Bases de Datos

Segunda Parte - Trabajo Práctico

Fecha de Entrega: 23/06/2010
Fecha de Recuperatorio: 16/07/2010

Enunciado

En la segunda parte del Trabajo Práctico, el objetivo será desarrollar una herramienta vinculada a la detección de *deadlocks* en una base de datos.

La administración de recursos es un problema clásico en los sistemas y las bases de datos no son la excepción. En este trabajo nos centraremos en historias de transacciones y se implementarán diferentes esquemas para intentar resolver el problema de los *deadlocks*, basándose en el modelo de *locking binario* (LOCK – UNLOCK).

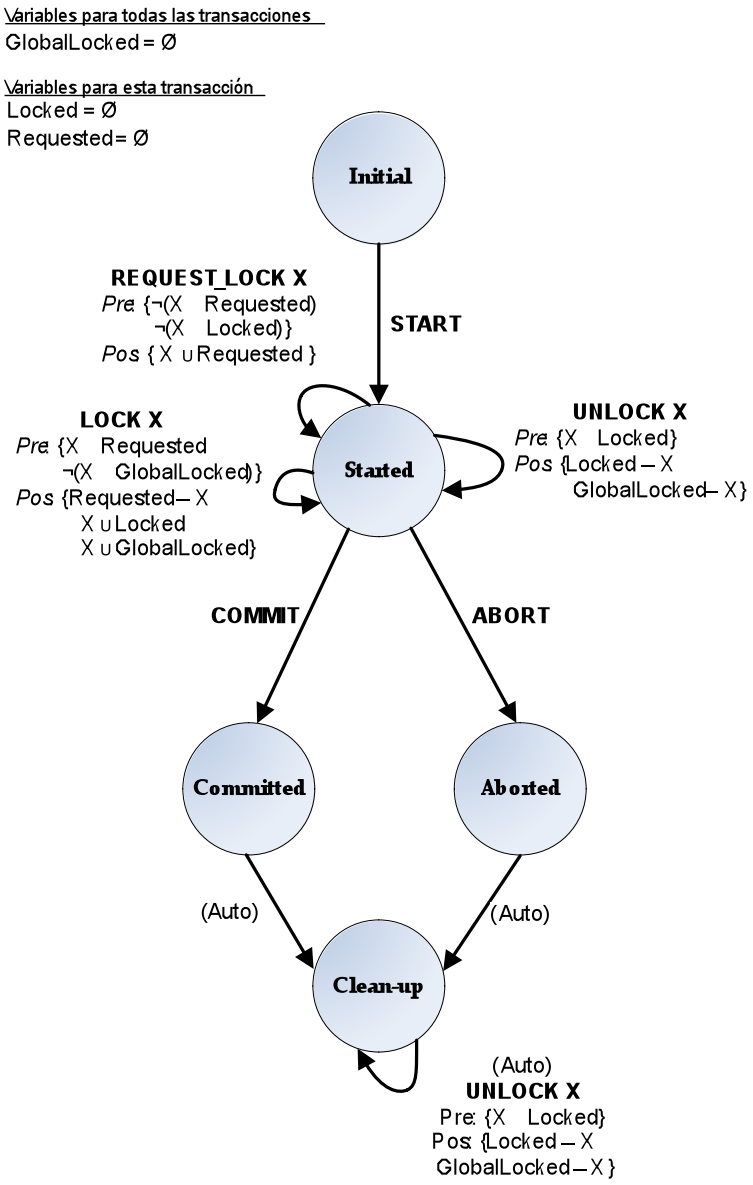
1) **Extender el modelo de *locking binario***

Actualmente, la herramienta soporta las directivas básicas del modelo binario: START, COMMIT, LOCK y UNLOCK. En este punto, deberán extenderlo para que permita modelar la solicitud de un *lock* (REQUEST_LOCK) y la finalización abrupta de una transacción (ABORT), creando las estructuras necesarias para incorporar estas directivas.

Cabe aclarar que estos cambios deben reflejarse también en la interfaz gráfica de la herramienta.

2) **Generación de historias válidas**

Una vez extendido el modelo, se pide agregar la lógica necesaria para que desde la herramienta solamente puedan crearse historias válidas. Para que una historia sea válida, cada transacción deberá seguir el diagrama de estado que se describe a continuación.



3) **Determinar si una historia cumple o no 2PL o S2PL**

Implementar los algoritmos correspondientes para determinar si un *schedule* dado cumple o no los protocolos 2PL (Two Phase Locking) o S2PL (Strict Two Phase Locking).

A través de la herramienta, se deberá poder chequear estas propiedades en los *schedules* generados y, en caso de existir violaciones, el resultado deberá describir detalladamente los motivos por los cuales no se cumple alguna propiedad.

NOTA: Solamente se deberá analizar a las transacciones que tengan su correspondiente COMMIT. Es decir, no debe hacerse nada con las transacciones abortadas o incompletas.

4) **Implementar algoritmos de prevención y detección de *deadlocks***

Existen diversas formas de atacar el problema de los deadlocks. Dos de los enfoques más conocidos consisten en la prevención y detección de deadlocks.

Una forma de prevenir los deadlocks consiste en asignarle a cada transacción un *timestamp*. Este dato servirá para determinar la prioridad de las transacciones: a menor *timestamp*, mayor prioridad. Utilizando lo anterior, si una transacción T_i solicita un *lock* sobre un objeto tomado por T_j: si la prioridad de T_i es mayor que la de T_j, se le permite esperar el lock; si no, T_i es abortada. Esta forma hará que sea imposible formar un *deadlock* en la base de datos.

Otro esquema más flexible consiste en no imponer restricciones sobre las solicitudes de los *locks*, pero sí verificar periódicamente la existencia de *deadlocks*. Es posible detectar un *deadlock* armando un grafo con las siguientes características: los nodos serán las transacciones activas y habrá un arco desde el nodo T_i hacia el nodo T_j si la transacción T_i está esperando un *lock* sobre un objeto tomado por la transacción T_j. La existencia de un ciclo determinará que el sistema se encuentra en un estado de *deadlock*.

En este punto se pide:

a. Implementar los esquemas descriptos anteriormente para la prevención y detección de *deadlocks*. Para la prevención de *deadlocks*, deberán permitir la generación de historias que cumplan con las restricciones ya mencionadas; mientras que para la detección deberán permitir desde la aplicación que el usuario corra el proceso para detectar *deadlocks* durante la generación del *schedule*.

b. Responder las siguientes preguntas:
   i. ¿El protocolo 2PL garantiza que no existan *deadlocks*?
   ii. ¿Y con S2PL qué sucede?
   iii. Comparar la prevención/detección de *deadlocks*, mencionando ventajas y desventajas de cada uno.

Consideraciones

El código entregado por la cátedra contará con una interfaz gráfica que les servirá como ayuda a la hora de armar historias 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 ***ubadbtools.deadlockAnalyzer***. 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 aumentando la funcionalidad de esta herramienta, 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.

- La interfaz gráfica cuenta con la funcionalidad básica y es bienvenida cualquier modificación que tienda a mejorarla.

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 la entrada y los resultados esperados) para la demo que se realizará en los laboratorios el día de la entrega.

Bibliografía

- "Database Management Systems", Raghu Ramakrishnan – Johannes Gehrke, 2° Edición