MVCC: Control de Concurrencia Multiversión sobre Bases de Datos. Comparación critica de implementaciones existentes

Diaz Ramirez, Rodrigo Marcos
Errecart, Rodolfo Emilio

INDICE

1 - INTRODUCCION

La técnica de control de concurrencia Multiversión es ampliamente usada en el mercado por marcas líderes. Esta posee características deseables tales como permitir lectores y escritores simultáneos, e indeseables tales como cancelar la transacción en algunas situaciones.

El análisis y entendimiento del funcionamiento del MVCC permitiría un uso correcto del mismo, aprovechando sus ventajas y evitando sus desventajas.

En los capítulos 2 y 3 se definen conceptos teóricos generales de bases de datos y las técnicas o modelos de control de concurrencia más conocidas.

En el capitulo 4 se describe específicamente el control de concurrencia multiversión. Se analiza la correctitud para lo cual se definen las historias multivaluadas y se analizan implementaciones teóricas de éste modelo. Además se analizan implementaciones en productos del mercado.

En el capitulo 5 se analizan los niveles de aislamiento de las técnicas de control de concurrencia existentes, se realiza una critica a la especificación ANSI y se amplía la definición de ésta para enmarcar los niveles de aislamientos provistos por MVCC.

En el capitulo 6 se define una aplicación cuyo objetivo es mostrar y comparar como se resuelven en los distintos modelos existentes los problemas de control de concurrencia entre lectores y escritores en una base de datos.

En el capitulo 7 se define otra aplicación, en este caso para comparar MVCC con otras técnicas de control de concurrencia en un caso real. Para realizar esto se utiliza un conjunto de procedimientos estándar para evaluar el rendimiento. El esquema de base de datos y las transacciones definidas están basados en el estándar TPC-C.

2 - DEFINICIONES GENERALES DE BASES DE DATOS

Base de Datos

Se puede definir a una base de datos como un conjunto no redundante de datos estructurados organizados independientemente de su utilización y su implementación, accesibles en cualquier momento por usuarios concurrentes con necesidades de información diferentes. [1]

SGBD (Sistema de Gestión de Base de Datos)

Un Sistema de Gestión de Base de Datos (SGBD) consiste en una colección de datos interrelacionados y un conjunto de programas para accederlos. El objetivo principal de un SGBD es proveer un entorno eficiente para obtener información y almacenarla en la base de datos.

Los sistemas de bases de datos están diseñados para manejar grandes volúmenes de información. El manejo de datos involucra la definición de estructuras para el almacenamiento de la información y mecanismos para la manipulación de la información. Además deben proveer seguridad para la información almacenada ante fallas del sistema o accesos no autorizados, y deben evitar posibles anomalías en los datos, cuando éstos son accedidos por varios usuarios en forma concurrente. [2]

Sistemas monousuario y multiusuario

Uno de los criterios para clasificar sistemas de bases de datos es por el número de usuarios que pueden usar el sistema concurrentemente, esto es al mismo tiempo.

Un SGBD es monousuario si a lo sumo un usuario puede usar el sistema al mismo tiempo, y es multiusuario si permite que varios usuarios usen el sistema concurrentemente en el mismo momento.

Los sistemas de bases de datos monousuarios están restringidos a sistemas de microcomputadoras, la mayoría de los SGBD son multiusuario.

En un SGBD multiusuario, los ítems de datos almacenados son el recurso principal que pueden ser accedidos por los programas de usuario concurrentemente, los cuales constantemente recuperan y modifican las base de datos.

Transacción

Una transacción es una unidad atómica de trabajo que se ejecuta completamente o no se ejecuta nada. [3]

Una transacción es una unidad de programa que accede y posiblemente actualiza varios ítems de la base de datos. Se exige que las transacciones no violen ninguna restricción de consistencia de la base de datos. Es decir, si la base de datos era consistente cuando comenzó la transacción, la base de datos debe ser consistente cuando la transacción termine con éxito. Sin embargo, durante la ejecución de una transacción puede ser necesario permitir inconsistencias temporalmente. [2]

Una secuencia de lecturas y/o escrituras sobre los ítems de datos almacenados en una base de datos se denomina **transacción.**

Una transacción es una secuencia de ejecución de sentencias, y es atómica con respecto a la recuperación. Esto implica que el resultado de la ejecución es completamente satisfactorio o bien no produce efectos sobre los datos. [4]

Si las operaciones realizadas en la transacción no actualizan ningún dato, es decir no hay escrituras, la transacción se denomina **transacción de solo lectura**.

Para propósitos de recuperación el sistema necesita mantener una marca de cuando la transacción comienza, termina, se compromete, o aborta.

El manejador de recuperaciones guarda las marcas de las siguientes operaciones:

- **Comienzo de Transacción (Begin Transaction)**: Marca el comienzo de la transacción.
- **Lectura o Escritura (Read o Write)**: Lecturas o escrituras que se ejecutan como parte de una transacción sobre los ítems de la base de datos.
- **Fin de Transacción (End Transaction)**: Especifica que las operaciones de lectura y escritura de una transacción han terminado y marca el límite final de la ejecución de la transacción. Este punto es necesario debido a que los cambios introducidos por la transacción pueden ser aplicados en forma permanente (comprometidos) o porque la transacción tiene que ser deshecha porque viola el control de concurrencia o por alguna otra razón.
- **Comprometer la Transacción (Commit Transaction)**: Indica un fin exitoso de la transacción por lo que cualquier cambio (actualización) ejecutado por la transacción puede ser comprometido en la base de datos en forma segura y no será deshecho.
- **Deshacer la Transacción (Rollback Transaction)**: indica que la transacción ha terminado sin éxito, por lo que cualquier cambio o efecto producido por la transacción en la base de datos debe ser deshecho.

Además de las operaciones anteriormente mencionadas, algunas técnicas de recuperación requieren las siguientes operaciones adicionales:

- **Undo**: Similar al Deshacer (Rollback) excepto que se aplica a una sola operación en vez de a una transacción.
- **Redo**: Esto especifica que ciertas operaciones de una transacción deben ser rehechas para asegurar que todas las operaciones de una transacción comprometida se hayan realizado en la base de datos satisfactoriamente.

Estados de una Transacción

La *figura 2.1* muestra un diagrama de transición de estados que describe como una transacción se mueve a través de sus estados de ejecución. Una transacción entra en estado activo (**ACTIVA)** inmediatamente después de que comienza su ejecución, esto es cuando se produce una operación de lectura o escritura.

Cuando la transacción termina pasa al estado Parcialmente Comprometida (**PARCIALMENTE COMETIDA)**. En este punto, algunas técnicas de control de concurrencia requieren que se realicen ciertos chequeos para asegurar que la transacción no hubiese interferido con la ejecución de otras transacciones concurrentes. Además algunos protocolos de recuperación necesitan asegurarse que una falla en el sistema no permita la corrupción de la base de datos, usualmente guardando cambios en un registro de sistema. Una vez que ambos chequeos se realizan con éxito se dice que la transacción entra en el estado de Comprometida (**COMETIDA**). Tanto en el estado activo o si falla alguno de los chequeos anteriores en el estado parcialmente cometida la transacción pasa al estado fallada (**FALLADA**). En este ultimo caso si hubo operaciones de escritura, se deben deshacer (Rollback).

El estado terminado (**TERMINADA**) se corresponde a una transacción que sale del sistema.

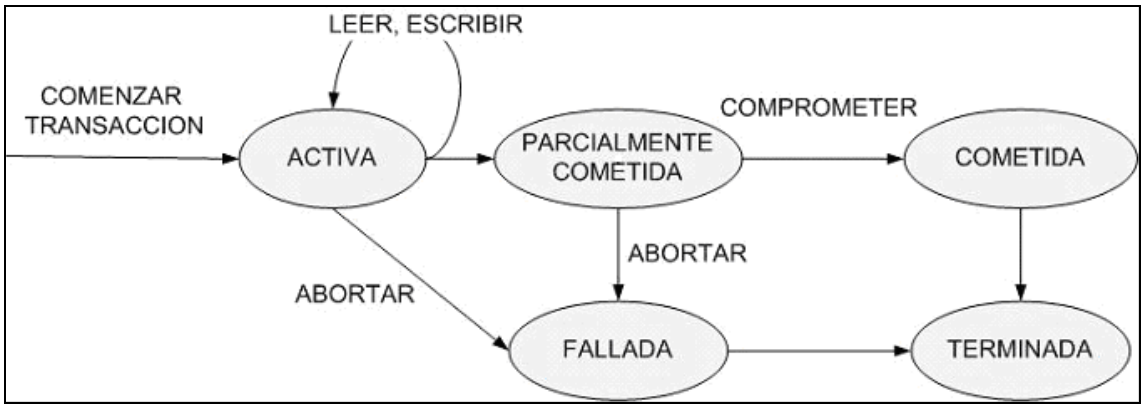


Figura 2.1

Propiedades ACID

Existen propiedades deseables para las transacciones que se denominan propiedades ACID, y deberían estar garantizadas por el control de concurrencia y los métodos de recuperación del SGBD.

Las propiedades ACID son [5]:

1) **Atomicidad (Atomicity):** Esto es que las operaciones incluidas en la transacción o bien se ejecutan todas o bien no se ejecuta ninguna.
2) **Consistencia (Consistency):** La ejecución correcta de una transacción cambia la base de datos de un estado consistente a otro consistente.
3) **Aislamiento (Isolation):** Una transacción no debería permitir que otras transacciones "vean" los cambios realizadas por ella misma, hasta que se comprometa.
4) **Durabilidad (Durability):** Una vez que la transacción cambia la base de datos y los cambios son comprometidos nunca se deben perder a causa de fallos subsecuentes.

La propiedad de atomicidad requiere que se ejecute una transacción por completo; esto es responsabilidad del método de recuperación del SGBD. Si la transacción no se puede completar por alguna razón, tal como una "caída" del sistema en el medio de la ejecución de una transacción, el método de recuperación debe deshacer los cambios hechos por la transacción en la base de datos.

La propiedad de consistencia es responsabilidad de los programadores que escriben los programas y/o del modulo del SGBD que fuerza las restricciones de integridad (integrity constraints).

El estado de una base de datos es una colección de todos los ítems de datos almacenados en la base de datos en un instante en el tiempo. Un estado consistente de una base de datos cumple con las restricciones especificadas en el SGBD como también cualquier otra restricción de negocio que debería cumplirse en la base de datos.

Un programa de base de datos debería ser escrito de manera que garantice ésta propiedad. Si la base de datos se encuentra en un estado consistente antes de comenzar a ejecutarse la transacción, la misma debería permanecer en un estado consistente luego de finalizada la ejecución de la transacción, asumiendo que no existen interferencias con otras transacciones.

El Aislamiento se lleva a cabo por el método de control de concurrencia ofrecido por el SGBD; el aislamiento es la propiedad ACID estudiada en detalle para el propósito de ésta tesis y se explicará con más detalle en capítulos siguientes.

La propiedad de durabilidad es responsabilidad del método de recuperación de errores. La misma se realiza usualmente guardando cambios en una bitácora de sistema.

PROPIEDAD	RESPONSABLE
Atomicidad	**Método de Recuperación de Errores**
Consistencia	**Restricciones de Integridad en el SGBD y programas escritos por los programadores**
Aislamiento	**Control de Concurrencia**
Durabilidad	**Método de Recuperación de Errores**

[1]

Procesamiento de Transacciones en línea (On-line Transaction Processing -OLTP)

Son las transacciones del día a día que resultan de las operaciones comunes de la empresa.

Son las encargadas de mantener actualizada la correspondencia implícita entre el estado de la base de datos y el estado de la empresa.

Procesamiento Analítico en Línea (On-line Analytic Processing - OLAP)

Es el análisis de la información histórica de la base de datos, usando consultas complejas, con el propósito de tomar decisiones empresariales.

Debido al gran volumen de datos utilizado y a la complejidad de las operaciones, OLAP utiliza generalmente Datawarehouse.

Ejemplo:

- **OLTP:** Un Cliente de un Supermercado compra una gaseosa y dos latas de salsa; se actualiza la base de datos para reflejar la operación.
- **OLAP:** Cuántos Clientes de barrio norte compraron latas de salsa y gaseosas en la misma compra en los últimos dos meses?
- **Data Mining:** Existen combinaciones de productos interesantes que los clientes compran con frecuencia?

Schedulers y Recuperación

Cuando las transacciones están ejecutándose concurrentemente en forma alternada, el orden de ejecución de las operaciones de las transacciones forma lo que se conoce como **Historia o Scheduler de las Transacciones**.

Un **Scheduler S** de n transacciones T1, T2,…, Tn es una ordenación de las operaciones de las transacciones con la restricción de que para cada Ti perteneciente a S, las operaciones de Ti en S deben aparecer en el mismo orden en el que ocurren en Ti.

Sin embargo las operaciones de otras transacciones Tj pueden alternarse con las operaciones de Ti en S.

Se define la siguiente notación para las operaciones de un Scheduler:

Operación	**Notación**
Read	r
Write	w
Commit	c
Abort	a

Ejemplo:
Sa: r1(X); r2(X); w1(X); r1(Y); w2(X); c2; w1(Y); c1;

T1	**T2**
Read(X)	
	Read(x)
Write(X)	
Read(Y)	
	Write(X)
	Commit
Write(Y)	
Commit	

Dos operaciones en un Scheduler se dicen que están **en conflicto** si pertenecen a diferentes transacciones, si acceden al mismo ítem X, y si una de las dos operaciones es una escritura sobre X. [1]

Se dice que dos operaciones están en conflicto si son operaciones de transacciones distintas sobre el mismo dato, y por lo menos una de éstas instrucciones es una operación de escritura. [2]

Por ejemplo en el Scheduler **Sa** las operaciones r1(X) y w2(X) están en conflicto, también lo están r2(X) con w1(X) y w1(X) con w2(X).

Un Scheduler S de N transacciones T1, T2,..., Tn es un **Scheduler Completo**[1] si cumple con las siguientes condiciones:

1. Las operaciones en S son exactamente las mismas que en T1, T2,..., Tn, incluyendo como última operación de cada transacción del Scheduler una operación de commit o abort.
2. Para cualquier par de operaciones de la misma transacción Ti, el orden en que aparecen en S es el mismo orden en el que aparecen en Ti.
3. Para cualquier par de operaciones en conflicto, una de las dos debe ejecutarse en el Scheduler antes que la otra.

Las 3 condiciones mencionadas anteriormente permiten que dos operaciones que no estén en conflicto se ejecuten en el Scheduler sin definir cual primero, por lo que se puede decir que un Scheduler es un **orden parcial** de las N operaciones de la transacción. Dado que cada transacción ejecuta un commit o un abort, un Scheduler completo no tiene transacciones activas.

En general, en un sistema de procesamiento de transacciones es difícil encontrar Schedulers completos porque continuamente se envían transacciones nuevas al sistema. Por esto es útil definir el concepto de **proyección cometida C(S) de un Scheduler S,** que toma en cuenta, de un Scheduler S, solo las operaciones que pertenecen a transacciones cometidas, es decir, transacciones Ti cuyas operaciones de commit ci están en S. [9]

Seriabilidad de Schedulers

Suponer que dos usuarios (vendedores de reservas aéreas) envían al SGBD las transacciones T1 y T2 aproximadamente al mismo tiempo. Si no se permite intercalar operaciones, entonces hay solo dos formas posibles de ordenar la ejecución de las operaciones de las dos transacciones:

1. Ejecutar todas las operaciones de la transacción T1 (en secuencia) seguido de todas las operaciones de la transacción T2 (en secuencia). *Tabla 2.1*

2. Ejecutar todas las operaciones de la transacción T2 (en secuencia) seguido de todas las operaciones de la transacción T1 (en secuencia). *Tabla 2.2*

T1	**T2**
read(X)	
X:=X-N	
write(X)	
read(Y)	
Y:=Y+N	
write(Y)	
	read(X)
	X:=X+M
	write(X)

Tabla 2.1

T1	T2
	read(X)
	X:=X+M
	Write(X)
read(X)	
X:=X-N	
write(X)	
read(Y)	
Y:=Y+N	
write(Y)	

Tabla 2.2

Si se permite intercalar operaciones, hay varios órdenes posibles en los cuales el sistema puede ejecutar las operaciones individuales de las transacciones. A continuación se muestran dos ejemplos:

T1	T2
read(X)	
X:=X-N	
	read(X)
	X:=X+M
write(X)	
read(Y)	
	Write(X)
Y:=Y+N	
write(Y)	

Tabla 2.3

T1	T2
read(X)	
X:=X-N	
write(X)	
	read(X)
	X:=X+M
	Write(X)
read(Y)	
Y:=Y+N	
write(Y)	

Tabla 2.4

Un aspecto importante del control de concurrencia, llamado teoría de seriabilidad, trata de determinar que Schedulers son “correctos” y cuales no y desarrollar técnicas que permitan solo Schedulers correctos.

Schedulers Seriales y Serializables

Se dice que un Scheduler es Serial cuando las operaciones de cada transacción se ejecutan consecutivamente, sin alternar operaciones de una transacción con otra. [1]

Un Scheduler es no Serial cuando se alternan las operaciones de una transacción con otra.

Formalmente, se puede decir que un **Scheduler S es serial** si, para cada transacción T que participa en el Scheduler, todas las operaciones de T se ejecutan consecutivamente en el Scheduler; de otro modo, el **Scheduler es no Serial**. Entonces si se considera que las transacciones son independientes, un **Scheduler Serial se considerará correcto**. Esto se da porque se asume que cada transacción es correcta si se ejecuta por separado (por la propiedad ACID de Consistencia) y que las transacciones no dependen una de la otra. De éste modo no importa que transacción se ejecute primero. Mientras que cada transacción se ejecute de principio a fin sin interferencias de operaciones de otras transacciones, se llega a un resultado final correcto en la base de datos.

El problema de los Schedulers Seriales es que limitan la concurrencia o alternancia de las operaciones.

En un Scheduler Serial, si una transacción espera a que una operación de Entrada/Salida termine, el procesador no puede ejecutar otra transacción, desperdiciando así tiempo de procesamiento de CPU y haciendo que los Schedulers Seriales sean inaceptables.

A continuación se analiza un ejemplo con lo discutido anteriormente:
Teniendo en cuenta las *Tablas 2.1, 2.2, 2.3* y 2.*4* con los valores iniciales de los ítems con X=90 e Y=90, N=3 y M=2, luego de ejecutar las transacciones T1 y T2, se esperaría que la base de datos tuviera los valores X=89 e Y=93, de acuerdo al significado de las transacciones.

Si se ejecutan las transacciones de las *Tablas 2.5* o *2.6* los resultados son correctos:

T1	**T2**	**Valores**
Read(X)		X=90, Y=90
X:= X-N		
write(X)		X=87, Y=90
Read(Y)		
Y:= Y+ N		
write(Y)		X=87, Y=93
	read(X)	
	X:= X + M	
	write(X)	X=89, Y=93

Tabla2.5

T1	T2	Valores
	read(X)	X=90, Y=90
	X:=X+M	
	write(X)	X=92, Y=90
Read(X)		
X:=X-N		
write(X)		X=89, Y=90
Read(Y)		
Y:=Y+N		
write(Y)		X=89, Y=93

Tabla 2.6

Ejecutando los Schedulers no seriales de las *Tablas 2.7* y *2.8*, se observa que el Scheduler de la *Tabla 2.8* se ejecuta dando resultados correctos, mientras que el de la *Tabla 2.7*, nos da como resultado X=92 e Y=93, por lo que el valor de X es erróneo.

T1	T2	Valores
Read(X)		X=90, Y=90
X:=X-N		
	read(X)	
	X:=X+M	
write(X)		X= 87, Y=90
Read(Y)		
	write(X)	X=92, Y=90
Y:=Y+N		
write(Y)		X=92, Y=93

Tabla 2.7

T1	T2	Valores
Read(X)		X=90, Y=90
X:=X-N		
write(X)		X=87, Y=90
	read(X)	
	X:=X+M	
	write(X)	X=89, Y=90
Read(Y)		
Y:=Y+N		
write(Y)		X=89, Y=93

Tabla 2.8

El Scheduler de la *Tabla 2.7* da un resultado erróneo por el problema de la actualización perdida; la transacción T2 lee el valor de X antes de que sea cambiado por la transacción T1, por lo que solo se refleja la modificación sobre X de la transacción

T2. El cambio realizado por la transacción T1 se pierde, lo sobrescribe la transacción T2, llevando a un resultado erróneo para el valor de X.

Sin embargo, algunos Schedulers no seriales dan el resultado correcto, como el Scheduler de la *Tabla 2.8*. Entonces sería útil poder determinar cuales de los Schedulers no seriales siempre dan un resultado correcto y cuales pueden dar resultados erróneos. El concepto que se utiliza para caracterizar Schedulers de esta manera es el de seriabilidad de un Scheduler.

Formalmente, un **Scheduler S de N transacciones es serializable** si es equivalente a algún Scheduler Serial que tiene las mismas N transacciones.

Cuando se dice que un Scheduler S es serializable es equivalente a decir que es correcto, porque es equivalente a un Scheduler serial, que es correcto.

Equivalencia de Schedulers, Conflicto Equivalente y Vista Equivalente

Hay varias formas de determinar cuando dos Schedulers son equivalentes. La forma mas simple es la de equivalencia por resultado. Dos Schedulers son llamados equivalentes por resultado si producen el mismo estado final en la base de datos.

Esta forma de definir equivalencia entre Schedulers no se utiliza ya que dos Schedulers pueden llegar al mismo resultado pero solo con ciertos valores iniciales de la base de datos o peor aun llegar a los mismos resultados ejecutando diferentes transacciones. Por lo tanto la equivalencia por resultados no se usa para definir equivalencia entre Schedulers.

La forma más segura y general para determinar cuando dos Schedulers son equivalentes es verificar que las operaciones aplicadas a cada ítem afectado por el Scheduler deberían ser aplicadas a los ítems en ambos Schedulers en el mismo orden. Dos definiciones de equivalencia de Scheduler son generalmente aceptadas: Conflicto equivalentes y Vista equivalentes.

Se dice que **dos Schedulers son conflicto equivalentes** si el orden de cualquier par de operaciones que esté en conflicto es el mismo en ambos Schedulers. [1]

Si un Scheduler S puede transformarse en un Scheduler S' mediante una serie de intercambios de instrucciones no conflictivas, se dice que S y S' son conflicto equivalentes. [2]

Usando la definición anterior de conflicto equivalencia se define a un **Scheduler S como conflicto serializable** si éste es (conflicto) equivalente con algún Scheduler serial S'. En éste caso se pueden reordenar las operaciones no conflictivas en S hasta formar el Scheduler Serial equivalente S'. [2]

Decir que un Scheduler S es (conflicto) serializable (esto significa que S es (conflicto) equivalente con un Scheduler serial) es equivalente a decir que S es correcto. Que un Scheduler sea serializable es distinto a que sea serial. Un Scheduler serial representa procesamiento ineficiente porque no hay intercalado de operaciones de diferentes transacciones. Esto lleva a una baja utilización de CPU mientras una transacción espera por E/S de disco. Un Scheduler serializable provee los beneficios de la ejecución concurrente sin dejar de lado la correctitud.

En la práctica, es muy difícil probar la seriabilidad de un Scheduler. El intercalado de operaciones de transacciones concurrentes lo determina el Scheduler del S.O. Factores tales como la carga del sistema, el tiempo en el que se envían las transacciones, y las prioridades de las transacciones contribuyen al orden de las operaciones en un Scheduler del S.O. Por lo tanto, es prácticamente imposible determinar de antemano como las operaciones de un Scheduler se intercalarán para garantizar seriabilidad. Si se ejecutan todas las transacciones y después se prueba el Scheduler resultante para determinar si es serializable, se debe cancelar el efecto del Scheduler si éste no resulta ser serializable, lo cual resulta impracticable.

Lo que se realiza en la mayoría de los sistemas es determinar métodos que garantizan seriabilidad sin tener que probar si los Schedulers son serializables luego que fueron ejecutados. Uno de estos métodos utiliza la teoría de seriabilidad para determinar protocolos o conjuntos de reglas que, si se aplican a cada transacción individualmente o si se fuerzan por el subsistema de control de concurrencia del SGBD, garantizarán la seriabilidad de todos los Schedulers en los cuales participan las transacciones.

Otro problema es que, cuando se envían transacciones continuamente al sistema, es difícil determinar cuando comienza y cuando termina un Scheduler. La teoría de Seriabilidad puede ser adaptada para tratar éste problema considerando solo la proyección cometida C(S) de un Scheduler S.

Un **Scheduler S es serializable** si su proyección cometida C(S) es equivalente a algún Scheduler serial, ya que solo las transacciones cometidas son garantizadas por el SGBD. [2]

Otro factor que afecta el control de concurrencia es la **granularidad** de los ítems, esto es que porción de la base de datos representa un ítem. Un ítem puede ser tan pequeño como un simple valor o tan grande como la base de datos. Por supuesto que en éste ultimo caso se permite muy poca concurrencia.

Anteriormente se definieron los conceptos **conflicto equivalente** y **conflicto serializable;** una definición de equivalencia de Schedulers menos restrictiva que estas ultimas es la de **Vista Equivalente**

Dos **Schedulers son vista equivalentes**[1] si se cumplen las siguientes tres condiciones:

1. En S y S' participan el mismo conjunto de transacciones e incluyen las mismas operaciones de esas transacciones.
2. Para cualquier operación ri(X) de una transacción Ti en S, si el valor de X que se obtiene en la operación ha sido escrito por una operación wj(X) de una transacción Tj (o si es el valor original de X antes de que comenzara el Scheduler S), debe ser el mismo valor de X leído por la operación ri(X) de la transacción Ti en S'. O sea que tanto la operación de lectura ri(X) de la transacción Ti en S como la operación de lectura ri(X) de la transacción Ti en S' deben obtener el mismo valor de X.
3. Si la operación wk(Y) de la transacción Tk es la última operación que escribe el ítem Y en S, entonces wk(Y) de la transacción Tk también debe ser la última operación que escribe el ítem Y en S'.

La idea de **vista equivalente** es que mientras que cada operación de lectura de una transacción "lea" el resultado de la misma operación de escritura (la operación que generó el dato que se está leyendo) en ambos Schedulers, las operaciones de escritura de cada transacción deben producir los mismos resultados. Las operaciones de lectura "ven la misma vista" en ambos Schedulers. La condición 3 garantiza que la última operación de escritura en cada ítem de datos es la misma en ambos Schedulers, entonces el estado de la base de datos debería ser el mismo al finalizar los Schedulers.

Un Scheduler S es **vista serializable** si es **vista equivalente** a un Scheduler serial.

La seriabilidad en Schedulers a veces es considerada como una condición muy restrictiva para garantizar correctitud en la ejecución de transacciones concurrentes.

3 – MODELOS Y TECNICAS PARA EL CONTROL DE LA CONCURRENCIA

Introducción

A continuación, se explican algunas técnicas que son usadas para asegurar el aislamiento de las transacciones que se ejecutan concurrentemente.

La mayoría de estas técnicas usan protocolos o reglas que aseguran seriabilidad. Un conjunto importante de protocolos usa técnicas de **bloqueos** (locking) de ítems de datos. Otro conjunto de protocolos de control de concurrencia usa **timestamps** de transacciones. También existen protocolos basados en el concepto de **validación** o **certificación** de la transacción después de que ésta ejecuta sus operaciones. Estos protocolos se denominan protocolos **optimistas**.

Por ultimo se describen protocolos de control de concurrencia **multiversión** que usan varias versiones de un mismo ítem de datos.

Técnicas de Bloqueo (Locking) para el control de la concurrencia

(Referencias: [1], [2], [8] y [10])

Una de las principales técnicas usadas para el control de la concurrencia esta basada en el concepto de bloqueo de datos. Un bloqueo se puede definir como una variable asociada a un ítem de datos que describe el estado del mismo con respecto a las operaciones que pueden realizarse sobre él.

Varios tipos de bloqueos pueden usarse para el control de la concurrencia, estos son: **bloqueo binario y bloqueo múltiple-modo (bloqueos compartidos y bloqueos exclusivos)**.

Un **bloqueo binario** puede tener dos estados: bloqueado o no-bloqueado (1 o 0). Se asocia un bloqueo a cada ítem de datos de la base de datos. Si el valor del bloqueo en X es 1 (bloqueado) el ítem de datos X no puede ser accedido por las operaciones de la base de datos que intentan usarlo. Si el valor del bloqueo del ítem de datos X es 0, el ítem puede ser usado por la operación que lo consultó. Se define la siguiente notación, **LOCK(X),** para referirnos al valor del bloqueo asociado al ítem de datos X.

Cuando se usa un esquema de bloqueo binario, cada transacción debe obedecer las siguientes reglas:

1) Una transacción T debe realizar la operación lock_ítem(X) antes de realizar tanto la operación read_ítem(X) como la operación write_ítem(X).

2) Una transacción T debe realizar la operación unlock_ítem(X) después de que se hallan realizado todas las operaciones read_ítem(X) y write_ítem(X) en T.
3) Una transacción T no realizará la operación lock_ítem(X) si ya posee el bloqueo sobre el ítem X.
4) Una transacción T no realizará la operación unlock_ítem(X) si no posee el bloqueo sobre el ítem X.

Estas reglas pueden ser forzadas por un modulo del SGBD. Entre la operación lock_ítem(X) y la operación unlock_ítem(X) en la transacción T, se dice que T mantiene el bloqueo sobre el ítem(X). A lo sumo una transacción puede mantener un bloqueo sobre un ítem de datos. Dos transacciones no pueden acceder al mismo ítem de datos concurrentemente. Los bloqueos binarios son muy fáciles de implementar.

El esquema de bloqueos binarios es muy restrictivo, a causa de que a lo sumo una y solo una transacción puede mantener un bloqueo sobre un ítem de datos concurrentemente. Se debería permitir que varias transacciones accedan a un mismo ítem de datos X si todas ellas acceden a X con el propósito de lectura. Sin embargo si una transacción desea escribir el ítem de datos X, debería tener acceso exclusivo a éste.

Para éste propósito, se define un tipo diferente de bloqueo llamado bloqueo **múltiple-modo**. En el esquema de bloqueo múltiple-modo se definen 3 operaciones de bloqueo: read_lock(X), write_lock(X) y unlocked(X). A diferencia del bloqueo binario donde cada ítem de datos posee dos estados posibles, en el esquema de bloqueo múltiple-modo cada ítem posee 3 estados posibles: read-locked (bloqueo de lectura), write-locked (bloqueo de escritura) y unlocked (sin bloqueo). El bloqueo de lectura también se denomina **share-locked** (bloqueo compartido), porque permite que otras transacciones puedan leer el ítem de datos sobre el cual existe el bloqueo, mientras que al bloqueo de escritura se lo denomina **exclusive-locked** (bloqueo exclusivo), porque una única transacción puede mantener el bloqueo sobre el ítem.

Cuando se usa un esquema de bloqueos múltiple-modo, cada transacción debe obedecer las siguientes reglas:

1) Una transacción T debe realizar la operación read_lock(X) o write_lock(X) antes de ejecutar la operación read_ítem(X).
2) Una transacción T debe realizar la operación write_lock(X) antes de ejecutar la operación write_ítem(X).
3) Una transacción T debe realizar la operación unlock(X) después de que todas las operación read_ítem(X) y write_ítem(X) fueron completadas.
4) Una transacción T no debe realizar la operación read_lock(X) si ya posee el bloqueo de lectura (compartido) o el bloqueo de escritura (exclusivo) en un ítem de datos X.
5) Una transacción T no debe realizar la operación write_lock(X) si ya posee el bloqueo de lectura (compartido) o el bloqueo de escritura (exclusivo) en un ítem de datos X.
6) Una transacción T no debe realizar la operación unlock(X) a menos que posea el bloqueo de lectura o escritura sobre el ítem X.

Usando bloqueos binarios o bloqueos múltiple-modo **no se garantiza seriabilidad** del scheduler en el cual las transacciones participan.

Ejemplo:

Se Definen las transacciones T1 y T2 de la siguiente manera:

T1	T2
read_lock(Y);	read_lock(X);
read_ítem(Y);	read_ítem(X);
unlock(Y);	Unlock(X);
write_lock(X);	write_lock(Y);
read_ítem(X);	read_ítem(Y)
X:=X + Y;	Y:=X + Y;
write_ítem(X);	write_ítem(Y);
unlock(X);	Unlock(Y);

Suponer los valores iniciales: X=20, Y=30
El resultado de un scheduler serial si T1 es seguido por T2 es: X=50, Y=80
El resultado de un scheduler serial si T2 es seguido por T1 es: X=70, Y=50

Ahora se analiza un scheduler que respeta las reglas definidas anteriormente en los esquemas de bloqueos.

T1	T2
read_lock(Y);	
read_ítem(Y);	
unlock(Y);	
	read_lock(X);
	read_ítem(X);
	Unlock(X);
	write_lock(Y);
	read_ítem(Y)
	Y:=X + Y;
	write_ítem(Y);
	Unlock(Y);
write_lock(X);	
read_ítem(X);	
X:=X + Y;	
write_ítem(X);	
unlock(X);	

El resultado del Scheduler es X=50 , Y=50, y **no es serializable.**

Como se observa en el ejemplo aun siguiendo las reglas definidas en los esquemas de bloqueos se puede obtener un scheduler no serializable. Esto es porque los

ítems Y en T1 y X en T2 fueron desbloqueados muy rápido. Para garantizar seriabilidad se debe respetar un protocolo adicional que tiene en cuenta la situación de las operaciones de bloqueo y desbloqueo en cada transacción; por ejemplo el protocolo de bloqueo de dos fases (Two-phase Locking).

Protocolo de bloqueo de dos fases (Two-phase Locking)
(Referencias: [1] y [8])

Se dice que una transacción respeta el protocolo de bloqueo de dos fases si todas las operaciones de bloqueo (read_lock, write_lock) preceden la primera operación de desbloqueo en la transacción. Una transacción se puede dividir en dos fases; una de expansión, durante la cual nuevos bloqueos en los ítems de datos pueden ser adquiridos pero ninguno puede ser liberado; y una fase de encogimiento durante la cual los bloqueos adquiridos pueden ser liberados pero no se pueden adquirir nuevos bloqueos.

Las transacciones T1 y T2 del ejemplo anterior no respetan el protocolo de dos fases. Esto es porque la operación write_lock(X) es posterior a la operación unlock(Y) en T1, y es similar en T2 donde write_lock(Y) es posterior a la operación unlock(X).

Se puede probar que si cada transacción en un scheduler respeta el protocolo de bloqueo de dos fases, entonces el scheduler garantiza seriabilidad. Entonces el mecanismo de bloqueos, respetando las reglas de bloqueo del protocolo de dos fases, asegura seriabilidad.

El bloqueo de dos fases puede limitar la cantidad de concurrencia que puede ocurrir en un scheduler. Esto es porque una transacción T no podría liberar el bloqueo sobre un ítem de datos X aun después de haberlo usado si T debe conseguir el bloqueo de otro ítem de datos Y mas adelante; de manera similar T debe bloquear un ítem de datos adicional Y antes de que este sea necesario así puede liberar el bloqueo del ítem de datos X. Por lo tanto X debe permanecer bloqueado por T hasta que todos los ítems de datos que la transacción T necesita hallan sido bloqueados; solo entonces el bloqueo de X puede ser liberado por T. Mientras tanto otra transacción intentando acceder a X podría tener que esperar, aun cuando T ya uso X; de manera similar, si Y es bloqueado antes de que sea necesitado por T, otra transacción buscando acceso a Y tendría que esperar aun cuando T todavía no este usando el ítem de datos Y. Este es el precio para poder garantizar seriabilidad.

Existen variaciones del protocolo de bloqueo de dos fases (2PL), estas son: Protocolo de Bloqueo de dos fases Básico, Conservador y Estricto.

La técnica descripta anteriormente es el protocolo de bloqueo de dos fases Básico. Una variación conocida como Conservador (o Estático) requiere que una transacción bloquee todos los ítems de datos a los que desea acceder antes que comience su ejecución, declarando el conjunto de todos los ítems de los que necesitará bloqueo de lectura y el conjunto de los cuales necesitará bloqueo de escritura. Si alguno de los ítems de datos declarados no puede ser bloqueado, entonces la transacción no bloquea

nada y espera hasta que todos estén disponibles para bloquear. La variante Conservadora de 2PL es un protocolo libre de Deadlock.

En la práctica la variación más popular es el 2PL Estricto. En esta variación, la transacción T no libera ningún bloqueo hasta que ella se halla comprometida o abortada. Esta variación no esta libre de Deadlocks, pero se puede combinar con la variación Conservadora para que si lo esté.

Por lo tanto el protocolo de dos fases en general garantiza seriabilidad, pero el uso de bloqueos puede causar dos problemas adicionales, estos son: Deadlock y Livelock.

El Deadlock (abrazo mortal) ocurre cuando dos transacciones están esperando entre si a que la otra libere el bloqueo en un ítem de datos determinado. En el siguiente ejemplo, las dos transacciones T1 y T2 están en Deadlock en un scheduler parcial. T1 esta esperando a que T2 libere el bloqueo en el ítem X, mientras que T2 está esperando a que T1 libere el bloqueo en el ítem Y.

T1	T2
read_lock(Y);	
read_ítem(Y);	
	read_lock(X);
	read_ítem(X);
write_lock(X);	
	write_lock(Y);

Una manera de prevenir el Deadlock es usar un protocolo de prevención de Deadlock. Uno de estos es usado en la variación conservadora del protocolo de bloqueo de dos fases. Éste requiere que cada transacción bloquee todos los ítems de datos que va a utilizar antes de comenzar. Esta solución obviamente limita la concurrencia enormemente. Existe una gran cantidad de protocolos para prevenir Deadlock.

Otro problema que puede ocurrir cuando se usan bloqueos es el Livelock. Una transacción esta en estado de Livelock si no puede proseguir por un periodo de tiempo indefinido mientras que las demás transacciones en el sistema continúan normalmente. Esto puede ocurrir si la espera en un sistema de bloqueos es injusta, dando prioridad a algunas transacciones sobre otras. La solución estándar para el Livelock es usar un esquema de espera justo. En tal esquema se usa una política de primero en llegar, primero en ser atendido; las transacciones están habilitadas a bloquear un ítem de datos en el orden en el cual realizaron la solicitud de bloqueo. En otros esquemas se permite tener distintas prioridades, pero la prioridad del que esta esperando aumenta, con lo cual eventualmente será atendido y podrá realizar el bloqueo del ítem.

Control de concurrencia por Timestamp

(Referencias: [1])

El uso de bloqueos combinados con el protocolo de bloqueo de dos fases nos permite garantizar la seriabilidad del Scheduler. El orden de las transacciones en el Scheduler serial equivalente esta basado en el orden en el cual las transacciones bloquean los ítems de datos que necesitan. Si una transacción necesita un ítem de datos que se encuentra bloqueado, puede ser forzada a esperar hasta que el ítem sea liberado.

Otra forma de garantizar seriabilidad es utilizar una ordenación de transacciones basada en Timestamps.

Un **Timestamp** es un identificador único creado por el SGBD para identificar la transacción; los valores de Timestamp se asignan en el orden en el cual las transacciones son enviadas al sistema, por lo que un Timestamp puede ser pensado como el tiempo de comienzo de la transacción. Se define al Timestamp de una transacción T como **TS(T)**.

Las técnicas de control de concurrencia basadas en Timestamps no usan bloqueos, por lo que no puede haber deadlocks.

Los Timestamps se pueden generar de diferentes formas: una posibilidad es usar un contador que se incrementa cada vez que su valor se asigna a una transacción (una secuencia). Otra forma de implementar Timestamps es utilizar el valor actual del reloj del sistema y asegurarse de que no se generen 2 valores de timestamps en el mismo instante del reloj.

Algoritmo de ordenación de Timestamp

La idea para este esquema es ordenar las transacciones basándose en su Timestamp. El scheduler serial equivalente tiene las transacciones ordenadas por los valores de sus timestamps mientras que en el protocolo de bloqueo de dos fases un scheduler es serializable siendo equivalente a un scheduler serial permitido por el protocolo de bloqueo. En la ordenación Timestamp, el scheduler es equivalente al scheduler serial particular que corresponde al orden de las transacciones timestamps.

El algoritmo debe asegurar que, para cada ítem accedido por más de una transacción en el scheduler, el orden en el cual el ítem es accedido no viole la seriabilidad de un scheduler. Para hacer esto, el algoritmo básico asocia a cada ítem X de la base de datos dos valores de timestamps:

1. **read_TS(X)**: el **Timestamp de lectura** del ítem X; este es el Timestamp mas grande entre todos los Timestamp de transacciones que han leído el ítem X satisfactoriamente.
2. **write_TS(X)**: el **Timestamp de escritura** del ítem X; este es el Timestamp mas grande entre todos los Timestamp de transacciones que han escrito el ítem X satisfactoriamente.

Cuando una transacción T intenta realizar un **read_ítem(X)** o un **write_ítem(X)**, el algoritmo básico compara el **TS(T)** con el **read_TS(X)** y con el **write_TS(X)** para asegurarse que el orden de ejecución de las transacciones no es violado. Si se viola el orden timestamp por la ejecución de la operación, entonces la transacción T violará el scheduler serial equivalente, por lo tanto T aborta. Entonces T reingresa al sistema como una nueva transacción con un nuevo Timestamp. Si T aborta y hace rollback cualquier transacción T1 que pudiera haber usado un valor escrito por T también realiza rollback; de manera similar cualquier transacción T2 que pudiera haber usado un valor escrito por T1 debe también realizar rollback y así sucesivamente. Este efecto se conoce como **rollback en cascada** y es uno de los problemas asociados al algoritmo de ordenación de Timestamp, dado que los schedulers producidos no son recuperables.

El algoritmo de control de concurrencia debe controlar si el ordenamiento de transacciones por Timestamp es violado en los siguientes dos casos:

1) La transacción T realiza un write_ítem(X):
   a. Si read_TS(X) > TS(T) o si write_TS(X) > TS(T) entonces abortar y hacer rollback de T y rechazar la operación. Esto debería realizarse porque alguna transacción con un Timestamp mayor que TS(T), y por lo tanto posterior a T en el orden Timestamp, ha leído o escrito el valor del ítem X antes que T tuviera la chance de escribirlo.
   b. Si la condición de a) no ocurre, entonces ejecutar la operación de T write_ítem(X) y asignar el write_TS(X) a TS(T)

2) La transacción T realiza un read_ítem(X):
   a. Si write_TS(X) > TS(T) entonces abortar y hacer rollback de T. Esto debería realizarse porque alguna transacción con un Timestamp mayor que TS(T), y por lo tanto posterior a T en el orden Timestamp, ha escrito el valor del ítem X antes que T tuviera la chance de leerlo.
   b. Si write_TS(X) <= TS(T) entonces ejecutar la operación de T read_ítem(X) y asignar al read_TS(X) el mayor entre el TS(T) y el read_TS(X).

Por lo tanto el algoritmo básico de ordenación por Timestamp chequea cuando dos operaciones en conflicto ocurren en el orden incorrecto, y rechaza la última de éstas abortando la transacción que realizó la operación. Los schedulers producidos por el algoritmo son **conflicto-serializables**.

El protocolo de ordenación por Timestamp y el protocolo de bloqueo de dos fases garantizan serializabilidad de schedulers. Sin embargo algunos schedulers que son posibles en un protocolo no están permitidos en el otro. Por lo tanto ningún protocolo permite todos los schedulers serializables posibles. No hay deadlock en el ordenamiento por Timestamp. Sin embargo puede ser que se produzca reinicio cíclico si una transacción es abortada y reiniciada continuamente.

El algoritmo básico de Timestamp fuerza **conflicto-serializabilidad**, pero no garantiza schedulers recuperables; y pueden ocurrir rollbacks en cascada.

Control de concurrencia basado en Validación (Optimista)

(Referencias: [1], [2] y [8])

En todas las técnicas de control de concurrencia vistas anteriormente, se realiza algún tipo de chequeo antes de que la operación se ejecute en la base de datos. Por ejemplo, en el modelo de bloqueos, se realiza un chequeo para determinar si un determinado ítem de datos se encuentra bloqueado, en el modelo por Timestamp, el Timestamp de la transacción se valida contra el Timestamp de lectura y escritura del ítem. Estos chequeos representan sobrecarga durante la ejecución de la transacción, haciendo que la misma tarde más tiempo.

En la técnica de **control de concurrencia optimista**, también conocida como **validación** o **certificación**, no se realiza ningún tipo de chequeo durante la ejecución de la transacción. Existen varias propuestas para modelos de control de concurrencia que utilizan la técnica de validación.

En este modelo las actualizaciones no se aplican directamente a los ítems de la base de datos hasta que la transacción llega a su fin. Durante la ejecución de la transacción las actualizaciones se aplican a copias locales de los ítems que se mantienen para la transacción. Al final de la ejecución de la transacción una **fase de validación** chequea si alguna de las actualizaciones viola la seriabilidad.

Hay cierta información que necesita guardar el sistema para la fase de validación. Si no se viola la seriabilidad la transacción se compromete y la base se actualiza desde las copias locales; de otro modo, la transacción se aborta y se vuelve a ejecutar mas tarde.

Hay tres fases para este protocolo de control de concurrencia:

1) **Fase de Lectura:** Una transacción puede leer los valores de los ítems de la base de datos. Sin embargo, las actualizaciones se aplican solamente a copias locales de ítems de datos que se almacenan en el entorno de trabajo de la transacción.
2) **Fase de Validación:** Se realiza un chequeo para asegurar que no se violaría la seriabilidad si la transacción actualizara la base de datos.
3) **Fase de Escritura:** Si la fase de validación tiene éxito, las actualizaciones realizadas en las copias locales son aplicadas a la base de datos; de otro modo, se descartan las actualizaciones y se vuelve a ejecutar la transacción.

La idea del control de concurrencia optimista es hacer todos lo chequeos de una sola vez, de este modo la ejecución de la transacción se realiza con un mínimo de sobrecarga hasta que se llega a la fase de validación. Si hay poca interferencia entre las

transacciones que se ejecutan concurrentemente, la mayoría se validará con éxito, si hay mucha interferencia la mayoría de las transacciones fallarán y tendrán que volver a ejecutarse. En este último caso las técnicas optimistas no funcionan bien. Esta técnica se denomina optimista porque asume que habrá poca interferencia y así no habrá necesidad de hacer chequeos durante la ejecución de las transacciones.

Una posible implementación del protocolo optimista utiliza timestamps y se necesita que el sistema mantenga un conjunto de escritura y de lectura para cada una de las transacciones. Además se necesitan guardar para cada transacción el tiempo de comienzo y de fin para alguna de las 3 fases. El conjunto de escritura de una transacción es el conjunto de ítems escritos por la transacción; análogamente el conjunto de lectura de una transacción es el conjunto de ítems leídos por la transacción.

La fase de validación para la transacción Ti chequea que, para cada transacción Tj que está comprometida o está en su fase de validación se cumpla una de las siguientes condiciones:

1) La transacción Tj completa su fase de escritura antes de que Ti comience su fase de lectura.
2) Ti comienza su fase de escritura después de que Tj completa su fase de escritura, y el conjunto de lectura de Ti no tiene ítems en común con el conjunto de escritura Tj.
3) Tanto el conjunto de lectura como el de escritura de Ti no tiene ítems en común con el de escritura de Tj, y Tj completa su fase de lectura antes de que Ti complete su fase de lectura.

Si alguna de las tres condiciones se cumple, no hay interferencia y Ti se valida satisfactoriamente. Si ninguna de las tres condiciones se cumple la validación de la transacción Ti falla, se aborta y se vuelve a ejecutar mas tarde porque puede haber interferencia.

Control de Concurrencia Multiversión

(Referencias: [1] y [8])

Los modelos de control de concurrencia vistos hasta ahora garantizan seriabilidad demorando una operación o abortando la transacción que realizó la operación.

En este modelo para el control de la concurrencia, se mantiene el valor viejo de un ítem de datos cuando el mismo es actualizado. Este modelo se denomina **multiversión** dado que existen varias versiones para un mismo ítem de datos. Cuando una transacción requiere acceso a un ítem de datos, se elige la versión correcta para mantener la seriabilidad del scheduler, si es posible.

La idea es que algunas operaciones de lectura que se rechazarían con otras técnicas puedan ser aceptadas leyendo una versión vieja de un ítem para mantener seriabilidad. Cuando una transacción escribe un ítem, escribe una nueva versión y la versión vieja del ítem se mantiene. En general los algoritmos de control de concurrencia multiversión usan el concepto de **vista serializable** en vez de **conflicto serializable**.

Un inconveniente obvio de este modelo para el control de la concurrencia es que necesita mas espacio de almacenamiento para mantener múltiples versiones de los ítems de la base de datos; sin embargo las versiones viejas se tienen que mantener de todos modos, por ejemplo para la recuperación de la base de datos en caso de fallas. Además algunas aplicaciones de base de datos requieren que se mantengan las versiones viejas para mantener una historia de la evolución de los valores de los ítems de datos (el caso extremo es una base de datos temporal, que mantiene el seguimiento de todos los cambios y los instantes en que ellos ocurren).

4 - MVCC: CONTROL DE CONCURRENCIA MULTIVERSION

Introducción

En un algoritmo de control de concurrencia multiversión, **cada Write de un ítem de datos x genera una nueva copia o versión de x**. El SGBD que maneja el ítem x mantiene una lista de versiones de x, la cual es la historia de los valores que el SGBD le ha asignado a x. Para cada operación Read(X), el scheduler no solo decide cuando enviar la operación de lectura, sino que además le dice al SGBD cual de las versiones de x debe leer.

La ventaja de tener múltiples versiones para el control de la concurrencia es ayudar al scheduler a evitar que rechace operaciones que llegan demasiado tarde. Por ejemplo, el scheduler normalmente rechaza una operación de lectura debido a que el valor que supuestamente debería leer ha sido sobrescrito o bloquea la operación hasta que el ítem esté disponible. Con multiversiones, estos valores viejos nunca se sobrescriben y están siempre disponibles para lecturas tardías. El scheduler puede evitar rechazar una lectura simplemente dejando que se lea una versión vieja.

Mantener múltiples versiones no aumenta el costo del control de la concurrencia debido a que las versiones se necesitan de todos modos para los algoritmos de recuperación.

Un inconveniente de mantener múltiples versiones es el espacio de almacenamiento. Para controlar el espacio requerido, las versiones se deben "limpiar" periódicamente o deben ser archivadas. Las transacciones activas pueden necesitar algunas versiones por lo que la "limpieza" de versiones se debe hacer de manera sincronizada respecto de las transacciones activas. La "limpieza" de versiones es otro de los costos del control de concurrencia multiversión.

Se asume que si una transacción aborta, cualquier versión que ésta haya creado se destruye. Una **versión** es el valor de un ítem de datos producido por una transacción que está activa o comprometida. Por lo tanto, cuando un scheduler decide asignar una versión particular de x a una operación Read(x), el valor utilizado no es proveniente de una transacción abortada. Si la versión leída fue producida por una transacción activa, el sistema de recuperación necesita que el commit de la transacción que realizó la lectura sea demorado hasta que la transacción que produjo el valor sea comprometida. Si la transacción aborta (invalida la versión), el lector debe también abortar. [8]

***Nota**: En Oracle esto no es así. Las transacciones concurrentes leen solo lo comprometido, no lo producido por una transacción activa no comprometida. Ver Capitulo 4 y Anexo I ("Niveles de Aislamiento").*

La existencia de múltiples versiones es visible solo por el scheduler, no por las transacciones de los usuarios. Los usuarios a través del SGBD se manejan como si

existiera una sola versión de cada ítem de datos (la última que fue escrita desde esa perspectiva de usuario). El scheduler puede usar múltiples versiones para mejorar la performance (se rechazan menos operaciones).

Correctitud en MVCC

(Referencias: [8], [1] y [22])

Para analizar la correctitud de algoritmos de control de concurrencia multiversión, se necesita extender la **teoría de seriabilidad**. Esta extensión necesita dos tipos de historias: **historia multiversión (MV)** que representa la ejecución de operaciones de un SGBD en una base de datos multiversión, e **historia única (1V)** que es la interpretación de historias multiversión desde la única vista que tiene el usuario de la base de datos. Las historias 1V seriales son las historias que el usuario considera correctas, pero el sistema produce historias MV. Entonces para demostrar que un algoritmo de control de concurrencia es correcto, se debe probar que cada una de las historias MV es equivalente a una historia 1V serial.

Para entender lo que significa para una historia MV ser equivalente a una historia 1V serial primero se necesita extender la definición de historia 1V de la siguiente manera: para cada ítem de datos x, se nombran las versiones de x como xi, xj,...,xn donde el sufijo (i,j,..,n) es el índice de la transacción que escribió la versión. Cada operación Write en una historia MV es de la forma wi(xi) (la transacción i escribo la versión i del ítem de datos x), donde el sufijo de la versión es igual al sufijo de la transacción. Las lecturas se denotan de la forma ri(xj) (la transacción i lee la versión j del ítem de datos x, creado por la transacción j). [8]

Además se define que una historia MV (Hmv) es equivalente a una historia 1V (H1v) si cada par de operaciones en conflicto en Hmv está en el mismo orden que en H1v.

Dada la historia MV:

$$H1 = w0(x0), c0, w1(x1), c1, r2(x0), w2(y2), c2$$

Las únicas dos operaciones en H1 que están en conflicto son w0(x0) y r2(x0). La operación w1(x1) no entra en conflicto con w0(x0) ni con r2(x0) porque operan con diferentes versiones de x.
Dada la historia 1V:

$$H2 = w0(x), c0, w1(x), c1, r2(x), w2(y), c2$$

H2 está construido de manera tal que concuerda cada operación con las operaciones de H1 con los correspondientes ítems de datos x e y. Las dos operaciones que entran en conflicto en H1 (w0(x0) y r2(x0)) están en el mismo orden en H1 y en H2. Entonces de acuerdo a la definición de equivalencia antes mencionada, H1 es equivalente a H2. Pero esto no es razonable, ya que en H2 T2 lee x de T1 mientras que en H1, T2 lee x de T0 lo que podría llevar a que T2 escribiera diferentes valores en y.

Por lo tanto la definición de equivalencia basada en conflictos no funciona correctamente porque las historias MV y 1V tienen operaciones levemente diferentes – operaciones con versiones en MV y operaciones sobre ítems de datos en 1V. Estas operaciones tienen diferentes propiedades en conflicto. Por Ej. W1(x1) no entra en conflicto con r2(x0), pero las mismas operaciones en 1V si están en conflicto (w1(x) y r2(x)).

Para resolver éste problema hay que hacer referencia a la definición de vista equivalente: dos historias son **vista equivalentes** si ambas obtienen el mismo valor para todas las lecturas que tengan sobre el mismo ítem de datos, y la operación final de escritura debe ser la misma en ambas historias. *Ver Capitulo 2.*

Si se comparan las historias H1 y H2, se observa que T2 lee el valor x de T0 en H1, pero en H2 lee el valor de T1. Entonces H1 y H2 no son vista equivalentes.

A pesar de que esta es una definición satisfactoria de equivalencia, o sea encontrar una historia 1V serial equivalente a una historia MV, no es suficiente ya que no todas las historias MV seriales son equivalentes a 1V serial y sin embargo son seriales. Las historias MV que son equivalentes a una historia 1V serial se denominan historias 1-serial MV.

La manera de probar que un algoritmo de control de concurrencia multiversión es correcto, es demostrar que sus historias MV son equivalentes a las historias 1-serial MV.

Se pueden definir Schedulers para el control de la concurrencia multiversión basándose en los tipos básicos de schedulers: Protocolo de Bloqueo de 2 Fases y Ordenamiento por TimeStamp.

Algoritmos teóricos

(Referencias: [8], [1], [9] y [21])

Multiversión por Ordenamiento Timestamp

Como en todos los schedulers de Ordenación por Timestamp, cada transacción tiene un único timestamp denotado por ts(Ti). Cada operación lleva el timestamp de su transacción. Cada versión se etiqueta con el timestamp de la transacción que la escribió.

Un scheduler multiversión TO (MVTO) procesa operaciones FCFS (First Come First Served). Transforma operaciones sobre ítems de datos en operaciones sobre versiones para hacer parecer que se procesan las operaciones en orden timestamp en una base de datos de una sola versión. El scheduler procesa ri(x) primero transformándolo en ri(xk), donde xk es la versión k de x con el timestamp más grande menor o igual a ts(Ti), y luego enviando ri(xk) al SGBD.

El SGBD procesa wi(x) teniendo en cuenta dos casos. Si acaba de procesar un read rj(xk) tal que ts(Tk)<ts(Ti)<ts(Tj), entonces rechaza wi(x). Sino, transforma wi(x) en wi(xi) y lo envía al SGBD.

Tk, ts(Tk)=1	**Ti, ts(Ti)=2**	**Tj, ts(Tj)=3**
Write(xk)		
	Write(xi)	
		Read(xk)

El scheduler rechaza wi(xi). Esto debe realizarse porque alguna transacción con un Timestamp mayor que TS(Tj), y por lo tanto posterior a Tj en el orden Timestamp, ha escrito el valor del ítem x antes que Tj tuviera la chance de leerlo. *Ver TimeStamp para una versión en Capítulo 3.*

Finalmente, para garantizar la recuperación, el scheduler debe retrasar el procesamiento de ci (commit de Ti) hasta que haya procesado cj (commit de Tj) para todas las transacciones Tj que escribieron versiones que fueron leídas por Ti.

Para entender MVTO, hay que comparar el efecto que produce la ejecución (H1v) en una base de datos de una sola versión en la cual las operaciones se ejecutan en orden timestamp. En H1v, cada Read, ri(x), lee el valor de x con el timestamp más grande menor o igual a ts(Ti). Este es el valor de la versión que el scheduler MVTO selecciona cuando procesa ri(x).

Dado que MVTO no necesita procesar las operaciones en orden timestamp, un Write podría llegar al SGBD y su procesamiento podría invalidar un Read que el scheduler ya procesó. Por Ej. Se tiene que w0(x0) < r2(x0) representa la ejecución hasta ahora, donde ts(Ti)=i para todas las transacciones. Si llega w1(x1), el scheduler tiene un problema. Si transforma w1(x) en w1(x1) y lo envía al SGBD, entonces produce una historia que ya no tiene el mismo efecto que una ejecución TO en una base de datos de una sola versión. En una ejecución como ésta, r2(x) debería leer el valor de x escrito por T1, pero en la ejecución w0(x0) r2(x0) w1(x1), T2 lee el valor escrito por T0. Se dice que w1(x) invalidaría r2(x0). Para evitar éste problema, el scheduler rechaza w1(x). En general, rechaza wi(x) si ya ha procesado un rj(xk) tal que ts(Tk) < ts(Ti) < ts(Tj).

T2, ts(T2)=1	**T0, ts(T0)=1**	**T1, ts(T1)=2**
	Write(x0)	
Read(x0)		
		write(x1)

El scheduler rechaza write(x1)

Para seleccionar la versión apropiada para leer y evitar invalidar lecturas, el scheduler mantiene información de timestamp sobre las operaciones que ya ha

procesado. Para cada versión (xi), mantiene un intervalo de timestamp, denotado intervalo(xi) = {intervalo(xi) | xi es una versión de x }.

Para procesar ri(x), el scheduler examina intervalo(x) para encontrar la versión xj cuyo intervalo, intervalo(xj)={wts,rts}, tiene el máximo wts menor o igual a ts(Ti). Si ts(Ti) > rts, entonces actualiza rts a ts(Ti). El intervalo almacena el instante de creación de la versión (wts) y el instante de la última lectura que se realizo sobre esta versión(rts). Por lo tanto para procesar ri(x) se busca la versión cuyo tiemstamp sea el inmediato anterior al timestamp de la transacción. Posteriormente actualiza el tiempo de lectura de esta versión si es la última lectura realizada para la misma.

Para procesar wi(x), el scheduler examina intervalo(x) para encontrar la versión xj cuyo intervalo(xj)={wts,rts} tiene el máximo wts menor a ts(Ti). Si rts > ts(Ti), entonces rechaza wi(x). Sino, envía wi(xi) al SGBD y crea un nuevo intervalo, intervalo(xi) ={wts,rts}, donde wts=rts=ts(Ti).

Eventualmente, el scheduler puede quedarse sin espacio para guardar intervalos, o el SGBD puede quedarse sin espacio para guardar versiones. Si esto ocurre, se deben borrar versiones viejas y sus correspondientes intervalos. Para evitar problemas, es esencial que se eliminen las versiones de las más viejas a las más nuevas, es decir utilizar cíclicamente el espacio.

Bloqueo de 2 fases

Con 2 versiones

En el protocolo de bloqueo de 2 fases, un bloqueo de escritura sobre un ítem de datos x no deja que se obtengan bloqueos de lectura sobre el mismo ítem de datos x. Se puede evitar este conflicto de bloqueos utilizando 2 versiones de x. Cuando una transacción Ti escribe en x, se crea una nueva versión xi de x. Se asigna un bloqueo sobre x que previene que otras transacciones lean xi o escriban una nueva versión de x. De esta manera, se permite que otras transacciones lean la versión anterior de x. Entonces, las lecturas sobre x no son demoradas por una escritura concurrente de x.

Para utilizar este esquema, el SGBD debe almacenar una o dos versiones de cada ítem de datos. Si un ítem de datos tiene dos versiones, entonces solo una de esas versiones fue escrita por una transacción que se ha comprometido. Una vez que una transacción T1 que escribió x es comprometida, su versión de x se transforma en la única versión comprometida de x, y la versión anterior comprometida de x se hace inaccesible.

Las 2 versiones de cada ítem de datos podrían ser las mismas 2 versiones que utiliza el algoritmo de recuperación del SGBD. Si T1 escribió x pero todavía no está comprometida, entonces las 2 versiones de x son la imagen anterior de x y el valor de x que se acaba de escribir. La imagen anterior de T1 de un valor x se pueden borrar una

vez que T1 se ha comprometido. Por lo tanto, una versión vieja se puede descartar tanto para el control de la concurrencia como también por cuestiones de recuperación aproximadamente al mismo tiempo.

Un scheduler de 2 versiones con protocolo de bloqueo de 2 fases (2V2PL) usa tres tipos de bloqueos: bloqueos de lectura, de escritura, y bloqueos certificados. Estos bloqueos los maneja la matriz de compatibilidad.

	Read	Write	Certify
Read	Si	Si	No
Write	Si	No	No
Certify	No	No	No

Matriz de Compatibilidad para 2V2PL

El scheduler asigna bloqueos de escritura y de lectura cuando procesa escrituras y lecturas respectivamente. Cuando el scheduler conoce que una transacción ha terminado y está por comprometerse, convierte todos los bloqueos de escritura de la transacción en bloqueos certificados.

Cuando un scheduler 2V2PL recibe un write, wi(x), intenta asignar wli(x). Dado que los bloqueos de escritura entran en conflicto con los bloqueos certificados, el scheduler retrasa wi(x) si otra transacción ya tiene un bloqueo de escritura o certificado sobre x. Sino, asigna wli(x), transforma wi(x) en wi(xi), y envía wi(xi) al SGBD.

Cuando el scheduler recibe un read, ri(x), intenta asignar rli(x). Dado que los bloqueos de lectura solo entran en conflicto con los bloqueos certificados, se puede asignar rli(x) mientras que no exista una transacción que ya tenga un bloqueo certificado sobre x. Si Ti ya tiene un wli(x) y por lo tanto ha escrito xi, entonces el scheduler transforma ri(x) en ri(xi), leyendo la versión de x no comprometida generada por la transacción actual. Sino, espera hasta que pueda asignar un bloqueo de lectura, y luego transforma ri(x) en ri(xj), donde xj es la versión comprometida mas reciente (y por lo tanto la única) de x, y envía ri(xj) al SGBD. Dado que solo se pueden leer versiones comprometidas, el scheduler evita abortar en cascada y garantiza que las historias MV producidas sean recuperables.

Cuando el scheduler recibe un commit de Ti, ci, indicando que Ti ha terminado, trata de transformar los bloqueos de escritura de Ti en bloqueos certificados. Dado que los bloqueos certificados entran en conflicto con bloqueos de lectura, el scheduler solo puede hacer esta conversión de bloqueos en aquellos ítems de datos que no tienen bloqueos de lectura adquiridos por otras Transacciones. En aquellos ítems de datos que tienen bloqueos de lectura, se retrasa la conversión de bloqueo hasta que todos los bloqueos de lectura son liberados. Entonces, el efecto de certificar bloqueos es retrasar el comprometimiento de las transacciones Ti hasta que no haya lectores - de ítems de datos que se están por sobrescribir- activos.

La conversión de bloqueos puede provocar deadlock, tal como ocurre en 2PL estándar. Por Ej. Si se supone que Ti tiene un bloqueo de lectura sobre x y Tj tiene un bloqueo de escritura sobre x. Si Ti intenta transformar su bloqueo de lectura por uno de escritura y Tj intenta convertir su bloqueo de escritura a uno certificado, entonces las transacciones están en deadlock.

Dado que una transacción puede caer en deadlock mientras intenta transformar sus bloqueos de escritura, se puede abortar durante éste periodo. Por lo tanto, no debe liberar sus bloqueos o ser comprometida hasta que no haya obtenida todos sus bloqueos certificados.

Los bloqueos certificados en 2V2PL se comportan casi como los bloqueos de escritura en 2PL estándar. Dado que el tiempo en certificar una transacción en general es menor que el tiempo total en ejecutarla, los bloqueos certificados en 2V2PL retrasan las lecturas por menos tiempo que lo que retrasan los escritores 2PL a los lectores. Sin embargo, dado que los bloqueos de lectura existentes retrasan la certificación de transacciones en 2V2PL, la concurrencia mejorada de las lecturas tiene el costo de retrasar la certificación y por lo tanto el comprometimiento de transacciones de actualización.

Usando mas de 2 versiones

El único propósito de los bloqueos de escritura es garantizar que solo existan dos versiones de un ítem de datos en un determinado momento. Estas no se necesitan para alcanzar 1-seriabilidad. Si se dejan de lado las reglas de conflictos permitiendo que los bloqueos de escritura no entren en conflicto, entonces un ítem de datos puede tener varias versiones sin certificar (por Ej. Versiones escritas por transacciones no comprometidas). Sin embargo, si se siguen las reglas de 2V2PL, entonces solo se puede leer la última versión certificada.

Si se desea afrontar el hecho de poder abortar en cascada, entonces se tiene un poco mas de flexibilidad al permitir que una transacción pueda leer cualquiera de las versiones no certificadas (En 2V2PL a lo sumo se puede leer una sola versión no certificada). Para lograr la misma sincronización (correcta) que en 2V2PL, se tiene que modificar el scheduler de dos maneras. Primero, una transacción no se puede certificar hasta que todas las versiones que lee (excepto las que escribió ella misma) hayan sido certificadas. Segundo, el scheduler solo puede convertir un bloqueo de escritura sobre x en un bloqueo certificado si no hay bloqueos de lectura sobre versiones certificadas de x.

El scheduler ignora un bloqueo de lectura sobre una versión no certificada hasta que la versión es certificada o hasta que la transacción que tiene el bloqueo de lectura trata de certificarse. Se retrasa el otorgamiento del bloqueo de lectura hasta que la versión que se quiere leer esté certificada. La única diferencia es que ahora se puede abortar en cascada. Si la transacción que produjo una versión no certificada aborta, entonces las transacciones que leyeron esa versión deben abortar también.

Método combinado

Las multiversiones le dan al scheduler más flexibilidad para planificar las lecturas. Si el scheduler conoce de antemano que transacciones solo realizarán lecturas (y no escrituras), entonces puede obtener mayor concurrencia entre transacciones.

Las transacciones que realizan lecturas pero no escrituras se llaman consultas, mientras que las que realizan escrituras (y posiblemente también lecturas) se llaman actualizadores.

Cuando una transacción comienza a ejecutarse, le avisa a su MT (Monitor de Transacciones) si es un actualizador o una consulta. Si es un actualizador, entonces el MT le avisa al scheduler que envía un actualizador, y el scheduler ejecuta operaciones de esa transacción utilizando 2PL Estricto. Cuando el MT recibe el comprometimiento del actualizador, indicando que éste ha terminado, el MT le asigna un timestamp al actualizador.

Al contrario que en 2PL, en éste método cada operación de escritura genera una nueva versión. El scheduler rotula cada versión con el timestamp de la transacción que la escribió. El scheduler usa las versiones rotuladas para sincronizar las lecturas de las transacciones de consulta utilizando MVTO.

Cuando un MT recibe la primera operación de una consulta, le asigna un timestamp más chico que el de cualquier actualizador comprometido o activo.

Cuando el scheduler recibe un ri(x) de una consulta Ti, busca la versión de x con el timestamp mas grande menor a ts(Ti). Para la regla de asignación de timestamps, ésta versión fue escrita por una transacción comprometida. Además, por la misma regla, asignando esta versión de x a ri(x) no invalidará la lectura en un futuro (entonces no se necesitan rechazar futuras escrituras).

Una consulta no asigna bloqueos. Por lo tanto nunca tiene que esperar por actualizadores y nunca los actualizadores deben esperar por ella. El scheduler siempre puede procesar una lectura sin tener que esperar.

Seleccionar el timestamp para una consulta en una Base de Datos centralizada es fácil, ya que los actualizadores activos no tienen asignados timestamps hasta que terminan.

Para evitar quedarse sin espacio, el scheduler debe tener una forma de eliminar versiones “viejas”. Cualquier versión comprometida puede ser eliminada tan pronto como el scheduler pueda garantizar que no exista ninguna consulta que necesitará leer esa copia en el futuro. Para esto, el scheduler tiene un valor mínimo (min) no decreciente, que es el menor timestamp que se le puede asignar a una consulta. Cuando el scheduler quiere liberar espacio utilizado por las versiones, asigna el valor mínimo

(min) como el menor timestamp asignado a cualquier consulta activa. Luego puede descartar una versión comprometida xi si ts(Ti)<min y existe otra versión comprometida xj, tal que ts(Ti)<ts(Tj).

El mayor beneficio de éste método es que las consultas y los actualizadores nunca se retrasan entre si. Una consulta puede siempre leer los datos que quiera sin retrasos. Aunque los actualizadores pueden retrasarse entre si, las consultas no asignan bloqueos y por lo tanto nunca retrasan a los actualizadores. Esto contrasta con 2PL, donde una consulta puede asignar varios bloqueos y entonces retrasar actualizadores. Este retraso también lo hereda 2PL y 2V2PL, ya que un actualizador Ti no puede comprometerse hasta que no haya bloqueos de lectura de otras transacciones sobre el mismo conjunto de datos que quiere escribir Ti.

Las principales desventajas del método son que las consultas pueden leer datos viejos y que el rotulado e interpretación de los timestamps sobre versiones puede agregar overhead significante al scheduler.

Utilizando Listas Comprometidas reemplazando Timestamps

Al utilizar timestamps, el rotulado de versiones con timestamps puede ser costoso porque cuando un scheduler procesa wi(x) creando una nueva versión de x, desconoce el timestamp de Ti. El scheduler conoce el timestamp de Ti después que ésta termina. Sin embargo, para ese entonces, la versión puede haber sido movida a disco; necesita ser releída para poder ser rotulada, y luego ser rescrita a disco.

Para evitar todos estos inconvenientes, se pueden utilizar listas comprometidas en vez de timestamps. Cuando una consulta comienza a ejecutarse, el MT realiza una copia de la lista comprometida y la asocia a la consulta. Se adjunta la lista comprometida a cada lectura que se envía al scheduler, básicamente manejando la lista como un timestamp. Cuando el scheduler recibe ri(x) de una consulta Ti, busca la versión comprometida de x mas reciente cuyo rotulado está en la copia de la lista comprometida de Ti. Para hacer esto de una manera eficiente, todas las versiones de un ítem de datos se guardan en una lista encadenada, de la más nueva a la más vieja. Cuando se crea una nueva versión, se agrega al tope de la lista de versiones. Dado que los actualizadores utilizan 2PL estricto, dos transacciones no pueden crear nuevas versiones sobre el mismo ítem de datos en forma concurrente.

Para procesar ri(x) de una consulta Ti, el scheduler busca en la lista de versiones comprometidas de x de la transacción Ti. Esto es lo mismo que leer la versión comprometida más reciente de x cuyo timestamp es menor que ts(Ti).

El problema con éste esquema es el tamaño y estructura de las listas comprometidas. Primero, cada lista debe ser pequeña. En un sistema centralizado, cada consulta tendrá una copia de la lista consumiendo memoria principal. En un sistema distribuido, cada lectura enviada tendrá una copia de la lista, lo cual consume ancho de

banda en la comunicación. Segundo, dado que el scheduler debe buscar en la lista ante cada lectura de una consulta, la lista debería ser estructurada para poder determinar fácilmente si un identificador de transacción está en la lista o no.

Una buena forma de hacer esto es almacenar la lista comprometida como un mapa de bits. La lista comprometida es un arreglo, LC, donde LC(i)=1 si Ti está comprometida, sino LC(i)=0. Sin embargo, a medida que transcurre el tiempo, la lista crece sin límite por lo que se necesita una forma de mantener la lista pequeña.

La lista se puede mantener pequeña, determinando que cuando ha excedido cierto tamaño, el scheduler le pide al MT por un identificador de transacciones (n), que es mas chico que todo aquel que ha sido asignado a cualquier consulta o actualizador activo, o que será asignado a cualquier consulta o actualizador futuro. El scheduler entonces puede descartar el prefijo de la lista comprometida a través del identificador de transacciones n, acortando así la lista. Para procesar ri(x) de alguna consulta Ti, el scheduler devuelve la primera versión de la lista de versiones de x escrita por una transacción cuyo identificador está dentro de la lista comprometida que se le dio a Ti cuando comenzó (o el identificador es mas chico que cualquier identificador que está en la lista).

Cuando una transacción aborta, todas las versiones producidas por ella se eliminan de la lista de versiones.

Cuando el scheduler recibe un valor n desde el MT con el propósito de reducir el tamaño de la lista comprometida, puede hacer una especie de garbage collector con las versiones. Puede descartar una versión comprometida de x, siempre y cuando exista una versión comprometida de x mas reciente cuyo identificador es menor que n.

MultiVersión en Implementaciones Reales

(Referencias: [6], [19], [23], [24], [25], [26], [27], [28] y [29])

La primera base de datos en implementar multiversión fue InterBase(Borland)[26]. Esta empresa en el año 2000 liberó el código fuente de su producto y éste fue utilizado por FireBird[29] para crear su propia base de datos Open Source.

Oracle[6] y[19] a partir de la versión 7 implementó Multiversión y desde entonces es uno de los principales productos comerciales en utilizar este mecanismo de concurrencia.

Dentro del mundo Open Source existen dos grandes bases de datos que son PostgreSql[24] y MySQL[25]. PostgreSQL ha implementado multiversión desde sus comienzos, mientras que MySQL lo permite a partir de la introducción de Tablas InnoDB.

Sql Server[23] a partir de su versión 2005 (YUKON) introduce el mecanismo de multiversión(Snapshot). Por lo tanto de las tres principales Bases de Datos, (Oracle, SQL Server y DB2) solo DB2[27] aun no implementa multiversión.

Multiversión en Oracle
(Referencias: [6] y [19])

Hasta ahora se definió el funcionamiento del control de concurrencia multiversión y se detallaron algunas implementaciones de algoritmos teóricos. A continuación se describe una implementación real de multiversión. Se toma como modelo la implementación de Oracle por ser un producto comercial ampliamente difundido y reconocido que implementa multiversión.

Este SGBD provee consistencia de datos usando multiversión combinada con varios tipos de bloqueos de ítems de datos. Por defecto provee consistencia de lectura a una sentencia, lo que significa que todos los datos que ve la consulta provienen de un mismo instante en el tiempo. Oracle denomina a este nivel como "consistencia de lectura a nivel de sentencia". También provee consistencia de lectura para todas las sentencias que están dentro de una transacción, y denomina a este nivel "consistencia de lectura a nivel de transacción".

Oracle utiliza una estructura de datos llamada Rollback Segment (Segmento de Deshacer) en donde almacena los datos que han sido modificados por transacciones no comprometidas o recientemente comprometidas.

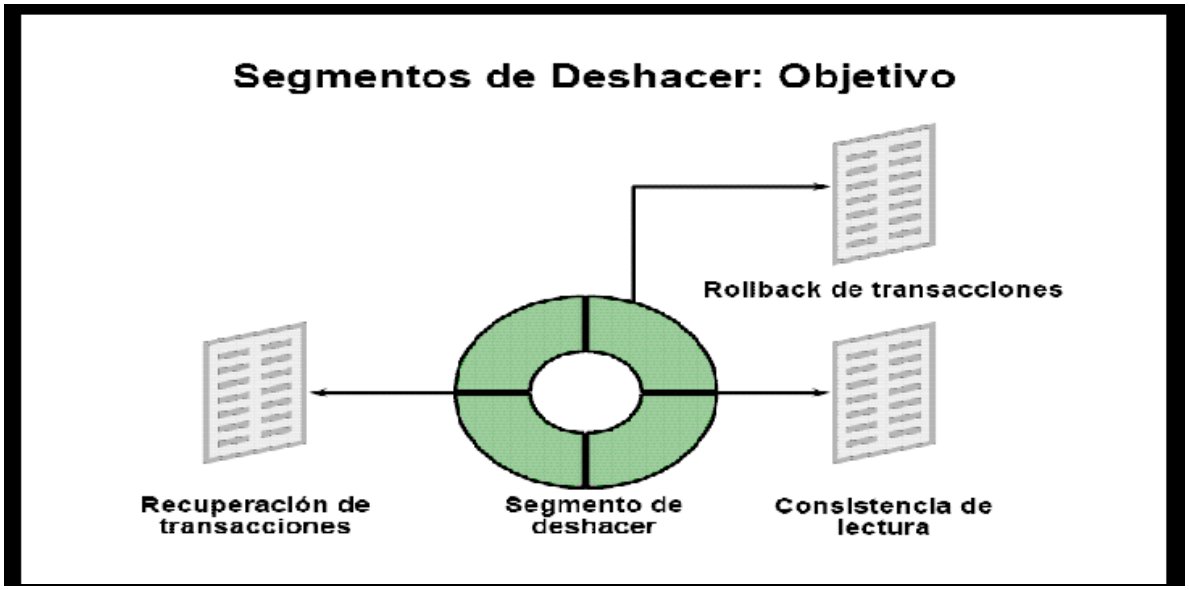


Figura 4.1

Uno de los motivos para almacenar estos datos en los Segmentos de Deshacer es brindar una vista consistente a las transacciones. (Consistencia de Lectura)

Ejemplo

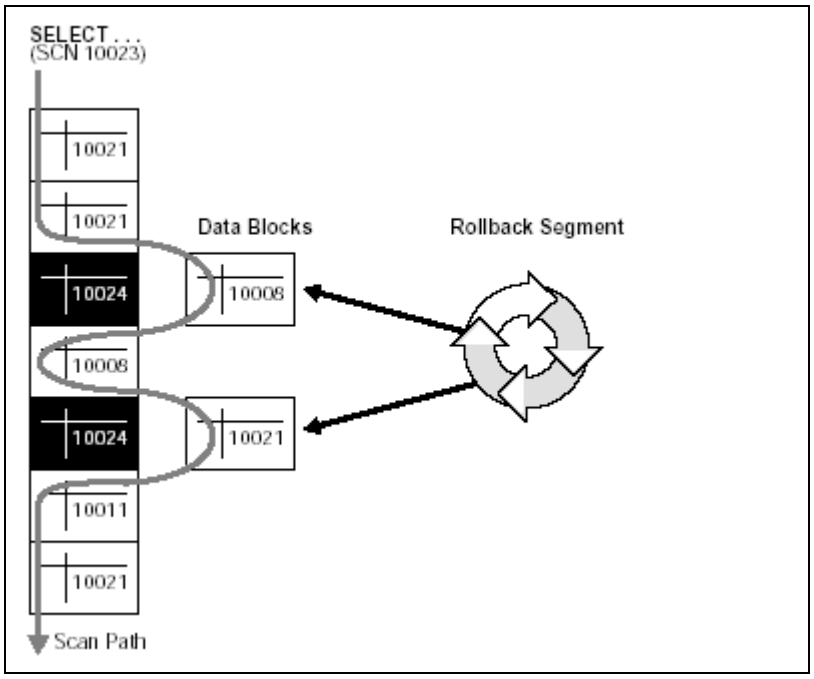


Figura 4.2

En el momento en que una transacción comienza se determina el SCN actual. El número de cambio del sistema o SCN es un valor que se incrementa automáticamente y no llega a reutilizarse nunca. El SCN identifica de forma única a una versión de la Base de Datos comprometida. En el ejemplo de la *figura 4.2* el SCN es el 10023.

Para realizar la consulta se leen los bloques de datos necesarios, y solo se usan los bloques con un SCN anterior al determinado al comienzo de la transacción (menores o iguales que 10023). Los bloques de datos con SCN más reciente se buscan en el Segmento de Deshacer.

Por lo tanto toda consulta retorna los datos comprometidos con anterioridad al SCN almacenado al comienzo de la transacción. Los cambios de otras transacciones que ocurren durante la ejecución de la transacción no son observados, garantizando así la consistencia de datos.

Un resultado consistente es retornado por cada sentencia, garantizando consistencia de datos.

Como se mencionó anteriormente, Oracle define 2 niveles de Consistencia de Lectura, uno a nivel de sentencia y otro a nivel de transacción. En la consistencia de lectura a nivel de sentencia Oracle garantiza que todos los datos son los existentes en la

base de datos al momento del inicio de la sentencia. Por lo tanto una consulta nunca "ve" datos sin comprometer o comprometidos durante su ejecución por transacciones concurrentes. En la segunda opción, "Consistencia de lectura a nivel de Transacción", los datos son consistentes durante toda la transacción; se garantiza que todos los datos son los existentes en la base de datos al momento de iniciada la transacción. En este último nivel, Oracle ofrece lectura repetida (Repeatable Read) y garantiza la ausencia de filas fantasmas (Phantom), 3er fenómeno ANSI.

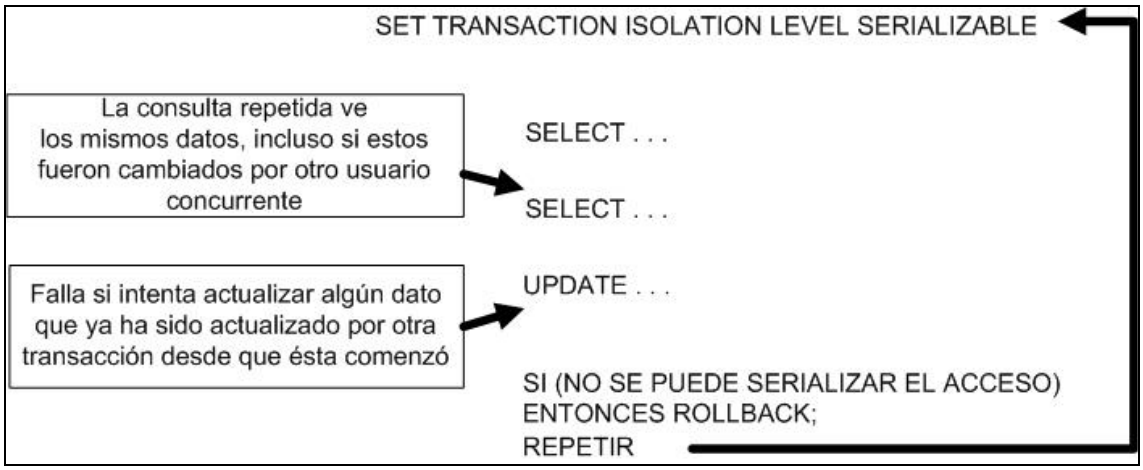
Problemas de Oracle en la implementación de Multiversión

Problema al intentar serializar la transacción

Cuando hay dos transacciones activas concurrentes que intentan modificar el mismo dato, y las dos están sin comprometerse, entonces la segunda transacción se bloqueará, esperando que la primera libere el bloqueo sobre el dato. Si la primer transacción libera el bloqueo ejecutando un Rollback la transacción que estaba bloqueada continuará sin problemas tanto en el nivel de aislamiento Read Committed como en el nivel de aislamiento Repeatable Read, que son 2 de los 3 niveles de aislamiento proporcionados por Oracle. El otro nivel de aislamiento es Read Only, pero no interesa tenerlo en cuenta para lo explicado anteriormente.

Sin embargo si la 1er transacción se compromete en vez de ejecutar Rollback, pueden suceder 2 cosas. En el nivel de aislamiento por defecto, Read Committed, es decir "consistencia de lectura a nivel de sentencia", la transacción bloqueada sobrescribirá el cambio de la 1er transacción, mientras que en el nivel de aislamiento Serializable, "consistencia de lectura a nivel de transacción", se obtiene un error advirtiendo que no se puede serializar la transacción.

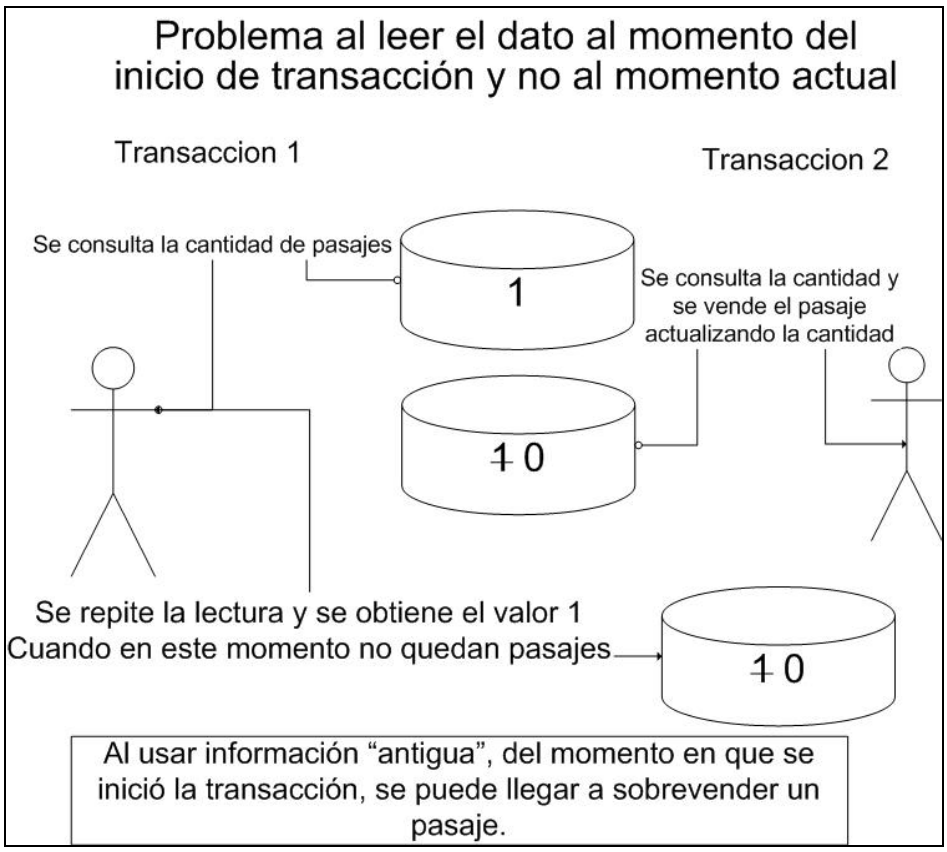
ORA-08177: can't serialize access for this transaction



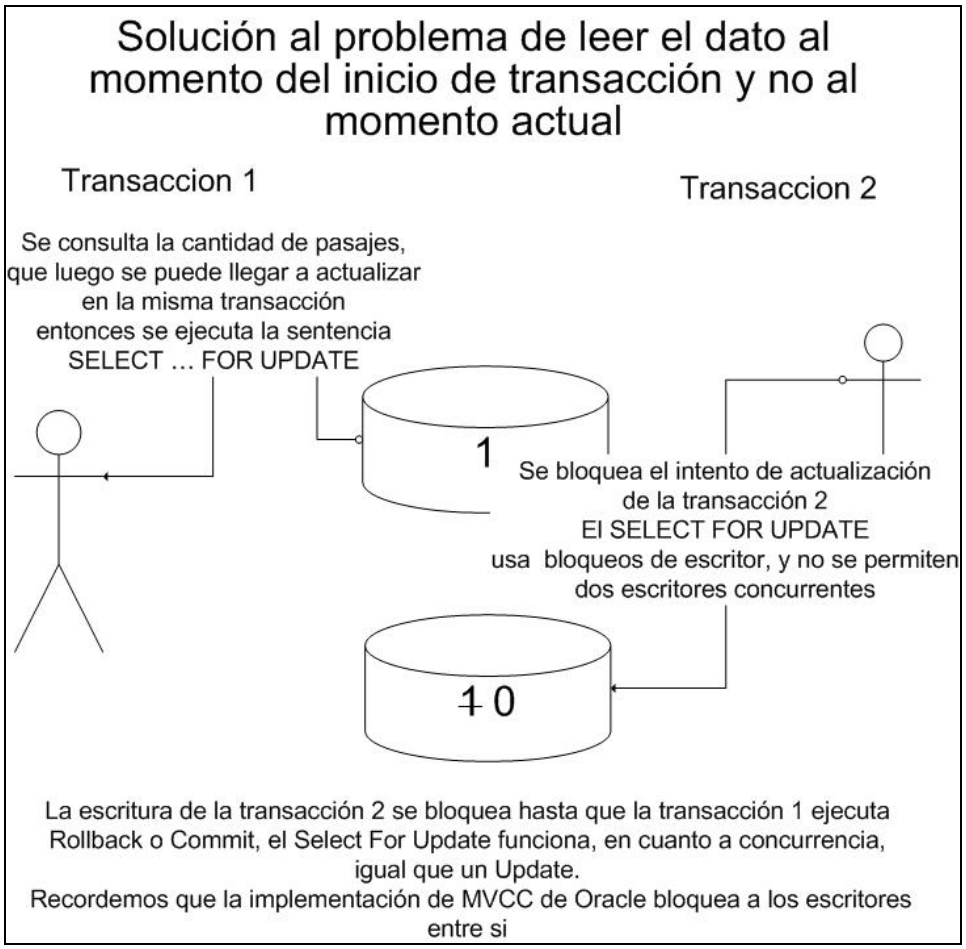
Problema de consistencia

A causa de que Oracle no usa bloqueos de lectura, los datos leídos por una transacción pueden ser sobrescritos por otra.

Las transacciones que realizan chequeos a nivel de aplicación no pueden asumir que los datos que leyeron permanecerán sin cambiar durante su ejecución debido a que los lectores no bloquean a los escritores. Pueden encontrarse entonces inconsistencias en la Base de datos, a menos que se codifique teniendo en cuenta ésta particularidad a nivel de aplicación.



La solución es codificar teniendo en cuenta el funcionamiento de MVCC en Oracle.



SELECT … FOR UPDATE usa bloqueos de escritor. Se debe codificar usando esta instrucción para evitar que otras transacciones puedan sobrescribir un dato que no puede modificarse o de lo contrario la base de datos perdería consistencia.

Esta anomalía propia de los modelos de concurrencia multiversión se conoce como "Write Skew" (escritura desviada), y se define formalmente en el capitulo 5.

Problema al definir en que momento un dato se puede borrar.
(Referencias: [6] y [18])

Uno de los problemas que se puede imaginar al ver el funcionamiento del mecanismo de control MVCC implementado por Oracle y compararlo con el modelo por bloqueos es determinar el momento en el cual dejan de ser útiles los datos. En un modelo por bloqueos una vez hecho el commit de una transacción los datos modificados pueden descartarse. En MVCC esto ya no es así, estos datos son necesarios para armar la "Consistencia de Lectura".

El problema de "imagen demasiado antigua" (Como lo denomina Oracle, “ORA-1555 snapshot too old”) se debe a que los valores antiguos que se han creado en el Segmento de Deshacer que hacen falta para la consistencia de las operaciones de lectura, desaparecen del segmento, por lo que no se cumple la consistencia de datos (pues no consigue todos los datos con valores referentes al mismo instante de tiempo), y por lo tanto se produce el error "ORA-01555: snapshot too old".

Para entender porqué desaparece el valor antiguo del segmento de rollback se debe entender la estructura y funcionamiento de este tipo de segmentos: la escritura en ellos es cíclica reutilizando el espacio que estaba siendo utilizado por las transacciones que ya hayan realizado commit o rollback.

Por lo tanto, puede ocurrir que una consulta muy larga precise que los datos de rollback se mantengan durante mucho tiempo para consistencia de datos dentro de la propia consulta, pero la transacción que los estaba actualizando hace commit; a partir de ese instante, cualquier otra transacción que tenga asignado ese mismo segmento de rollback podrá sobrescribir los valores antiguos, por lo que la consulta no podrá obtenerlos de ningún otro sitio y se producirá el error. (Rollback Failure)

Existe otro caso en el que se puede producir error, y es cuando quizá no ha desaparecido la información de rollback, pero sí ha desaparecido la entrada de transacción de la cabecera de rollback que indica de dónde leer los datos de rollback para una transacción. (CleanOut Failure)

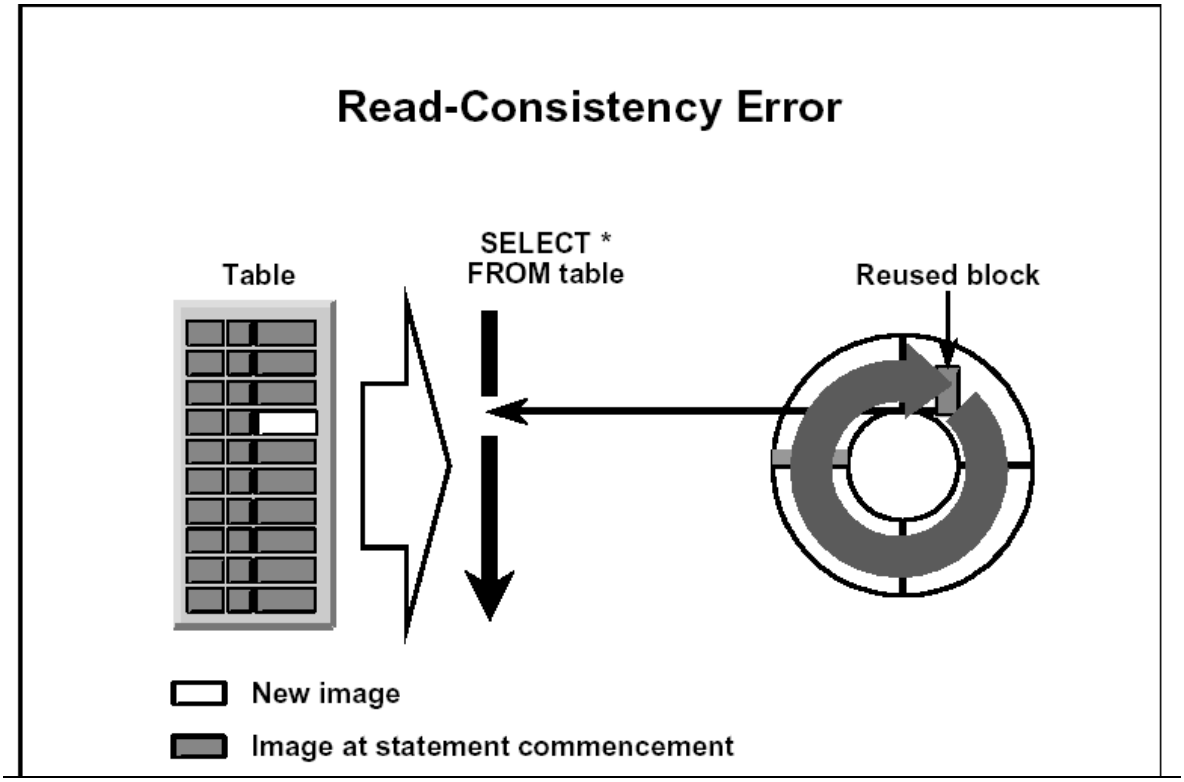
Existen varias recomendaciones para reducir el riesgo de error pero no hay una solución definitiva. Los casos más comunes que pueden presentar un rollback failure o cleanout failure son:

- Hacer fetch de un cursor abierto después de un commit. Oracle lo permite pero no es una operación ANSI standard. Se puede producir el error debido a que al abrir el cursor, éste adquiere el snapshot SCN de la consulta, y los commits irán generando nuevos commit SCNs.
- Segmento de rollback corrupto. Debido a que ciertos bloques no pueden ser leídos, no permitirá realizar la consistencia en lectura.
- Segmentos de rollback muy pequeños. Las entradas de rollback de una transacción confirmada se reutilizan demasiado rápido por nuevas transacciones.
- Pocos segmentos de rollback. Existen muchas transacciones usando el mismo segmento de rollback, por lo que existe mayor probabilidad de reutilización.

Para reducir la probabilidad de error frente a estos casos se proponen las siguientes recomendaciones:

- No ejecutar transacciones discretas mientras se estén ejecutando consultas o transacciones que puedan verse perjudicadas, a menos que los dos conjuntos de datos sean totalmente independientes.

- Planificar en el tiempo consultas y transacciones largas, de forma que las lecturas consistentes no necesiten consultar datos de rollback desde el snapshot SCN.
- Codificar los procesos largos como módulos que se puedan ejecutar de nuevo independientemente de cualquier proceso anterior.
- Comprimir todos los segmentos de rollback a su tamaño óptimo manualmente antes de ejecutar una consulta o transacción que pueda obtener el error de rollback debido a desasignación de extensiones durante su ejecución.



Multiversión en Otros Productos
(Referencias: [23], [24], [25], [26], [28], [29] y [30])

MySQL

MySQL usando tablas de tipo InnoDB usa un mecanismo similar a Oracle con una estructura de datos denominada Rollback Segment para almacenar las versiones antiguas de los datos.

PostgreSql

A diferencia de Oracle y MySQL, PostgreSQL no almacena las versiones antiguas en una estructura de datos auxiliar, sino que lo hace manteniéndolas en la misma tabla. Debido a esta implementación de Multiversión cuando se realiza un Update se inserta una nueva fila en la base de datos manteniendo en la misma la versión anterior. Simplemente se usan marcas para poder identificar cual es la versión correcta para la transacción que consulte la misma.

Ya que no se realiza un borrado real de los datos de las tablas estas podrían crecer indefinidamente, para lo cual hay que correr un proceso que PostgreSQL denomina VACUUM, que borra las versiones que no se necesitan mas. Este proceso se debe ejecutar periódicamente, con una recurrencia relacionada con la cantidad de actualizaciones.

SQL Server 2005

Sql Server 2005 almacena las versiones en una estructura de datos que denomina TempDB. Cuando se modifica un registro de la base de datos, el nuevo registro se etiqueta con el número de secuencia de la transacción que está modificando el dato. La versión vieja del registro se copia al TempDB, y el registro nuevo contiene un puntero al registro "viejo" almacenado en TempDB. Si existen múltiples transacciones largas corriendo en la base de datos y se requieren múltiples versiones, los registros en TempDB pueden tener punteros a versiones aún más viejas del dato. Todas las versiones más viejas de un registro en particular se almacenan en una lista encadenada.

Además, se puede asignar que la base de datos/log crezca según un porcentaje dado o en forma absoluta, pero obviamente siempre existe una restricción física dada por el tamaño del disco.

Cuando TempDB se queda sin espacio, se ejecuta una función de limpieza antes de autoincrementar el espacio en disco. Cuando el disco está lleno y no se puede incrementar el tamaño de TempDB, se detiene el mecanismo de versionado. Si después una consulta quiere leer una versión vieja de un registro que no fue generado por problemas de espacio, la consulta falla. Las actualizaciones y borrados no fallan. Una alternativa es detectar una transacción larga y terminarla para que no crezca el TempDB.

FireBird

En este producto**,** cuando una transacción modifica un registro, se escribe una versión nueva en la base de datos y una versión anterior (representando solo la

diferencia entre la versión del registro que leyó la transacción y el nuevo valor del registro) se almacena como una versión vieja del registro.

Cuando comienza una transacción, ésta recibe un número que se va incrementando e identifica unívocamente a la transacción dentro del sistema desde la última recuperación de la base de datos. Cada cambio que realiza la transacción se "firma" con el número de transacción. Cuando se lee un registro en medio de una transacción, el SGBD compara la "firma" del registro con el numero de transacción. Si la "firma" pertenece a una transacción que se comprometió cuando la transacción actual comenzó, ésta es la versión del registro que se utiliza. Sino, el SGBD calcula la versión que necesita utilizando el estado del registro actual y las versiones anteriores de ese registro sin considerar los lockeos que tenga la transacción.

Los mecanismos que usa FireBird para manejar las versiones anteriores son similares a los segmentos de rollback utilizados por Oracle. Pero la diferencia más importante es que FireBird no puede producir un error similar al que produce Oracle: ORA-1555. No hay necesidad de estimar el tamaño de los segmentos de rollback, ya que toda la información necesaria para las operaciones de rollback y para calcular las versiones anteriores de los registros se almacena dentro de la misma base de datos y el tamaño de la base crece automáticamente si se necesita más espacio.

5 – NIVELES DE AISLAMIENTO

Motivación

Cuando se estudia el control de concurrencia multiversión y se analizan los niveles de aislamiento proporcionados, se observa que no existe una correspondencia directa con la definición dada para los niveles de aislamiento en ANSI SQL 92 o ANSI SQL 99.

La definición provista por ANSI no tiene en cuenta los mecanismos multiversión y tampoco define varias anomalías o "fenómenos" que pueden ocurrir en los mismos. De esta manera muchos de los productos comerciales que implementan multiversión dicen proveer el nivel de aislamiento SERIALIZABLE y de acuerdo a la definición de ANSI lo hacen, pero no tienen en cuenta otras anomalías, como por ejemplo Write Skew. Esta anomalía se produce básicamente al permitir que las lecturas no bloqueen a las escrituras en multiversión, por lo tanto las transacciones podrían estar tomando decisiones basadas en datos "viejos".

En realidad con el control de concurrencia multiversión no se obtiene una verdadera seriabilidad en el sentido de una equivalencia con una historia serial.

Aunque ANSI SQL no define niveles de aislamiento para los mecanismos multiversión existen definiciones para los mismos en otras fuentes; en la mayoría de la bibliografía analizada se define al nivel de aislamiento para el control de concurrencia multiversión como nivel de aislamiento SNAPSHOT.

En este capitulo se describen los niveles de aislamiento, los mecanismos de control de concurrencia y se comparan las anomalías que éstos permiten o evitan.

Introducción

(Referencias: [4], [9], [10] y [20])

El permitir que las transacciones concurrentes se puedan ejecutar en niveles de aislamiento inferiores a la seriabilidad, permite a los diseñadores de aplicaciones poder sacrificar correctitud por eficiencia. Niveles de aislamiento menores incrementan la concurrencia de transacciones con el riesgo de permitir que las transacciones puedan observar un estado incorrecto o "sucio" de la base de datos.

El ANSI/ISO SQL-92 define cuatro niveles de aislamiento[4]:

1. READ UNCOMMITED
2. READ COMMITED
3. REPEATABLE READ
4. SERIALIZABLE

Estos niveles están basados en la definición de seriabilidad y en tres operaciones prohibidas, llamadas “fenómenos”, que son:

1. Dirty Read
2. Non-Repeatable Read
3. Phantom.

El concepto de fenómeno no esta definido explícitamente en las especificaciones de ANSI, pero este sugiere que los fenómenos son operaciones subsecuentes que podrían conducir a un comportamiento anómalo o no-serializable.

Los niveles de aislamiento ANSI están relacionados con el comportamiento de los schedulers basados en bloqueos.

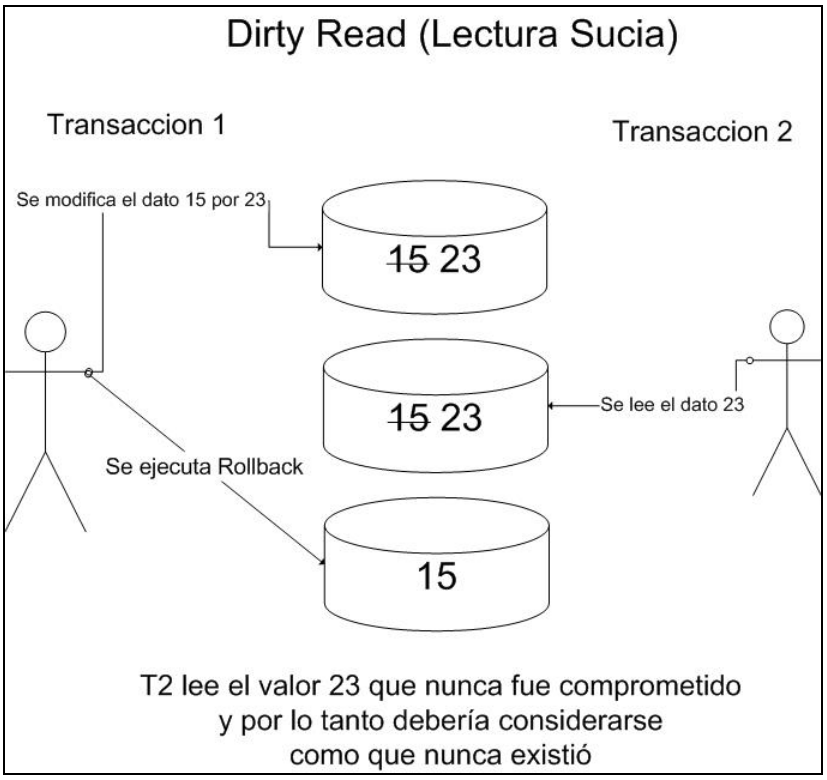
Los tres fenómenos ANSI son ambiguos, e incluso sus interpretaciones permisivas no excluyen algunos comportamientos anómalos que podrían surgir en las historias de ejecución de las transacciones. Además, los fenómenos ANSI no distinguen algunos de los niveles de aislamiento que son populares en los sistemas de bases de datos reales, ya sea comerciales y/o open source.

El control de concurrencia Multiversión usa un nivel de aislamiento que se denomina **SNAPSHOT ISOLATION;** este evita los fenómenos ANSI pero no es serializable [9] y [20].

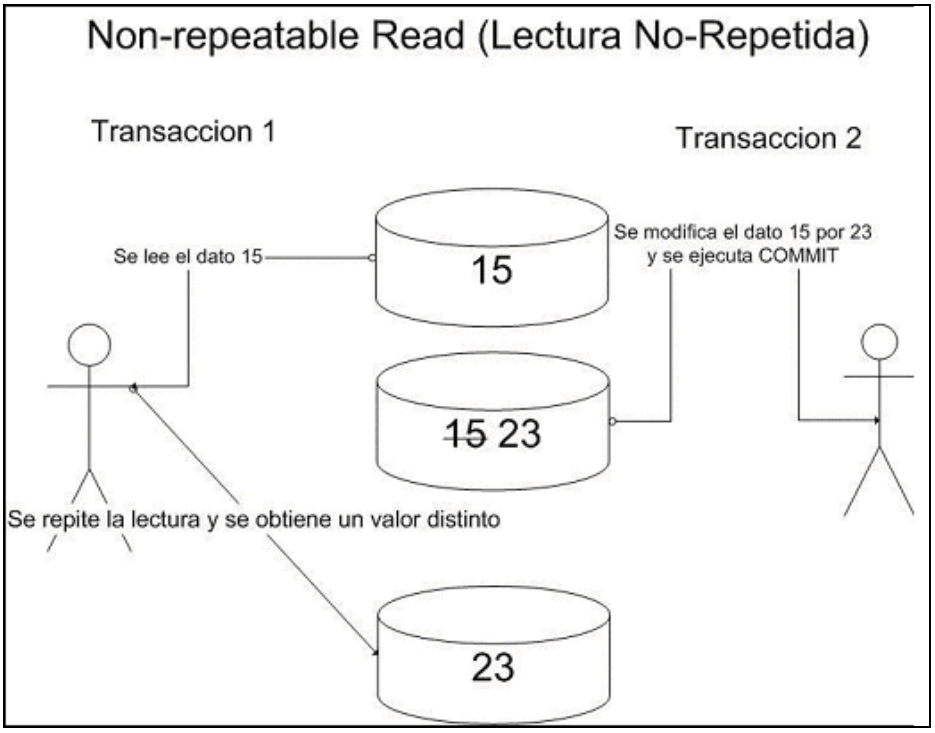
Niveles de aislamiento ANSI SQL

El ANSI SQL define los niveles de aislamiento en base a tres fenómenos que no deberían ocurrir, estos son:

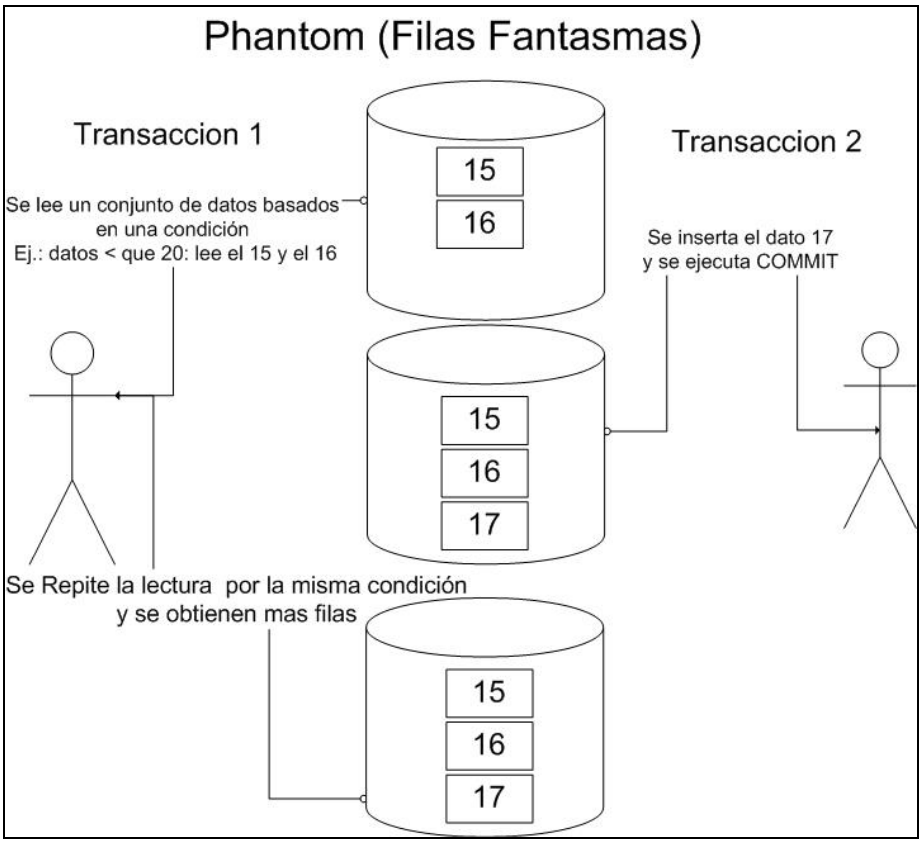
P1 (Dirty Read): La transacción T1 modifica un ítem de datos. Otra transacción T2 lee el mismo ítem de datos, antes que T1 realice un COMMIT o un ROLLBACK. Si T1 realiza luego un ROLLBACK, T2 habrá leído un ítem de datos que nunca fue comprometido y por lo tanto un valor que se considera que nunca debería haber existido.



P2 (Non-repeatable Read): La transacción T1 lee un ítem de datos. Otra transacción T2 modifica o borra el mismo ítem de datos y realiza COMMIT. Si T1 intenta releer el ítem de datos, encuentra que el ítem de datos tiene un valor distinto o ni siquiera existe ya que ha sido borrado.



P3 (Phantom): La transacción T1 lee un conjunto de ítems datos que satisface algún predicado P (condición de búsqueda). La transacción T2 crea o modifica un ítem de datos que satisface el predicado P de T1 y realiza COMMIT. Si T1 repite su lectura con el mismo predicado P, obtiene un conjunto de ítem de datos distinto de la primera lectura.



Ninguno de estos fenómenos pueden ocurrir en una historia serial. Por lo tanto por el teorema de Seriabilidad no pueden ocurrir en una historia serializable.

ANSI SQL define cuatro niveles de aislamiento, cada uno de ellos esta caracterizado por el fenómeno que las transacciones concurrentes (ejecutándose en ese nivel) no pueden experimentar. Sin embargo, la especificación dada por ANSI SQL no define el nivel de aislamiento SERIALIZABLE solamente en términos de estos fenómenos, sino que define que el nivel de aislamiento SERIALIZABLE debe proveer lo que se conoce como ejecución puramente serializable. La tabla definida en ANSI

que relaciona los fenómenos con los niveles de aislamiento, puede llevar a la idea falsa de que evitar los 3 fenómenos implica seriabilidad. Se denominan a las historias que evitan los fenómenos como ANOMALY SERIALIZABLE.

Isolation Level	P1 - Dirty Read	P2 – Fuzzy Read	P3 – Phantom
ANSI READ UNCOMMITTED	Posible	Posible	Posible
ANSI READ COMMITTED	Imposible	Posible	Posible
ANSI REPEATABLE READ	Imposible	Imposible	Posible
ANOMALY SERIALIZABLE	Imposible	Imposible	Imposible

Niveles de Aislamiento ANSI SQL definidos en términos de los tres fenómenos

***Nota**: Fuzzy Read (Lectura Borrosa) es también conocido como Non-Repeatable Read*

Reformulación de los fenómenos ANSI

Si de los 3 fenómenos ANSI explicados anteriormente se toma la interpretación estricta dada por ANSI SQL 92 o ANSI SQL 99 existirían historias obviamente no serializables que no están contempladas por estos fenómenos. Se presenta una interpretación más amplia que se supone que es lo que quiere definir ANSI.

Ejemplo de porque es necesario reformular la definición de los P1, P2 y P3:

T1	T2
Read X = 50	
Write X = 10	
	Read X = 10
	Read Y = 50
	COMMIT
Read Y = 50	
Write Y = 90	
COMMIT	

La transacción T1 transfiere $40 de la cuenta X a la cuenta Y manteniendo el balance total en $100. Pero T2 lee un estado inconsistente, al permitírsele leer en medio de la transacción T1 y lee un balance total inconsistente de $60. Se denomina a esta historia, obviamente no serializable como H1.

H1 no viola ninguno de los fenómenos debido a que:

- En el caso de P1 (Dirty Read), una de las transacciones tendría que abortar y esto no sucede.

- En el caso de P2 (Non Repeatable Read), un ítem de datos debería ser releído y esto tampoco sucede.
- Para el caso de P3 (Phantom), requiere que una transacción cambie algún dato cumpliendo el predicado P de alguna otra transacción concurrente y esto obviamente tampoco sucede.

Por lo tanto H1 no viola ninguno de los fenómenos definidos en ANSI.

Considerar la siguiente definición más amplia para Dirty Read:

P1 (Dirty Read): La transacción T1 modifica un ítem de datos. Otra transacción T2 lee el mismo ítem de datos, antes que T1 realice un COMMIT o un ROLLBACK. Y si T1 realiza luego un ROLLBACK *o un COMMIT y T2 realiza un ROLLBACK o un COMMIT en cualquier orden*, T2 habrá leído un ítem de datos que nunca fue comprometido y por lo tanto un valor que se considera que nunca debería haber existido.

P1: w1[x]...r2[x]...(c1 o a1) y (c2 o a2) en cualquier orden)

La historia H1 del ejemplo violaría esta definición de Dirty Read. Por lo que se supone que esta es la definición que quiso dar ANSI.

Ahora, analizar la siguiente historia H2:

T1	T2
Read X = 50	
	Read X = 50
	Write X = 10
	Read Y = 50
	Write Y = 90
	COMMIT
Read Y = 90	
COMMIT	

H2 obviamente no es serializable, dado que la Transacción 1 ve un estado inconsistente en donde encuentra un balance de 140, dado que la Transacción T2 cambio los valores de X e Y manteniendo el balance en 100 pero T1 leyó un valor antes de los cambios y otro valor después.

En H2 no se produce Dirty Read ya que ninguna de las dos transacciones lee datos sin comprometer. Tampoco se produce P2 y P3 como están definidos según la interpretación estricta de ANSI.

Considerar la siguiente definición más amplia para Non-repeatable Read:

P2 (Non-repeatable Read): La transacción T1 lee un ítem de datos. Otra transacción T2 modifica o borra el mismo ítem de datos. Cualquiera de las dos transacciones realiza COMMIT o ROLLBACK.

P2: r1[x]...w2[x]...((c1 o a1) y (c2 o a2) en cualquier orden)

***Nota**: El fenómeno que se produce en H2 en realidad se conoce como Read Skew y esta definida más adelante. Las historias que evitan Non Repeatable Read deberían evitar Read Skew.*

Finalmente analizar la siguiente historia H3:

T1	T2
Read valores<10	
	Write X = 5
	Read Y = 50
	Write Y = 90
	COMMIT
Read Y = 90	
COMMIT	

H3 es similar a H2, con la diferencia de que la transacción 1 en vez de leer un valor concreto lee un predicado y T2 en vez de modificar este valor modifica el predicado. Al igual que antes habría que reformular P3 de la siguiente manera:

P3 (Phantom): La transacción T1 lee un conjunto de ítems de datos que satisface algún predicado P (condición de búsqueda). La transacción T2 crea o modifica un ítem de datos que satisface el predicado P de T1.

P3: r1[P]...w2[y en P]...((c1 o a1) y (c2 o a2) en cualquier orden)

Dirty Write (P0)

El ANSI SQL excluye un fenómeno en su definición basada en fenómenos; este se conoce como **Dirty Write** y es evitado por los schedulers basados en bloqueos cuando en su nivel mas bajo, READ UNCOMMITED, los escritores se bloquean entre si.

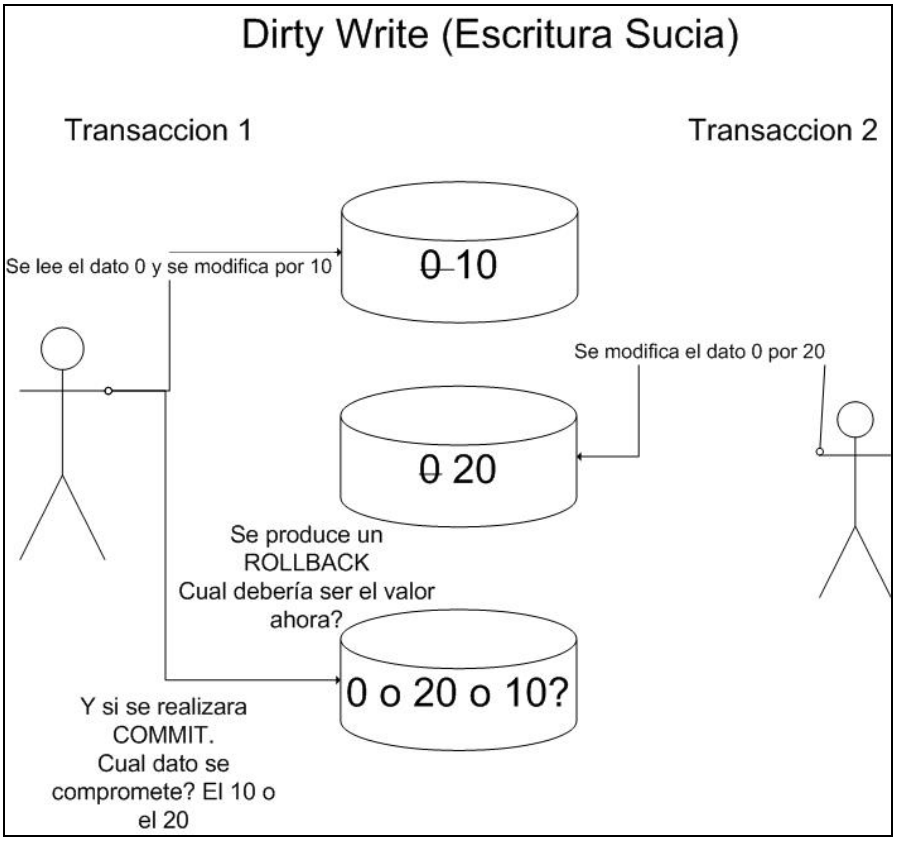
El fenómeno se produce cuando la transacción T1 modifica un ítem de datos. Otra transacción T2 mas tarde modifica el mismo ítem de datos antes que T1 realice un

COMMIT o un ROLLBACK. Si T1 o T2 realizan un ROLLBACK, no queda claro cual debería ser el valor correcto del dato.

Ejemplo:

T1	T2
X = 0	
Write X = 10	
	Write X = 20
Rollback	
Cuanto vale X?	

¿Luego del Rollback, X cuanto valdría? ¿0 o 20? No esta determinado cual debería ser el valor de X



El ANSI SQL debería modificar su definición de niveles de aislamiento para que todos éstos eviten el fenómeno de Dirty Write.

Isolation Level	P0 - Dirty Write	P1 – Dirty Read	P2 – Fuzzy Read	P3 – Phantom
READ UNCOMMITTED	Imposible	Posible	Posible	Posible
READ COMMITTED	Imposible	Imposible	Posible	Posible
REPEATABLE READ	Imposible	Imposible	Imposible	Posible
SERIALIZABLE	Imposible	Imposible	Imposible	Imposible

Niveles de Aislamiento ANSI SQL definidos en términos de 4 fenómenos

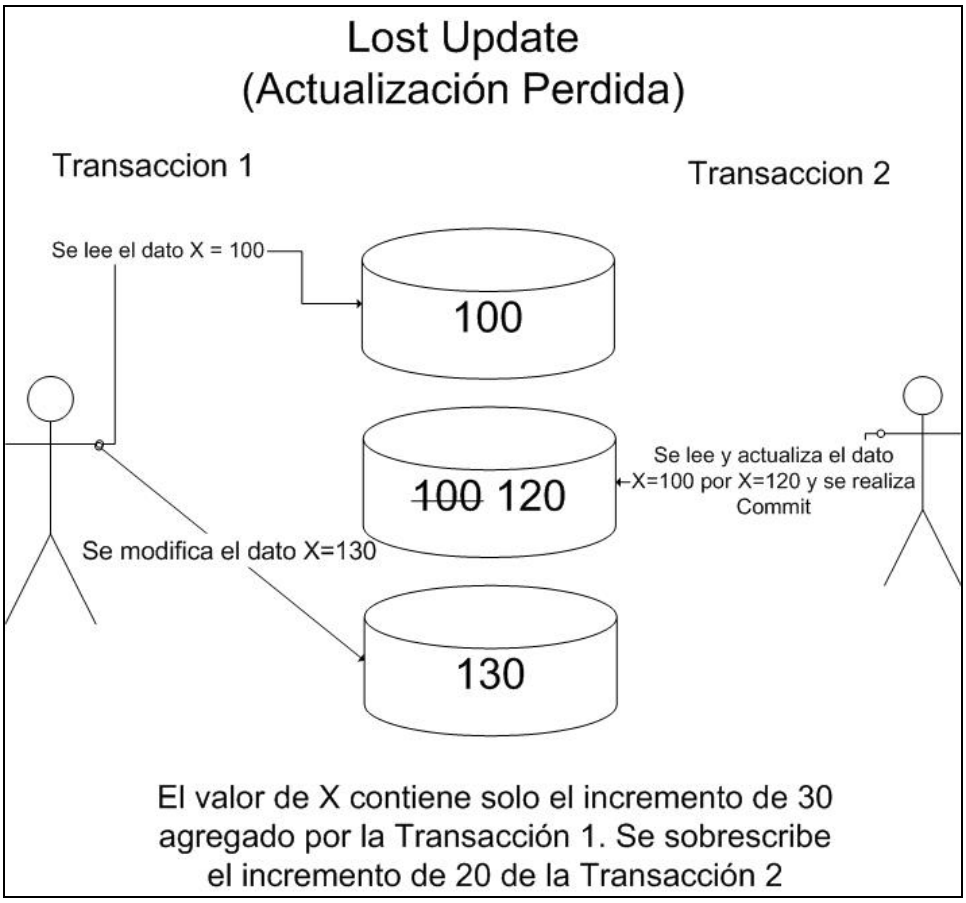
Lost Update (P4):

Existe otro fenómeno a tener en cuenta, este se denomina **Lost Update (P4).** El fenómeno Lost Update ocurre cuando una transacción T1 lee un ítem de datos, luego una transacción T2 actualiza ese mismo ítem de datos, posiblemente basado en una lectura previa, entonces T1 basado en la lectura que realizó anteriormente actualiza el ítem de datos y realiza COMMIT.

Si se analiza una historia H1 ilustrando el problema, se observa que es independiente si T2 realiza o no COMMIT.

T1	T2
Read X=100	
	Read X=100
	Write X=120
	COMMIT
Write X=130	
COMMIT	

El valor final de X contiene solo el incremento de 30 agregado por T1.



El nivel de aislamiento **CURSOR STABILITY** esta diseñado para prevenir este fenómeno.

Se utiliza la siguiente definición para comparar niveles de aislamiento:

El nivel de aislamiento L1 es más **débil** que el nivel de aislamiento L2 (o L2 más **fuerte** que L1) y se denota como L1 << L2, si todas las historias no-serializables que obedecen el criterio de L2 también satisfacen el de L1 y hay al menos 1 historia no-serializable que ocurre en el nivel L1 pero no puede ocurrir en el nivel L2. Se dice que dos niveles de aislamiento son **equivalentes**, y se denota como L1 = = L2, cuando los conjuntos de historias no-serializables que satisfacen L1 y L2 son idénticos. Dos niveles de aislamiento son **incomparables**, denotado L1>> <<L2, cuando uno de los niveles de aislamiento permite historias no-serializables que el otro no permite.

En base a lo definido anteriormente se puede decir que los niveles ANSI cumplen con lo siguiente:

READ UNCOMMITTED >> READ COMMITTED >> REPEATABLE READ >> SERIALIZABLE

Comparación del nivel de aislamiento Cursor Stability con los ya definidos

El **fenómeno P4 (Lost Update)**, es posible en el nivel de aislamiento READ COMMITED, dado que en la historia del ejemplo anterior se prohíbe P0 (Dirty Write) y P1 (Dirty Read), sin embargo prohibir P2 (Non Repeatable Read) también prohibiría p4 (Lost Update) ya que el Write X de la transacción T2 se realiza después del Read X de la transacción T1 y antes que T1 se comprometa o aborte (situación prohibida en P2). Por lo tanto el fenómeno P4 es útil para distinguir niveles de aislamiento intermedios entre READ COMMITED y REPEATABLE READ.

Teniendo en cuenta el fenómeno P4 y el nivel de aislamiento originado para evitarlo, CURSOR STABILITY, se dice que:

READ COMMITTED >> CURSOR STABILITY >>REPEATABLE READ

Violación de Restricción de Item de Datos (P5)

(Referencias: [9], [10] y [20])

Existe un tipo de comportamiento anómalo importante en la ejecución de transacciones concurrentes, llamado **violación de restricción** (Constraint Violation). Las bases de datos satisfacen restricciones sobre múltiples ítems de datos (unicidad de las claves, integridad referencial, replicación de filas en dos tablas, etc.). Todas estas restricciones juntas forman el predicado de restricción invariante de la base de datos, C(DB). El invariante es verdadero si el estado de la base de datos es consistente con las restricciones y es falso de otro modo. Las transacciones deben preservar el predicado de restricciones para mantener consistencia, si la base de datos es consistente cuando la transacción comienza, la base de datos debería ser consistente cuando la transacción se compromete. Si una transacción lee un estado de la base de datos que viola el predicado de restricciones, entonces la transacción sufre una violación de restricción en la ejecución concurrente. Estas violaciones de restricción se denominan análisis inconsistentes.

Teniendo en cuenta este comportamiento anómalo se define el siguiente fenómeno como **Violación de Restricción de Ítem de Datos (P5, Data Item Constraint Violation).**

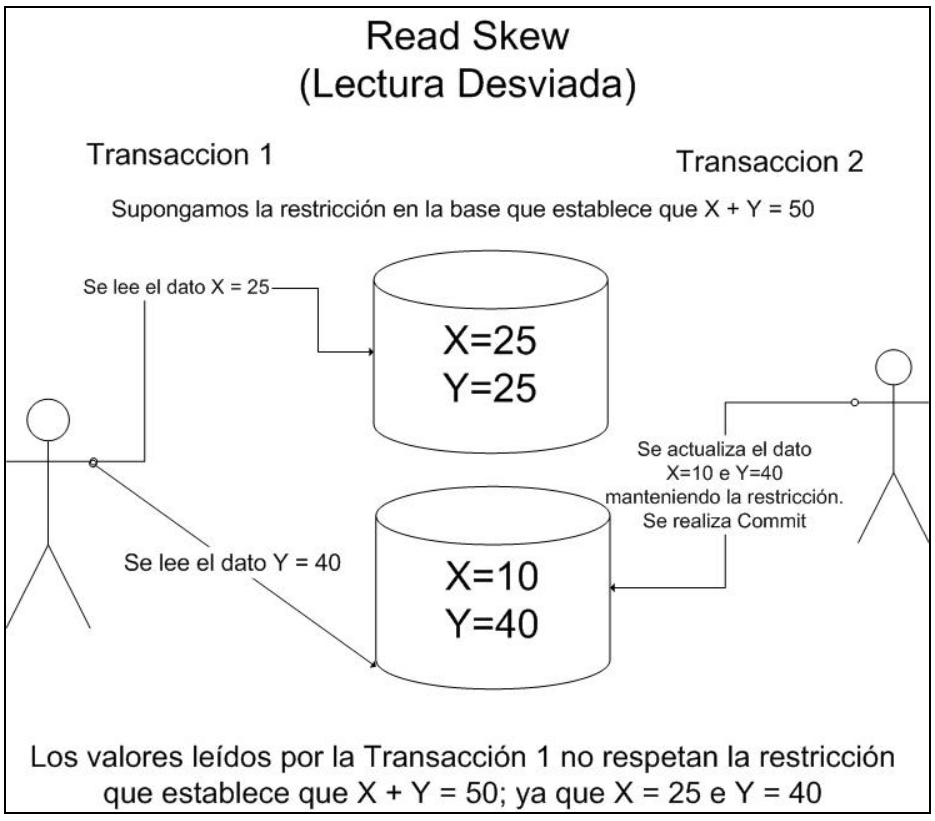
Se establece C( ) como una restricción de base de datos entre los ítems de datos X e Y. Se definen dos fenómenos que surgen de la violación de restricción (P5):

P5A Read Skew (Lectura Desviada)

Se establece que la transacción T1 lee el ítem de datos X, y una segunda transacción T2 actualiza X e Y a nuevos valores y se compromete. Si ahora T1 lee Y, es posible que vea un estado inconsistente, y por lo tanto produzca una estado inconsistente como salida.

T1	T2
Read X	
	Write X
	Write Y
	COMMIT
Read Y	
Commit o Abort	

Se establece que X + Y deberían ser igual a 50 de acuerdo a alguna regla de negocio existente en nuestra base. Si T1 lee que la X vale 25 y luego T2 modifica los valores de X e Y en 10 y 40 manteniendo la consistencia de que la suma valga 50; cuando T1 lea el dato Y encontrara que este vale 40 y la suma con X que valía 25 seria de 65 teniendo una inconsistencia. Esto se debe a que la lectura de la X es obsoleta y por lo tanto se produce Read Skew.



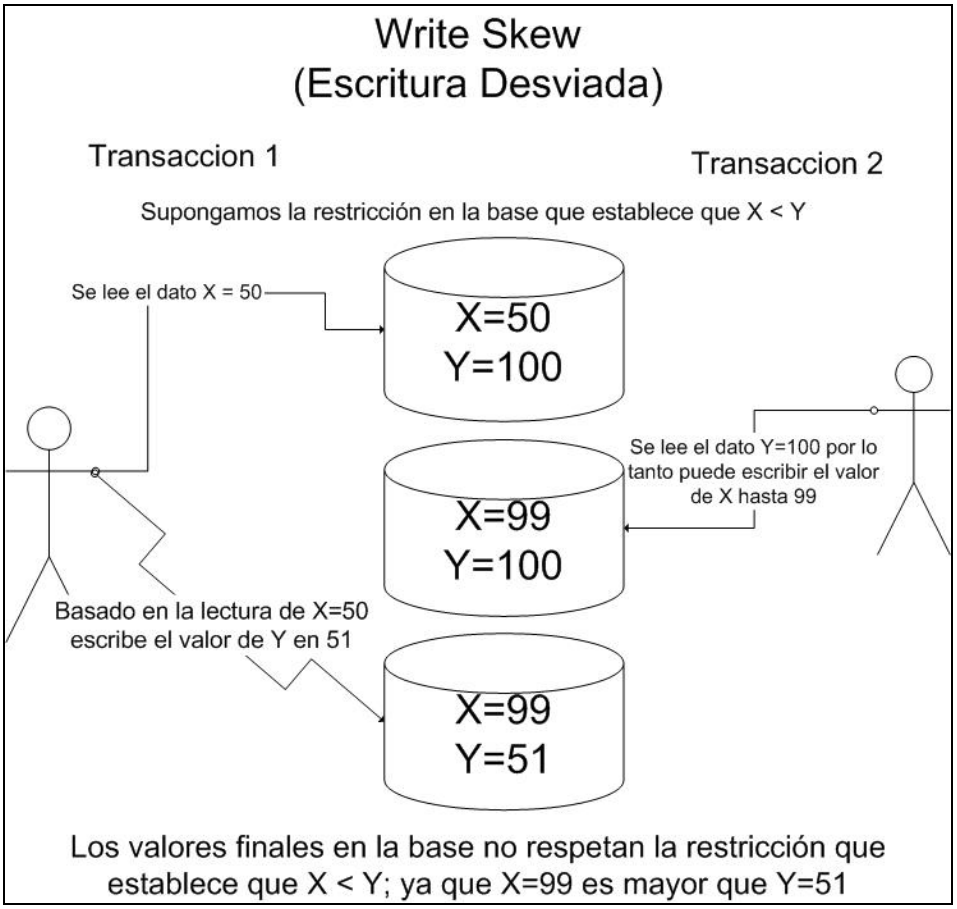
El fenómeno P2, Non Repeatable Read, es una forma degenerada de Read Skew donde x = y.

P5B Write Skew (Escritura Desviada)

Una transacción T1 lee el ítem de datos X y el ítem de datos Y, los cuales son consistentes con C(), y otra transacción T2 lee X e Y, luego escribe X y se compromete. Entonces T1 escribe Y. Si existe una restricción entre X e Y, ésta podría ser violada.

T1	T2
Read X	
	Read Y
	Write X
Write Y	
Commit	Commit

Se define que la restricción de la base de datos es que X tiene que ser menor que Y, e inicialmente X = 50 e Y = 100. Si la transacción 1 lee que el valor de X vale 50 y podría querer escribir el valor de Y en 51 o mas, la transacción 2 lee que el valor de Y vale 100 podría querer escribir el valor de X hasta 99, por lo tanto podrían violar la restricción y valer la X = 99 y la Y = 51.



P5A y P5B no pueden surgir en historias donde P2 esta prohibido, dado que en P5A y P5B esta presente la transacción T2 que escribe un ítem de datos que ha sido previamente leído por T1 que todavía no se comprometió. Los fenómenos P5A y P5B surgen en los niveles de aislamiento que son más débiles que REPEATABLE READ.

Nivel de Aislamiento SNAPSHOT
(Referencias: [7] ,[9] y [20])

Existe otro nivel de aislamiento muy importante, ya que es el nivel de aislamiento de los controles de concurrencia multiversión. Este nivel de aislamiento se denomina **SNAPSHOT**.

Una transacción ejecutando en este nivel siempre lee los datos de un snapshot (vista) como se encontraban al comienzo de la transacción. Una transacción ejecutándose en un nivel de aislamiento SNAPSHOT nunca se bloquea cuando intenta leer. Las escrituras de esta transacción se verán reflejadas en su propio snapshot si la transacción accede nuevamente a esos datos modificados. Las actualizaciones por otras transacciones concurrentes activas luego de que comenzó la transacción son invisibles a esta. *(Ver capitulo 4)*

El nivel de aislamiento SNAPSHOT, como se mencionó anteriormente, es el nivel de los controles de concurrencia multiversión, por lo que las historias con una única versión para cada dato no reflejan apropiadamente la secuencia de operaciones. En cualquier momento, cada ítem de datos podría tener varias versiones creadas por transacciones activas y/o comprometidas. Las lecturas de una transacción deben elegir la versión correcta del ítem de datos a leer.

Las historias generadas por la concurrencia multiversión, se denominan historias multivaluadas, como se describió en el capitulo 4, en el control de concurrencia multiversión existe un posible mapeo entre historias multivaluadas e historias simples, este mapeo permite ubicar al Snapshot Isolation en la jerarquía de niveles de aislamientos.

Comparación del nivel de aislamiento Snapshot con los ya definidos.

Snapshot Isolation es bastante fuerte en cuanto a consistencia, incluso más fuerte que Read Committed. En el nivel de aislamiento Snapshot se evitan los fenómenos P0 (Dirty Writes) y P1 (Dirty Reads) por lo tanto no puede ser más débil que Read Committed. Además P5A (Read Skew) es posible en Read Committed pero no en Snapshot Isolation, por lo tanto:

SNAPSHOT >> READ COMMITTED

Es difícil mostrar historias del nivel de aislamiento Snapshot con el fenómeno P2 en una interpretación de un único valor (recordar el mapeo existente entre historias multivaluadas y de un único valor). Sin embargo P5B (Write Skew) obviamente puede ocurrir en las historias del nivel de aislamiento Snapshot y en la interpretación de un único valor de estas historias. Pero P2 prohíbe P5B. Por lo tanto Snapshot admite historias anómalas que REPEATABLE READ no permite.

En el nivel de aislamiento Snapshot no se puede experimentar el fenómeno P3 (Phantom). Una transacción releyendo un predicado P después de la actualización de otra transacción siempre verá el mismo conjunto de datos (los datos al comienzo de la misma). Pero REPEATABLE READ puede experimentar el fenómeno P3. Por lo tanto Snapshot no admite historias que REPEATABLE READ permite.

REPEATABLE READ >> << SNAPSHOT

Lo más interesante del nivel de aislamiento Snapshot, es que en Snapshot no se producen filas fantasmas. Cada transacción nunca ve las actualizaciones de las transacciones concurrentes. Por lo tanto recordando la tabla 1 donde se definía ANOMALY SERIALIZABLE por ANSI SQL 92 se puede decir que:

SNAPSHOT >> ANOMALY SERIALIZABLE

El nivel de aislamiento Snapshot tiene varias implementaciones comerciales de bases de datos multiversión con la característica de “el primero que se compromete gana”, el Snapshot explicado anteriormente. Esta característica requiere que el sistema recuerde todas las actualizaciones a lo largo de cualquier transacción que se compromete después del comienzo de cada transacción activa. Se aborta la transacción si una actualización entra en conflicto con las actualizaciones recordadas de las otras transacciones.

Existen otros modelos de control de concurrencia multiversión.
Oracle provee un nivel de consistencia que da a cada sentencia SQL la versión mas reciente comprometida de la base de datos al instante en que comenzó la sentencia.

Esto es como si el tiempo de comienzo de la transacción se actualizará con cada sentencia. Los miembros de un cursor son los del instante en que el se realiza el Open del cursor. El mecanismo de este control de concurrencia recalcula la versión apropiada con el comienzo de cada sentencia. La inserción, actualización y borrado esta cubierta por los bloqueos de escritura que proporcionan una política de “el primero que escribe gana” antes que “el primero que compromete gana”. Este control de concurrencia provisto por Oracle y denominado Consistencia de Lectura, es más fuerte que READ COMMITTED pero permite lecturas no repetidas (P2), lost update (P4)y read skew (P5A).

Oracle también provee "consistencia a nivel de transacción" asignando el nivel de aislamiento como Serializable, aunque en realidad es análogo al nivel de aislamiento Snapshot. La diferencia entre el Snapshot implementado por Oracle y el explicado anteriormente es que en Oracle "el primero que actualiza gana" mientras que en Snapshot "el primero que compromete gana". Ver Anexo 1 Capitulo 5.3 y Capitulo 4 de ésta tesis.

Tipos de Niveles de Aislamiento caracterizados por las posibles anomalías permitidas

(Referencias: [9])

Nivel de Aislamiento	P0 Dirty Write	P1 Dirty Read	P4 Lost Update	P2 Fuzzy Read	P3 Phantom	P5A Read Skew	P5B Write Skew
READ UNCOMMITTED	No Posible	Posible	Posible	Posible	Posible	Posible	Posible
READ COMMITTED	No Posible	No Posible	Posible	Posible	Posible	Posible	Posible
CURSOR STABILTY	No Posible	No Posible	A veces Posible	A veces Posible	Posible	Posible	A veces Posible
REPEATABLE READ	No Posible	No Posible	No Posible	No Posible	Posible	No Posible	No Posible
SNAPSHOT	No Posible	No Posible	No Posible	No Posible	A veces Posible	No Posible	Posible
ANSI SQL SERIALIZABLE	No Posible	No Posible	No Posible	No Posible	No Posible	No Posible	No Posible

En Oracle
(Referencias: Anexo I)

Nivel de Aislamiento	P0 Dirty Write	P1 Dirty Read	P4 Lost Update	P2 Fuzzy Read	P3 Phantom	P5A Read Skew	P5B Write Skew
READ UNCOMMITTED	NO OFRECE ESTE NIVEL DE AISLAMIENTO						
READ COMMITTED	No Posible	No Posible	Posible	Posible	Posible	Posible	Posible
CURSOR STABILTY	NO OFRECE ESTE NIVEL DE AISLAMIENTO						
REPEATABLE READ	NO OFRECE ESTE NIVEL DE AISLAMIENTO						
SNAPSHOT (ORACLE SERIALIZABLE)	No Posible	No Posible	No Posible	No Posible	No Posible	No Posible	Posible
ANSI SQL SERIALIZABLE	NO OFRECE ESTE NIVEL DE AISLAMIENTO						

En SQL Server 2000
(Referencias: Anexo 1)

Nivel de Aislamiento	P0 Dirty Write	P1 Dirty Read	P4 Lost Update	P2 Fuzzy Read	P3 Phantom	P5A Read Skew	P5B Write Skew
READ UNCOMMITTED	No Posible	Posible	Posible	Posible	Posible	Posible	Posible
READ COMMITTED	No Posible	No Posible	Posible	Posible	Posible	Posible	Posible
CURSOR STABILTY	NO OFRECE ESTE NIVEL DE AISLAMIENTO						
REPEATABLE READ	No Posible	No Posible	No Posible	No Posible	Posible	No Posible	No Posible
SNAPSHOT	NO OFRECE ESTE NIVEL DE AISLAMIENTO						
ANSI SQL SERIALIZABLE	No Posible	No Posible	No Posible	No Posible	No Posible	No Posible	No Posible

Conclusión

Si se realiza un análisis para poder determinar en que casos conviene utilizar cada tipo de nivel de aislamiento, podemos decir:

READ UNCOMMITTED es conveniente utilizarlo en los casos en los cuales se necesitan resultados rápidos, sin bloquear transacciones, y cuando no se necesita que los datos sean precisos ni que estén actualizados. Cuando se necesita el acceso a datos precisos y actualizados sin bloquear transacciones y no se puede tener una réplica desconectada de la base de datos entonces se debe utilizar otro nivel de aislamiento.

READ COMMITTED con bloqueos es conveniente utilizarlo en los casos en los cuales se puedan utilizar los bloqueos para serializar el acceso a datos, y en entornos con transacciones con bloqueos de corta duración.

Por otro lado, no es conveniente usarlo cuando las demoras por bloqueos son demasiado largas ya que pueden llevar a que a las transacciones les expire su tiempo de espera y aumentar la probabilidad de deadlock. Además los datos pueden cambiar mientras se ejecutan una serie de consultas que esperan datos consistentes (non repeatable read).

Existen varias técnicas para evitar los problemas que existen utilizando el nivel de aislamiento READ COMMITTED:

- Hacer una réplica de solo lectura de los datos con propósitos de realizar reportes.

- Cambiar al nivel de aislamiento READ UNCOMMITTED para evitar la espera por bloqueos.

- Cambiar al nivel de aislamiento REPEATABLE READ/SERIALIZABLE en una sola transacción para la consulta que necesite el reporte para evitar que cambien los datos mientras se realiza el proceso.

- Utilizar MVCC al nivel de aislamiento REPEATABLE READ/SERIALIZABLE para evitar bloqueos y que cambien los datos mientras se realiza el proceso.

READ COMMITED con MVCC es conveniente utilizarlo cuando la carga de trabajo tiene una mezcla de actualizaciones y lecturas de larga duración. Además tiene la característica de no bloquear lectores con escritores y "trabajar" con una vista consistente de la base de datos, al contrario del comportamiento que tiene el nivel de aislamiento READ UNCOMMITTED en donde se trabaja con datos que quizás nunca se comprometan y que pueden mostrar vistas inconsistentes de la base de datos. Se utiliza para los casos en que las aplicaciones quieren consistencia a nivel de sentencia y no a nivel de transacción.

Un punto en contra de RC con MVCC es que los datos pueden cambiar mientras se están leyendo lo cual produce una vista inconsistente de la base de datos. Existen dos niveles de aislamiento que evitan éste inconveniente y son REPEATABLE READ (que bloquea los datos que se leen dentro de la transacción) y SERIALIZABLE (que bloquea los conjuntos de datos que se leen dentro de la transacción, es decir los que cumplen con un predicado). Estos dos niveles de aislamiento no son buenos en escenarios donde hay combinaciones de actualizaciones y lecturas de larga duración y el panorama empeora a medida que aumentan los bloqueos.

REPEATABLE READ con bloqueos es conveniente usarlo cuando la aplicación requiere precisión absoluta en transacciones largas y de varias sentencias y debe mantener todos los datos bloqueados hasta que ésta transacción termine. La aplicación necesita consistencia para todos los datos que se leen repetidamente dentro de la transacción y que otras transacciones no los modifiquen. Esto puede impactar en la concurrencia de un sistema multiusuario si existen transacciones que intentan modificar datos que han sido bloqueados por un lector.

SERIALIZABLE con MVCC es conveniente utilizarlo cuando se necesita que los datos que se leen sean consistentes aunque haya actualizaciones realizándose y comprometiéndose mientras se realizan las lecturas. Esta consistencia se logra sin tener los efectos negativos que se producen utilizando los niveles de aislamiento READ COMMITTED y SERIALIZABLE. Como READ COMMITTED se aplica a nivel de sentencia, no existe "memoria" a través de las sentencias.

SERIALIZABLE con bloqueos es conveniente utilizarlo cuando la aplicación necesita precisión absoluta en transacciones largas de varias sentencias y deben bloquear toda la información involucrada para que no la modifique otra transacción hasta que ésta termine. Los datos que se deben bloquear no son sólo los que ya se han

leído dentro de la transacción sino también los posibles datos nuevos que cumplan con el predicado. Este nivel se debe tener en cuenta cuando se quiere trabajar con datos consistentes y precisos y se planea modificar datos que fueron leídos antes dentro de la misma transacción.

6 – APLICACIÓN LECTORES / ESCRITORES

Objetivo

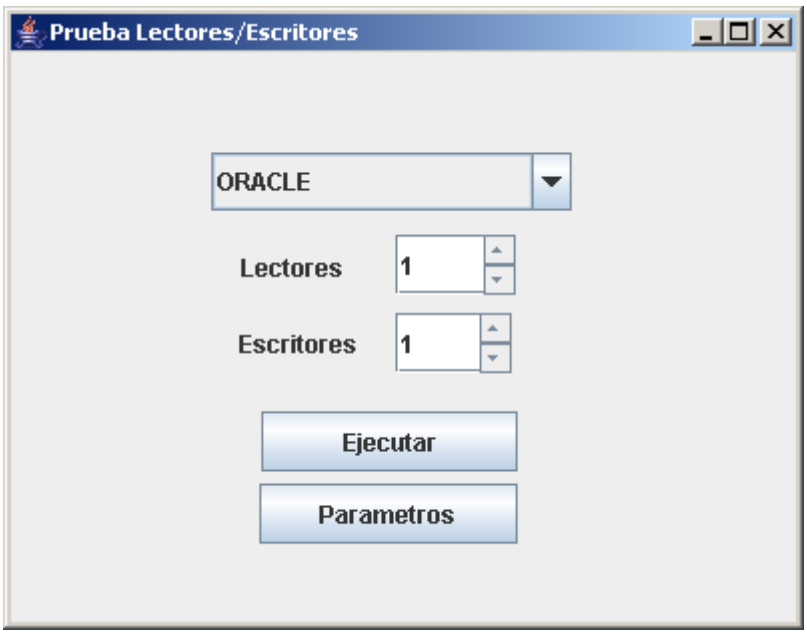
El objetivo es crear una aplicación para mostrar y comparar como se resuelven en los distintos productos comerciales los problemas de concurrencia entre lectores y escritores en una base de datos.

La idea principal de esta aplicación es visualizar la concurrencia que permiten los distintos modelos que utilizan las bases de datos en el mercado actual.

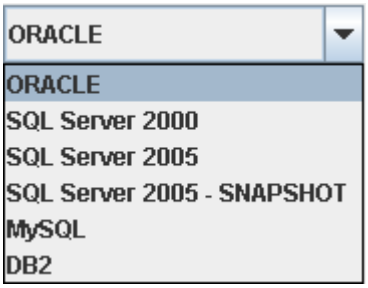
De ésta manera, se pueden comparar los modelos de concurrencia manejados por **Bloqueos** como el que implementa SQL Server y DB2 y el modelo de **Múltiples Versiones** como el que implementa Oracle, PostgreSQL y MySql (en este último si se usan tablas tipo InnoDB).

Explicación de los componentes de la Aplicación

Pantalla principal de la aplicación:



Se puede observar el siguiente Combo que permite elegir la Base de Datos con la cual se van a realizar las pruebas.

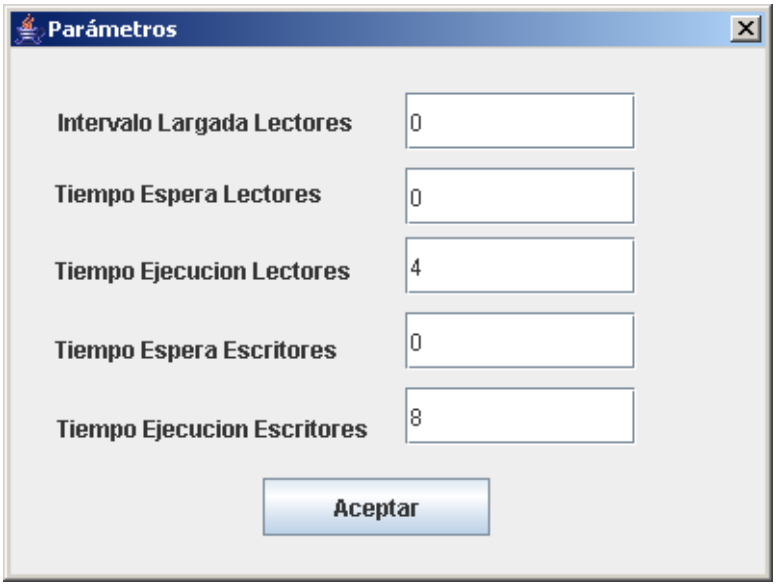


Debajo del Combo aparecen dos campos editables para elegir la cantidad de Lectores y Escritores:

Se puede tipear directamente una cantidad escribiendo el número en el campo o aumentar o disminuir la misma con las flechas ubicadas a la derecha del campo.

Los Botones que restan:

Se utilizan uno para ejecutar la prueba y otro para asignar varios parámetros. Haciendo click en este último se abre la ventana:



En esta ventana se pueden observar 5 campos editables:

Intervalo Largada Lectores: Indica el intervalo de comienzo de los Lectores.

Si se decide hacer la prueba con 4 lectores y se asigna el valor del campo Intervalo Largada Lectores a 3 segundos, el 1er lector comenzara en el instante 0, el 2do en el instante 3, el 3ero en el instante 6 y el 4to en el instante 9. Por la palabra "comenzar" se entiende intentar entrar a la base.

Solo se ofrece el Intervalo de Largada para Lectores y no para Escritores porque esta opción surge a partir de que los lectores pueden ejecutar concurrentemente entre si en casi todos los casos en los modelos de concurrencia de transacciones. Por lo tanto es lo mismo elegir cualquier cantidad de lectores ya sea 1 o 100, si estos comienzan juntos terminarán juntos. Distinto es el caso de los escritores donde los mismos no pueden ejecutar concurrentemente entre si.

Tiempo Espera Lectores: Es el tiempo que espera un Lector antes de intentar entrar en la base.

Tiempo Ejecución Lectores: Es el tiempo que durará un Lector dentro de la base.

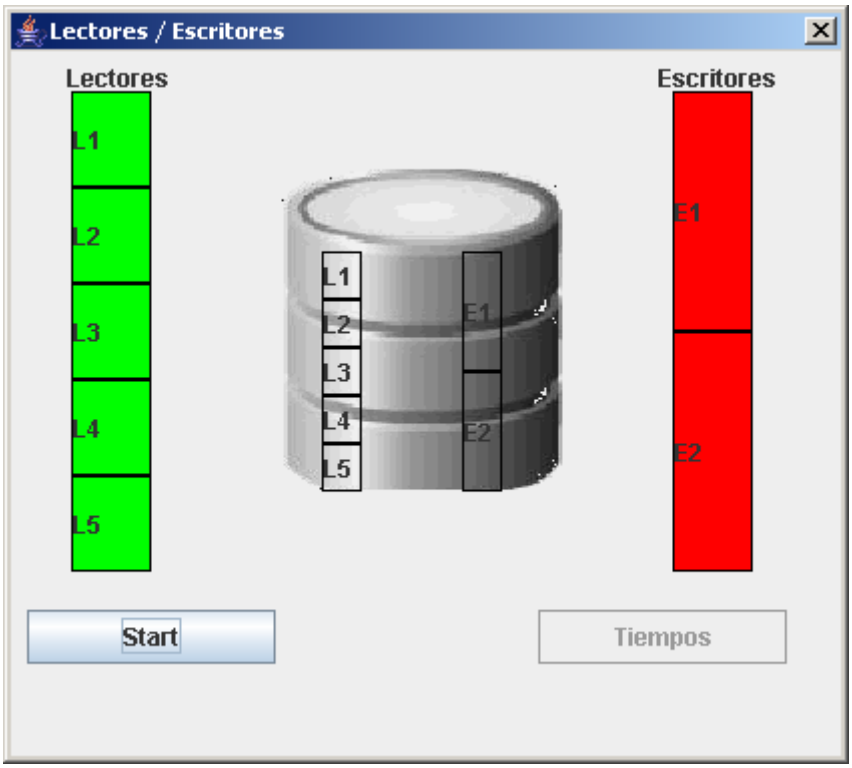
Tiempo Espera Escritores: Es el tiempo que espera un Escritor antes de intentar entrar en la base.

Tiempo Ejecución Escritores: Es el tiempo que durará el Escritor dentro de la base.

Todos los tiempos están en segundos**.**

Los tiempos de ejecución son los mismos independientemente de la base de datos elegida, ya que se obtienen mediante un bloqueo de escritura o de lectura según corresponda y realizando un sleep durante el valor indicado por el parámetro liberando el bloqueo luego de expirado el tiempo.

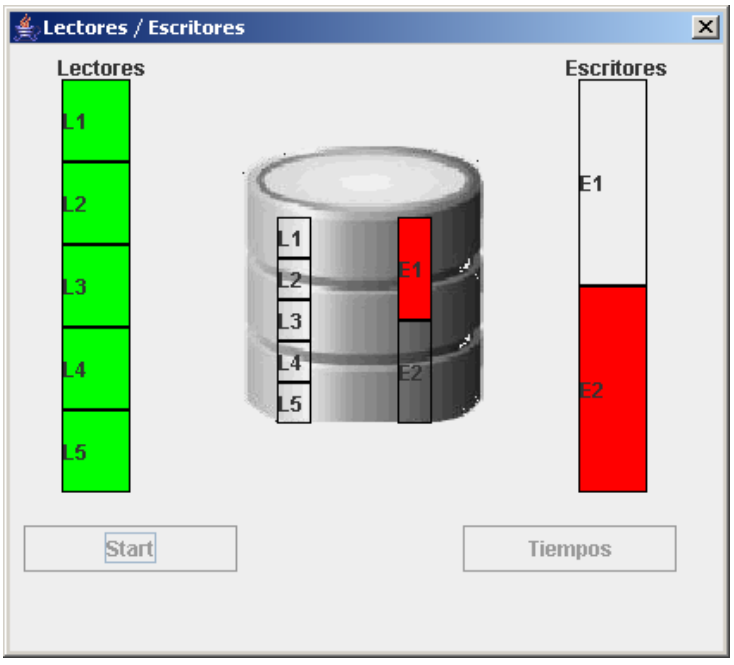
Una vez elegidos todos los parámetros, la cantidad de lectores, de escritores y seleccionada la base de datos para la cual se quiere realizar la prueba se hace click en el botón ejecutar y se abre la siguiente ventana:



La ventana de arriba es el resultado de haber seleccionado 5 Lectores que se muestran en color verde y 2 Escritores (en color rojo).

Para que arranque la prueba se debe hacer click en el botón **Start.**

En el instante en que se hace click los colores irán señalando los movimientos de lectores y escritores.



Por ultimo, al hacer click en el botón **Tiempos** se abre una nueva ventana, la cual muestra lo demorado por Lectores y Escritores de una forma grafica que ayuda a entender las diferencias entre los manejos de concurrencia de las bases de datos.

En la siguiente ventana se puede observar que el Escritor 1 (E1) se encuentra en la base con los 5 lectores, por lo tanto se puede deducir que este es un manejo MVCC en donde los lectores no bloquean a los escritores y viceversa.

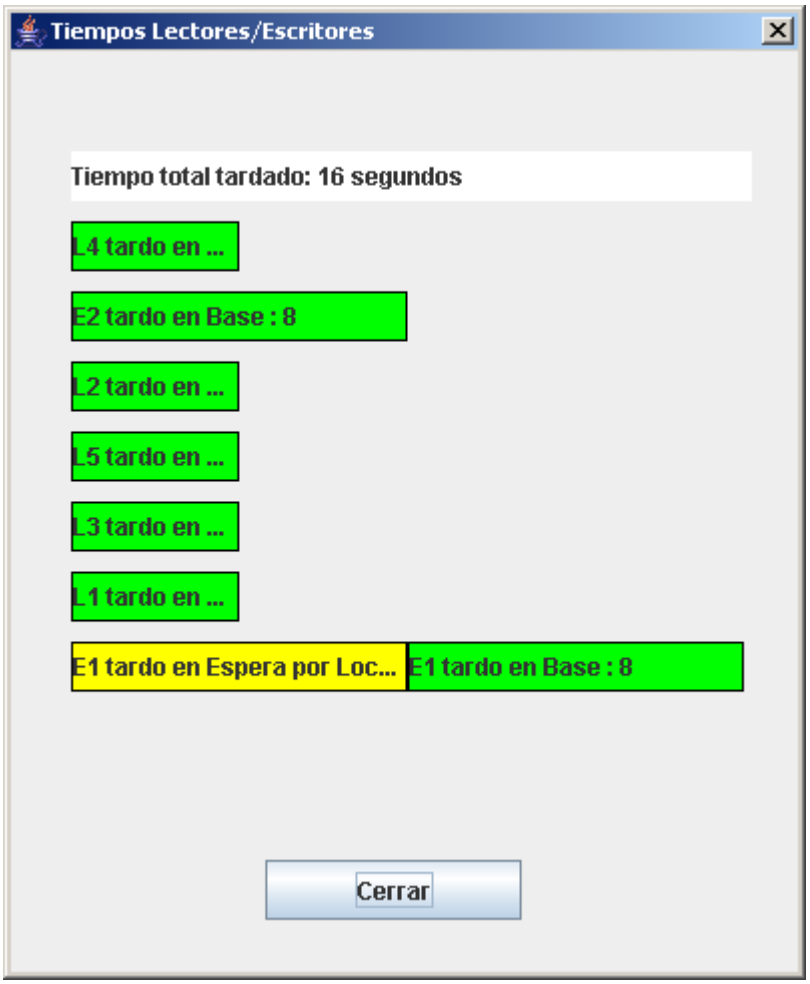
En color Rojo se marca el Tiempo de Espera por Parámetro. Este es el tiempo que tardan los participantes, tanto lectores como escritores, en intentar entrar a la base y está dado por lo indicado en la ventana de Parámetros mas el tiempo tardado por la aplicación en registrar el driver, generar la interfaz, etc.

En color Amarillo se muestra el tiempo demorado por un participante en entrar a la Base de Datos a causa de un bloqueo, es decir el tiempo utilizado por las distintas formas de manejar la concurrencia de cada modelo.

En el ejemplo se puede ver que el único tiempo "perdido" por bloqueos fue el del Escritor 2 (E2) que desde el primer instante quiso entrar a la base y no pudo porque se encontraba el otro escritor.

En color Verde se muestra el tiempo de demora dentro de la Base de Datos. Es el tiempo necesario para leer o escribir dependido de la operación que se trate.

Y por ultimo en color Blanco en la primera barra que se muestra arriba muestra el total empleado en el ejemplo tanto por los lectores como por los escritores para que éstos cumplan su tarea.



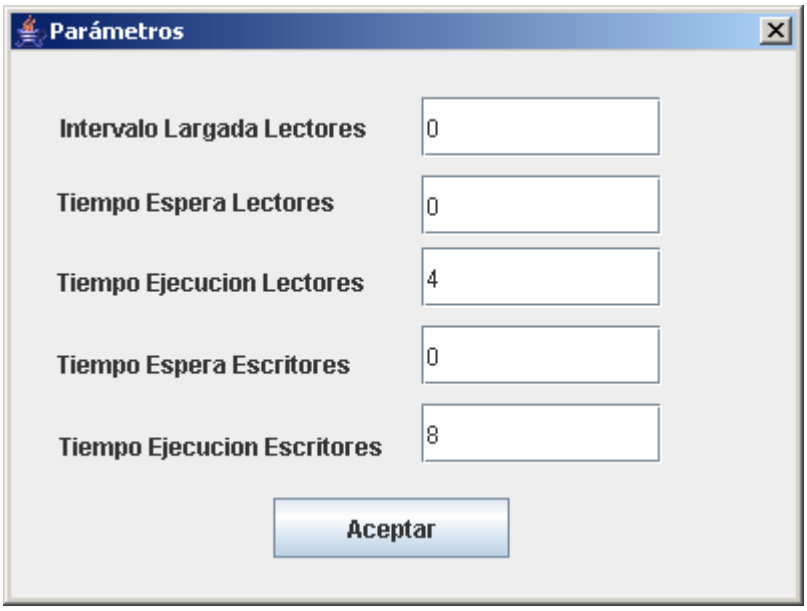
Ejemplo de Uso

Se realizaron dos pruebas, una seleccionando Oracle como base de datos, que usa MVCC para el manejo de la concurrencia y otra con SQL Server 2000 que usa manejo por Bloqueos.

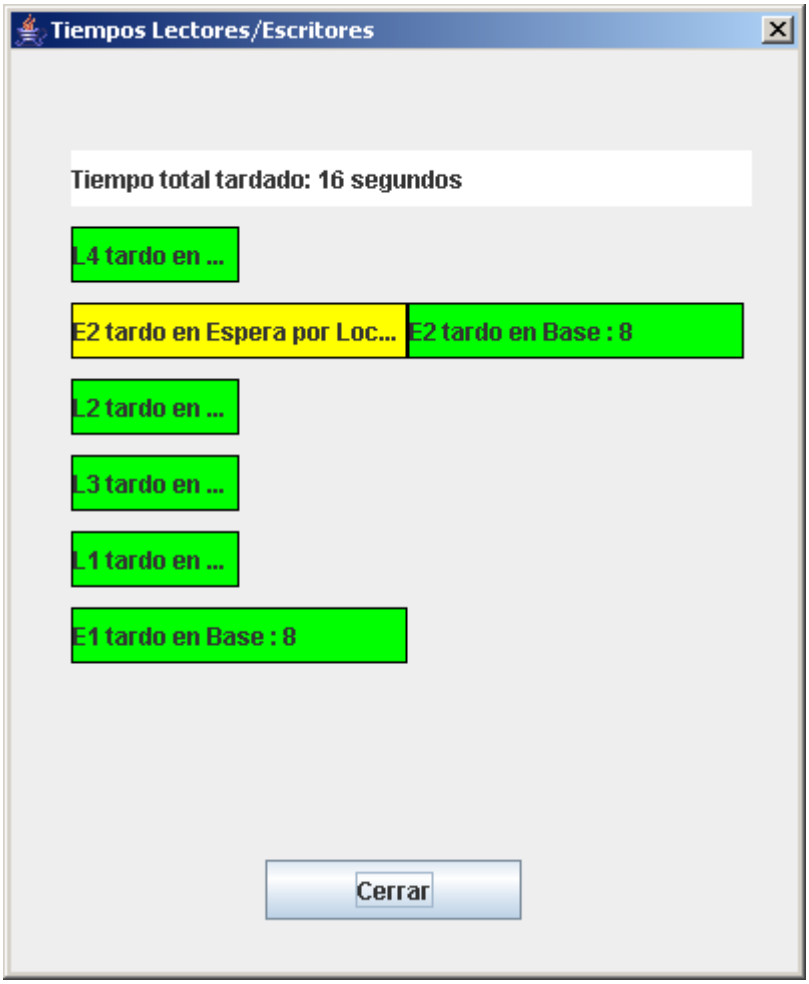
En cuanto a la cantidad de lectores, se seleccionan **4 Lectores y 2 Escritores**.

En la pantalla de parámetros se dejan los tiempos que vienen por defecto:

Intervalo Largada Lectores: 0
Tiempo Espera Lectores: 0
Tiempo Ejecución Lectores: 4
Tiempo Espera Escritores: 0
Tiempo Ejecución Lectores: 8

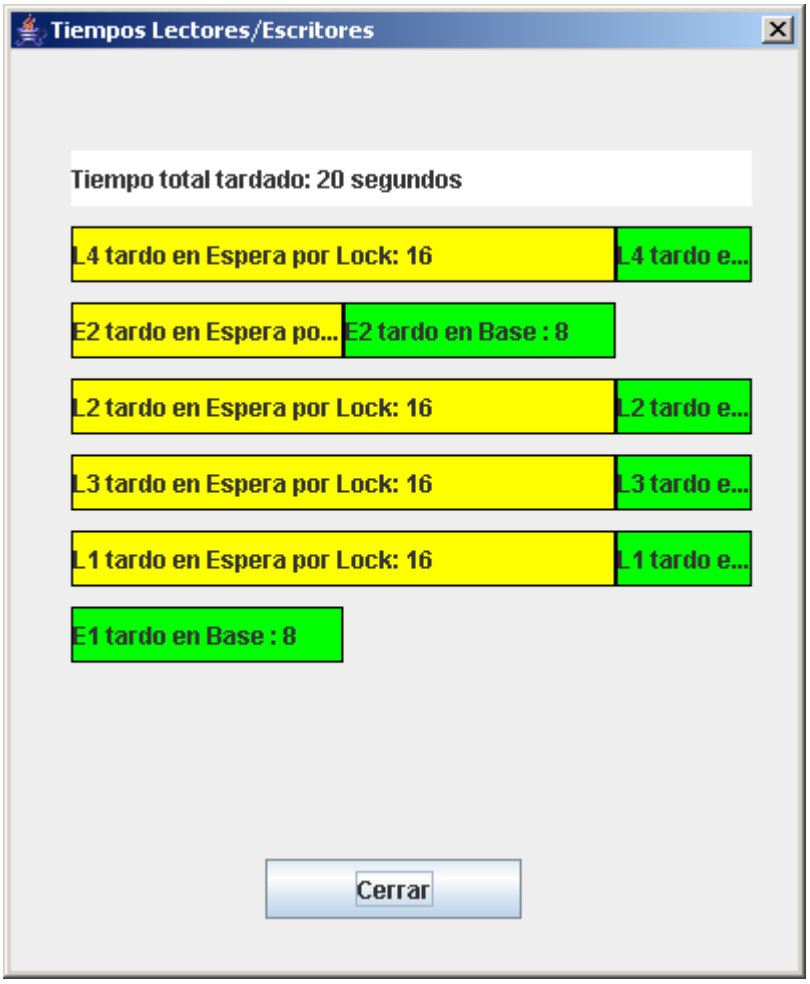


Se ejecuta primero con Oracle y se obtiene la siguiente pantalla de Tiempos:



Ahora se ejecuta con los mismos parámetros pero con SQL Server 2000.

Se obtiene la siguiente pantalla:



Resultados obtenidos

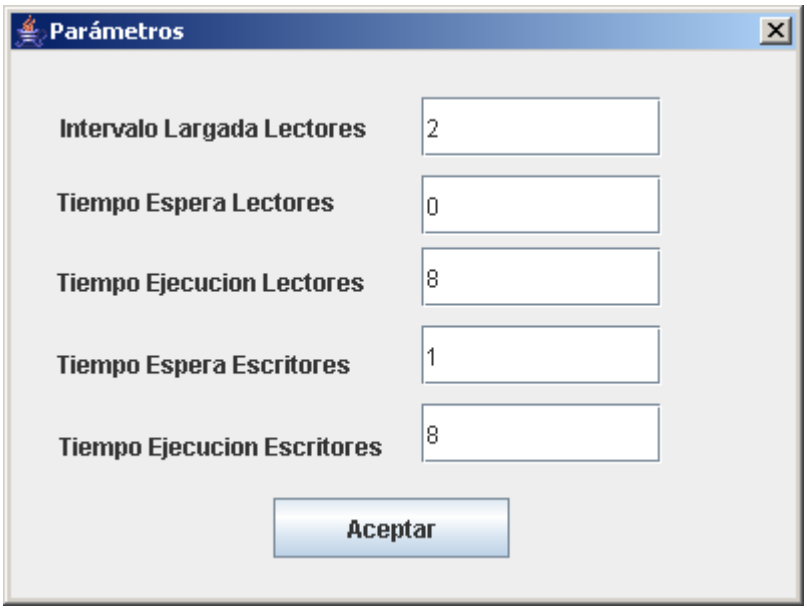
El primer resultado a comparar es la barra de color blanco, que señala el tiempo total demorado. Oracle demora 16 segundos mientras que SQL Server 2000 demora 20 segundos. Esto es 4 segundos más, que son los perdidos por el manejo de bloqueos al no permitir que los lectores se ejecuten concurrentemente con ningún escritor.

Lo más importante de estas pruebas es el tiempo tardado por bloqueo, y es el que se representa en color Amarillo. Si se suman los tiempos perdidos por bloqueos, se puede observar que en Oracle son solamente los **8 segundos** perdidos por el Escritor 2, ya que se bloqueo su entrada al encontrarse el Escritor 1 en la base, mientras que en SQL Server 2000 los tiempos perdidos acumulan **72 segundos,** esto es 16 de cada uno

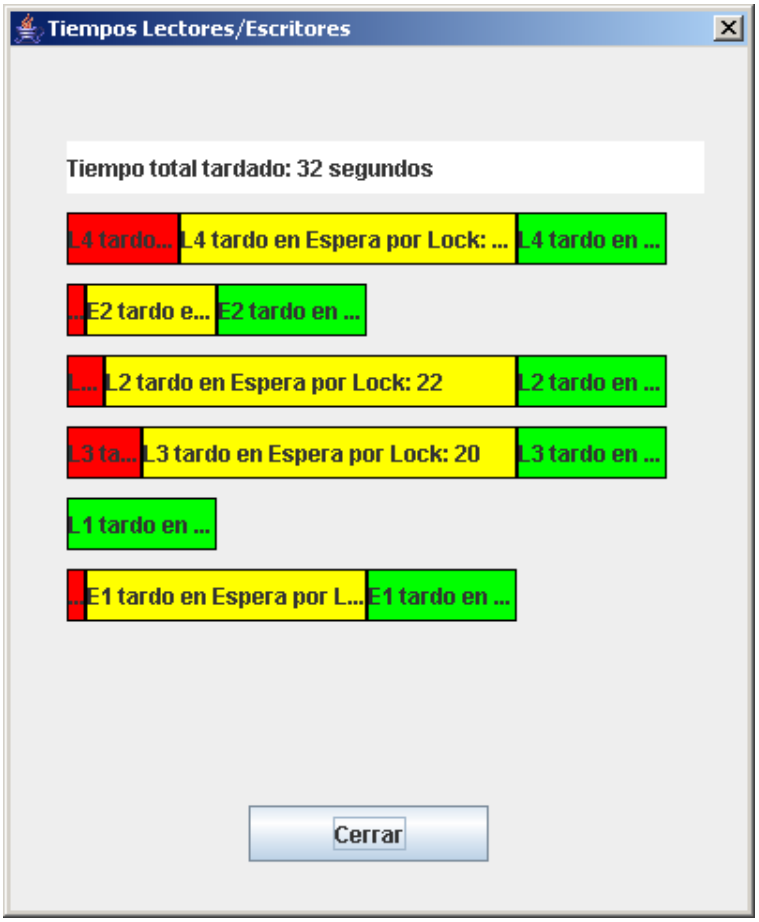
de los Lectores que esperaron 8 segundos de cada uno de los 2 escritores y 8 de un escritor que esperó al otro.

Por ultimo en el siguiente ejemplo se muestra una particularidad de SQL Server 2000.

Se asignan los siguientes valores a los parámetros:



Se ejecuta y se obtiene la siguiente Ventana de Tiempos:



Particularidades de este resultado:

Se ejecuta primero el **Lector 1** durante 8 segundos; en el instante 1 intenta entrar el **Escritor 1** y el **Escritor 2** pero no pueden debido a que se encuentra en la base el **Lector 1.**

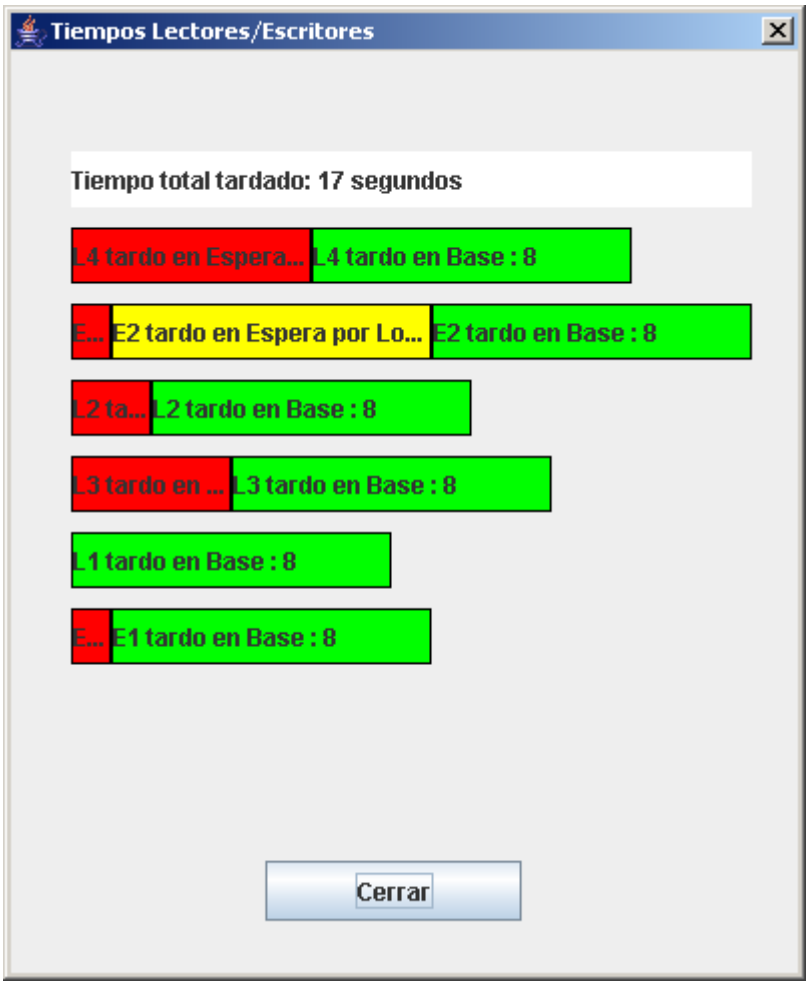
En el Instante 2 intenta acceder el **Lector 2,** y aquí está la particularidad del manejo que implementa SQL Server 2000. No lo deja acceder a la base de datos, no porque se encuentre el **Lector 1** ya que los lectores pueden estar en la base de datos al mismo tiempo, sino porque está esperando el **Escritor 1** y el **Escritor2.** De esta forma es tan fuerte la prioridad que ejerce SQL Server 2000 favoreciendo a los escritores sobre los lectores que nadie entrará hasta que termine el **Lector 1, luego** entre el **Escritor 2 y termine, y por ultimo entre el Escritor 1** y éste también termine.

De esta forma intenta que los escritores accedan cuanto antes; en el ejemplo el **Escritor 2** entra en el instante 8, si SQL Server no bloqueara a los lectores, entraría en el instante 14.

Los tiempos perdidos por bloqueo suman en este ejemplo **82 segundos** y el total tardado **32 segundos.**

La particularidad explicada es para evitar lo que se conoce como live-lock y consiste en evitar que los escritores mueran de inanición *(Ver capítulo 3).*

Se prueba el ejemplo anterior en Oracle



Los tiempos perdidos por Bloqueo son de **8 segundos en vez de 82 segundos** y el tiempo total de la prueba es de **17 segundos contra 32 segundos de SQL Server 2000.**

7 – APLICACION BASADA EN TPC-C

(Referencias: [5])

Objetivo

El objetivo es crear una aplicación para comparar los tiempos de respuesta de los modelos de concurrencia manejados por **Bloqueos** como el que implementa SQL Server y DB2 y el modelo de **Múltiples Versiones** como el que implementa Oracle, PostgreSQL y MySql (en este último si se usan tablas tipo InnoDB).

La idea de ésta aplicación es representar la mayoría de las aplicaciones reales. Luego de analizar varias opciones se decidió usar el benchmark TPC-C como modelo para nuestra aplicación.

El **Benchmark TPC-C** es un trabajo de carga OLTP, que tiene transacciones intensivas de solo lectura y actualización que simulan las actividades que se encuentran en entornos de aplicaciones con complejo OLTP. Esto se logra con un conjunto de componentes de sistema asociados a éstos ambientes, que se caracterizan por:

- Permitir la ejecución simultánea de múltiples tipos de transacciones que permiten mayor complejidad.
- Modo de ejecución de transacciones en línea y diferidas.
- Tiempo de ejecución del sistema razonable.
- Importante Entrada/Salida de disco.
- Integridad de Transacciones (Propiedades ACID)
- Distribución no uniforme de acceso a datos a través de clave primaria y secundaria.
- Base de Datos formada por varias tablas con una amplia variedad de tamaños, atributos y relaciones.

Aunque éstas especificaciones expresan la implementación en términos de un modelo de datos relacional con un esquema convencional de bloqueo, la base de datos se puede implementar usando cualquier manejador de base de datos disponible (SGBD), servidor de base de datos, sistema de archivos u otro repositorio de datos que provea una implementación funcionalmente equivalente.

Diseño Lógico de la Base de Datos

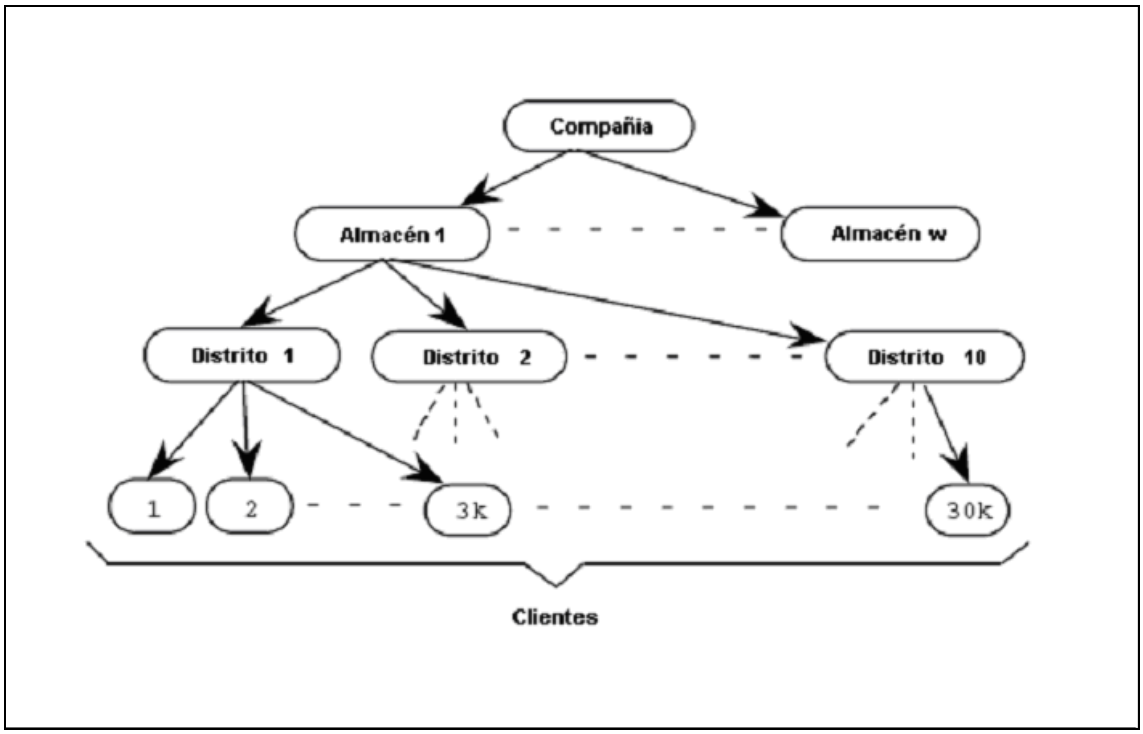
El Benchmark está compuesto por un conjunto básico de operaciones diseñadas para probar la funcionalidad del sistema de una manera que represente un ambiente con

complejas operaciones OLTP. Para que el Benchmark sea más fácil e intuitivo se define un sistema de Distribución de productos.

TPC-C no representa la actividad de ningún segmento de negocios en particular, sino cualquier industria que debe manejar, vender o distribuir un producto o servicio (Ej. alquiler de autos, distribución de comida, autopartes, etc.). TPC-C no trata de ser un modelo de cómo construir una aplicación.

El propósito del Benchmark es reducir la diversidad de operaciones que se encuentran en una aplicación en producción mientras se mantienen las características de performance esenciales de una aplicación (el nivel de utilización del sistema y la complejidad de las operaciones).

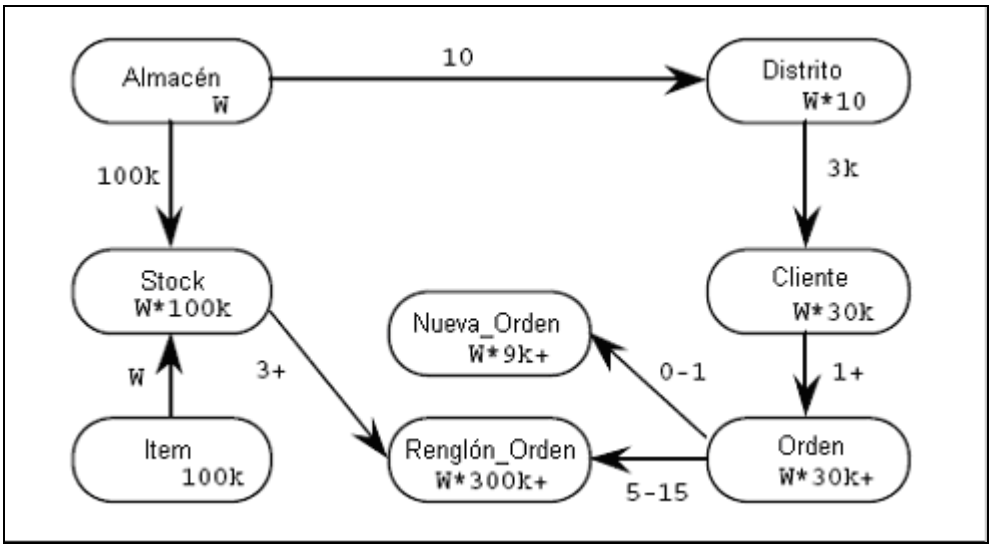
La Compañía que se toma como modelo para el Benchmark es una Distribuidora con varios Distritos de venta, geográficamente distribuidos con sus almacenes asociados. A medida que se expande la Compañía se crean nuevos Almacenes asociados a sus Distritos. Cada Almacén cubre 10 distritos. Cada Distrito sirve a 3000 Clientes. Todos los Almacenes mantienen stock para los 100.000 productos que vende la compañía.



Los Clientes llaman a la Compañía para realizar una nueva orden o consultar el estado de una orden existente. Las Ordenes se componen de un promedio de 10 renglones (10 renglones de ítems). El 1% de los renglones de una Orden son de ítems que no tiene el Almacén de la Región en stock y se deben abastecer de otros Almacenes.

Entidades de la Base de Datos, Relaciones y características

Los componentes de la base de datos de TPC-C son nueve tablas. Por una cuestión de simplificar el modelo para adaptarlo a nuestras necesidades, se quitó la tabla Historial, quedando así ocho tablas en nuestro modelo. A continuación se definen en un diagrama las relaciones entre las tablas:



Transacciones

TPC-C define diferentes tipos de Transacciones que se describen a continuación:

Transacción Nueva Orden

La transacción de negocios Nueva Orden consiste en ingresar una orden completa a través de una sola transacción. Representa una transacción de mediana carga, y de lectura-escritura con una alta frecuencia de ejecución y un alto requerimiento de tiempo de respuesta para satisfacer a los usuarios en línea. Esta transacción es el esqueleto del trabajo de carga. Está diseñada para establecer una carga variable en el sistema para reflejar la actividad en línea de la base de datos como típicamente se encuentra en ambientes en producción.

Transacción de Pago

Esta transacción de negocios actualiza el balance del cliente y refleja el pago en el distrito y las estadísticas de venta en el almacén. Representa una transacción de carga

liviana, y de lectura-escritura con una alta frecuencia de ejecución y un alto requerimiento de tiempo de respuesta para satisfacer a los usuarios en línea. Además, ésta transacción incluye acceso a datos de la tabla CLIENTE sin utilizar la clave primaria.

Transacción Estado de una Orden

La transacción de negocios Estado de una Orden consulta el estado de la última orden de un cliente. Representa una transacción de solo lectura de mediana escala con una baja frecuencia de ejecución y tiempo de respuesta para satisfacer a usuarios conectados. Además, ésta transacción incluye acceso a datos de la tabla CLIENTE sin utilizar la clave primaria.

Transacción de Entrega

La transacción de negocios de Entrega consiste en un procesamiento por lotes de 10 órdenes nuevas (todavía no entregadas). Cada orden es procesada (entregada) completamente, con el alcance de una transacción de lectura-escritura. El número de órdenes que se entregan como un grupo (o lote) dentro de una misma transacción es específicamente implementado. Esta transacción que está compuesta de una o mas transacciones en la base de datos (hasta 10), tiene una baja frecuencia de ejecución y se debe completar con un tiempo de respuesta bajo.

Se intenta que la transacción se ejecute en modo diferido a través de un mecanismo de colas, en vez de que su ejecución sea interactiva con respuesta a la terminal avisando cuando haya finalizado. El resultado de la ejecución diferida se graba en un archivo.

Transacción de Nivel de Stock

Determina la cantidad de ítems vendidos recientemente que tienen el nivel de stock por debajo de un valor específico. Representa una transacción de solo lectura de alta carga con una baja frecuencia de ejecución, un tiempo de respuesta bajo, y bajos requerimientos de consistencia.

Pruebas propuestas por TPC-C

TPC-C especifica una serie de pruebas que se deben llevar a cabo para demostrar que las propiedades ACID se cumplen.

El sistema sobre el cual se prueba el Benchmark debe cumplir con las propiedades ACID de sistemas de procesamiento de transacciones. La única excepción a esta regla es permitir non-repeatable reads para la transacción de Nivel de Stock.

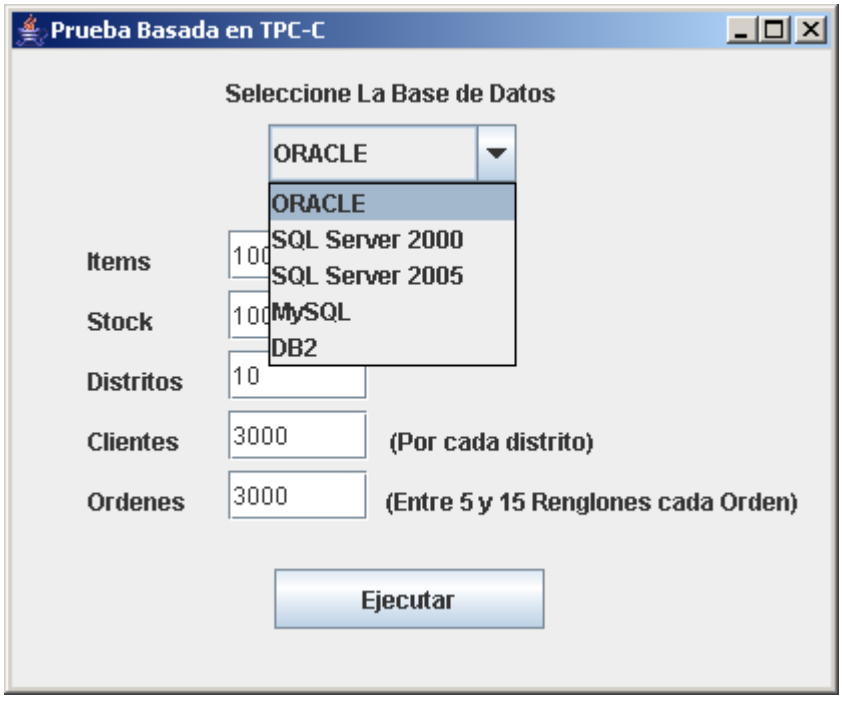
Ninguna serie de pruebas finitas puede probar que las propiedades ACID se cumplan. Pasar las pruebas indicadas es una condición necesaria pero no suficiente para cumplir con los requerimientos ACID.

TPC-C define pruebas para probar Atomicidad, Consistencia, Aislamiento y Durabilidad. Para el caso de estudio de ésta tesis, se utilizan las pruebas de Aislamiento, dado que se define en términos de fenómenos que pueden ocurrir durante la ejecución de transacciones concurrentes en una base de datos.

Los fenómenos que pueden ocurrir son: Dirty Write, Dirty Read, Non-repeatable Read, Phantom.

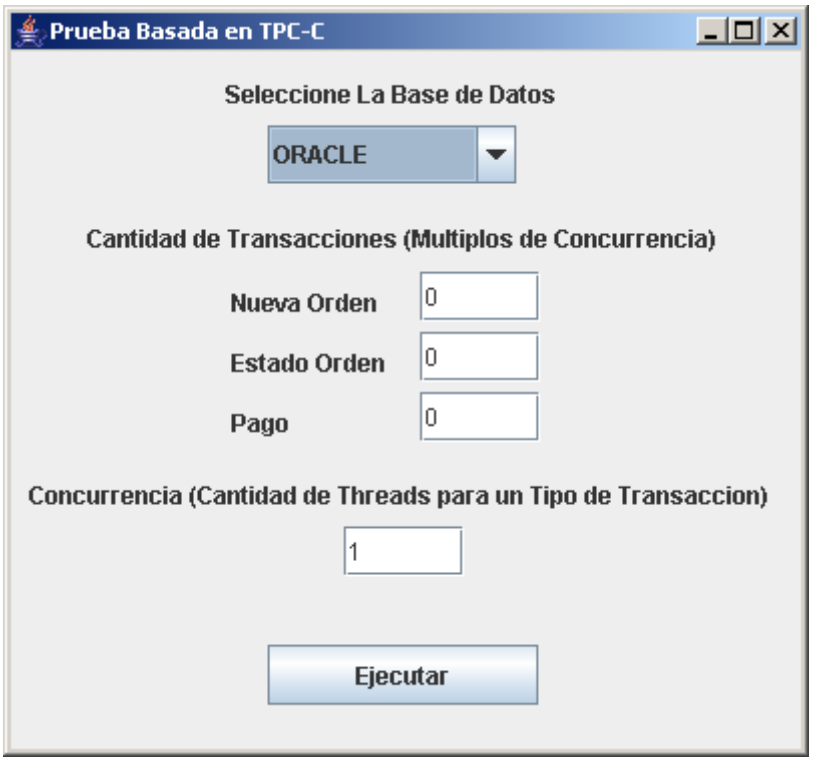
Implementación de TPC-C

Para comparar el modelo de concurrencia MVCC con el modelo tradicional de bloqueos en una aplicación estándar como la que define TPC-C se desarrolló un modulo de carga y una aplicación basados en éste benchmark.



Se decidió crear el programa de carga en lenguaje JAVA y no mediante mecanismos permitidos por cada DBMS en particular. Esta decisión fue para tener un programa de carga unificado y permitir ejecutarlo contra cualquier base de datos. Este programa de carga genera los datos para las tablas de la manera que se define en TPC-C y permite elegir la base de datos en la cual se crearán.

La otra aplicación basada en TPC-C, también desarrollada en JAVA, ejecuta las transacciones definidas en éste benchmark; estas transacciones se ejecutan cada una con una conexión independiente a la base de datos y "corren" como threads concurrentemente.



Diseño de la aplicación TPC-C

Existe una clase principal llamada Director que se encarga de crear los threads y ejecutarlos. Básicamente, cada uno de estos threads abre una conexión con la base de datos, se ejecuta, realiza Commit o Rollback y cierra su conexión.

En la Aplicación realizada se simplifico a 3 tipos de transacciones, una de escritura, una de lectura y una mixta.

En la Ventana que se puede visualizar arriba, el campo Concurrencia indica la cantidad de threads en los que se va a dividir la cantidad de transacciones elegidas. Es decir si se elige que se ejecuten 10 transacciones Nueva Orden y se quiere que cada una "corra" en un thread distinto, habrá que indicar el valor 10 en el campo concurrencia; si se desea que cada thread ejecute 2 transacciones se deberá escribir el valor 5 en el

campo concurrencia. Si se quieren ejecutar 4 transacciones de cada uno de los 3 tipos existentes y se quiere que se ejecuten en un thread cada una se deberá indicar en el campo concurrencia el valor 4 (Este campo indica la cantidad de threads para un tipo de transacción), por lo que habrá 12 threads corriendo. Si se quiere que se ejecuten 2 transacciones por thread se deberá indicar en el campo concurrencia el valor 2.

Cada ejecución de una transacción que realiza un thread implica registrar el driver, abrir la conexión, ejecutar la transacción propiamente dicha y cerrar la conexión.

Se realizaron pruebas con el "debugger" del ambiente de desarrollo forzando interacciones entre transacciones de manera de poder probar que se respetan las propiedades ACID, en particular el aislamiento. Estas pruebas fueron satisfactorias tanto en el modelo MVCC como por bloqueos.

Otro tipo de pruebas que se realizó fue ejecutar varias transacciones de cada uno de los tipos: "Nueva Orden", "Estado de una Orden" y "Pago", comprobando luego de haberse ejecutado las mismas, la consistencia de la base de datos. Estas pruebas también resultaron correctas.

Tiempos obtenidos en TPC-C

Recordemos que usamos el Benchmarck TPC-C como modelo para desarrollar nuestra aplicación de tiempos.

Se decidió implementar tres operaciones de las definidas en este bechmark. Una de lectura "**Estado Orden**", otra de escritura "**Nueva Orden**" y una última mixta "**Pago**".

Se decidió ejecutar las transacciones cada una por separado y luego una prueba combinada con las tres transacciones simultaneas. Cada una de estas pruebas se realizó con 500 y 1000 transacciones. En el caso de la prueba combinada fueron 1500 y 3000 transacciones respectivamente.

Con las pruebas de 500 transacciones se decidió utilizar 1, 2, 10, 50, 250 y 500 threads y con las pruebas de 1000 transacciones se utilizaron 1, 2, 10, 100, 500 y 1000 threads. Recordar que cada thread ejecuta la cantidad necesaria de transacciones para llegar a la cantidad de transacciones solicitadas (500 o 1000) de acuerdo a la cantidad de threads que sean.

Ejemplo:

Objetivo: 500 Transacciones	Objetivo: 1000 Transacciones
1 thread ejecutará 500 transacciones	1 thread ejecutará 1000 transacciones
2 threads ejecutaran 250 transacciones	2 threads ejecutaran 500 transacciones
10 threads ejecutaran 50 transacciones	10 threads ejecutaran 100 transacciones
50 threads ejecutaran 10 transacciones	100 threads ejecutaran 10 transacciones
250 threads ejecutaran 2 transacciones	500 threads ejecutaran 2 transacciones
500 threads ejecutaran 1 transacción	1000 threads ejecutaran 1 transacción

Las bases de datos elegidas fueron:

- Oracle 9i
- MySql 5.0 (Con tablas de tipo InnoDB)
- SqlServer 2000
- SqlServer 2005 (con y sin Snapshot)
- DB2

Se decidió utilizar Oracle y DB2 como representantes de bases de datos empresariales que soportan grandes volúmenes de datos y altos niveles de transacciones. Oracle utiliza MVCC y DB2 utiliza bloqueos como mecanismos de control de Concurrencia.

Como representantes de bases de datos de mediana escala se eligió SqlServer 2000 y MySql 5.0, el primero utilizando bloqueos y el segundo con MVCC mediante sus tablas de tipo InnoDB.

Por ultimo se eligió SqlServer 2005 que resultó óptimo para estas pruebas ya que proporciona la opción de elegir el control de concurrencia por bloqueos y por MVCC (Snapshot), permitiendo comparar los dos controles de concurrencia en una misma base de datos y con el mismo DBMS.

Para cada ejecución se calcularon tres tiempos:

El tiempo Total: Es el tiempo transcurrido desde el comienzo de la aplicación hasta que termina el último Thread.

El tiempo Total con conexiones: Es la suma de todos los tiempos de los threads desde que comienzan hasta que terminan.

El tiempo Total sin conexiones: Es la suma de todos los tiempos de los threads desde que comienzan la ejecución, después de conseguir la conexión, hasta que terminan.

El hardware utilizado para las pruebas fue una PC con un Procesador Intel® Hyper Threading Pentium® 4 3GHz con 512Mb de RAM y Windows XP como S.O.

Los tiempos obtenidos de estas pruebas se encuentran en el **Anexo IV** y en el documento Adjunto **"Tiempos TPC-C.xls".**

En cuanto a la comparación de tiempos de ejecución en los diferentes modelos de concurrencia, se puede decir que los resultados obtenidos son relativos dado que se considera que una comparación de tiempos precisa exige otro tipo de consideraciones como el tipo de hardware elegido, el driver de conexión a la base de datos utilizada (JDBC, ODBC, etc.), el tuning de los objetos creados en la base de datos, el tráfico de red, la cantidad de threads que permite ejecutar el hardware, etc.

Los tiempos obtenidos en DB2 y Oracle son notablemente superiores a los de las otras bases de datos. Se considera que se debe a que a éstas bases -que son altamente configurables- no se les realizó ningún tipo de configuración específica, utilizando los parámetros por defecto. Además fueron instaladas en una PC, con Windows XP, cuando en realidad el ambiente natural de estas bases son grandes Servidores con varios procesadores, con UNIX como S.O.

Los mejores tiempos se obtuvieron con SQL Server 2000. Se considera que se debe a que se utilizó ODBC en vez de JDBC, y sobre todo por el hardware utilizado que es el ambiente natural para ésta base de datos.

La mejor comparación se obtiene de SqlServer 2005 ya que proporciona la opción de elegir el control de concurrencia por bloqueos y por MVCC (Snapshot), permitiendo comparar los dos controles de concurrencia en una misma base de datos y con el mismo DBMS. Los tiempos obtenidos fueron similares, siendo sutilmente mayores los de Snapshot. Conseguir la conexión con la base de datos desde Java utilizando JDBC es mas "cara" (lleva más tiempo) en Snapshot. En la transacción de Escritura los tiempos son similares (la relación Snapshot/Bloqueos es 1.05s:1s), mientras que en la de Lectura son mayores los de Snapshot (1.35s:1s).

8 - CONCLUSION

Bajo una mirada rápida y sin realizar un análisis en detalle, el uso principal de MVCC podría ser para ambientes con lecturas intensivas como por ejemplo data warehousing y sistemas de reportes en donde podría haber un fuerte impacto de concurrencia generado por los bloqueos de lectura a nivel de tabla, de consultas largas y complejas (con agregaciones o joins entre tablas grandes que requieren una vista consistente de la base de datos) que pueden bloquear transacciones que necesitan actualizar datos.

Cuando se utilizan las técnicas de bloqueo tradicional, las aplicaciones "sufren" bloqueos. El acceso simultáneo de transacciones de lectores y escritores que entran en conflicto y se bloquean es común y el bloqueo es de corta duración y no genera un cuello de botella. Pero ésta situación puede cambiar en sistemas bajo estrés en donde el aumento del procesamiento de una transacción puede provocar un bloqueo desproporcionado. Cuanto mas demora la ejecución de una transacción, mas se mantienen los bloqueos en los datos y mayor es la probabilidad de bloqueos.

Analicemos Algunos Ejemplos

Analicemos algunos casos reales en donde se puede identificar en que casos conviene utilizar MVCC o Bloqueos como mecanismos de control de Concurrencia, teniendo en cuenta las ventajas/desventajas de uno u otro:

1) Sistemas de Servicios

Supongamos que tenemos un Sistema de servicios (vuelos, autos, hospedaje, etc.) que funciona en una intranet e internet simultáneamente. Antes de contratar un servicio, sería bueno que el cliente pudiera seleccionar entre varias opciones, según las características del servicio, la fecha en la que desea contratarlo, el lugar, etc. Para conseguir toda ésta información, es muy probable que se deba realizar un join entre por lo menos varias tablas como: Servicio, Clase de Servicio, Reserva, Lugar, etc. Además, la información obtenida debe provenir sólo de datos consistentes de la base de datos.

Entonces la consulta se podría guardar "desconectada" en una estructura auxiliar (recordset, resultset, etc.) para no mantener los datos bloqueados mientras el cliente selecciona el servicio de su preferencia. Esta técnica es optimista ya que a pesar de que los datos no están bloqueados, la probabilidad de que existan conflictos es baja.

En el momento de realizar efectivamente la reserva, la aplicación se tendrá que reconectar con la base de datos y comparar la columna de timestamp de la fila que se quiere reservar (la que está almacenada en la estructura auxiliar) con el timestamp de la fila almacenada en la base de datos para identificar si los datos han cambiado (mientras los datos estaban "desconectados") y si todavía se puede hacer la operación. Para realizar ésta validación se debe hacer un bloqueo exclusivo sobre los datos, y en el caso en que se pueda realizar la operación, actualizar la base de datos para reflejar los cambios.

Pero la técnica explicada anteriormente bajo un mecanismo de control de concurrencia tradicional o por bloqueos no es realmente optimista. Existe la posibilidad de una demora muy grande cuando se buscan las opciones de los servicios candidatos para que el cliente los seleccione. Mientras se "traen" los datos de la consulta para almacenarlos en una estructura auxiliar, no se podrá realizar ninguna operación de escritura como por ejemplo reservar un servicio si los datos involucrados en la lectura entran en conflicto con la escritura.

En cambio, si se utiliza MVCC como técnica de control de concurrencia, la consulta para obtener las opciones de los servicios no produce bloqueos mientras se está ejecutando. De esta manera se reduce la carga de bloqueos en el servidor y los datos no se bloquean para otros clientes que quieran reservar un servicio. Este modelo no mejora necesariamente las posibilidades de que un servicio que se quiere reservar (que aparece en la consulta de las opciones de servicios) esté disponible en el momento de la reserva en si, pero las reservas son más rápidas y no se bloquearán por pedidos simultáneos. Esto lleva a mejorar la performance de transacciones, en especial en picos de carga de trabajo

2) Reportes utilizando la información online

Cada vez es más notable en los sistemas informáticos la continua necesidad de reducir costos y a la vez ampliar la capacidad de disponibilidad de la información. Uno de los objetivos para lograr esto es eliminar la latencia entre el momento en que se guardan los datos en la base de datos y su posterior disponibilidad para el uso en reportes para poder tomar decisiones gerenciales dentro de una organización.

Utilizando el método tradicional por bloqueos, una manera de evitar bloqueos largos sobre los datos online es utilizar el nivel de aislamiento Read Uncommitted. Este nivel es difícil de usar, sobre todo cuando se deben hacer joins entre varias tablas porque provee acceso a nivel de sentencia sobre una vista inconsistente de la base de datos, sin bloquear las transacciones, en donde los datos de una transacción pueden ser parciales/inconsistentes. Cuando se necesitan consultas complejas para realizar reportes de alto nivel jerárquico dentro de una organización, el procesamiento es largo y lleva a que los datos sean cada vez más inconsistentes a medida que llega más información a la base de datos y el tiempo transcurre.

En situaciones donde se debe trabajar con una vista consistente de los datos para realizar un reporte, no queda otra solución más que "correr" la consulta en un horario en el cual no interfiera con la concurrencia del sistema online, perdiendo así la disponibilidad de la información actual para poder tomar decisiones. Una consulta larga de solo lectura para realizar un reporte en un horario con mucha carga de trabajo podría terminar bloqueando a todos los escritores que están actualizando la base de datos.

En cambio, si se utiliza MVCC como técnica de control de concurrencia, se pueden realizar reportes en cualquier momento, sin correr el riesgo de bloquear escritores que estén actualizando datos del sistema online. Además, el acceso a los datos puede ser basado sobre una vista consistente de la base de datos a nivel de sentencia o a nivel de transacción.

El uso de MVCC agrega carga adicional sobre el servidor de base de datos. Para las transacciones de escritura, sus cambios deben ser versionados y para las de lectura, deben "buscar" la versión apropiada en el momento en que comienza la transacción.

3) Reportes utilizando una copia de la base de datos

En sistemas donde la información cambia mucho, la utilización de MVCC puede no ser la mejor opción dado el procesamiento extra que se produce al crear y manejar las versiones previas de un dato. En un sistema con una carga de trabajo considerable, puede ser que no valga la pena trabajar con los datos online para hacer reportes, para no sobrecargar aun más el sistema.

Para solucionar un problema de éste tipo, se puede mantener una copia o replica de la base de datos, de manera que esté lo suficientemente actualizada como para que se puedan hacer reportes con datos razonables, y almacenarla en otro servidor para no sobrecargar el servidor con la base de datos online.

Conclusión

En un mecanismo de control de concurrencia pesimista o por bloqueos, los bloqueos se utilizan para prevenir que las transacciones modifiquen datos de manera que afecten a otras transacciones. Una vez que se aplica un bloqueo, las otras transacciones no pueden modificar datos de manera que entren en conflicto con otras transacciones hasta que se libere el bloqueo. Este mecanismo es útil para ambientes en donde hay alta contención de datos y en donde el costo de proteger los datos usando bloqueos es menor que el costo de hacer rollback en las transacciones cuando hay conflictos de concurrencia.

En un mecanismo de control de concurrencia optimista, las transacciones no bloquean los datos cuando realizan lecturas. Cuando se realiza una actualización el sistema verifica si otra transacción cambió los datos involucrados en ella después de que fueron leídos. Si otra transacción actualizó los datos, ocurre un error. Este mecanismo es útil en ambientes con baja contención de datos, y en donde el costo de hacer ocasionalmente rollback de una transacción es menor que el costo de bloquear los datos cuando se leen.

En un mecanismo de control de concurrencia pesimista o por bloqueos, si una transacción cambia una fila, entonces otra transacción no puede leer la fila hasta que el escritor realiza commit. Hay casos en donde esperar a que se realice el cambio es la forma correcta, pero existen otros casos en donde obtener el estado previo consistente del dato es suficiente.

En un mecanismo de control de concurrencia optimista basado en versiones como MVCC, un lector puede leer la versión previa de un dato con el costo de tener que mantener esta versión cuando el dato se modifica, aunque nadie esté accediendo actualmente al dato. Esto significa que todas las consultas, actualizaciones y eliminaciones (pero no inserciones) deben pagar el costo de versionar los datos, lo cual genera una carga adicional. Es importante entender que a pesar de que cada sentencia puede costar más en cuanto a tiempo de ejecución debido al versionado, en definitiva se aumenta la performance porque se reduce la contención. Cuando la contención degrada la performance de un sistema es útil usar MVCC.

Utilizando ésta técnica de control de concurrencia, el programador se preocupa menos por posibles deadlocks y bloqueos pagando el costo de sobrecarga por versionado.

Cuando se necesita utilizar bloqueos para controlar el acceso a recursos y se utiliza MVCC como mecanismo de control de concurrencia, entonces se debe tener en cuenta que los lectores no bloquean a los escritores y el bloqueo se debe programar explícitamente.

Cuando la naturaleza de la aplicación consiste en actualizaciones del estilo procesamiento por lotes en donde se modifican varias filas, no se recomienda utilizar MVCC con consistencia a nivel de transacción ya que la probabilidad de que ocurran conflictos es alta. En éste caso se recomienda utilizar un mecanismo por bloqueos (READ COMMITED, REPEATABLE READ o SERIALIZABLE), mantener transacciones cortas y programar la aplicación de manera de minimizar conflictos entre recursos y en consecuencia minimizar posibilidades de deadlocks.

Por todo lo analizado anteriormente se considera que no es mejor el mecanismo por bloqueos o pesimista que el mecanismo por multiversión ni viceversa, sino que simplemente hay que analizar en que caso es conveniente usar cada uno, teniendo en ouenta por sobre todo la contención de los datos de la aplicación a desarrollar.

Bibliografía

1. Fundamentals of Database Systems. Ramez Elmasri, Shamkant B. Navathe.

2. Database System Concepts. Second Edition. Henry F. Korth y Abraham Silverschatz

3. El concepto de Transacción: Virtudes y Limitaciones. Jim Gray (1981). http://www.stanford.edu/~candea/teaching/cs444a-fall-2003/readings/Gray81.pdf

4. ANSI-ISO SQL 99. http://www.ncb.ernet.in/education/modules/SGBD/SQL99/

5. TPC BENCHMARK™ C - Standard Specification - Revision 5.4 - April 2005 http://www.tpc.org/tpcc/default.asp

6. Oracle 9i – Database Concepts – Release 2 (9.2) – Part VII Data Protection http://download-west.oracle.com/docs/cd/B10501_01/server.920/a96524/part_7.htm#435986

7. Introducción a los sistemas de Base de Datos. C. J. Date 2nd Edition. Benjamin/Cummings 1994, ISBN 0-8053-1748-1

8. Concurrency Control and Recovery in Database Systems. Philip A. Bernstein, Vassos Hadzilacos, Nathan Goodman. http://research.microsoft.com/pubs/ccontrol/

9. A Critique of ANSI SQL Isolation Levels. H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, P. O'Neil. http://research.microsoft.com/research/pubs/view.aspx?type=Technical%20Report&id=5

10. Generalized Isolation Level Definitions. Atul Adya. Barbara Liskov. Patrick O'Neil. http://research.microsoft.com/~adya/pubs/icde00.pdf

11. A Technical Discussion of Multi Version Read Consistency ftp://ftp.software.ibm.com/software/data/pubs/papers/readconsistency.pdf

12. "Granularity of Locks and Degrees of Consistency in a Shared Database," Modeling in Data Base Management Systems, Amsterdam: Elsevier North-Holland, 1976. J. Gray, R. Lorie, G. Putzolu, and I. Traiger,

13. “The theory of database concurrency control" (CS Press 1988) Papadimitriou.

14. "A Theoretical Formulation for Degrees of Isolation in Databases," Technical Report, George Mason University, Fairfax, VA, 1995. V. Atluri, E. Bertino, S. Jajodia,

15. “The Notions of Consistency and Predicate Locks in a Database System,” K. P. Eswaran, J. Gray, R. Lorie, I. Traiger,

16. “Transaction Processing: Concepts and Techniques,” Corrected Second Printing, Morgan Kaufmann 1993, J. Gray and A. Reuter

17. “An Investigation of Transactional Isolation Levels” P. O'Neil, E. O'Neil, H. Berenson, P. Bernstein, J. Gray, J. Melton.

18. Información sobre “Snapshot too old (ORA-1555)” http://www.redcientifica.com/doc/doc200202020002.html

19. Oracle9i Application Developer's Guide – Fundamentals - Release 2 (9.2) Capitulo 7 http://download-west.oracle.com/docs/cd/B10501_01/appdev.920/a96590/adg08sql.htm

20. A Read-Only Transaction Anomaly Under Snapshot Isolation By Alan Fekete, Elizabeth O'Neil, and Patrick O'Neil http://www.sigmod.org/sigmod/record/issues/0409/2.ROAnomONeil.pdf

21. MULTIVERSION POST ORDERING: A NEW CONCURRENCY CONTROL METHOD By Tanya Jane Harris, William Perrizo and Qin Ding, North Dakota State University http://cs.hbg.psu.edu/~ding/publications/CAINE-1059A.pdf

22. Correct Execution of Transactions at Different Isolation Levels Shiyong Lu, Member, IEEE, Arthur Bernstein, Fellow, IEEE, and Philip Lewis, Fellow, IEEE http://www.cs.wayne.edu/~shiyong/papers/tkde04.pdf

23. SQL Server 2005 Beta 2 Snapshot Isolation http://www.microsoft.com/technet/prodtechnol/sql/2005/SQL05B.mspx

24. PostgreSQL 8.0.3 Documentation
http://www.postgresql.org/docs/8.0/interactive/mvcc.html

25. MySQL Reference Manual. 15.11 InnoDB Transaction Model and Locking
http://dev.mysql.com/doc/mysql/en/innodb-transaction-model.html

26. InterBase 6. Embedded SQLGuide
http://www.ibphoenix.com/downloads/60EmbedSQL.zip

27. DB2 Universal Database concurrency
http://www-128.ibm.com/developerworks/db2/library/techarticle/dm-0406whitlark/

28. A not-so-very technical discussion of Multi Version Concurrency Control
http://firebird.sourceforge.net/doc/whitepapers/fb_vs_ibm_vs_oracle.htm

29. Firebird for the Database Expert: Episode 4 - OAT, OIT, & Sweep
http://firebird.sourceforge.net/
http://www.ibphoenix.com/main.nfs?a=ibphoenix&page=ibp_expert4

30. How Logs Work On MySQL With InnoDB Tables
http://www.devarticles.com/c/a/MySQL/How-Logs-Work-On-MySQL-With-InnoDB-Tables/1/