75.15 / 75.28 / 95.05 - Base de Datos

operaciones simultaneas sobre los datos

Concurrencia y Transacciones

Mariano Beiró

Dpto. de Computación - Facultad de Ingeniería (UBA)

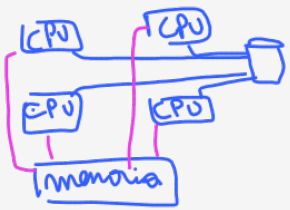
17 de mayo de 2022

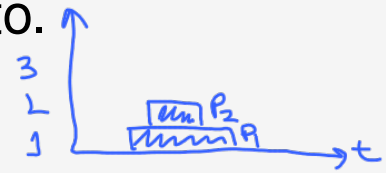
Temas

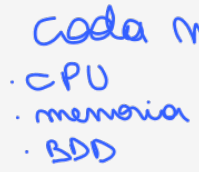
Concurrencia

Sistemas monoprocesador, multiprocesador y distribuídos

- En las bases de datos reales múltiples usuarios realizan operaciones de consulta y/o actualización simultáneamente.
- Nos gustaría aprovechar la capacidad de procesamiento lo mejor posible al atender a los usuarios.
- Sistemas monoprocesador **cuando solapamos procesos podemos realizar cosas en paralelo en otros recursos como por ejemplo escribir en disco**
  - Permiten hacer *multitasking* (multitarea). Varios hilos o procesos pueden estar corriendo concurrentemente.
- Sistemas multiprocesador y sistemas distribuídos
  - Disponen de varias unidades de procesamiento que funcionan en forma simultánea.
  - Suelen replicar la base de datos, disponiendo de varias copias de algunas tablas (o fragmentos de tabla) en distintas unidades de procesamiento.







Concurrencia

Transacciones

En este contexto utilizaremos el concepto de transacción como *"unidad lógica de trabajo"* o, más en detalle, *"secuencia ordenada de instrucciones atómicas"*.

- Una misma transacción puede realizar varias operaciones de consulta/ABM durante su ejecución.
- Antes de existir el multitasking, las transacciones se serializaban. Hasta tanto no se terminara una, no se iniciaba la siguiente.
- Serializar es en general una mala idea. Nos gustaría poder ejecutarlas en forma simultánea, aunque garantizando ciertas propiedades básicas.

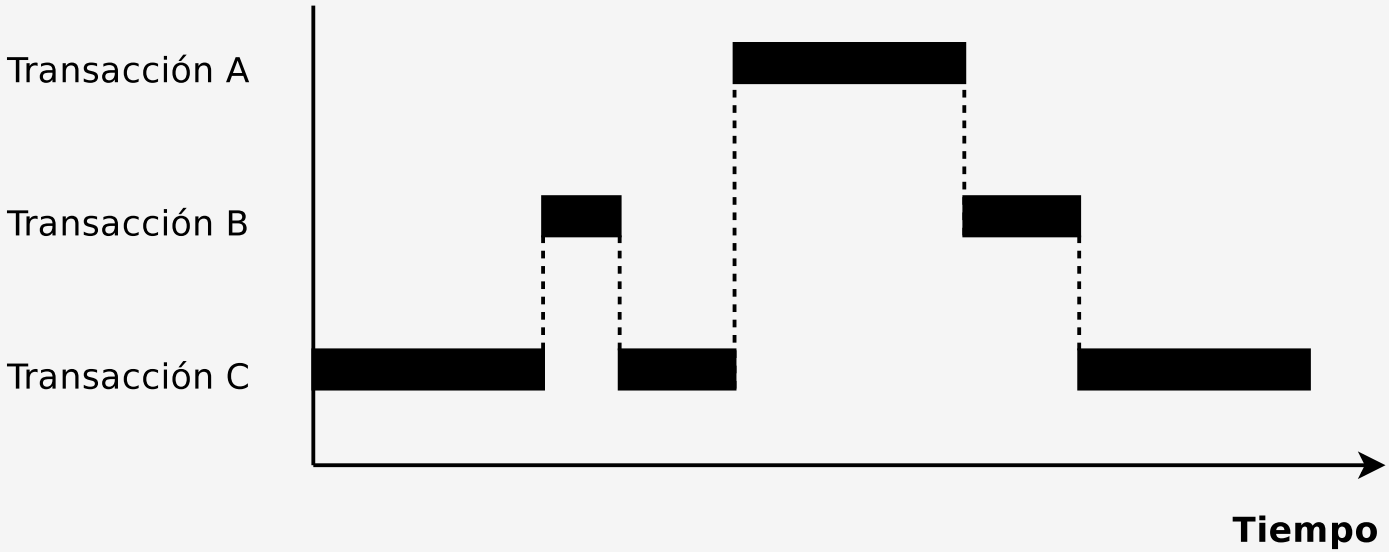
Concurrencia

Concepto

- La concurrencia es la posibilidad de ejecutar múltiples transacciones (tareas, en la jerga de los sistemas operativos) en forma simultánea.
- En sistemas distribuídos y multiprocesador, vamos a querer aprovechar toda la capacidad de cómputo:
  - Si dos transacciones utilizan sets de datos distintos deberían poder ejecutarse concurrentemente en distintos procesadores.
- Aún en sistemas monoprocesador, serializar no es una opción:
  - No es deseable que la ejecución de una transacción lenta demore a otras de ejecución rápida.
  - Mientras una transacción espera que el SO escriba una página en disco, otra transacción podría realizar una operación en memoria.
- El problema que genera la ejecución concurrente es la gestión de los recursos compartidos. Al nivel de los SGBDs, los recursos compartidos son los datos, a los cuales distintas transacciones querrán acceder en forma simultánea.

Modelo de procesamiento concurrente [ELM16 20.1.1]

- El modelo de procesamiento que utilizaremos es el de concurrencia solapada *(interleaved concurrency)*, que considera las siguientes hipótesis:
  1. Disponemos de un único procesador que puede ejecutar múltiples transacciones simultáneamente.
  2. Cada transacción está formada por una secuencia de instrucciones *atómicas*, que el procesador ejecuta *de a una a la vez*.
  3. En cualquier momento el *scheduler* puede suspender la ejecución de una transacción, e iniciar o retomar la ejecución de otra.



Modelo de procesamiento concurrente

- Si tuvieramos múltiples unidades de procesamiento, el modelo a utilizar sería el de procesamiento paralelo.
- Las herramientas teóricas que desarrollaremos para concurrencia solapada, son en su mayoría extensibles al caso de procesamiento paralelo.

Ejemplo de ejecución concurrente

- Consideremos los siguientes esquemas de la base de datos de un banco:
  - Sucursales(codigo, localidad, ...)
    (110, 'La Paternal', ...)
  - Clientes(CUIT, nombre, cod_sucursal, saldo)
    ('27-40182490-5', 'Juana Hass', 110, 2500)
- Por un lado, un usuario quiere consultar los nombres de los clientes de la sucursal de La Paternal, mientras que otro intenta cambiar al cliente cuyo CUIT es '27-40182490-5' de la sucursal 110 a la sucursal 220.

Operación 1	Operación 2
$PAT \leftarrow \sigma_{nombre='LaPaternal'}(Sucursales)$	
$ID\_PAT \leftarrow \pi_{codigo}(PAT)$	
	$UN\_CLI \leftarrow \sigma_{CUIT='27\text{-}40182490\text{-}5'}(Clientes)$
	$Clientes \leftarrow \sigma_{CUIT\neq'27\text{-}40182490\text{-}5'}(Clientes)$
$CLI \leftarrow ID\_PAT \bowtie_{codigo=cod\_sucursal} Clientes$	
	$Clientes \leftarrow Clientes \cup \{'27\text{-}40182490\text{-}5', 'Juana\ Hass', 220, 2500\}$
$\pi_{nombre}(CLI)$	

- Nota: El ordenamiento vertical corresponde al orden temporal de ejecución en nuestro modelo. Como vemos, el procesador sólo puede ejecutar una instrucción de una de las dos transacciones a la vez.

Ejemplo de ejecución concurrente

- En este ejemplo, las instrucciones se corresponden con las operaciones del álgebra relacional.
  - Problema: Una junta puede ser muy costosa. El SGBD debería poder solaparla con otras transacciones más sencillas.
- $\rightarrow$ Dado que las instrucciones deben ser atómicas, es conveniente que nuestras instrucciones tengan un nivel de granularidad más pequeño.

Modelo de datos

[ELM16 20.1.2]

Items e instrucciones atómicas

- Consideraremos que nuestra base de datos está formada por ítems.
- Un ítem puede representar:
  - El valor de un atributo en una fila determinada de una tabla.
  - Una fila de una tabla.
  - Un bloque del disco.
  - Una tabla.
- Las instrucciones atómicas básicas de una transacción sobre la base de datos serán:
  - leer_item(X): Lee el valor del ítem X, cargándolo en una variable en memoria
  - escribir_item(X): Ordena escribir el valor que está en memoria del ítem X en la base de datos
- Nota: El tamaño de ítem escogido se conoce como granularidad, y afecta sustancialmente al control de concurrencia. [ELM16 21.5]

Observaciones

- Observación 1: Desde ya, el código de la transacción contendrá instrucciones que involucran la manipulación de datos en memoria (por ejemplo, realizar la junta en memoria de dos tablas ya leídas), pero las mismas no afectan al análisis de concurrencia.
- Observación 2: Ordenar escribir no es lo mismo que efectivamente escribir en el medio de almacenamiento persistente en que se encuentra la base de datos! El nuevo valor podría quedar temporalmente en un buffer en memoria.

Transacciones

Concepto

[SILB19 17.1]

- Una transacción es una unidad lógica de trabajo en los SGBD.
- Es una secuencia ordenada de instrucciones que deben ser ejecutadas en su totalidad o bien no ser ejecutadas, al margen de la interferencia con otras transacciones simultáneas. completitud
- Ejemplos:
  - Una transferencia de dinero de una cuenta corriente bancaria a otra.
  - La reserva de un pasaje aéreo.

Transacciones

Propiedades ACID

- La ejecución de transacciones por un SGBD debería cumplir con 4 propiedades deseables, conocidas como propiedades ACID:
  - Atomicidad: Desde el punto de vista del usuario, las transacciones deben ejecutarse de manera atómica. Esto quiere decir que, o bien la transacción se realiza por completo, o bien no se realiza.
  - Consistencia: Cada ejecución, por sí misma, debe preservar la consistencia de los datos. La consistencia se define a través de reglas de integridad: condiciones que deben verificarse sobre los datos en todo momento.
    - Por ejemplo, la base de datos de una empresa puede tener como restricción que no pueda haber más de un gerente por departamento.

Transacciones

Propiedades ACID

- Aislamiento: El resultado de la ejecución concurrente de las transacciones debe ser el mismo que si las transacciones se ejecutaran en forma aislada una tras otra, es decir en forma serial. La ejecución concurrente debe entonces ser equivalente a alguna ejecución serial.
- Durabilidad: Una vez que el SGBD informa que la transacción se ha completado, debe garantizarse la persistencia de la misma, independientemente de toda falla que pueda ocurrir.

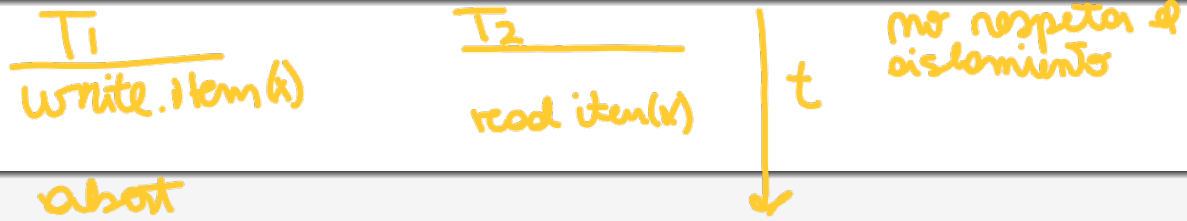
Transacciones

Recuperación

- Para garantizar las propiedades ACID, los SGBD disponen de mecanismos de recuperación que permiten deshacer/rehacer una transacción en caso de que se produzca un error o falla.
  - Se debe garantizar la visión de "todo o nada" de las transacciones (atomicidad), y que todos los cambios realizados por la transacción sean efectivamente almacenados.
- Para ello es necesario agregar a la secuencia de instrucciones de cada transacción algunas instrucciones especiales:
  - begin: Indica el comienzo de la transacción.
  - commit: Indica que la transacción ha terminado exitosamente, y se espera que su resultado haya sido efectivamente almacenado en forma persistente.
  - abort: Indica que se produjo algún error o falla, y que por lo tanto todos los efectos de la transacción deben ser deshechos *(rolled back)*.

Anomalías

Problema de la lectura sucia



- Cuando se ejecutan transacciones en forma concurrente se da lugar a distintas situaciones anómalas que pueden violar las propiedades ACID.
- La anomalía de la lectura sucia (dirty read) se presenta cuando una transacción T_2 lee un ítem que ha sido modificado por otra transacción T_1.
- Si luego T_1 se deshace, la lectura que hizo T_2 no es válida en el sentido de que la ejecución resultante puede no ser equivalente a una ejecución serial de las transacciones.
- También se lo conoce con el nombre de *"Temporary update"* ó *"Read uncommitted data"*.
- Es un conflicto de tipo WR: $W_{T_1}(X)...R_{T_2}(X)...(a_{T_1}$ ó $c_{T_1})$.

Anomalías

Problema de la lectura sucia: Ejemplo

- T_1 transfiere 100 de la cuenta A a la cuenta B, mientras que T_2 aplica una tasa que incrementa el capital de ambas cuentas en un 10%. Los saldos iniciales de las cuentas A y B son de \$1100 y \$900 respectivamente. Consideremos el siguiente solapamiento:

Transacción T_1	Transacción T_2
begin	
	begin
leer_item(A)	
$A = A - 100$	
escribir_item(A)	
	leer_item(A)
	$A = A \cdot 1,10$
	escribir_item(A)
	leer_item(B)
	$B = B \cdot 1,10$
	escribir_item(B)
	commit
abort	

- La lectura que hace T_2 de A es una lectura sucia, porque la transacción T_1 luego será abortada.

Anomalías

Problema de la actualización perdida y lectura no repetible

- La anomalía de la actualización perdida (lost update) ocurre cuando una transacción modifica un ítem que fue leído anteriormente por una primera transacción que aún no terminó.
- En este caso, si la primera transacción luego modifica y escribe el ítem que leyó, el valor escrito por la segunda se perderá.
- Si en cambio la primera transacción volviera a leer el ítem luego de que la segunda lo escribiera, se encontraría con un valor distinto. En este caso se lo conoce como lectura no repetible (unrepeatable read).
- Ambas situaciones presentan un conficto de tipo RW *(read-write)*, seguido por otro de tipo WW ó WR, respectivamente.
- Caracterización: $R_{T_1}(X)...W_{T_2}(X)...(a_{T_1}$ ó $c_{T_1})$.

Anomalías

Problemas de la actualización perdida y lectura no repetible: Ejemplo

- La transacción T_1 realiza un depósito de \$100 en la cuenta A, mientras que T_2 extrae \$100 de la misma. El saldo inicial de A es de \$500.

Transacción T_1	Transacción T_2
begin	
	begin
leer_item(A)	
$A = A + 100$	
	leer_item(A)
	$A = A - 100$
escribir_item(A)	
	escribir_item(A)
commit	
	commit

- Claramente, el saldo final de la cuenta A debía ser de \$500.
- Sin embargo, con este solapamiento la cuenta termina con un saldo de \$400, porque T_2 lo actualizó después de que T_1 lo leyera.

Anomalías

Problema de la escritura sucia

- La anomalía de la escritura sucia (dirty write) ocurre cuando una transacción T_2 escribe un ítem que ya había sido escrito por otra transacción T_1 que luego se deshace. doble escritura
- El problema se dará si los mecanismos de recuperación vuelven al ítem a su valor inicial, deshaciendo la modificación realizada por T_2.

Transacción T_1	Transacción T_2
begin	
	begin
leer_item(A)	
$A = A + 100$	
escribir_item(A)	
	leer_item(A)
	$A = A + 200$
	escribir_item(A)
	commit
abort	

- También se conoce con el nombre de *overwrite uncommitted*.
- Es un conficto de tipo WW *(write-write)*: $W_{T_1}(X)...W_{T_2}(X)...(a_{T_1}$ ó $c_{T_1})$.

Anomalías

Problema del fantasma

- La anomalía del fantasma (phantom) se produce cuando una transacción T_1 observa un conjunto de ítems que cumplen determinada condición, y luego dicho conjunto cambia porque algunos de sus items son modificados/creados/eliminados por otra transacción T_2.
- Si está modificación se hace mientras T_1 aún se está ejecutando, T_1 podría encontrarse con que el conjunto de ítems que cumplen la condición cambió.
- Esta anomalía atenta contra la serializabilidad[1].
- Caracterización: $R_{T_1}(\{X|cond\})...W_{T_2}(X_{cond})...(a_{T_1}$ ó $c_{T_1})$, en donde el ítem X_{cond} cumple con la condición con que fueron seleccionados los X en la lectura.

[1] Para resolverla suelen necesitarse *locks* a nivel de tabla, o bien *locks* de predicados.

Anomalías

Problema del fantasma: Ejemplo

- Consideremos una única cuenta *A* con un saldo de $400. Una transacción T_1 calcula la cantidad de cuentas con saldo menor a $500 y cuando termina cuenta aquellas con saldo igual ó superior a $10000. Mientras tanto, otra transacción crea una nueva cuenta *B* con saldo de $12000 y una cuenta *C* con saldo de $300.

Transacción T_1	Transacción T_2
begin	
	begin
leer_item(A)	
menores_500=1	
	B=12000
	escribir_item(B)
leer_item(A)	
leer_item(B)	
mayores_10000=1	
	C=300
	escribir_item(C)
	commit
commit	

- El resultado no será equivalente a ninguno de los 2 órdenes seriales posibles.

Notación

Transacción

[ELM16 20.4.1]

- Para analizar la serializabilidad de un conjunto de transacciones en nuestro modelo de concurrencia solapada, utilizaremos la siguiente notación breve para las instrucciones:
  - $R_T(X)$: La transacción T lee el ítem X.
  - $W_T(X)$: La transacción T escribe el ítem X.
  - b_T: Comienzo de la transacción T.
  - c_T: La transacción T realiza el *commit*.
  - a_T: Se aborta la transacción T *(abort)*.
- Con esta notación, podemos escribir una transacción general T como una lista de instrucciones $\{I_T^1; I_T^2; ...; I_T^{m(T)}\}$, en donde $m(T)$ representa la cantidad de instrucciones de T.
- Ejemplo:
  - $T_1 : b_{T_1}; R_{T_1}(X); R_{T_1}(Y); W_{T_1}(Y); c_{T_1};$
  - $T_2 : b_{T_2}; R_{T_2}(X); W_{T_2}(X); c_{T_2};$

Notación

Solapamiento

Solap. ej:
$T_1 = x, y, 3$
$T_2 = a, b$
$S_1 = x\,a\,y\,b\,3$ ✓
$S_2? = b\,a\,x\,y\,3$ ✗

$\frac{5!\,3!}{2!} = 10!$ combs

- Un solapamiento entre dos transacciones T_1 y T_2 es una lista de $m(T_1) + m(T_2)$ instrucciones, en donde cada instrucción de T_1 y T_2 aparece una única vez, y las instrucciones de cada transacción conservan el orden entre ellas dentro del solapamiento.
- ¿Cuántos solapamientos distintos existen entre T_1 y T_2?
  $\rightarrow \frac{(m(T_1)+m(T_2))!}{m(T_1)!m(T_2)!}$
- En el ejemplo anterior, podemos representar un solapamiento entre T_1 y T_2 de la siguiente manera:
  - $b_{T_1}; R_{T_1}(X); b_{T_2}; R_{T_2}(X); W_{T_2}(X); R_{T_1}(Y); W_{T_1}(Y); c_{T_2}; c_{T_1};$
- La pregunta que nos haremos es si dicho solapamiento es serializable ó no.

Ejecución serial

Definición

[ELM16 20.5.1]

- Dado un conjunto de transacciones $T_1, T_2, ..., T_n$ una ejecución serial es aquella en que las transacciones se ejecutan por completo una detrás de otra, en base a algún orden $T_{i_1}, T_{i_2}, ..., T_{i_n}$.
- Ejemplo:
  - $T_1 : b_{T_1}; R_{T_1}(X); R_{T_1}(Y); W_{T_1}(Y); c_{T_1};$
  - $T_2 : b_{T_2}; R_{T_2}(X); W_{T_2}(X); c_{T_2};$
- Para este par de transacciones existen dos ejecuciones seriales posibles:
  - $b_{T_1}; R_{T_1}(X); R_{T_1}(Y); W_{T_1}(Y); c_{T_1}; b_{T_2}; R_{T_2}(X); W_{T_2}(X); c_{T_2};$
  - $b_{T_2}; R_{T_2}(X); W_{T_2}(X); c_{T_2}; b_{T_1}; R_{T_1}(X); R_{T_1}(Y); W_{T_1}(Y); c_{T_1};$
- ¿Cuántas ejecuciones seriales distintas existen entre n transacciones? $\rightarrow n!$ #

Serializabilidad

Definición

- Decimos que un solapamiento de un conjunto de transacciones $T_1, T_2, ..., T_n$ es serializable cuando la ejecución de sus instrucciones en dicho orden deja a la base de datos en un estado equivalente a aquél en que la hubiera dejado alguna ejecución serial de $T_1, T_2, ..., T_n$.
- Nos interesa que los solapamientos producidos sean serializables, porque ellos garantizan la propiedad de *aislamiento* de las transacciones.
- Pero, ¿cómo evaluamos esta "equivalencia" entre ordenes de ejecución?
- Deberíamos no sólo mirar nuestra base de datos actual, que depende de un estado inicial particular anterior a la ejecución de las transacciones, sino pensar en cualquier estado inicial posible.

Equivalencia de solapamientos

Nociones

- Existen entonces distintas nociones de equivalencia entre órdenes de ejecución de transacciones:
  - Equivalencia de resultados: Cuando, dado un estado inicial particular, ambos órdenes de ejecución dejan a la base de datos en el mismo estado.
  - Equivalencia de conflictos: Cuando ambos órdenes de ejecución poseen los mismos conflictos entre instrucciones.
    - Esta noción es particularmente interesante porque no depende del estado inicial de la base de datos. Es la más fuerte de las tres.
  - Equivalencia de vistas: Cuando en cada órden de ejecución, cada lectura $R_{T_i}(X)$ lee el valor escrito por la misma transacción j, $W_{T_j}(X)$. Además se pide que en ambos órdenes la última modificación de cada ítem X haya sido hecha por la misma transacción.
- Desarrollaremos a continuación la noción de equivalencia de conflictos.

Conflictos

Definición

- Dado un orden de ejecución, un conflicto es un par de instrucciones (I_1, I_2) ejecutadas por dos transacciones *distintas* T_i y T_j, tales que I_2 se encuentra más tarde que I_1 en el orden, y que responde a alguno de los siguientes esquemas:
  - $(R_{T_i}(X), W_{T_j}(X))$: Una transacción escribe un ítem que otra leyó.
  - $(W_{T_i}(X), R_{T_j}(X))$: Una transacción lee un ítem que otra escribió.
  - $(W_{T_i}(X), W_{T_j}(X))$: Dos transacciones escriben un mismo ítem.
- En otras palabras, tenemos un conflicto cuando dos transacciones *distintas* ejecutan instrucciones sobre un mismo ítem X, y al menos una de las dos instrucciones es una escritura.
- Todo par de instrucciones *consecutivas* (I_1, I_2) de un solapamiento que no constituye un conflicto puede ser invertido en su ejecución (es decir, reemplazado por el par (I_2, I_1)) obteniendo un solapamiento equivalente por conflictos al inicial.

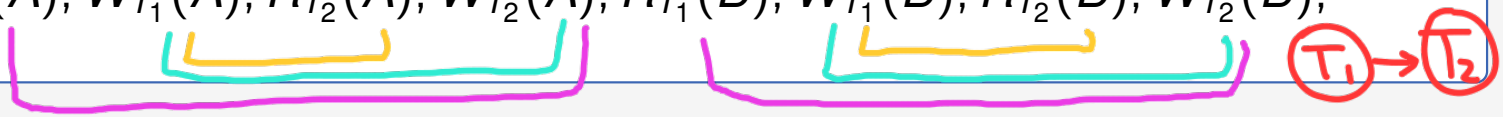
Serializabilidad por conflictos

Ejemplo

Si no hay conflicto, puedo hacer switch y obtener algo equivalente (no pueden ser del mismo T)

Indicar si el siguiente solapamiento de dos transacciones T_1 y T_2 es serializable por conflictos.

$R_{T_1}(A); W_{T_1}(A); R_{T_2}(A); W_{T_2}(A); R_{T_1}(B); W_{T_1}(B); R_{T_2}(B); W_{T_2}(B);$



Comenzamos buscando los conflictos existentes:

$$(W_{T_1}(A); R_{T_2}(A)), ((R_{T_1}(A), W_{T_2}(A)), ((W_{T_1}(A), W_{T_2}(A)),$$

$$(W_{T_1}(B); R_{T_2}(B)), ((R_{T_1}(B), W_{T_2}(B)), ((W_{T_1}(B), W_{T_2}(B))$$

Observemos que T_2 podría realizar la lectura y escritura de B antes de que T_1 realice la lectura y escritura de A, sin cambiar el resultado:

$$R_{T_1}(A); W_{T_1}(A); R_{T_1}(B); W_{T_1}(B); R_{T_2}(A); W_{T_2}(A); R_{T_2}(B); W_{T_2}(B);$$

Obtenemos así una ejecución serial (concretamente: T_1, T_2) que es equivalente por conflictos al solapamiento inicial.
Entonces el solapamiento es serializable por conflictos.

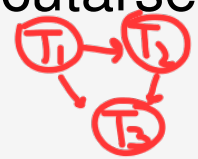
Grafo de precedencias

Construcción

[ELM16 20.5.2; SILB19 17.6]

- La serializabilidad por conflictos puede ser evaluada a través del grafo de precedencias.
- Dado un conjunto de transacciones $T_1, T_2, ..., T_n$ que acceden a determinados ítems $X_1, X_2, ..., X_p$, el grafo de precedencias es un *grafo dirigido simple* que se construye de la siguiente forma:
  1. Se crea un nodo por cada transacción $T_1, T_2, ..., T_n$.
  2. Se agrega un arco entre los nodos T_i y T_j (con $i \neq j$) si y sólo si existe algún conflicto de la forma $(R_{T_i}(X_k), W_{T_j}(X_k))$, $(W_{T_i}(X_k), R_{T_j}(X_k))$ ó $(W_{T_i}(X_k), W_{T_j}(X_k))$.
- Cada arco (T_i, T_j) en el grafo representa una precedencia entre T_i y T_j, e indica que para que el resultado sea equivalente por conflictos a una ejecución serial, entonces en dicha ejecución serial T_i debe T_j preceder a (es decir, "ejecutarse antes que") T_j.

Puede haber grafos sin aristas (conflictos)



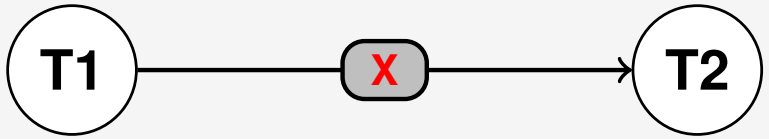
Ej serial eg: T1,T2,T3

Grafo de precedencias

Construcción

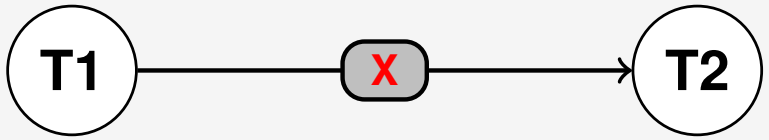
- Opcionalmente podemos etiquetar el arco con el nombre del recurso que causa el conflicto.
- Conflicto RW:

$R_{T_1}(X); W_{T_2}(X);$



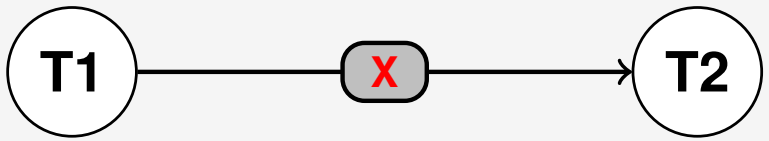
- Conflicto WR:

$W_{T_1}(X); R_{T_2}(X);$



- Conflicto WW:

$W_{T_1}(X); W_{T_2}(X);$



Grafo de precedencias

Ejemplo

Ejemplo

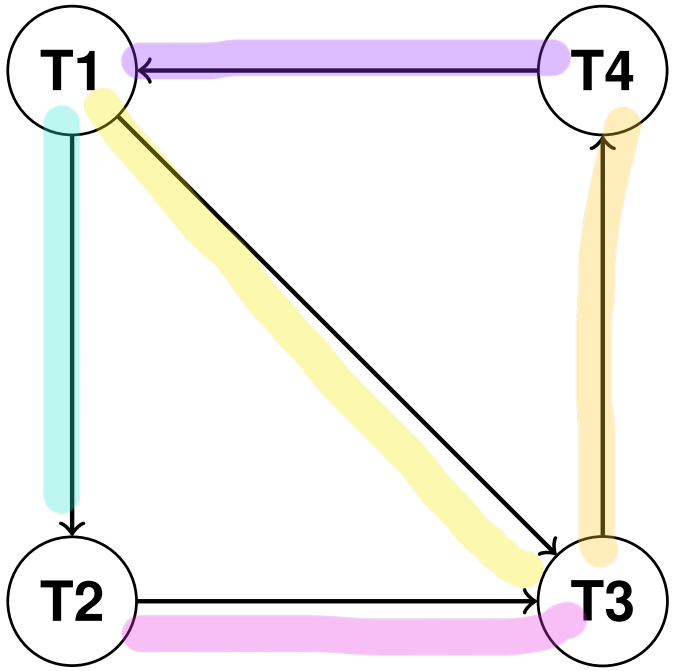
Construya el grafo de precedencias del siguiente solapamiento:

Transacción T_1	Transacción T_2	Transacción T_3	Transacción T_4
leer_item(X)			
	leer_item(X)		
escribir_item(Y)			
		leer_item(Y)	
	leer_item(Y)		
	escribir_item(X)		
		leer_item(W)	
		escribir_item(Y)	
			leer_item(W)
			leer_item(Z)
			escribir_item(W)
leer_item(Z)			
escribir_item(Z)			

Grafo de precedencias

Ejemplo

Solución



hay ciclos!
⇒ No serializable
x conflictos

Grafo de precedencias

Resultados

- Un orden de ejecución es serializable por conflictos si y sólo si su grafo de precedencias no tiene ciclos. (Prueba en [GM09 18.2.3])
- Si un orden de ejecución es serializable por conflictos, el orden de ejecución serial equivalente puede ser calculado a partir del grafo de precedencias, utilizando el algoritmo de ordenamiento topológico.
  - Dado un grafo dirigido acíclico, un orden topológico es un ordenamiento de los nodos del grafo tal que para todo arco (x, y), el nodo x precede al nodo y.
  - Es sencillo encontrar uno eliminando los nodos que no poseen predecesores en forma recursiva y de a uno a la vez, hasta que no quede ninguno. El orden en que los nodos fueron eliminados constituirá un orden topológico del grafo.

Control de concurrencia

Enfoques

- El problema del control de concurrencia en vistas de garantizar el aislamiento admite dos enfoques:
  - Enfoque pesimista: Busca garantizar que no se produzcan conflictos.
    - Control de concurrencia basado en *locks*.
  - Enfoque optimista: Consiste en "dejar hacer" a las transacciones, y deshacer *(rollback)* una de ellas si en *fase de validación* se descubre un conflicto. Conveniente cuando la probabilidad de conflicto es baja.
    - Control de concurrencia basado en *timestamps*.
    - *Snapshot Isolation*.
    - *Control de concurrencia multiversión (MVCC).*

Control de concurrencia basado en *locks*

- En este método, el SGBD utiliza *locks* para bloquear a los recursos (los ítems) y no permitir que más de una transacción los use en forma simultánea.
- Los *locks* son insertados por el SGBD como instrucciones especiales en medio de la transacción.
- Una vez insertados, las transacciones compiten entre ellas por su ejecución.
- Veremos que es posible -*aunque no trivial*- garantizar la serializabilidad utilizando *locks*.

Control de concurrencia basado en locks

Locks

[ELM16 21.1.1]

- Los *locks* ó candados son variables asociadas a determinados recursos, y que permiten regular el acceso a los mismos en los sistemas concurrentes.
- Son una herramienta para resolver el problema de la exclusión mutua.
- Un *lock* debe disponer de dos primitivas de uso, que permiten tomar y liberar el recurso X asociado al mismo:
  - Acquire(X) ó Lock(X) (L(X)).
  - Release(X) ó Unlock(X) (U(X)).
- Tienen caracter bloqueante: Cuando una transacción *tiene* un *lock* sobre un ítem X, ninguna otra transacción puede adquirir un *lock* sobre el mismo ítem hasta tanto la primera no lo libere.
- Es fundamental que dichas primitivas sean atómicas. Es decir, la ejecución de la primitiva $Lock(X)$ sobre un recurso X no puede estar solapada con una ejecución semejante en otra transacción.
  - ¡No trivial! $Lock(X)$ requiere leer y escribir una variable.

Control de concurrencia basado en locks

Locks - Ejemplo

- Ejemplo:

Transacción T_1	Transacción T_2
begin	
	begin
lock(A)	
lock(B)	
leer_item(A)	
leer_item(B)	
$A = A + B$	
escribir_item(A)	
unlock(A)	
unlock(B)	
	lock(B)
	leer_item(B)
	unlock(B)
commit	
	commit

Control de concurrencia basado en locks

Tipos de locks

- En general, los SGBD implementan *locks* de varios tipos. Los dos tipos de *locks* principales son:
  - Locks de escritura o "de acceso exclusivo" ($L_{EX}(X)$).
  - Locks de lectura o "de acceso compartido" ($L_{SH}(X)$).
- Cuando una transacción posee un *lock* de acceso exclusivo *(EX)* sobre un ítem, ninguna otra transacción puede tener un *lock* de ningún tipo sobre ese mismo ítem.
- Pero muchas transacciones pueden poseer *locks* de acceso compartido *(SH, shared)* sobre un mismo ítem simultáneamente.
- Tabla de compatibilidad de locks:

↓ Se puede otorgar otro de tipo

Si alguien tiene un lock →	SH	EX
SH	✓	✗
EX	✗	✗

Control de concurrencia basado en locks

Protocolo de lock de dos fases (2PL)

[ELM16 21.1.2; SILB19 18.1.3]

- El empleo de *locks* por sí sólos no basta. Una transacción podría adquirir un *lock* sobre un ítem para leerlo, luego liberarlo, y más tarde volver a adquirirlo para leerlo y modificarlo. Si en el intervalo otra transacción lo lee y escribe (aún tomando un *lock*), podría producirse la anomalía de la lectura no repetible.
- El protoloco más comunmente utilizado para la adquisición y liberación de locks es el protocolo de lock de dos fases (2PL, two-phase lock).
- El 2PL se rige por la siguiente regla:

Protocolo de lock de dos fases (2PL)

Una transacción no puede adquirir un *lock* luego de haber liberado un *lock* que había adquirido.

Control de concurrencia basado en locks

Protocolo de lock de dos fases (2PL)

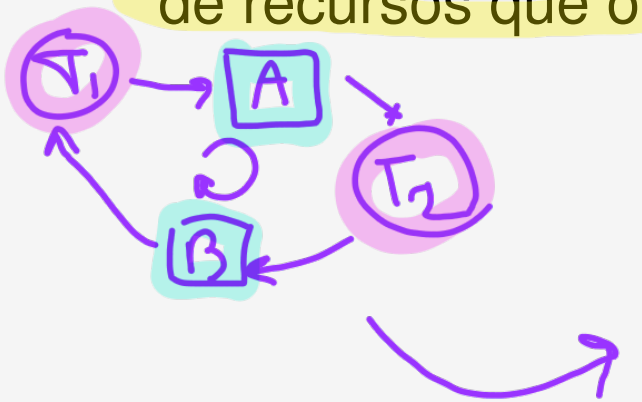
- La regla divide naturalmente en dos fases a la ejecución de la transacción:
  - Una fase de adquisición de locks, en la que la cantidad de locks adquiridos crece.
  - Una fase de liberación de locks, en que la cantidad de locks adquiridos decrece.
- El cumplimiento de este protocolo es condición suficiente para garantizar que cualquier orden de ejecución de un conjunto de transacciones sea serializable. (Prueba en [GM09 18.3.4])
- Sin embargo, la utilización de *locks* introduce otros dos problemas potenciales que antes no teníamos:
  - Bloqueo *(deadlock)*
  - Inanición o postergación indefinida *(livelock)*

Control de concurrencia basado en locks

Deadlocks

[ELM16 21.1.3; SILB19 18.2]

- Un deadlock es una condición en que un conjunto de transacciones quedan cada una de ellas bloqueada a la espera de recursos que otra de ellas posee.



Transacción T_1	Transacción T_2
begin	
	begin
lock(A)	
leer_item(A)	
	lock(B)
	leer_item(B)
lock(B) †	
	lock(A) †

- Observemos que el problema no habría existido si las transacciones hubieran adquirido los *locks* en el mismo orden.

Control de concurrencia basado en locks

Deadlocks - Mecanismos de prevención

- Mecanismos de prevención de *deadlocks*:
  1. Que cada transacción adquiera todos los *locks* que necesita antes de comenzar su primera instrucción, y en forma simultánea. ($Lock(X_1, X_2, ...X_n)$).
  2. Definir un ordenamiento de los recursos, y obligar a que luego todas las transacciones respeten dicho ordenamiento en la adquisición de *locks*.
  3. Métodos basados en timestamps.
- La limitación del enfoque preventivo es que será necesario saber qué recursos serán necesarios de antemano.

Control de concurrencia basado en locks

Deadlocks - Métodos de detección

- Mecanismos de detección de *deadlocks*:
  1. Analizar el grafo de alocación de recursos: un grafo dirigido que posee a las transacciones y los recursos como nodos, y en el cual se coloca un arco de una transacción a un recurso cada vez que una transacción espera por un recurso, y un arco de un recurso a una transacción cada vez que la transacción posee el *lock* de dicho recurso.
     - Cuando se detecta un ciclo en este grafo, se aborta *(rollback)* una de las transacciones involucradas.
     - El concepto es muy similar al del grafo de precedencias para un solapamiento.
  2. Definir un *timeout* para la adquisición del $Lock(X)$, después del cual se aborta la transacción.

Control de concurrencia basado en locks

Inanición

- La inanición es una condición vinculada con el *deadlock*, y ocurre cuando una transacción no logra ejecutarse por un periodo de tiempo indefinido.
- Puede suceder por ejemplo, si ante la detección de un *deadlock* se elije siempre a la misma transacción para ser abortada.
- La solución más común consiste en encolar los pedidos de *locks*, de manera que las transacciones que esperan desde hace más tiempo por un recurso tengan prioridad en la adquisición de su *lock*.

Control de concurrencia basado en locks

Acceso a estructuras de árbol

[SILB19 18.10.2]

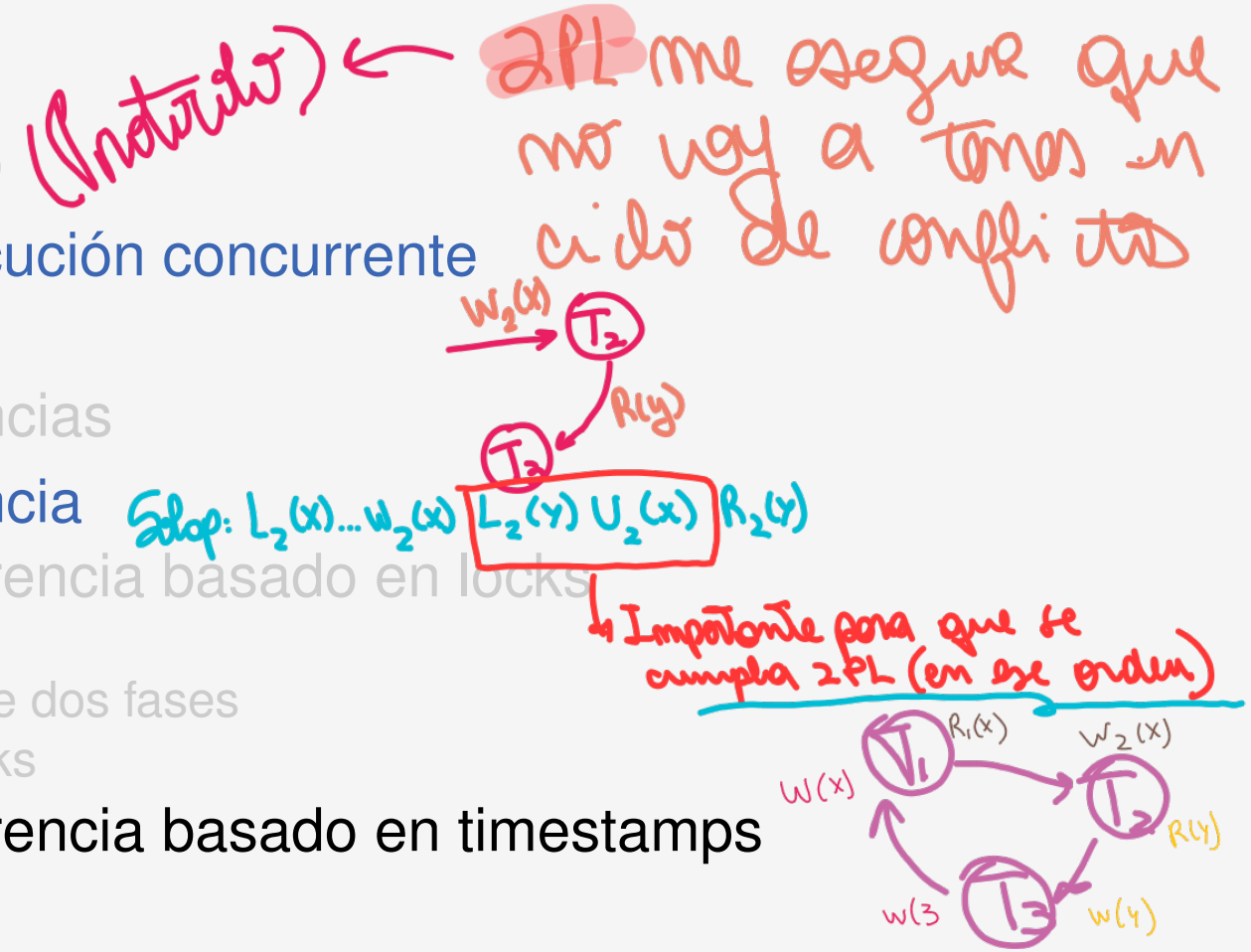
- Generalmente los SGBD's cuentan con estructuras de búsqueda de tipo árbol B+, tales que los bloques de datos se encuentran en las hojas.
- A los *locks* que se aplican sobre los nodos de un índice se los denomina index locks.
- Para mantener la serializabilidad en el acceso a estas estructuras, es necesario seguir las siguientes *reglas*:
  1. Todos los nodos accedidos deben ser lockeados.
  2. Cualquier nodo puede ser el primero en ser lockeado por la transacción (aunque generalmente es la raíz).
  3. Cada nodo subsecuente puede ser lockeado sólo si se posee un *lock* sobre su nodo padre.
  4. Los nodos pueden ser deslockeados en cualquier momento.
  5. Un nodo que fue deslockeado no puede volver a ser lockeado.

Control de concurrencia basado en locks

Protocolo del cangrejo

[SILB19 18.10.2]

- A partir de las reglas anteriores podemos proponer el siguiente protocolo para el acceso concurrente a estructuras de árbol, conocido como protocolo del cangrejo *(crabbing protocol)*:
  1. Comenzar obteniendo un *lock* sobre el nodo raíz.
  2. Hasta llegar a el/los nodo/s deseado/s, adquirir un *lock* sobre el/los hijo/s que se quiere acceder, y liberar el *lock* sobre el padre si los nodos hijo son *seguros* (es decir, el nodo hijo no está lleno si estamos haciendo una inserción, ni está justo por la mitad en el caso de una eliminación).
  3. Una vez terminada la operación, deslockear todos los nodos.



Control de concurrencia basado en timestamps

[ELM16 21.2; GM09 18.8.4; SILB19 18.5]

- Se asigna a cada transacción T_i un *timestamp* `TS(Ti)` (p.ej, un número de secuencia, o la fecha actual del reloj).
- Los *timestamps* deben ser únicos, y determinarán el orden serial respecto al cual el solapamiento deberá ser equivalente.
- Se permite la ocurrencia de conflictos, pero siempre que las transacciones de cada conflicto aparezcan de acuerdo al orden serial equivalente:

$$(W_{T_i}(X), R_{T_j}(X)) \rightarrow \texttt{TS(Ti)} < \texttt{TS(Tj)}$$

- Al no emplear *locks*, este método está exento de *deadlocks*.

Control de concurrencia basado en timestamps

Implementación

- Se debe mantener en todo instante, para cada ítem X, la siguiente información:
  - `read_TS(X)`: Es el `TS(T)` correspondiente a la transacción más joven –de mayor `TS(T)`– que leyó el ítem X.
  - `write_TS(X)`: Es el `TS(T)` correspondiente a la transacción más joven –de mayor `TS(T)`– que escribió el ítem X.
- Lógica de funcionamiento:
  1. Cuando una transacción T_i quiere ejecutar un $R(X)$:
     - Si una transacción posterior T_j modificó el ítem, T_i deberá ser abortada *(read too late)*. $W_{57}(x) \ldots R_{56}(x)$ (Aborto o sino actualizo el read ts
     - De lo contrario, actualiza `read_TS(X)` y lee.
  2. Cuando una transacción T_i quiere ejecutar un $W(X)$:
     - Si una transacción posterior T_j leyó ó escribió el ítem, T_i deberá ser abortada *(write too late)*. $R_{58}(x)$ $W_{56}(x)$ ← 8 → Aborto W_{56}
     - De lo contrario, actualiza `write_TS(X)` y escribe.

Aborto solo si el item no tiene el ordenn correcto

- Observación: La ejecución de los $R(X)$ y $W(X)$, y actualización de los `read_TS(X)` y `write_TS(X)` se centraliza en el *scheduler*.

Control de concurrencia basado en timestamps

Mejora: Thomas's Write Rule

- La lógica en el caso de ejecutar una escritura puede ser mejorada, utilizando la regla conocida como Thomas's Write Rule:

Thomas's Write Rule

Si cuando T_i intenta escribir un ítem encuentra que una transacción posterior T_j ya lo escribió, entonces T_i puede descartar su actualización sin riesgos, siempre y cuando el ítem no haya sido leído por ninguna transacción posterior a T_i.

- Entonces: Si `read_TS(X)` $\leq$ `TS(T_i)`, no hacemos ninguna escritura y no modificamos nada. Si en cambio `read_TS(X)` $>$ `TS(T_i)`, definitivamente debemos abortar T_i.
- Al utilizar esta mejora no queda garantizada la serializabilidad por conflictos, pero sí la serializabilidad por vistas.

Snapshot Isolation

[ELM16 21.4; SILB19 18.8]

- En el método de Snapshot Isolation, cada transacción ve una *snapshot* de la base de datos correspondiente al instante de su inicio.
  - Ventaja: Permite un mayor solapamiento, ya que lecturas que hubieran sido bloqueadas utilizando *locks*, ahora siempre pueden realizarse.
  - Desventajas:
    - Requiere de mayor espacio en disco o memoria, al tener que mantener múltiples versiones de los mismos ítems.
    - Cuando ocurren confictos de tipo WW entre transacciones, obliga a deshacer una de ellas.

Snapshot Isolation

Implementación

[ELM16 21.4; SILB19 18.8]

- Cuando dos transacciones intentan modificar un mismo ítem de datos, generalmente gana aquella que hace primero su *commit*, mientras que la otra deberá ser abortada *(first-committer-wins)*.
- Esto por sí solo no alcanza para garantizar la serializabilidad. Debe combinarse con dos elementos más:
  - Validación permanente con el grafo de precedencias buscando ciclos de conflictos RW.
  - *Locks de predicados* en el proceso de detección de conflictos, para evitar la anomalía del fantasma.

Ejercicio

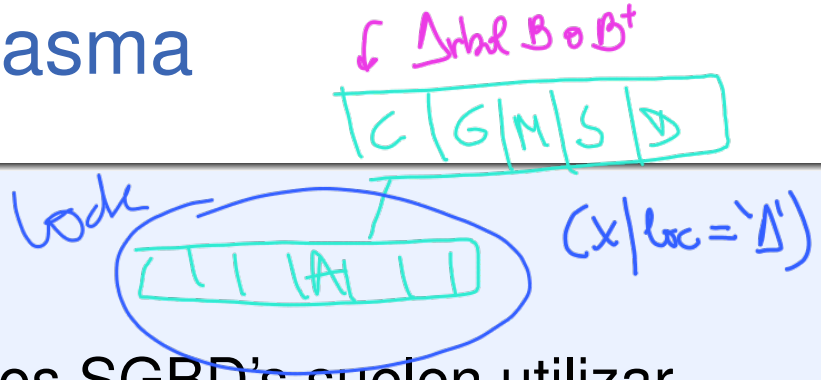
Proponer un solapamiento bajo *"Snapshot Isolation"* que no sea serializable.

Solución a la anomalía del fantasma

1. Bajo control de concurrencia basado en *locks*:
   - *Locks* de tablas: cuando se produce un $R(\{X|cond\})$, bloquear el acceso a toda la tabla a la que X pertenece. Es una solución casi trivial, y poco eficiente.
   - *Locks* de predicados: bloquear todas aquellas tuplas que podrían cumplir la condición, evitando incluso futuras inserciones de tuplas que también la cumplan.
2. Bajo *Snapshot Isolation*
   - *Locks* de predicados.
3. Bajo control de concurrencia basado en *timestamps*:
   - Utilizar índices de tipo árbol, y mantener registros `read_TS(I)` y `write_TS(I)` también para los nodos del árbol.

Solución a la anomalía del fantasma

Locks por rango y next-key locks



- Como mencionamos previamente, los SGBD's suelen utilizar estructuras de tipo árbol para indexar el acceso a los datos.
- La implementación común de los *locks de predicados* es el concepto de *range lock (lock por rango)* utilizando un índice de tipo árbol.
- Para tomar un *lock* por rango se suele tomar un *index lock* sobre todos los nodos del árbol que están dentro del rango, más un *lock* sobre el ítem inmediato posterior *(next-key lock)*.
- Esto evitará que puedan hacerse inserciones de nuevos ítems que cumplan con la condición (estén dentro del rango).

Recuperabilidad

Definición

[ELM16 20.4.2]

- La serializabilidad de las transacciones ya nos asegura la propiedad de *aislamiento*.
- Nos interesa ahora asegurar que una vez que una transacción commiteó, la misma no deba ser deshecha. Esto nos ayudará a implementar de una forma sencilla la propiedad de *durabilidad*.
- Definición: Un solapamiento es recuperable si y sólo si ninguna transacción T realiza el *commit* hasta tanto todas las transacciones que escribieron datos antes de que T los leyera hayan commiteado.

Recuperabilidad

Ejemplos

- Ejemplos:
  - $b_{T_1}; r_{T_1}(X); b_{T_2}; r_{T_2}(X); w_{T_1}(X); r_{T_1}(Y); w_{T_2}(X); c_{T_2}; w_{T_1}(Y); C_{T_1};$
  - ¿Es recuperable? $\rightarrow$ ✓
  - ¿Es serializable? $\rightarrow$ ✗
  - $b_{T_1}; r_{T_1}(X); b_{T_2}; w_{T_1}(X); r_{T_2}(X); r_{T_1}(Y); w_{T_2}(X); \ldots$
  - Si T_2 commitea ahora, ¿el solapamiento es recuperable? $\rightarrow$ ✗
  - (Si T_1 luego aborta, T_2 va a haber escrito en un forma persistente datos inválidos.)
- ¿Recuperable $\Rightarrow$ Serializable?
  - ✗ Falso
- ¿Serializable $\Rightarrow$ Recuperable?
  - ✗ Falso
  - Un solapamiento serializable podría tener un conflicto WR.

Recuperabilidad

El gestor de recuperación

- Dado un solapamiento recuperable, puede ser necesario deshacer (abortar) una transacción antes de llegado su *commit*, y para ello el SGBD deberá contar con una serie de información que es almacenada por su gestor de recuperación en un *log* (bitácora).
- El *log* almacena generalmente los siguientes registros:
  - (BEGIN, T_{id}): Indica que la transacción T_{id} comenzó.
  - (WRITE, T_{id}, X, x_{old}, x_{new}): Indica que la transacción T_{id} escribió el ítem X, cambiando su viejo valor x_{old} por un nuevo valor x_{new}.
  - (READ, T_{id}, X): Indica que la transacción T_{id} leyó el ítem X[2].
  - (COMMIT, T_{id}): Indica que la transacción T_{id} committeó.
  - (ABORT, T_{id}): Indica que la transacción T_{id} abortó.
- En particular, los valores viejos de cada ítem almacenados en los registros WRITE del *log* son los que permitirán deshacer los efectos de la transacción en el momento de hacer el *rollback*.

[2] En general no es necesario loguear los READ's, pero los incluímos por motivos didácticos.

Recuperabilidad

Rollback

- Un SGBD no debería jamás permitir la ejecución de un solapamiento que no sea recuperable.

Transacción T_1	Transacción T_2
begin	
	begin
leer_item(A)	
$A = A + 100$	
escribir_item(A)	
	leer_item(A)
	$A = A + 200$
	escribir_item(A)
	commit
abort	

- A pesar de que el solapamiento es serializable, el SGBD no podrá deshacer los efectos de la transacción T_1 sin eliminar también los efectos de T_2, que ya commiteó.
- Este tipo de solapamientos no debe producirse.

Recuperabilidad

Rollback

- ¿Cómo deshacemos los efectos de una transacción T_j que hay que abortar, sin afectar la serializabilidad de las transacciones restantes?
- Si las modificaciones hechas por T_j no fueron leídas por nadie, entonces basta con procesar el *log* de T_j en forma inversa para deshacer sus efectos *(rollback)*.
- Pero si una transacción T_i leyó un dato modificado por T_j, entonces será necesario hacer el *rollback* de T_i para volverla a ejecutar. ¡Sería conveniente que T_i no hubiera commiteado aún!
- Resultado: Si un solapamiento de transacciones es recuperable, entonces nunca será necesario deshacer transacciones que ya hayan commiteado.
  - Aún así puede ser necesario deshacer transacciones que no aún no han commiteado.
  - ¡Y puede que ésto produzca una cascada de rollbacks!

Recuperabilidad

Cascadas de rollbacks

- Que un solapamiento sea recuperable, no implica que no sea necesario tener que hacer *rollbacks* en cascada de transacciones que aún no commitearon.
- Ejemplo:
  - $b_{T_1}; r_{T_1}(X); b_{T_2}; w_{T_1}(X); r_{T_2}(X); r_{T_1}(Y); w_{T_2}(X); w_{T_1}(Y); C_{T_1}; c_{T_2};$
  - ¿Es recuperable? $\rightarrow$ ✓
  - ¿Es serializable? $\rightarrow$ ✓
  - Pero, ¿qué sucede si T_1 aborta después de $w_{T_2}(X)$?

Transacción T_1	Transacción T_2
begin	
leer_item(X)	
	begin
escribir_item(X)	
	leer_item(X)
leer_item(Y)	
	escribir_item(X)
escribir_item(Y)	
commit	
	commit

  - Como T_2 lee un ítem que T_1 había modificado, la lectura que hace T_2 ya no es válida. Entonces, T_2 deberá ser abortada en cascada.

Recuperabilidad

Cascadas de rollbacks

- Para evitar los *rollbacks* en cascada es necesario que una transacción no lea valores que aún no fueron commiteados. Esto es más fuerte que la condición de recuperabilidad.
- Esta definición implica que quedan prohibidos los conflictos de la forma $(W_{T_i}(X); R_{T_j}(X))$ sin que en el medio exista un *commit* c_{T_i}.
- Se evita entonces la anomalía de la lectura sucia.
- ¿Evita *rollbacks* en cascada $\Rightarrow$ Recuperable?
  - ✓ Verdadero
- ¿Evita *rollbacks* en cascada $\Rightarrow$ Serializable?
  - ✗ Falso

Recuperabilidad

Cascadas de rollbacks

- ¿Qué anomalía de las que vimos no cubre esta definición?
  - La actualización perdida.
- Ejemplo:

Transacción T_1	Transacción T_2
begin	
leer_item(X)	
	begin
	leer_item(X)
escribir_item(X)	
leer_item(Y)	
	escribir_item(X)
abort	

- ¿Es recuperable? $\rightarrow$ ✓
- ¿Evita *rollbacks* en cascada? $\rightarrow$ ✓
- ¿Es serializable? $\rightarrow$ ✗

Recuperabilidad

Protocolo de lock de dos fases estricto (S2PL)

- Los *locks* también pueden ayudar a asegurar la recuperabilidad.
- El protocolo de 2PL estricto (S2PL) emplea la siguiente regla:

> **Protocolo de lock de dos fases estricto (S2PL)**
>
> Una transacción no puede adquirir un *lock* luego de haber liberado un *lock* que había adquirido, y además los *locks* de escritura sólo pueden ser liberados después de haber commiteado la transacción.

- En caso de no diferenciar tipos de *lock*, se convierte en riguroso:

> **Protocolo de lock de dos fases riguroso (R2PL)**
>
> Los *locks* sólo pueden ser liberados después del commit.

- Resultado: S2PL y R2PL garantizan que todo solapamiento sea no sólo serializable, sino también recuperable, y que no se producirán cascadas de rollbacks al deshacer una transacción.

Recuperabilidad

Control de concurrencia basado en timestamps

- En el método de control de concurrencia basado en *timestamps*, cuando se aborta una transacción T_i, cualquier transacción que haya usado datos que T_i modificó debe ser abortada en cascada.
- Para garantizar la recuperabilidad se puede escoger entre varias opciones:
  (a) No hacer el *commit* de una transacción hasta que todas aquellas transacciones que modificaron datos que ella leyó hayan hecho su *commit*. Esto garantiza recuperabilidad.
  (b) Bloquear a la transacción lectora hasta tanto la escritora haya hecho su *commit*. Esto evita rollbacks en cascada.
  (c) Hacer todas las escrituras durante el commit, manteniendo una copia paralela de cada ítem para cada transacción. Para esto, la escritura de los ítems en el *commit* deberá estar centralizada y ser atómica.

Niveles de aislamiento

Estándar SQL

[ELM16 20.6]

- SQL permite definir el nivel de aislamiento de las transacciones con el comando **SET TRANSACTION ISOLATION LEVEL**.

```
SET TRANSACTION ISOLATION LEVEL
READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ | SERIALIZABLE;
```

- Y para definir una transacción:

```
START TRANSACTION [ISOLATION LEVEL ...]
.... (sql commands)
COMMIT | ROLLBACK
```

- Distintos motores pueden implementar los niveles de aislamiento con distintas técnicas, pero de acuerdo con el nivel de aislamiento configurado, el SGBD debe garantizar que ciertas anomalías no ocurran.

Niveles de aislamiento

Tabla de anomalías

- De acuerdo con el nivel de aislamiento elegido, pueden producirse ó no ciertas anomalías:
  1. *Read Uncommitted:* Es la carencia total de aislamiento: No se emplean *locks*, y se accede a los ítems sin tomar ninguna precaución.
  2. *Read Committed:* Evita la anomalía de lectura sucia.
  3. *Repeatable Read:* Evita la lectura no repetible y la lectura sucia.
  4. *Serializable:* Evita todas las anomalías, y asegura que el resultado de la ejecución de las transacciones es equivalente al de algún orden serial.

Nivel de aislamiento	Lectura sucia	Lectura no repetible	Fantasma
READ UNCOMMITTED	✗	✗	✗
READ COMMITTED	✓	✗	✗
REPEATABLE READ	✓	✓	✗
SERIALIZABLE	✓	✓	✓

Niveles de aislamiento

Aclaración

- Si bien la anomalía de “escritura sucia” *(dirty write)* no se menciona en el estándar, la misma debería ser proscripta en todos los niveles de aislamiento, y prácticamente todos los SGBD lo hacen. (Berenson et al. [3] definen un nivel de aislamiento ’0’ en que ni siquiera se evitan las escrituras sucias).

[3] *“A Critique of ANSI SQL Isolation Levels”*, Berenson et al., SIGMOD ’95

Niveles de aislamiento

Ejercicio

Ejercicio

Un SGBD implementa el siguiente protocolo de locks:
"Cada vez que una transacción va a leer un ítem, adquiere un lock de lectura sobre el mismo inmediatamente antes, y lo libera inmediatamente después. Cuando va a modificar un ítem, adquiere un lock de escritura inmediatamente antes, y sólo lo libera después de realizar el commit".
¿Qué nivel de aislamiento logra este protocolo?

Respuesta

Read Committed, dado que sólo evita la lectura sucia (y la escritura sucia).

Implementaciones

Oracle y MySQL

- El nivel de granularidad es la fila, para minimizar el bloqueo de recursos y maximizar la concurrencia.
- Para evitar inconsistencias de lectura se utiliza un control de concurrencia multiversión (MVCC):
  - Query's Snapshot Time: Cada *consulta* dentro una transacción ve los datos que estaban commiteados cuando t_1 en que comenzó la consulta. Si luego la fila resulta ser modificada por otra transacción que había comenzado antes de t_1, se hace *rollback*. Esto garantiza un nivel de aislamiento *Read Committed*.
  - Transaction's Snapshot Time: Todas las consultas de una transacción verán los valores de las filas commiteados antes de comenzar la transacción (ésto no llega a garantizar *Serializable*).

- MySQL bajo InnoDB utiliza control multiversión (MVCC). Para el nivel de aislamiento *Serializable* lo combina con *range locks* y *next-key lock*.

Implementaciones

DB2

- Implementa únicamente el control de concurrencia basado en *locks*, con un sistema bastante complejo.
- Pueden obtenerse a distintos niveles de granularidad, pero lo más común es la tabla ó la fila.
- Los tipos de *lock*, por nivel de control creciente, son: Intent None (IN), Intent Share (IS), Next Key Share (NS), Share(S), Intent Exclusive (IX), Share with Intent Exclusive (SIX), Update(U), Next Key Exclusive (NX), Next Key Weak Exclusive (NW), Exclusive (X), Weak Exclusive (WE), Super Exclusive (Z).
- El funcionamiento básico es:
  - Los *Shared locks* sólo permiten leer el ítem.
  - Los *Update locks* permiten leer, pero con aviso de que luego se intentará actualizar el ítem.
  - Los *Exclusive locks* permiten actualizar el ítem.

Implementaciones

DB2

- Por ejemplo...
  - Mientras T_1 posee un *update lock*, T_2 puede pedir un *share lock*.
  - Mientras T_1 posee un *share lock*, T_2 puede pedir un *update lock*.
  - Mientras T_1 posee un *update lock*, T_2 no puede pedir un *update lock*.
  - Mientras T_1 posee un *exclusive lock*, T_2 no puede pedir siquiera un *share lock*.
- Se utiliza una técnica de *Lock Scalation* para convertir *locks* de fila en un *lock* de tabla.
- Se pueden elegir 4 niveles de aislamiento distintos.

Implementaciones

MS-SQL Server y PostgreSQL

- El control de concurrencia de MS-SQL Server está basado en *locks* (granularidad variable), aunque también implementa un mecanismo de control multiversión (granularidad de filas).
- Al igual que en DB2, en SQL Server hay una gran cantidad de tipos de *locks*.
- Se implementan 5 niveles de aislamiento. El parámetro `ALLOW_SNAPSHOT_ISOLATION` permite, en algunos de ellos, elegir entre un control basado en *locks* y un control multiversión.

- PostgreSQL utiliza Snapshot Isolation combinado con la evaluación permanente del grafo de precedencias y *locks de predicados*, para garantizar un nivel de aislamiento *Serializable*.

Bibliografía

[ELM16] Fundamentals of Database Systems, 7th Edition.
R. Elmasri, S. Navathe, 2016.
Capítulo 20, Capítulo 21

[GM09] Database Systems, The Complete Book, 2nd Edition.
H. García-Molina, J. Ullman, J. Widom, 2009.
Capítulo 18, Capítulo 19

[SILB19] Database System Concepts, 7th Edition.
A. Silberschatz, H. Korth, S. Sudarshan, 2019.
Capítulo 17, Capítulo 18

[CONN15] Database Systems, a Practical Approach to Design, Implementation and Management, 6th Edition.
T. Connolly, C. Begg, 2015.
Capítulo 22