irenchuchu
www.wuolah.com/student/irenchuchu

Resumen-Tema-5.pdf

Apuntes Examen Final (Parcial 2)

3° Administración de Bases de Datos

Grado en Ingeniería Informática

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación
Universidad de Granada

TEMA 5: Gestión y control de concurrencia.

Introducción:

- Ejecución concurrente de transacciones para optimizar la eficiencia.
- Controlador de concurrencia gestiona el uso de recursos.
- Hay que garantizar la consistencia de la BD.
- Siempre se ha distinguido entre operación de lectura y operación de escritura:
    - La operación de lectura, como no tiene un efecto secundario, no provoca nada, no es problemática.
    - La operación de escritura: puede llegar a ser problemática, aunque no siempre, eso viene a ser porque el acceso al mismo objeto por parte de más de una transacción.
      Dos usuarios no podrían acceder a la misma tabla, ni podrían hacer un "insert" a la vez ya que surgirían problemas debido a que no se sabría qué "insert" iría primero.
      Para ello están los algoritmos de control de la concurrencia.
- La concurrencia lo que intenta es mejorar la eficiencia, que un montón de usuarios tengan la "ilusión" de que todos están trabajando a la vez cuando lo que el sistema está haciendo es saltar de uno a otro constantemente.
- Habitualmente, en este tipo de situaciones lo que se suele tener es un "controlador" de la concurrencia que lo que hace es organizar las cosas, incluso hacer cambios de contexto.
- En cualquier caso, hay que garantizar lo que conocemos como consistencia de la base de datos. Es algo que cuando el sistema se cae y se vuelve a levantar hay que incorporar de nuevo todo lo de la tabla. Lo que se persigue es la fiabilidad de los datos, y la fiabilidad lo que garantiza es la consistencia. Los datos cumplen unas reglas semánticas asociadas a la BD, que almacena esos datos.

Problemas producidos por la concurrencia:

- Una de las formas de garantizar esta concurrencia es la anulación de la transacción.
- Anulación (**aborto**) de transacción: cuando una transacción quiere acceder a un dato que está siendo modificado por otra o cuando es eliminando por otra.
- Tenemos dos transacciones que están intercaladas en el tiempo, pero no aparecen instrucciones en la misma línea significa que son dos transacciones que ocurren concurrentemente, es decir, que son una sola cola de sentencias. Es esa "ilusión" que parece que se hace seguido, pero en realidad primero se hacen las de T1, luego T2, luego T1.

T1	T2
lee(X,x_i)	
	lee(X,x_i)
	x_i:=x_i+1
	escribe(X,x_i)
x_i:=x_i*3	
escribe(X,x_i)	

- El programa que se ejecuta es uno formado por todas las transacciones, lo que se llama una ejecución concurrente.
- Los algoritmos más antiguos lo que hacían es:
    - La primera transacción como vemos hace algo sobre "x", una de las opciones era anulación de la transacción.
    - Automáticamente la transacción T2 por intentar acceder a "x" se anula, aborta. Significa que no va a poder ejecutarse hasta que la T1 termine completamente. Se llama **serialización**.
    - Lo que es concurrente lo convertimos en algo en serie, uno detrás de otro, la forzamos.
    - El resultado de ejecutar primero T1 y luego T2 y el resultado de ejecutar T2 y luego T1 no es para nada el mismo, ya que las operaciones que se están haciendo (multiplicar por tres y sumar uno) no son conmutativas.

- **Estado inconsistente de la BD:**
    - Transacciones concurrentes que violan temporalmente las restricciones de la BD.
    - El invariante de representación, se cumple siempre y cuando no se esté ejecutando una operación sobre el dato, es decir, antes de una operación debe cumplirse el invariante y después también, pero durante la operación no tiene porqué.
    - La consistencia de la BD se garantiza antes y después de la transacción, pero, cuando tienes varias transacciones entrelazadas ya se tiene un problema porque eso amplia el estado de inconsistencia de la BD, hasta que todo el conjunto de transacciones concurrentes han terminado.
    - Supongamos que B es clave externa a A, luego no deben tener valores distintos en ningún momento.
    - **EJEMPLO BASTANTE COMÚN:** Imaginar que se modifica algún dato, haciendo un *update* de un dato que es el destino de una llave externa. De hecho, haces el *update* en cascada (modificación en cascada) de forma que si el dato que estas modificando tiene una llave externa se va a la otra tabla y modifica el dato en la otra tabla, pero si ese dato de la otra tabla tiene una llave externa se va a otra tabla, de forma que es una transacción tras transacción tras transacción… Pero hasta que no se hayan hecho todos los cambios, la consistencia no es válida, luego el problema es que no puede haber valores distintos. No se puede analizar la consistencia hasta que todos los valores no se hayan cambiado porque si garantizas la consistencia a mitad de dos transacciones concurrentes has cambiado el valor de una y no de la otra, luego una que tenía que estar apuntando a un valor que existe ya apunta a un valor que no existe. Luego hasta que no se haya terminado, no se puede decir que se cumpla la consistencia. Este tipo de transacciones puede ocasionar este tipo de errores.

T1	T2
lee(A,xi)	
xi:=xi+1	
escribe(A,xi)	
	lee(A,zi)
	zi := zi * 2
	escribe(A,zi)
	lee(B,zj)
	zj := zj * 2
	escribe(B,zj)
lee(B,xj)	
xj:=xj+1	
escribe(B,xj)	

- **Operaciones en conflicto:** Son operaciones que aparecen en distintas transacciones, además acceden al mismo dato y alguna de ellas modifica. Incluso cuando se evalúan las operaciones en conflicto ni siquiera se considera que una o dos transacciones que hagan solo lectura, provocan una operación en conflicto. Es decir, se podrían tener dos lee(x) y no hacer ningún escribe(x) y las transacciones no provocan un conflicto por x pero podrían hacerlo por cualquier otro dato. Lo importante es que alguna de ellas haga un escribe(dato).
    - Lee(dato) escribe (dato), escribe(dato) lee(dato) o escribe(dato) escribe (dato) son conflictivas.
    - Pertenecen a distintas transacciones,
    - Acceden al mismo dato,
    - Alguna de ellas ejecuta la orden escribe( ).

Ejecuciones concurrentes sin conflicto:

- Un SGBD ejecuta transacciones concurrentes, formadas por sentencias.
- Una transacción es un conjunto de operaciones que están formadas por instrucciones, instrucciones más básicas (tipos de registros, de dato etc), que lo que hacen es acudir a esa capacidad que tiene el SGBD de saber dónde está un dato sin tener que preocuparse por dicho dato.
- La independencia de las BD se aplica a un montón de medios.
- Lo que se hace es ejecutar muchas transacciones concurrentes, y hay bastantes transacciones a ejecutar y transacciones cada una de una sesión de un usuario o usuarios. Así que, se le llama **plan de ejecución** al orden preciso en el que se entrelazan las sentencias de esa transacción/transacciones concurrentes.
    - El plan se construye a medida que van sucediendo las transacciones.
    - No se puede esperar a que se haya ejecutado todo para decidir que hacer.
    - Hay que garantizar desde el principio, que no se van a violar ciertas condiciones para garantizar consistencia de la BD.
    - Al orden especifico de las transacciones, hay que controlar si ejecutan operaciones conflictivas o no.

- **Átomo:** Fragmento de información en la BD cuyo acceso concurrente debe controlarse.

- Problema de la granularidad (surge del átomo): ¿Qué es para ti un átomo? ¿Qué es lo que tienes que controlar o no permitir el acceso concurrente? ¿A que no debemos permitir el acceso concurrente?
- A menor tamaño, más difícil de controlar (hacen falta más recursos); a mayor tamaño, más transacciones tienen que esperar su turno con el átomo.
- El problema es, los mecanismos adicionales que necesitas para controlar/prohibir la concurrencia.
- **EJEMPLO:**

T1	T2
lee(A,x_i)	lee(A,z_i)
$x_i := x_i + 1$	$z_i := z_i * 2$
escribe(A,x_i)	escribe(A,z_i)
lee(B,x_j)	lee(B,z_j)
$x_j := x_j + 1$	$z_j := z_j * 2$
escribe(B,x_j)	escribe(B,z_j)

- Plan de ejecución 1:
  La cuestión es si el plan es correcto o no lo es.
  ¿Hay accesos simultáneos? T1 accede a A y termina de acceder, T2 lo mismo, T1 accede a B y termina de acceder a B, lo mismo que T2.
  Resultado final de A: (A + 1) * 2.
  Resultado final de B: (B + 1) * 2.
  Desde esa perspectiva decimos, que, al fin y al cabo, el plan es correcto, porque no ocasiona ningún problema extraño ya que uno lee y escribe y el otro lee y escribe y no se entremezclan ni se produce una operación conflictiva.
  ¿Qué pasaría al ejecutar primero T1 completo y luego T2 completo (sin entrelazarlas)? ¿Cuánto valdrían A y B sin entrelazar las transacciones?: El resultado sería el mismo, porque, al fin y al cabo, primero se toca a A y luego a B.

T1	T2
lee(A,x_i)	
$x_i := x_i + 1$	
escribe(A,x_i)	
	lee(A,z_i)
	$z_i := z_i * 2$
	escribe(A,z_i)
lee(B,x_j)	
$x_j := x_j + 1$	
escribe(B,x_j)	
	lee(B,z_j)
	$z_j := z_j * 2$
	escribe(B,z_j)

  Se dice que es serializable, significa, que, si al entrelazar este plan ejecutado como tal y luego se prueba a ejecutar primero la primera y luego la segunda, sale el mismo resultado.
  Esto es algo que sucede muy pocas veces, pero es lo que debemos tratar de buscar. Es exactamente lo que los algoritmos de control fuerzan.

T1	T2
	lee(A,z_i)
	$z_i := z_i * 2$
lee(A,x_i)	
$x_i := x_i + 1$	
	escribe(A,z_i)
	lee(B,z_j)
	$z_j := z_j * 2$
escribe(A,x_i)	
lee(B,x_j)	
$x_j := x_j + 1$	
escribe(B,x_j)	
	escribe(B,z_j)

- Plan de ejecución 2:

Siguiendo el mismo plan que antes, de que no hubiese dos transacciones haciendo lo mismo, a la vez sobre el mismo átomo, en este caso no ocurre eso.
La primera hace una lectura de A, pero su escritura, es decir, sus sentencias sobre A se entrelazan con las sentencias sobre A de la otra y eso no tiene buena pinta.
Valor final de A: A + 1
Valor final de B: B * 2
Hay una escritura detrás de una escritura así que se produce una sobreescritura. El último que escribe manda.
¿Qué orden se pondría?: Da lo mismo.
Cuando se hace la transacción T1 y T2 después, te sale lo mismo que el plan de ejecución y además, haces la T2 primero y luego T1, y sale lo mismo se dice que son serializable e intercambiable.
Pero en este caso, sale que el plan es incorrecto porque su solución ni siquiera coincide con ninguna de las dos serializaciones, ni con T1 delante de T2 ni viceversa.

- Si el resultado de la ejecución de un conjunto de transacciones concurrentes coincide con la ejecución secuencial de las transacciones, se dice que esa ejecución es **serializable**.
- El número de ejecuciones posibles crece con el número de transacciones y no se pueden explorar todas. Hay ciertas combinaciones que permiten determinar si la ejecución es correcta o no.
- Se usan los semáforos para cada átomo y garantizamos un acceso exclusivo para elementos concurrentes.

Operación:

- Para intentar detectar esas posibles situaciones que no deben darse hablamos de operaciones.
- Conjunto de sentencias que actúa sobre un átomo o variable concretos.
- El problema es el alcance de esas operaciones ya que puede ser más largo o más corto.
- Dos operaciones a que no modifican el átomo y que pertenecen a dos transacciones distintas pueden ejecutarse simultáneamente. Es decir, las operaciones de "solo lectura" pueden intercalarse.

Operación compatible:

- Oi y Oj son compatibles si toda ejecución simultánea de ambas da el mismo resultado que la ejecución secuencial de ambas en cualquier orden.
- Son compatibles si se pueden hacer una delante de la otra y viceversa, y el resultado es el mismo.

La operación va del "lee" al "lee" ya que "imprime" no se considera operación ya que no ocasiona ningún cambio.

O_1
lee(A,x_1)
$x_1 := x_1 + 1$
imprime(x_1)

O_2
lee(A,x_2)
$x_2 := x_2 * 2$
imprime(x_2)

➔ **Operación compatible.**

¿Compatibles o incompatibles?:
Si hago O1 y después O2, el resultado de A es (A + 1) * 2. Si hago O2 primero y luego O1, el resultado de A es (A*2) + 1. Y no valen lo mismo.
En este caso, por tanto, son dos operaciones incompatibles, el hecho de dar incompatibilidad viene provocado porque si ni poniendo una delante u otra delante sale lo mismo, si las intercalas te sale un resultado distinto a los otros dos.

O_1
lee(A,x_1)
$x_1 := x_1 + 1$
escribe(A,x_1)

O_2
lee(A,x_2)
$x_2 := x_2 * 2$
escribe(A,x_2)

➔ **Operación NO compatible.**

- Cuando las operaciones son compatibles se pueden separar. El algoritmo lo que hace es forzar la compatibilidad.

Operaciones permutables:

- Oi y Oj son permutables si la ejecución de Oj tras Oi da el mismo resultado que la ejecución de Oi tras Oj.
- Dos operaciones permutables las puedes poner en cualquier orden y se obtendrá el mismo resultado.
- Cuando las operaciones son permutables se pueden intercambiar. El algoritmo lo que hace es forzar la permutabilidad.

O_1
lee(A,x_1)
$x_1 := x_1 + 1$
escribe(A,x_1)

O_2
lee(A,x_2)
$x_2 := x_2 + 10$
escribe(A,x_2)

A + 1 + 10 es lo mismo que A +10 + 1. La operación que se está haciendo es conmutativa.

➔ **Operación permutable.**
➔ **Operación NO compatible**.

- Permutabilidad no garantiza compatibilidad ni viceversa.
- Los algoritmos para resolver los problemas de acceso o de mirar si una transacción puede terminar etc., lo que hacen es garantizar estas condiciones de una manera más forzosa o menos. Dejando que ocurran esas operaciones, compatibles o incompatibles, o forzando que no ocurran por si nos encontramos con operaciones no compatibles y para ello destruir una de las dos transacciones.
- Persiguen lo que se llaman convertir una serie de transacciones que en realidad no son concurrentes, en un único programa que se conoce como ejecución, y además convertirlo en una ejecución que se llama serializable, es decir, que podríamos haber ejecutado esas transacciones en una serie de transacciones que se ejecutan en serie, una detrás de otra.

Ejecuciones serializables:

- Para lograrlo, lo primero que se busca es hacer unas transformaciones sobre una ejecución, sobre el plan de sentencias entrelazadas.
- Si se tiene un plan totalmente completo, que entrelaza las sentencias de dos o más transacciones, se puede intentar buscar la forma de transformarlo. Y si mediante esas transformaciones se consigue lo que se llama una ejecución en serie, es decir, que una serie de entrelazamientos de las sentencias se puedan separar de forma que se conviertan en transacciones en secuencia, se llama un plan serializable.

- Estas transformaciones no se pueden hacer de cualquier manera, no se pueden transformar las operaciones porque si, si están entrelazadas entre sí a lo mejor no pueden ser transformables.
- Las dos operaciones que se pueden realizar sobre un plan o una ejecución entrelazada para transformarla, son:
    - Separación de operaciones compatibles: dadas dos operaciones entrelazadas de transacciones distintas y cambiarlas por una secuencia de operaciones que den el mismo resultado.
      Digamos que tenemos entrelazadas sentencias de dos operaciones que afectan al mismo átomo, si estas operaciones son compatibles entre sí automáticamente son separables, significa que se puede ir intercambiando las sentencias de las dos transacciones empujando una de ellas hacia arriba y otra de ellas hacia abajo hasta conseguir que las dos operaciones queden una detrás de otra. Al ser compatibles da igual que una se haga antes que la otra o la otra antes que la una.
    - Re-ordenación de operaciones compatibles: cambiar el orden de ejecución de operaciones permutables.
      Después de haber separado las operaciones compatibles también puedo darle la vuelta, sin alterar el orden de las transacciones. Hasta que al final consigo todas las operaciones de una en secuencia arriba y las de la otra en secuencia abajo.

- Teorema:
    - Una condición suficiente para una ejecución serializable es que pueda ser transformada por separación y permutación en una sucesión de transacciones. Es decir, si se puede conseguir separar primero las operaciones que son compatibles entre sí, y luego permutarlas y se consigue hacer con todas las operaciones del conjunto de transacciones de forma que todas las operaciones de la misma transacción se queden juntas, se consigue una serialización.
    - Persigue que los algoritmos con los que se trabaja, cumplen la función de aplicar el teorema de la serialización, es decir, forzar la serialización.
    - **EJEMPLO:**

T_1	T_2
lee(A,x_1)	
$x_1 := x_1 * 10$	
escribe(A,x_1)	
	lee(A,x_3)
lee(B,x_2)	
	$x_3:= x_3 * 5$
...	...

T_1	T_2
...	...
	escribe(A,x_3)
$x_2 := x_2 * 3$	
escribe(B,x_2)	
	lee(B,x_4)
	$x_4 := x_4 * 3$
	escribe(B,x_4)

La primera transacción de este ejemplo está en la primera línea de T1. Luego esta operación no tiene ningún problema ya que no hay nadie en medio usando u operando sobre ese mismo átomo.
La segunda operación la realizaría T2 sobre A, y la tercera T1 sobre B. Estas dos últimas operaciones se encuentran entrelazadas, asi que nos preguntamos ¿son compatibles o incompatibles? Como se ha dicho antes, dos operaciones sobre átomos distintos son siempre compatibles. Al ser compatibles, se podría intercambiar cualquier sentencia de una de las operaciones con las de la otra. Por ejemplo, se podrían empujar todas las sentencias de T1 hacia arriba y todas las sentencias de T2 hacia abajo, es decir, se podría separar las dos operaciones que son compatibles. Al final, la mejor opción es, ya que A se lee antes que B, mejor subir las sentencias de T2, y T1 bajarlas.

- La segunda parte del teorema dice que, si dos operaciones entre si son intercalables, se podrían intercambiar.

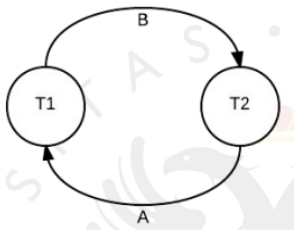
- Los algoritmos más bestias cuando detectan una operación incompatible con otra que ya está en curso, automáticamente matan a la transacción. "Matar" significa posponerla al futuro, tiene que esperar a que toda la transacción haya terminado.
- Los algoritmos más sutiles usan semáforos.

Grafo de precedencia:

- Hay algunos sistemas que lo que hacen es ir creando un grafo de precedencia, lo que se hace es que cuando se llega a un átomo, se indica la llegada y para los demás átomos se sabrá que hacer cuando lleguen.
- Nodos: transacciones (en cada nodo tiene una transacción, va añadiendo nuevos nodos al grafo).
- Arcos: restricciones en la ejecución. Restricción: si T1 tiene que actuar sobre A primero, T2 va después.
- Ti precede a Tj si, y sólo si, existen dos operaciones no permutables sobre el mismo átomo en las dos transacciones y la operación de Ti es anterior a la de Tj.
- Habrá un arco entre dos transacciones si una precede a otra, y se marca el arco con el nombre del átomo implicado.

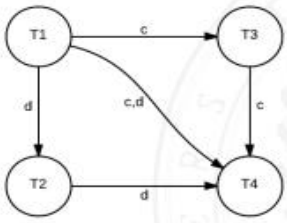
T1	T2
	lee(A,zi)
	zi := zi * 2
lee(A,xi)	
xi:=xi+1	
	escribe(A,zi)
escribe(A,xi)	
lee(B,xj)	
	lee(B,zj)
	zj := zj * 2
xj:=xj+1	
escribe(B,xj)	
	escribe(B,zj)

- Representamos las transacciones, el arco y en el arco el átomo que crea la restricción temporal entre las dos.
Entrelazado de sentencias: 1. Transacción de T2 sobre A.
2. Transacción de T1 sobre A. No son intercambiables.
En el grafo se muestra que: T1 está esperando a T2 por culpa de A, T2 no puede empezar porque necesita A.



- Teorema:
  - Condición suficiente para que una ejecución sea serializable es que el grafo de dependencias no presente ciclos dirigidos (que no se recorra un ciclo cerrado).
  - En cuanto haya ciclos, sería hacerlo serializable a la fuerza, "matando" a alguno.

- Un ejemplo:
  - lee(T_1,c), lee(T_2,b), lee(T_1,d), escribe(T_1,c), lee(T_3,c), escribe(T_3,c), lee(T_2,d), lee(T_3,a), lee(T_4,c), escribe(T_2,d), escribe(T_2,b), lee(T_4,d), escribe(T_4,d)



Algoritmos de control de concurrencia.

Los algoritmos garantizan secuencias serializables a lo burro. Sobre todo porque lo que hacen es deshacer aquellas situaciones que provocan conflicto, ordenando por marcas de tiempo, es decir, matando transacciones y deshaciendo las situaciones.

Además, tienen que garantizar que no ocurren bloqueos (interbloqueos).

Técnicas para garantizar el aislamiento de las transacciones:

- Permitir la ejecución y deshacer las que produzcan conflicto: técnicas de ordenación por marcas de tiempo.
- Evitar ciclos mediante esperas: técnicas de bloqueo.

Técnicas de ordenación por marcas de tiempo:

- Algoritmos que fuerzan la serialización en cuanto detectan una situación rara, trabajan con una única marca de transacción por transacción.
- Cada transacción recibe una marca de tiempo única cuando comienza.
- A cada átomo x que se accede, se asocia una referencia a la última transacción que opera sobre él: R(x) (controlador: especifica la transacción que ha accedido en último lugar, normalmente la más moderna).
- Algunos algoritmos de este tipo son:
  - Ordenación total.
  - Ordenación parcial (generaron el multi-versión, ya que hacia cosas raras).
  - Ordenación parcial multi-versión.
  - Control de concurrencia mediante validación de transacciones.

Algoritmo de ordenación total:

- Sean dos transacciones Ti y Tj con i y j marcas de tiempo tales que i<j, el algoritmo garantiza que Ti accede antes que Tj.
- Si una operación falla (ABORT), se deshace la transacción y se re-lanza más tarde, asignándole a la retrasada una referencia mayor que la de todas las actuales. Todas las operaciones hechas se deshacen y toda la transacción se pospone hasta que la transacción con la que tiene el conflicto se haya terminado.
- **Problema:** Garantiza o provoca que lecturas concurrentes provoquen abortar una transacción siendo solo una operación de lectura que no tendrían que ocasionar absolutamente nada. Una transacción más vieja accede a un átomo que ya ha accedido una transacción más moderna no puedes fiarte de lo que ha pasado. Garantiza la consistencia de forma un poco bruta.

```
Procedimiento lee (Ti, a);
   INICIO
      Si (R(a) <= i) entonces
        {Ejecutar la lectura}
        R(a) := i;
      si-no
        {ABORT: abortar la ejecucion}
      Fin-si
   FIN

Procedimiento escribe (Ti, a);
   INICIO
      Si (R(a) <= i) entonces
        {Ejecutar la escritura}
        R(a) := i;
      si-no
        {ABORT: abortar la ejecucion}
      Fin-si
   FIN
```

Hay dos operaciones que pueden provocar problemas (lectura y escritura). Aplicamos la comprobación de consistencia a ambos problemas.

Lectura/Escritura (no se distinguen entre estas operaciones): Si la última que accedió al átomo es anterior a mí, yo puedo acceder porque la otra ya habrá terminado y actualizo. Si una más moderna ya ha hecho algo, yo no puedo hacer nada así que tengo que morirme y posponerme a mí.

Lectura/escritura después de lectura: se muere. Si T2 lee después de T1 no hay problema, si T1 accede después de T2 se muere.

Algoritmo de ordenación parcial:

- Este algoritmo lo primero que hacer es incorporar un segundo controlador.
- Sólo se ordenan las parejas de operaciones lee/escribe, escribe/lee y escribe/escribe.
- Cada átomo x tiene dos referencias: una para determinar la última transferencia que lo leyó RR(x) y otra para la que lo actualizó WR(x) (se incorporan dos nuevos controladores).
- Lectura: si el ultimo que ha escrito es anterior a mi o yo mismo entonces puedo leer.
- Escritura: si el ultimo que ha leído es anterior a mi o yo mismo y el ultimo es anterior a mi o yo mismo entonces puedo escribir. Si alguien posterior a mí ha leído no puedo escribir porque ya ha leído así que tengo que hacer ABORT.
- Si una operación falla (ABORT), se deshace la transacción y se re-lanza más tarde, asignándole a la retrasada una referencia mayor que la de todas las actuales.

- Considera el hecho de que una transacción más antigua intente escribir después de que otra más moderna haya escrito es un error. Si se intenta escribir después de que alguien más moderno haya leído, error.
- La modificación del algoritmo parcial es considerar que hay una posibilidad en la que no debe dar error, entonces no hace nada, deja pasar la operación de escritura.

```
Procedimiento lee (Ti, a); {* escribe/lee *}
   INICIO
      Si (WR(a) <= i) entonces
        {Ejecutar la lectura}
        RR(a) := Max(RR(a),i);
      si-no
        {ABORT: abortar la ejecucion}
      Fin-si
   FIN
Procedimiento escribe (Ti, a); {* lee/escribe y escribe/escribe *}
   INICIO
      Si (RR(a) <= i) y (WR(a) <= i) entonces
        {Ejecutar la escritura}
        WR(a) := i;
      si-no
        {ABORT: abortar la ejecucion}
      Fin-si
   FIN
```

Si el último que ha escrito es anterior a mi yo puedo leer, pero a quien actualizo. Sin embargo, yo estoy haciendo una operación que es lectura así que tengo que actualizar el controlador ¿Cómo? Si uno más moderno ya ha leído se queda el actualizado, si yo soy más moderno entonces yo.

Si yo entiendo leer algo de una transacción que empezó después de la mía, tengo que dejarlo, y me aborto y me ejecuto en el futuro ya que los demás habrán terminado.

ESCRITURA: Si el último que leyó/escribió fue posterior a mí, no puedo escribir porque me cargo lo que ha escrito el átomo anterior a mí o la lectura que haya hecho.

- **Ventaja:** Mejora el anterior: no aborta por dos lecturas sucesivas ya que no tiene ningún sentido.
- **Problema:** hace trabajo en vano cuando RR(a) <= i < WR(a), porque escribe información obsoleta.
    - Si el ultimo leído y escrito es anterior a mi o yo mismo, yo puedo escribir. Pero si el último que ha leído/escrito es posterior a mí, yo me muero.
    - Ese problema viene de que las condiciones que pueda fallar un if son tres posibles, y como se están agrupando todas, no se permite separar entre uno y otro.

- Alternativa:
    - Si escribo supone mi muerte, no escribo y no me muero.
    - Ese problema viene por culpa de las condiciones de un if. La alternativa dice que si el que ha escrito es yo o anterior a mi puedo intentar escribir. Si el último que escribió no es anterior a mí, no hago nada. El algoritmo sigue adelante sin escribir ni actualizar controladores ni nada, y la transacción no falla. Si el ultimo que ha leído es posterior a mi yo no puedo escribir porque otro ha leído así que me muero.

```
Procedimiento escribe (Ti, a); {* lee/escribe y escribe/escribe *}
   INICIO
      Si (RR(a) <= i) entonces
        Si (WR(a) <= i) entonces
           {Ejecutar la escritura}
           WR(a) := i;
        Fin-si
      si-no
        {ABORT: abortar la ejecucion}
      Fin-si
   FIN
```

Si la última que escribió es posterior, se ignora la operación de escritura.

Algoritmo de ordenación parcial multi-version:

- Si el problema es que, cada vez que una transacción hace una lectura lee una versión del átomo que ha de ser la más moderna, entonces se busca en una cola de versiones, retrocediendo hasta que se encuentre el último que se ha escrito, que tiene el valor más moderno porque ese es el que debo leer yo, y pongo mi valor de lectura.
- Problema: necesita un montón de espacio.
- Persigue que las lecturas no aborten una transacción, pero implica que haya distintas versiones de cada átomo.
- Solo habrá que buscar la última versión escrita con referencia menor que la de la transacción en curso.

```
Procedimiento lee (Ti, a);
   INICIO
      j := {ultima version de a};
      Repite mientras (WRj(a) > i)
        j := j - 1;
      Fin-repite
      {Ejecutar la lectura de la version j}
      RRj(a) := Max(RR(a),i);
   FIN
```

Cuando tienes que leer, buscas la última versión de "a" que encuentres en la cola (compuesta de controladores y el átomo en cuestión), mientras que el que haya escrito sea posterior a mi sigue bajando, hasta encontrar mi átomo o uno anterior a mí. Si no has leído nunca, encuentras el anterior a ti con su propia versión.

Como se acaba de leer, en esa versión que has encontrado le cambias la lectura.

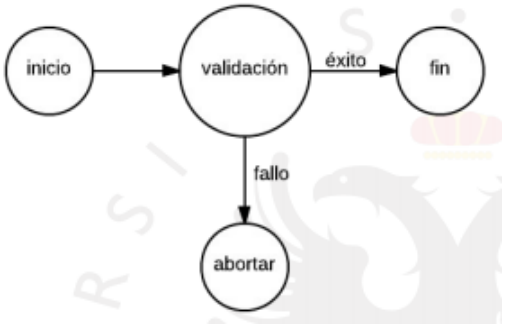
Al escribir un átomo, se busca la última versión de dicho átomo y mientras el ultimo que haya leído sea posterior hay que irse para atrás.

```
Procedimiento escribe (Ti, a);
   INICIO
      j := {ultima version de a};
      Repite mientras (RRj(a) > i)
        j := j - 1;
      Fin-repite
      Si (WRj(a) > i) entonces --Solo si 0 en lectura
        {ABORT: Abortar ejecucion}
      si-no
        {Ejecutar la escritura insertando una versión (j+1) de a}
        WRj+1(a) := i;
      Fin-si
   FIN
```

La consistencia está repartida entre las lecturas y las escrituras.

Control de concurrencia mediante validación:

- Cada vez que se lee un átomo se mete en átomos leído y cuando escribes en átomos escritos.
- Las transacciones se dividen en tres fases:
    - Lectura: donde se leen átomos, realizan cálculos y actualizan variables.
    - Validación: se comprueba la validez de los datos, operaciones correctas.
    - Escritura: se vuelcan los átomos al buffer. Los cambios se hacen efectivos (commit).
- Para cada transacción, se guardan las marcas de tiempo para inicio, validación y fin. Estas tres fases cuando han empezado, cuándo se ha validad y si la validación es correcta o incorrecta y la marca de fin. Si es incorrecta o la validación no es positiva, no hay fin.



Hay un inicio de la transacción, se realizan todas las operaciones y tras la última se realiza una validación, en la que se comprueba si las operaciones son consistentes con el resto de las transacciones. Si la validación es correcta entonces pasamos al "fin", y sino directamente se aborta y se deshace cualquier cambio hecho, o si están ciertos valores almacenados en un bloque de memoria, directamente se elimina ese bloque y se libera.

```
Procedimiento COMPROBAR (i);
INICIO
   Para cada j tal que (inicio(i) < fin(j))
      /* Tj activas con marcas inicio o válida /
      Si (AR(i) ∩ AW(j) ≠ ∅) entonces
        {ABORT} /* Abortar la ejecucion */
      Fin-si
   Fin-para
   Para cada j tal que (fin(j) < valida(i))
      /* Tj activas con marca válida /
      Si (AW(i) ∩ AW(j) ≠ ∅) entonces
        {ABORT} /* Abortar la ejecucion de Ti */
      Fin-si
   Fin-para
FIN
```

Transacciones

Átomos leídos

Átomos escritos

Para cada transacción que haya terminado después de que yo empezara, significa que son aquellas que han podido cambiar cosas que yo he intentado acceder, si los átomos a los que he leído yo no tienen nada que ver con los átomos que ha accedido otro después, entonces, no hay problema.

Pero si yo he leído algo, que otro haya escrito, se muere.

Para todas aquellas que hayan terminado antes de mi validación (las que tengan un fin de J), si lo que han escrito y lo que yo he escrito, es lo mismo, no se puede, se muere.

Hasta ahora, lo que hemos visto, planifica el retraso de transacciones que ejecutan accesos conflictivos y la repetición de operaciones ya ejecutadas.

Estas técnicas detectan ejecuciones no serializables y cuando es así, se matan por culpa de una operación compatible. En cambio, otras técnicas, cuando se detecta esa situación, se para.

Técnicas de bloqueo:

- Evitan ejecuciones incorrectas manteniendo en espera las transacciones con operaciones conflictivas sobre el mismo átomo.
- Intercambian operaciones para evitarlo.
- Permiten la ejecución simultánea de operaciones compatibles en base a su modo.
- Modo de operación (¿Qué se va a hacer sobre un átomo?):
    - Modos clásicos: lectura, actualización/escritura.
    - Carácter: exclusivo (nadie más puede hacer algo salvo ese átomo) y protegido (importa poco como quede la cosa).
    - Propuestas CODASYL (grupo de expertos de BD a nivel mundial que definen el estándar SQL y todas las cuestiones de BD), determinaron que hay seis formas de acceso a un átomo: *consulta no protegida, consulta protegida, actualización no protegida, actualización protegida, consulta exclusiva y actualización exclusiva.*

		Consulta		Actualización		Exclusiva	
		Prot.	No Prot.	Prot.	No Prot.	Consulta	Actualiz.
Consulta	Prot.	1	1	1	1	0	0
	No Prot.	1	1	0	0	0	0
Actualiz.	Prot.	1	0	1	0	0	0
	No Prot.	1	0	0	0	0	0
Exclus.	Consulta	0	0	0	0	0	0
	Actualiz.	0	0	0	0	0	0

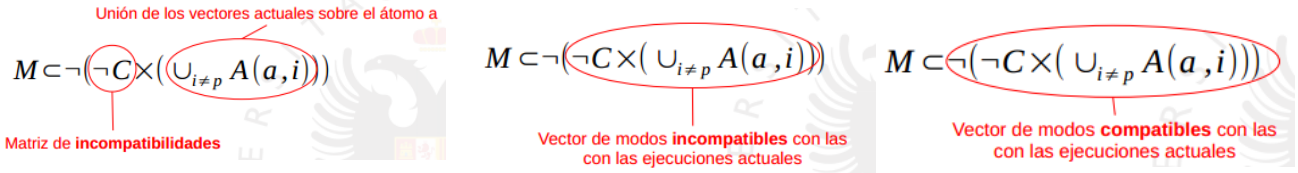
    - Establecen una matriz de compatibilidad, especificando si los modos son compatibles entre ellos, ya que al ser seis es una matriz cuadrada.
    - La actualización exclusiva es incompatible con cualquier otro modo de acceso. Significa que hay 0 en todas las columnas. No es compatible ni consigo misma, esto quiere decir, que, si una transacción está accediendo a un átomo en modo de

consulta exclusiva, como otra transacción intente acceder en modo de consulta exclusiva, se ha de esperar.
    - Sin embargo, las tres primeras consultas, tanto protegida como no protegida, son compatibles consigo mismas.
    - Lo único que se hace es comprobar específicamente, si dos transacciones pueden acceder al mismo átomo en el mismo momento.

- Compatibilidad:
    - El controlador sólo permite la ejecución de operaciones compatibles.
    - Los protocolos de bloqueo (mecanismos) se basan en dos operaciones:
        - LOCK (a, M): bloqueo del átomo *a* en modo *M*. Reserva de recurso.
        - UNLOCK (a): desbloqueo del átomo *a*. Liberación de recurso.

    - Se suele usar:
        - Un controlador por cada átomo.
        - Un vector de 6 bits para cada átomo y transacción *A (a, i)* que almacena el acceso actual de la transacción *i* al átomo *a*, es decir, el modo de operación aplicado. Este vector tiene que coincidir con alguna combinación de la matriz de compatibilidad.
        - Un vector de 6 bits por cada modo (seis vectores en total), con un 1 en la correspondiente posición.

    - La operación de bloqueo lo único que hace es comprobar una situación concreta, cuando se intenta bloquear se intenta usar un modo concreto, un vector de modo. Si ese vector es un subconjunto de lo que se llama el vector de compatibilidades significa que hay un uno en la misma posición que el otro tienen un uno, entonces son compatibles. El problema es calcular el vector de compatibilidades.
    - Una operación de bloqueo LOCK (a, M) es compatible con los modos actuales de ejecución si:




```
Procedimiento LOCK (a, M);
INICIO
  Si M⊂¬(¬C×(∪i≠p A(a,i))) entonces
    A(a,p) := A(a,p) ∪ M;
  si-no
    Insertar (p, M) en Q(a);
    Bloquear p;
  Fin-si
FIN
```

Transacción

Cola de transacciones bloqueadas y modos por operación incompatible sobre el átomo *a*

Transacción y modo de acceso a un átomo

```
Procedimiento UNLOCK (a);
INICIO
  A(a,p) := 0;
  Para cada (q, M') de Q(a) hacer
    Si M⊂¬(¬C×(∪_{i≠p} A(a,i))) entonces
      A(a,q) := A(a,p) ∪ M';
      Extraer (q, M') de Q(a);
      Desbloquear q;
    Fin-si
  Fin-para
FIN
```

Transacción

Cola de transacciones bloqueadas y modos por operación incompatible sobre el átomo *a*

Transacción y modo de acceso a un átomo

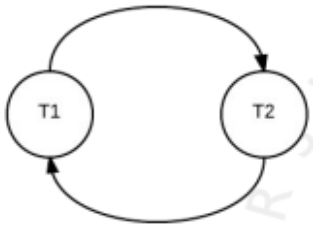
- Algoritmos de bloqueo:
  - Nunca voy a matar una transacción, sino que voy a matarla. ¿Cómo resolverlo de forma concurrente? Semáforos, controlador. Cuando se necesita un átomo, se dice que se necesita y para qué.
  - Permiten la ejecución simultánea de operaciones compatibles sobre el mismo átomo.
  - Una transacción sólo se ejecuta si consigue el bloqueo de todos sus átomos.

- Transacciones en dos fases:
  - Antes de acceder a un átomo, hay que bloquearlo.
  - **Transacción en dos fases:** aquella que no ejecuta LOCK después de un UNLOCK (primero los LOCKs y luego los UNLOCKs). Si se necesitan átomos, se hacen todos los bloqueos al principio y todos los desbloqueos después.
  - Cuando una transacción alcanza bloqueo máximo, comienza la ejecución de sus operaciones.
  - El orden de ejecución de transacciones lo establece el instante en el que alcanzan el estado de bloqueo máximo.
  - El problema está en si alguno no consigue bloquearlos todos, y nadie puede bloquear más, eso es lo que se llama interbloqueo (alguien espera lo que otros tienen).
  - Toda ejecución completa de un conjunto de transacciones de dos fases es serializable.
  - Para que la ejecución concurrente sea serializable, toda transacción debe cumplir:
    - Que las operaciones LOCK se hagan con el modo correcto y sobre el átomo necesario.
    - Que las operaciones UNLOCK sobre el átomo se realice una vez terminadas todas las operaciones sobre dicho átomo.
    - Que no se haga una operación LOCK después de ninguna operación UNLOCK.

- Bloqueo mortal:
  - Un grupo de transacciones está esperando átomos y nadie desbloquea ninguno.
  - Se da un bloqueo mortal (o deadlock) cuando:
    - Un grupo de transacciones están a la espera de que otras desbloqueen un átomo.
    - La ejecución de transacciones no bloqueadas no desbloquea ningún átomo requerido por ninguna de las transacciones bloqueadas.
  - Posibles tratamientos de esta situación:
    - Prevenir: garantizar que nunca se va a dar un interbloqueo (proactiva).
    - Detectar: cuando se detecta un interbloqueo, hay que resolverlo (reactiva).

- Grafos de bloqueo: grafos en los que los nodos son transacciones que acceden a átomos y los arcos determinan la relación de espera. Se construye sobre la marcha.
  - Cada vez que una transacción tiene que esperar a otra por culpa de un bloqueo, representamos en el grafo quien espera a quien y porqué.
- Se produce un bloqueo mortal si y sólo si el grafo de bloqueo contiene un ciclo.
- Ejemplo:

T1 espera a T2 (arco de T1 a T2) y T2 espera a T1 (arco de T2 a T1)



- Prevención:
  - Ordenar las transacciones, de modo que no puede haber una transacción antigua esperando por una nueva.
    - DIE-WAIT: una transacción sólo espera a otra si esta otra es más joven; si no, se repite desde el principio.
    - WOUND-WAIT: una transacción espera a otra más vieja; si no, mata a una transacción más joven.
  - Problema: aborta muchas transacciones no conflictivas realmente.
- Detección:
  - Explorar el grafo para encontrar ciclos.
  - Si existe un ciclo, matar una transacción del ciclo:
    - La que bloquea el átomo más solicitado,
    - La que bloquea el mayor número de átomos.
- N(k) = 0 supone que el nodo está resuelto.
- Los nodos resueltos no provocan bloqueo y se eliminan, para re-evaluar los que quedan.
- Para aplicar este método, hay que:
  - Re-marcar nodos después de borrar uno,
  - Tener una función *SLOCK (a, k, j)* que determina si la operación j puede llevarse a cabo por la transacción *k* sobre el átomo *a* dado el estado actual del grafo.

```
Función DETECTAR;
INICIO
   T = {transacciones j tales que N(j) = 0}
   R = {lista de átomos de las transacciones j incluidas en T}
   Repetir mientras aristas tachadas
      Para cada átomo i de R hacer
         Para cada par (Tk, Mj) en espera del átomo i hacer
            Si (SLOCK (i, k, j)) entonces
               {Tachar arista (ai,Mj) con origen en Tk};
               N(k) := N(k) - 1;
               Si (N(k) = 0) entonces
                  {Quitar Tk de T};
               Fin-si;
            Fin-si;
         Fin-para;
      Fin-para;
   Fin-repetir;
   Si (T = ∅) entonces
      Detectar = falso;
   si-no
      Detectar = verdadero;
   Fin-si;
FIN
```