el área A_1 y Fa en modo IS (y en ese orden), y después bloquear ra_2 en modo S .
- Sea T_{20} la transacción que modifica el registro ra_9 del archivo Fa . T_{20} necesita bloquear la base de datos, el área A_1 y el archivo Fa en modo IX , y después bloquear ra_9 en modo X .
- Sea T_{21} la transacción que lee todos los registros del archivo Fa . T_{21} necesita bloquear la base de datos y el área A_1 en modo IS , y después bloquear Fa en modo X .
- Sea T_{22} la transacción que lee toda la base de datos. T_{22} puede hacerlo bloqueando la base de datos en modo S .

Observamos que la transacción T_{19} , T_{21} y T_{22} pueden acceder a la base de datos de manera concurrente. La transacción T_{20} puede ejecutarse concurrentemente con T_{19} pero no con T_{21} ni con T_{22} .

Este protocolo intensifica la concurrencia y reduce el gasto extra de bloqueo. Esto es particularmente útil en aplicaciones que incluyen una mezcla de:

- Transacciones cortas que acceden sólo a unos pocos datos.
- Transacciones largas que producen informes de un archivo completo o de un conjunto de archivos.

3.10. Esquema multiversión.

Los esquemas de control de concurrencia tratados hasta ahora garantizan la serializabilidad retrasando una operación o bien abortando la transacción que solicitó la operación [7]. Por ejemplo, una operación **read** puede retrasarse porque todavía no se ha escrito el valor adecuado, o puede rechazarse (es decir, la transacción que la solicitó debe abortarse) porque ya se escribió sobre el valor que supuestamente debía leer. Estos problemas pueden evitarse si se mantienen copias antiguas de todos los datos en el sistema.

En los sistemas de base de datos de *multiversión* cada versión **write** crea una nueva versión de Q . Cuando se solicita una operación **read**(Q), el sistema selecciona una de las versiones de Q para leerse. El esquema de control de concurrencia debe garantizar que la selección de la versión que se va a leer se haga de tal manera que se garantice la serializabilidad. También es esencial, por razones de rendimiento, que una transacción pueda determinar fácil y rápidamente cual es la versión del dato que se debe leer.

La técnica más común que se utiliza entre los esquemas de multiversión es la de hora de entrada. A cada transacción T_i en el sistema se asocia una hora de entrada estática única, representada por $TS(T_i)$. Esta hora de entrada se asigna de la misma manera que se describió en la sección 3.6.

A cada dato Q se asocia una secuencia de versiones $\langle Q_1, Q_2 \dots Q_m \rangle$. Cada versión Q_k contiene tres campos de información.

- Contenido, el valor de la versión Q_k .
- W-hora-de-entrada (Q_k), que representa la hora de entrada de la transacción que creó la versión Q_k .
- R-hora-de-entrada (Q_k), que representa la hora de entrada más grande de cualquier transacción que haya leído con éxito la versión Q_k .

Una transacción, por ejemplo T_i crea una nueva versión Q_k del dato Q cuando ejecuta una operación **write**(Q). El campo de contenido de la versión tiene el valor que escribió T_i . La W-hora-de-entrada y la R-hora-de-entrada se inicializan a $TS(T_i)$. El valor de R-hora-de-entrada se actualiza cada vez que una transacción T_j lee el contenido de Q_k y **R-hora-de-entrada** (Q_k) $< TS(T_j)$.

El esquema de hora de entrada de multiversión que se presenta a continuación, garantiza la serializabilidad. El esquema opera como sigue. Supóngase que la transacción T_i solicita una operación **read**(Q) o **write**(Q). Sea Q_k una versión de Q cuya hora de entrada de escritura sea la hora de entrada más grande, menor que $TS(T_i)$.

1. Si la transacción T_i solicita **read**(Q), entonces el valor devuelto es el contenido de la versión Q_k .
2. Si la transacción T_i solicita **write**(Q), y si $TS(T_i) < \text{R-hora-de-entrada}(Q_k)$, entonces la transacción retrocede; en caso contrario se crea una nueva versión de Q_k .

La justificación de la regla 1 es obvia. Una transacción lee la versión más reciente anterior a ella en el tiempo. La segunda regla obliga a abortar una transacción si es <<demasiado grande>> para hacer una escritura. De manera más precisa. Si T_i intenta escribir una versión que hubiera leído otra transacción, entonces no podemos permitir que se lleve a cabo esa escritura.

El esquema tiene la propiedad deseable de que una solicitud de lectura nunca falla y nunca tiene que esperar. En los sistemas de base de datos más comunes, donde la lectura es una operación más frecuente que la escritura, esta ventaja puede tener gran importancia práctica.

Sin embargo, el esquema tiene dos propiedades indeseables. En primer lugar, la lectura de un dato requiere también la actualización del campo R-hora-de-entrada, que resulta en dos accesos potenciales a disco en lugar de uno. En segundo lugar, los conflictos entre transacciones se resuelven por medio de retrocesos en vez de esperas. Esto puede ser muy costoso.

3.11. Las operaciones insertar y suprimir.

Hasta ahora hemos restringido la atención a las operaciones **read** y **write** [7]. Esta restricción limita las transacciones a datos que ya existen en la base de datos. Algunas transacciones no sólo requieren acceso a datos que ya existen, sino también a la posibilidad de crear nuevos datos. Otras requieren la posibilidad de eliminar datos. Para examinar como afectan estas transacciones al control de concurrencia, introducimos las siguientes operaciones adicionales:

- Delete (Q), que elimina el dato Q de la base de datos.
- Insert, que inserta un nuevo dato Q en la base de datos y asigna a Q un valor inicial.

Un intento por una transacción T_i de realizar una operación $\text{read}(Q)$, después de que se haya eliminado Q , resulta en un error lógico en T_i . Igualmente, un intento por una transacción T_i de realizar una operación $\text{read}(Q)$ antes de que se haya insertado Q resulta en un error lógico en T_i . También es un error lógico intentar eliminar un dato que no existe.

Supresión.

Para entender como afecta la presencia de instrucciones **delete** al control de concurrencia, debemos decidir cuando una transacción **delete** va a tener conflictos con otra instrucción. Sean I_i e I_j instrucciones de T_i y T_j respectivamente, que aparecen en la planificación S en orden consecutivo. Sean $I_i = \text{delete}(Q)$. A continuación consideramos varias instrucciones I_j :

- $I_j = \text{read}(Q)$. I_i e I_j tienen conflictos. Si I_i viene antes de I_j , T_j tendrá un error lógico. Si I_j viene antes de I_i , T_j puede ejecutar la operación read con éxito.

- $I_j = \text{write}(Q)$. I_i e I_j tienen conflictos. Si I_i viene antes de I_j , T_j tendrá un error lógico. Si I_j viene antes de I_i , T_j puede ejecutar la operación write con éxito.
- $I_j = \text{delete}(Q)$. I_i e I_j tienen conflictos. Si I_i viene antes de I_j , T_i tendrá un error lógico. Si I_j viene antes de I_i , T_i puede ejecutar la operación write con éxito.
- $I_j = \text{insert}(Q)$. I_i e I_j tienen conflictos. Supóngase que el dato Q no existía antes de la ejecución de I_i e I_j . Entonces, si I_i viene antes de I_j , T_i tendrá un error lógico. Si I_j viene antes de I_i , entonces no resulta ningún error lógico. Igualmente, si Q existía antes la ejecución de I_i e I_j , entonces resulta un error lógico si I_j viene antes de I_i , pero no en caso contrario.

Llegamos a la conclusión de que, si se emplea bloqueo de dos fases, se requiere un bloqueo exclusivo en un dato antes de que pueda eliminarse. Bajo el protocolo de ordenación por hora de entrada se debe realizar una prueba similar a la de write. Supóngase que la transacción T_i solicita **delete**(Q).

- $TS(T_i) < R\text{-hora-de-entrada}(Q)$, entonces esto implica que el valor de Q que tiene T_i iba a suprimir ya ha sido suprimido por una transacción T_j con $TS(T_j) > TS(T_i)$. Por tanto, la operación delete se rechaza y T_i retrocede.
- $TS(T_i) < W\text{-hora-de-entrada}(Q)$, entonces esto implica que una transacción T_j con $TS(T_j) > TS(T_i)$ ha escrito Q . Por tanto, esta operación delete se rechaza y T_i retrocede.
- En caso contrario, la operación delete se ejecuta.

Inserción.

Ya hemos visto que una operación **insert**(Q) tiene conflictos con una operación **delete**(Q). De la misma forma, **insert**(Q) tiene conflictos con una operación **read**(Q) o con una operación **write**(Q). No se puede realizar **read** o **write** en un dato antes de que exista.

Puesto que un **insert**(Q) asigna un valor al dato Q , un **insert** se trata de la misma manera que un **write** para propósitos de control de concurrencia:

- Bajo el protocolo de bloqueo de dos fases, si T_i realiza una operación **insert**(Q), a T_i se le da un bloqueo exclusivo en el dato Q recién creado.
- Bajo el protocolo de ordenación por hora de entrada, si T_i realiza una operación **insert**(Q), los valores $R\text{-hora-de-entrada}(Q)$ y $W\text{-hora-de-entrada}(Q)$ se ponen a $TS(T_i)$.

3.12. BLOQUEO MORTAL O DEADLOCK

El deadlock¹ es una condición que ningún sistema o conjunto de procesos quisiera exhibir, ya que consiste en que se presentan al mismo tiempo cuatro condiciones necesarias: La condición de no apropiación, la condición de espera circular, la condición de exclusión mutua y la condición de ocupar y esperar un recurso [21 y 22]. Ante esto, si el deadlock involucra a todos los procesos del sistema, el sistema ya no podrá hacer algo productivo. Si el deadlock involucra algunos procesos, éstos quedarán congelados para siempre.

¹ Al bloqueo mortal también se le conoce, coloquialmente, como abrazo mortal.

Causas del Dead Lock.

Los puntos muertos ocurren cuando dos o más transacciones solicitan recursos en forma incremental y se bloquean mutuamente impidiéndose una a otra la conclusión. Las transacciones y recursos forman un ciclo [21 y 22].

Hasta es posible que un punto muerto se cree cuando se realiza al acceso a un solo objeto mediante dos transacciones y se aceptan reclamaciones incrementales. Este tipo de punto muerto tiene una mayor probabilidad de ocurrir si los objetos que se están asegurando son grandes cuando la unidad de aseguramiento es un archivo la primera reclamación puede emitirse para lograr acceso a un registro y la reclamación incremental emitirse a fin de actualizar otro registro de archivo.

Puntos muertos debidos a recursos compartidos del sistema, los puntos muertos también pueden provocarse debido a la competencia por objetos que no estén identificados en forma específica, pero que sean miembros de una clase compartida de recursos. Un bloqueo temporal ocurre cuando los límites de un recurso se alcanzan. La posibilidad de puntos muertos existen 4 condiciones:

- **Seguros:** La interferencia de acceso se resuelve posesionando y reputando los seguros.
- **Bloqueo:** Un propietario de un objeto está bloqueado cuando solicita un objeto asegurado.
- **Garantía de Conclusión:** Los objetos no pueden quitarse de sus propietarios.
- **Circulación:** Existe una secuencia circular de solicitud como se muestra en el.

En el área de la informática, el problema del deadlock ha provocado y producido una serie de estudios y técnicas muy útiles, ya que éste puede surgir en una sola máquina o como consecuencia de compartir recursos en una red.

En el área de las bases de datos y sistemas distribuidos han surgido técnicas como el 'two phase locking' y el 'two phase commit' que van más allá de este trabajo. Sin embargo, el interés principal sobre este problema se centra en generar técnicas para detectar, prevenir o corregir el deadlock.

Las técnicas para prevenir el deadlock consisten en proveer mecanismos para evitar que se presente una o varias de las cuatro condiciones necesarias del deadlock. Algunas de ellas son:

- **Asignar recursos en orden lineal:** Esto significa que todos los recursos están etiquetados con un valor diferente y los procesos solo pueden hacer peticiones de recursos 'hacia adelante'. Esto es, que si un proceso tiene el recurso con etiqueta '5' no puede pedir recursos cuya etiqueta sea menor que '5'. Con esto se evita la condición de ocupar y esperar un recurso.
- **Asignar todo o nada:** Este mecanismo consiste en que el proceso pida todos los recursos que va a necesitar de una vez y el sistema se los da solamente si puede dárselos todos, si no, no le da nada y lo bloquea.
- **Algoritmo del banquero:** Este algoritmo usa una tabla de recursos para saber cuántos recursos tiene de todo tipo. También requiere que los procesos informen del máximo de recursos que va a usar de cada tipo. Cuando un proceso pide un recurso, el algoritmo verifica si asignándole ese recurso todavía le quedan otros

del mismo tipo para que alguno de los procesos en el sistema todavía se le pueda dar hasta su máximo. Si la respuesta es afirmativa, el sistema se dice que está en 'estado seguro' y se otorga el recurso. Si la respuesta es negativa, se dice que el sistema está en estado inseguro y se hace esperar a ese proceso.

Para detectar un deadlock, se puede usar el mismo algoritmo del banquero, que aunque no dice que hay un deadlock, sí dice cuándo se está en estado inseguro que es la antesala del deadlock. Sin embargo, para detectar el deadlock se pueden usar las 'gráficas de recursos'. En ellas se pueden usar cuadrados para indicar procesos y círculos para los recursos, y flechas para indicar si un recurso ya está asignado a un proceso o si un proceso está esperando un recurso. El deadlock es detectado cuando se puede hacer un viaje de ida y vuelta desde un proceso o recurso. Por ejemplo, suponga los siguientes eventos:

evento 1: Proceso A pide recurso 1 y se le asigna.

evento 2: Proceso A termina su time slice.

evento 3: Proceso B pide recurso 2 y se le asigna.

evento 4: Proceso B termina su time slice.

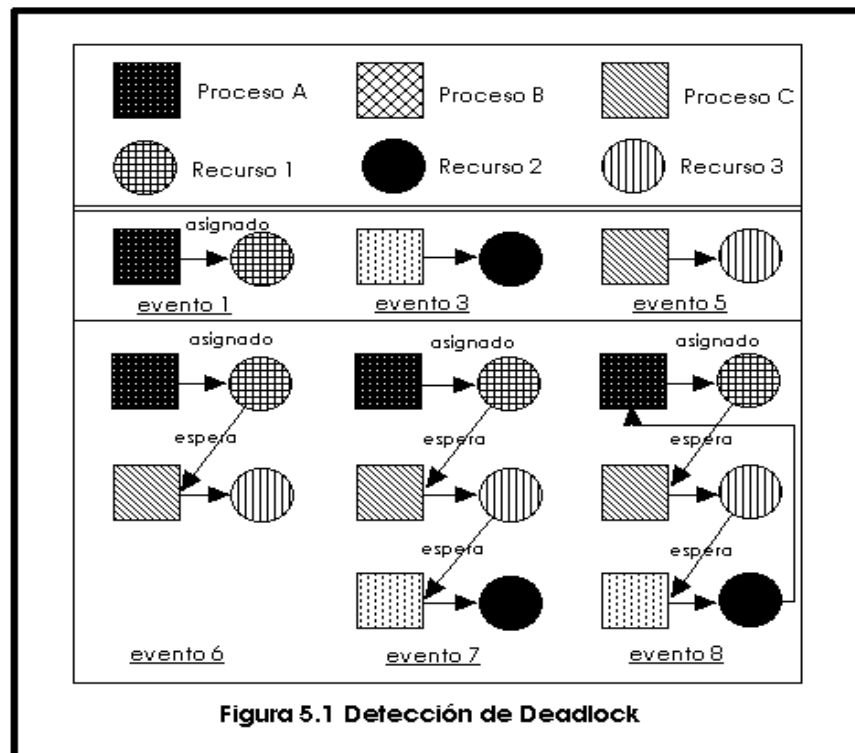
evento 5: Proceso C pide recurso 3 y se le asigna.

evento 6: Proceso C pide recurso 1 y como lo está ocupando el proceso A, espera.

evento 7: Proceso B pide recurso 3 y se bloquea porque lo ocupa el proceso C.

evento 8: Proceso A pide recurso 2 y se bloquea porque lo ocupa el proceso B.

En la figura 5.1 se observa como el 'resource graph' fue evolucionando hasta que se presentó el deadlock, el cual significa que se puede viajar por las flechas desde un proceso o recurso hasta regresar al punto de partida. En el deadlock están involucrados los procesos A, B y C.



Una vez que un deadlock se detecta, es obvio que el sistema está en problemas y lo único que resta por hacer es una de dos cosas: tener algún mecanismo de suspensión o reanudación que permita copiar todo el contexto de un proceso incluyendo valores de memoria y aspecto de los periféricos que esté usando para reanudarlo otro día, o simplemente eliminar un proceso o arrebatárle el recurso, causando para ese proceso la pérdida de datos y tiempo.

Técnicas para prevenir los DEADLOCK

Los métodos de corregir un deadlock una vez que ya se presentó éste conllevan la pérdida de tiempo y no de información, ya que las transacciones pueden abortarse de manera segura debido a las facilidades de las bitácoras, el protocolo de bloqueo de dos fases y el protocolo commit de dos fases [21 y 22].

El método de prevención del deadlock también hace necesario abortar transacciones y siempre serán las transacciones más nuevas o jóvenes las abortadas para evitar los deadlocks.

Existen dos métodos: **el método apropiativo** y **el no apropiativo**. Ambos se basan en la hora en que las transacciones son creadas, podemos hablar entonces de transacciones viejas y jóvenes.

En el **método no apropiativo**, si una transacción Ta pide un recurso bloqueado por Tb, se permite que Ta espere solamente si Ta es más vieja que Tb. Ta es reinicializada si es más joven que Tb, conservando siempre su antigüedad. Lo que se consigue con el método no apropiativo es que ninguna transacción joven esperará por las transacciones viejas, eliminando la condición de espera circular distribuida.

En el método apropiativo la regla es la inversa: Si Ta pide un bloqueo sobre un recurso cuyo dueño es Tb, se le permite esperar si Ta es más joven que Tb. Si Ta es más vieja, no espera sino que Tb es abortada por ser más joven y luego se reinicializa conservando su antigüedad.

También podemos anular alguna de las 4 condiciones necesarias para que se produzca un deadlock:

- No puede ser anulada porque existen recursos que deben ser usados en modalidad exclusiva.
- La alternativa sería hacer que todos los procesos solicitaran todos los recursos que habrán de utilizar antes de utilizarlos al momento de su ejecución lo cual sería muy ineficiente.
- Para anular esta condición cuando un proceso solicita un recurso y este es negado el proceso deberá liberar sus recursos y solicitarlos nuevamente con los recursos adicionales. El problema es que hay recursos que no pueden ser interrumpidos.
- Espera Circular: esta estrategia consiste en que el sistema operativo numere en forma exclusiva los recursos y obligue a los procesos a solicitar recursos en forma ascendente. El problema de esto es que quita posibilidades a la aplicación.

* Nota: En la práctica no todos los sistemas detectan los bloqueos mortales, sino que algunos usan un mecanismo de tiempo y asumen simplemente que una transacción que no ha realizado algún trabajo durante cierto periodo restablecido, está bloqueada mortalmente.

Si se tiene cuidado en la forma de asignar los recursos se pueden evitar situaciones de Deadlock. Supongamos un ambiente en el que todos los procesos declaren a priori la cantidad máxima de recursos que habrá de usar.

Estado Seguro: Un estado es seguro si se pueden asignar recursos a cada proceso (hasta su máximo) en algún orden sin que se genere Deadlock. El estado es seguro si existe un ordenamiento de un conjunto de procesos $\{P_1 \dots P_n\}$ tal que para cada P_i los recursos que P_i podrá utilizar pueden ser otorgados por los recursos disponibles mas los recursos utilizados por los procesos $P_j, j < i$. Si los recursos solicitados por P_i no pueden ser otorgados, P_i espera a que todos los procesos P_j hayan terminado.

Para evitar puntos muertos puede simplificar muchas de elecciones alternativas. Los esquemas para evitar estos casos imponen a los usuarios restricciones que pueden resultar difíciles de aceptar. Existen cuatro enfoques para puntos muertos:

- 1.- **Reparación posterior:** No utilizar seguros y arreglar después las fallas por inconsistencia. El primer enfoque, es reparar posteriormente los problemas debido a no asegurar, puede responder a un válido en sistemas experimentales y educativos, pero resulta inaceptable en la mayoría de las aplicaciones comerciales.
- 2.- **No bloquear:** Solamente afectar a quienes efectuaron solicitudes que provocaron reclamaciones en conflicto. Este segundo enfoque asigna la responsabilidad a la transacción, el sistema proporcionara un aviso de interferencia potencial al negar la solicitud de acceso explosivo.
- 3.- **Asignación Previa:** Si existe algún conflicto, quitar los objetos a sus propietarios. La asignación previa de reclamaciones otorgadas a las transacciones reúne de una capacidad de “volver a enrollar”. La transacción, cuando se le notifica que no puede continuar, tiene que restaurar la base de datos y colocarse así misma en la línea de espera para un nuevo turno o el sistema tiene que eliminar la transacción, restaurar la base de datos y volver a iniciar de nuevo la transacción.
- 4.- **Aseguramiento de dos fases:** Se realizan primero todas las reclamaciones y si ninguna esta bloqueada se inician todas las modificaciones. Un enfoque simple para evitar la circularidad consiste en hacer que se reclamen previo todos los objetos antes de otorgar ningún seguro, el reclamar los recursos antes de prometer otorgar el acceso a ellos significa que posiblemente una transacción no sea capaz de concluir la fase de reclamación previa. El problema de dos fases es que la reclamación previa puede llegar a tener que reclamar más y mayores objetos de los que en realidad se necesitan, si un cálculo sobre los datos determina que objetos se necesitan después, es posible que se reclame en forma previa todo un archivo en vez de un registro.

Capítulo 4. CASO DE ESTUDIO: TRANSACCIONES EN MySQL.

En este capítulo, se expone un caso de estudio para analizar las transacciones mediante MySQL, en donde se ejemplifican los diferentes tipos de transacciones que son ejecutadas por usuarios, a la vez de caracterizar cada una de las mismas.

4. Caso de Estudio: Transacciones en MySQL.

4.1. Características de MySQL

MySQL es un manejador de bases de datos principalmente utilizado en las aplicaciones WEB en conjunto con PHP. Con su versión MYSQL 5.0 anexa características importantes que lo hacen más competitivo con manejadores como SQL Server, Sybase y Oracle [18 y 50]. Las Especificaciones Técnicas de MySQL se encontraran descritas en el ANEXO A.

Entre sus características tenemos:

- **Uso de transacciones ACID** (*Atomic, Consistent Isolated, Durable*): Para construir aplicaciones más seguras mediante commit, rollback, crash recovery y bloqueo por registro.
- **Store Procedures**: Para mejorar la programación.
- **Triggers**: Para mejorar las reglas del negocio.
- **Vistas**: Para que la información sensible sea más segura.
- **Information Schema**: Para un fácil acceso a los metadatos.
- **Transacciones Distribuidas (XA)**: Para soportar transacciones entre múltiples ambientes de bases de datos.
- **Es un gestor de base de datos**: Una base de datos es un conjunto de datos y un gestor de base de datos es una aplicación capaz de manejar este conjunto de datos de manera eficiente y cómoda.
- **Es una base de datos relacional**: Una base de datos relacional es un conjunto de datos que están almacenados en tablas entre las cuales se establecen unas relaciones para manejar los datos de una forma eficiente y segura. Para usar y gestionar una base de datos relacional se usa el lenguaje estándar de programación SQL.
- **Es Open Source**: El código fuente de MySQL se puede descargar y está accesible a cualquiera, por otra parte, usa la licencia GPL para aplicaciones no comerciales.
- **Es una base de datos muy rápida**, segura y fácil de usar. Gracias a la colaboración de muchos usuarios, la base de datos se ha ido mejorando optimizándose en velocidad. Por eso es una de las bases de datos más usadas en Internet.

4.2. Tipos de tablas usadas por MySQL

ISAM: es el formato de almacenaje más antiguo, y posiblemente podría desaparecer en futuras versiones. Presentaba limitaciones importantes como la no exportación de ficheros entre maquinas de distintas arquitecturas o que no podía usar mayores de 4 GigaBytes [18 y 50].

MYISAM: es el tipo de tabla por defecto en MySQL desde la versión 3.23. Optimizada para sistemas operativos de 64 bits, permite ficheros de tamaños mayores que las ISAM. Los datos se almacenan en un formato independiente, lo que permite pasar tablas entre distintas plataformas. Los índices se almacenan en un archivo con extensión ".MYI" y los datos en otro archivo con extensión ".MYD".

Ofrece la posibilidad de indexar campos BLOB y TEXT. Además este tipo de tablas soportan el tipo de dato VARCHAR.

Un inconveniente es que las tablas pueden llegar a corromperse, almacenando datos incorrectos. Esto puede ser causado por:

- El proceso mysqld haya sido eliminado en el transcurso de una escritura.
- problemas de hardware.
- Una caída del sistema durante su utilización.
- Un gusano en el código Mysql o MyISAM.

INNODB: InnoDB provee a MySQL con el soporte para trabajar con transacciones, además de hacer un mejor bloqueo de registros para las instrucciones SELECT muy parecido al usado por Oracle, con lo que incrementa el rendimiento y la concurrencia en ambientes multiusuario, por otro lado, InnoDB es el único formato que tiene MySQL para soportar llaves foráneas (FOREIGN KEY). Además de todo lo comentado, InnoDB ofrece unos rendimientos superiores a la anterior tecnología de tablas de MySQL (MyISAM).

InnoDB es un motor de bases de datos muy completo que ha sido integrado dentro de MySQL.

Otras de sus características son:

- Recuperación automática ante fallas. Si MySQL se da de baja de una forma anormal, InnoDB automáticamente completará las transacciones que quedaron incompletas.
- Integridad referencial. Ahora se pueden definir llaves foráneas entre tablas InnoDB relacionadas para asegurarse de que un registro no puede ser eliminado de una tabla si aún está siendo referenciado por otra tabla.
- Bloqueo a nivel de filas. Al usar tablas MyISAM, y tener consultas muy grandes que requieren de mucho tiempo, simplemente no se podían ejecutar más consultas hasta que terminaran las consultas que estaban en ejecución. En cambio, las tablas InnoDB usan bloqueo a nivel de filas para mejorar de manera impresionante el rendimiento.
- SELECTs sin bloqueo. El motor InnoDB usa una técnica conocida como multi-versioning (similar a PostgreSQL) que elimina la necesidad de hacer bloqueos en consultas SELECT muy simples. Ya no será necesario molestarse porque una simple consulta de sólo lectura está siendo bloqueada por otra consulta que está haciendo cambios en una misma tabla.

HEAP: Tablas en memoria. Son temporales y desaparecen cuando el servidor se cierra, a diferencia de una tabla TEMPORARY, que solo puede ser accedida por el usuario que la crea, una tabla HEAP puede ser utilizada por diversos usuarios. No soportan columnas de autoincremento ni que haya valores nulos en los índices. Los datos son almacenados en pequeños bloques.

BDB: Base de datos Berkeley. TST(Transactions safe tables). Solo en MySQL

MAX: Este tipo de tablas permite la realización de transacciones (a partir de la versión 3.23.34), por lo que es posible la recuperación de datos (COMMIT y ROLLBACK). Estas tablas necesitan de una clave primaria en cada tabla, que ha de crear el administrador o de lo contrario Mysql creará una oculta. Otra de sus características es que pueden ser bloqueadas con el comando LOCK. Estas tablas són almacenadas en archivos “.DB”.

El TST '*Transactions safe tables*', o tablas para transacciones seguras. Son menos rápidas y ocupan mas memoria, pero a cambio ofrecen mayor seguridad frente a fallos durante la consulta. Las tablas TST permiten ir introduciendo consultas y finalizar con un COMMIT 0(que las ejecuta) o ROLLBACK (que ignora los cambios). Disponbles a partir de la versión 4 de MySQL.

4.3. Que es un Stored Procedure?

Un stored procedure (S.P.) es un procedimiento (como una subrutina en cualquier lenguaje de programación) almacenado en una B.D. que puede ser invocado externamente [18 y 50].

- Está formado por un encabezado y un cuerpo.
- El encabezado contiene el nombre del procedimiento y una lista de parámetros.
- El cuerpo puede contener sentencias SQL y sentencias propias de un lenguaje de programación imperativo: declaración y asignación de variables, condicionales, repetición, etc.

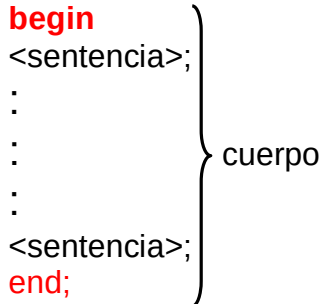
4.3.1. Stored Procedures: Ventajas

- Mejora la performance: Reduce la comunicación por red (se ejecuta en el servidor).
- Las operaciones comunes y frecuentes pueden almacenarse en la B.D. como un S.P. y ser invocadas desde cualquier aplicación.
 - Reduce la complejidad de las aplicaciones que usan la B.D.
 - Las aplicaciones pueden compartir código.
 - Separamos la interfase de la lógica del sistema.
- Mejora la portabilidad del sistema. Los S.P. son portables, se pueden ejecutar en cualquier plataforma donde se ejecuta MySQL.
- Permite asociar los datos con las operaciones para manejar los datos.

4.3.2. Creación de Stored Procedure en MySQL

- Se utiliza la sentencia **create procedure**.

create procedure <nombre>  **encabezado**

begin  **cuerpo**

<sentencia>;
:
:
:
:
<sentencia>;
end;

- **In, out e inout** definen parámetros de **entrada, salida y entrada-salida** respectivamente.
- **<tipo>**: cualquier tipo válido de MySQL: int,char,date,etc.
- **<sentencia>**: sentencia SQL, condicionales, repetición, ...
- El S.P. es validado sintácticamente y optimizado en el momento de la creación.

4.3.3. Cursores

Los *cursores* permiten recuperar las tuplas resultantes de una consulta *select* dentro de un S.P.

Los cursores se declaran de la sig. forma:

declare <nombre_cursor> **cursor for** <consulta_select>

Para acceder al resultado de la consulta debemos primero abrir el cursor:

open <nombre_cursor>

Una vez abierto se puede recuperar cada tupla mediante la siguiente sentencia:

fetch <nombre_cursor> **into** <variable1>[...,<variableN>]

recupera la próxima tupla (si existe) utilizando el cursor especificado y avanza el puntero a la siguiente tupla.

Una vez utilizado podemos cerrar el cursor mediante:

close <nombre_cursor>

4.3.4. Que es un trigger?

Un trigger es un objeto que se almacena en la B.D. asociado a una tabla y se activa cuando ocurre un evento en particular para esa tabla.

Los trigger contienen sentencias que se ejecutan cuando el trigger se activa.

Los eventos que pueden activar un trigger son:

- Se inserta una nueva fila en la tabla (**INSERT**).
- Se borra un fila de la tabla (**DELETE**).
- Se actualiza una o mas filas de la tabla (**UPDATE**).

También es posible definir si el trigger se activará antes (**BEFORE**) o después (**AFTER**) que se produzca el evento.

No puede haber dos triggers asociados a una misma tabla que correspondan al mismo.

Para que usar triggers?

- Implementar reglas de consistencia de datos no provistas por el modelo relacional.
- Verificar y prevenir que se inserten datos inválidos.
- Replicación de datos.
- Auditoria: monitorear la actividad de los cambios sobre los datos.
- Acciones en cascadas.
- Autorización y seguridad.

Creación de triggers

CREATE TRIGGER <nombre del trigger>
{ **BEFORE** | **AFTER** } cuando se ejecuta: antes o después
{ **INSERT** | **UPDATE** | **DELETE** } que evento lo activa
ON <nombre de tabla> tabla donde se produce el evento
FOR EACH ROW <sentencia> Qué se ejecuta. Cualquier sentencia válida para un S.P., incluyendo: sentencias SQL, sentencias compuestas, llamadas a S.P., condicionales, etc.

- Las variables especiales de transición **OLD** y **NEW** hacen referencia a los valores de la fila afectada antes y después de la modificación respectivamente.
- **OLD** contiene los valores de la fila afectada antes de ser modificada.
- **NEW** contiene los valores de la fila afectada después de ser modificada.
- Si el evento es **INSERT** solo se puede utilizar **NEW** y si es **DELETE** solo se puede usar **OLD**. Con **UPDATE** se pueden usar ambas.

4.4. Transacciones

Transacción: secuencia de instrucciones (sentencias SQL) relacionadas, que deben ser tratadas como una unidad indivisible [18 y 50]. El uso de las B.D. Es más seguro y eficiente:

- Reglas ACAD (ACID): atomicidad, consistencia, aislamiento (isolation), durabilidad.
- muchos programas y usuarios pueden acceder concurrentemente a las B.D.
- Control de concurrencia.
- Recuperación de fallos.

Transacciones: ejemplo 1

Consideremos nuevamente la B.D. *bank* que almacena el saldo de cada cuenta.

Operación habitual: **transferir** un monto **M** de una cuenta **A** a una cuenta **B**.

```
Transferir(M,A,B)
Read(SaldoA);
if SaldoA > M then
    SaldoA:=SaldoA-M;
```

```

Write(SaldoA);
Read(SaldoB);
SaldoB:=SaldoB+M;
write(SaldoB);

```

En una transferencia se modifican los saldos de dos cuentas. Ambos saldos deben modificarse o no debe modificarse ninguno (atomicidad).

```

Transferir(500,A,B)
Read(SaldoA);
if SaldoA > 500 then
  SaldoA:=SaldoA-500;
  Write(SaldoA);

```

FALLA

Valores iniciales:	Valores finales:
SaldoA = 1000	SaldoA = 500
SaldoB = 2000	SaldoB = 2000
SaldoC = 3000	SaldoC = 3000

Perdimos \$500 !

Si se realizan dos transferencias concurrentemente y estas acceden a la misma cuenta, puede violarse la serializabilidad y dejar la B.D en un estado inconsistente.

- Se necesita un protocolo de control de concurrencia

Transacciones: ejemplo 2

Transferir(500,A,B)	Transferir(500,A,C)
Read(SaldoA);	Read(SaldoA);
if SaldoA > 500 then SaldoA:=SaldoA-500; Write(SaldoA); Read(SaldoB); SaldoB:=SaldoB+500; Write(SaldoB);.	if SaldoA > 500 then SaldoA:=SaldoA-500; Write(SaldoA); Read(SaldoC); SaldoC:=SaldoC+500; Write(SaldoC);

Valores iniciales:	Valores finales:
SaldoA = 1000	SaldoA = 500
SaldoB = 2000	SaldoB = 2500
SaldoC = 3000	SaldoC = 3500

Inconsistencia!

4.5. Transacciones en MySQL

Las transacciones aportan una fiabilidad superior a las bases de datos [18 y 50]. Si disponemos de una serie de consultas SQL que deben ejecutarse en conjunto, con el uso de transacciones podemos tener la certeza de que nunca nos quedaremos a medio camino de su ejecución. De hecho, podríamos decir que las transacciones aportan una característica de "deshacer" a las aplicaciones de bases de datos.

Para este fin, las tablas que soportan transacciones, como es el caso de InnoDB, son mucho más seguras y fáciles de recuperar si se produce algún fallo en el servidor, ya que las consultas se ejecutan o no en su totalidad. Por otra parte, las transacciones pueden hacer que las consultas tarden más tiempo en ejecutarse.

Para asegurarnos que tenemos soporte para el tipo de tablas InnoDB podemos ejecutar la siguiente sentencia:

```
mysql> SHOW VARIABLES LIKE '%innodb%';
```

Variable_name	Value
have_innodb	YES
innodb_additional_mem_pool_size	1048576
innodb_buffer_pool_size	8388608
innodb_data_file_path	ibdata:30M
innodb_data_home_dir	
innodb_file_io_threads	4
innodb_force_recovery	0
innodb_thread_concurrency	8
innodb_flush_log_at_trx_commit	1
innodb_fast_shutdown	ON
innodb_flush_method	
innodb_lock_wait_timeout	50
innodb_log_arch_dir	.\
innodb_log_archive	OFF
innodb_log_buffer_size	1048576
innodb_log_file_size	5242880
innodb_log_files_in_group	2
innodb_log_group_home_dir	.\
innodb_mirrored_log_groups	1
innodb_max_dirty_pages_pct	90

20 rows in set (0.00 sec)

La variable más importante es por supuesto `have_innodb` que tiene el valor YES.

En efecto, una de las principales características de las tablas del tipo InnoDB es que pueden trabajar con transacciones, o sentencias SQL que son agrupadas como una sola.

Los pasos para usar transacciones en MySQL son:

- Iniciar una transacción con el uso de la sentencia `BEGIN`.
- Actualizar, insertar o eliminar registros en la base de datos.

- Si se quieren los cambios a la base de datos, completar la transacción con el uso de la sentencia COMMIT. Únicamente cuando se procesa un COMMIT los cambios hechos por las consultas serán permanentes.
- Si sucede algún problema, podemos hacer uso de la sentencia ROLLBACK para cancelar los cambios que han sido realizados por las consultas que han sido ejecutadas hasta el momento.

Vamos a ejecutar algunas consultas para ver cómo trabajan las transacciones. Lo primero que tenemos que hacer es crear una tabla del tipo InnoDB e insertar algunos datos.

Para crear una tabla InnoDB, procedemos con el código SQL estándar CREATE TABLE, pero debemos especificar que se trata de una tabla del tipo InnoDB (TYPE= InnoDB). Esto es aplicable a cualquier tipo de tabla, pero cuando no se especifica nada, MySQL supone que se trata de una tabla MyISAM.

```
mysql> CREATE TABLE innotest (campo INT NOT NULL PRIMARY KEY) TYPE = InnoDB;
Query OK, 0 rows affected (0.10 sec)
```

- Las transacciones están disponibles solo para las tablas de tipo InnoDB.
- Respetar las reglas ACID (ACID).
- Protocolo de bloqueo riguroso para control de concurrencia (mantiene los bloqueos exclusivos y compartidos hasta que la transacción comete).
- Bloqueo a nivel de fila de una tabla.
- Recuperación de fallos
- Detección y prevención de Deadlocks (timeout).
- Todas las sentencias SQL (select, update, insert, etc.) se ejecutan de manera atómica.
- Para ejecutar que una secuencia de sentencias SQL de manera atómica, hay que agruparlos dentro de una transacción.
- Para esto se utilizan dos sentencias SQL especiales: **start transaction** y **commit**.

```
start transaction;
sentencia_SQL_1;
```

```
.
```

```
sentencia_SQL_N;
commit;
```

- La sentencia **start transaction** comienza una transacción.
- La sentencia **commit** termina la transacción y almacena todos los cambios.
- Las sentencias SQL ejecutadas entre start transaction y commit se ejecutan atómicamente.
- La sentencia **rollback** puede utilizarse para retroceder la transacción y volver al estado previo al comienzo de la misma.

4.5.1. Transacciones en MySQL: bloqueos

Por default, las tablas InnoDB ejecutan un lectura consistente (consistent read). Esto significa que cuando una sentencia SELECT es ejecutada, MySQL regresa

los valores presentes en la base de datos hasta la transacción más reciente que ha sido completada. Si alguna transacción está en progreso, los cambios hechos por alguna sentencia INSERT o UPDATE no serán reflejados. Sin embargo, existe una excepción: las transacciones abiertas si pueden ver sus propios cambios.

- protocolo de bloqueo riguroso: mantiene todos los bloqueos hasta que la transacción comete.
- Dentro de una transacción, las sentencias **insert, delete, update** bloquean automáticamente las filas involucradas **en modo exclusivo**.
- Para la sentencia **select** debemos indicar el modo en el que vamos a acceder a los datos.
- **select ... lock in share mode** bloquea las filas involucradas en la consulta en **modo compartido**.
- **select ... for update** bloquea las filas involucradas en la consulta en **modo exclusivo**.

4.5.2. Transacciones en MySQL: ejemplos

Consideremos nuevamente la B.D. de un banco que mantiene el saldo de cada cuenta.

```
CREATE DATABASE bank;
USE bank;
CREATE TABLE cuentas(
    numero INT UNSIGNED NOT NULL,
    saldo DECIMAL(7,2) NOT NULL,
    PRIMARY KEY(numero)
) ENGINE=InnoDB;
INSERT INTO cuentas VALUES (1, 1000);
INSERT INTO cuentas VALUES (2, 2000);
INSERT INTO cuentas VALUES (3, 3000);
```

Transacciones en MySQL: ejemplo 1

- Transferir \$500 de la cuenta 1 a la cuenta 2:

```
mysql> use bank;
mysql> select * from cuentas;
```

numero	saldo
1	1000.00
2	2000.00
3	3000.00

```
mysql> start transaction;
mysql> select saldo from cuentas where numero=1 for update;
```

saldo
1000.00

```
mysql> update cuentas set saldo= saldo-500 where numero=1;
mysql> update cuentas set saldo= saldo+500 where numero=2;
mysql> commit;
mysql> select * from cuentas;
```

numero	saldo
1	500.00
2	2500.00
3	3000.00

Transacciones en MySQL: ejemplo 2

- Al Transferir \$500 de la cuenta 1 a la cuenta 2 simulamos una **falla** cortando la conexión con **exit**;}

```
mysql> use bank;
mysql> select * from cuentas;
```

numero	saldo
1	500.00
2	2000.00
3	3000.00

```
mysql> start transaction;
mysql> select saldo from cuentas where numero=1 for update;
```

saldo
500.00

```
mysql> update cuentas set saldo= saldo-500 where numero=1;
mysql> select * from cuentas;
```

numero	saldo
1	0.00
2	2500.00
3	3000.00

```
mysql> exit # simulamos una falla cortando la conexión
c:\Program Files\MySQL\MySQL Server 5.0\bin> mysql -u root
... # restablecemos la conexión
mysql> use bank;
mysql> select * from cuentas;
```

numero	saldo
1	500.00

```
| 2 | 2000.00 |
| 3 | 3000.00 |
+-----+
```

la transacción se deshizo automáticamente dejando la B.D.
en el estado previo a su ejecución

Transacciones en MySQL: ejemplo 3

- Dos transacciones concurrentes:
 - Conexión 1: T1= transferir \$500 de la cuenta 2 a la 1
 - Conexión 2: T2= transferir \$500 de la cuenta 2 a la 3

Conexión 1: T1= \$500 de 2 a 1	Conexión 2: T2= \$500 de 2 a 3
mysql> select * from cuentas; +-----+ numero saldo +-----+ 1 500.00 2 2500.00 3 3000.00 +-----+	
mysql> start transaction;	
	mysql> start transaction;
mysql> select saldo from cuentas where numero=2 for update; +-----+ saldo +-----+ 2500.00 +-----+	
	mysql> select saldo from cuentas where numero=2 for update;
mysql> update cuentas set saldo= saldo-500 where numero=2;	 T2 debe esperar por que T1 selecciono la cuenta 2 para actualizar. T2 queda bloqueada hasta que T1 comete y libera el dato.
mysql> update cuentas set saldo= saldo+500 where numero=1;	
mysql> commit;	
	+-----+ saldo +-----+ 2000.00 +-----+
	mysql> update cuentas set saldo= saldo-500 where numero=2;
	mysql> update cuentas set saldo= saldo+500 where numero=3;
	mysql> commit;

	<pre>mysql> select * from cuentas; +-----+-----+ numero saldo +-----+-----+ 1 1000.00 2 1500.00 3 3500.00 +-----+-----+</pre>
--	---

Transacciones en MySQL: ejemplo 4

- Dos transacciones concurrentes:
 - Conexión 1: T1= transferir \$500 de la cuenta 1 a la 3
 - Conexión 2: T2= transferir \$500 de la cuenta 2 a la 3

Conexión 1: T1= \$500 de 1 a 3	Conexión 2: T2= \$500 de 2 a 3
<pre>mysql> select * from cuentas; +-----+-----+ numero saldo +-----+-----+ 1 1000.00 2 1500.00 3 3500.00 +-----+-----+ 3 rows in set (0.00 sec)</pre>	
<pre>mysql> start transaction; Query OK, 0 rows affected ...</pre>	
<pre>mysql> select saldo from cuentas where numero=1 for update; +-----+ saldo +-----+ 1000.00 +-----+ 3 rows in set (0.00 sec)</pre>	<pre>mysql> start transaction; Query OK, 0 rows affected ...</pre>
	<pre>mysql> select saldo from cuentas where numero=2 for update; +-----+ saldo +-----+ 1500.00 +-----+ 3 rows in set (0.00 sec)</pre>
<pre>mysql> update cuentas set saldo= saldo-500 where numero=1; Query OK, 1 rows affected ...</pre>	
	<pre>mysql> update cuentas set saldo= saldo-500 where numero=2; Query OK, 1 rows affected ...</pre>
<pre>mysql> update cuentas set saldo= saldo+500 where numero=3; Query OK, 1 rows affected ...</pre>	

# bloquea implícitamente en # modo exclusivo la cuenta 3	
	mysql> update cuentas set saldo= saldo+500 where numero=3;
mysql>commit Query OK, 0 rows affected ...	 T2 debe esperar por que T1 actualizo la cuenta 2. T2 queda bloqueada hasta que T1 comete y libera el dato.
	Query OK,1 rows affected ...
	mysql> commit Query OK, 0 rows affected ...
	mysql> select * from cuentas; +-----+-----+ numero saldo +-----+-----+ 1 500.00 2 1000.00 3 4500.00 +-----+-----+

ANEXO A

Especificaciones Técnicas

Arquitectura Flexible

- Open Source
- Multi-threaded
- Pluggable Storage-Engine Architecture
 - MyISAM
 - InnoDB
 - Merge
 - Memory
 - Archive
 - Cluster

OLTP y Transacciones

- ACID Transactions
- Commit, Rollback
- Foreign Keys, Referential Integrity
- Row-level Locking
- Customizable Lock Isolation Levels
- Distributed Transactions (XA)
- Snapshot Repeatable Reads (readers don't block writers and vice-versa)
- Timed Deadlock Detection

ANSI SQL Standards

- ANSI SQL
- SubQueries, Joins, Cursors
- Prepared Statements
- Views
- Triggers
- Stored Procedures
- User-Defined Functions
- Information Schema
- Unicode
- Referential Integrity
- Server-enforced Data Integrity
- Strong Data type Support (e.g. Numeric, VARCHAR, BLOB, etc.)
- Optimized Datatype Support (e.g. Tinyint, Smallint, Mediumint, Bigint, etc.)

Replicación, Cluster & Alta Disponibilidad

- Built-in Replication
 - EngineMaster/Slave
 - Dual Master
 - Replication Ring
 - Replication Tree
- Proven Load Balancing

- Distributed AutoIncrement
- Main Memory Cluster Engine with instant failover (available separately)

RespalDOS

- Auto-Restart/Recovery
- Backup and Point-in-Time Recovery
- Logical Online Hot-Backup
- Physical Online Hot-Backup (available separately)

Manejo del Almacenamiento

- Hassle-Free Storage Administration
- Automatic Space Expansion
- Built-in Space Reclamation
- Auto Undo/Rollback Management
- Customizable Storage Assignments
- Compressed and Archive Tables
- Packed Index Storage
- Works with NAS or SAN storage

Características optimizadas para aplicaciones Web

- Server-side Thread Pool
- Connection Thread Caching
- Query Results Caching
- Load Balancing
- Prepared Statements
- Full Text Search
- Multiple Character Sets
- Geospatial Support
- Multi-Terabyte Scalability

Características optimizadas para Data Warehouse

- Fast Data Load Utility
- High-Speed Multi-Insert Function
- GROUP BY WITH ROLLUP
- Aggregate UDF
- Analytic SQL Functions
- Geospatial Support
- Archive Storage Engine for historical data
- Multi-Terabyte Scalability

Seguridad

- SSL Support
- Built-in Data Encryption/Decryption
- Granular External Security Control Fine Grained Object Privileges
- View Support
- Triggers for auditing
- Query Logs for auditing
- Archive Storage Engine for auditing

Fully backed by MySQL Network

- Certified Testing
- Indemnification
- 24x7 Production Support
- Online Knowledge Base
- Customized Advisors
- Targeted Alerts

Buen Desempeño

- Dynamic Memory Caches
- Unique Query Cache
- Multiple Index Type Support (B-tree, R-tree, Hash, etc.)
- Cost-based Optimizer
- High-speed Query Optimizer
- Diagnostics and SQL Tracing

Fácil de usar

- Fácil de instalar y configurar (aproximadamente 15 minutos para todas las plataformas)
- Cambios dinámicos en la configuración
- Información del esquema

Herramientas Gráficas

- MySQL Workbench
- MySQL Query Browser
- MySQL Administrator
- MySQL Migration Toolkit

Drivers

- MySQL Native C Library
- MySQL Drivers for ODBC, JDBC, .Net
- Community Drivers for PHP, Perl, Python, Ruby, etc
- MySQL Connector/MXJ for deployment as a JMX MBean

Plataformas

- 32 and 64 bit native binaries
- Operating Systems
 - Linux: RedHat, SuSE, Fedora, Debian
 - Unix: Solaris, HP-UX, AIX, SCO
 - BSD: FreeBSD
 - Apple: Mac OS
 - Windows: 2000, 2003, XP
 - Novell: NetWare
- CPUs
 - Intel x86, IA32, IA64
 - AMD Opteron
 - IBM PowerPC
 - Sun SPARC
- Additional platforms available

CONCLUSIONES

El control de transacciones concurrentes en una base de datos brinda un eficiente desempeño del Sistema de Base de Datos, puesto que permite controlar la ejecución de transacciones que operan en paralelo, accediendo a información compartida y, por lo tanto, interfiriendo potencialmente unas con otras.

El hecho de reservar un asiento en un avión mediante un sistema basado en aplicaciones web, cuando decenas de personas en el mundo pueden reservarlo también, nos da una idea de lo importante y crucial que es el control de concurrencia en un sistema de base de datos a mediana o gran escala.

Otro ejemplo en el que podemos observar la incidencia del control de concurrencia en el siguiente: en una Base de Datos bancaria podría ocurrir que se paguen dos cheques en forma simultánea sobre una cuenta que no tiene saldo suficiente para cubrirlos en su totalidad, esto es posible evitarlo si se tiene un control de concurrencia.

Informalmente, una transacción es la ejecución de ciertas instrucciones que accedan a una base de datos compartida. El objetivo del control de concurrencia y recuperación es asegurar que dichas transacciones se ejecuten atómicamente.

Cabe resaltar que la contribución de esta investigación, va mas allá de aquellos trabajos de tesis, que se enfocan mas en el desarrollo de Aplicaciones en Base de Datos, solo que no abordan los aspectos complementarios como en este trabajo que es la administración de transacciones, la recuperación y el control de concurrencia en Base de Datos.

Como experiencia personal, se comprobó que MySQL es un gestor de base de datos interesante, sencillo de usar e increíblemente rápido, así como también es uno de los motores de base de datos más usados en Internet, la principal razón de esto es que es gratis para aplicaciones no comerciales, entre las mayores ventajas que ofrece MySQL entre otros es que es una base de datos relacional, el código fuente de MySQL se puede descargar y está accesible a cualquiera, una base de datos muy rápida, segura y fácil de usar, además de que existe una gran cantidad de software que la usa.

MySQL ofrece uno de los motores de bases de datos transaccionales más potentes del mercado. Las características incluyen un soporte completo de ACID (atómica, consistente, aislada, duradera), bloqueo a nivel de filas, posibilidad de transacciones distribuidas, y soporte de transacciones con múltiples versiones donde los lectores no bloquean a los escritores y viceversa. También se asegura una integridad completa de los datos mediante integridad referencial, niveles de aislamiento de transacciones especializados y detección de *deadlocks*.

El poder aprender una herramienta tan importante como MySQL, la cual puede interactuar con otras (PHP, Perl, etc.) y complementarse, ha sido de mucha satisfacción, ya que en un futuro me puede servir para el desarrollo de aplicaciones o bien el poder compartir dichos conocimientos con mis compañeros, así como lo importante que es el estar actualizando en otras herramientas.

BIBLIOGRAFIA

- [1] Adams, E. W. Honda, M., and Miller, T.C. 1989. Object Management in a CASE Environment. In *11th International Conference on Software Engineering*, 154-163. IEEE Computer Society Press, Pittsburgh.
- [2] Barghouti, N.S. 1992. *Concurrency Control in Rule-Based Software Development Environments*. PhD thesis, Columbia University
- [3] Barghouti N.S., and Kaiser, G.E. 1991. Concurrency Control in Advanced Database Applications. *ACM Computing Surveys Vol. 23*, No. 3, 269-317.
- [4] Ben-Shaul, I.Z., Kaiser, G.E., and Heineman, G.T. 1993. An Architecture for Multi-User Software Development Environments. *Computing Systems, The Journal of the USENIX Association, Vol. 6*, No. 2, 65-103.
- [5] Bernstein, P.A. , Hadzilacos V., and Goodman N. 1987. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading Mass.
- [6] Chrysanthis P.K. and Ramamritham K. 1990. ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behavior. *1990 ACM SIGMOD International Conference on Management of Data*, 194-203. Atlantic City.
- [7] Date, C. J., 2001. Introducción a los Sistemas de Bases de Datos, 1ª, 2ª y 7ª. Edición. Addison-Wesley Iberoamericana. (Capitulos 10, 12, 14 y 15).
- [8] Davies, C.T., Jr. 1973. Recovery Semantics for A DB/DC System. Twenty-eighth *ACM National Conference*, 136-141. Atlanta.
- [9] Davies, C. T. Jr. 1978. Data Processing Spheres of Control. *IBM Systems Journal, Vol. 17*, No. 2, 179-198.
- [10] Elmasri, R.; Navarthe, S.B. 2002 Fundamentos de Sistemas de Base de Datos Sistemas de Bases de Datos, Conceptos Fundamentales. 3ª Edición, Addison-Wesley Iberoamericana. (Capitulo 19).
- [11] Elmasri, R.; Navarthe, S.B. 1997. Sistemas de bases de datos. Conceptos fundamentales. 2ª Edición, Addison-Wesley Iberoamericana. (Capitulo 17).
- [12] Ege, A., and Ellis, C.A. 1987. Design and Implementation of Gordion, an Object Base Management System. *Third International Conference on Data Engineering*, 226-234. Los Angeles.
- [13] Estublier, J., Ghoul, S., and Krakowiak, S. 1984. Preliminary Experience with a Configuration Control System for Modular Programs. In *ACM SIGSOFT/SIGPLAN SoftwareEngineering Symposium on Practical Software Development Environments*, 149-156. P. Henderson, ed.
- [14] Eswaran, K. P., Gray, J.N. Lorie, R.A., and Traiger, I.L. 1976. The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM* , Vol. 19, No. 11, 624-632.

- [15] Fernandez, M.F., and Zdonik, S.B. 1989. Transaction Groups: A Model for Controlling Cooperative Work. *Third International Workshop on Persistent Object Systems: Their Design, Implementation and Use*, 128-138. Queensland, Australia.
- [16] Garcia-Molina, H., and Salem, K. 1978. SAGAS. *ACM SIGMOD 1987 Annual Conference*, 249-259. San Francisco.
- [17] Gray, J., Lorie, R., and Putzolu, G. 1975. Granularity of Locks and Degrees of Consistency in a Shared Database. *International Conference on Very Large Data Bases*, 428-451. Morgan Kaufmann.
- [18] Gilfillan, I., 2003, My SQL, Anaya Multimedia, CAP 1 al 15.
- [19] Heineman, G.T. 1993. *A Transaction Manager Component for Cooperative Transaction Models*. Technical Report CUCS-017-93, Columbia University Department of Computer Science.
- [20] Henry F. Korth, Abraham Silberschatz. Fundamentos de Bases de Datos. University of Texas at Austin, 1ª. Edición, McGraw Hill.
- [21] Henry F. Korth, Abraham Silberschatz. Fundamentos de Bases de Datos. University of Texas at Austin, 2ª. Edición, McGraw Hill.
- [22] Honda, M. 1988. Support for Parallel Development in the Sun Network Software Environment. *Second International Workshop on Computer-Aided Software Engineering*, 5-5 - 5-7.
- [23] Hornick, M.F., and Zdonik, S.B. 1987. A Shared, Segmented Memory System for an Object-Oriented Database. *ACM Transactions on Office Automation Systems Vol. 5*, No. 1, 70-95.
- [24] Horwitz, S., Prins, J., and Reps, T. 1989. Integrating Noninterfering Versions of Programs. *ACM Transactions on Programming Languages and Systems, Vol. 11*, No. 3, 345-387.
- [25] Johnson, S.C. 1978. Lint, a C Program Checker. *Unix Programmer's Manual*. AT&T Bell Laboratories.
- [26] Kaiser, G.E. 1990. A Flexible Transaction Model for Software Engineering. *Sixth International Conference on Data Engineering*, 560-567. IEEE Computer Society Press, Los Angeles.
- [27] Kaiser, G.E. 1991. Interfacing Cooperative Transactions to Software Development Environments. *Office Knowledge Engineering Vol. 4*, No. 1, 56-78.
- [28] Kaiser, G.E., and Feiler, P.H. 1987. Intelligent Assistance without Artificial Intelligence. *Thirty-second IEEE Computer Society International Conference*, 236-241. IEEE Computer Society Press, San Francisco.

- [29] Kaiser, G.E., and Perry, D.E. 1987. Workspaces and Experimental Databases: Automated Support for Software Maintenance and Evolution. *Conference on Software Maintenance*, 108-114. IEEE Computer Society Press, Austin.
- [30] Kaiser, G.E., and Perry, D.E. 1991. Making Progress in Cooperative Transaction Models. *Data Engineering*, Vol. 14, No. 1, 19-23.
- [31] Kaiser, G.E., and Perry, D.E. 1987. Infuse: Fusing Integration Test Management with Change Management. In *COMPSAC 89: The Thirteenth Annual International Computer Software & Applications Conference*, 552-558. IEEE Computer Society Press, Orlando.
- [32] Kaiser, G.E., and Pu, C. 1992. Dynamic Restructuring of Transactions. In: *Database Transaction Models for Advanced Applications*, 265-295. A.K. Elmagarmid, ed. Morgan Kaufmann, San Mateo, Cal.
- [33] Katz, R.H. 1990. Toward a Unified Framework for Version Modeling in Engineering Databases. *ACM Computing Surveys*, Vol. 22, No. 4, 375-408.
- [34] Kim, W., Ballou, N., Garza, J.F. and Woelk, D. 1991. A Distributed Object-Oriented Database System Supporting Shared and Private Databases. *ACM Transactions on Information Systems*, Vol. 9, No. 1, 31-51.
- [35] Kim, W. 1995. Modern Database Systems: The Object Model, Interoperability, and Beyond, ACM Press, Cap. 20, Cooperative Transactions for Multi-User Environments, Kaiser, G.E., 409-433.
- [36] Klahold, P., Schlageter, G., Unland., R., and Wilkes, W. 1985. A Transaction Model Supporting Complex Applications in Integrated Information Systems. In *ACM-SIGMOD 1985 International Conference on Management of Data*, 388-401. Austin.
- [37] Kung, H.T., and Robinson, J. 1981. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems* Vol. 6, No. 2, 213-226.
- [38] Leblang, D.B. and Chase, R.P., Jr. 1984. Computer-Aided Software Engineering in a Distributed Workstation Environment. In: *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, 104-112. P. Henderson, ed. ACM Press.
- [39] Leblang, D.B. and Chase, R.P., Jr. 1987. Parallel Software Configuration Management in a Network Environment. *IEEE Software* Vol. 4, No. 6, 28-35.
- [40] Moss, J.E.B. 1982. Nested Transactions and Reliable Distributed Computing. *Second Symposium on Reliability in Distributed Software and Database Systems*, 33-39. IEEE Computer Society Press, Pittsburgh..
- [41] Perry, D.E., and Kaiser, G.E. 1991. Models of Software Development Environments. *IEEE Transactions on Software Engineering* Vol. 17, No. 3, 283-295.

- [42] Pu, C., Kaiser, G.E. and Hutchinson, N. 1988. Split-Transactions for Open-Ended Activities. In Francois Bancilhon and David J. Dewitt (editor), *14th International Conference on Very Large Data Bases*, 26-37. Morgan Kaufman, Los Angeles.
- [43] Reed, D.P. 1978. *Naming and Synchronization in a Decentralized Computer System*. Ph.D. dissertation, Massachusetts Institute of Technology.
- [44] Rochkind, M.J. 1975. The Source Code Control System. *IEEE Transactions on Software Engineering* SE-1, 364-370.
- [45] Salem, K. 1993. *Impleemnting Extended Transaction Models Using Transaction Groups*. Technical Report CS-TR-3051, University of Maryland Department of Computer Science.
- [46] Schwanke, R.W., Cohen, , E.S., Gluecker, R., Hasling, W.M., Soni, D.A., and Wagner M.E. 1989. Configuration Management in BiiN SMS. In *11th International Conference on Software Engineering*, 383-393. IEEE Computer Society Press, Pittsburgh.
- [47] Skarra, A.H. 1991. Localized Correctness Specifications for Cooperating Transactions in an Object-Oriented Database. *Office Knowledge Engineering Vol. 4*, No. 1, 79-106.
- [48] Tichy, W.F. 1986. RCS — A System for Version Control. *Software — Practice & Experience Vol. 15*, No. 7, 637-654.
- [49] Tichy, W.F. 1986. Smart Recompilation. *ACM Transactions on Programming Languages and Systems Vol. 8*, No. 3, 273-291.
- [50] Varios Autores, 1997-2002, act. 2010. Manual de Referencia MySQL 5.0. MySQL AB
- [51] Walpole, J., Blair, G.S., Malik, J., and Nicol, J.R. 1988. A Unifying Model for Consistent Distributed Software Development Environments. *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, 183-190.
- [52] Walpole, J., Blair, G.S., Malik, J., and Nicol, J.R. 1988... Maintaining Consistency in Distributed Software Engineering Environments. In *8th International Conference on Distributed Computing Systems*, 418-425. IEEE Computer Society Press, San Jose, Cal.
- [53] Yeh, S., Ellis, C., Ege, A., and Korth, H. 1987. *Performance Analysis of Two Concurrency Control Schemas for Design Environments*. Technical Report STP-036-87.