



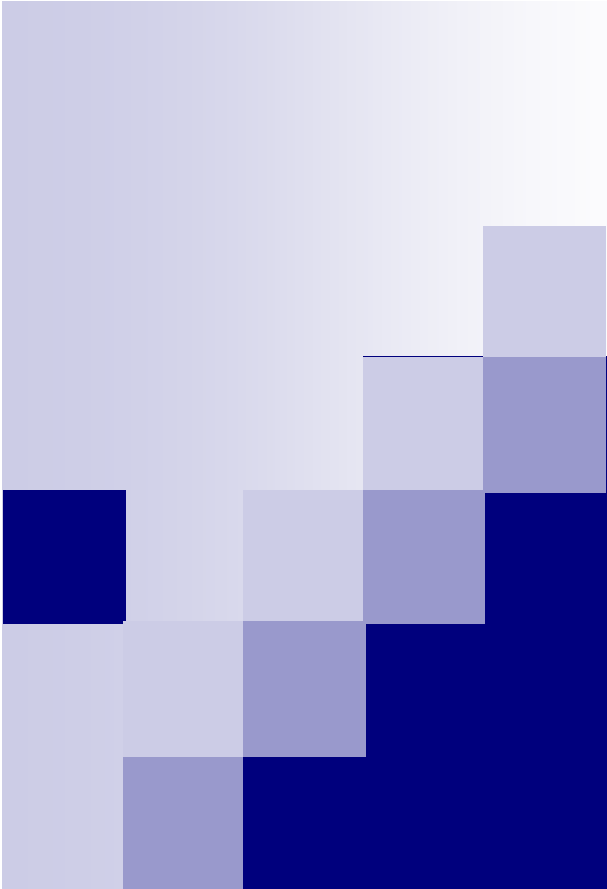
Unidad 3: Seguridad, Transacciones, Concurrency y Recuperación

Bases de Datos Avanzadas, Sesión 11,12,13 :
Transacciones, Concurrency y Recuperación

*Iván González Diego
Dept. Ciencias de la Computación
Universidad de Alcalá*

INDICE

- Transacciones.
- Concurrencia.
- Recuperación.



Tema 5.1: Transacciones

Tema 5.1: Transacciones

- Concepto de Transacción
- Estados de una Transacción
- Ejecución Concurrente
- Secuencialidad
- Recuperación
- Implementación del Aislamiento
- Definición de Transacción en SQL

Concepto de Transacción

- Una transacción es una unidad de ejecución de programa que accede, y posiblemente actualiza, varios ítems de datos.
- Ej.: Transacción para traspasar 50€ de la cuenta A a la cuenta B:
 - 1.read(A)
 - 2.A := A – 50
 - 3.write(A)
 - 4.read(B)
 - 5.B := B + 50
 - 6.write(B)
- Dos problemas principales a considerar:
 - Fallos de varios tipos, tales como fallos de HW y caídas del sistema
 - Ejecución concurrente de múltiple transacciones

Ej.: Traspaso de Efectivo

- Transacción que traspasa 50 € de la cuenta A a la cuenta B:
 1. read(A)
 2. $A := A - 50$
 3. write(A)
 4. read(B)
 5. $B := B + 50$
 6. write(B)
- Requisito de **Atomicidad**
 - Si la transacción falla después del paso 3 y antes del paso 6, el dinero se habrá “perdido” dando lugar a un estado inconsistente de la Base de Datos
 - Los fallos pueden deberse a SW ó HW
 - El sistema debe asegurar que las actualizaciones de una transacción ejecutada parcialmente no se reflejen en la Base de Datos
- Requisito de **Durabilidad**
 - Si se ha notificado al usuario que la transacción se ha completado (i.e., el traspaso de los 50€ se ha realizado), las actualizaciones en la base de datos deben persistir incluso si hay fallos de HW ó SW

Ej: Traspaso de Efectivo (cont.)

- Transacción que traspasa 50 € de la cuenta A a la cuenta B:
 1. read(A)
 2. $A := A - 50$
 3. write(A)
 4. read(B)
 5. $B := B + 50$
 6. write(B)
- Requisito de **Consistencia**:
 - ☐ La suma de A y B no se altera por la ejecución de la transacción
 - ☐ En general, el requisito de consistencia incluye:
 - Restricciones de integridad explícitas, como claves primarias y ajenas
 - Restricciones de integridad implícitas
 - ☐ ej: la suma de los saldos de todas las cuentas menos la suma de las cantidades en préstamos debe ser igual al dinero en efectivo
 - ☐ Una transacción debe ver una base de datos consistente
 - ☐ Durante la ejecución de una transacción la base de datos puede estar temporalmente inconsistente
 - ☐ Si la transacción se completa satisfactoriamente la base de datos debe estar consistente
 - Una lógica de transacción errónea puede dar lugar a inconsistencias

Ej: Traspaso de Efectivo (cont.)

■ Requisito de Aislamiento

- Si entre los pasos 3 y 6, se permite a otra transacción T2 acceder a la base de datos parcialmente actualizada, verá una base de datos inconsistente (la suma $A + B$ será menor de lo que debería ser)

T1
1. read(A)
2. $A := A - 50$
3. write(A)

4. read(B)
5. $B := B + 50$
6. write(B)

T2

read(A), read(B), print(A+B)

- El aislamiento se puede conseguir trivialmente ejecutando transacciones de forma secuencial: una detrás de otra
- Sin embargo, ejecutar varias transacciones de forma concurrente tiene múltiples ventajas

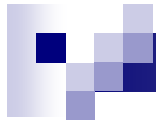
Propiedades ACID

Una transacción es una unidad de ejecución de programa que accede, y posiblemente actualiza, varios ítems de datos. Para mantener la integridad de los datos, un sistema de base de datos debe asegurar:

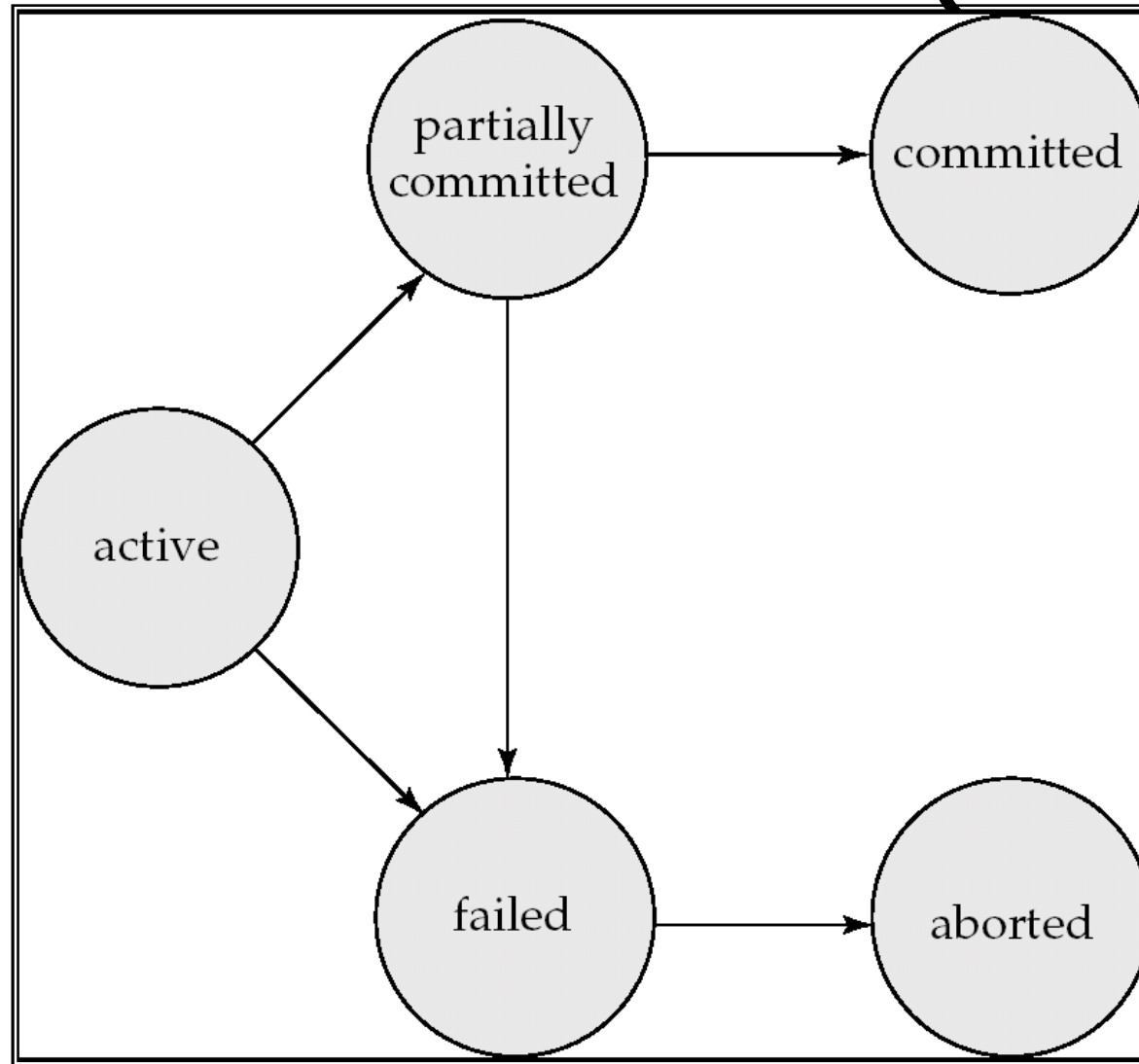
- **Atomicidad.** Bien todas las operaciones de una transacción están reflejadas en la base de datos, o bien ninguna lo está
- **Consistencia.** La ejecución de una transacción en aislamiento debe mantener la consistencia de la base de datos
- **Isolation** (Aislamiento). Aunque se puedan ejecutar varias transacciones concurrentemente, cada transacción debe ser ignorante de las otras transacciones concurrentes
 - Esto es, para cada par de transacciones T_i y T_j , le parece a T_i que, o bien T_j acabó su ejecución antes de que T_i empezara, o T_j empezó la ejecución tras finalizar T_i
- **Durabilidad.** Si una transacción se completa de forma satisfactoria, los cambios que ésta haya hecho en la base de datos persisten, incluso si hay fallos de sistema

Estados de una Transacción

- **Activa** – el estado inicial; la transacción permanece en este estado mientras se ejecuta
- **Parcialmente comprometida** – tras la ejecución de la última instrucción
- **Fallida** – tras descubrirse que la ejecución normal no puede continuar
- **Abortada** – tras deshacerse (Roll-back) la transacción, y la base de datos haya vuelto a su estado anterior al inicio de la transacción. Dos opciones tras ser abortada:
 - ☐ Reiniciar la transacción
 - se puede hacer sólo si no hay un error lógico interno
 - ☐ Matar la transacción
- **Comprometida** – tras completarse de forma satisfactoria



Estados de una Transacción (cont.)



Ejecución Concurrente

- Se permite ejecutar múltiples transacciones de forma concurrentemente en el sistema
- Ventajas:
 - Incremento de la utilización del procesador y del disco, lo que dan lugar a un mayor rendimiento de transacciones (throughput)
 - Ej. Una transacción puede usar la CPU mientras otra está leyendo o escribiendo en el disco
 - Reducción del tiempo medio de respuesta para las transacciones: las transacciones cortas no necesitan esperar detrás de las transacciones largas
- Esquemas de control de Concurrencia – mecanismos para conseguir el aislamiento
 - Controlar la interacción entre transacciones concurrentes para evitar que destruyan la consistencia de la base de datos
 - En 5.2, tras estudiar las nociones de corrección de ejecución concurrente

Planificación (Schedule)

- **Planificación** (schedule) – secuencias de instrucciones que especifican el orden cronológico en el que se ejecutan las instrucciones de una transacción concurrente
 - Una planificación de un conjunto de transacciones debe incluir todas las instrucciones de esas transacciones
 - Debe mantener el orden en el que las instrucciones aparecen en cada transacción individual
- Una transacción que tiene éxito en completar su ejecución, tendrá una instrucción de *COMMIT* como instrucción final
 - Por defecto, las transacciones suponen que ejecutan una instrucción de *COMMIT* en el último paso
- Una transacción que fracasa en completar su ejecución, tendrá una instrucción de *ABORT* como instrucción final

Planificación 1

- Sea T1 traspasar 50€ de A a B, y T2 traspasar el 10% del saldo de A a B
- Una planificación secuencial (*serial*) en la que T1 es seguida por T2:

T_1	T_2
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)

Planificación 2

- Una planificación secuencial (*serial*) en la que T2 es seguida por T1:

T_1	T_2
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)

Planificación 3

- Sean T1 y T2 las transacciones definidas anteriormente
- La siguiente planificación no es una planificación secuencial, pero es equivalente a la Planificación 1

T ₁	T ₂
read(A) $A := A - 50$ write(A)	read(A) $temp := A * 0.1$ $A := A - temp$ write(A)
read(B) $B := B + 50$ write(B)	read(B) $B := B + temp$ write(B)

En las planificaciones 1, 2 y 3,
la suma $A + B$ se mantiene

Planificación 4

- La siguiente planificación concurrente no mantiene el valor de $(A + B)$

T_1	T_2
$\text{read}(A)$ $A := A - 50$	$\text{read}(A)$ $\text{temp} := A * 0.1$ $A := A - \text{temp}$ $\text{write}(A)$ $\text{read}(B)$
$\text{write}(A)$ $\text{read}(B)$ $B := B + 50$ $\text{write}(B)$	$B := B + \text{temp}$ $\text{write}(B)$

Secuencialidad

- Supuesto básico – Cada transacción mantiene la consistencia de la base de datos
- Por lo tanto, la ejecución secuencial de un conjunto de transacciones mantiene la consistencia de la base de datos
- Una planificación (posiblemente concurrente) es secuenciable si es equivalente a una planificación secuencial. Diferentes formas de equivalencia de planificación dan lugar a las nociones de:
 1. Secuencialidad en conflictos
 2. Secuencialidad en vistas
- Visión simplificada de las transacciones
 - Ignoraremos todas las operaciones que no sean instrucciones de read y write
 - Supondremos que las transacciones pueden ejecutar cálculos arbitrarios sobre datos en memoria local entre reads y writes.
 - Nuestra planificación simplificada consiste sólo en instrucciones de read y write

Conflicto de Instrucciones

- Las instrucciones li y lj de las transacciones T_i y T_j respectivamente, están en conflicto *si y sólo si* existe algún ítem Q accedido por ambas li and lj , y al menos una de estas instrucciones escribe Q
 - ☐ $li = \text{read}(Q)$, $lj = \text{read}(Q)$ SIN Conflicto
 - ☐ $li = \text{read}(Q)$, $lj = \text{write}(Q)$ Conflicto
 - ☐ $li = \text{write}(Q)$, $lj = \text{read}(Q)$ Conflicto
 - ☐ $li = \text{write}(Q)$, $lj = \text{write}(Q)$ Conflicto
- Intuitivamente, un conflicto entre li y lj fuerza un orden (lógico) temporal entre ellas
 - ☐ Si li y lj están consecutivas en una planificación y no existe conflicto entre ellas, sus resultados serían los mismos, incluso si se intercambiara su orden en la planificación

Secuencialidad en conflictos

- Si una planificación S puede transformarse en otra planificación S' mediante una serie de intercambios de instrucciones que no tienen conflictos, entonces se dice que S y S' son equivalentes en conflictos
- Se dice que una planificación S es secuenciable en conflictos, si es equivalente en conflictos a una planificación secuencial

Secuencialidad en conflictos (cont.)

- La Planificación 3 se puede transformar en la Planificación 6, una planificación secuencial donde T2 sigue a T1, por una serie de intercambios de instrucciones sin conflictos
- Por lo tanto la Planificación 3 es secuenciable en conflictos

T_1	T_2
read(A) write(A)	
	read(A) write(A)
read(B) write(B)	
	read(B) write(B)

Planificación 3

T_1	T_2
read(A) write(A) read(B) write(B)	
	read(A) write(A) read(B) write(B)

Planificación 6

Secuencialidad en conflictos (cont.)

- Ejemplo de planificación que no es secuenciable en conflictos:

T_3	T_4
read(Q)	write(Q)
write(Q)	

- No es posible intercambiar instrucciones en la planificación anterior que generen bien la planificación secuencial $\langle T_3, T_4 \rangle$, o bien la planificación secuencial $\langle T_4, T_3 \rangle$.

Secuencialidad en Vistas

- Sean S y S' dos planificaciones con el mismo conjunto de transacciones. S y S' son equivalentes en vistas si las siguientes tres condiciones se cumplen, para cada ítem de datos Q ,
 1. Si en la planificación S la transacción T_i lee el valor inicial de Q , entonces en la planificación S' también la transacción T_i debe leer el valor inicial de Q
 2. Si en la planificación S , la transacción T_i ejecuta $\text{read}(Q)$, y ese valor estaba producido por la transacción T_j (si existe), entonces en la planificación S' también la transacción T_i debe leer el valor de Q que produjo la misma operación $\text{write}(Q)$ de la transacción T_j
 3. La transacción (si existe) que realiza la operación final $\text{write}(Q)$ en la planificación S , debe también realizar la última operación $\text{write}(Q)$ en la planificación S'
- Como se puede ver, la equivalencia de vistas está también basada en reads y writes únicamente

Secuencialidad en Vistas (cont.)

- Una planificación S es secuenciable en vistas si es equivalente en vistas a una planificación secuencial
- Cada planificación secuenciable en conflictos es también secuenciable en vistas
 - Ej.: Una planificación que es secuenciable en vistas pero no secuenciable en conflictos

T_3	T_4	T_6
read(Q)	write(Q)	
write(Q)		
		write(Q)

- ¿A qué planificación secuencial es equivalente?
- Toda planificación secuenciable en vistas que no sea secuenciable en conflictos tiene escrituras a ciegas (blind writes)

Otras Nociones de Secuencialidad

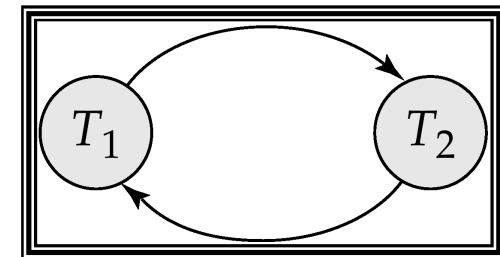
- La planificación siguiente produce el mismo resultado que la planificación secuencial $\langle T_1, T_5 \rangle$, a pesar de no ser equivalente en conflictos ni equivalente en vistas a aquella

T_1	T_5
read(A) $A := A - 50$ write(A)	read(B) $B := B - 10$ write(B)
read(B) $B := B + 50$ write(B)	read(A) $A := A + 10$ write(A)

- Determinar tal equivalencia requiere un análisis de las operaciones además de las de read y write.

Test de Secuencialidad en Conflictos

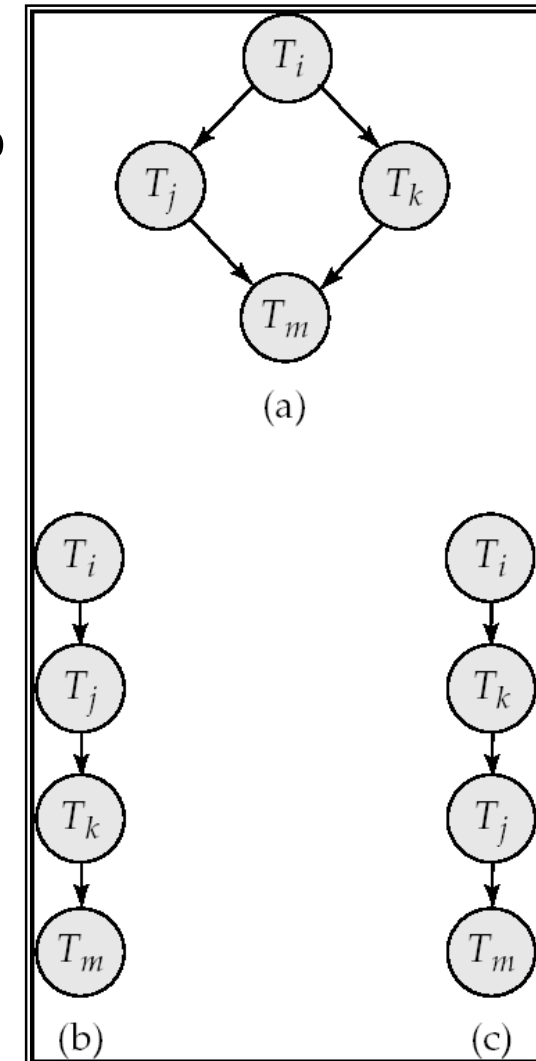
- Planificaciones que se generan son secuenciables.
- Grafo de Precedencia: $G(V,A)$.
 - V es el número de vértices \Rightarrow todas transacciones
 - A un conjunto de arcos: $T_i \rightarrow T_j$, una de las tres condiciones:
 - T_i ejecuta $\text{write}(Q)$ antes de que T_j ejecute $\text{read}(Q)$
 - T_i ejecuta $\text{read}(Q)$ antes de que T_j ejecute $\text{write}(Q)$
 - T_i ejecuta $\text{write}(Q)$ antes de que T_j ejecute $\text{write}(Q)$



Test de Secuencialidad en Conflictos

- Una planificación es secuenciable en conflictos si y sólo si su grafo de precedencias es acíclico
- Existen algoritmos de detección de ciclos con complejidad de orden n^2 en el tiempo, donde n es el número de vértices en el grafo
 - Los mejores algoritmos son de orden $n + e$, donde e es el número de arcos
- Si el grafo de precedencia es acíclico, el orden de secuencialidad se puede obtener por una *ordenación topológica* del grafo
 - Es un orden lineal consistente con el orden parcial del grafo
 - Ej.: un orden de secuencialidad para la planificación a) sería

$$T_i \rightarrow T_j \rightarrow T_k \rightarrow T_m$$
 - ¿Hay otros?



Test de Secuencialidad en Vistas

- El grafo de precedencia para secuencialidad en conflictos no se puede usar directamente como test de secuencialidad en vistas
 - Extensiones para comprobar la secuencialidad en vistas tiene un coste exponencial con el tamaño del grafo de precedencia
- El problema de comprobar si una planificación es secuenciable en vistas pertenece a la clase de problemas NP-completo
 - Por ello, la existencia de un algoritmo eficiente es muy poco probable
 - Sin embargo, se pueden usar algoritmos prácticos que sólo comprueben algunas condiciones suficientes para la secuencialidad en vistas

Planificación Recuperable

Es necesario contemplar el efecto de los fallos de las transacciones en transacciones ejecutándose concurrentemente

- **Planificación Recuperable** — si una transacción T_j lee un ítem de datos previamente escrito por una transacción T_i , entonces la operación de COMMIT de T_i aparece antes de la operación de COMMIT de T_j
- La siguiente planificación (planificación 11) no es recuperable si T_9 ejecuta COMMIT justo después de $\text{read}(A)$

T_8	T_9
$\text{read}(A)$ $\text{write}(A)$ $\text{read}(B)$	$\text{read}(A)$

- Si T_8 abortara, T_9 podría haber leído (y posiblemente mostrado al usuario) un estado inconsistente de la base de datos. La base de datos debe asegurar que las planificaciones son recuperables

Rollback en cascada

- Rollback en cascada – un fallo en una sola transacción da lugar a una serie de vuelta-atrás (rollbacks) de transacción
 - Considerar la siguiente planificación donde ninguna de las transacciones se ha comprometido (por lo que la planificación es recuperable)

T_{10}	T_{11}	T_{12}
read(A) read(B) write(A)	read(A) write(A)	read(A)

Si T_{10} falla, T_{11} y T_{12} deben también volver atrás

- Puede dar lugar a deshacer una gran cantidad de trabajo

Planificación sin Cascada

- **Planificación sin cascada** — No pueden ocurrir Rollbacks en cascada; para cada par de transacciones T_i y T_j tal que T_j lee un ítem de datos previamente escrito por T_i , la operación de COMMIT de T_i aparece antes de la operación de lectura de T_j
- Cada planificación sin cascada es también recuperable
- Es deseable restringir las planificaciones a aquéllas que son sin cascada

Control de Concurrencia

Implementación de Aislamiento

- Una base de datos debe proporcionar un mecanismo que asegure que todas las posibles planificaciones son
 - ☐ Secuenciables o en conflicto o en vistas, y
 - ☐ Recuperables y preferiblemente sin cascada
- Una política en la que solo se pueda ejecutar una transacción a la vez genera planificaciones secuenciales, pero proporciona un nivel muy pobre de concurrencia
 - ☐ ¿Son las planificaciones secuenciales recuperables/sin cascada?
- Comprobar la secuencialidad de una planificación después de que se haya ejecutado ¡es muy tarde!
- Objetivo – desarrollar protocolos de control de concurrencia que aseguren la secuencialidad

Definición de Transacción en SQL

- El lenguaje de manipulación de datos (DML) debe incluir sentencias para especificar el conjunto de acciones que comprende una transacción
- En SQL, una transacción empieza implícitamente
- Una transacción en SQL acaba con:
 - COMMIT [WORK] compromete la transacción actual y comienza una nueva
 - ROLLBACK [WORK] provoca que la transacción actual aborte
- SQL no especifica que ocurre si se omiten las dos
- En casi cualquier sistema de base de datos, por defecto, cada instrucción de SQL también se compromete de forma implícita si se ejecuta satisfactoriamente
 - El COMMIT implícito se puede desactivar mediante una directiva de la base de datos
 - Ej.: En JDBC, `connection.setAutoCommit(false);`



Tema 5.2:

Control de la Concurrencia

Control de la Concurrency

- Propiedad de aislamiento en las transacciones.
- Controlar la interacción de las transacciones ⇒
Esquemas de control de la concurrency
- Basados en secuencialidad.
- Tipos:
 - ☐ Basados en bloqueos.
 - ☐ Basados en marcas temporales.
 - ☐ Basados en validación.

Protocolos basados en Bloqueo

- Bloqueo \Rightarrow mecanismo de control concurrente para el acceso a un elemento de datos.
- Dos modos de bloqueo:
 - Compartido (C o S) : transacción T_i bloqueo de sobre Q en modo C $\Rightarrow T_i$ puede leer Q pero no escribir.
 - Exclusivo (X) : transacción T_i bloqueo de sobre Q en modo X $\Rightarrow T_i$ puede leer y escribir Q.
- Gestor control de la concurrencia. La transacción procede después de la concesión del bloqueo.

Protocolos basados en Bloqueo

- Función de Compatibilidad \Rightarrow Matriz de Compatibilidad

	C	X
C	cierto	falso
X	falso	falso

- Un bloqueo se concede sobre un elemento si es compatible con los bloqueos actuales del elemento concedidos a otras transacciones.
- Cualquier número de transacciones pueden mantener bloqueos compartidos.
- Si cualquier transacción mantiene un bloqueo exclusivo, ninguna transacción puede mantener ningún otro bloqueo.
- Si una transacción no puede acceder a un bloqueo \Rightarrow la transacción espera hasta que todos los bloqueos incompatibles desaparezcan.

Protocolos basados en Bloqueo

Bloquear- $X(B)$

leer(B);

$B := B - 50$;

escribir(B);

desbloquear(B);

bloquear- $X(A)$;

leer(A);

$A := A + 50$;

escribir(A);

desbloquear

Transacción T_1 .

T_2 : bloquear- $C(A)$;

leer(A);

desbloquear(A);

bloquear- $C(B)$;

leer(B);

desbloquear(B);

visualizar($A + B$).

Transacción T_2 .



Protocolos basados en Bloqueo

T_1	T_2	Gestor de control de concurrencia
bloquear-X(B)		conceder-X(B, T_1)
leer(B)		
$B := B - 50$		
escribir(B)		
desbloquear(B)		
	bloquear-C(A)	conceder-C(A, T_2)
	leer(A)	
	desbloquear(A)	
	bloquear-C(B)	conceder-C(B, T_2)
	leer(B)	
	desbloquear(B)	
	visualizar($A + B$)	
bloquear-X(A)		conceder-X(A, T_1)
leer(A)		
$A := A + 50$		
escribir(A)		
desbloquear(A)		

Planificación 1.

Protocolos basados en Bloqueo

■ Considerar

T_3 : bloquear-X(B);
 leer(B);
 $B := B - 50$;
 escribir(B);
 bloquear-X(A);
 leer(A);
 $A := A + 50$;
 escribir(A);
 desbloquear(B);
 desbloquear(A).

Transacción T_3 .

T_4 : bloquear-C(A);
 leer(A);
 bloquear-C(B);
 leer(B);
 visualizar($A + B$).
 desbloquear(A);
 desbloquear(B).

Transacción T_4 .

T_3	T_4
bloquear-X(B)	
leer(B)	
$B := B - 50$	
escribir(B)	
	bloquear-C(A)
	leer(A)
	bloquear-C(B)
bloquear-X(A)	

Planificación 2.

- Interbloqueo (deadlock). Sistema debe de retroceder una de las dos transacciones.
- Si no se usan bloqueos \Rightarrow estados inconsistentes.
- Protocolo de Bloqueo \Rightarrow conjunto de reglas seguido por todas las transacciones mientras de solicitan y sueltan bloqueos. Restringen el número de planificaciones posibles. (legales)

Protocolos basados en Bloqueo

- T_i precede a T_j : $T_i \rightarrow T_j$,
 - Existe un elemento de datos Q , T_i ha obtenido bloqueo en modo A
 - T_j ha obtenido bloqueo en modo B
 - $\text{Comp}(A,b)=\text{falso}$.
- Si $T_i \rightarrow T_j$, cualquier planificación secuencial equivalente, T_i debe aparecer antes que T_j .
- Inanición \Rightarrow Transacción nunca progresa debido a bloqueos sucesivos de otras transacciones
- Se puede evitar:
 - No exista una transacción que posea un bloqueo sobre Q que esté en conflicto con el modo M
 - No haya otra transacción que esté esperando un bloqueo sobre Q y que lo haya solicitado antes.

Protocolos de Bloqueo de dos fases

- Protocolo que asegura la secuencialidad
- Dos fases:
 - Fase de crecimiento \Rightarrow Puede obtener bloqueos. No liberarlos
 - Fase de decrecimiento \Rightarrow Puede liberar bloqueos. No obtenerlos
- Transacciones T3 y T4.
- Transacciones T1 y T2 NO.
- Asegura la secuencialidad en cuanto a conflictos.
- Punto de bloqueo \Rightarrow punto donde la transacción adquiere el último bloqueo. \Rightarrow Ordenar transacciones por punto bloqueo.
- No asegura la ausencia de interbloqueos (Planificación 2)
- Puede ocurrir retroceso en cascada.



Protocolos de Bloqueo de dos fases

- Ejemplo:

T_5	T_8	T_7
bloquear-X(A) leer(A) bloquear-C(B) leer(B) escribir(A) desbloquear(A)	bloquear-X(A) leer(A) escribir(A) desbloquear(A)	bloquear-C(A) leer(A)

Protocolos de Bloqueo de dos fases

- Protocolo de bloqueo estricto de dos fases \Rightarrow evita retrocesos en cascada.
 - Dos fases + poseer todos bloqueos exclusivos hasta que termina.
- Protocolo de bloqueo riguroso de dos fases \Rightarrow poseer todos los bloqueos hasta comprometer la transacción.

T_8 : leer(a_1);
leer(a_2);
...
leer(a_n);
escribir(a_1).

T_9 : leer(a_1);
leer(a_2);
visualizar($a_1 + a_2$).

Protocolos de Bloqueo de dos fases

- Conversiones de bloqueo (refinado) \Rightarrow aumentar concurrencia
 - Fase Crecimiento:
 - Adquirir bloqueo C o Bloqueo X
 - Modo compartido \Rightarrow modo exclusivo (subir)
 - Fase Decrecimiento
 - Soltar bloqueo C o Bloqueo X
 - Modo exclusivo \Rightarrow modo compartido (bajar) (fase de decrecimiento)

T_8	T_9
bloquear-C(a_1)	bloquear-C(a_1)
bloquear-C(a_2)	bloquear-C(a_2)
bloquear-C(a_3)	
bloquear-C(a_4)	
	desbloquear(a_1)
	desbloquear(a_2)
bloquear-C(a_n)	
subir(a_1)	

Adquisición Automática de Bloqueos

- Generar automáticamente las instrucciones de bloqueo y desbloqueo para una transacción basándose en peticiones de lectura y escritura.

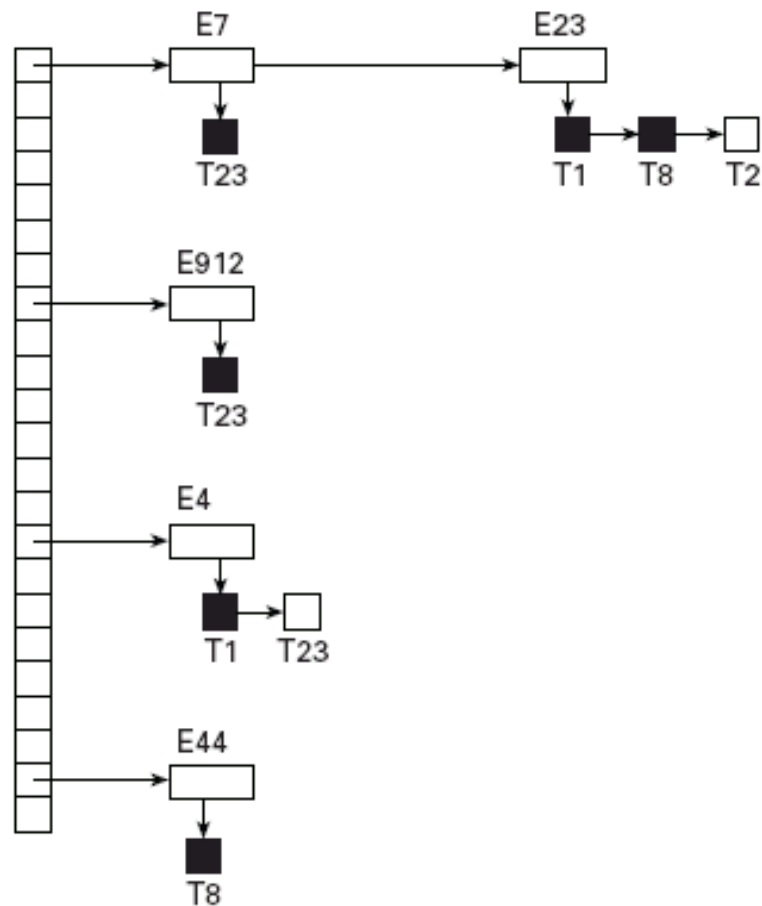
```
If  $T_i$  tiene un bloqueo en  $D$ 
  then
    leer( $D$ )
  else
    begin
      if necesario esperar hasta ninguna
        transacción tenga un bloqueo-X sobre  $D$ 
      Permitir  $T_i$  un bloqueo-C sobre  $D$ ;
      leer( $D$ )
    end
```

Adquisición Automática de Bloqueos

```
if  $T_i$  tiene un bloqueo-X sobre  $D$ 
  then
    escribir( $D$ )
  else
    begin
      if necesario esperar hasta ninguna
        transacción tenga cualquier bloqueo sobre  $D$ ,
      if  $T_i$  tiene un bloqueo-C sobre  $D$ 
        then
          subir bloqueo sobre  $D$  a bloqueo-X
        else
          permitir  $T_i$  un bloqueo-X sobre  $D$ 
      escribir( $D$ )
    end;
```

Implementación de Bloqueos

■ Gestor de Bloqueos



Tratamiento de Bloqueos

- Sistema está bloqueado si hay un conjunto de transacciones tal que cada transacción en el conjunto está esperando a otra transacción del conjunto.
- Protocolos de Prevención de Interbloqueos
- Estrategias:
 - Cada transacción bloquee todos sus elementos de datos antes de comenzar la ejecución (predeclaración)
 - Imponer un orden parcial a todos los elementos de datos y requerir que una transacción puede bloquear elementos en el orden especificado por el orden parcial. (protocolos basados en grafos)
 - Uso de expropiaciones y retrocesos de transacciones. Se utilizan marcas temporales a la transacción que se le ha expropiado y se sigue utilizando bloqueos para el control de la concurrencia.

Tratamiento de Bloqueos

- Esquema Esperar-Morir \Rightarrow Sin expropiación.
 - Transacciones más viejas pueden esperar a que las más jóvenes suelten los elementos. Las más jóvenes nunca esperan a las más viejas, estas se deshacen.
 - Una transacción puede morir varias veces antes de obtener los datos solicitados.
- Esquema Herir-Esperar \Rightarrow Con expropiación.
 - Transacciones más viejas hieren (fuerzan ROLL-BACK) de las transacciones más jóvenes en vez de esperar. Las más jóvenes pueden esperar a las más viejas
 - Puede tener menos roll-back que el esquema anterior.
- En ambos esquemas, las transacciones de reinician con su marca temporal original.
- Las más viejas tienen precedencia sobre las más nuevas \Rightarrow se evita la inanición.

Tratamiento de Bloqueos

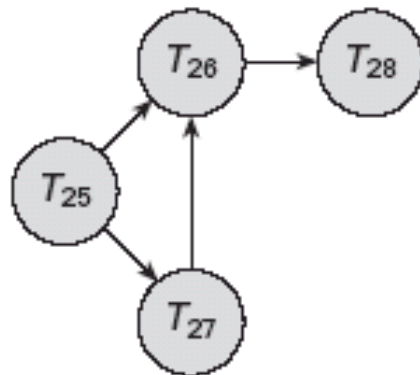
- Esquemas basados en límites de tiempo
 - Una transacción espera a un bloqueo una cantidad de tiempo. Después de ese tiempo se deshace.
 - No se dan interbloqueos
 - Simple de implementar.
 - Es posible la inanición.
 - Difícil de determinar el intervalo de tiempo.

Detección de Bloqueos

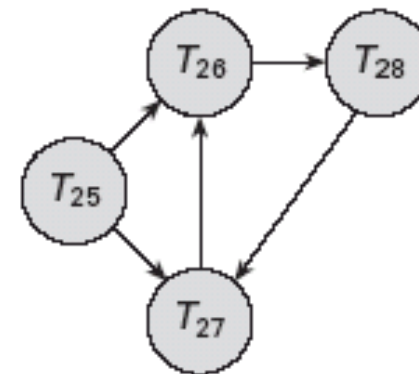
- Interbloqueos se pueden describir como grafos de espera.
- Consisten de un par $G = (V, E)$,
 - V es un conjunto de vértices (todas las transacciones)
 - E es un conjunto de arcos; par ordenado $T_i \rightarrow T_j$.
- Si $T_i \rightarrow T_j$ está en $E \Rightarrow$ hay un arco de T_i a T_j , donde T_i espera a que T_j suelte el elemento de datos .
- Cuando T_i pide un elemento de datos que tiene $T_j \Rightarrow$ un arco entre T_i T_j se inserta en el grafo. Se elimina cuando T_j suelta el elemento necesitado por T_i .
- El sistema está en interbloqueo si y solo si el grafo tiene un ciclo.
- Se debe de utilizar un algoritmo periódicamente para detectar los ciclos.

Detección de Bloqueos

$$T_{26} \rightarrow T_{28} \rightarrow T_{27} \rightarrow T_{26}$$



Grafo de espera sin ciclos.



Grafo de espera con un ciclo.

Recuperación de Bloqueos

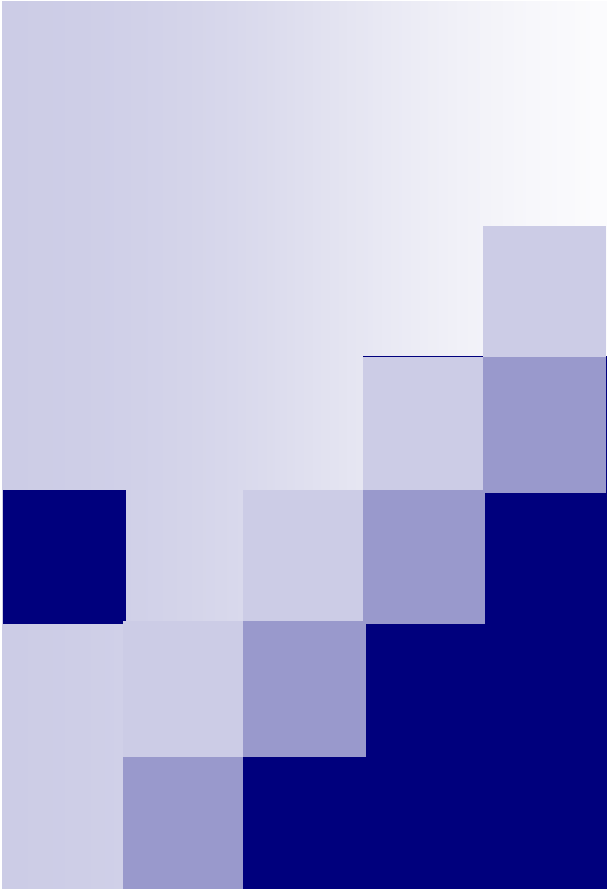
- Cuando se detecta un interbloqueo:
 - alguna transacción (víctima) será deshecha para romper interbloqueo. Seleccionar la transacción víctima que involucre menor coste.
 - Rollback → determinar cuanto hay que deshacer la transacción.
 - Total: Abortar y reiniciar.
 - Deshacer lo necesario para romper el interbloqueo.
 - Inanición sucede si siempre se elige la misma transacción como víctima. Incluir el número de rollbacks en el factor de coste.

Operaciones de Inserción y Borrado

- Si se usa un algoritmo de bloqueo dos fases:
 - Un borrado sólo se puede realizar si la transacción que borra la tupla tiene un bloqueo-X sobre la tupla a ser borrada.
 - Una transacción que inserta una nueva tupla se le da un bloqueo-X sobre la tupla.
- Inserciones y borrados pueden producir el fenómeno fantasma
 - Una transacción que lee una relación y una transacción que inserta una tupla en la relación pueden causar conflictos a pesar de no acceder a la tupla en común.
 - Si se utilizan sólo bloqueos de tuplas, pueden resultar planificaciones no secuenciables: la relación que lee puede no ver la nueva tupla, pudiendo ser secuenciable antes de la transacción que inserta.

Operaciones de Inserción y Borrado

- T1 lee la relación y T2 inserta información \Rightarrow información debería bloquearse.
- Solución:
 - Asociar elementos de datos con la relación.
 - T que lee, adquirir un bloqueo-C.
 - T que escribe, adquirir bloqueo-X.
- Producen una concurrencia baja para insertar/borrar
- Protocolos de bloqueo de índices proporcionan mayor concurrencia, previniendo efecto fantasma, proporcionando bloqueos sobre ciertos nodos del índice.



Tema 5.3: Recuperación

Tema 5.3: Recuperación

- Clasificación de Fallos
- Estructura del Almacenamiento
- Recuperación y Atomicidad
- Recuperación basada en Log

Clasificación de Fallos

- Fallo de Transacción:
 - Errores Lógicos: La transacción no se puede completar debido a alguna condición de error interna
 - Errores de Sistema: El sistema de base de datos debe terminar una transacción activa debido a una condición de error (ej: deadlock)
- Caída del Sistema: Un fallo de alimentación u otro fallo SW o HW que cause que el sistema se caiga
 - Supuesto de Fail-stop: El contenido del almacenamiento no-volátil se supone que no se corrompe por una caída del sistema
 - Los sistemas de base de datos tienen numerosas comprobaciones de integridad para prevenir corrupciones de datos en los discos
- Fallo de Disco: Una caída de una cabeza o un fallo similar de disco destruye todo o parte del almacenamiento de disco
 - La destrucción se supone que es detectable: los drivers de disco disponen de checksums para detectar fallos

Algoritmos de Recuperación

- Los Algoritmos de Recuperación son técnicas para asegurar:
 - La consistencia de una base de datos, y
 - La atomicidad y durabilidad de una transacción a pesar de que existan fallos
- Los Algoritmos de Recuperación tienen dos partes
 1. Acciones tomadas **durante** el procesamiento normal de una transacción, para asegurar que existe suficiente información para recuperarse de fallos
 2. Acciones tomadas **tras un fallo**, para recuperar el contenido de una base de datos hasta un estado que asegure la atomicidad, la consistencia y la durabilidad

Estructura del Almacenamiento

- Almacenamiento volátil:
 - NO sobrevive a caídas del sistema
 - Ejemplos: memoria principal, memoria caché
- Almacenamiento no-volátil:
 - Sobrevive a caídas del sistema
 - Ejemplos: disco, cinta, memoria flash,
RAM no-volátil (alimentada por batería)
- Almacenamiento estable:
 - Una forma mítica de almacenamiento que sobrevive a todos los fallos
 - Aproximación: mantener múltiples copias en diferentes medios no-volátiles

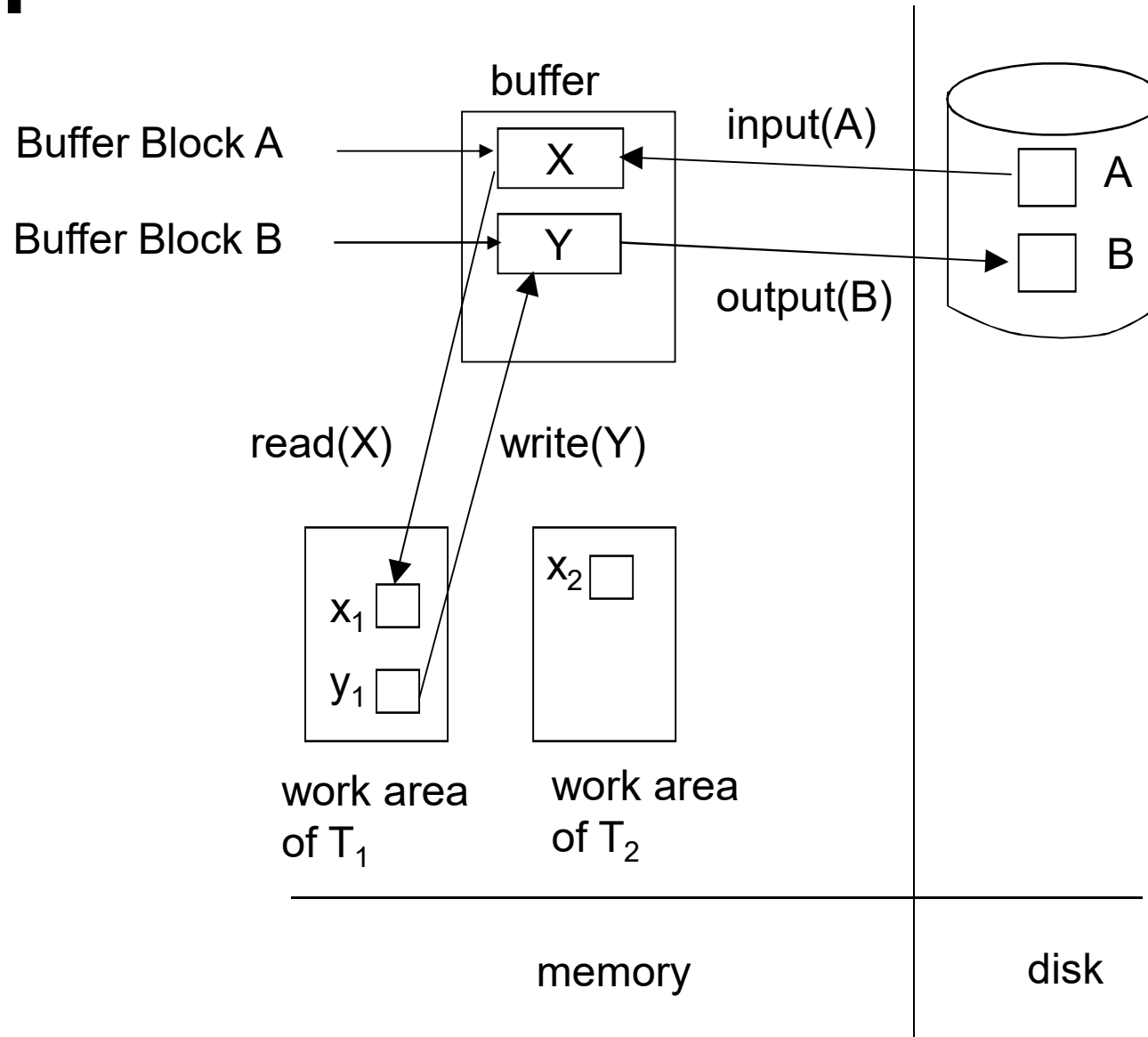
Acceso a Datos

- **Bloques Físicos** son aquellos bloques que residen en disco
- **Bloques de Buffer** son aquellos bloques que residen temporalmente en memoria principal
- Los movimientos de bloques entre disco y memoria principal se inician mediante las siguientes dos operaciones:
 - **input**(B) transfiere el bloque físico B a memoria principal
 - **output**(B) transfiere el bloque de buffer B a disco, y sustituye el bloque físico correspondiente allí
- Cada transacción T_i tiene su área de trabajo privada en la que se guardan copias locales de todos los ítems de datos y a los que se acceden y se actualizan por ella
 - La copia local de T_i de un ítem de datos X se llama x_i
- Supondremos, por simplicidad, que cada ítem de datos se almacena y cabe en un solo bloque

Acceso a Datos (cont.)

- Una transacción transfiere ítems de datos entre bloques de buffer de sistema y su area de trabajo privada usando las siguientes operaciones:
 - **read**(X) asigna el valor del ítem de datos X a la variable local x_i
 - **write**(X) asigna el valor de la variable local x_i al ítem de datos $\{X\}$ en el bloque de buffer
 - Ambas operaciones pueden necesitar ejecutar la instrucción **input**(B_X) antes de la asignación, si el bloque B_X en el que reside X no está ya en memoria
- Las transacciones
 - Ejecutan **read**(X) cuando acceden a X por primera vez
 - Todos los accesos posteriores se hacen sobre la copia local
 - Tras el último acceso, la transacción ejecuta **write**(X)
- **output**(B_X) no necesita realizarse a continuación de **write**(X). El sistema puede realizar la operación de **output** cuando le venga bien

Ejemplo de Acceso a Datos



Recuperación y Atomicidad

- Modificar la base de datos sin asegurar que la transacción se comprometerá puede dejar la base de datos en un estado inconsistente
- Consideremos la transacción T_i que traspasa 50 € de la cuenta A a la cuenta B; el objetivo es o bien realizar todas las modificaciones en la base de datos efectuadas por T_i o bien ninguna
- T_i puede requerir varias operaciones de output (para volcar A y B). Un fallo puede ocurrir tras efectuar una de estas modificaciones pero antes de que todas ellas se hayan realizado

Recuperación y Atomicidad (cont.)

- Para asegurar la atomicidad a pesar de fallos, primero se escribirá información describiendo las modificaciones en almacenamiento estable sin modificar la base de datos misma
- Estudiaremos dos aproximaciones:
 - **Recuperación basada en Log, y**
 - **Recuperación basada en Shadow-paging**
- Supondremos (inicialmente) que las transacciones se ejecutan secuencialmente, esto es, una tras de otra

Recuperación basada en Log

- Un **log** se mantiene en almacenamiento estable
 - El **log** es una secuencia de **registros de log**, y mantiene un histórico de las actividades de actualización sobre la base de datos
- Cuando la transacción T_i comienza, ella misma se registra escribiendo un registro $\langle T_i, \text{start} \rangle$ en el log
- *Antes* que T_i ejecute **write**(X), un registro $\langle T_i, X, V_1, V_2 \rangle$ se escribe en el log, donde V_1 es el valor de X antes del write, y V_2 es el valor a escribir en X
 - El registro del log anota que: T_i ha realizado write en el item de datos X , X tenía el valor V_1 antes del write, y tendrá el valor V_2 tras el write
- Cuando T_i termine su última instrucción, se escribirá un registro $\langle T_i, \text{commit} \rangle$ en el log
- Supondremos que los registros de log se escriben directamente en almacenamiento estable (esto es, no existe buffer intermedios)
- Dos aproximaciones
 - Modificación Diferida de BB.DD.
 - Modificación Inmediata de BB.DD.

Modificación Diferida de BB.DD.

- El esquema de modificación diferida de **base de datos** registra todas las modificaciones en el log, pero pospone todos los **writes** hasta tras realizar un commit parcial
- Supone que las transacciones se ejecutan secuencialmente
- Las transacciones empiezan escribiendo un registro $\langle T_i, \text{start} \rangle$ en el log
- Una operación **write**(X), hace que se escriba un registro $\langle T_i, X, V \rangle$ en el log, donde V es el nuevo valor de X
 - En este esquema no es necesario el valor antiguo
- No se ejecuta la escritura de X en este momento, sino que se pospone
- Cuando T_i se consolida parcialmente, se escribe $\langle T_i, \text{commit} \rangle$ en el log
- Finalmente, los registros de log se leen y se usan para ejecutar realmente los writes diferidos previamente

Modificación Diferida de BB.DD. (cont.)

- Durante la recuperación tras una caída, se necesita rehacer (redo) una transacción si y solo si ambos $\langle T_i \text{ start} \rangle$ y $\langle T_i \text{ commit} \rangle$ se encuentran en el log
- Rehacer una transacción T_i (**redo** T_i) pone los valores de los ítems de datos actualizados por la transacción, en los nuevos valores
- Las caídas pueden ocurrir mientras
 - la transacción esta ejecutando las actualizaciones originales, o
 - Transacciones ejemplo T_0 y T_1 (T_0 se ejecuta antes que T_1):
 - Mientras se esta llevando a cabo la recuperación

T_0 : **read** (A)
 $A := A - 50$
 write (A)
 read (B)
 $B := B + 50$
 write (B)

T_1 : **read** (C)
 $C := C - 100$
 write (C)

Modificación Diferida de BB.DD. (cont.)

- Estado del log según aparece en tres instantes de tiempo

$\langle T_0 \text{ start} \rangle$	$\langle T_0 \text{ start} \rangle$	$\langle T_0 \text{ start} \rangle$
$\langle T_0, A, 950 \rangle$	$\langle T_0, A, 950 \rangle$	$\langle T_0, A, 950 \rangle$
$\langle T_0, B, 2050 \rangle$	$\langle T_0, B, 2050 \rangle$	$\langle T_0, B, 2050 \rangle$
	$\langle T_0 \text{ commit} \rangle$	$\langle T_0 \text{ commit} \rangle$
	$\langle T_1 \text{ start} \rangle$	$\langle T_1 \text{ start} \rangle$
	$\langle T_1, C, 600 \rangle$	$\langle T_1, C, 600 \rangle$
		$\langle T_1 \text{ commit} \rangle$
(a)	(b)	(c)

- Si el log en almacenamiento estable en el momento de la caída fuera, según el caso,...
 - (a) No se necesita realizar acciones de **redo**
 - (b) Se debe realizar **redo**(T_0) puesto que $\langle T_0 \text{ commit} \rangle$ está presente
 - (c) Se debe realizar **redo**(T_0) seguido de **redo**(T_1) puesto que $\langle T_0 \text{ commit} \rangle$ y $\langle T_1 \text{ commit} \rangle$ están presentes

Modificación Inmediata de BB.DD. (cont.)

- El esquema de **modificación inmediata** permite a una base de datos que las modificaciones de una transacción no consolidada se realicen según se ejecuten las operaciones de **write**
 - Puesto que se pueden necesitar la operación de **undo**, los registros del log deben incluir tanto el valor viejo como el nuevo
- Los registros del log se deben escribir **antes** de que el ítem se escriba en la base de datos
 - Supondremos que el registro de log se escribe directamente en almacenamiento estable
 - Extension: posponer la escritura del registro de log, siempre que antes de ejecutar una operación **output(B)** para un bloque de datos B, todos los registros de log correspondientes al ítem B se hayan volcado físicamente (**flush**) a almacenamiento estable
- El volcado de bloques actualizados puede ocurrir en cualquier momento antes o después de consolidar (**commit**) la transacción
- El orden en el que los bloques se vuelcan (**output**) a disco puede ser diferente del orden en el que se escriben (**write**)

Modificación Inmediata de BB.DD. Ejemplo

Log	Write	Output
-----	-------	--------

$\langle T_0 \text{ start} \rangle$		
-------------------------------------	--	--

$\langle T_0, A, 1000, 950 \rangle$		
-------------------------------------	--	--

$\langle T_0, B, 2000, 2050 \rangle$		
--------------------------------------	--	--

	$A = 950$	
--	-----------	--

	$B = 2050$	
--	------------	--

$\langle T_0 \text{ commit} \rangle$		
--------------------------------------	--	--

$\langle T_1 \text{ start} \rangle$		
-------------------------------------	--	--

$\langle T_1, C, 700, 600 \rangle$		
------------------------------------	--	--

	$C = 600$	
--	-----------	--

		B_B, B_C
--	--	------------

$\langle T_1 \text{ commit} \rangle$		
--------------------------------------	--	--

		B_A
--	--	-------

■ Nota: B_X denota al bloque que contiene a X

Modificación Inmediata de BB.DD. (cont.)

- El procedimiento de recuperación consta de dos operaciones:
 - **undo**(T_i) recupera el valor de todos los ítems de datos actualizados por T_i a sus valores originales, yendo **hacia atrás** desde el **ultimo** registro para T_i
 - **redo**(T_i) coloca el valor de todos los ítems de datos actualizados por T_i a los nuevos valores, yendo **hacia adelante** desde el **primer** registro para T_i
- Las dos operaciones deben ser idempotentes: Esto es, incluso si la operación se ejecuta múltiples veces el efecto es el mismo que si se ejecutara una vez
 - Necesario: las operaciones podrían re-ejecutarse durante la recuperación
- Recuperación tras un fallo:
 - Se necesita deshacer la transacción T_i si el log contiene el registro $\langle T_i \text{ start} \rangle$, pero NO contiene el registro $\langle T_i \text{ commit} \rangle$
 - Se necesita rehacer la transacción T_i si el log contiene AMBOS, el registro $\langle T_i \text{ start} \rangle$ Y el registro $\langle T_i \text{ commit} \rangle$
 - Las operaciones de deshacer (**undo**) se ejecutan primero, después las operaciones de rehacer (**redo**)

Modificación Inmediata de BB.DD. Ejemplo de Recuperación

- Estado del log según aparece en tres instantes de tiempo

$\langle T_0 \text{ start} \rangle$	$\langle T_0 \text{ start} \rangle$	$\langle T_0 \text{ start} \rangle$
$\langle T_0, A, 1000, 950 \rangle$	$\langle T_0, A, 1000, 950 \rangle$	$\langle T_0, A, 1000, 950 \rangle$
$\langle T_0, B, 2000, 2050 \rangle$	$\langle T_0, B, 2000, 2050 \rangle$	$\langle T_0, B, 2000, 2050 \rangle$
	$\langle T_0 \text{ commit} \rangle$	$\langle T_0 \text{ commit} \rangle$
	$\langle T_1 \text{ start} \rangle$	$\langle T_1 \text{ start} \rangle$
	$\langle T_1, C, 700, 600 \rangle$	$\langle T_1, C, 700, 600 \rangle$
		$\langle T_1 \text{ commit} \rangle$
(a)	(b)	(c)

Las acciones de recuperación en cada caso de los anteriores son:

- (a) undo (T_0): se restaura B a 2000 y A a 1000
- (b) undo (T_1) y redo (T_0): se restaura C a 700, y después A y B toman los valores de 950 y 2050 respectivamente
- (c) redo (T_0) y redo (T_1): A y B toman los valores de 950 y 2050 respectivamente. Luego C toma el valor 600

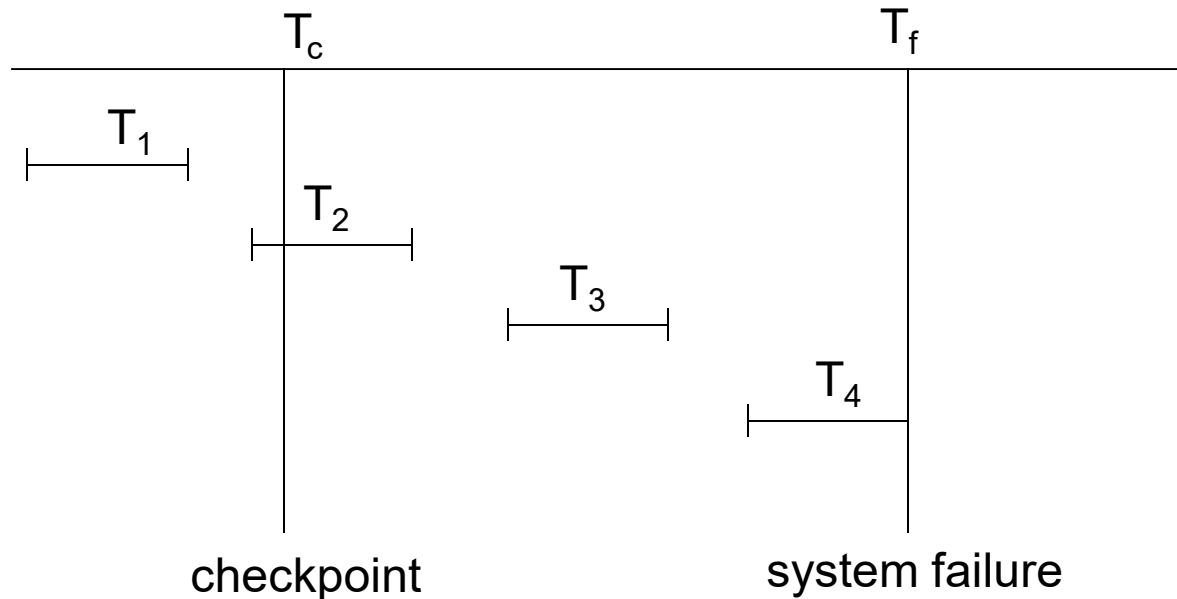
Checkpoints

- Problemas en el procedimiento de recuperación (comentados anteriormente):
 1. Buscar en el log entero es caro en tiempo
 2. Se pueden rehacer innecesariamente transacciones que ya habían volcado sus actualizaciones en la base de datos
- Ajustar el procedimiento de recuperación al ejecutar periódicamente **checkpoints**
 1. Escribir un registro de log <**Start checkpoint**> en almacenamiento estable
 2. Volcar (**output**) todos los registros de log que residan en memoria principal en almacenamiento estable
 3. Volcar (**output**) todos los registros modificados de buffer en el disco
 4. Escribir un registro de log <**End checkpoint**> en almacenamiento estable, para indicar que el proceso ha finalizado con éxito.

Checkpoints (cont.)

- Durante la recuperación necesitamos considerar **sólo** la transacción T_i más reciente que empezara antes del checkpoint, y las transacciones que empezaron tras T_i
 1. Buscar hacia atrás desde el final del log, hasta encontrar el registro de log **<Start checkpoint>** más reciente
 2. Continuar buscando hacia atrás hasta encontrar un registro **< T_i start>**
 3. Se necesita considerar la parte del log que sigue al registro **start**. Se puede ignorar durante la recuperación la parte anterior del log, y se puede borrar cuando se desee
 4. Para todas las transacciones (empezando por T_i o posteriores) que no tengan **< T_i commit>**, ejecutar **undo(T_i)** (Ejecutar solo en caso de modificación inmediata)
 5. Buscando hacia delante en el log, para todas las transacciones empezando desde T_i o posteriores con un **< T_i commit>**, ejecutar **redo(T_i)**

Checkpoints. Ejemplo



- Se puede ignorar T_1 (las actualizaciones ya están volcadas a disco debido al *checkpoint*)
- Rehacer T_2 y T_3
- Deshacer T_4

Recuperación con Transacciones Concurrentes

- Modificaremos el esquema de recuperación basado en log para permitir ejecución concurrente de múltiples transacciones
 - Todas las transacciones comparten un único buffer de disco y un único log
 - Un bloque de buffer puede tener ítems actualizados por una o más transacciones
- Supondremos control de concurrencia usando un bloqueo estricto de dos fases (i.e. Las actualizaciones de una transacción no consolidada no deberían ser visibles a otras transacciones)
 - De otra forma ¿cómo se podría deshacer si T1 modifica A, luego T2 modifica A y consolida, y finalmente T1 tiene que abortar?
- La gestión del Log se realiza como antes
 - Los registros de log de diferentes transacciones pueden estar entremezcladas en el log
- La técnica de checkpoint y las acciones que se toman durante la recuperación tienen que cambiarse
 - Puesto que varias transacciones pueden estar activas cuando se ejecuta un checkpoint

Recuperación con Transacciones Concurrentes (cont.)

- Los checkpoints se realizan como antes, excepto que el registro de log del checkpoint log record ahora tiene la forma
 <Start checkpoint L>
donde L es la lista de las transacciones activas en el momento del checkpoint
 - Supondremos que no hay actualizaciones pendientes mientras se ejecuta el checkpoint (aunque no lo necesitaremos mas adelante)
- Cuando el sistema se recupera de una caída, lo primero que hace es:
 1. Iniciar dos listas *undo-list* y *redo-list* a valores vacíos
 2. Buscar en el log hacia atrás desde el final, parándose cuando encuentra el primer registro **<Start checkpoint L>**
Para cada registro encontrado durante esta búsqueda:
 - Si el registro es **< T_i commit>**, añadir T_i a *redo-list*
 - Si el registro es **< T_i start>**, entonces si T_i no está en *redo-list*, añadir T_i a *undo-list*
 3. Para cada T_i en L , si T_i no está en *redo-list*, añadir T_i a *undo-list*

Recuperación con Transacciones Concurrentes (cont.)

- En este punto *undo-list* consta de transacciones incompletas que deben deshacerse, y *redo-list* consta de transacciones terminadas que deben rehacerse
- La recuperación continúa como sigue:
 1. Buscar hacia atrás en el log desde el registro más reciente, parándose cuando se hayan encontrado registros $\langle T_i \text{ start} \rangle$ para cada T_i en *undo-list*
 - Durante la búsqueda, ejecutar **undo** para cada registro de log que pertenezca a las transacciones en *undo-list*
 2. Localizar el registro **<Start checkpoint L>** más reciente
 3. Buscar hacia delante en el log desde este registro **<Start checkpoint L>** hasta el final del log
 - Durante la búsqueda, ejecutar **redo** para cada registro de log que pertenezca a las transacciones en *redo-list*

Recuperación. Ejemplo

- Seguir los pasos del algoritmo de recuperación en el siguiente log:

```
<T0 start>  
<T0, A, 0, 10>  
<T0 commit>  
<T1 start>      /* Búsqueda del paso 1 llega hasta aquí */  
<T1, B, 0, 10>  
<T2 start>  
<T2, C, 0, 10>  
<T2, C, 10, 20>  
<Start checkpoint {T1, T2}>  
<End checkpoint >  
<T3 start>  
<T3, A, 10, 20>  
<T3, D, 0, 10>  
<T3 commit>
```

Recuperación. Ejemplo

- En el caso anterior se suspende ejecución de transacciones en checkpoint, pero puede darse el caso de que no sea así.

```
<T0 start>
<T0, A, 0, 10>
<T0 commit>
<T1 start>      /* Búsqueda del paso 1 llega hasta aquí */
<T1, B, 0, 10>
<T2 start>
<T2, C, 0, 10>
<Start checkpoint {T1, T2}>
<T2, C, 10, 20>
<End checkpoint >
<T3 start>
<T3, A, 10, 20>
<T3, D, 0, 10>
<T3 commit>
```