1

BASES DE DATOS DISTRIBUIDAS

TEMA 4

PROFESOR: M.C. ALEJANDRO GUTIÉRREZ DÍAZ

2

4. MANEJO DE TRANSACCIONES

4.1 Conceptos de Transacciones

4.2 Control de concurrencia

4.3 Serialización de transacciones

4.4 Algoritmos de control de concurrencia

3

Conceptos de Transacciones

Hasta este momento, las primitivas básicas de acceso que se han considerado son las consultas (queries). Sin embargo, no se ha discutido qué pasa cuando, por ejemplo, dos consultas tratan de actualizar el mismo elemento de datos, o si ocurre una falla del sistema durante la ejecución de una consulta.

Dada la naturaleza de una consulta de lectura o actualización, a veces no se puede simplemente reactivar la ejecución de una consulta, puesto que algunos datos pueden haber sido modificados antes de la falla. El no tomar en cuenta esos factores puede conducir a que la información en la base de datos contenga datos incorrectos.

El concepto fundamental aquí es la noción de "ejecución consistente" o "procesamiento confiable" asociada con el concepto de una consulta. El concepto de una *transacción* es usado dentro del dominio de la base de datos como una unidad básica de cómputo consistente y confiable.

4

Definición de una transacción

Una ***transacción*** es una colección de acciones que hacen transformaciones consistentes de los estados de un sistema preservando la consistencia del sistema. Una base de datos está en un estado

consistente si obedece todas las restricciones de integridad definidas sobre ella. Los cambios de estado ocurren debido a actualizaciones, inserciones, y supresiones de información. Por supuesto, se quiere asegurar que la base de datos nunca entra en un estado de inconsistencia. Sin embargo, durante la ejecución de una transacción, la base de datos puede estar temporalmente en un estado inconsistente. El punto importante aquí es asegurar que la base de datos regresa a un estado consistente al fin de la ejecución de una transacción (Ver Figura 5.1)

Lo que se persigue con el manejo de transacciones es por un lado tener una transparencia adecuada de las acciones concurrentes a una base de

datos y por otro lado tener una transparencia adecuada en el manejo de las fallas que se pueden presentar en una base de datos.

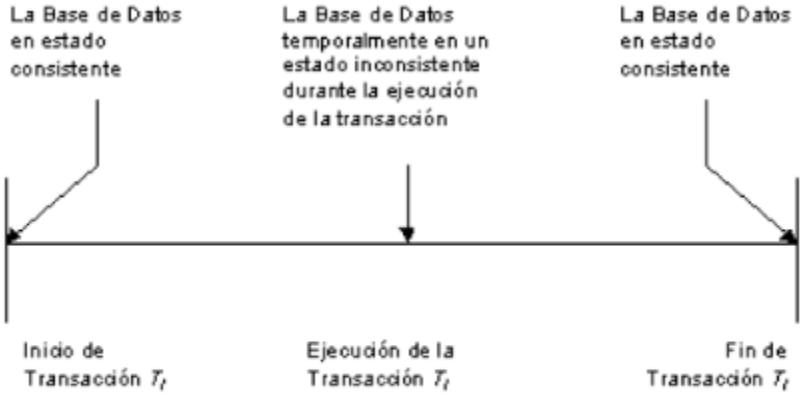


Figura 5.1. Un modelo de transacción.

Figura 5.1. Un modelo de transacción.

5

Ejemplo 5.2. Considere una agencia de reservaciones para líneas aéreas con las siguientes relaciones:

FLIGHT(FNO, DATE, SRC, DEST, STSOLD, CAP)

CUST(CNAME, ADDR, BAL)

FC(FNO, DATE, CNAME, SPECIAL)

Una versión simplificada de una reservación típica puede ser implementada mediante la siguiente transacción:

Begin_transaction Reservación

begin

input(flight_no, date, customer_name);

EXEC SQL UPDATE FLIGHT

SET STSOLD = STSOLD + 1
WHERE FNO = flight_no
AND DATE = date
EXEC SQL INSERT
INTO FC(FNAME, DATE, CNAME, SPECIAL)
VALUES (flight_no, date, customer_name, *null*
)
output("reservación terminada");
end.
BASES DE DATOSDISTRIBUIDAS MIS 515
4

6

Condiciones de terminación de una transacción
Una transacción siempre termina, aun en la presencia de fallas. Si una transacción termina de manera exitosa se dice que la transacción hace un *commit* (se usará el término en inglés cuando no exista un término en español que refleje con brevedad el sentido del término en inglés). Si la transacción se detiene sin terminar su tarea, se dice que la transacción *aborta*. Cuando la transacción es abortada, su ejecución es detenida y todas sus acciones ejecutadas hasta el momento son deshechas (*undone*) regresando a la base de datos al estado antes de su ejecución. A esta operación también se le conoce como *rollback*.

7

Ejemplo 5.3. Considerando de nuevo el Ejemplo 5.2, veamos el caso cuando no esisten asientos disponibles para hacer la reservación.
Begin_transaction Reservación
begin
input(flight_no, date, customer_name);
EXEC SQL SELECT STSOLD, CAP
INTO temp1, temp2
FROM FLIGHT
WHERE FNO = flight_no AND DATE = date
if temp1 = temp2 **then**
output("no hay asientos libres")
Abort
else
EXEC SQL UPDATE FLIGHT
SET STSOLD = STSOLD +
1
WHERE FNO = flight_no
AND DATE = date
EXEC SQL INSERT

INTO FC(FNAME, DATE,
CNAME, SPECIAL)
VALUES (flight_no, date,
customer_name, *null*)
Commit
output("reservación terminada");
endif
end.

8

Caracterización de transacciones

Observe que en el ejemplo anterior las transacciones leen y escriben datos. Estas acciones se utilizan como base para caracterizar a las transacciones. Los elementos de datos que lee una transacción se dice que constituyen el *conjunto de lectura* (*RS*). Similarmente, los elementos de datos que una transacción escribe se les denominan el *conjunto de escritura* (*WS*). Note que los conjuntos de lectura y escritura no tienen que ser necesariamente disjuntos. La unión de ambos conjuntos se le conoce como el *conjunto base* de la transacción $F = (FR \cup DWS)$

Ejemplo 5.4. Los conjuntos de lectura y escritura de la transacción del Ejemplo 5.3 son:

RS[Reservación] = { FLIGHT.STSOLD, FLIGHT.CAP }
WS[Reservación] = { FLIGHT.STSOLD, FC.FNO, FC.DATE, FC.NAME, FC.SPECIAL }

9

Propiedades de las transacciones

La discusión en la sección previa clarifica el concepto de transacción. Sin embargo, aun no se ha dado ninguna justificación para afirmar que las transacciones son unidades de procesamiento consistentes y confiables. Las propiedades de una transacción son las siguientes:

1. ***Atomicidad***. Se refiere al hecho de que una transacción se trata como una unidad de operación. Por lo tanto, o todas las acciones de la transacción se realizan o ninguna de ellas se lleva a cabo. La atomicidad requiere que si una transacción se interrumpe por una falla, sus resultados parciales deben ser deshechos. La actividad referente a preservar la atomicidad de transacciones en presencia de abortos debido a errores de entrada, sobrecarga del sistema o

interbloqueos se le llama *recuperación de transacciones*. La actividad de asegurar la atomicidad en presencia de caídas del sistema se le llama *recuperación de caídas*.

2. ***Consistencia***. La consistencia de una transacción es simplemente su correctitud. En otras palabras, una transacción es un programa correcto que lleva la base de datos de un estado consistente a otro con la misma característica. Debido a esto, las transacciones no violan las restricciones de integridad de una base de datos.

3. ***Aislamiento***. Una transacción en ejecución no puede revelar sus resultados a otras transacciones concurrentes antes de su *commit*. Más aún, si varias transacciones se ejecutan concurrentemente, los resultados deben ser los mismos que si ellas se hubieran ejecutado de manera secuencial (seriabilidad).

4. ***Durabilidad***. Es la propiedad de las transacciones que asegura que una vez que una transacción hace su *commit*, sus resultados son permanentes y no pueden ser borrados de la base de datos. Por lo tanto, los DBMS aseguran que los resultados de una transacción sobrevivirán a fallas del sistema. Esta propiedad motiva el aspecto de *recuperación de bases de datos*, el cual trata sobre como recuperar la base de datos a un estado consistente en donde todas las acciones que han hecho un commit queden reflejadas.

En resumen, las transacciones proporcionan una ejecución atómica y confiable en presencia de fallas, una ejecución correcta en presencia de accesos de usuario múltiples y un manejo correcto de réplicas (en el caso de que se soporten).

10

Tipos de Transacciones

Las transacciones pueden pertenecer a varias clases. Aun cuando los problemas fundamentales son los mismos para las diferentes clases, los

algoritmos y técnicas que se usan para tratarlas pueden ser considerablemente diferentes. Las transacciones pueden ser agrupadas a lo largo de las siguientes dimensiones:

1. **Aéreas de aplicación**. En primer lugar, las transacciones se pueden ejecutar en aplicaciones no distribuidas. Las transacciones que operan en datos distribuidos se les conoce como transacciones distribuidas. Por otro lado, dado que los resultados de una transacción que realiza un commit son durables, la única forma de deshacer los efectos de una transacción con commit es mediante otra transacción. A este tipo de transacciones se les conoce como transacciones *compensatorias*. Finalmente, en

ambientes heterogéneos se presentan transacciones *heterogéneas* sobre los datos.
2. **Tiempo de duración**. Tomando en cuenta el tiempo que transcurre desde que se inicia una transacción hasta que se realiza un commit o se aborta, las transacciones pueden ser de tipo batch o en línea. Estas se pueden diferencias también como transacciones de corta y larga vida. Las transacciones en línea se caracterizan por tiempos de respuesta muy cortos y por accesar un porción relativamente pequeña de la base de datos. Por otro lado, las transacciones de tipo batch toman tiempos relativamente largos y accesan grandes porciones de la base de datos.
3. **Estructura**. Considerando la estructura que puede tener una transacción se examinan dos aspectos: si una transacción puede contener a su vez subtransacciones o el orden de las acciones de lectura y escritura dentro de una transacción.

11

Estructura de transacciones

Las transacciones planas consisten de una secuencia de operaciones primitivas encerradas entre las palabras clave **begin** y **end**. Por ejemplo,
Begin_transaction Reservación
. . .
end.

En las transacciones anidadas las operaciones de una transacción pueden ser así mismo transacciones.
Una transacción anidada dentro de otra transacción conserva las mismas propiedades que la de sus padres, esto implica, que puede contener así mismo transacciones dentro de ella. Existen restricciones obvias en una transacción anidada: debe empezar *después* que su padre y debe terminar *antes* que él. Más aún, el commit de una subtransacción es condicional al commit de su padre, en otras palabras, si el padre de una o varias transacciones aborta, las subtransacciones hijas también serán abortadas.

12

Aspectos relacionados al procesamiento de transacciones

Los siguientes son los aspectos más importantes relacionados con el procesamiento de transacciones:
1. Modelo de estructura de transacciones. Es importante considerar si las transacciones son planas o pueden estar anidadas.
2. Consistencia de la base de datos interna. Los algoritmos de control de datos semántico tienen que satisfacer siempre las restricciones

de integridad cuando una transacción pretende hacer un commit.
3. Protocolos de confiabilidad. En transacciones distribuidas es necesario introducir medios de comunicación entre los diferentes nodos de una red para garantizar la atomicidad y durabilidad de las transacciones. Así también, se requieren protocolos para la recuperación local y para efectuar los compromisos (commit) globales.
4. Algoritmos de control de concurrencia. Los algoritmos de control de concurrencia deben sincronizar la ejecución de transacciones concurrentes bajo el criterio de correctitud. La consistencia entre transacciones se garantiza mediante el aislamiento de las mismas.
5. Protocolos de control de réplicas. El control de réplicas se refiere a cómo garantizar la consistencia mutua de datos replicados. Por ejemplo se puede seguir la estrategia read-one-write-all (ROWA).

13

Incorporación del manejador de transacciones a la arquitectura de un SMBD

Con la introducción del concepto de transacción, se requiere revisar el modelo arquitectural presentado en el capítulo 2. En esta sección, se expande el papel del monitor de ejecución distribuida.
El monitor de ejecución distribuida consiste de dos módulos: El *administrador de transacciones* (TM) y el *despachador* (SC). Como se puede apreciar en la Figura 5.2, el administrador de transacciones es responsable de coordinar la ejecución en la base de datos de las operaciones que realiza una aplicación. El despachador, por otra parte, es responsable de implementar un algoritmo específico de control de concurrencia para sincronizar los accesos a la base de datos.
Un tercer componente que participa en el manejo de transacciones distribuidas es el *administrador de recuperación local* cuya función es implementar procedimientos locales que le permitan a una base de datos local recuperarse a un estado consistente después de una falla.

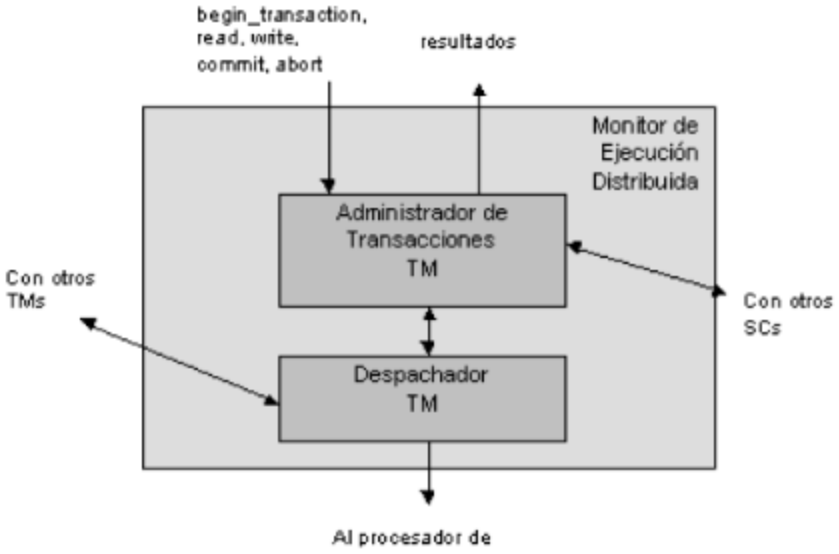
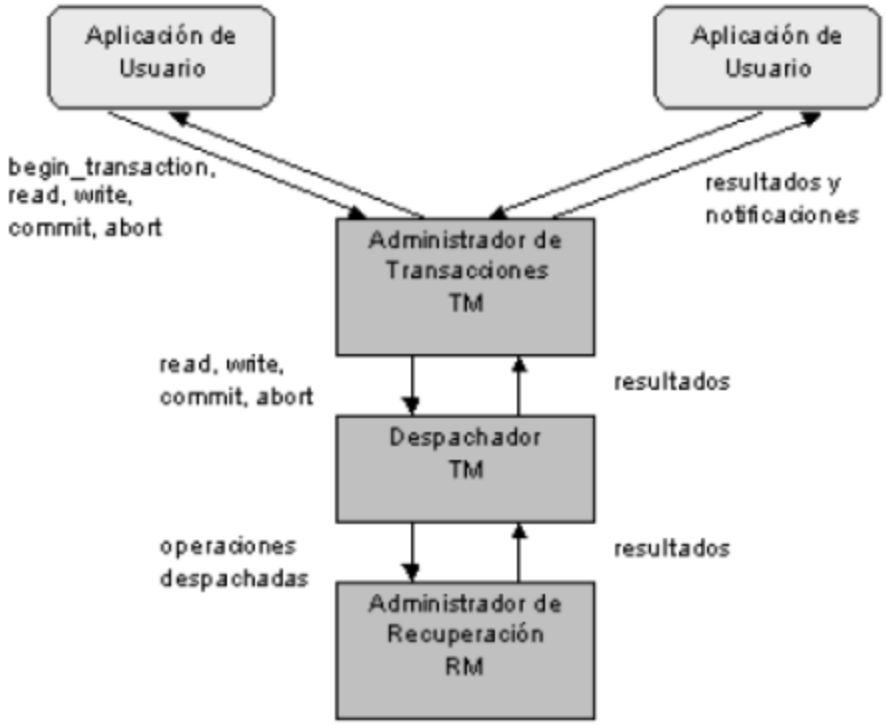


Figura 5.2. Un modelo del administrador de transacciones.

14

Los administradores de transacciones implementan una interfaz para los programas de aplicación que consiste de los comandos:

1. **Begin_transaction**.
2. **Read**.
3. **Write**.
4. **Commit**.
5. **Abort**.



En la Figura 5.3 se presenta la arquitectura requerida para la ejecución centralizada de transacciones. Las modificaciones requeridas en la arquitectura para una ejecución distribuida se pueden apreciar en las Figura 5.4. En esta última figura se presentan también los protocolos de comunicación necesarios para el manejo de transacciones distribuidas.

Figura 5.3. Ejecución centralizada de transacciones.

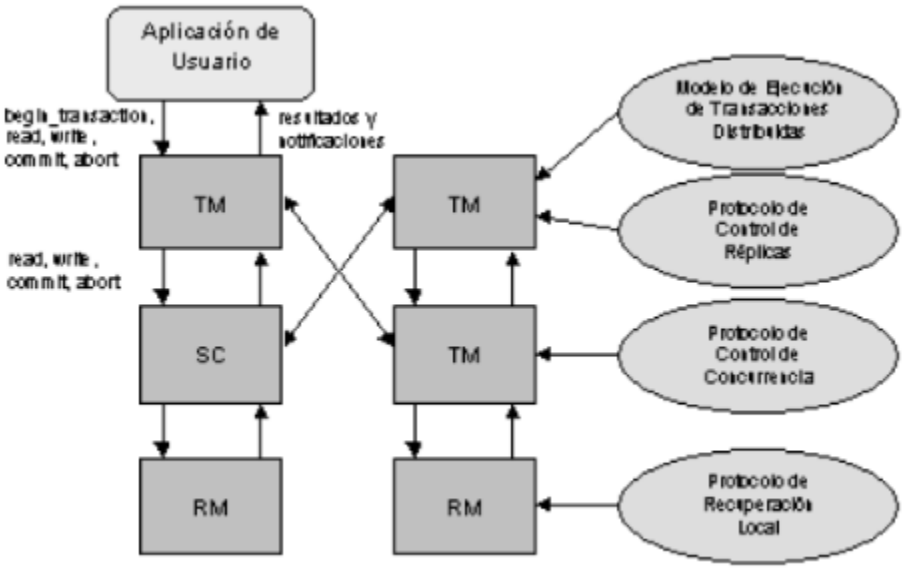


Figura 5.4. Ejecución distribuida de transacciones.

15

BREAK

16

Control de concurrencia

El control de concurrencia trata con los problemas de aislamiento y consistencia del procesamiento de transacciones. El control de concurrencia distribuido de una DDBMS asegura que la consistencia de la base de datos se mantiene en un ambiente distribuido multiusuario.

Si las transacciones son internamente consistentes, la manera más simple de lograr este objetivo es ejecutar cada transacción sola, una después de otra. Sin embargo, esto puede afectar grandemente el desempeño de un DDBMS dado que el nivel de concurrencia se reduce al mínimo.

El nivel de concurrencia, el número de transacciones activas, es probablemente el parámetro más importante en sistemas distribuidos. Por lo tanto, los

mecanismos de control de concurrencia buscan encontrar un balance entre el mantenimiento de la consistencia de la base de datos y el mantenimiento de un alto nivel de concurrencia.

Si no se hace un adecuado control de concurrencia, se pueden presentar dos anomalías. En primer lugar, se pueden perder actualizaciones provocando que los efectos de algunas transacciones no se reflejen en la base de datos. En segundo término, pueden presentarse recuperaciones de información inconsistentes.

En este capítulo se hace la suposición de que el sistema distribuido es completamente confiable y no experimente falla alguna. En el capítulo siguiente, se considerará la presencia de fallas para obtener sistemas confiables.

17

4.3 Serialización de transacciones

Teoría de la seriabilidad

Una *calendarización* (*schedule*), también llamado una *historia*, se define sobre un conjunto de transacciones $T = \{ T_1, T_2, ..., T_n \}$ y especifica un orden entrelazado de la ejecución de las operaciones de las transacciones. La calendarización puede ser especificada como un orden parcial sobre T.

Ejemplo 6.1. Considere las siguientes tres transacciones:

T_1: Read(x) T_2: Write(x) T_3: Read(x)

Write(x) Write(y) Read(y)

Commit Read(z) Read(z)
Commit Commit
Una calendarización de las acciones de las tres transacciones anteriores puede ser:
H_1 = { $W_2(x)$, $R_1(x)$, $R_3(x)$, $W_1(x)$, C_1, $W_2(y)$, $R_3(y)$, $R_2(z)$, C_2, $R_3(z)$, C_3 }
I

Dos operaciones $O_{ij}(x)$ y $O_{kl}(x)$ (i y k no necesariamente distintos) que accedan el mismo dato de la base de datos x se dice que están en *conflicto* si al menos una de ellas es una escritura. De esta manera, las operaciones de lectura no tienen conflictos consigo mismas.
Por tanto, existen dos tipos de conflictos *read-write* (o *write-read*) y *writewrite*. Las dos operaciones en conflicto pueden pertenecer a la misma transacción o a transacciones diferentes.

18

Seriabilidad en SMBD distribuidos
En bases de datos distribuidas es necesario considerar dos tipos de historia para poder generar calendarizaciones serializables: la calendarización de la ejecución de transacciones en un nodo conocido como *calendarización local* y la *calendarización global* de las transacciones en el sistema. Para que las transacciones globales sean serializables se deben satisfacer las siguientes condiciones:
- cada historia local debe ser serializable, y
- dos operaciones en conflicto deben estar en el mismo orden relativo en todas las historias locales donde las operaciones aparecen juntas.

La segunda condición simplemente asegura que el orden de serialización sea el mismo en todos los nodos en donde las transacciones en conflicto se ejecutan juntas.
Ejemplo 6.6. Considere las siguientes transacciones:
T_1: Read(x) T_2: Read(x)
x ← x + 5 x ← x * 5
Write(x) Write(x)
Commit Commit
Las siguientes historias locales son individualmente serializables (de hecho son seriales),
LH_1 = { $R_1(x)$, $W_1(x)$, C_1, $R_2(x)$, $W_2(x)$, C_2 }

LH_2 = { $R_2(x)$, $W_2(x)$, C_2, $R_1(x)$, $W_1(x)$, C_1 }

19

4.4 Algoritmos de control de concurrencia

El criterio de clasificación más común de los algoritmos de control de concurrencia es el tipo de primitiva de sincronización. Esto resulta en dos clases: aquellos algoritmos que están basados en acceso mutuamente exclusivo a datos compartidos (candados) y aquellos que intentar ordenar la ejecución de las transacciones de acuerdo a un conjunto de reglas (protocolos).

Sin embargo, esas primitivas se pueden usar en algoritmos con dos puntos de vista diferentes: el punto de vista pesimista que considera que muchas transacciones tienen conflictos con otras, o el punto de vista optimista que supone que no se presentan muchos conflictos entre transacciones.

Los algoritmos *pesimistas* sincronizan la ejecución concurrente de las transacciones en su etapa inicial de su ciclo de ejecución. Los algoritmos *optimistas* retrasan la sincronización de las transacciones hasta su terminación.

El grupo de algoritmos pesimistas consiste de algoritmos *basados en candados*, algoritmos *basados en ordenamiento por estampas de tiempo* y algoritmos *híbridos*. El grupo de los algoritmos optimistas se clasifican por basados en candados y basados en estampas de tiempo (Ver. Figura 6.1).

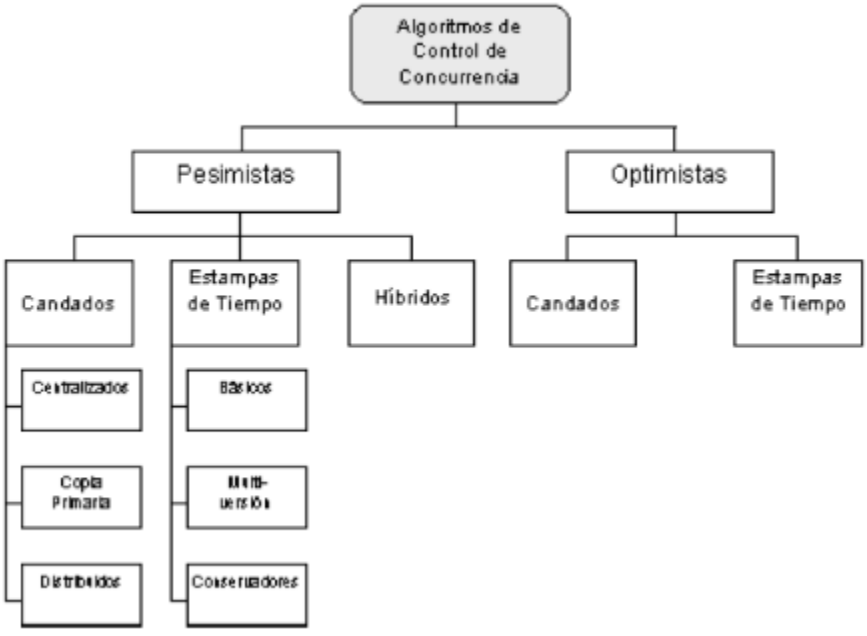


Figura 6.1. Clasificación de los algoritmos de control de concurrencia.

20

Algoritmos basados en candados

En los algoritmos basados en candados, las transacciones indican sus intenciones solicitando candados al despachador (llamado el *administrador de candados*). Los candados son de lectura (*rl*), también llamados *compartidos*, o de escritura (*wl*), también llamados *exclusivos*. Como se aprecia en la tabla siguiente, los candados de lectura presentan conflictos con los candados de escritura, dado que las operaciones de lectura y escritura son incompatibles.

rl wl

rl si no

wl no no

En sistemas basados en candados, el despachador es un *administrador de candados* (LM). El administrador de transacciones le pasa al administrador de candados la operación sobre la base de datos (lectura o

escritura) e información asociada, como por ejemplo el elemento de datos que es accesado y el identificador de la transacción que está enviando la operación a la base de datos.

El administrador de candados verifica si el elemento de datos que se quiere accesar ya ha sido bloqueado por un candado. Si candado solicitado es incompatible con el candado con que el dato está bloqueado, entonces, la transacción solicitante es retrasada.

De otra forma, el candado se define sobre el dato en el modo deseado y la operación a la base de datos es transferida al procesador de datos. El administrador de transacciones es informado luego sobre el resultado de la operación.

La terminación de una transacción libera todos los candados y se puede iniciar otra transacción que estaba esperando el acceso al mismo dato.

21

Candados de dos fases (2PL)

En los candados de dos fases una transacción le pone un candado a un objeto antes de usarlo. Cuando un objeto es bloqueado con un candado por otra transacción, la transacción solicitante debe esperar.

Cuando una transacción libera un candado, ya no puede solicitar más candados. Así una transacción que utiliza candados de dos fases se comporta como en la Figura izquierda. En la primera fase solicita y adquiere todos los candados sobre los elementos que va a utilizar y en la segunda fase libera los candados obtenidos uno por uno.

La importancia de los candados de dos fases es que se ha demostrado de manera teórica que todos las calendarizaciones generadas por

algoritmos de control de concurrencia que obedecen a los candados de dos fases son serializables.
Puede suceder que si una transacción aborta después de liberar un candado, otras transacciones que hayan accesado el mismo elemento de datos aborten también provocando lo que se conoce como *abortos en cascada*.

Para evitar lo anterior, los despachadores para candados de dos fases implementan lo que se conoce como los *candados estrictos de dos fases* en los cuales se liberan todos los candados juntos cuando la transacción termina (con commit o aborta). El comportamiento anterior se muestra en la Figura derecha
Figura 6.2. Gráfica del uso de los candados de dos fases.

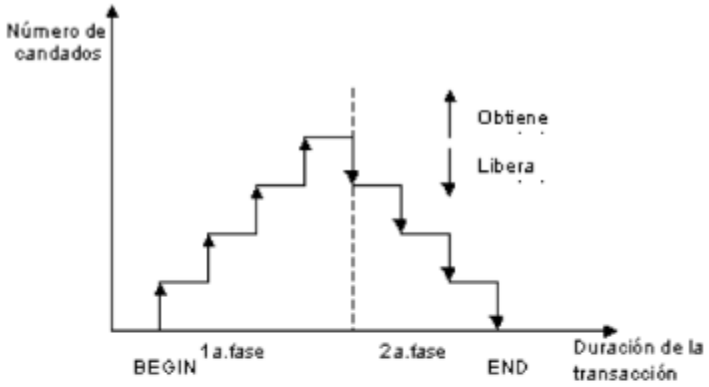


Figura 6.2. Gráfica del uso de los candados de dos fases.

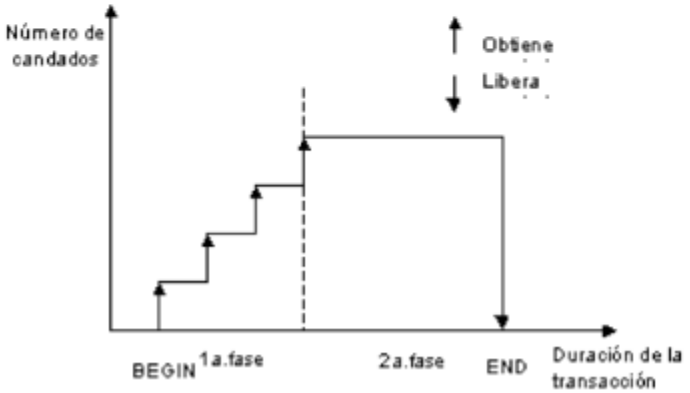
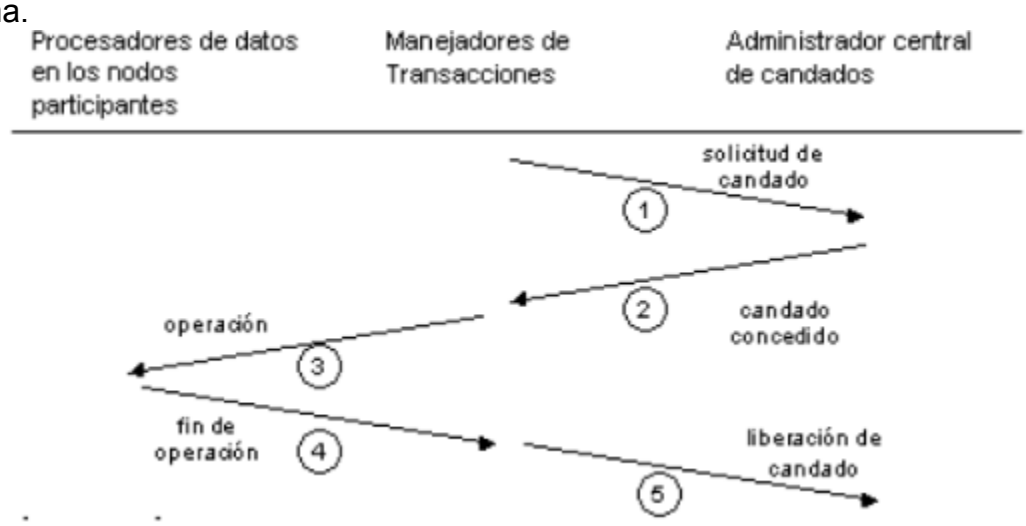


Figura 6.3. Comportamiento de los candados de dos fases estrictos.

22

Candados de dos fases centralizados
En sistemas distribuidos puede que la administración de los candados se dedique a un solo nodo del sistema, por lo tanto, se tiene un

despachador central el cual recibe todas las solicitudes de candados del sistema.



En la Figura 6.4 se presenta la estructura de la comunicación de un administrador centralizado de candados de dos fases. La comunicación

se presenta entre el administrador de transacciones del nodo en donde se origina la transacción (llamado el *coordinador* TM), el administrador de candados en el nodo central y los procesadores de datos (DP) de todos los nodos participantes. Los nodos participantes son todos aquellos en donde la operación se va a llevar a cabo.

Figura 6.4. Comunicación en un administrador centralizado de candados de dos fases estrictos.

La crítica más fuerte a los algoritmos centralizados es el "cuello de botella" que se forma alrededor del nodo central reduciendo los tiempos de respuesta de todo el sistema. Más aún, su disponibilidad es reducida a cero cuando se presentan fallas en el nodo central.

23

Candados de dos fases distribuidos

En los candados de dos fases distribuidos se presentan despachadores en cada nodo del sistema. Cada despachador maneja las solicitudes de candados para los datos en ese nodo. Una transacción puede leer cualquiera de las copias replicada del elemento x, obteniendo un candado de lectura en cualquiera de las copias de x.

La escritura sobre x requiere que se obtengan candados para todas las copias de x. La comunicación entre los nodos que cooperan para ejecutar una transacción de acuerdo al protocolo de candados distribuidos de dos fases se presenta en la Figura 6.5. Los mensajes de solicitud de candados se envían a todos los administradores de candados que participan en el sistema.

Las operaciones son pasadas a los procesadores de datos por los administradores de candados. Los procesadores de datos envía su mensaje de "fin de operación" al administrador de transacciones coordinador.

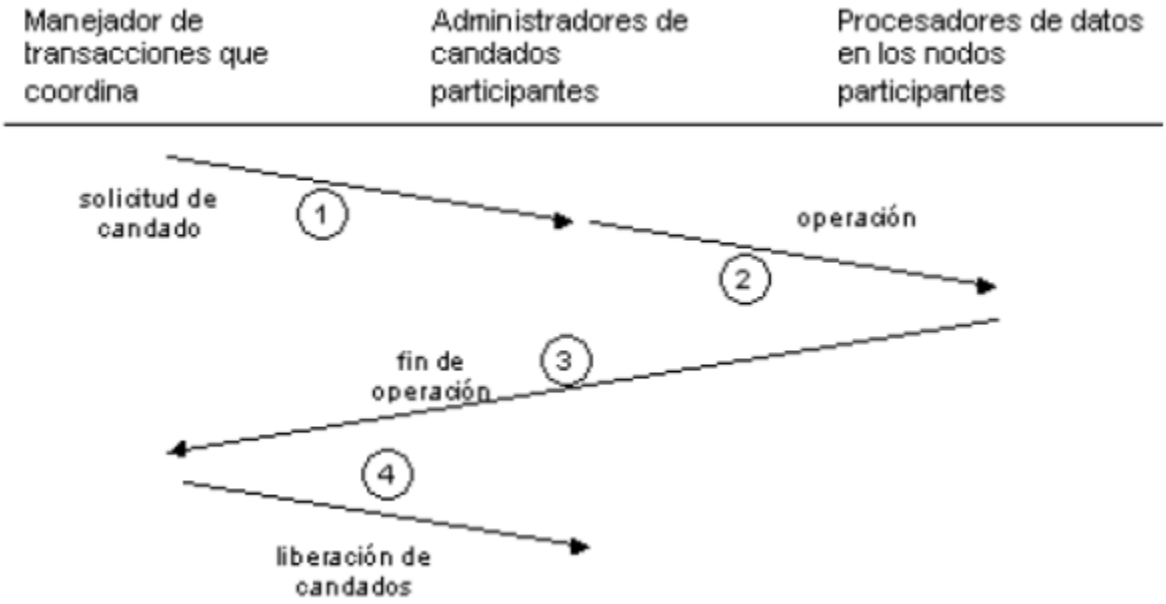


Figura 6.5. Comunicación en candados de dos fases distribuidos.

24

Control de concurrencia optimista

Los algoritmos de control de concurrencia discutidos antes son por naturaleza pesimistas. En otras palabras, ellos asumen que los conflictos entre transacciones son muy frecuentes y no permiten el acceso a un dato si existe una transacción conflictiva que accesa el mismo dato. Así, la ejecución de cualquier operación de una transacción sigue la secuencia de fases: validación (V), lectura (R), cómputo (C) y escritura (W) (ver Figura 6.6a).

Los algoritmos optimistas, por otra parte, retrasan la fase de validación justo antes de la fase de escritura (ver Figura 6.6b). De esta manera, una operación sometida a un despachador optimista nunca es retrasada. Las operaciones de lectura, cómputo y escrita de cada transacción se procesan libremente sin actualizar la base de datos corriente. Cada transacción inicialmente hace sus cambios en copias locales de los datos.

La fase de validación consiste en verificar si esas actualizaciones conservan la consistencia de la base de datos. Si la respuesta es

positiva, los cambios se hacen globales (escritos en la base de datos corriente). De otra manera, la transacción es abortada y tiene que reiniciar

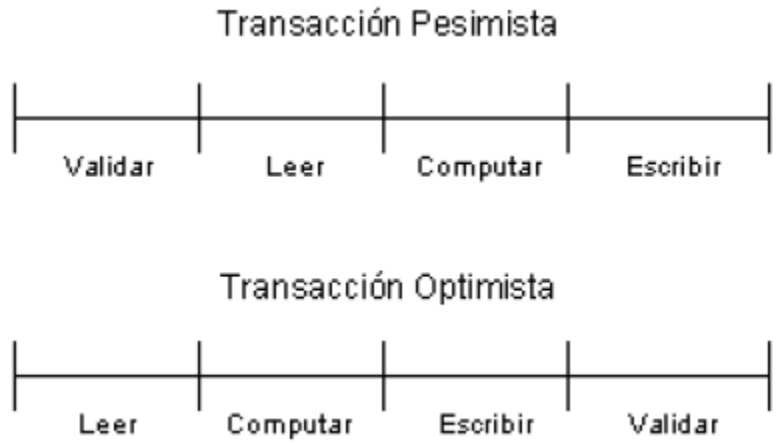


Figura 6.6. Fases de la ejecución de una transacción a) pesimista, b) optimista.

Los mecanismos optimistas para control de concurrencia fueron propuestos originalmente con el uso de estampas de tiempo. Sin embargo, en este tipo de mecanismos las estampas de tiempo se asocian únicamente con las transacciones, no con los datos.

Más aún, las estampas de tiempo no se asignan al inicio de una transacción sino justamente al inicio de su fase de validación. Esto se debe a que las estampas se requieren únicamente durante la fase de validación.

Cada transacción T_i se subdivide en varias subtransacciones, cada una de las cuales se puede ejecutar en nodos diferentes. Sea T_{ij} una subtransacción de T_i que se ejecuta en el nodo j. Supongamos que las transacciones se ejecutan de manera independiente y ellas alcanzan el fin de sus fases de lectura.

A todas las subtransacciones se les asigna una estampa de tiempo al final de su fase de lectura. Durante la fase de validación se realiza una prueba de validación, si una transacción falla, todas las transacciones se rechazan.

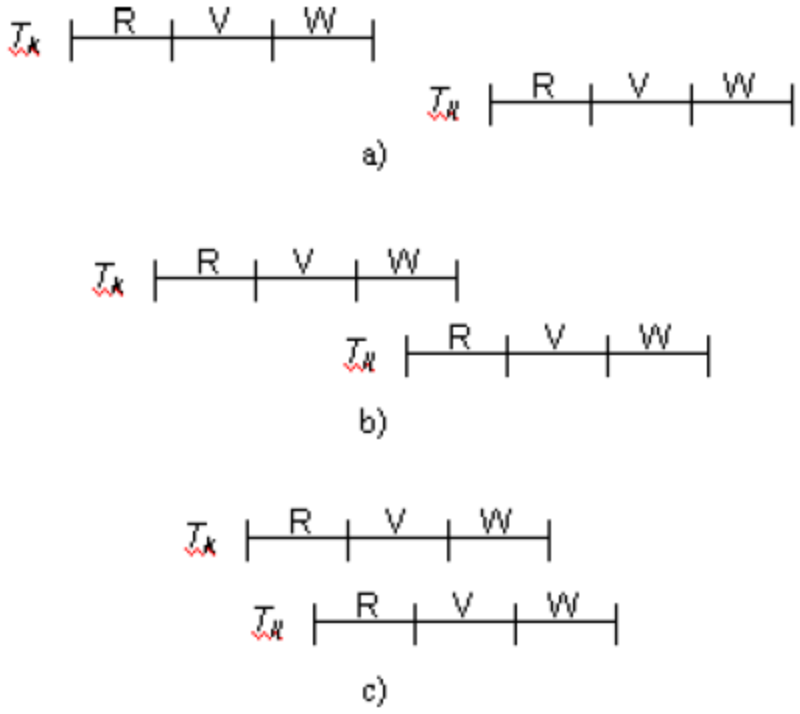


Figura 6.7. Casos diferentes de las pruebas de validación para control de concurrencia optimista.