Bases de Datos

Segunda Parte Trabajo Práctico

Fecha de Entrega: 01/07/2009
Fecha de Recuperatorio: 15/07/2009

Enunciado

En la segunda parte del Trabajo Práctico, el objetivo será desarrollar una herramienta vinculada al manejo de transacciones de una base de datos. La idea será realizar una implementación que permita analizar características de los planes de ejecución de transacciones, tales como serializabilidad, recuperabilidad, entre otras.

En un motor de bases de datos, el *Transaction Manager* es el módulo que se encarga del manejo de transacciones. La base de datos es particionada en ítems, que según la granularidad definida pueden ser tablas, páginas de tablas, registros, etc. Las transacciones operan sobre esos ítems, pudiendo realizar lecturas y escrituras sobre los mismos. Un plan es una posible ejecución de una serie de transacciones y está formado por las acciones de cada transacción. El *Transaction Manager* es el encargado de generar estos planes y su objetivo será permitir o denegar acciones de las transacciones para asegurar que se preserve la atomicidad, consistencia, aislamiento y durabilidad de las transacciones (características conocidas con la sigla *ACID* -Atomicity, Consistency, Isolation & Durability-).

Durante este trabajo se desarrollará una herramienta con el fin de establecer si un plan es legal, serial, serializable y además definir su nivel de recuperabilidad. Trabajaremos con los 3 modelos de transacciones vistos en la Práctica: sin locking, locking binario y ternario. Esta implementación será parte del *UBADB*, motor de uso académico que fue creado por la cátedra. A continuación presentaremos en detalle los puntos que cada grupo debe resolver:

1) **Determinar si un plan es legal**

Según el modelo de transacciones usado, deberán verificar si un plan cumple con las condiciones para ser legal. Las validaciones necesarias son las mencionadas a continuación:

Sin Locking

- Cada transacción posee como máximo un COMMIT.
- Si T tiene COMMIT, éste es el último paso de la transacción.

Locking Binario

- Cada transacción T posee como máximo un COMMIT.
- Si T tiene COMMIT, éste es el último paso de la transacción.
- Si T hace LOCK A, luego debe hacer UNLOCK A.
- Si T hace UNLOCK A, antes debe haber hecho LOCK A.
- Si T hace LOCK A, no puede volver a hacer LOCK A a menos que antes haya hecho UNLOCK A.
- Si T hace LOCK A, ninguna otra transacción T' puede hacer LOCK A hasta que T libere a A.

Locking Ternario

- Cada transacción T posee como máximo un COMMIT.
- Si T tiene COMMIT, éste es el último paso de la transacción.
- Si T hace RLOCK A o WLOCK A, luego debe hacer UNLOCK A.
- Si T hace UNLOCK A, antes debe haber hecho RLOCK A o WLOCK A.

- Si T hace RLOCK A o WLOCK A, no puede volver a hacer RLOCK A o WLOCK A a menos que antes haya hecho UNLOCK A.
- Si T hace RLOCK A, ninguna otra transacción T' puede hacer WLOCK A hasta que T libere a A.
- Si T hace WLOCK A, ninguna otra transacción T' puede hacer RLOCK A o WLOCK A hasta que T libere a A.

Como resultado, debe devolverse si el plan es legal y, en caso de no serlo, una transacción ilegal (si hubiera más de una, devolver cualquiera) y el motivo por el cual viola las restricciones.

2) **Determinar si un plan es serial**

Un plan es serial cuando las acciones de cada transacción aparecen consecutivas, es decir, no debe haber un entrelazamiento entre las acciones de diferentes transacciones. Notar que para los 3 modelos, la verificación es la misma y se debe resolver el problema en la forma más general posible.

Como resultado, debe devolverse si el plan es serial y, en caso de no serlo, una transacción que viole la restricción (si hubiera más de una, devolver cualquiera).

3) **Determinar si un plan es serializable**

Un plan se dice que es serializable cuando su ejecución produce el mismo resultado que una ejecución serial. A diferencia de los planes seriales, aquí se permite el entrelazamiento de acciones de diferentes transacciones, pero al mismo tiempo, se imponen restricciones sobre cómo entrecruzar transacciones. Típicamente, se construye un grafo con arcos direccionales para determinar la precedencia de las transacciones y, si el grafo no tiene ciclos, el plan se considera serializable. Dependiendo del tipo de modelo de transacciones, la construcción del grafo varía y a continuación presentamos en detalle cuándo deben agregarse arcos entre transacciones T1 y T2 según el modelo:

<u>Sin Locking</u>

- T1 lee un ítem A y T2 luego escribe A
- T1 escribe un ítem A y T2 luego lee A
- T1 escribe un ítem A y T2 luego escribe A

<u>Locking Binario</u>

- T1 hace LOCK de un ítem A y luego T2 hace LOCK de A

<u>Locking Ternario</u>

- T1 hace RLOCK de un ítem A y T2 luego hace WLOCK de A
- T1 hace WLOCK de un ítem A y T2 luego hace RLOCK de A
- T1 hace WLOCK de un ítem A y T2 luego hace WLOCK de A

Nota: Como optimización a la hora de implementarlo, pediremos que se omitan arcos que se deduzcan por transitividad.

Una vez construido el grafo en forma particular dependiendo del modelo, se deberá verificar si un grafo tiene ciclos utilizando *Teoría de Grafos*. Como resultado, debe devolverse el grafo e indicar si el plan es serializable. Además, si es serializable, retornar una lista con las posibles ejecuciones y, en caso contrario, presentar un ciclo (si hubiera más de uno, devolver cualquiera).

4) **Determinar el nivel de recuperabilidad de un plan**

Un plan puede ser no recuperable, recuperable, evitar aborts en cascada o estricto. El nivel de recuperabilidad no depende del modelo de transacciones, sino de analizar si las acciones de cada transacción son lecturas, escrituras o commit y del orden en que estas acciones se dan dentro del plan. Por este motivo, este inciso deberá resolverse en la forma más general posible.

Por definición, se dice que un plan es *recuperable* si toda transacción T hace COMMIT después de que lo hayan hecho todas las transacciones que escribieron algo que T lee. Por otro lado, se dice que *evita aborts en cascada* si toda transacción lee de ítems escritos por transacciones que hicieron COMMIT. Por último, es *estricto* si toda transacción lee y escribe ítems escritos por transacciones que hicieron COMMIT.

Como resultado, debe devolverse el nivel de recuperabilidad y, en caso de haber conflictos, un par de transacciones que estén involucradas en el mismo (si hay más de una, devolver cualquiera) además de indicar el motivo del conflicto.

Consideraciones

El código entregado por la cátedra contará con una interfaz gráfica que les servirá como ayuda a la hora de armar planes y de testear su implementación. Además, se les proveerán casos de test con los resultados esperados para que previo a la entrega puedan realizar los chequeos correspondientes. Por último, para la resolución deberán tener en cuenta las siguientes consideraciones:

- Las modificaciones introducidas deben limitarse a las clases dentro del paquete ***ubadb.tools.scheduleAnalyzer***. En caso de considerar necesario realizar modificaciones extra, deberán consultarlo previamente.

- Para la corrección no sólo se evaluará que lo implementado funcione correctamente sino también la calidad del código generado. Con calidad nos referimos a usar comentarios, nombre declarativo para las variables o métodos, uso de métodos auxiliares, etc.

- Los métodos ya existentes deben conservar la misma aridad y, en caso de creer necesario realizar una modificación, se deberá consultar previamente.

- Debido a que la idea es seguir evolucionando con este motor, se pide que se intente respetar las convenciones (tanto de nombres de paquetes, métodos, uso de excepciones, etc) para poder contar con código lo más uniforme posible.

Entrega

La entrega deberá contar con el código que implementa lo pedido, un breve informe con detalles de implementación, decisiones tomadas y todo lo que crean conveniente y finalmente, casos de test *propios* (incluye el plan de entrada y los resultados esperados) para la demo que se realizará en los laboratorios el día de la entrega.