Concurrencia y Transacciones

Vamos a empezar a ver cómo funciona internamente una Base de Datos.

Concurrencia: cuando hay mas de un usuario usando la base al mismo tiempo

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: 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.

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

Una misma trx 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.

> 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:

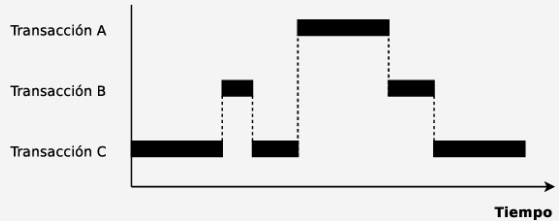
- ✓ Si dos transacciones utilizan sets de datos distintos deberían poder ejecutarse concurrentemente en distintos procesadores.

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

Vamos a usar el modelo 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.



Si tuviéramos múltiples unidades de procesamiento, el modelo a utilizar seria el 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
$PAT \leftarrow \sigma_{nombre='LaPaternal'}(Sucursales)$ $ID\_PAT \leftarrow \pi_{codigo}(PAT)$	
	$UN\_CLI \leftarrow \sigma_{CUIT='27\text{-}40182490\text{-}5'}(Clientes)$ $Clientes \leftarrow \sigma_{CUIT\neq'27\text{-}40182490\text{-}5'}(Clientes)$
$CLI \leftarrow ID\_PAT \bowtie_{codigo=cod\_sucursal} Clientes$	
	$Clientes \leftarrow Clientes \cup \{'27\text{-}40182490\text{-}5', 'Juana\ Hass', 220, 2500\}$
$\pi_{nombre}(CLI)$	

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.

Modelo de datos: 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.

Una vez que elegimos el item con el que vamos a trabajar, vamos a aplicar alguna de las siguientes **instrucciones atómicas** básicas de una transacción sobre la base de datos:

- ✓ 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.

- 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.

Nos vamos a abstraer únicamente a analizar que pasa con la lectura/escritura de una trx nada más.

TRANSACCIONES

Una transacción es una unidad lógica de trabajo en los SGBD. Es una secuencia ordenada de instrucciones atómicas.

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.

Ejemplos:

- ✓ Una transferencia de dinero de una cuenta corriente bancaria a otra.
- ✓ La reserva de un pasaje aéreo.

La ejecución de transacciones debería cumplir 4 propiedades **deseables**, conocidas como PROPIEDADES ACID

1. **ATOMICIDAD:** las trx se ejecuta de manera atómica cuando se realiza por completo, o bien se ejecutan todas las transacciones o no se realizan
2. **CONSISTENCIA:** cada ejecución por si 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.
   Ejemplo: la bdd de una empresa puede tener como restricción que no puede haber mas de un gerente por dpto
3. **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.
4. **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. Si yo le digo al usuario se transfirió de Pablo a Pedro, no puede suceder que eso después no haya quedado grabado en disco.

Recuperación

Para garantizar las propiedades ACID, se disponen de mecanismos de recuperación que permiten deshacer/rehacer una transacción en caso de que se produzca un error/falla. Se debe garantizar el a todo o nada.

- 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 DE EJECUCION CONCURRENTE

Con la concurrencia hay varias anomalías que pueden violar las propiedades ACID.

Lectura sucia (dirty read): se da cuando una transacción T2 lee un item que ha sido modificado por otra transacción T1.

PROBLEMA: si T2 lee algo que modifico T1, y después T1 se deshace (da abort), la lectura que hizo T2 no es válida en el sentido de que la ejecución resultante puede no ser equivalente a una ejecución seria de las transacciones. Osea, ***leímos un dato que al final no quedo en la base y actuamos en base a ese dato***.

- Es un conflicto de tipo WR: $W_{T_1}(X)...R_{T_2}(X)...(a_{T_1}$ ó $c_{T_1})$.

Escritura… Lectura antes de que una aborte o comitee.

Ejemplo de lectura sucia

- 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.

Cuando T1 aborta, A vuelve a tener 1100, y B con 990. Si solo se hubiera ejecutado T2, A tendría 1200 y B 900, pero no quedo así… ahí tenemos un problema

Actualización perdida (lost update) y lectura no repetible: ocurre cuando una transacción modifica un ítem que fue leído anteriormente por una primera transacción que aún no terminó.

Ejemplo: Dos personas me quieren hacer una transferencia. Los dos leen mi saldo, el primero me lo escribe haciendo mi salgo + \$100, el segundo hace mi saldo + \$100, y al final me quedo con solo 100 más, no 200.

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 $c_{T_1})$.

Ejemplo de actualización perdida y lectura no repetible

- 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.

Por más que se tenga un solapamiento recuperable, puede suceder una lectura no repetible

Escritura sucia (dirty write): ocurre cuando una transacción T2 escribe un ítem que ya había sido escrito por otra transacción T1 que luego se deshace.

El problema se dará si los mecanismos de recuperación vuelven al ítem a su valor inicial, deshaciendo la modificación realizada por T2.

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 $c_{T_1})$.

Fantasma (phantom): se produce cuando una transacción T1 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 T2.

Básicamente: estoy trabajando sobre datos y de la nada aparece un dato que no estaba

Si esta modificación se hace mientras T1 aún se está ejecutando, T1 podría encontrarse con que el conjunto de ítems que cumplen la condición cambió.

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.

Esta anomalía atenta contra la serializabilidad. Para resolverla, suelen necesitarse *locks* a nivel de tabla, o bien *locks* de predicados

Ejemplo de problema del fantasma

- Consideremos una única cuenta A con un saldo de \$600. Una transacción T_1 calcula la cantidad de cuentas con saldo mayor a \$500 y cuando termina cuenta aquellas con saldo igual ó superior a \$1000. Mientras tanto, otra transacción crea una nueva cuenta B con saldo de \$12000.

Transacción T_1	Transacción T_2
begin	
	begin
leer_item(A)	
mayores_500=1	
	B=12000
	escribir_item(B)
leer_item(A)	
leer_item(B)	
mayores_1000=2	
	commit
commit	

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

SERIALIZABILIDAD

Si una ejecución solapada de transacciones es igual a si se hubieran ejecutado de manera secuencial.

- 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:
  - $T_1 : b_{T_1}; R_{T_1}(X); R_{T_1}(Y); W_{T_1}(Y); c_{T_1};$
  - $T_2 : b_{T_2}; R_{T_2}(X); W_{T_2}(X); c_{T_2};$

Vamos a concentrarnos únicamente lo que sucede en disco, no en las grabaciones en memoria.

Solapamiento

Un solapamiento entre dos transacciones T1 y T2 es una lista de *m(T1) +m(T2)* instrucciones, en donde cada instrucción de T1 y T2 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:
  - $b_{T_1}; R_{T_1}(X); b_{T_2}; R_{T_2}(X); W_{T_2}(X); R_{T_1}(Y); W_{T_1}(Y); c_{T_2}; c_{T_1};$

Importante: Cuando miro solo las de T1, tienen que estar en el orden de T1, y cuando miro las T2 tienen que estar en su orden también dentro de la misma transacción.

Lo que nos vamos a preguntar es si dicho solapamiento es serializable o no.

Ejecución Serial

Dado un conjunto de transacciones T1,T2,...,Tn una ejecución serial es aquella en que las transacciones se ejecutan por completo una detrás de otra, en base a algún orden T_{i1} ,T_{i2} ,...,T_{in} .

Osea sin nada en el medio. Acá solo hay dos posibilidades. Hago todo lo de T1 primero, y luego todo lo de T2 o viceversa.

- 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:
  - $b_{T_1}; R_{T_1}(X); R_{T_1}(Y); W_{T_1}(Y); c_{T_1}; b_{T_2}; R_{T_2}(X); W_{T_2}(X); c_{T_2};$
  - $b_{T_2}; R_{T_2}(X); W_{T_2}(X); c_{T_2}; b_{T_1}; R_{T_1}(X); R_{T_1}(Y); W_{T_1}(Y); c_{T_1};$

Decimos que un solapamiento de un conjunto de transacciones T1,T2,..., Tn 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 T1,T2,...,Tn. Ahora.. ¿Qué significa que tengan estado equivalente?

Nos interesa que los solapamientos producidos sean serializables, porque ellos garantizan la propiedad de aislamiento de las transacciones.

Para evaluar esa 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.

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 RTi (X) lee el valor escrito por la misma transacción j, WTj (X). Además, se pide que en ambos órdenes la última modificación de cada ítem X haya sido hecha por la misma transacción.

Equivalencia de Conflictos

Dado un orden de ejecución, un conflicto es un par de instrucciones (I1,I2) ejecutadas por dos transacciones distintas Ti y Tj, tales que I2 se encuentra más tarde que I1 en el orden, y que responde a alguno de los siguientes esquemas:

- ✓ ($R_I(X)$, $W_J(X)$): Una transacción escribe un ítem que otra leyó.
- ✓ ($W_I(X)$, $R_J(X)$): Una transacción lee un ítem que otra escribió
- ✓ ($W_I(X)$,$W_J(X)$): Dos transacciones escriben un mismo ítem.

Osea: **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.**

Entonces NO hay conflicto cuando dos transacciones distintas están leyendo el mismo item o cuando dos transacciones distintas trabajan sobre ítems distintos.

Todo par de instrucciones consecutivas (I1,I2) de un solapamiento que **no constituye un conflicto puede ser invertido en su ejecución** (es decir, reemplazado por el par (I2,I1)) obteniendo un solapamiento equivalente por conflictos al inicial.

Ejemplo: si dos leen el mismo item es lo mismo cual lee primero y cual lee después. Ahora cuando hay conflicto no puedo invertir las transacciones porque ya no es lo mismo cual pasa primero

Ejemplo

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.

Como es lo mismo a ejecutar primero todo T1 y después todo T2, este solapamiento es seriabilizable.

Grafo de precedencias

La seriabilidad por conflictos puede ser evaluada a través del grafo de precedencias.

Es para evaluar esto que vimos en el ejemplo pero de manera mas sencilla.

Dado un conjunto de transacciones T1,T2,...,Tn que acceden a determinados ítems X1,X2,...,Xp, el grafo de precedencias es un grafo dirigido simple que se construye de la siguiente forma:

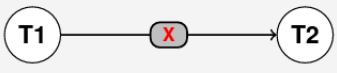
1. Se crea un nodo por cada transacción T1,T2,...,Tn.
2. Se agrega un arco entre los nodos Ti y Tj (con i != j) si y sólo si existe algún conflicto de la forma (RTi (Xk),WTj (Xk)), (WTi (Xk),RTj (Xk)) ó (WTi (Xk),WTj (Xk)).

Hacemos un grafo con tantos nodos como transacciones haya, y para cada conflicto que encontremos, hacemos un arco dirigido desde la Transacción que está a la izquierda del conflicto, hacia la transacción que está a la derecha. Básicamente lo que dice es que la trx de la izquierda debe venir antes que la otra.

Cada arco (Ti,Tj) en el grafo representa una **precedencia** entre Ti y Tj, e indica que para que el resultado sea equivalente por conflictos a una ejecución serial, entonces en dicha ejecución serial Ti debe Tj preceder a (es decir, “ejecutarse antes que”) Tj.

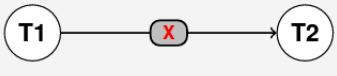
- Opcionalmente podemos etiquetar el arco con el nombre del recurso que causa el conflicto.
- Conflicto RW:

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



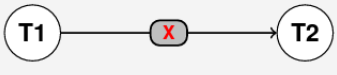
- Conflicto WR:

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



- Conflicto WW:

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

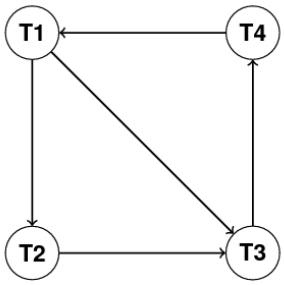


Cuando nos dan la preferencia, voy a **buscar todas las operaciones Write,** y ver todos los read y write previos y posteriores a ella para ver si hay conflictos con ítems compartidos con otras transacciones. Voy a listar los conflictos y dibujar un arco para cada uno. El arco siempre va desde lo que sucede primero hasta lo que sucede segundo.

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)			

Solución:



Una orden de ejecución es serializable por conflictos si y solo si su grafo de precedencias no tiene ciclos

Para ir buscando los conflictos vamos a ver **que ítems tienen escritura.** En este ejemplo arrancamos con X. T1 lee X y T2 quiere escribir X, entonces T1 debe venir antes que T2 por el conflicto X.

Ahora por el item Y: tenemos dos escrituras, en T1 y T2. La Escritura de T1 tiene que venir antes que la lectura de Y, y T1 tiene que ir antes que T3 porque dos escrituras son conflicto tmb. Y la T2 debe ir antes que T3 porque lee Y.

Para Z tenemos la lectura en T4 y escritura en T1. Entonces, T4 debe venir antes que T1.

Luego en W en T3 tengo lectura, debe venir antes que T4

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.

Osea: busco nodos a los que no le lleguen flechas y los vos descartando. El orden en que los descarte es el orden en que ejecuto.0

Control de Concurrencia

Para lograr el objetivo de que se cumpla el aislamiento hay dos enfoques:

- Optimista: Deja hacer y luego valida; si falló, hace rollback
- Pesimista: busca garantizar que no se produzcan conflictos. Esta la técnica de control basada en locks y la basada en timestamps. La mas usada en los SGBD es la de locks.

CONTROL BASADO EN LOCKS

El SGBD utiliza ***locks*** para bloquear a los recursos (los ítems) y no permitir que mas 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.

LOCKS: son variables asociadas a determinados recursos, y que permiten regular el acceso a los mismos en los sistemas concurrentes. Sirven para resolver el problema de EXCLUSION MUTUA.

Un lock debe disponer de dos ***primitivas*** de uso, que permiten tomar y liberar el recurso X asociado al mismo

- ✓ **Acquire(X) o Lock(X):** (L(X))
- ✓ **Reléase(X) o 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***.

Lock(X) requiere escribir y leer una variable, por lo tanto no puede estar solapada con una ejecución similar en otra transacción.

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.

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

Tipos de locks

- ✓ **Lock de escritura** (o lock de acceso exclusivo)**:: $L_{ex}(X)$**

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.

- ✓ **Lock de lectura** (o lock de acceso compartido)**: $L_{SH}(X)$**

Muchas trx pueden poseer locks de acceso compartido sobre un mismo item simultáneamente.

		↓ Se puede otorgar otro de tipo	
		SH	EX
Si alguien tiene un lock →	SH	✓	✗
	EX	✗	✗

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 (aun tomando un lock), podría producirse la anomalía de la lectura no repetible.

Entonces, **hace falta un protocolo que nos garantice la seriabilidad**

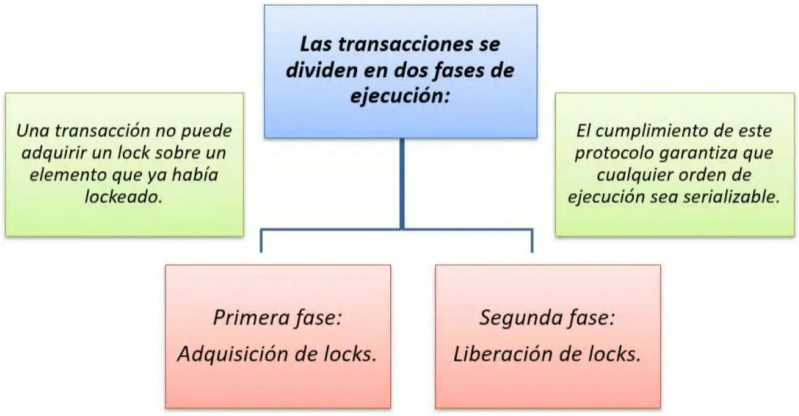
Protocolo de lock de dos fases (2PL – two phase lock)

El protoloco más comúnmente utilizado para la adquisición y liberación de locks es el protocolo de lock de dos fases (2PL, two-phase lock).

> **Protocolo de lock de dos fases (2PL)**
>
> Una transacción no puede adquirir un *lock* luego de haber liberado un *lock* que había adquirido.

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.



En el momento que libero un lock, NO puedo volver a adquirir locks, por eso hay dos fases.

Este protocolo garantiza que una ejecución va a ser serializable

Ejemplo de un 2PL básico

T1	T2	A	B
l(A); r(A)		25	25
A = A +100			
w(A); l(B); u(A)		125	
	l(A); r(A)		
	A = A * 2		
	w(A)	250	
	l(B) **DENIED**		
r(B); B + B + 100			
w(B); u(B)			125
	l(B); u(A); r(B)		
	B = B * 2		
	w(B); u(B)		250

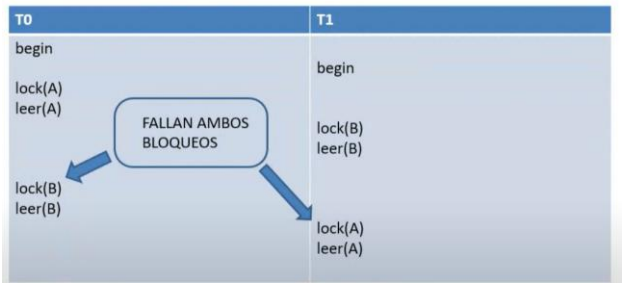
El cumplimiento de este protocolo es condición suficiente para garantizar que cualquier orden de ejecución de un conjunto de transacciones sea serializable, porque si va a escribir algo, recién al final libera todo y hasta que no libere otro no puede leer. Sin embargo, la utilización de locks introduce otros **dos problemas** potenciales que antes no teníamos:

- ✓ Bloqueo **(deadlock)**
- ✓ Inanición o postergación indefinida **(livelock)**

DEADLOCKS

Es una condición en que un conjunto de transacciones queda cada una de ellas bloqueada a la espera de recursos que otra de ellas posee.

Ejemplo de deadlock:



El problema no habría existido si la transaccione si hubieran adquirido los lock en el mismo orden.

¿La solución? Abortar alguna de las transacciones para desbloquear.

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(X1,X2,...Xn)).
2. Definir un ordenamiento de los recursos, y obligar a que luego todas las transacciones respeten dicho ordenamiento en la adquisición de locks. Si yo digo el item A viene antes que el B, se lockea el A antes que B, pero esto tampoco es practico porque necesito saber dicho orden.
3. Métodos basados en timestamps.

Protocolos de prevención de deadlocks

2PL conservador: *que cada transacción **bloquee con antelación** todos los elementos que necesita. Si alguno no puede bloquearlo, espere y reintenta. Limita la concurrencia.*

***Ordena todos los elementos** de la base y se asegura de que una transacción que necesite varios elementos los bloqueará en ese orden. Obliga a conocer este orden.*

La limitación del enfoque preventivo es que será necesario saber que recursos serán necesarios de antemano.

En vez del enfoque de prevención, vamos a usar un enfoque de detección.

Métodos de detección de deadlocks

1. Grafo de alocación de recursos:

   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 adquirir el Lock(X) después del cual se aborta la transacción

inanición

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 elige 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.

Rollback en cascada

Aunque tengamos un solapamiento serializable de transacciones, si una transacción Ti es abortada, el SGBD debe mantener la consistencia de la base. Si las modificaciones hechas por Ti fueron leídas por otras transacciones, entonces será necesario hacer el rollback de todas las transacciones involucradas en **una cascada de transacciones.**

Necesitamos nuevos protocolos:

Protocolo de lock de dos fases ESTRIICTURO Y RIGUROSO

Estricto S2PL: *Una transacción no puede adquirir un lock luego de haber liberado uno que había adquirido, y los locks de escritura sólo pueden ser liberados después de haber commiteado la transacción.*

Riguroso R2PL: *no diferencia los tipos de locks. Los locks sólo pueden ser liberados después del commit.*

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

Pero.. no evitan los interbloqueos, por lo que se combinan con 2PL conservador o por marcas de tiempo.

RESUMEN DE PROTOCOLOS:

2PL básico: *Tenemos dos fases, en la primera adquirimos locks, en la segunda los liberamos.*

2PL Conservador: *Bloquea con antelación todos los recursos.*

Estricto S2PL: *Una transacción no puede adquirir un lock luego de haber liberado uno que había adquirido, y los locks de escritura sólo pueden ser liberados después de haber commiteado la transacción.*

Riguroso R2PL: *no diferencia los tipos de locks. Los locks sólo pueden ser liberados después del commit.*

CONTROL BASADO EN TIMESTAMPS

Se asigna a cada transacción Ti un *timestamp* TS(T_i). por ejemplo, un numero 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. Yo no busco cualquier equivalencia posible, sino la equivalencia serial que fuera en el orden en que vinieron las transacción.

Por lo tanto a veces yo voy a estar imposibilitando cosas que hubieran sido seriabilizables, pero en otro orden en que venían las transacciones. ***Gano que no tengo lockeos, pero pierdo en que a veces por ahí voy a abortar una transacción que no era abortable,*** porque se podía seriabilizar en otra forma.

Básicamente la transacción aborta en el momento en que quiere hacer algo que no esta en el orden en que vinieron las transacciones.

Se permite la ocurrencia de conflictos, pero siempre que las transacciones de cada conflicto aparezcan de acuerdo al orden serial equivalente:

$$(W_{T_i}(X), R_{T_j}(X)) \rightarrow \mathrm{TS}(T_i) < \mathrm{TS}(T_j)$$

Al no usar locks, este método esta exento de deadlocks.

Implementación

Se debe mantener, en todo isntante, ***para cada item X,*** la siguiente info:

- `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*.

Son dos variables de cuando fue leído por la transacción mas joven, y cuando fue escrito por la transacción mas joven. Las vamos a ir actualizando, pero también vamos a ir chequeándolas antes de ver si podemos trabajar con el item o no.

- 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)*.
     - 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)*.
     - De lo contrario, actualiza `write_TS(X)` y escribe.

CONTROL DE CONCURRENCIA SNAPSHOT ISOLATION

En el método de Snapshot Isolation, ***cada transacción ve una snapshot de la base de datos correspondiente al instante de su inicio***. Es como si le sacara una foto a la base cuando arranca y ve: esos datos + lo que el vaya modificando. No ve nada de lo que ocurre en la bdd que no haya afectado el. Si toda transacción viene y toca cosas, yo no tengo que enterarme de nada de lo que hizo esa transacción.

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).** El conflicto está cuando dos quieren escribir.

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 detectar precedencias.

GRADOS DE AISLAMIENTO EN SQL

Nivel de aislamiento	Lectura sucia	Lectura no repetible	Lectura fantasma
READ UNCOMMITTED	SI	SI	SI
READ COMMITTED	--	SI	SI
REPEATEABLE READ	--	--	SI
SERIALIZABLE	--	--	--

Read Uncommitted: Permite hacer lecturas sucias (dirty reads), donde las consultas dentro de una transacción son afectadas por cambios no confirmados (not commited) de otras transacciones. Esta opción es apenas transaccional, es como no tener transacciones.

Read committed (non-repeatable reads): Los cambios confirmados son visibles dentro de otra transacción, esto significa que dos consultas dentro de una misma transacción pueden retornar diferentes resultados. Generalmente este es el comportamiento por defecto en los SGBD.

Repeatable reads (phanton reads): Dentro de una transacción todas las lecturas son consistentes. En este nivel de aislamiento, el SGBD implementa el control de concurrencia basado en bloqueos, mantiene los bloqueos de lectura y escritura -de los datos seleccionados- hasta el final de la transacción. Sin embargo, no se gestionan los bloqueos de rango, por lo que las lecturas fantasmas pueden ocurrir

Serializable: No se permiten actualizaciones en otras transacciones si una transacción ha realizado una consulta sobre ciertos datos. En este caso las distintas transacciones no se afectan entre sí. Las transacciones están completamente aisladas entre sí, lo que conlleva un costo asociado. ¡¡Que puede hasta detener todos los procesos!!