# *INSTITUTO POLITÉCNICO NACIONAL*

## CENTRO DE INVESTIGACIÓN EN COMPUTACIÓN

Tesis de Doctorado

# "Modelo de Control de Concurrencia basado en bloqueos con nivel de aislamiento Lecturas-No-Confirmadas para Transacciones Anidadas"


Presentada por :

**Luis Antonio Gama Moreno**

Director:

**Dr. José Matías Alvarado Mentado**


México, D.F., Septiembre de 2007

SIP-14

# INSTITUTO POLITÉCNICO NACIONAL

## SECRETARÍA DE INVESTIGACIÓN Y POSGRADO

*ACTA DE REVISIÓN DE TESIS*

En la Ciudad de México, D. F. Siendo las 12:00 horas del día 08 del mes de Febrero de 2007 se reunieron los miembros de la Comisión Revisora de Tesis designada por el Colegio de Profesores de Estudios de Posgrado e Investigación del:

***Centro de Investigación en Computación***

para examinar la tesis de grado titulada:

**"MODELO DE CONTROL DE CONCURRENCIA BASADO EN BLOQUEOS CON NIVEL DE AISLAMIENTO LECTURAS-NO-CONFIRMADAS PARA TRANSACCIONES ANIDADAS"**

Presentada por la alumno:

| **GAMA** | **MORENO** | **LUIS ANTONIO** |
|---|---|---|
| Apellido paterno | Materno | nombre(s) |

Con registro:

| A | 0 | 2 | 0 | 2 | 6 | 6 |
|---|---|---|---|---|---|---|

aspirante al grado de: ***DOCTORADO EN CIENCIAS DE LA COMPUTACIÓN***

Después de intercambiar opiniones los miembros de la Comisión manifestaron ***SU APROBACIÓN DE LA TESIS***, en virtud de que satisface los requisitos señalados por las disposiciones reglamentarias vigentes.

LA COMISIÓN REVISORA

Presidente

DR. IGOR ALEXEYEVICH BOLSHAKOV

Secretario

DR. CARLOS FERNANDO AGUILAR IBAÑEZ

Primer vocal
(Director de Tesis)

DR. JOSÉ MATÍAS ALVARADO MENTADO

Segundo vocal

DR. SERGIO SUAREZ GUERRA

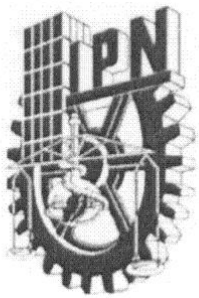
Tercer vocal

DR. HÉCTOR BENÍTEZ PÉREZ

EL PRESIDENTE DEL COLEGIO

DR. HUGO CÉSAR COYOTE ESTRADA

INSTITUTO POLITECNICO NACIONAL
CENTRO DE INVESTIGACION
EN COMPUTACION
DIRECCION



# *INSTITUTO POLITECNICO NACIONAL*

*SECRETARIA DE INVESTIGACIÓN Y POSGRADO*

*CARTA CESION DE DERECHOS*

En la ciudad de México el día 4 de Septiembre del año 2007, el (la) que suscribe **Luis Antonio Gama Moreno** alumno (a) del Programa de **Doctorado en Ciencias de la Computación** con numero de registro **A020266**, adscrito al **Centro de Investigación en Computación**, manifiesta que es autor (a) intelectual del presente trabajo de Tesis bajo la dirección de **Dr. José Matías Alvarado Mentado** y cede los derechos del trabajo intitulado *"**Modelo de Control de Concurrencia basado en bloqueos con nivel de aislamiento Lecturas-No-Confirmadas para Transacciones Anidadas**"*, al Instituto Politécnico Nacional para su difusión, con fines académicos y de investigación.

Los usuarios de la información no deben reproducir el contenido textual, graficas o datos del trabajo sin el permiso expreso del autor y/o director del trabajo. Este puede ser obtenido escribiendo a la siguiente dirección **lgama@mail.itzacatepec.edu.mx**. Si el permiso se otorga, el usuario deberá dar el agradecimiento correspondiente y citar la fuente del mismo.

**Luis Antonio Gama Moreno**

Nombre y firma

# “Modelo de Control de Concurrencia basado en bloqueos con nivel de aislamiento Lecturas-No-Confirmadas para Transacciones Anidadas”

*Candidato a doctor:* ***Luis Antonio Gama Moreno***
*lgama@mail.itzacatepec.edu.mx*
*Director de tesis: Dr. Matías Alvarado Mentado*
*matias@delta.cs.cinvestav.mx*

**Doctorado en Ciencias de la Computación**
Centro de Investigación en Computación
**Instituto Politécnico Nacional**

***Resumen:*** Nuestro modelo de control de concurrencia extendido para transacciones anidadas, CCxTA, se basa en el protocolo 2PL (Two-Phase Lock), y las principales ventajas de utilizarlo son: 1) las transacciones concurrentes pueden trabajar con datos no confirmados por otra transacción sin caer en inconsistencias y 2) lograr un número significativo de transacciones anidadas, cerradas y abiertas, terminadas con éxito utilizando el nivel de aislamiento *read uncommitted,* el más relajado –asimismo, se minimizan los abrazos mortales. Al relajar el nivel de aislamiento de los datos en operaciones de lectura y escritura, al máximo, de manera que sea posible para transacciones concurrentes utilizar datos sin confirmar que están siendo utilizados por otra transacción, pueden ocurrir malas dependencias tales como lecturas sucias, fantasmas, actualizaciones perdidas o lecturas no-repetibles, las cuales pueden generar inconsistencias en los datos. Para administrar la concurrencia de transacciones anidadas cerradas/abiertas, se ha implementado un Monitor, el cual vigila la ocurrencia de malas dependencias y opera para neutralizar los efectos de inconsistencia: si ocurre una mala dependencia generando inconsitencias, se envían mensajes a las transacciones involucradas indicándoles las cancelaciones necesarias, parciales la mayoría de las veces, así como las instrucciones para rehacer las operaciones canceladas. Procediendo de esta manera, se obtiene un mayor número de transacciones terminadas con éxito, al no cancelar indiscriminadamente sino solo en función de los tiempos límite asignados a las operaciones involucradas. Por la robustez que proporcionan frente a fallos de consistencia de datos al operar transacciones concurrentes, el modelo CCxTA y el Monitor tiene aplicabilidad en transacciones que se ejecutan sobre redes inalámbricas de dispositivos móviles y en transacciones de larga duración, así como en Flujo de Trabajo (*Workflow) transaccional*.

**Palabras claves:** *control de concurrencia, transacciones anidadas, relajar nivel de aislamiento, aislamiento read uncommitted, incremento de transacciones terminadas con éxito.*

**Abstract**

Our extended concurrency control model for nested transactions (ECCNT) is based on 2PL (Two-Phase Lock) protocol, and its main advantages are: 1) concurrent transactions are able to work with non-committed data without obtaining inconsistencies and 2) a significant number of committed nested transactions (closed or opened) can be achieved by using the isolation level read-uncommitted, the most relaxed – and in this manner, deadlocks are reduced. When the isolation level of data on reading and writing operations is relaxed, it is possible for concurrent transactions to get access to uncom-mitted data used by one or more prior transactions; however, granting access to uncommitted data could produce bad dependencies such as dirty-reads, phantoms, lost-updates and unrepeatable reads, generating inconsistent data at the end of the concurrent transactions execution. In order to manage the concurrency of closed/opened nested transactions a Monitor has been deployed. This monitor seeks out bad dependencies and neutralizes the effects of bad dependencies: if a bad dependency occurs that generates inconsistencies, messages are sent to involved transactions indicating the necessary actions to cancel, partial in most of the cases, as well as the instructions to redo the actions that have been cancelled. By using this approach, a greater number of committed transactions are achieved, without cancelling indiscriminately but only in function of the timeout assigned to the operations involved. Thanks to the robustness of the ECCNT model as well as of the Monitor that verifies the consistency of data in concurrent transactions, they can be applied throughout transactions executed on mobile devices over wireless networks and during very long transactions as well as on so-called Transactional WorkFlow.

Dedico esta tesis a mis hijos:
*Yael Ramses y Sergio Antonio, por darme su amor.*

Doy las gracias a todas las personas que me apoyaron y motivaron durante este camino. A DIOS por darme el ser; a mis padres Ernestina Moreno Garcés y Francisco Gama Martínez, por darme la vida y por todo el apoyo brindado.

A mi director de tesis, Dr. José Matías Alvarado Mentado por ser mi mentor. A mis sinodales, Dr. Igor Bolshakov, Dr. Hugo Coyote Estrada, Dr. Héctor Benítez Pérez, Dr. Sergio Suárez Guerra, Dr. Carlos Aguilar Ibañez, por guiarme con sus conocimientos.

Agradezco el apoyo de aquellos que con sus consejos y ánimos ayudaron a alcanzar esta meta: a mi padrino Jorge Martínez Gómez, mis amigos: Pamela Ríos, Karen Lusnia, Jorge Martínez (el negro), Alma Delia Cuevas, Antonio Armenta, Rolando Quintero, Leandro Balladares, Giovanni Guzmán, Boris Aranda, Claudia Noguerón. A mis alumnos que con su energía y juventud ayudaron a que el camino fuera más divertido: Edgar García Núñez, Raúl García Flores (1ª generación), Adriana Martínez Benítez y Tere Robledo Villagomez (2ª generación). Al Instituto Tecnológico de Zacatepec, por darme el soporte y la oportunidad de crecer profesionalmente. Al CONACYT por el apoyo económico.

Y a todas las demás personas que han contribuido con un "granito de arena" para que hoy pueda llegar a la meta.

A todos ustedes, MUCHAS GRACIAS.

***Luis Antonio Gama Moreno***
Septiembre de 2007

# Índice

**Capítulo 1 Introducción ..... 4**
1.1 El problema: bajo desempeño en control de concurrencia basado en bloqueos ..... 4
1.2 Objetivos ..... 4
1.3 Propuesta de solución: relajar nivel de aislamiento sin caer en inconsistencias ..... 5
1.4 Metodología ..... 6
1.5 Resultados ..... 11
**Capítulo 2 Transacciones: Estado del arte ..... 13**
2.1 Limitaciones ..... 13
2.2 Modelos ..... 14
2.3 Control de concurrencia ..... 19
2.4 Teoría del aislamiento ..... 24
2.5 Malas dependencias ..... 26
2.6 Niveles de aislamiento ..... 29
2.7 Compatibilidad en DBMS comerciales ..... 31
**Capítulo 3 Control de concurrencia extendido ..... 34**
3.1 Análisis del modelo ..... 34
3.2 Diseño en UML ..... 35
**Capítulo 4 Implementación: administración de malas dependencias ..... 49**
4.1 Transacciones Anidadas Cerradas ..... 49
4.2 Algoritmo para administración de abortos ..... 54
4.4 Transacciones Anidadas Abiertas ..... 64
4.5 Algoritmo para administrar Transacción de Compensación ..... 66
4.6 Control de Concurrencia Extendido ..... 70
**Capítulo 5 Pruebas y resultados ..... 73**
5.1 Diseño de las pruebas ..... 73
5.2 Resultados ..... 74
5.3 Prueba de correctez ..... 82
**Capítulo 6 Ventajas y aplicabilidad ..... 84**
6.1 Tolerancia a defectos (Fault tolerance) ..... 84
6.2 Más transacciones terminadas con éxito ..... 84
6.3 Disminución de fallas ..... 84
6.4 Comparación con otros modelos de control de concurrencia ..... 84
**Capítulo 7 Aplicaciones ..... 86**
7.1 Transacciones Anidadas Móviles ..... 86
7.2 Workflow transaccional ..... 86
**Aportaciones y conclusiones ..... 89**
**Glosario de términos ..... 91**
**Referencias ..... 92**

# Índice de figuras

Figura 1-1 Herencia de un bloqueo hacia los padres. 7
Figura 1-2 Herencia del bloqueo a todos los niveles. 8
Figura 1-3 Lectura sucia. 9
Figura 1-4 Grafo de dependencias entre Ti y Tj. 9
Figura 1-5 Aborto parcial hasta un estado consistente. 10
Figura 2-1 Utilizando savepoints dentro de una transacción. 15
Figura 2-2 Estado final de la transacción con savepoints. 16
Figura 2-3 Representación gráfica de las transacciones anidadas. 17
Figura 2-4 Arquitectura de una Transacción Móvil. 19
Figura 2-5 Grafos de dependencias. 25
Figura 2-6 Grafo de dependencia de la lectura sucia. 26
Figura 2-7 Grafo de dependencia de la lectura no-repetible. 27
Figura 2-8 Grafo de dependencias de *Phantom*. 28
Figura 2-9 Grafo de dependencia de la actualización perdida. 28
Figura 2-10 Nivel de aislamiento y sus efectos en la consistencia y concurrencia. 30
Figura 3-1 Arquitectura de JDBC. 34
Figura 3-2 Arquitectura de un escenario de Transacciones Anidadas Cerradas. 35
Figura 3-3 Diagrama de componentes para Transacciones Anidadas Cerradas. 36
Figura 3-4 Diagrama de casos de uso de una Transacción Anidada Cerrada. 37
Figura 3-5 Diagrama de clases. 38
Figura 3-6 Clase ConnectionDB. 39
Figura 3-7 Clase Transaction. 40
Figura 3-8 Capturando la excepción `TransactionException`. 41
Figura 3-9 Clase `TransactionException`. 42
Figura 3-10 Capturando la excepcion `PartialRollbackTransaction`. 43
Figura 3-11 Clase `PartialRollbackTransaction`. 43
Figura 3-12 Clase `TransactionThread`. 44
Figura 3-13 Diagrama de secuencias de una transacción anidada. 46
Figura 3-14 Secuencia de eventos entre una aplicación y el monitor. 47
Figura 4-1 Herencia de un bloqueo hacia los padres. 50
Figura 4-2 Herencia del bloqueo a todos los niveles. 51
Figura 4-3 Lectura sucia. 52
Figura 4-4 Grafo de dependencias entre $T_i$ y $T_j$. 53
Figura 4-5 Aborto parcial hasta un estado consistente. 53
Figura 4-6 Rollback hasta el Savepoint 2. 54
Figura 4-7 Monitoreo de transacciones concurrentes. 55
Figura 4-8 Código para lanzar una Transacción tolerante a fallas. 58
Figura 4-9 Creación de *savepoints* por cada instrucción. 58
Figura 4-10 Obteniendo llaves primarias. 61
Figura 4-11 Creando identificadores por objeto. 62
Figura 4-12 Estructura de la tabla dept_updated_objects. 63
Figura 4-13 Instrucción antes de su re-definición. 63
Figura 4-14 Instrucción sql después de su re-asignación. 64
Figura 4-15 Compensación de Ti después del aborto de Tj. 65

Figura 4-16 Aborto de Tj generando malas dependencias. ..... 66
Figura 4-17 Compensación parcial. ..... 68
Figura 4-18 Algoritmo para deshacer una lectura-sucia. ..... 70
Figura 4-19 Evitando lecturas sucias. ..... 71
Figura 4-20 Problemas en la dependencia write-read. ..... 72
Figura 4-21 Evitando los efectos de aborto en dependencias write-read. ..... 72
Figura 5-1 Serialización de la historia. ..... 75
Figura 5-2 Monitoreo de las Ts con el nivel de aislamiento READ COMMITTED. ..... 75
Figura 5-3 Transacciones T1 y T3 acceden a un objeto bloqueado por T5 ..... 76
Figura 5-4 Cantidad de cancelaciones ejecutando 50 transacciones concurrentes. ..... 77
Figura 5-5 Cantidad de cancelaciones ejecutando 100 transacciones concurrentes. ..... 78
Figura 5-6 Cantidad de cancelaciones ejecutando 500 transacciones concurrentes. ..... 79
Figura 5-7 Ejecución con dos procesadores. ..... 80
Figura 5-8 Comparación de resultados entre en un sistema uni-procesador y un multi-procesador. ..... 81
Figura 5-9 Promedio de cancelaciones entre sistemas uni-procesamiento y multi-procesamiento. ..... 81
Figura 5-10 Serialización de operaciones en conflicto. ..... 83
Figura 6-1 Comparación de CCxTA vs. JDBC. ..... 85
Figura 7-1 Validación de A, a pesar del aborto de B. ..... 86
Figura 7-2 Grupo de tareas bajo el contexto de workflow. ..... 87
Figura 7-3 Arquitectura de workflow transaccional. ..... 87
Figura 7-4 Cancelación de todo el proceso debido a la Atomicidad. ..... 88
Figura 7-5 Ejecución de una cancelación parcial. ..... 88

# Índice de tablas

Tabla 1 Matriz de compatibilidad entre bloqueos. ..... 21
Tabla 2 Niveles de aislamiento y su relación con las malas dependencias. ..... 30
Tabla 3 Compatibilidad con DBMS comerciales. ..... 33
Tabla 4 Niveles de prioridad. ..... 59
Tabla 5 Descripción de la tabla dept. ..... 61
Tabla 6 Consulta a tabla dept. ..... 61
Tabla 7 Identificación de objetos. ..... 62
Tabla 8 Incluyendo la llave primaria. ..... 62
Tabla 9 Descripción de atributos agregados a la tabla "dept_updated_objects". ..... 63

# Capítulo 1 Introducción

El procesamiento de transacciones ha evolucionado pasando de ambientes centralizados a distribuidos y hoy día en ambientes móviles. Sin embargo, el cómputo móvil se caracteriza por sus características inestables, tales como: desconexiones frecuentes (y en ocasiones anunciadas), asimetría en el ancho de banda, limitaciones en energía, poco espacio de almacenamiento, tamaño del display pequeño, entre otros [2, 10, 24]. Por lo tanto, se requieren de nuevos modelos para el procesamiento de transacciones que sean capaces de disminuir los efectos de tales características [13].

En [18, 26] se describen diversos modelos para el procesamiento de transacciones móviles, enfocados en los efectos sobre las propiedades ACID (*Atomicity, Consistency, Isolation* y *Durability*) y sobre el modelo de ejecución. En [15] se ha desarrollado un modelo de transacciones para la plataforma móvil, proponiendo nuevos protocolos de validación y presentando un tratamiento formal para el manejo de datos dependientes de su ubicación.

## 1.1 El problema: bajo desempeño en control de concurrencia basado en bloqueos

En un ambiente de procesamiento de transacciones concurrentes, el rendimiento de estas se verá afectado por el tiempo de espera al que son sometidas, debido a los mecanismos de control de concurrencia basados en bloqueo, con el objetivo de garantizar la consistencia de los datos [16].

Los protocolos basados en bloqueo tales como 2PL (Two-Phase Lock) retardan las operaciones que pueden causar conflicto entre dos o más transacciones, serializando su ejecución para evitar efectos conocidos como malas dependencias (*lecturas sucias*, *fantasmas, actualizaciones perdidas* o *lecturas no-repetibles*) [14, 20]. Sin embargo, este mecanismo de serialización disminuye el rendimiento de las aplicaciones debido a los bloqueos, ya que las transacciones tienen que esperar para tener acceso a objetos que se encuentren actualmente en proceso. Además la presencia de abrazos mortales, es latente. Por otro lado, si el procesamiento de transacciones es llevado a cabo mediante dispositivos móviles, entonces se incrementan las fallas debido a la inestabilidad característica de los ambientes móviles.

## 1.2 Objetivos

### Objetivo principal

Incrementar el número de transacciones concurrentes terminadas con éxito, relajando el nivel de aislamiento a *read uncommitted*, y aplicando cancelaciones parciales para deshacer los efectos de inconsistencias generadas por malas dependencias; y de esta manera garantizar la consistencia de los datos y disminuir la presencia de abrazos mortales.

**Objetivos específicos**

- Extender el modelo de control de concurrencia del protocolo 2PL utilizado en Transacciones Anidadas, para los siguientes tipos de operaciones:

  **1.** *Lectura:* extender la disponibilidad de los objetos bloqueados de manera compartida (shared lock: slock) a todos los niveles del árbol de transacciones [11].

  **2.** *Escritura:* relajar el nivel de aislamiento para permitir el acceso a objetos previamente bloqueados, y en el caso de malas de dependencias resolver el conflicto mediante abortos parciales.

- Creación de un API (*Application Program Interface*) para el procesamiento de transacciones anidadas *(Cerradas y Abiertas)*.

- Diseño y construcción de un Monitor de Transacciones Anidadas (MTA) Cerradas/Abiertas, para vigilar y resolver la presencia de malas dependencias producto de la relajación del nivel de aislamiento entre las transacciones concurrentes.

## 1.3 Propuesta de solución: relajar nivel de aislamiento sin caer en inconsistencias

Se propone utilizar el modelo de transacciones anidadas (TA) [19] para mejorar la tolerancia a fallas, debido a sus características tales como:

- La creación de un árbol jerárquico de transacciones (llamadas sub-transacciones) distribuyendo el procesamiento.

- La posibilidad de validar la transacción con aquellas sub-transacciones exitosas aún cuando existan fallas en alguna de ellas.

- Mayor rendimiento de las aplicaciones, debido a que las sub-transacciones de una rama no tienen que esperar hasta la finalización de las transacciones en otras ramas del árbol.

El modelo descrito en [19], es conocido también como modelo de Transacciones Anidadas Cerradas (TAC), debido a que las sub-transacciones no son capaces de confirmar el trabajo realizado hasta que la transacción raíz, decida confirmar. Si la transacción raíz decide abortar, entonces todas las sub-transacciones deberán abortar también. Sin embargo bajo el modelo TAC los recursos no son liberados hasta que la transacción raíz decida terminar, dejando objetos bloqueados por periodos largos de tiempo. Para ello se definió el modelo de Transacciones Anidadas Abiertas (TAA) en donde las sub-transacciones no tienen que esperar al padre para terminar la transacción, sino que van terminando y

liberando los recursos bloqueados, esto permite mayor rendimiento a las transacciones ya que no tienen que esperar por objetos bloqueados del modelo (TAC).

El control de concurrencia del modelo TAC/TAA está basado en el protocolo 2PL con herencia. Una desventaja es que otras transacciones que soliciten bloqueos a objetos previamente bloqueados, tengan que esperar hasta que estos objetos sean liberados o sean administrados por su transacción-padre. Por lo tanto, se propone extender el control de concurrencia de 2PL de la siguiente manera:

1. No bloquear los objetos sometidos a operaciones de lectura y extender así su visibilidad a todos los niveles del árbol de transacciones [11]. Esto se basa en la compatibilidad de los bloqueos para lectura, debido a que: *dos lecturas de diferentes transacciones no cambian el estado de los objetos* [5, 14].

2. Para operaciones de escritura, relajar el nivel de aislamiento al de menor restricción (read uncommitted) [1] para permitir el acceso a objetos previamente bloqueados. En el caso de presentarse malas dependencias tales como: *lecturas sucias*, *actualizaciones perdidas* o *lecturas no-repetibles*, se resuelve el conflicto llevando a cabo un aborto parcial, hasta el último estado consistente antes del surgimiento de la mala dependencia. Esto es posible gracias al uso de *savepoints,* el cual permite ir guardando estados consistentes de una transacción, y poder llevar a cabo cancelaciones parciales, sin tener que abortar una transacción completa debido a cualquier falla.

Con la implementación de estas extensiones, se conseguirá mayor rendimiento en las aplicaciones y un mayor número de transacciones finalizadas con éxito.

## 1.4 Metodología

A continuación se describe la metodología para extender el control de concurrencia del protocolo 2PL utilizado en este modelo para operaciones de lectura y/o escritura.

### 1.4.1 Operaciones de solo-lectura

En TA, los bloqueos se van adquiriendo por dos opciones, (1) si el objeto solicitado está libre o (2) si ha sido bloqueado previamente y el padre de la transacción solicitante lo administra. Sin embargo, con este enfoque, si una transacción A solicita el acceso a un objeto que ha sido bloqueado por una transacción B en diferentes ramas del árbol, la transacción A tendrá que esperar hasta que el objeto sea liberado o heredado a su padre inmediato. Este mecanismo de bloqueos con herencia garantiza el aislamiento pero disminuye el rendimiento de las aplicaciones.

La figura 1-1 muestra un ejemplo de este escenario. En la figura 1-1a la transacción E solicita un bloqueo compartido sobre el objeto *q*. Posteriormente la transacción C solicita un bloqueo compartido sobre el mismo objeto *q* bloqueado por E (ver figura 1-1b). Entonces C debe esperar hasta que el objeto *q* sea liberado y administrado por el predecesor

de C (transacción A). Las figuras 1-1c y 1-1d muestran el proceso de herencia hacia los padres, por parte de las sub-transacciones E y B al momento de terminar ya sea con *commit* o *abort*.

Finalmente en la figura 1-1e se muestra la adquisición del bloqueo sobre *q* solicitado por la transacción C. Para evitar este tiempo de espera, se propone extender la visibilidad de un objeto bloqueado en modo compartido a todos los niveles del árbol. Para que cada transacción sea capaz de bloquear un objeto en modo compartido aún cuando éste ya haya sido bloqueado en el mismo modo.

**Nuestro enfoque se basa en la compatibilidad que tienen los bloqueos de lectura (slock) ya que estos no generan malas dependencias a pesar del aborto de las transacciones** [14]**.**

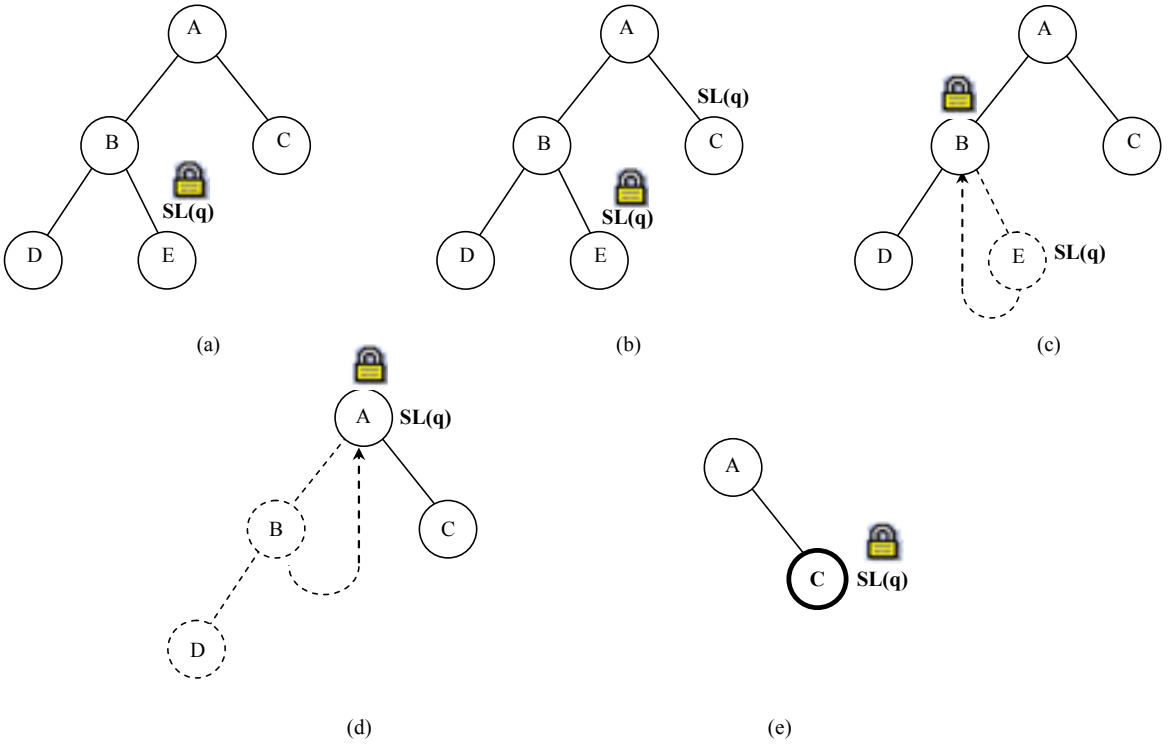



**Figura 1-1** Herencia de un bloqueo hacia los padres.

En la figura 1-2, se muestra el mismo escenario, donde la transacción E, bloquea en primer instancia al objeto *q*, heredando el bloqueo al resto de las transacciones. Es decir, el objeto *q* permanece disponible a otras transacciones aún cuando E aborte. Entonces la transacción C es capaz de obtener un bloqueo compartido sobre el objeto *q,* aún cuando E no haya terminado (ver figura 1-2c). Con este enfoque, las aplicaciones mejorarán su rendimiento para aquellas transacciones de solo-lectura bajo el modelo de TA, debido a

que, como ya se ha dicho, la compatibilidad de los bloqueos compartidos, no genera malas dependencias aún cuando las transacciones involucradas tengan que abortar.

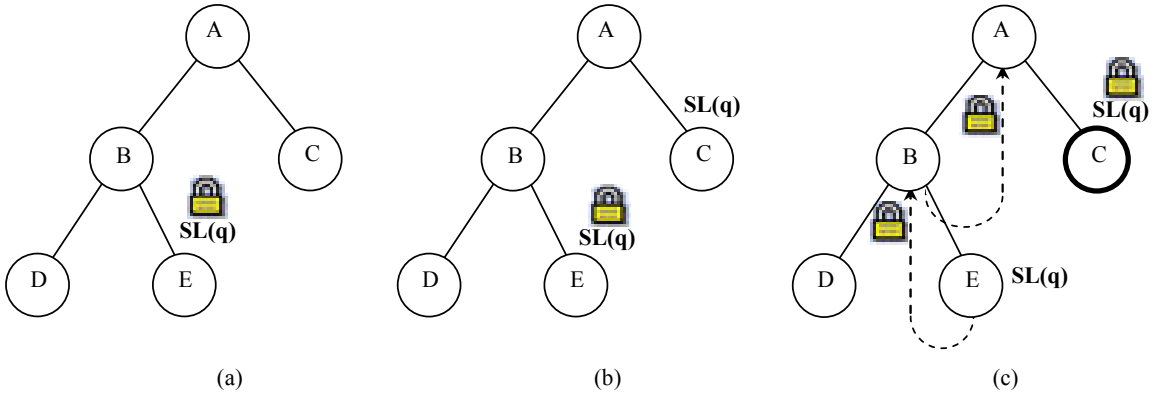



**Figura 1-2** Herencia del bloqueo a todos los niveles.

### 1.4.2 Operaciones de escritura

Las operaciones de escritura bajo el protocolo 2PL requieren de la protección de un bloqueo exclusivo (exclusive lock: xlock), debido a que cambian el estado de los objetos. Los bloqueos exclusivos no permiten a otras transacciones acceder a un recurso bloqueado, hasta que éste sea liberado. Por lo tanto, las aplicaciones disminuyen su rendimiento debido a los tiempos de espera. La norma ANSI-SQL92 [1] ha definido cuatro niveles de aislamiento con el propósito de relajar el aislamiento de las transacciones y mejorar el rendimiento de las aplicaciones.

Utilizando un nivel de aislamiento menos restrictivo, es posible aumentar el rendimiento de una aplicación debido a que no es necesario que ésta tenga que esperar hasta que los recursos sean liberados. En varios DBMS (*DataBase Management System)* comerciales, el nivel de aislamiento por defecto es *READ COMMITTED.* Este nivel evita la presencia de *lecturas sucias,* debido a que una transacción sólo puede leer valores confirmados por otras transacciones.

### 1.4.3 Evitando abortos en cascada

Relajar el nivel de aislamiento de las transacciones concurrentes puede ocasionar malas dependencias. Por ejemplo, si existe una $T_i$ que lee el objeto x, $r_i(x) := x$, obteniendo la misma versión del objeto (versión 1), posteriormente una transacción $T_j$ modifica el mismo objeto x mediante una escritura cambiando la versión del objeto x, $w_j(x) := x_j$ (versión 2), como se ilustra en la figura 1-3.

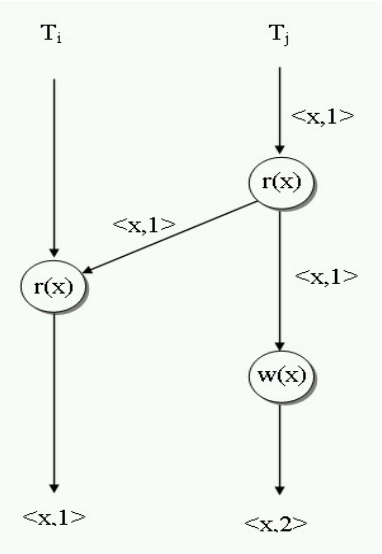



**Figura 1-3** Lectura sucia.

En la figura 1-4, las transacciones $T_i$ y $T_j$, tienen acceso al mismo objeto *x*. En primer instancia, $T_j$ realiza una lectura sobre x: $r_j(x) := x1$. Posteriormente $T_i$ realiza una lectura sobre x : $r_i(x) := x1$ sin cambiar el valor, en este momento $r_i(x) = r_j(x)$. La operación $r_i(x)$ se lleva acabo en el tiempo 1: *(t1)*. A continuación $T_j$ lleva a cabo una operación de escritura sobre el objeto x: $w_j(x) = x2$ (esta operación es permitida siempre y cuando $T_i$ y $T_j$ estén ejecutándose bajo el nivel de aislamiento *READ UNCOMMITED)*.

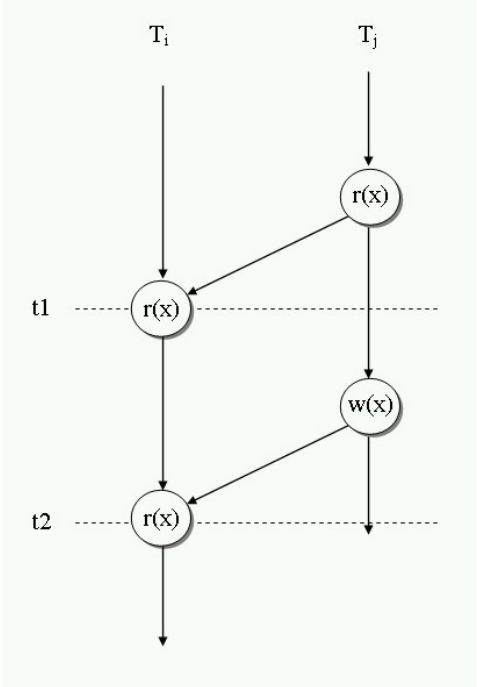



**Figura 1-4** Grafo de dependencias entre Ti y Tj.

En este momento, las versiones del objeto x son diferentes para las transacciones $T_i$ y $T_j$. $T_i$ mantiene el valor de la versión x1, mientras que $T_j$ mantiene el valor de la versión

x2. Por lo tanto $T_i$ debe abortar el trabajo realizado, debido a la mala dependencia generada (lectura-sucia) presentando un escenario de aborto en cascada.

Note que las operaciones de $T_i$ con el valor de la versión 1 (x1) son consistentes hasta que $T_j$ lo modifica a la versión x2. Es decir, $T_i$ es consistente hasta el tiempo *t1*, luego cuando sucede un cambio de versión del objeto x por la $T_j$ en el tiempo *t2,* $T_i$ deja de ser consistente.

### 1.4.4 Solución planteada: cancelaciones parciales

En la figura 1-5 se ilustra que sólo es necesario abortar las operaciones realizadas por $T_i$ después del tiempo t1, y posteriormente llevar a cabo una operación de redo() de aquellas operaciones canceladas.

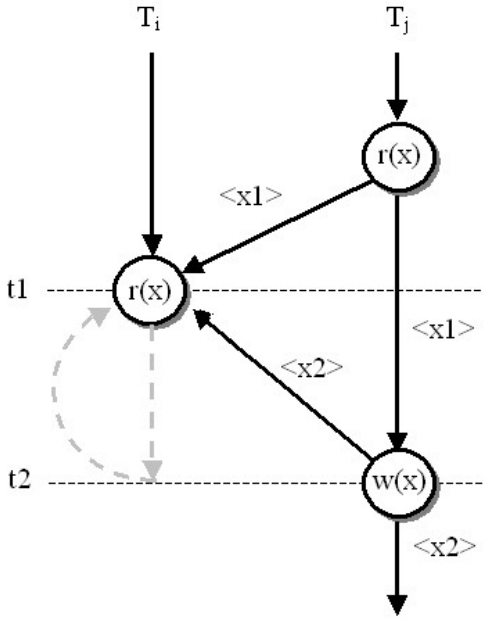



**Figura 1-5** Aborto parcial hasta un estado consistente.

De esta forma, $T_i$ no requiere del aborto total de la transacción, en su lugar, puede realizar un aborto parcial hasta el tiempo *t1*, cancelando sólo aquellas operaciones que se convirtieron en inconsistentes.

Con esta solución es posible obtener más transacciones terminadas con éxito, debido a las cancelaciones parciales. Además relajando el nivel de aislamiento se consiguen mas rendimiento de las transacciones concurrentes bajo un esquema de monitoreo de malas de dependencias.

## 1.5 Resultados

- Mejorar el rendimiento de las Transacciones concurrentes, empleando el modelo de control de concurrencia extendido para transacciones anidadas anteriormente.

- Disminuir la presencia de "abrazos mortales", relajando el nivel de aislamiento de las transacciones, para poder acceder a objetos previamente bloqueados aún cuando no hayan sido confirmados.

- Aumentar el número de transacciones concurrentes terminadas con éxito, evitando la cancelación total del trabajo realizado al tiempo de presentarse malas dependencias, mediante abortos parciales.

- La creación de una API para el procesamiento de transacciones anidadas con control de concurrencia extendido, de tal forma que los desarrolladores de aplicaciones distribuidas sean capaces de mejorar el rendimiento de sus aplicaciones. A continuación se describe un ejemplo de una transacción procesada bajo este modelo:

```
Transaction t = new Transaction()
try {
    t.begin();
    // se crea el savepoint 1 automáticamente
    t.doQuery("select * from emp where deptno = 10;");
    // se crea el savepoint 2 automáticamente
    t.doUpdate(" update emp " +
               " set sal = sal + 1000 "
               " where deptno = 10;");
    t.commit();
} catch( PartialRollbackException e){
  try {
     // obtener el ultimo savepoint en la transacción
     SavePoint sp = t.getLastSavepoint();

     // re-intentar las acciones
     if(sp.getValue() == 1) {
        // reintentar la lectura ...
        t.doQuery("select * from emp where deptno = 10;");
        t.doUpdate(" update emp " +
                   " set sal = sal + 1000 "
                   " where deptno = 10;");
     } else if(sp.getValue() == 2)
        t.doUpdate(" update emp " +
                   " set sal = sal + 1000 "
                   " where deptno = 10;");
     t.commit();
  } catch(TransactionException e) {
      t.rollback();
  }

} catch( TransactionException e) {
  // cancelar
  t.rollback();
```

```
    } finally {
      t.close()
    } // :~
```

- La creación de un monitor de transacciones anidadas con control de concurrencia extendido para el control y monitoreo de transacciones anidadas.

Utilizar este modelo en ambientes de **cómputo móvil** donde se requiere un mecanismo más robusto para la tolerancia a fallas.

# Capítulo 2 Transacciones: Estado del arte

Una transacción es una unidad de trabajo lógica, es decir, es un conjunto de operaciones de lectura/escritura vistas como una unidad indivisible. Así tenemos que, una transacción $T_i$ es un conjunto de operaciones sobre el que se define un orden parcial [5]:

$T_i \subseteq L \cup \{a_i, c_i\}$ donde:

1. $L$ es el conjunto de operaciones {$lectura_i(q)$, $escritura_i(q)$ donde $q$ es cualquier objeto[1]};
2. $a_i$ indica que la transacción no ha sido completada satisfactoriamente;
3. $c_i$ indica que la transacción fue completada satisfactoriamente;

- $a_i \in T_i$ *sii* $c_i \notin T_i$;
- $c_i \in T_i$ *sii* $a_i \notin T_i$;
- si $t = a_i \vee t = c_i$, entonces $\forall p \in T_i, p <_i t$;

si $lectura_i(q), escritura_i(q) \in T_i$, entonces ya sea que:

$lectura_i(q) <_i escritura_i(q) \vee escritura_i(q) <_i lectura_i(q)$.

Una transacción debe garantizar las propiedades ACID [5, 6], con el fin de garantizar la fiabilidad de los datos procesados por una transacción. La ***atomicidad*** asegura que todas las operaciones de una transacción conforman una unidad, es decir, o se ejecutan todas las operaciones o ninguna. La ***consistencia*** indica que una transacción debe llevar a una base de datos de un estado consistente a otro. El ***aislamiento*** garantiza que cada transacción observe un estado consistente de los datos, a pesar de que existan otras transacciones ejecutándose concurrentemente. Esta característica protege a las transacciones de los efectos de las actualizaciones de otras transacciones concurrentes. La ***durabilidad*** define que si una transacción es confirmada, los datos permanecen en un estado persistente, aún cuando sucedan fallas en el sistema. Es decir, una vez que la transacción termina con éxito, los datos pueden ser modificados sólo bajo el contexto de otra transacción (transacción de compensación).

## 2.1 Limitaciones

La propiedad atomicidad de las transacciones puede ser tanto una ventaja como una desventaja. Si algo sucede mal durante la ejecución de una transacción, se deben cancelar todas las operaciones o intentar rehacerlas para garantizar la atomicidad. Para transacciones

[1] El término objeto no sólo incluye objetos de almacenamiento como datos; también se refiere a objetos tales como, ventanas, menús, etc. del sistema operativo y a objetos reales: impresoras, taladros, reactores, puertas, etc. De tal forma que una lectura implica censar su estado, mientras que una escritura implica cambiar su estado.

cortas este enfoque es ideal pero no para aquellas de larga duración. De ahí se deriva el concepto de Transacción Plana (TP). Una TP es aquella que no puede ser dividida en piezas lógicas y validar o cancelar sólo ciertas partes de la transacción, ya que debe ser atómica.

## 2.2 Modelos

Actualmente, se han desarrollado esfuerzos para resolver ciertas limitaciones del modelo clásico de transacciones planas (TP). Las investigaciones han guiado a la definición de nuevos modelos que extienden las características del modelo tradicional. Tales como Transacciones con *savepoints* [5], transacciones encadenadas [14], transacciones anidadas [19], entre otros.

### 2.2.1 Planas

Una TP [14] es un bloque básico de operaciones en una aplicación las cuales no se pueden dividir; es decir, no existen partes lógicas de la transacción que puedan ser tratadas en forma individual, de tal forma que sea posible la validación o cancelación sólo de ciertas partes de la transacción. Esto debido a la propiedad de atomicidad. Tal característica es importante para los sistemas centralizados pero no es útil para las arquitecturas distribuidas. Por lo tanto, es una desventaja de las TP que no sea posible realizar validaciones o cancelaciones de pequeñas partes de la transacción; o validar resultados en varios pasos.

### 2.2.2 Planas con savepoints

El mecanismo de "*Savepoints"* permite ir guardando estados de la transacción que son válidos, para que en caso de fallas no tenga que abortar toda la transacción, sino hasta ciertos puntos (*savepoints)*. Se establece un *savepoint* cuando se invoca a la función SAVE WORK, esto hace que el sistema registre el estado actual del proceso de la transacción. Esta función retorna un identificador que posteriormente puede ser utilizado para reestablecer el estado hasta dicho identificador mediante la función ROLLBACK WORK, en lugar de indicar que toda la transacción sea abortada, sólo se indica el punto hasta donde deshacer los efectos realizados. La figura 2-1 muestra este esquema.

Al iniciar la transacción se crea un *savepoint:1* de manera implícita, las acciones 1 y 2 son protegidas por un nuevo *savepoint:2* indicando que ese bloque de instrucciones puede conservarse en caso de un aborto parcial. Sin embargo las acciones 3, 4, 5, 6 y 7 son canceladas debido al *ROLLBACK WORK(2),* indicando que se aborten las operaciones hechas después del *savepoint:2*. De igual manera sucede con las acciones 13 y 14, que son abortadas por la función *ROLLBACK WORK(7),* cancelando todas las operaciones realizadas después del *savepoint:7,* así hasta alcanzar el final de la transacción y su validación.

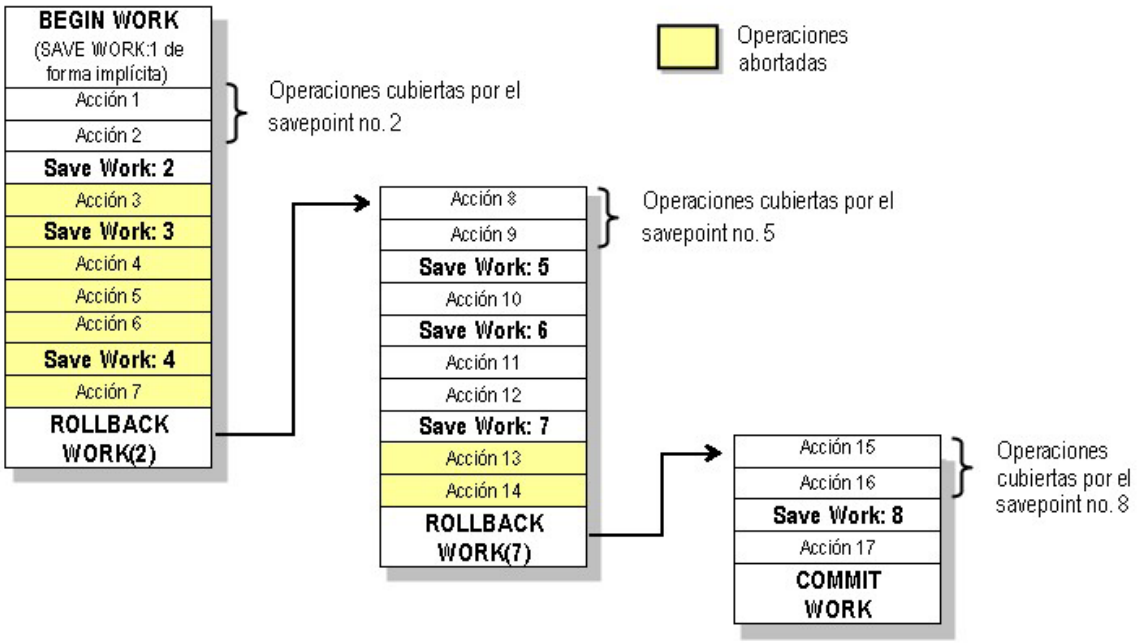



**Figura 2-1** Utilizando savepoints dentro de una transacción.

Note que la ejecución de un *ROLLBACK WORK(1),* es equivalente a ejecutar un *ROLLBACK* de toda la transacción. En la figura 2-2 se muestra el estado final de las acciones que se concluyeron con éxito al término de la transacción. Otro aspecto es que la numeración de los *savepoints* se va incrementando secuencialmente dentro de la transacción.

Podemos observar las siguientes características:

- Los *savepoints* pueden ser utilizados para estructurar una transacción dentro de una secuencia de ejecuciones parciales que pueden ser abortadas individualmente.

- Debido a que ninguna de las acciones que la transacción ha ejecutado son "visibles" antes del fin, éstas pueden ser abortadas en cualquier momento.

- Cuando la transacción es abortada hacia un *savepoint,* las acciones que son deshechas aparecen como si nunca se hubieran ejecutado.

- Cuando ocurre un error externo, la transacción entera es abortada, independientemente de los *savepoints* que existan.

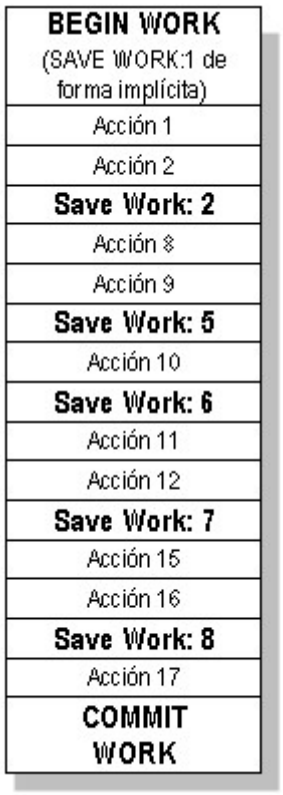



**Figura 2-2** Estado final de la transacción con savepoints.

### 2.2.3 Anidadas

El modelo de TA fue introducido por Moss [19]. En este modelo una transacción contiene un cierto número de sub-transacciones, y cada sub-transacción, a su vez, puede también contener sub-transacciones. La transacción completa forma un *árbol* de transacciones, donde la transacción de mayor nivel en el tope del árbol es llamada transacción *raíz*.

Una TA es un árbol de transacciones de profundidad arbitraria, donde los componentes son sub-transacciones que ejecuten partes del trabajo, cada sub-transacción es en sí una transacción plana. Las transacciones que tienen sub-transacciones son llamadas padres, y sus sub-transacciones se denominan hijas. Las transacciones que no poseen sub-transacciones son llamadas hojas [8, 19]. La figura 2-3 muestra una representación del modelo de TA.

Cada transacción comienza antes y termina después que sus hijas, por tanto,

- controla la ejecución de sus hijas,
- cada sub-transacción se ejecuta independientemente y,
- eventualmente en paralelo con las otras sub-transacciones, lo cual significa que,
- puede decidir validar o abandonar de forma independiente.

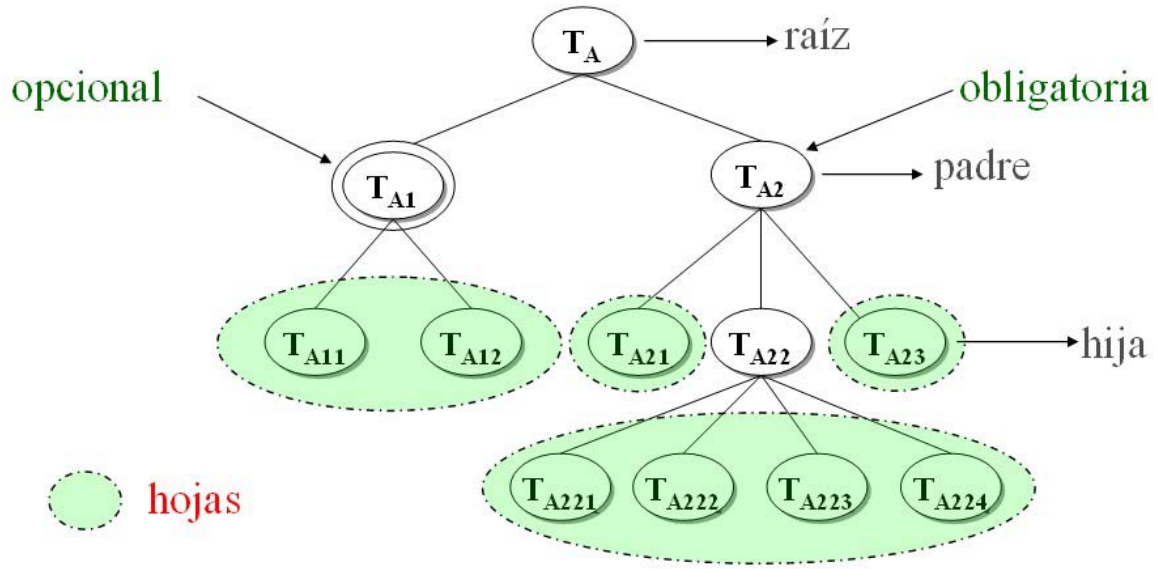



**Figura 2-3** Representación gráfica de las transacciones anidadas.

Este modelo utiliza el protocolo 2PL para el control de concurrencia. Este protocolo se basa en bloqueos, si T es una transacción que requiere un bloqueo en modo A sobre el objeto Q, entonces tendremos las siguientes reglas para adquirir bloqueos:

R1. T puede adquirir un bloque en modo exclusivo sobre el objeto Q si:

1. no existe otra transacción manteniendo un bloqueo en modo compartido o exclusivo sobre Q ó,

2. cada transacción que mantenga un bloqueo compartido o exclusivo sobre Q sea predecesor de T.

R2. T puede adquirir un bloqueo en modo compartido sobre el objeto Q si:

1. no hay otra transacción que mantenga un bloqueo en modo exclusivo sobre el objeto Q ó,

2. cada transacción que mantenga un bloqueo compartido o exclusivo sobre Q, sea el predecesor de T.

Cada vez que una sub-transacción deja de manipular un objeto, lo hereda a su predecesor para que éste sea administrado a ese nivel del árbol.

### 2.2.4 Anidadas abiertas

En el modelo de transacciones anidadas abiertas (TAA) [5] al igual que las transacciones anidadas, una transacción puede tener varias sub-transacciones, las cuales a su vez, pueden tener varias sub-transacciones formando así un árbol de transacciones. Las transacciones anidadas abiertas (*Open Nested Transaction*) están conformadas por una jerarquía de transacciones sobre diferentes niveles de abstracción, y de un conjunto de relaciones de conmutatividad que definen a cada nivel los conflictos entre operaciones predefinidas.

Su principal diferencia con las TA radica en que las sub-transacciones pueden validar (esto significa que sus efectos son visibles a otras transacciones), independientemente de la validación de la transacción raíz. El abandono de la TAA se hace efectivo a través de operaciones inversas de alto nivel que compensan las sub-transacciones ya validadas [22]. Otra diferencia con las transacciones anidadas (cerradas), es que las transacciones anidadas abiertas pueden contener sub-transacciones cerradas, que no liberan resultados parciales, y sub-transacciones abiertas que si liberan sus resultados parciales.

### 2.2.5 Transacciones Móviles

Los métodos tradicionales para el acceso de información están basados en el hecho de que la ubicación de un *host* en un sistema distribuido o centralizado no cambia, y la comunicación entre estos *hosts* tampoco cambia durante una transacción. En un ambiente móvil, sin embargo, estas condiciones no se cumplen. Cuando en una transacción se involucran uno o más dispositivos móviles en su procesamiento, se denomina *Transacción Móvil* (TM) [18, 26].

La computación móvil se distingue del modelo clásico en "*la movilidad*" de los usuarios y sus dispositivos. Las condiciones y limitaciones de los recursos móviles tales como anchos de banda muy bajos y vida limitada de las baterías lo hace un ambiente muy inestable para el procesamiento de transacciones [9, 10, 17]. El constante cambio de un usuario móvil implica que se conectará de diferentes puntos de acceso a través de enlaces inalámbricos y durante el tiempo que permanezca conectado podría experimentar frecuentes interrupciones. Además, los hosts móviles dependen de la vida de sus baterías. Estas condiciones y limitaciones dejan mucho trabajo incompleto durante el proceso de una transacción [10].

El procesamiento de TM es similar al de las Transacciones Distribuidas (TD), la diferencia radica en que el coordinador que inicia y termina la transacción pudiera cambiar de ubicación física, incluso pudiera ser un dispositivo móvil el que inicie la transacción, y podría ser otro dispositivo móvil el que termine la transacción e incluso en una ubicación física diferente (ver figura 2-4). Sin embargo solamente ésta característica es diferente, debido a que los fragmentos de ejecución de la TM se comportan como los fragmentos de las TD, y entonces todo el procesamiento de las TM sigue quedando dentro de la red fija.

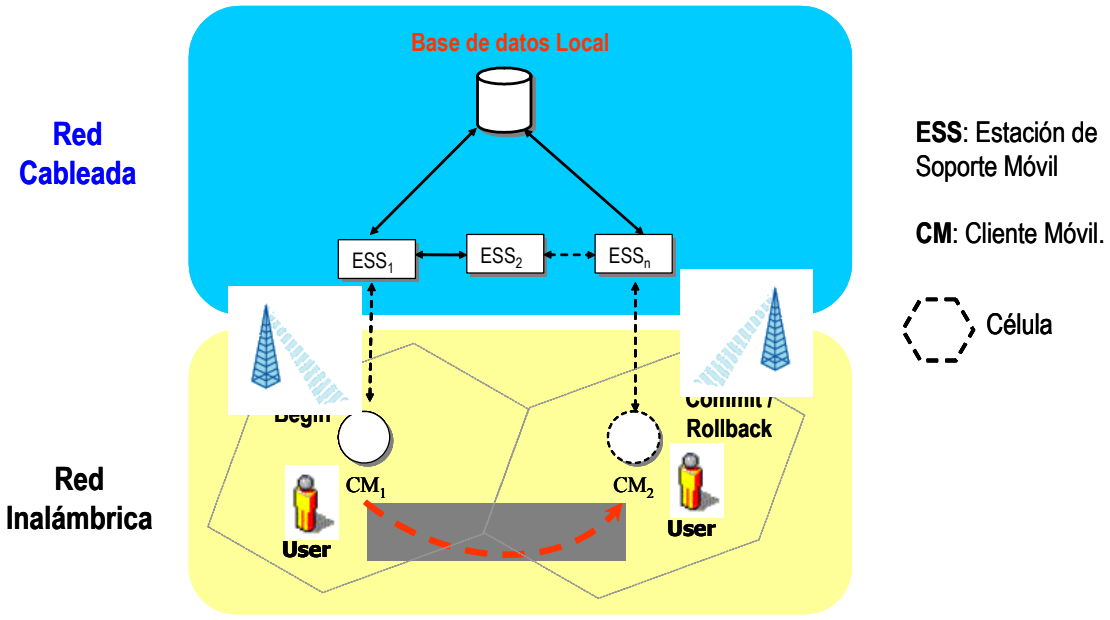



**Figura 2-4** Arquitectura de una Transacción Móvil.

## 2.3 Control de concurrencia

El control de concurrencia es la actividad de sincronizar las operaciones realizadas por la ejecución concurrente de procesos en una base de datos compartida. Es un mecanismo utilizado para prevenir inconsistencias en cualquier DBMS que permita ejecutar transacciones concurrentes [14, 20]. El propósito es generar una ejecución que tenga los mismos efectos que se obtienen en una ejecución serial y garantizar el aislamiento de transacciones concurrentes siguiendo las reglas de las propiedades ACID [4]. Es decir, los usuarios deben percibir que son los únicos que tienen acceso a los datos, aún cuando otras transacciones puedan tener acceso a estos.

Existen dos categorías de mecanismos de control de concurrencia:

- **Optimistas**: retardan la sincronización de las transacciones hasta que las operaciones se hayan ejecutado, los conflictos son menores y sólo se detectan hasta que hayan sucedido.

- **Pesimistas:** Las ejecuciones concurrentes de las transacciones se sincronizan desde el inicio de la transacción, de tal manera que cada transacción debe adquirir bloqueos apropiados para evitar colisiones.

### 2.3.1 Ejecución serial

Cuando un conjunto de transacciones es ejecutada una después de la otra, entonces se obtiene una ejecución serial. En este caso no existe la posibilidad de obtener valores inconsistentes en los objetos utilizados por las transacciones involucradas, debido a que en

ningún momento fueron concurrentes. Es decir, en una ejecución serial las transacciones no compiten por los recursos.

### 2.3.2 Serialización

Cuando una historia de dos o más transacciones concurrentes demuestra obtener resultados equivalentes a los obtenidos en una ejecución serial, se dice que la historia es serializable. Esta característica esta relacionada con la propiedad de "aislamiento" de una transacción, fundamental para el control de concurrencia. Normalmente las transacciones se ejecutan concurrentemente mientras que las ejecuciones seriales son extremadamente ineficientes y poco prácticas.

### 2.3.3 Protocolos

Se han desarrollado diversos trabajos sobre protocolos de control de concurrencia, como son, Semáforos de Dijkstra [7] protocolos basados en bloqueo como el protocolo a dos fases (2PL: *Two Phase Lock*) el cual es implementado por diversos DBMS comerciales [5, 14], los protocolos basados en estampas de tiempo (*Time Stamping Protocol*) [23], y los protocolos basados en multiversión (*Multiversion Protocol*) [4].

- **Semáforos de Dijkstra**

  Un semáforo es una variable apropiada (o un tipo de dato abstracto) que constituye el método clásico para restringir el acceso a recursos compartidos en ambientes de multiprogramación. Fueron desarrollados por Edsger Dijkstra [7] y utilizados por primera vez en el sistema operativo "THE-OS".

  El valor del semáforo es inicializado con cantidad equivalente de recursos compartidos que se van a controlar y particularmente donde exista un recurso que compartir. Los semáforos representan la solución clásica para el problema de los "dining philosophers" aunque no preveen "abrazos mortales"[2].

  Los semáforos siguen siendo utilizados en lenguajes de programación que no soportan intrínsicamente alguna otra forma de sincronización. Y son las primitivas de sincronización en muchos sistemas operativos. Sin embargo la tendencia en los lenguajes de programación apunta hacia mecanismos más estructurados de sincronización, tales como monitores y canales.

  Además del inconveniente de no poder evitar abrazos mortales, los semáforos no protegen al programador del error común de utilizar un semáforo que actualmente se

[2] Abrazo Mortal (*deadlock)*: Situación en la que la transacción Ti se encuentran esperando a otra transacción Tj ($i \neq j$) la cual mantiene bloqueado el objeto que Ti desea bloquear, y viceversa. Estas transacciones no avanzarán a menos que alguna de ellas pare de esperar, en esta situación alguna de las transacciones deberá abortar (rollback).

encuentre utilizado por el mismo proceso, y del problema de libera un semáforo que ha sido utilizado.

- **Protocolo de bloqueo a dos fases**

Este protocolo retarda la ejecución de operaciones en conflicto tales como:

$$w_i(x),\ w_j(x)\ \ \forall i \neq j$$

en un instante dado, mediante el bloqueo de objetos antes de ejecutar cualquier operación de Lectura/Escritura. Existen dos modos en que los objetos pueden ser bloqueados:

- *Shared:* (Compartido) si una transacción *T* obtiene un bloqueo compartido sobre el objeto *Q*, entonces *T* puede leer, pero no escribir sobre *Q*.

- *Exclusive:* (Exclusivo) si una transacción *T* obtiene un bloqueo exclusivo sobre el objeto *Q*, entonces *T* puede tanto leer, como escribir sobre *Q*.

Una transacción sólo puede acceder a un objeto si y sólo si se obtiene un bloqueo apropiado para dicho objeto. Cada transacción debe adquirir un bloqueo en un modo apropiado sobre un objeto Q, dependiendo del tipo de operación a ejecutar sobre Q. Dado un conjunto de modos de bloqueo, podemos definir una *función de compatibilidad* entre ellos de la siguiente forma:

- Sean *A* y *B* representaciones de dos modos de bloqueo arbitrarios. Suponga que una $T_i$ pide un bloqueo en el modo *A* sobre el objeto *Q*.

- $\exists T_j\ (i \neq j)$ que actualmente mantiene un bloqueo en modo *B* al objeto *Q*.

Si $T_i$ puede adquirir el bloque solicitado sobre *Q*, entonces se dice que el ***modo de bloqueo A es compatible con el modo de bloqueo B.*** Tal función de compatibilidad puede ser representada por una matriz M:

$$\text{donde } M(i,j) = verdad,$$

si el modo de bloqueo *i*, es compatible con el modo de bloqueo *j*. La tabla 1 muestra la matriz *M* de compatibilidades entre los diferentes modos de bloqueo.

**Tabla 1** Matriz de compatibilidad entre bloqueos.

| **Bloqueo** | Compartido | Exclusivo |
|---|---|---|
| Compartido | Si | No |
| Exclusivo | No | No |

El protocolo 2PL obliga a que cada transacción obtenga bloqueos y desbloqueos de objetos en dos fases:

- *Fase de crecimiento.* Una transacción, durante esta fase, puede únicamente obtener nuevos bloqueos, pero no puede liberarlos.

- *Fase de decrecimiento*. Una transacción, durante esta fase, sólo puede liberar los bloqueos, pero no puede obtener uno nuevo.

Si la transacción cumple con estas etapas, se dice que la transacción es a dos fases. El protocolo 2PL garantiza la serialiazación de las transacciones. Sin embargo, es muy posible que se encuentren situaciones de *abrazo mortal.*

Existen variantes de este protocolo tales como: *Strict two-phase locking*, *Non-strict two-phase locking*, *Conservative two-phase locking*, *Index locking* y *Multiple granularity locking*.

- **Protocolo de ordenamiento por estampas de tiempo**

Otra forma de establecer un orden de serialización de transacciones es mediante la selección de un orden progresivo, entre las transacciones. El enfoque más común para implementar este enfoque es utilizar un *ordenamiento por estampas de tiempo.*

A cada transacción $T_i$ en el sistema, se le asigna una única *estampa de tiempo,* denotada por TS($T_i$). Esta etiqueta es asignada por el sistema de base de datos antes de que la transacción $T_i$ inicie su ejecución. Si se asigna un TS a la transacción $T_i$ y una nueva transacción $T_j$ se inicia en el sistema, entonces TS($T_i$) < TS($T_j$). Se utilizan dos métodos para implementar este esquema:

1. Utilizando el reloj del sistema como estampas de tiempo. La TS de la transacción es el valor que el reloj tiene cuando la transacción entra al sistema.

2. Utilizando un contador que se incrementa después de asignar cada TS. La TS de la transacción es el valor que el contador tiene cuando la transacción entra al sistema.

Las estampas de tiempo de las transacciones establecen un orden de serialización de transacciones. De esta forma, si TS($T_i$) < TS($T_j$), el sistema debe asegurar que el cronograma generado sea equivalente a un ejecución serial en donde $T_i$ aparezca antes que $T_j$ . Para implementar este esquema, se asocian dos estampas de tiempo por cada objeto $Q$.

1. *W-TS(Q).*Representa la TS mayor entre las estampas de tiempo de todas las transacciones que han tenido éxito al ejecutar una escritura sobre el objeto $Q$.

2. *R-TS(Q)*. Representa la TS mayor entre las estampas de tiempo de todas las transacciones que han tenido éxito al ejecutar una lectura sobre el objeto *Q*.

Estos valores se actualizan cuando se generan nuevas lecturas o escrituras sobre el objeto *Q*. El protocolo de ordenamiento por estampas de tiempo garantiza que todas las operaciones en conflicto de lectura/escritura sean ejecutadas en el orden generado por las estampas de tiempo. Este protocolo se organiza de la siguiente forma:

1. Suponga que una transacción $T_i$ necesita ejecutar una lectura sobre *Q*,

    a) Si TS($T_i$) < W-TS(Q), esto significa que $T_i$ necesita leer un valor de Q que ya ha sido re-escrito. Entonces la operación de lectura debe ser rechazada y la transacción $T_i$ debe ser abortada.

    b) Si TS($T_i$) ≥ W-TS(Q), significa que la operación de lectura sobre el objeto Q, puede llevarse a cabo, y entonces a R-TS(Q) se le asigna el valor máximo entre R-TS(Q) y TS($T_i$).

2. Suponga que la transacción $T_i$ necesita ejecutar una escritura sobre *Q*,

    a) Si TS($T_i$) < R-TS(Q), esto significa que el valor del objeto *Q,* al cual $T_i$ se encuentra modificando, ha sido previamente leído por otra transacción entonces no es posible realizar cualquier actualización.

    b) Si TS($T_i$) < W-TS(Q), esto significa que $T_i$, intenta escribir un valor obsoleto de *Q*. La operación de escritura por lo tanto simplemente es ignorada.

    c) Si ninguno de los casos anteriores se cumple, la operación de escritura es ejecutada, y a W-TS(Q) se le asigna el valor máximo entre W-TS(Q) y TS($T_i$).

A cada transacción que es abortada debido al control de concurrencia, se le asigna una nueva estampa de tiempo y es reiniciada. El protocolo de ordenamiento por estampas de tiempo garantiza la serialización, debido a que las operaciones en conflicto se procesan bajo un orden por estampas de tiempo. También asegura la ausencia de *deadlocks*, debido a que las transacciones nunca tendrán que esperar. Sin embargo, una desventaja de este protocolo es que se pueden tener transacciones abortadas en cascada.

- **Protocolo Multiversión (MVCC)**

El protocolo MVCC provee a cada usuario conectado a la base de datos con una "copia" de los datos para que pueda operar sobre ellos. Cualquier cambio realizado por otras transacciones no estará disponible a otros usuarios hasta que la transacción haya confirmado con éxito.

MVCC utiliza estampas de tiempo o identificadores crecientes para cada transacción para conseguir la serialización. MVCC garantiza que una transacción nunca tendrá que esperar por un objeto de la base de datos mientras se mantienes diversas versiones de un objeto. Cada versión tendrá una estampa de tiempo para una escritura y permitiría a una $T_i$ leer la versión mas reciente de un objeto que preceda a la estampa de tiempo TS($T_i$). Si $T_i$ intenta escribir a un objeto, y hubiera otra transacción $T_j$, el orden de las estampas de tiempo debería ser TS($T_i$) < TS($T_j$) para que dicha escritura tenga éxito.

También se asignan estampas de tiempo para lecturas a cada objeto, y si $T_i$ intenta escribir a un objeto $p$ y las estampas de tiempo son TS($T_i$) < RTS(P), entonces $T_i$ es cancelada y reiniciada. De otra manera, $T_i$ creará una nueva versión de $p$ y establecerá las estampas de tiempo de lectura/escritura de $p$ a TS($T_i$).

El inconveniente de este mecanismo es el costo de almacenar múltiples versiones de los objetos en la base de datos. Pero por el otro lado, se consigue que las "lecturas" no bloqueen a las "escrituras" y viceversa. Lo cual es importante para operaciones que incluyen muchas lecturas a la base de datos.

Los DBMSs que basan su mecanismo de control de concurrencia en MVCC son: *Berkeley DB* [21], *Firebird (database server)* [25], *Borland InterBase*, *Microsoft SQL Server (*sólo *SQL Server 2005)*, *MySQL* utilizando el *storage-engine InnoDB, ObjectStore* (sólo en el modo *read-only*), *Oracle*, *PostgreSQL* y *ThinkSQL*.

## 2.4 Teoría del aislamiento

Una de las propiedades clave que una transacción debe preservar es la de Aislamiento que garantiza la consistencia de la base de datos. En una base de datos compartida por diversas aplicaciones, el DBMS debe garantizar el Aislamiento aún cuando existan transacciones ejecutándose concurrentemente. Sin embargo, evitar que una transacción pueda acceder a los datos que en ese momento esté modificando otra transacción, puede disminuir el rendimiento de las aplicaciones, debido a que las transacciones deben esperar a que otras liberen los recursos que se encuentren modificando. Por el contrario si se permiten a las transacciones acceder a los datos modificados por otras transacciones, se pueden generar inconsistencias en la base de datos.

### 2.4.1 Aislamiento

Para observar los estados que pueden ocurrir cuando se trata de preservar la propiedad de aislamiento, podemos ver a una transacción como un conjunto de operaciones de lectura/escritura sobre los objetos del sistema. Una lectura no cambia el estado del objeto, dos lecturas al mismo objeto en diferentes transacciones no violan la consistencia, una escritura si cambia el estado del objeto, dos escrituras al mismo objeto en la misma transacción no violan la consistencia. Sólo dos escrituras de transacciones concurrentes pueden crear inconsistencia o violar el aislamiento.

### 2.4.2 Dependencias

Sea $L_i$ es el conjunto de objetos leídos por la transacción $T_i$ (sus entradas) y $O_i$ el conjunto de objetos escritos por $T_i$ (sus salidas). Entonces, dos transacciones pueden ejecutarse en paralelo sin violar el aislamiento si sus salidas son disjuntas de las entradas y salidas de la otra:

$$O_i \cap ( L_j \cup O_j ) = \phi \text{ para todo } i \neq j$$

Los objetos pasan a través de diferentes versiones[3] a medida que son manipulados por estas acciones. Las lecturas no cambian la versión del objeto y las escrituras generan nuevas versiones del objeto afectado.

Podemos definir un grafo de dependencias (GD) como: una secuencia de tiempo, donde, si existe una arista entre $T_1$ y $T_2$ entonces:

- $T_1$ accesa un objeto que posteriormente es accesado por $T_2$;
- al menos uno de estos accesos crea una nueva versión del objeto.

Un GD sin ciclos implica una ejecución aislada de las transacciones. En las siguientes secuencias de ejecución, los GD que pueden generarse entre dos transacciones ejecutándose concurrentemente se muestran en la figura 2-5. Estas dependencias son conocidas como dependencia *Read-Write,* dependencia *Write-Read,* dependencia *Write-Write,* note que el caso de la dependencia *Read-Read*, no existe debido a que ésta no compromete la consistencia de la base de datos ni viola el aislamiento de las transacciones.

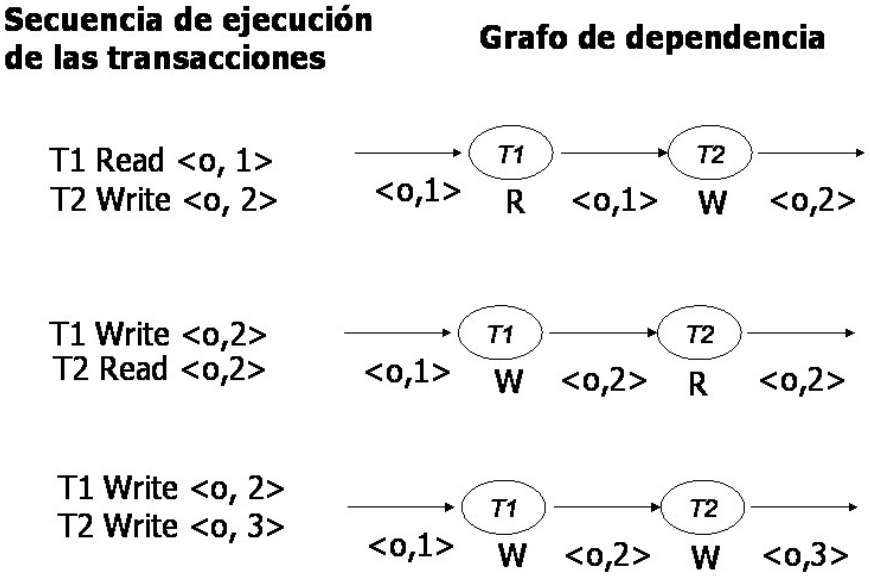



**Figura 2-5** Grafos de dependencias.

[3] *Versión:* indica el estado actual del objeto que se está manipulando.

## 2.5 Malas dependencias

Limitar el acceso a los datos utilizando bloqueos con el objetivo de preservar la consistencia de los datos, puede disminuir el rendimiento (*throughput*) de las aplicaciones. Por otro lado si se permite el acceso concurrente sin control a los datos, sin un mecanismo que garantice la serialización de las operaciones de lectura/escritura se pueden presentar cuatro situaciones conocidas como "malas dependencias" [3, 5, 14], que pueden comprometer la consistencia de los datos como son:

- lecturas sucias (*dirty reads*)
- lecturas no-repetibles (*unrepeatable reads*)
- fantasmas (*phantom*), y
- actualizaciones perdidas (*lost update*)

### 2.5.1 Lecturas sucias (Dirty Reads)

Esta mala dependencia sucede cuando dos transacciones T1 y T2, llevan a cabo las siguientes acciones: <T2,W,O>[4], <T1,R,O>, <T2,W,O>. Donde R es una lectura y W es una escritura sobre el objeto O. T2 ejecuta una escritura sobre el objeto O generando una nueva versión del objeto, posteriormente, T1 lee el objeto O con la versión que produjo la escritura previa de T2, entonces T2 vuelve a escribir sobre el objeto O generando una nueva versión, entonces el valor que leyó T1 es conocida como *Lectura Sucia.* La figura 2-6 muestra el grafo de dependencias para la *Lectura Sucia.*

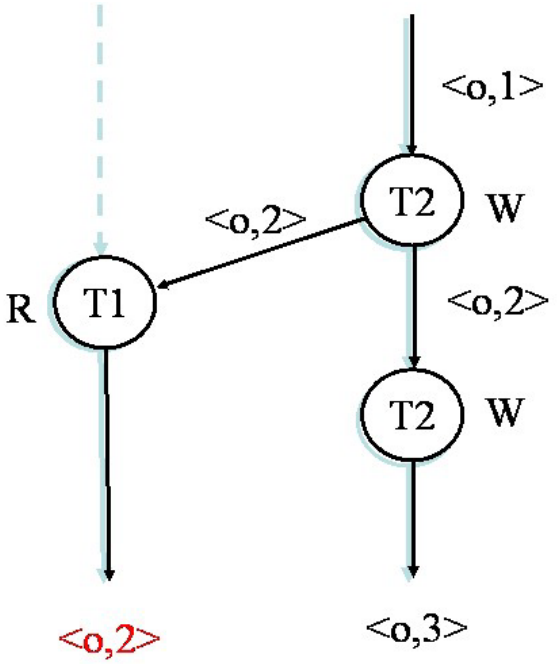



**Figura 2-6** Grafo de dependencia de la lectura sucia.

[4] La triupla < t, a, o > define a la acción (a) que se lleva a cabo sobre el objeto (o) en la transacción (t).

### 2.5.2 Lectura no-repetible (Unrepeatable Read)

Esta sucede cuando dos transacciones T1 y T2, llevan a cabo las siguientes acciones: <T1,R,O>, <T2,W,O>, <T1,R,O>. La versión del objeto que lee T1 no es la misma en ambas lecturas debido a que T2 ejecuta una escritura al objeto O cambiando la versión del objeto. La figura 2-7muestra el grafo de dependencia de la lectura no-repetible.

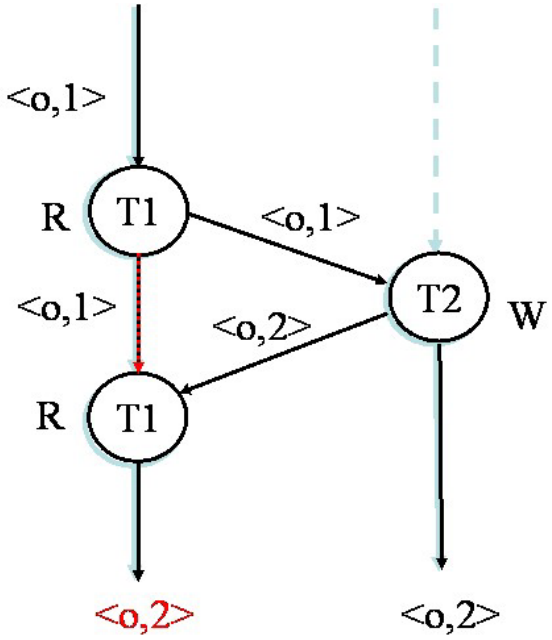



**Figura 2-7** Grafo de dependencia de la lectura no-repetible.

T1 lee un objeto dos veces, una vez antes que T2 lo modifique y la otra después que T2 lo ha modificado. Las dos lecturas devuelven valores diferentes del mismo objeto leído. Note que la lectura sucia, puede convertirse además en una lectura no-repetible que se describe a continuación, en caso de que T1 vuelva a releer el objeto O después de que T2 haya cambiado la versión del objeto O.

### 2.5.3 Fantasmas (Phantom)

Esta sucede cuando T1 lee un conjunto de elementos que satisfacen un criterio de búsqueda <*search condition*>. Posteriormente T2 crea un nuevo elemento que satisface el criterio de búsqueda de T1 ya sea por una inserción, actualización o incluso una eliminación. Si T1 repite la lectura previa con el mismo criterio de búsqueda, entonces obtendrá un conjunto de elementos diferentes a los obtenidos en la primera lectura. El grafo de dependencia de la mala dependencia *phantom* (ver figura 2-8) es similar al de la lectura no-repetible (ver figura 2-7).

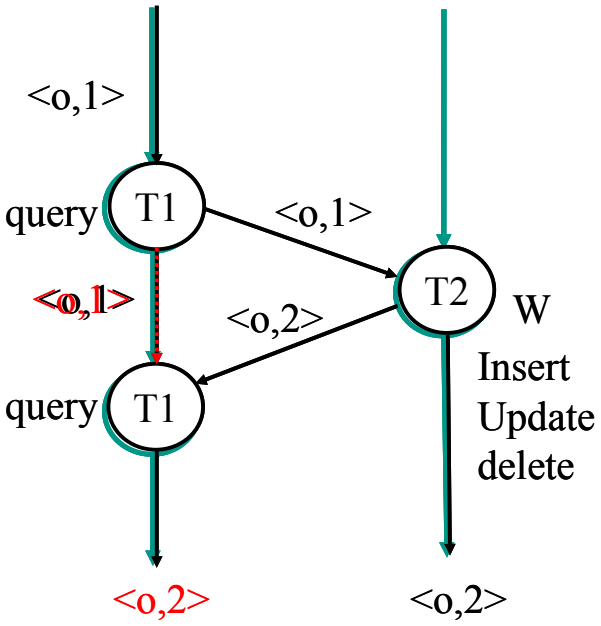



**Figura 2-8** Grafo de dependencias de *Phantom*.

### 2.5.4 Actualización perdida (Lost Update)

Sucede cuando dos transacciones T1 y T2, llevan a cabo las siguientes acciones: <T1,R,O>, <T1,W,O>, <T2,W,O>. La figura 2-9 muestra el grafo de dependencias para la actualización perdida.

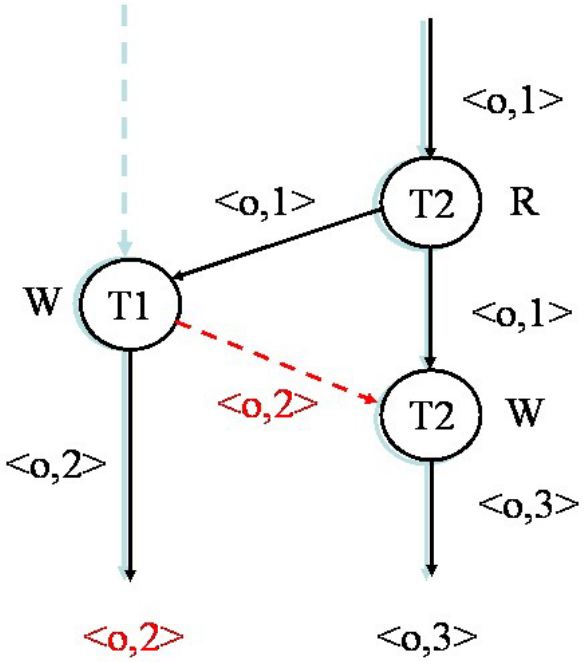



**Figura 2-9** Grafo de dependencia de la actualización perdida.

La escritura de la transacción T1 es ignorada por la transacción T2 y reemplazada, la cual escribe el objeto basado en el valor original <o,1> generando la versión <o,3>. Ambas transacciones se ejecutan con versiones diferentes del mismo objeto. Para evitar estas inconsistencias y mejorar el rendimiento de las aplicaciones se introdujo el concepto de *Niveles de aislamiento.*

## 2.6 Niveles de aislamiento

Para disminuir los efectos que conlleva el bloqueo de los recursos, en [1] se han definido cuatro niveles de aislamiento:

- *read-uncommited,*
- *read-commited* (por defecto en la mayoría de los DBMS),
- *repeatable-reads y*
- *serializable.*

### 2.6.1 Lecturas no-confirmadas (*Read Uncommitted*)

Con este nivel pueden ocurrir las tres malas dependencias. Permite que un objeto que ha cambiado otra transacción, pueda ser leído por otra transacción, antes de que cualquier cambio sea confirmado. Si los cambios son abortados, la segunda transacción obtendrá valores inválidos.

### 2.6.2 Lecturas confirmadas (*Read Committed*)

Con este nivel las lecturas sucias pueden ser prevenidas, pero las Lecturas no-repetibles y los fantasmas pueden ocurrir. Este nivel prohíbe a una transacción leer datos que no han sido validados.

### 2.6.3 Lecturas Repetibles (*Repeatable Read*)

En este nivel las lecturas sucias y lecturas no-repetibles pueden ser prevenidas, pero los fantasmas pueden ocurrir. Este nivel prohíbe a una transacción leer datos que no han sido validados. También prohíbe la situación cuando una transacción lee un objeto, y una segunda transacción altera el objeto, y la primera transacción relee el objeto obteniendo un valor diferente la segunda vez.

### 2.6.4 Serialización (*Serializable*)

Con este nivel se evitan las tres malas dependencias, es decir, no permite que otras transacciones interfieran con valores que no hayan sido confirmados por otras transacciones. Este nivel puede afectar al rendimiento del sistema debido a que ninguna transacción puede acceder a los objetos que opera otra transacción hasta que ésta los libere. La ventaja del nivel de serialización es que garantiza la propiedad del aislamiento de las transacciones.

**Tabla 2** Niveles de aislamiento y su relación con las malas dependencias.

| Nivel de aislamiento | Lost Update | Dirty Reads | Unrepeatable Read | Phantom |
|---|---|---|---|---|
| Read Uncommitted | No | Si | Si | Si |
| Read Committed | No | No | Si | Si |
| Repeatable Read | No | No | No | Si |
| Serializable | No | No | No | No |

La Tabla 2 muestra la relación que existe entre los diferentes niveles de aislamiento y las posibles malas dependencias que puedan suceder. Note que el nivel de aislamiento *read uncommited* es el menos restrictivo, aumenta el rendimiento de las aplicaciones pero pueden suceder las tres malas dependencias. Por otro lado, el nivel de aislamiento *serializable* es el más restrictivo, debido a que se evitan las malas dependencias, pero se pueden crear escenarios de abrazos mortales (*deadlokcs*). La selección del nivel de independencia adecuado dependerá de las operaciones a realizar en cada transacción y en general del problema a resolver en sí. En la figura 2-10 se ilustra los efectos que tiene cambiar el nivel de aislamiento sobre la concurrencia de las transacciones y la consistencia de los datos. A mayor nivel de aislamiento (*serializable*) se disminuye la concurrencia de las transacciones pero se garantiza la consistencia de los datos. A un nivel de aislamiento menos restrictivo (*read uncommitted*) se consigue aumentar la concurrencia, evitar abrazos mortales pero no se garantiza la consistencia de los datos.


**Figura 2-10** Nivel de aislamiento y sus efectos en la consistencia y concurrencia.

### 2.6.5 Otros niveles de aislamiento "Cursor Stability"

Este nivel fue creado para evitar la mala dependencia "actualización perdida". Este nivel de aislamiento extiende el comportamiento de bloqueo del nivel READ COMMITTED para los cursores en SQL, requiriendo que se mantenga un bloqueo para el elemento activo del cursor. El bloqueo se mantendrá hasta que exista un movimiento en el cursor o este sea

cerrado, posiblemente por *commit*. Naturalmente que la transacción puede actualizar el cursor, en este caso un bloque para escritura se mantendrá aún cuando el cursor se mueva por FETCHS posteriores, hasta que la transacción termine.

## 2.7 Compatibilidad en DBMS comerciales

Se llevaron a cabo pruebas con los siguientes manejadores de bases de datos comerciales:

- Microsoft Sql Server ver. 2000
- Informix ver. 9.20
- IBM DB2 Universal Database ver. 7.1
- MySql para Linux ver. 5.0.16
- PostgreSQL para Linux ver. 8.0.4
- Oracle ver. 8i

### 2.7.1 Microsoft Sql Server ver. 2000

Este manejador soporta todos los niveles de aislamiento definidos en [1]. Se lanzaron transacciones concurrentes que efectuaran al azar operaciones de lectura/escritura con el nivel de aislamiento READ-COMMITTED, que es el nivel por defecto del manejador. Bajo este nivel es posible eliminar las lecturas sucias, pero pueden ocurrir las lecturas-no-repetibles y actualizaciones-perdidas. Sin embargo no es posible reproducir un escenario de actualizaciones-perdidas a pesar de que la teoría del aislamiento (ver sección 2.4) especifica que es posible bajos los niveles de aislamiento READ UNCOMMITTED, READ COMMITTED y REPEATABLE READS.

### 2.7.2 Informix ver. 9.20

Informix database server utiliza bloqueos compartidos para soportar diferentes niveles de aislamientos entre transacciones que intentan leer un mismo dato (objeto). Los procesos de actualización y eliminación siempre adquieren un bloqueo exclusivo sobre el renglón afectado. El nivel de aislamiento no interfiere con tales renglones, pero si el modo de acceso ya sea para actualización o eliminación. Si una transacción t1 intenta actualizar o eliminar un reglón que actualmente haya sido leído por otra transacción t2 con el nivel de aislamiento SERIALIZABLE o (ANSI) REPEATABLE READ, entonces se le denegará el acceso a t1.

### 2.7.3 IBM DB2 Universal Database ver. 7.1

El gestor de bases de datos de DB2 Universal Database da soporte a cuatro niveles de aislamiento. Independientemente del nivel de aislamiento, el gestor de bases de datos coloca bloqueos de exclusividad en cada fila que se inserta, actualiza o elimina. Por lo tanto, los niveles de aislamiento aseguran que las filas que cambia una transacción t1 durante una unidad de trabajo no las pueda modificar ninguna otra transacción t2 hasta que t1 haya finalizado. Los niveles de aislamiento soportados son:

- Lectura repetible (Repeatable Read)
- Estabilidad de lectura (Read Stability)
- Estabilidad del cursor (Cursor Stability)
- Lectura no confirmada (Uncommitted Read)

### 2.7.4 MySql para Linux ver. 5

El gestor de bases de datos de MySql permite demarcar una transacción entre las sentencias BEGIN y COMMIT, sólo para aquellas tablas que hayan sido creadas con el tipo InnoDB, por ejemplo:

```
mysql> CREATE TABLE t (f INT) TYPE=InnoDB;
```

El nivel de aislamiento por defecto para InnoDB es REPEATABLE READ. MySQL/InnoDB soporta los cuatro niveles de aislamiento definidos en [1]. Cuando utiliza el nivel de aislamiento por defecto, MySql utiliza el concepto de ***Consistent Non-Locking Read***, el cual utiliza un mecanismo de multi-versión para presentar una copia de los datos a una consulta en un momento dado, es decir, la consulta obtiene los cambios realizados por transacciones que han terminado antes de la consulta y no puede observar los cambios realizados por transacciones sin confirmar. Para obtener los cambios realizados por otras transacciones sin confirmar se debe utilizar el nivel de aislamiento READ UNCOMMITTED.

### 2.7.5 PostgreSQL para Linux ver. 8.0.4

A diferencia de los manejadores de bases de datos tradicionales que utilizan un control de concurrencia basado en bloqueos, PostgreSQL mantiene la consistencia de los datos utilizando un modelo de multiversión (Multiversion Concurrency Control, MVCC). Esto implica que, mientras una transacción realiza consultas a una base de datos, observa copias de los datos (una versión de la base de datos) anteriores, en lugar de observar el estado actual de los datos modificados por otras transacciones sin confirmar. Protegiendo a las transacciones de observar datos inconsistentes ocasionadas por actualizaciones de los mismos datos en otras transacciones concurrentes.

En PostgreSQL, puede solicitar cualquiera de los cuatro niveles de aislamiento definidos en [1]. Sin embargo, internamente sólo existen dos niveles de aislamiento, los cuales corresponden a Read Committed y Serializable. Cuando selecciona el nivel Read Uncommitted realmente se establece el nivel Read Committed. Y cuando selecciona el nivel Repeatable Read realmente se obtiene el nivel Serializable. La razón por la que PostgreSQL sólo provee dos niveles de aislamiento, es debido a que es la única forma de conseguir que los niveles de aislamiento concuerden con la arquitectura del control de concurrencia multi-versión.

### 2.7.6 Oracle ver. 8i

Oracle provee dos niveles de aislamiento: read committed y serializable así como modos de solo-lectura el cual no es parte de la norma ANSI-SQL92. Suministrando a los desarrolladores de aplicaciones con modos de operación los cuales preservan la consistencia y proveen de un mayor rendimiento. Read committed es el nivel de aislamiento por defecto. Oracle mantiene la consistencia de los datos en un ambiente multiusuarios a través del modelo de control de concurrencia multi-versión y una variedad de tipos de bloqueo y manejo de transacciones.

En la tabla 3 se ilustra la relación de los niveles de aislamientos definidos en ANSI-SQL 92 [1] y los niveles definidos por cada manejador de bases de datos comercial.

**Tabla 3** Compatibilidad con DBMS comerciales.

| **ANSI Isolation Level** | **Read Uncommitted** | **Read Committed** | **Repeatable Read** | **Serializable** |
|---|---|---|---|---|
| Microsoft Sql Server | Read Uncommitted | Read Committed | Repeatable Read | Serializable |
| MySQL | Read Uncommitted | Read Committed | Repeatable Read | Serializable |
| IBM DB2 | Uncommitted read | Cursor Stability | Read Stability | Repeatable read |
| Informix | Dirty Read | Committed Read | Cursor Stability | Repeatable Read |
| Sybase | Level 0 | Level 1 | Level 2 | Level 3 |
| **Manejadores que usan Multiversión** | | | | |
| PostgreSql | --- | Read Committed | --- | Serializable |
| Oracle | --- | Read Committed | --- | Serializable |

# Capítulo 3 Control de concurrencia extendido

En este capítulo se dan a conocer los detalles del análisis y el diseño utilizados para la implementación del modelo para el Control de Concurrencia Extendido en Transacciones Anidadas (CCxTA).

## 3.1 Análisis del modelo

La API JDBC (*Java Database Connectivity*) de Java 2 es la base de implementación del modelo CCxTA. Esta API provee a las aplicaciones de Java acceso de "medio nivel" (indirecto) a cualquier sistema de bases de datos, a través del Lenguaje Estructurado de Consultas SQL (*Structured Query Language*). La figura 3-1 ilustra la arquitectura de JDBC.

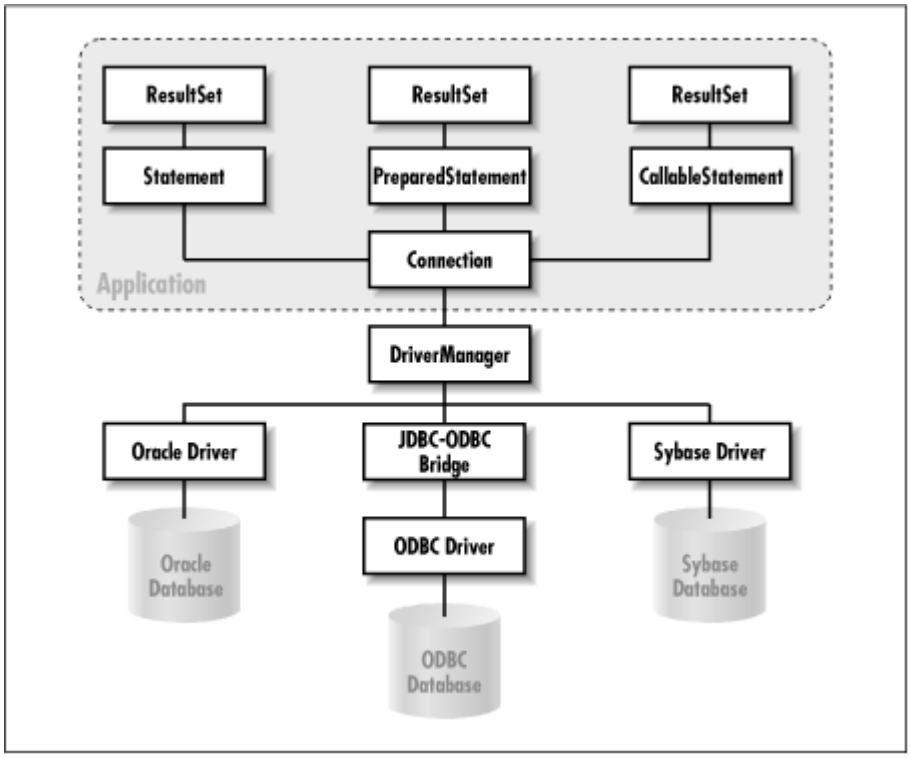



**Figura 3-1** Arquitectura de JDBC.

El paquete java.sql que está incluido desde la versión JDK 1.2 (conocido como el API JDBC 2.0) el cual incluye características tales como:

- Navegar un *ResultSet* hacia adelante o hacia atrás.

- Llevar a cabo actualizaciones de las tablas utilizando métodos Java en lugar de utilizar comandos SQL.
- Enviar múltiples secuencias SQL a la base de datos como una unidad, o batch, así como la ejecución de Transacciones.
- Utilizar los nuevos tipos de datos SQL3 como valores de columnas.

## 3.2 Diseño en UML

### 3.2.1 Diagrama de Componentes

Para que una aplicación lleve a cabo el procesamiento de Transacciones Anidadas (TA) requiere del uso de dos componentes: un *Monitor de Transacciones Anidadas (MTA)* y del *API para Transacciones Anidadas (API-TA)*. La figura 3-2 ilustra la relación entre estos componentes, y el escenario para la implementación de transacciones anidadas.

El MTA funciona como una capa intermedia entre las aplicaciones y el puente JDBC-ODBC, lleva a cabo el monitoreo de cada transacción anidada lanzada por las aplicaciones, registra los objetos bloqueados por cada transacción y administra los bloqueos bajo un conjunto de políticas que permiten obtener el comportamiento del modelo de Transacciones Anidadas ya sea Cerradas (TAC)o Abiertas (TAA).

La API-TA proporciona a las aplicaciones la funcionalidad para el procesamiento de transacciones bajo los modelos de Transacciones Anidas Cerradas o Abiertas. Permite la creación y terminación de transacciones anidadas, así como el intercambio de mensajes entre el MTA para llevar a cabo la finalización por la transacción padre como en el modelo TAC, o la finalización independiente del modelo TAA.

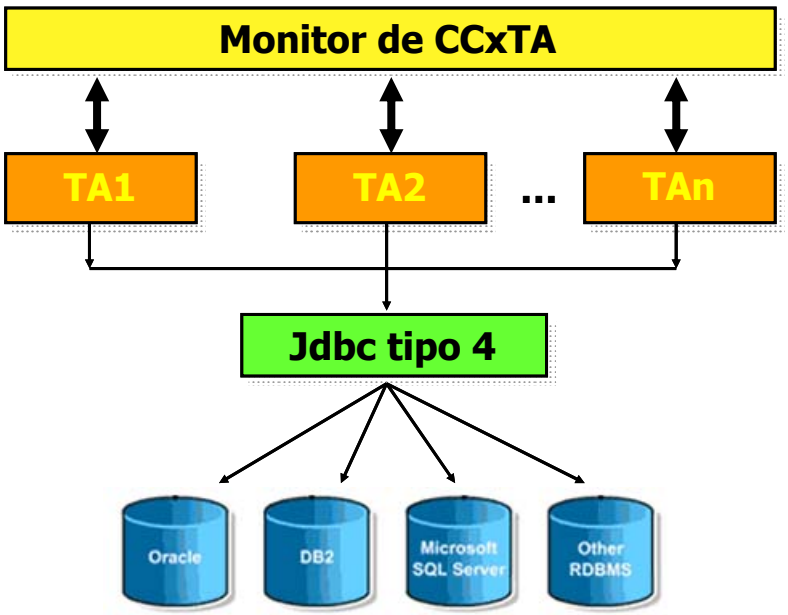



**Figura 3-2** Arquitectura de un escenario de Transacciones Anidadas Cerradas.

Las clases desarrolladas en la API-TA para el modelo CCxTA, tales como *ConnectionDB.class* y *Transaction.class* que proporcionan la conexión a cualquier recurso de bases de datos y los métodos para implementar transacciones anidadas cerradas tales como: iniciar una transacción *(begin),* finalizarla con éxito *(commit)* y cancelarla *(rollback)*. Estas clases lanzan dos tipos de excepciones cuando se presentan fallas durante la ejecución de una transacción. Por lo tanto cada aplicación debe capturar las siguientes excepciones *TransacctionException.class* para fallas emitidas por la fuente de datos, es decir, fallas emitidas por el manejador de base de datos, tales como: fallos en la conexión, fuentes de datos no disponibles, errores de sintaxis en la sentencias SQL, entre otros. Y la excepción *PartialRollbackException.class* que es lanzada cuando se presentan problemas de control de concurrencia, tales como "malas dependencias".

El diagrama de componentes de la figura 3-3 ilustra las clases que una aplicación debe utilizar para utilizar TACs. Por ejemplo, el componente denominado TAC1.class (Transacción Anidada Cerrada 1) debe crear objetos de la clase *Transaction.class* la cual hereda la funcionalidad de conectividad a base de datos proporcionada por la clase *ConnectionDB.class*. Así cada TACn es monitoreado por la clase *Consola.class* creando un escenario de TACs con la supervisión de "malas dependencias" y mejorar la tolerancia a fallas.

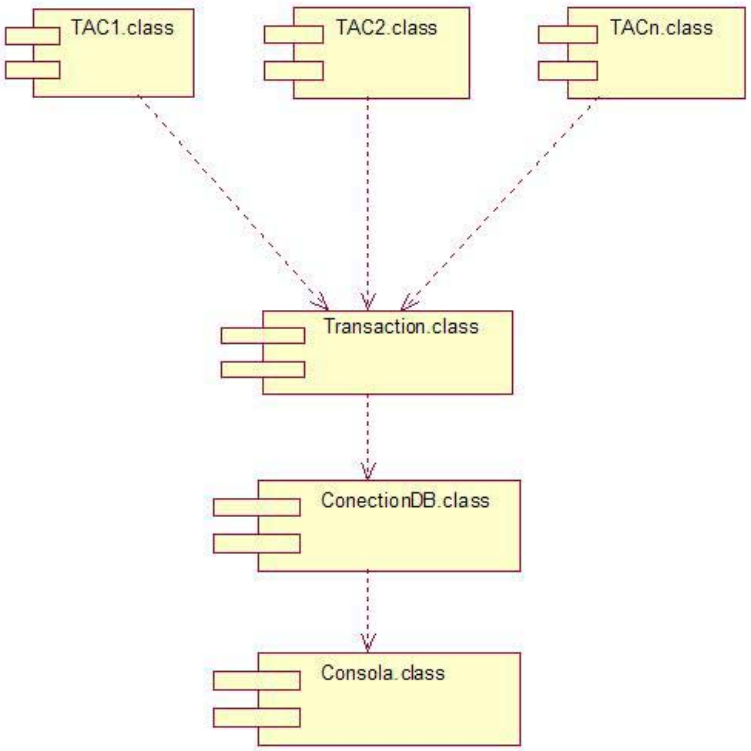



**Figura 3-3** Diagrama de componentes para Transacciones Anidadas Cerradas.

### 3.2.2 Casos de uso

La figura 3-4 ilustra los casos de uso y los actores que intervienen en el proceso de una Transacción Anidad Cerrada. El actor "Usuario" representa a un desarrollador de aplicaciones el cual implementa una Transacción Anidad Cerrada bajo el modelo CCxTA. El proceso Transacción está basado en la clase *Transaction.class* y contiene los métodos necesarios para llevar a cabo la inicialización de una transacción, lanzar instrucciones SQL y emitir las operaciones de terminación tales como *commit/rollback.* El MTA, es capaz de monitorear y registrar todas las operaciones llevadas a cabo por la Transacción y de esta manera llevar un control de objetos bloqueados por las transacciones concurrentes, con el objetivo de identificar inconsistencias y emitir mecanismos de recuperación evitando tales inconsistencias.

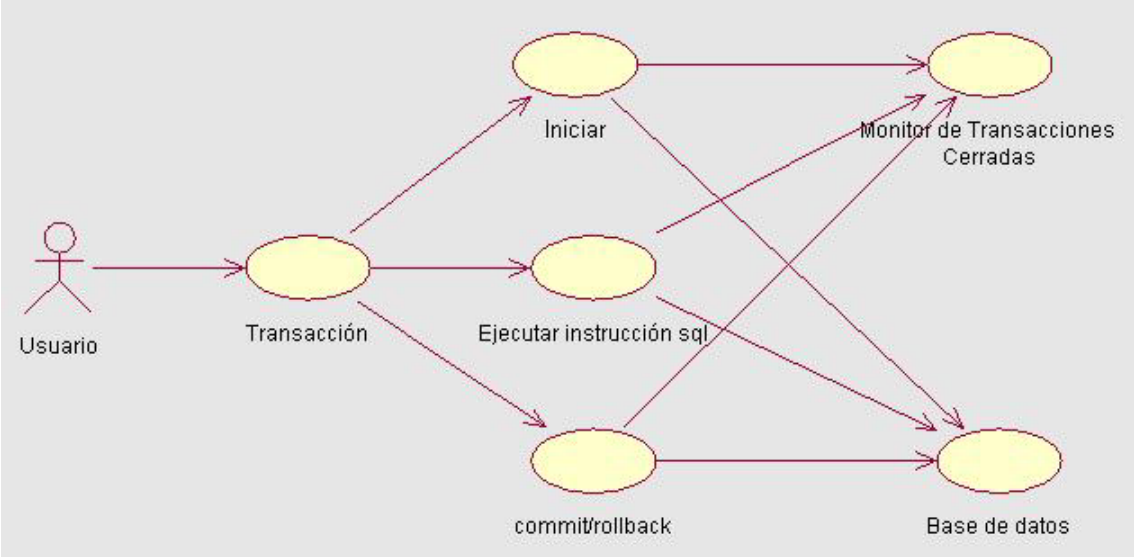



**Figura 3-4** Diagrama de casos de uso de una Transacción Anidada Cerrada.

### 3.2.3 Diagrama de clases

A continuación se describen las clases que se desarrollaron para implementar el modelo CCxTA, el cual consta de las siguientes clases (ver figura 3-5):

- ConnectionDB
- Transaction
- TransactionException
- PartialRollbackException

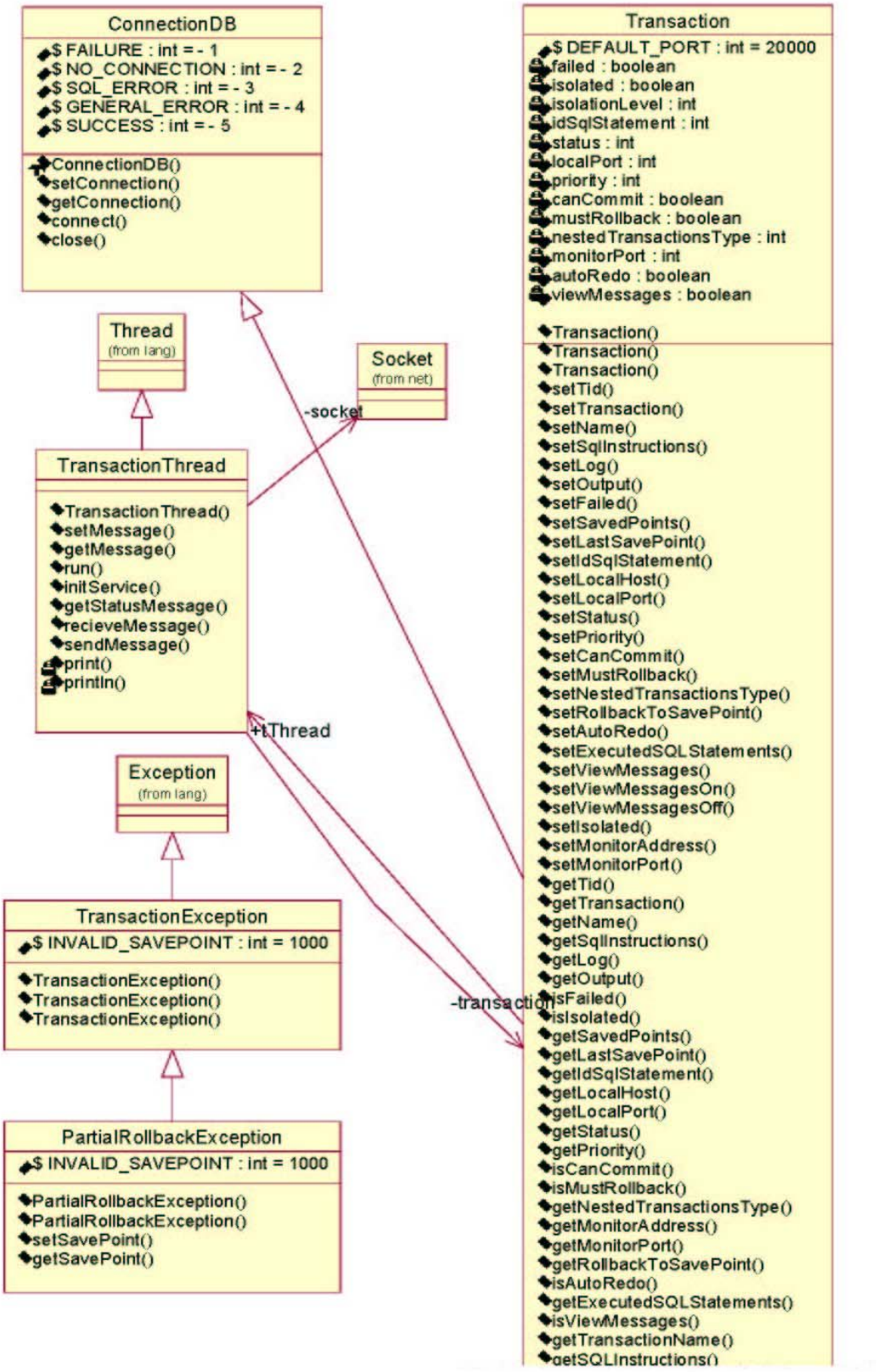



**Figura 3-5** Diagrama de clases.

- **Clase ConnectionDB**

En la figura 3-6 se ilustra la clase **ConnectionDB.** Esta clase proporciona los mecanismos de conexión a un manejador de bases de datos relacional a través de ODBC (*Open DataBase Connectivity).*

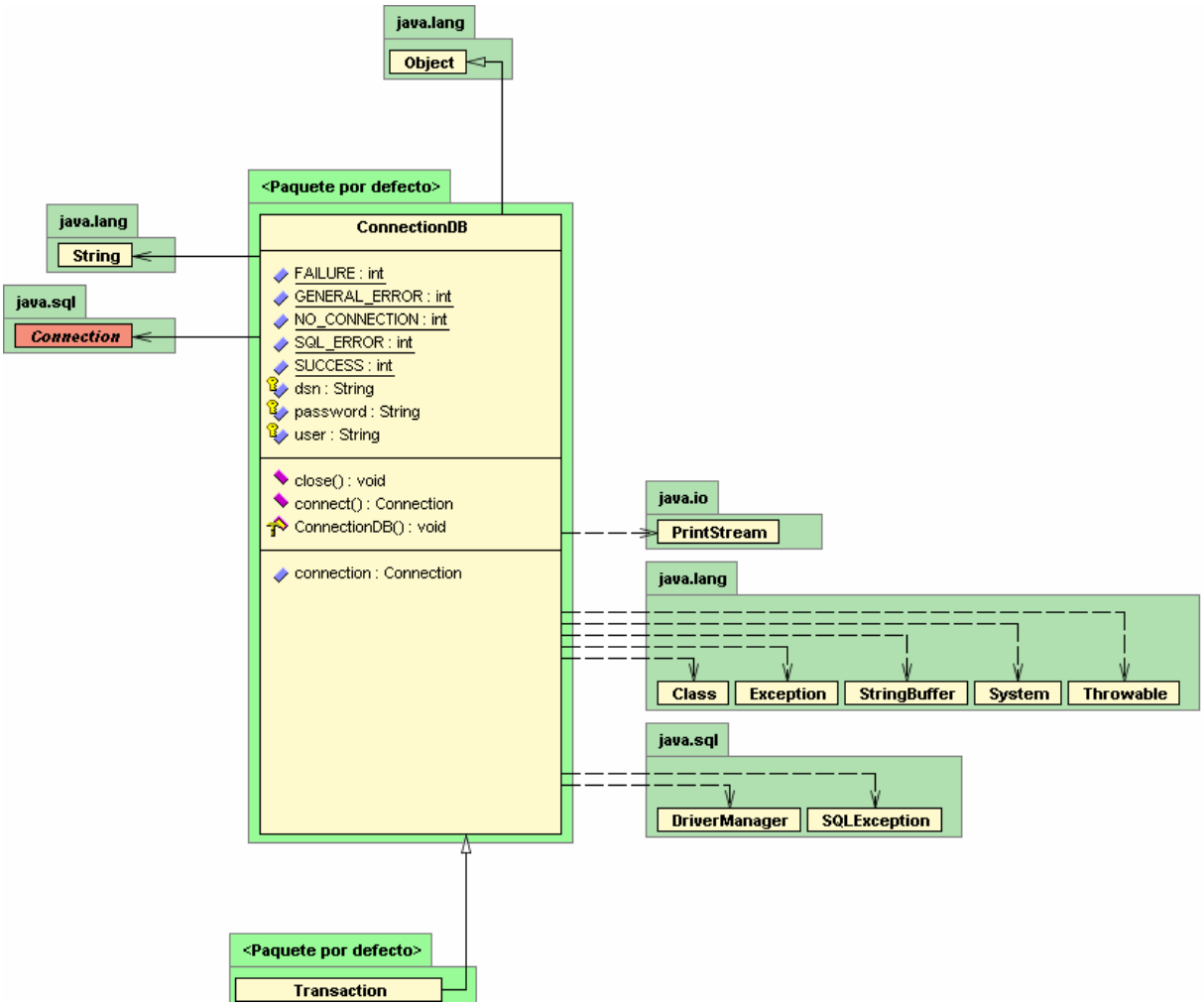



**Figura 3-6** Clase ConnectionDB.

El campo connection mantiene almacenada una instancia de la clase java.sql.Connection que contiene la conexión actual a un driver ODBC de un DBMS en particular. Los campos dsn (*data source name*), user y password almacenan información para la conexión a la base de datos (configurado desde el administrador de drivers ODBC).

- **Clase Transaction**

Esta clase proporciona métodos para llevar a cabo el procesamiento de Transacciones Anidadas Cerradas y Transacciones Anidadas Abiertas. Esta clase implementa el comportamiento clásico de un sistema transaccional agregando la funcionalidad del modelo de TAC y TAA, modelos con los que es posible dividir el procesamiento de las

transacciones en un árbol jerárquico de sub-transacciones, mejorando el rendimiento de las aplicaciones basadas en estos modelos así como la tolerancia a fallas. La figura 3-7 ilustra la clase `Transaction`.

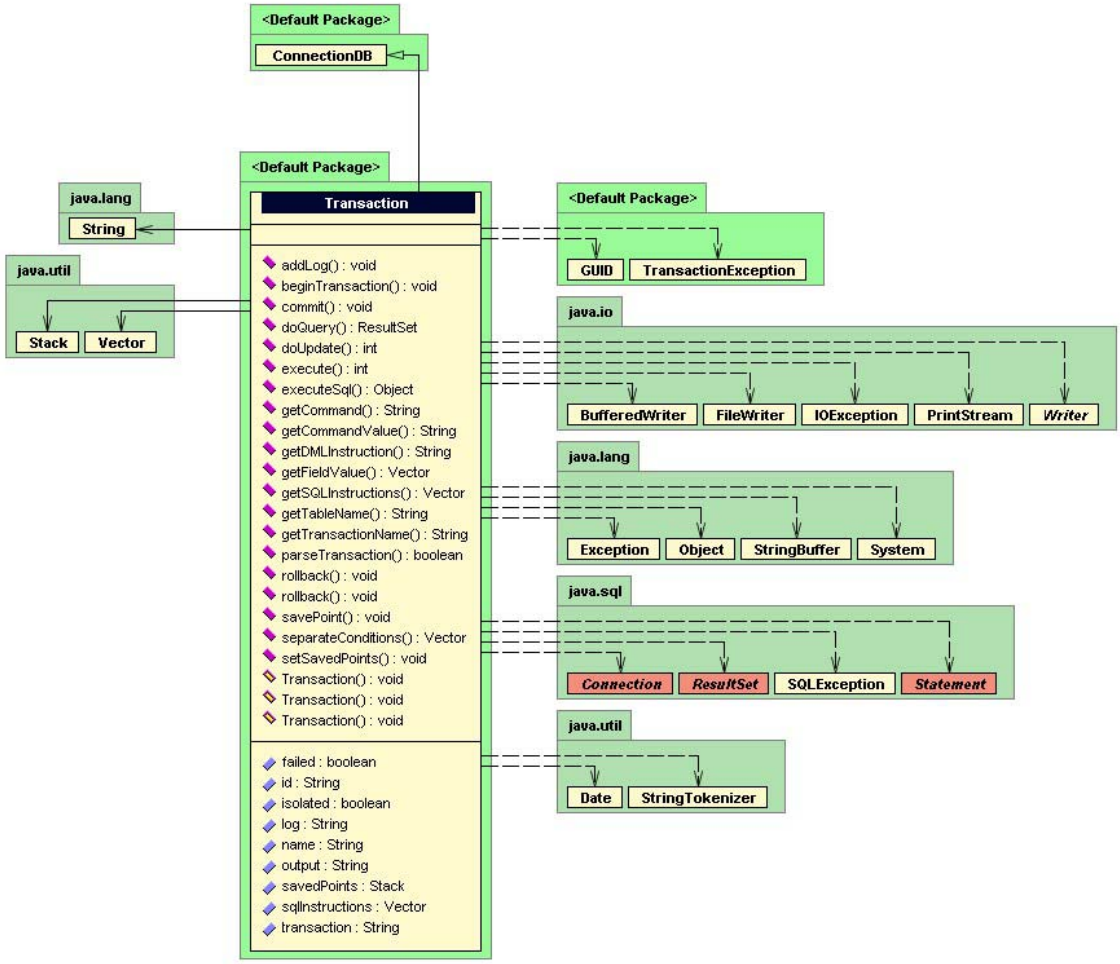



**Figura 3-7** Clase Transaction.

Entre los métodos básicos para el procesamiento de transacciones tales como: *begin(), commit()* y *rollback()*, la clase `Transaction` cuenta con métodos para dar a las transacciones el comportamiento de los modelos TAC y TAA los cuales pueden comunicarse con un MTA, el cual controla y administra los recursos utilizados por cada transacción, en especial los objetos bloqueados por cada transacción. Este comportamiento permite a las transacciones enviar al monitor cada objeto utilizado durante el procesamiento, el monitor a su vez controla las versiones de estos objetos con el objetivo de vigilar su consistencia. Debido a que el nivel de aislamiento es relajado entre las transacciones, cada transacción puede observar valores no-confirmados por otras transacciones pudiendo resultar en un escenario de malas dependencias. Sin embargo, el monitor se basa en prioridades para seleccionar a las transacciones candidatas para llevar a cabo una cancelación parcial e incluso reintentar las operaciones canceladas con el objetivo de garantizar la consistencia de los datos.

- **Clase TransactionException**

Esta clase que hereda la funcionalidad y comportamiento de la clase base `Exception`, permite capturar cualquier situación anómala durante el procesamiento de las transacciones. Los desarrolladores deben de capturar esta excepción con la cual pueden implementar un mecanismo de tratamiento de fallas adecuado.

La clase `TransactionException` es lanzada bajo cualquier falla ocasionada durante el procesamiento de las transacciones tales como: errores al comunicarse con la fuente de los datos, fallas de comunicación con el MTA, fallas por tiempo de espera agotado, entre otras. Los desarrolladores pueden tomar una acción de cancelación si obtienen una excepción bajo este contexto como una cancelación total de la transacción. La figura 3-8 ilustra un ejemplo de una acción de cancelación.

```
Transaction t = new Transaction()
try {
    t.begin();
    t.doUpdate(" update emp " +
               " set sal = sal + 1000 "
               " where deptno = 10;");
    t.commit();
} catch( TransactionException e) {
   // cancelar
   t.rollback();
} finally {
  t.close()
 } // :~
```

**Figura 3-8** Capturando la excepción `TransactionException`.

La clase `TransactionException` está compuesta por tres constructores que se ilustran en la figura 3-9. Estos constructores permiten lanzar una excepción de tres formas diferentes:

- `public TransactionException()` : se origina por un error genérico dentro del procesamiento de la transacción, por ejemplo, por un error de comunicación entre el monitor y la transacción.
- `public TransactionException(String msg)` : esta excepción es lanzada dentro de los métodos de la clase `Transaction` o `ConnectionDB`, con un mensaje de error indicando el origen de la falla.
- `public TransactionException(int codigo)` : esta excepción se obtiene por un error definido en una lista de códigos de error dentro de la clase, como INVALID_SAVEPOINT cuando se intenta llevar a cabo una cancelación parcial a un punto inválido.

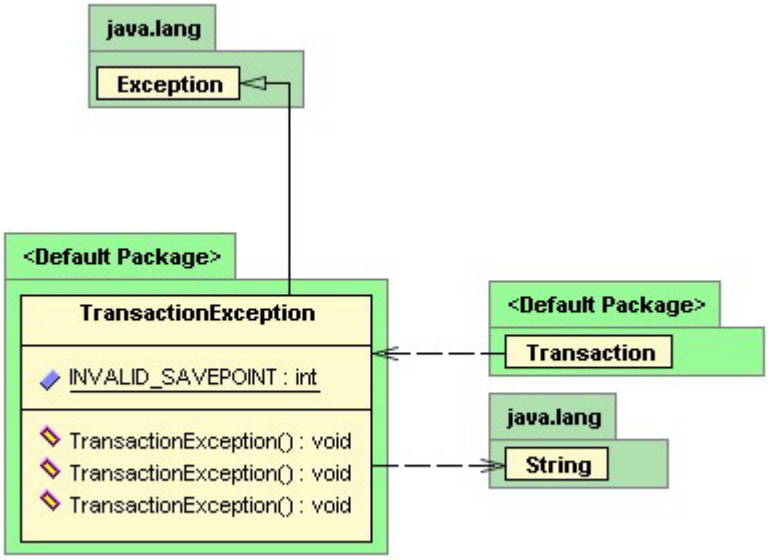



**Figura 3-9** Clase `TransactionException`.

- **Clase PartialRollbackException**

Esta clase hereda la funcionalidad y comportamiento de la clase base `TransactionException` permite capturar errores ocasionadas por malas dependencias. Es decir, inconsistencias generadas por la concurrencia de las transacciones-resultado del relajamiento del nivel de aislamiento de las transacciones.

Los desarrolladores pueden entonces capturar estas anomalías y tomar una de dos acciones:

1. Llevar a cabo una acción alternativa para solucionar el problema ocasionado por las malas dependencias capturando específicamente la excepción `PartialRollbackException` como se ilustra en la figura 3-10, ó
2. Dejar que el monitor reintente las operaciones que obtuvieron o que generaron valores inconsistentes entre las transacciones, capturando la excepción `TransactionException`.

```
Transaction t = new Transaction("t1");

try {
  t.begin();
  t.doQuery("select dname from dept where deptno = 10;");
  t.doQuery("select loc from dept where deptno = 10;");
  t.commit();
} catch(PartialRollbackException e) {
  try {
    String savepoint = e.getSavePoint();
    t.rollback(savepoint);
    t.doQuery("select loc from dept where deptno = 10;");
```

```
      t.commit();
    } catch(TransactionException e) {
      t.rollback();
    } finally {
      t.close();
    } // try ... catch ... finally
```

**Figura 3-10** Capturando la excepcion `PartialRollbackTransaction`.

La figura 3-11 ilustra los constructores y los métodos que dan la funcionalidad a la clase **`PartialRollbackTransaction`**. El objeto que se crea con la excepción contiene los métodos necesarios para obtener información acerca del **`savepoint`** al que se debe llevar a cabo una cancelación parcial:

```
} catch(PartialRollbackException e) {
  try {
    String savepoint = e.getSavePoint();
    t.rollback(savepoint);
    t.doQuery("select loc from dept where deptno = 10;");
    t.commit();
  } catch(TransactionException err) {
    // cancelar transacción
    t.rollback();
  }
}
```

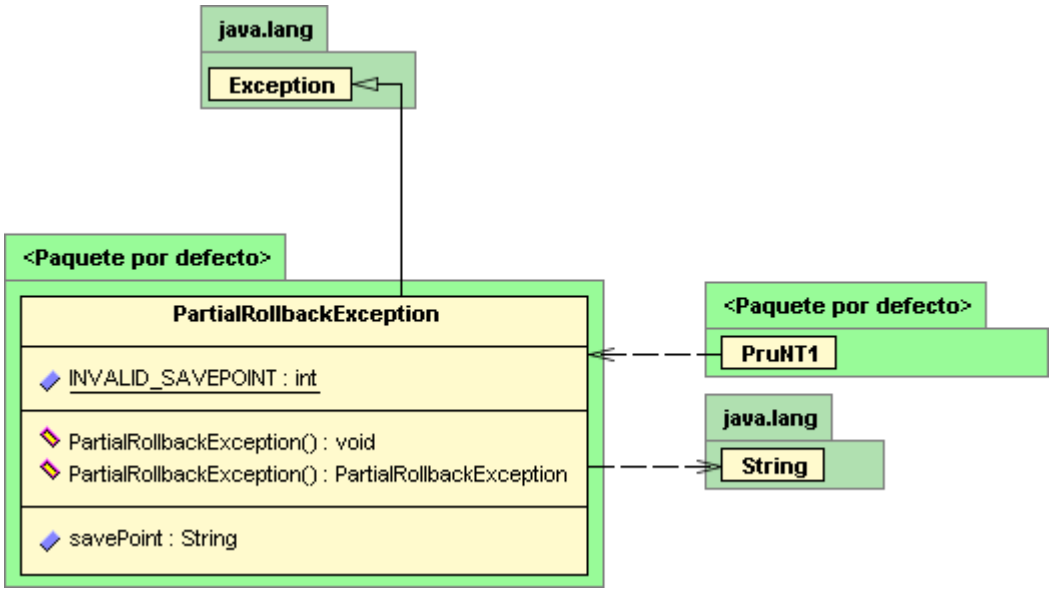



**Figura 3-11** Clase `PartialRollbackTransaction`.

- **Clase TransactionThread**

Esta clase le da a las transacciones la capacidad de comunicación con el monitor y las demás transacciones involucradas en el procesamiento de una transacción anidad cerrada o abierta. Esta clase hereda la funcionalidad y comportamiento de la clase Thread, con la que es posible crear hilos de ejecución para explotar el multiprocesamiento entre las aplicaciones (ver figura 3-12).

Cada vez una nueva transacción es creada e iniciada, se levanta un servicio (`TransactionThread`) el cual es el encargado de establecer la comunicación con el MTA, enviando mensajes de inicio, operaciones realizadas y objetos bloqueados. A su vez, el monitor le informa a la transacción el modelo bajo el cual debe operar pudiendo ser TAC o TAA. En el caso de las TAC, cada transacción que ha terminado con éxito todas sus operaciones y esté lista para finalizar, debe esperar a que el monitor padre emita el mensaje de finalización para cada transacción. Esto es posible gracias a diálogos entre el monitor y la transacción, cuyas directivas se encuentran definidas en la clase `TransactionDefinitions`.

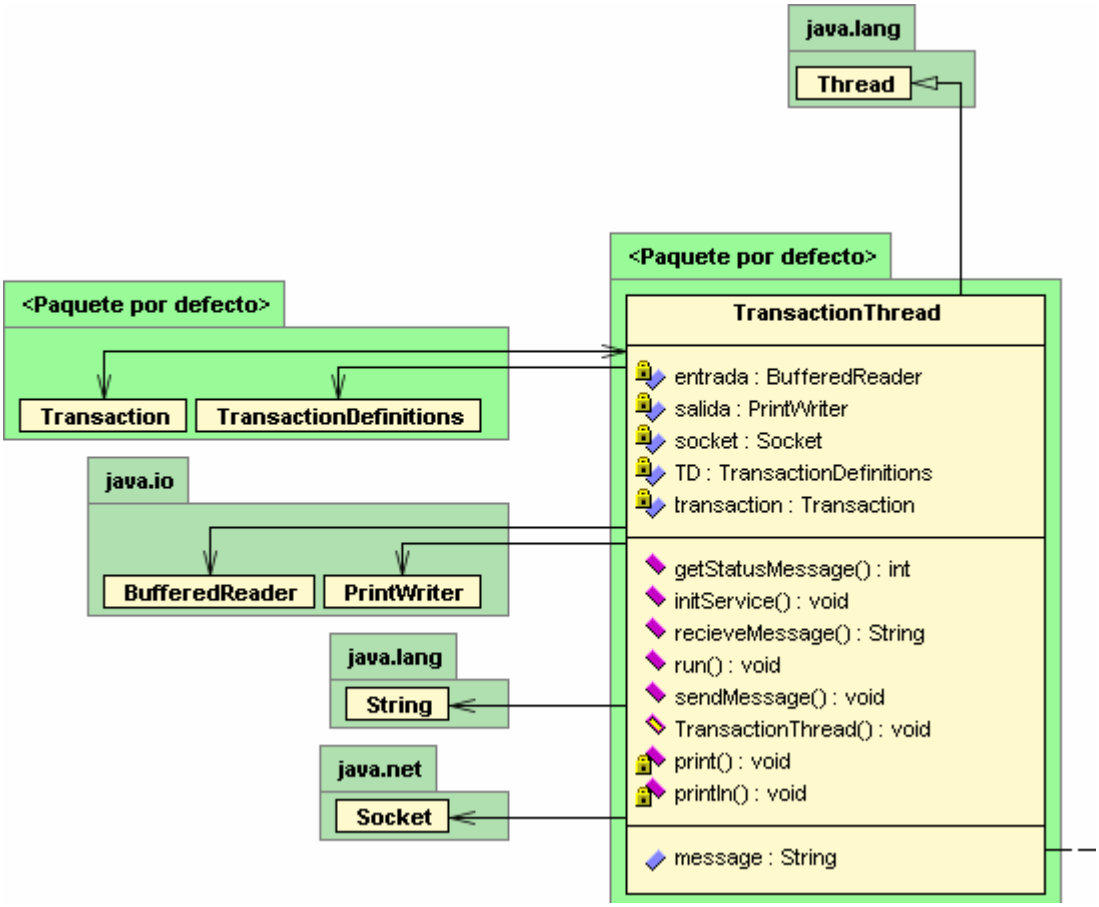



**Figura 3-12 Clase `TransactionThread`.**

### 3.2.4 Diagrama de secuencias

El envío de mensajes entre las transacciones y el MTA permiten crear un escenario de TAC ó TAA,

- **Secuencias para la inicialización de los servicios**

En la figura 3-13 se ilustra la secuencia de eventos que ocurren entre una transacción y su monitor hijo (transacción padre) y este a su vez con su respectivo monitor padre (transacción raíz).

El proceso inicia con la inicialización del monitor padre (1), el cual funciona a través de una consola de administración donde es posible visualizar las transacciones que este monitor administra así como otros monitores (dependientes) en diferentes niveles del árbol de transacciones. La consola inicia el servicio (2) creando un hilo (thread) el cual recibirá mensajes de otros monitores así como de sus transacciones. En el paso (3) se inicia la ejecución del monitor hijo el cual debe especificar la ubicación (dirección IP y puerto) del monitor padre (4). Una vez conseguida la conexión con el monitor padre (5) este le envía un mensaje con el modelo de transacciones a procesar, ya sea TAC o TAA (6) y el monitor hijo procede a iniciar su servicio (7) tal y como lo realizó el monitor padre.

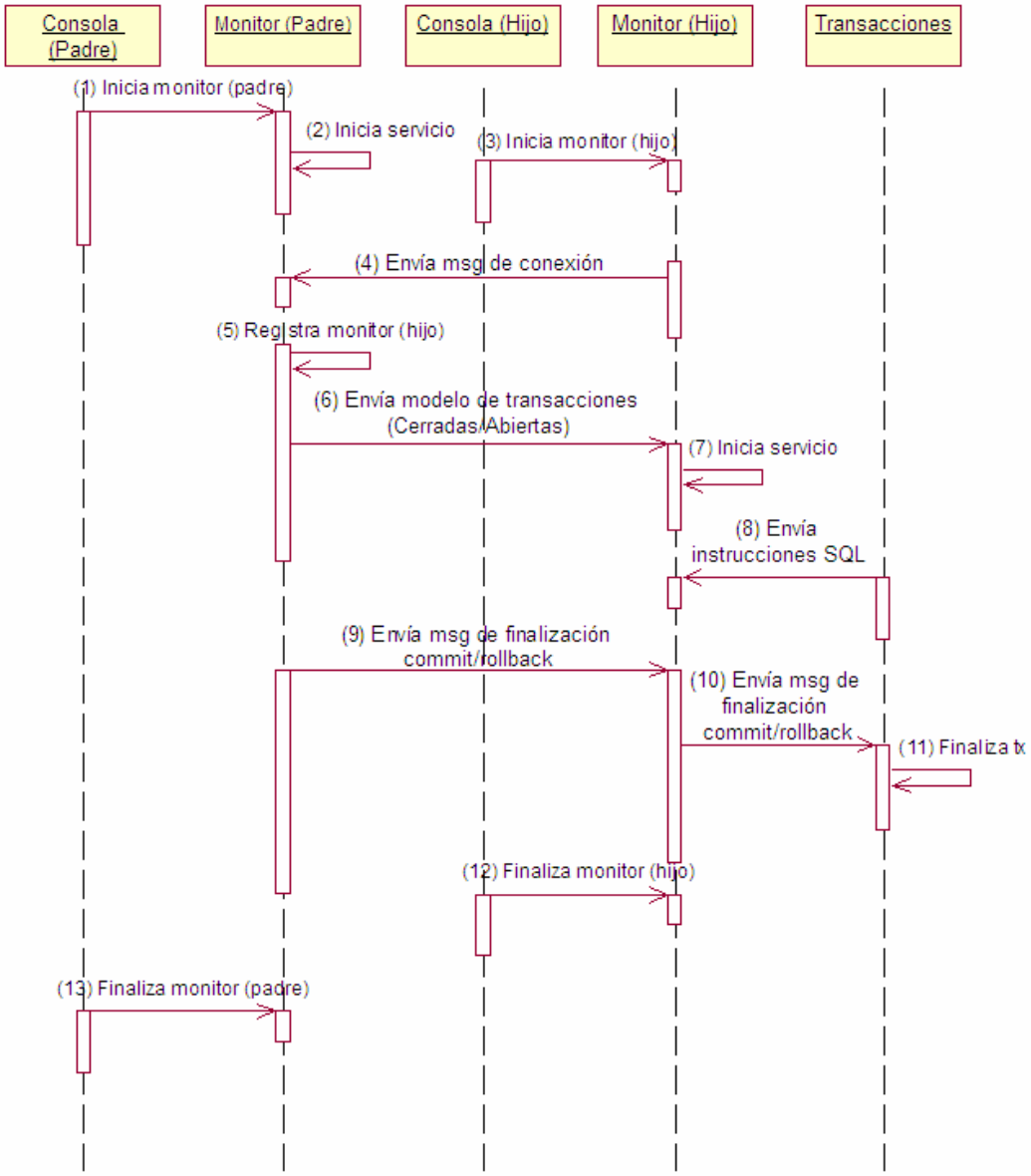



**Figura 3-13** Diagrama de secuencias de una transacción anidada.

En este punto ya es posible recibir peticiones de cualquier transacción (8) a través de instrucciones en lenguaje SQL. Así cada transacción enviará mensajes de operaciones realizadas al respectivo monitor llevando a cabo el rol de "Transacción Padre". Una vez que una transacción ha finalizado espera el mensaje de finalización (9) de su respectivo monitor, terminando con ello la transacción (10,11). Eventualmente los monitores pueden seguir activos para recibir nuevos mensajes de transacciones o finalizar su ejecución (12, 13).

- **Secuencia de eventos entre una aplicación y el monitor**

La figura 3-14 ilustra la secuencia de eventos entre una aplicación que es capaz de procesar transacciones y el MTA.

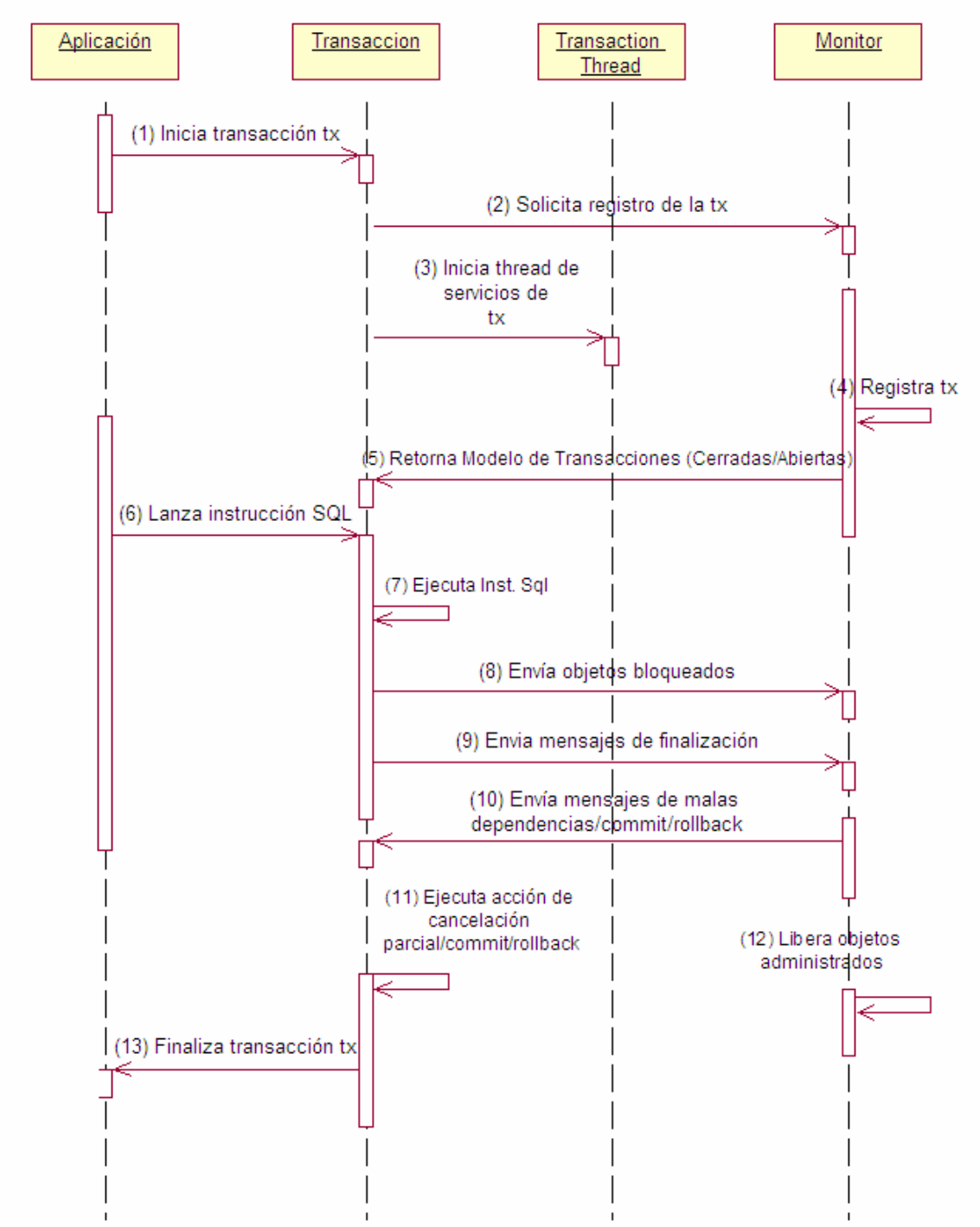



**Figura 3-14** Secuencia de eventos entre una aplicación y el monitor.

La aplicación inicia la transacción Tx (1) y a su vez puede lanzar varias transacciones. Tx envía un mensaje de inicio al monitor (2) el cual recibe como respuesta del monitor el modelo de transacciones a procesar (5). La transacción ejecuta instrucciones sql interactuando con el monitor enviándole los objetos procesados por cada instrucción (6, 7, 8). En el paso (9) la transacción finaliza la ejecución y espera hasta que el monitor confirme la finalización ya sea con éxito o cancelación (10, 11). El monitor libera los objetos que administró (12) y Tx termina su ejecución (13).

# Capítulo 4 Implementación: administración de malas dependencias

En [19] se propone el modelo de transacciones anidadas (TA) para mejorar la tolerancia a fallas en ambientes distribuidos. Este modelo cuenta con un mecanismo de control de concurrencia basado en el protocolo 2PL con herencia. 2PL es un protocolo basado en bloqueos, por lo que las peticiones de acceso a los recursos (datos), deben ser protegidos por bloqueos. Para operaciones de lectura se solicita un bloqueo compartido (shared-lock o simplemente slock), y para una operación de escritura se requiere de un bloque exclusivo (exclusive-lock o simplemente xlock).

Las reglas para adquirir los bloqueos en el modelo TA, son descritos en la sección 2.2.3. Una desventaja de este control de concurrencia es que otras transacciones que soliciten bloqueos a objetos que previamente han sido bloqueados, tengan que esperar hasta que los bloqueos sean liberados o sean administrados por su transacción-padre. Esto ocasiona que el rendimiento de las transacciones disminuya. En este trabajo se proponen dos extensiones:

1. Para operaciones de lectura, elevar la visibilidad de los objetos a todos los niveles del árbol, y

2. Para operaciones de escritura, relajar el nivel de aislamiento de las transacciones, con el objetivo de que puedan observar valores no-confirmados de objetos previamente bloqueados, y en el caso de surgir malas dependencias, utilizar un mecanismo de abortos parciales utilizando *savepoints*.

## 4.1 Transacciones Anidadas Cerradas

Al modelo propuesto por Moss en [19] es conocido también como modelo de Transacciones Anidadas Cerradas. Debido a que las sub-transacciones no son capaces de confirmar el trabajo realizado hasta que la transacción raíz, decida confirmar. Si la transacción raíz decide abortar, entonces todas las sub-transacciones deberán abortar también. A continuación se describe una extensión al control de concurrencia utilizado en este modelo para operaciones de lectura y/o escritura.

### 4.1.1 Operaciones de solo-lectura

El control de concurrencia para el modelo TA, está basado en el del modelo de transacciones anidadas [19], el cual utiliza el protocolo 2PL extendiéndolo con herencia de bloqueos. Es decir, los bloqueos se van adquiriendo por dos opciones, (1) si el objeto solicitado esta libre o (2) si ha sido bloqueado previamente y el padre de la transacción solicitante lo administra. Sin embargo, con este enfoque, si una transacción A solicita el acceso a un objeto que ha sido bloqueado por una transacción B en diferentes ramas del árbol, la transacción A tendrá que esperar hasta que el objeto sea liberado o heredado a su

padre inmediato. Este mecanismo de bloqueos con herencia garantiza el aislamiento pero disminuye el rendimiento de las aplicaciones.

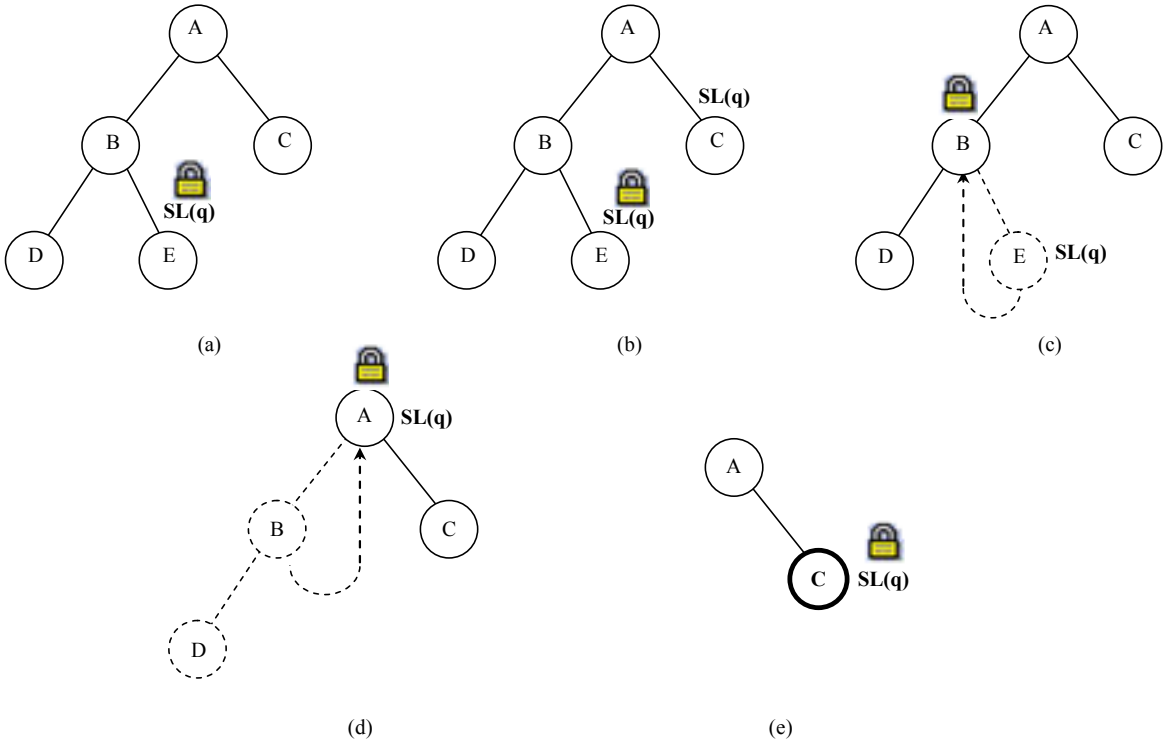



**Figura 4-1** Herencia de un bloqueo hacia los padres.

La figura 4.1 muestra un ejemplo de este escenario. En la figura 4.1a la transacción E solicita un bloqueo compartido sobre el objeto *q*. Posteriormente la transacción C solicita un bloqueo compartido sobre el mismo objeto *q* bloqueado por E (ver figura 4.1b). Entonces C debe esperar hasta que el objeto *q* sea liberado y administrado por el predecesor de C (transacción A). Las figuras 4.1c y 4.1d muestran el proceso de herencia hacia los padres, por parte de las sub-transacciones E y B al momento de terminar ya sea con *commit* o *abort*.

Finalmente en la figura 4.1e se muestra la adquisición del bloqueo sobre *q* solicitado por la transacción C. Para evitar este tiempo de espera, se propone extender la visibilidad de un objeto bloqueado en modo compartido a todos los niveles del árbol. Para que cada transacción sea capaz de bloquear un objeto en modo compartido aún cuando este ya haya sido bloqueado en el mismo modo. Nuestro enfoque se basa en la compatibilidad que tienen los bloqueos de lectura (slocks) ya que estos no generan malas dependencias a pesar del aborto de las transacciones [15].

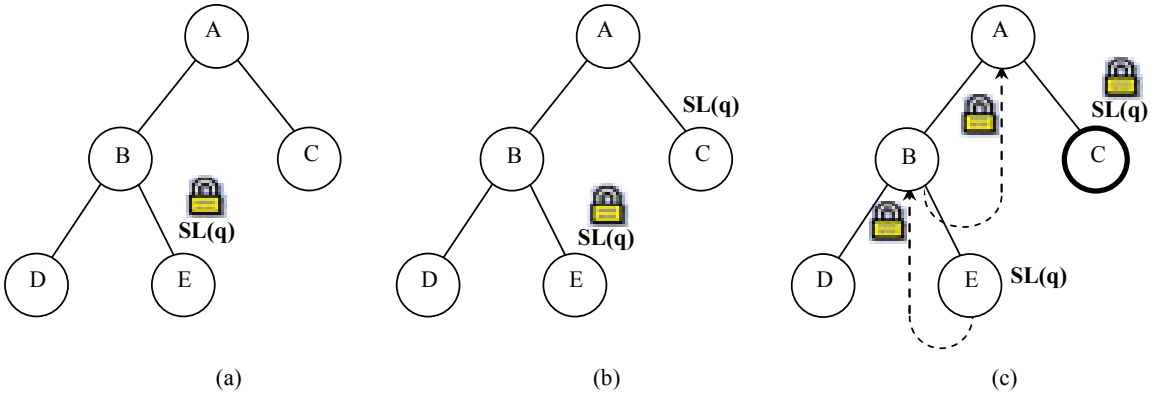



**Figura 4-2** Herencia del bloqueo a todos los niveles.

En la figura 4.2, se muestra el mismo escenario, donde la transacción E, bloquea en primer instancia al objeto *q*, heredando el bloqueo al resto de las transacciones. Es decir, el objeto *q* permanece disponible a otras transacciones aún cuando E aborte. Entonces la transacción C es capaz de obtener un bloqueo compartido sobre el objeto *q,* aún cuando E no haya terminado (ver figura 4.2c).

Con este enfoque, las aplicaciones mejorarán su rendimiento para aquellas transacciones de solo-lectura bajo el modelo de TA, debido a que, como ya se ha dicho, la compatibilidad de los bloqueos compartidos, no genera malas dependencias aún cuando las transacciones involucradas tengan que abortar por fallas [14].

### 4.1.2 Operaciones de lectura/escritura

Las operaciones de escritura requieren de un bloqueo exclusivo de cada objeto bajo el protocolo 2PL, debido a que cambian el estado de los objetos. De esta forma, quedan protegidas cuando otras transacciones concurrentes requieran acceder al mismo objeto y así generar malas dependencias. Sólo dos escrituras al mismo objeto en la misma transacción no comprometen la consistencia [14]. Sin embargo, los bloqueos exclusivos no permiten a otras transacciones acceder a un recurso bloqueado, hasta que éste sea liberado. Por lo tanto, las aplicaciones disminuyen su rendimiento debido a los tiempos de espera. La norma ANSI-SQL92 [1] ha definido cuatro niveles de aislamiento con el propósito de relajar los bloqueos dentro de una transacción y mejorar el rendimiento de las aplicaciones, como se describió en la sección 2.6.

Utilizando un nivel de aislamiento menos restrictivo, es posible aumentar el rendimiento de una aplicación debido a que no es necesario que ésta tenga que esperar hasta que los recursos sean liberados. En varios DBMS comerciales, el nivel de aislamiento por defecto es *READ COMMITTED* (ver sección 2.6). Este nivel evita la presencia de *lecturas sucia,* debido a que una transacción sólo puede leer valores confirmados por otras transacciones.

Por un lado, relajar el nivel de aislamiento mejora el rendimiento de las aplicaciones pero puede ocasionar la presencia de malas dependencias. Por otro lado, si se restringe el acceso a datos no confirmados con un nivel de aislamiento más restrictivo *(Serializable)* se

pueden presentar casos de abrazo mortal *(deadlock)* o abortos en cascada [12]. En ambas situaciones la naturaleza del problema puede ayudar a escoger el nivel de aislamiento adecuado para cada situación.

### 4.1.3 *Savepoints* para evitar abortos en cascada

Relajar el nivel de aislamiento de las transacciones concurrentes puede ocasionar malas dependencias. Por ejemplo, si existe una $T_i$ que lee el objeto x, $r_i(x) := x$, obteniendo la misma versión del objeto (versión 1), posteriormente una transacción $T_j$ modifica el mismo objeto x mediante una escritura cambiando la versión del objeto x, $w_j(x) := x_j$ (versión 2), como se ilustra en la figura 4.3.

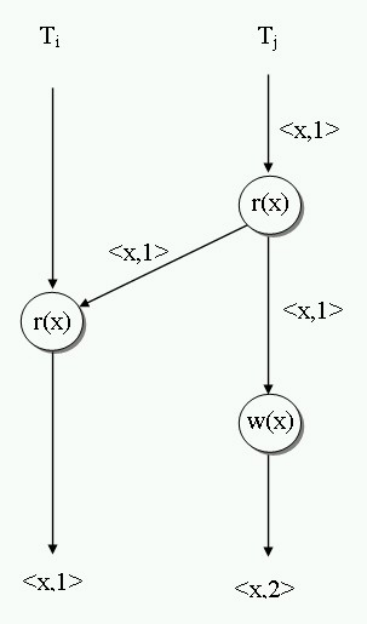



**Figura 4-3** Lectura sucia.

Si $T_i$ realizara una nueva lectura sobre x obtendrá un valor diferente, conocida como *lectura no-repetible* (ver sección 2.3). Si $T_j$ realiza un *rollback* entonces $T_i$ deberá hacer *rollback* también debido a la dependencia creada entre $T_i$ y $T_j$. Esto resulta en un escenario de "abortos en cascada"; en general para todas aquellas transacciones que hayan creado dependencias con $T_j$.

En la figura 4.4, las transacciones $T_i$ y $T_j$, tienen acceso al mismo objeto *x*. En primer instancia, $T_j$ realiza una lectura sobre x: $r_j(x) := x1$, posteriormente $T_i$ realiza la misma operación sobre x : $r_i(x) := x1$ sin cambiar el valor, en este momento $r_i(x) = r_j(x)$. La operación $r_i(x)$ se lleva acabo en el tiempo 1: *(t1)*. A continuación $T_j$ lleva a cabo una operación de escritura sobre el objeto x: $w_j(x) = x2$ (esta operación es permitida siempre y cuando $T_i$ y $T_j$ estén ejecutándose bajo el nivel de aislamiento *READ UNCOMMITED)*.

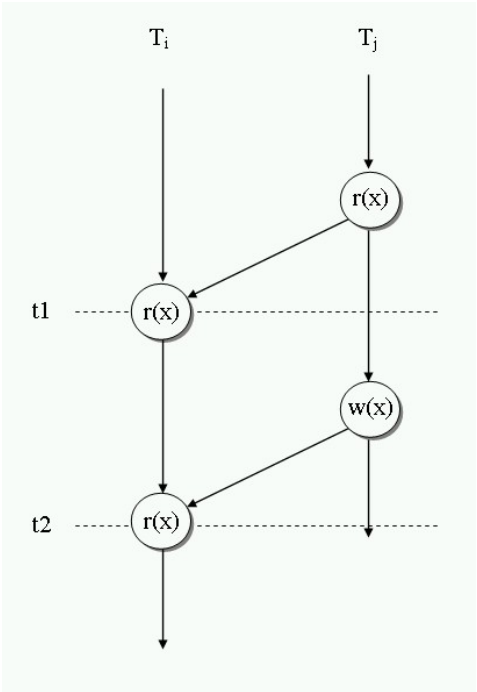



**Figura 4-4** Grafo de dependencias entre $T_i$ y $T_j$.

En este momento, las versiones del objeto x para las transacciones $T_i$ y $T_j$ son diferentes. $T_i$ mantiene el valor de la versión x1, mientras que $T_j$ mantiene el valor de la versión x2. Por lo tanto $T_i$ debe abortar el trabajo realizado, debido a la mala dependencia generada (lectura-sucia) presentando un escenario de aborto en cascada. Note que las operaciones de $T_i$ con el valor de la versión 1 (x1) son consistentes hasta que $T_j$ lo modifica a la versión x2. Es decir, $T_i$ es consistente hasta el tiempo *t1*, cuando sucede un cambio de versión del objeto x por la $T_j$ en el tiempo *t2*. Por lo tanto sólo es necesario abortar las operaciones realizadas después del tiempo t1 como se muestra en la figura 4.5.

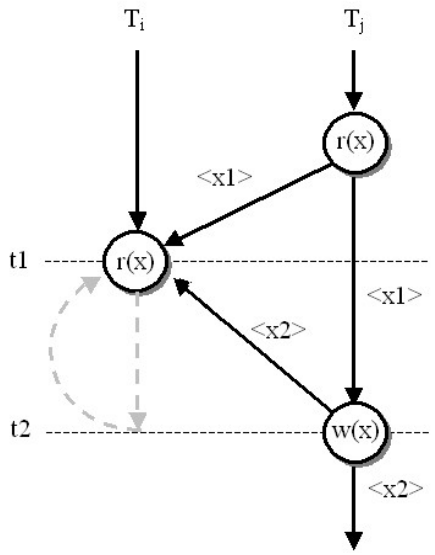



**Figura 4-5** Aborto parcial hasta un estado consistente.

De esta forma, $T_i$ no requiere del aborto total de la transacción, en su lugar, puede realizar un aborto parcial hasta el tiempo *t1*, cancelando sólo aquellas operaciones que se convirtieron en inconsistentes.

Un *savepoint (SP)* permite demarcar en unidades lógicas a una transacción. Cada unidad de trabajo marcada por un SP, representa un estado consistente que puede ser preservado aún cuando ciertas partes de la transacción tengan que ser abortadas [14]. Utilizando SP es posible realizar abortos de las operaciones realizadas posteriores al savepoint, a lo cual se denomina **aborto parcial** de la transacción.

Sin embargo, el uso de savepoints es secuencial. Por ejemplo, suponga que ha creado las siguientes marcas en una transacción *SP1, SP2* y *SP3*. No puede cancelar únicamente el SP2 sin tener que cancelar primeramente el SP3, de la misma forma no podrá cancelar el SP1 sin antes cancelar las marcas SP3 y SP2. Por otro lado, los SPs sólo demarcan la transacción, pero los cambios realizados en cada unidad lógica de trabajo, permanecen sin ser confirmados hasta que la transacción realice un *commit.* Si la aplicación o el sistema fallan, la transacción tendrá que realizar un *rollback* sin importar los SPs creados.

## 4.2 Algoritmo para administración de abortos

Los *savepoints* son útiles para recuperar la consistencia de los objetos inconsistentes debido a la presencia de malas dependencias, deshaciendo aquellas operaciones posteriores al SP.

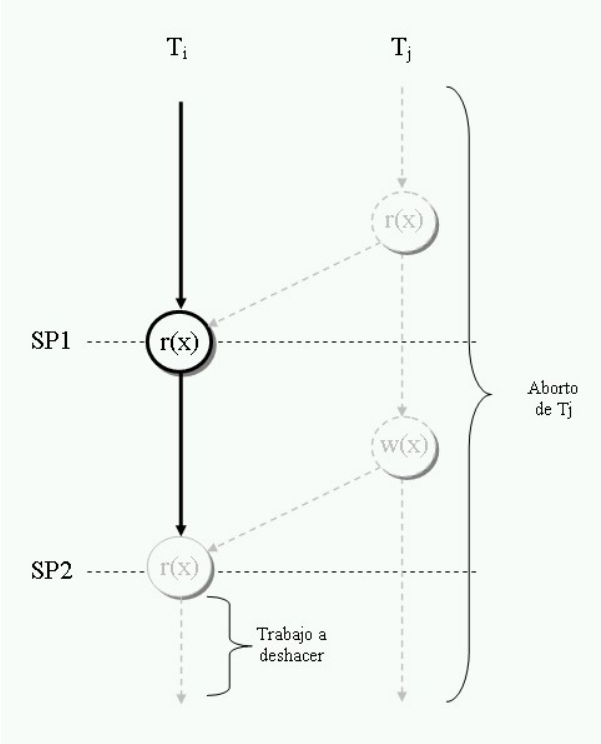



**Figura 4-6** Rollback hasta el Savepoint 2.

En la figura 4.6, la transacción Tj presenta una falla y es necesario abortar. Sin embargo, Ti puede solo deshacer los cambios realizados hasta el SP2, que es donde existe dependencia con Tj, sin tener que cancelar Ti totalmente.

Para llevar a cabo abortos parciales, se requiere del monitoreo de los bloqueos y las versiones de los objetos que se van generando durante las operaciones de lectura/escritura. Además es necesario un mecanismo de paso de mensajes entre las transacciones que crean dependencias, para mantener el conocimiento de:

1. cambios de versión de los objetos procesados concurrentemente,

2. el aborto total o parcial de una transacción que mantiene dependencias con otras transacciones,

3. la finalización de una transacción de manera exitosa liberando los bloqueos adquiridos.

La figura 4.7, ilustra un ejemplo del envío de mensajes entre un MTA y dos transacciones (T1 y T2) las cuales son concurrentes y crean dependencias entre ellas.

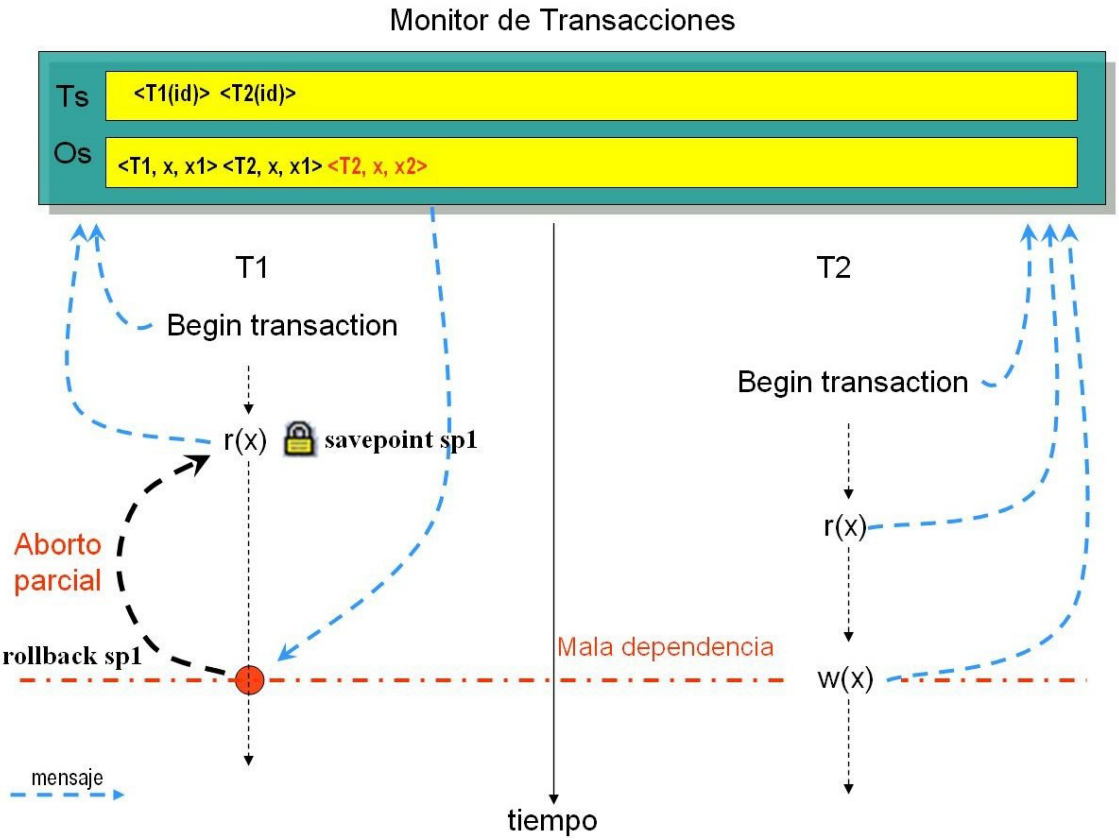



**Figura 4-7** Monitoreo de transacciones concurrentes.

El monitor de transacciones (MTA), mantiene una lista de las transacciones que se encuentran en ejecución (Ts). Así mismo, una lista para los objetos que han sido procesados

por cada transacción (Os). En la figura 4.7 se puede observar que el MTA ha registrado el inicio de las transacciones T1 y T2, almacenando el identificador de cada transacción (ID) - <T1(id)> <T2(id)> -. Cuando una transacción es iniciada, ésta adquiere un identificador único, que la distingue de las demás transacciones. En la lista Ts se registra el estado de cada transacción, una vez que una transacción ha finalizado con éxito (*commit*) su registro es removido de la lista Ts, así como el registro de sus objetos procesados.

La lista Os, mantiene el registro de los objetos actualmente procesados por las transacciones activas en la lista Ts. Cuando una transacción realiza una operación de lectura o escritura, envía un mensaje al MTA, informando el objeto procesado y la versión obtenida resultado de la operación ya sea una lectura o una escritura (las lecturas no cambian la versión del objeto leído y las escrituras generan nuevas versiones). Los datos del mensaje incluyen:

- El ID de la transacción,
- y una triupla <t,x,v> que define: la transacción *t* procesando el objeto *x* y generó la versión *v.*

Por ejemplo, la triupla <T1, x, x1>, indica que la transacción T1 procesa el objeto *x* generando la versión *x1*. Entonces el MTA, verifica las versiones registradas para el objeto *x,* comparando sólo aquellas triuplas que sean capaces de producir una mala dependencia (ver sección 2.5). Este enfoque se basa en que: dos lecturas al mismo objeto de diferentes transacciones, no comprometen el aislamiento, y dos escrituras al mismo objeto en la misma transacción, tampoco compromete el aislamiento. Sólo si dos operaciones a un mismo objeto en transacciones diferentes y al menos una de ellas es una escritura, entonces se puede ver afectado el aislamiento y por consiguiente la consistencia de los datos [14, 5].

Para corregir los efectos de las malas dependencias, el MTA utiliza la lista Os para verificar constantemente las versiones de los objetos procesados. Por ejemplo, considere dos transacciones T1 y T2, cuyas operaciones son mostradas en la siguiente historia:

```
<T1, r, x>
<T2, w, x>
<T1, r, x>
```

En esta historia, se genera la mala dependencia conocida como *lectura no-repetible.* Con el enfoque basado en bloqueos (sección 2.3.3), esta mala dependencia puede ser evitada ya que T2 adquiere un bloqueo exclusivo sobre el objeto x, y no es compatible con los bloqueos compartidos de T1:

```
<T2, w, x>
<T1, r, x>
<T1, r, x>
```

La serialización de las lecturas de T1 garantiza un valor consistente, sin embargo los bloqueos tienen la desventaja de presentar problemas de bajo rendimiento de las aplicaciones y abrazos mortales entre las transacciones. Con el enfoque de abortos parciales la historia puede ser ejecutada de la siguiente manera:

```
<T1, r, x>
 <T1, sp1>
<T2, w, x>
 <T2, sp1>
<T1, r, x>
 <T1, sp2>
<T1, rollback sp1>
 <T1, redo: r(x)>.
```

Con este enfoque la transacción T1 no tienen que esperar por objetos previamente bloqueados, entonces T1 evitará cancelar totalmente debido a que puede ejecutar un aborto parcial hasta el *savepoint* sp1 (`<T1, sp1>`) y sólo tendrá que rehacer aquellas operaciones posteriores al sp1 (la segunda lectura).

### 4.2.1 Algoritmo para abortos parciales

A continuación se describe el mecanismo para llevar a cabo un aborto parcial, con el cual es posible preservar piezas de trabajo consistentes – marcadas por savepoints[5] – en lugar de cancelar una transacción completa ante la presencia de malas dependencias:

1. Si existe una mala dependencia entre dos transacciones Tx y Ty, entonces el MTA selecciona a una de ellas como candidata a llevar a cabo ya sea un aborto parcial o un aborto total.
2. Una vez seleccionada la transacción candidata al aborto, el MTA le envía un mensaje de malas dependencias `<ST_INCONSISTENT>`.

El código de la figura 4-8 ilustra una transacción que es capaz de tolerar fallas con el uso del método `redo()` el cual es ejecutado si la transacción lanza una excepción del tipo `PartialRollbackException`.

```
Transaction t = new Transaction();

try {
  t.connect("dsn", "usr", "password");
  t.begin();
  t.doQuery("select loc from dept where deptno = 10;");
  t.commit();
} catch(PartialRollbackException e) {
  t.redo();
```

[5] Los *savepoints* permiten llevar a cabo abortos parciales de unidades de trabajo previamente marcadas. Sin embargo la cancelación individual hasta una de estas marcas ocasionará la cancelación de aquellas marcas posteriores a la marca que se está cancelando.

```
} catch(TransactionException e) {
  t.rollback();
} catch(SQLException e) {
  System.out.println("Error SQL: " + e);
} finally {
  t.close();
}
```

**Figura 4-8** Código para lanzar una Transacción tolerante a fallas.

El método `redo()` indica a la transacción que reintente las operaciones que generaron datos inconsistentes. La excepción `PartialRollbackException`, es lanzada cuando se presentan malas dependencias, por lo tanto, el método `redo()` vuelve a ejecutar aquellas instrucciones que crearon inconsistencias, para así llegar a un nuevo estado consistente.

Este enfoque está basado en *savepoints* donde cada transacción establece un *savepoint* por cada instrucción ejecutada. Esto permite la cancelación de ciertas partes de la transacción en lugar de cancelar la transacción completa. Esto hace posible preservar más cantidad de trabajo a pesar de la presencia de fallas. Lo cual lo convierte en una opción robusta para los ambientes de cómputo móvil.

- **Detectando el punto (savepoint) a deshacer**

Cuando surge una mala dependencia entre dos o más transacciones, el monitor MTA envía el mensaje `<ST_INCONSISTENT>` a las transacciones involucradas, por lo tanto cada transacción debe llevar a cabo un "aborto parcial" al punto donde la transacción se convirtió en inconsistente.

Para descubrir el punto al que cada transacción debe realizar un aborto parcial, se emplea el siguiente mecanismo:

1. Cada transacción conserva un vector de los *savepoints* creados durante la ejecución de la misma, por ejemplo, si una transacción ha ejecutado 5 operaciones el vector de *savepoints* contendrá una longitud de 4 elementos como se ilustra en la figura 4-9. Note que no es necesario asignar un *savepoint* a la primera instrucción, ya que un aborto parcial para deshacer hasta la primera instrucción equivale a un *rollback* a toda la transacción.

| |
|---|
| ***begin*** |
| instrucción 1 |
| ***savepoint*** "sp0" |
| instrucción 2 |
| ***savepoint*** "sp1" |
| instrucción 3 |
| ***savepoint*** "sp2" |
| instrucción 4 |
| ***savepoint*** "sp3" |
| instrucción 4 |
| ***commit*** |

**Figura 4-9** Creación de *savepoints* por cada instrucción.

2. Cada transacción conserva un vector de sus respectivos *savepoints*. Para detectar malas dependencias, el monitor MTA lleva a cabo una inspección de los objetos que se están actualizando (sólo las actualizaciones porque las lecturas a objetos no generan malas dependencias aún cuando pertenezcan a diferentes transacciones). Cada objeto que es actualizado es almacenado en un vector de recursos denominado "`dataResources`", el cual es revisado por cada operación de escritura recibida, con el objetivo de detectar "malas dependencias".

3. El monitor MTA enviará un mensaje de inconsistencia a cada transacción si se detecta una mala dependencia si se cumplen las siguientes políticas:

    a) Si el objeto actualmente procesado existe en el vector de recursos bloqueados (`dataResources`) por cada transacción, entonces verificar si

    b) si todas las escrituras previas almacenadas en el vector de recursos bloqueados pertenecen a la transacción actual, no se lleva a cabo acción alguna, debido a que "dos escrituras al mismo objeto en la misma transacción no comprometen su consistencia", en caso contrario:

    c) se trata de una transacción diferente, por lo tanto se verifica la versión generada por la nueva escritura. Si las versiones son diferentes entonces se selecciona a la transacción como candidata al aborto.

El proceso anterior crea un vector con las transacciones que están afectadas por la mala dependencia, sin embargo, el monitor selecciona a aquellas transacciones candidatas al aborto total o parcial con un análisis de prioridades, que se describe a continuación.

- **Niveles de prioridad**

El monitor lleva a cabo un análisis de las prioridades de las transacciones involucradas, para decidir a cual de ellas enviar el mensaje de cancelación. Existen dos niveles de prioridad básicos: PR_CRITICAL indica que la transacción es crítica por lo tanto ante fallas se deben preservar sus acciones sobre las transacciones con nivel PR_OPTIONAL, las cuales ante la presencia de fallas son las candidatas inmediatas a la cancelación ya sea total o parcial. Ambos niveles pueden presentar una situación de haber finalizado su ejecución, por lo tanto un nivel PR_OPTIONAL_FINISHED tiene mayor prioridad que un nivel PR_OPTIONAL. La tabla 4 muestra la descripción de estos niveles de prioridad:

**Tabla 4** Niveles de prioridad.

| **Prioridad** | **Descripción** |
|---|---|
| PR_CRITICAL_FINISHED | Indica que es una transacción crítica, es decir sus operaciones y que además está lista para hacer *commit* (sólo en el caso de la modalidad de Transacciones Anidadas Cerradas). |

| | |
|---|---|
| PR_CRITICAL | Indica que la transacción es crítica pero aún no ha terminado sus operaciones. |
| PR_OPTIONAL_FINISHED | Indica que es una transacción es decir, las operaciones pueden ser abortadas en cualquier momento. Además está lista para hacer *commit*. |
| PR_OPTIONAL | Indica que es una transacción opcional, las operaciones pueden ser abortadas en cualquier momento y aún no ha finalizado sus operaciones. |

4. El monitor selecciona a las transacciones de la siguiente manera:
    a. Verificar si existe al menos una transacción con alta prioridad (ver tabla 3) para sólo enviar mensajes de aborto a aquellas que tengan prioridad baja.
    b. Eliminar todas las transacciones con alta prioridad
    c. Si existen mas de una transacción con la misma prioridad, entonces verificar por cantidad de escrituras realizadas
    d. Si la cantidad de escrituras entre ellas es la misma, entonces la transacción que tenga más tiempo no es seleccionada para la cancelación.

5. Una vez que el vector contiene sólo aquellas transacciones que deberán de abortar, un mensaje es enviado a cada transacción para llevar a cabo el aborto parcial.

### 4.2.2 Metadatos: identificación de objetos bloqueados

En el procesamiento de transacciones, se tiene acceso a diversos objetos para operaciones de lectura/escritura a través de instrucciones del lenguaje SQL tales como SELECT, UPDATE, INSERT o DELETE. Por lo tanto es importante identificar cada objeto procesado, esto es posible a través de la consulta de *metadatos.*

- **Lecturas**

Cuando se lleva a cabo una lectura de objetos con la instrucción SELECT, esta puede leer desde uno hasta varios objetos en una sola instrucción así como obtener desde cero, uno o varios registros, por lo que el monitor requiere de la identificación de cada uno de estos objetos.

Para poder identificar cada objeto accesado por una transacción, el monitor obtiene los metadatos de la base de datos utilizada al construir un objeto de la clase `ConnectionDB`. El código de la figura 4-10 ilustra como se obtienen los nombres de tablas y sus respectivas llaves primarias, las cuales son almacenadas en el `Vector` "tables".

```
DatabaseMetaData md = this.connection.getMetaData();

// obtener las Tablas
ResultSet rsTb = md.getTables(databaseName, "", "", null );

while(rsTb.next()) {
    // obtener las Primary Keys
    String tableName = rsTb.getString("TABLE_NAME");
    ResultSet rsPk = md.getPrimaryKeys(databaseName,null,tableName);
    if(rsPk.next()) {
        Table tb = new Table(tableName,
                rsPk.getString("COLUMN_NAME"));
        //agregar la nueva tabla con la info de su pk
        tables.add(tb);
    } // if...

} // while
```

**Figura 4-10** Obteniendo llaves primarias.

Cuando una transacción lleve a cabo una instrucción sql, por ejemplo, un SELECT, es importante asociar cada objeto leído con su identificador. El siguiente ejemplo ilustra el mecanismo para asociar el identificador de cada objeto leído.

1. Suponga que se tiene una tabla dept (ver tabla 5):

**Tabla 5** Descripción de la tabla dept.

| deptno | dname | loc | cantidad |
|---|---|---|---|
| 10 | Ventas | Mexico | 0 |
| 20 | Compras | Guadalajara | 0 |
| 30 | Almacén | Mexico | 0 |
| 40 | Gerencia | Monterrey | 0 |

2. la siguiente consulta obtendrá los datos mostrados en la tabla 6:

```
SELECT dname, loc
FROM dept
WHERE loc = 'Mexico';
```

**Tabla 6** Consulta a tabla dept.

| dname | loc |
|---|---|
| ventas | Mexico |
| almacen | Mexico |

3. Cada objeto carece de identificación ya que la consulta no incluyó la llave primaria (deptno) de la tabla dept. Por lo tanto, a la consulta se agrega la columna (**`table_pk`**) con los valores de identificación para cada objeto tomados de su respectiva llave primaria. Así el monitor será capaz de detectar malas dependencias e identificarlas apropiadamente (ver tabla 7).

**Tabla 7** Identificación de objetos.

| table_pk | dname | loc |
|---|---|---|
| 10 | ventas | Mexico |
| 30 | almacen | Mexico |

4. ahora se tienen los objetos leídos **`dname_10`**, **`dname_30, loc_10`** y **`loc_30`**. El monitor es capaz de identificar un objeto leído de manera única, sin importar cuantos renglones son leídos en una misma columna. Si la consulta incluye la llave primaria, de cualquier manera se agrega la columna **`table_pk`**, como en el ejemplo de la tabla 8.

```
SELECT *
FROM dept
WHERE loc = 'Mexico';
```

**Tabla 8** Incluyendo la llave primaria.

| table_pk | deptno | dname | loc | cantidad |
|---|---|---|---|---|
| 10 | 10 | ventas | Mexico | 0 |
| 30 | 30 | almacen | Mexico | 0 |

El código de la figura 4-11 ilustra la asignación del nombre de identificación a cada columna, antes de ser enviada al monitor:

```
while(rs.next()) {
    String rowid = "";
    for(int i = 1; i <= numberOfColumns; i++) {
        String fieldName = rsmd.getColumnName(i);
        String fieldValue = rs.getString(fieldName);
        if(fieldName.equalsIgnoreCase("table_pk")) {
            // agregar a la columna el identificador unico de renglon
            rowid = fieldValue;
        } else {
            fieldName += "_" + rowId;
            sendMessage("<read object><tid="+getTid()
            +"><"+fieldName+"="+fieldValue+">"
                    +"<sp=" + getLastSavePoint()
                    + "></read object>");
        } // if .. else
    } // for por cada campo
} // while, por cada registro.
```

**Figura 4-11** Creando identificadores por objeto.

- **Escrituras**

Para las operaciones de escritura tales como **`UPDATE`**, existe una tabla para almacenar los objetos modificados con el nombre <**`tablename_updated_objects`**>, donde **`tablename`** indica la tabla donde se llevó a cabo la actualización (para **`dept:`** "**`dept_updated_objects`**"). Para esto se define un *trigger* (disparador) denominado "*dept_au*", donde "**`dept`**" es el nombre de la tabla donde se definió el *trigger*. "a" especifica el tiempo de ejecución del trigger pudiendo ser "a" de *after* (después de) o "b" de *before*

"antes de"[6]. La sigla "u" especifica la operación que activará el trigger, pudiendo ser "u" de *update* (actualización), "i" de *insert* (inserción) ó "d" de *delete* (borrado). Por ejemplo, para consultar la tabla de objetos actualizados en dept, la estructura sería la que se ilustra en la figura 4-12.

```
CREATE TABLE dept_updated_objects (
 tid varchar(25),
 statement_id int,
 status int,
 deptno int,
 dname varchar(20),
 loc varchar(20),
 cantidad int
);
```

**Figura 4-12** Estructura de la tabla dept_updated_objects.

La tabla "**`dept_updated_objects`**" tiene la misma estructura que la tabla original "**`dept`**" más los siguientes atributos (ver tabla 9):

**Tabla 9** Descripción de atributos agregados a la tabla "dept_updated_objects".

| Columna | Descripción |
|---|---|
| `tid` **`varchar`**`(25)` | Identifica la transacción que llevó a cabo la actualización. |
| `statement_id` **`int`** | El identificador de la operación realizada. |
| `status` **`int`** | Indica el status del valor, es decir si es el valor antes de la modificación (0) o si representa el valor después de la modificación (1). |

Con estos datos es posible identificar los renglones modificados por cada transacción concurrente. Por lo tanto la sentencia sql original invocada por el usuario (ver figura 4-13) es modificada para agregarle el identificador de la transacción y el de la operación realizada (ver figura 4-14).

```
Transaction t = new Transaction();

t.doUpdate(" UPDATE dept "+
           " SET dname = 'valor' " +
           "   , loc = 'valor' " +
           " WHERE deptno = 10;" );
```

**Figura 4-13** Instrucción antes de su re-definición.

[6] Para MySQL es posible definir un trigger que se ejecute antes (*before*) de la operación asociada. El trigger "dept_au" almacena los nuevos valores modificados obteniendolos de una tabla de mysql denominada "*NEW*" esta tabla contiene los objetos modificados, para Sql Server es la tabla *INSERTED*.

```
Transaction t = new Transaction();

t.doUpdate(" UPDATE dept "+
           " SET tid = '" + this.getTid() + "' " +
           "   , statement_id = " + this.getIdSqlStatement() +
           "' " +
           "   , dname = 'valor' " +
           "   , loc = 'valor' " +
           " WHERE deptno = 10;" );
```

**Figura 4-14** Instrucción sql después de su re-asignación.

## 4.4 Transacciones Anidadas Abiertas

El modelo de Transacciones Anidadas Abiertas (TAA) descrito en la sección 2.2.4, incorpora una mejora en el rendimiento de las transacciones, debido a que permite a estas confirmar el trabajo realizado, aún cuando la transacción raíz no haya decidido confirmar o validar. Esto permite la liberación de los bloqueos adquiridos por las sub-transacciones, así otras transacciones disminuirán sus tiempos de espera para la adquisición de bloqueos. Sin embargo, si la transacción raíz decide abortar, entonces se utiliza el mecanismo de *Transacciones de Compensación* para deshacer los efectos de las transacciones que hayan confirmado.

### 4.4.1 Efectos en la confirmación y cancelación

Si dos transacciones Ti y Tj crean dependencias entre ellas y ambas alcanzan exitosamente el fin de la transacción, no se presentan escenarios con malas dependencias. Pero no es la misma situación si por ejemplo, Ti termina exitosamente con versiones generadas por Tj, y Tj presenta fallas y requiere abortar. A continuación se describe este escenario, el cual puede comprometer la consistencia de los datos.

### 4.4.2 Aborto posterior a confirmación

La siguiente historia <Tj,w,x><Ti,r,x><Ti,c><Tj,a>, ilustra un escenario donde la confirmación de la transacción Ti que es dependiente de Tj, finaliza exitosamente con versiones generadas por Tj y ésta requiere de la cancelación del trabajo realizado.

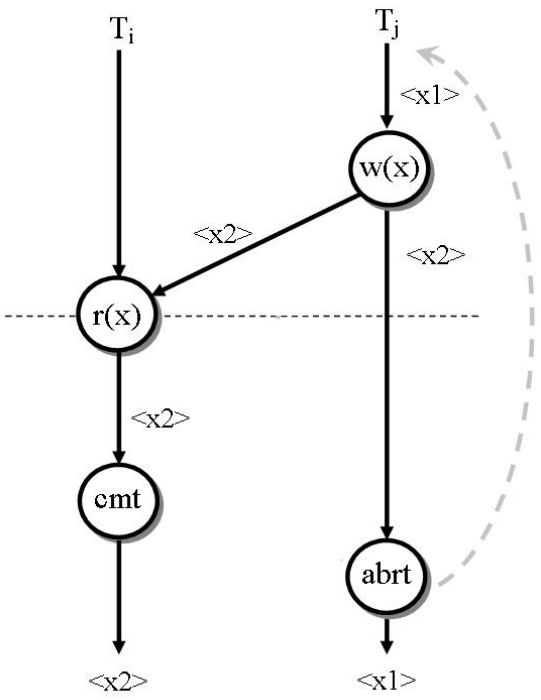



**Figura 4-15** Compensación de Ti después del aborto de Tj.

En la figura 4-15 se ilustra este escenario. Ti y Tj son dos transacciones concurrentes que comparten el objeto *x*. Tj obtiene acceso sobre el objeto *x* cuya versión inicial es <x1>, y lleva a cabo una escritura generando una nueva versión de *x*:<x2>. Posteriormente Ti lleva a cabo una lectura del objeto *x*, obteniendo la versión hasta ese momento correcta <x2>. A continuación Ti termina sus operaciones con el valor del objeto *x* y realiza la confirmación de Ti exitosamente. Sin embargo, la versión utilizada por Ti fue creada por la escritura de Tj, entonces si Tj decide abortar, el valor leído por Ti se convierte en una mala dependencia. Cabe hacer notar, que el aborto de Tj devuelve la versión <x1> al objeto *x*. La figura 4-16 ilustra este escenario, Tj tiene que abortar por cualquier falla que haya presentado, la cancelación hace que la versión <x2> creada en la escritura de Tj, sea revertida y el objeto *x* regrese a su estado original antes de iniciada la transacción Tj (versión <x1>). Por otro lado, la transacción Ti utilizó con éxito la versión creada por Tj antes de su cancelación, resultando en la mala dependencia conocida como *dirty-reads.*

Como la transacción Ti ha finalizado con un commit y liberado sus bloqueos, no puede ser cancelada debido a la propiedad *durabilidad* [14]. Para deshacer los efectos llevados a cabo por la transacción Ti se puede utilizar el mecanismo de ***compensación***.

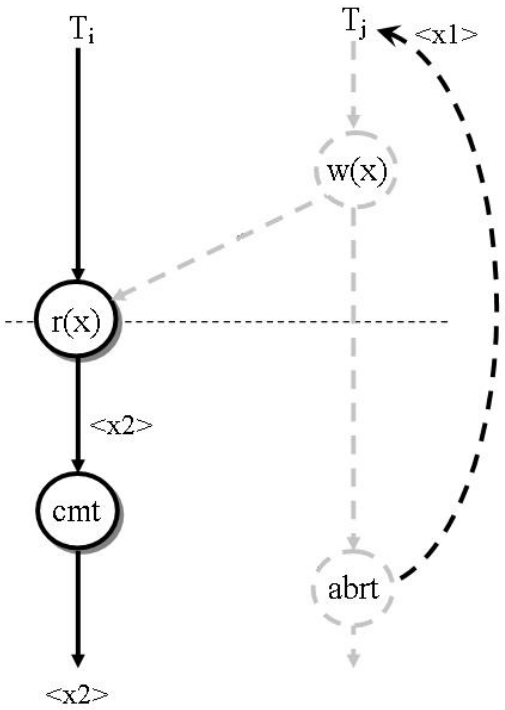



**Figura 4-16** Aborto de Tj generando malas dependencias.

## 4.5 Algoritmo para administrar Transacción de Compensación

Una transacción de compensación (TC), tiene la función de eliminar los efectos permanentes dejados por transacciones confirmadas con éxito [14]. Es decir, una TC es capaz de regresar datos al estado original que existía justo cuando se inició la transacción que se va a compensar. Se dice que una TC lleva a cabo un rollback semántico. Esto se logra obteniendo la información a deshacer de archivo que lleva el registro las transacciones llamado *log*. Por ejemplo, si una transacción lleva a cabo las siguientes operaciones:

*Begin transaction T1*
*Leer saldo (saldo = 2000)*
*Saldo = saldo – 1000;*
*Escribir saldo*
*Commit T1*

Si la transacción T1 que ha sido confirmada con éxito, forma parte de un grupo de transacciones que debe de cancelarse, entonces una TC llevaría a cabo las siguientes acciones:

*Begin transaction TC1*

*Leer saldo (saldo = 1000)*

*obtener información del* ***log***

*saldo = saldo + 1000;*

*Escribir saldo*

*Commit TC1*

De esta forma, se consigue llegar a un estado consistente aún cuando una transacción haya finalizado exitosamente y los cambios realizados deban ser cancelados, según la propiedad de *durabilidad* [14].

En el ejemplo ilustrado en la figura 4-16, Ti y Tj son dos transacciones que crean dependencias debido a que Ti lee el valor del objeto *x* que previamente ha modificado Tj. Cabe hacer notar que hasta ese momento, no existen malas dependencias debido a que Ti, ha obtenido la última versión <x2> que Tj ha generado sobre el objeto *x*. El problema surge cuando Ti finaliza exitosamente antes de Tj cancele el trabajo, como se mencionó anteriormente. Un mecanismo para regresar a un estado consistente al objeto *x* es mediante una TC. Sin embargo, una TC revertirá incluso aquellas operaciones consistentes realizadas antes de la lectura, donde no se dependencia de un valor consistente de *x*. Para evitar esta situación, la TC debería cancelar sólo aquellas operaciones que fueron llevadas a cabo después de la lectura al objeto *x*. De esta forma, no será necesario compensar todas las operaciones en Ti. A este mecanismo le denominamos ***Compensación Parcial***, y es implementado con el mecanismo de savepoints. Así, es posible preservar más trabajo realizado con éxito, en lugar de cancelarlo totalmente.

### 4.5.1 Compensación parcial

La propiedad durabilidad de las transacciones permite a los datos permanecer en un estado persistente, y no pueden ser revertidos más que por otra transacción. El mecanismo que una TC lleva a cabo, es revertir semánticamente todas las operaciones llevadas a cabo por una transacción que ha finalizado exitosamente. Sin embargo en el escenario de la figura 4-17 se puede observar que no todas las operaciones deberían ser revertidas por la TC, sino sólo aquellas que se convirtieron en malas dependencias.

Para poder llevar a cabo esta compensación parcial, es necesario que la transacción utilice el mecanismo de *savepoints*. Así, con cada SP, se estable un punto en la transacción hasta donde puede llevarse a cabo ya sea un rollback parcial o una compensación parcial. La figura 4-17 muestra un ejemplo de una compensación parcial. En la figura, Ti es una transacción que termina con éxito haciendo *commit*. Sin embargo, por fallas en el sistema es necesario que la transacción sea cancelada. La única manera de llevar a cabo esta situación es realizando una TC (ver figura 4-17a), hasta el punto inicial de Ti.

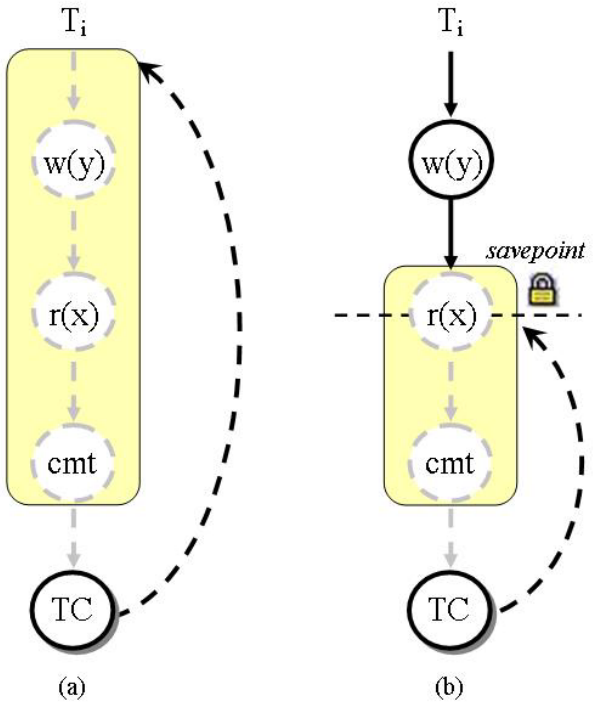



**Figura 4-17** Compensación parcial.

Como se dijo anteriormente, las operaciones llevadas a cabo antes de la lectura r(x) son consistentes hasta que la transacción Tj abortara (ver figura 4-16). Por lo tanto, se puede realizar una transacción de compensación parcial, hasta el punto donde las operaciones son consistentes. En la figura 4.17b se ilustra este escenario. La TC revierte las operaciones que se convirtieron en inconsistentes en Ti después de la cancelación de Tj (ver figura 4-16). Para poder realizar esta cancelación parcial, se requiere de:

1. El conocimiento de las operaciones realizadas por la transacción a compensar. Esta información es tomada del archivo *log*.

2. Crear un grafo de dependencias (manipulado por el monitor de las transacciones) entre las transacciones que comparten los mismos objetos.

3. La utilización de SPs en las operaciones que sean propicias a generar malas dependencias.

4. Mantener los SPs, aún cuando la transacción haya finalizado con éxito. Si y sólo si, las transacciones con las que creó dependencias aún no hayan terminado.

5. Finalmente, rehacer sólo aquellas operaciones donde se presenta la mala dependencia.

De esta forma, las transacciones que hayan efectuado una cantidad considerable de operaciones, hasta el momento de que surja una mala dependencia, pueden ser preservadas con los mecanismos propuestos de *abortos* y *compensaciones parciales*.

### 4.5.2 Compensación parcial para Transacciones de solo-lectura

Para las transacciones de solo-lectura, el mecanismo de *abortos* y *compensaciones parciales* puede relajarse, debido a que las operaciones de lectura no generan nuevas versiones. Por lo tanto, si dos transacciones comparten el mismo objeto en operaciones de lectura, no se generan malas dependencias. De tal forma que, no es necesario controlar las dependencias entre ambas transacciones.

### 4.5.3 Manejadores sin soporte de savepoints

En esta sección se plantea una alternativa para el uso del mecanismo de transacciones anidadas como modelo enfocado a proveer un mejor mecanismo tolerante a fallas para aquellos manejadores de bases de datos que no soporten el uso de *savepoints*. Por lo tanto, las transacciones ejecutadas y monitoreadas bajo el modelo TAM, serán capaces de crear árboles jerárquicos de transacciones pero no tendrán la capacidad de llevar a cabo *abortos parciales.*

- **Manejadores sin savepoints**

Existen actualmente en el mercado manejadores de bases de datos relaciones, la gran mayoría de ellos soportan el uso de *savepoints* (SP). Esta funcionalidad hace posible demarcar en unidades lógicas a una transacción [14]. El mecanismo de abortos parciales descrito en la sección 2.2.2, está basado en el uso de SPs. El mecanismo de SPs es un modelo estándar implementado por diversos DBMS comerciales. Sin embargo, este modelo no está implementado en algunos DBMSs tales como: *PostgreSql* hasta la versión 7.1 para la plataforma Linux e *Interbase* de Borland hasta la versión 4. Por lo tanto, una alternativa de solución es utilizar transacciones anidadas. Esta alternativa es descrita a continuación.

- **Transacciones Anidadas para abortos parciales**

Para obtener el mecanismo de abortos parciales que se consigue con el uso de SPs en DBMS que no posean ese mecanismo, una alternativa es utilizar el esquema de división en sub-transacciones que permite el modelo de TA. Las TAs son una generalización de los *savepoints,* debido a que permiten marcar piezas lógicas de trabajo en una transacción y en caso de fallas, es posible deshacer piezas individuales sin tener que cancelar todo el trabajo realizado.

## 4.6 Control de Concurrencia Extendido

### 4.6.1 Dependencia Read-Write

La dependencia *read-write* define la relación que existe entre dos transacciones que comparten un mismo recurso. Donde una transacción Ti obtiene acceso a un objeto x para lectura y posteriormente una transacción Tj, obtiene acceso al mismo objeto x para escritura. En este escenario, la mala dependencia *lectura-sucia* puede ser generada. Esto debido a que la transacción Ti lee la versión <x1> de un objeto que posteriormente Tj sobre escribe con una nueva versión <x2>.

Esta mala dependencia puede evitarse utilizando un nivel de aislamiento tal como: *read committed, repeatable read o serializable.* Sin embargo, estos niveles de aislamiento ocasionarán el retardo de las operaciones en conflicto (en este caso la escritura de Tj) al bloquear el objeto x hasta que éste sea liberado por Ti ya sea por *commit* o *abort.*

Para conseguir un mejor rendimiento en la ejecución de las transacciones concurrentes, se propone relajar el nivel de aislamiento de las transacciones involucradas al nivel: *read uncommitted.* Con este nivel pueden suceder las tres malas dependencias, pero la ejecución de ambas transacciones es controlada por un monitor de transacciones que registra las operaciones hechas por cada transacción, para efectos de poder controlar y en todo caso revertir los efectos causados por la presencia de las malas dependencias. La figura 4-18 ilustra un algoritmo para revertir los efectos causados por una lectura sucia.

```
sp_i(1)
r_i(x) := <x1>
registrar la versión <x1> en el log
sp_j(1)
w_j(x) := <x2>
registrar la versión <x1> en el log
comparar las versiones de ambas operaciones
si <x1> != <x2>
     enviar mensaje a T_j de rollback a sp_j(1)
     reintentar T_j
sino
      las versiones son iguales
      continuar
end si
```

**Figura 4-18** Algoritmo para deshacer una lectura-sucia.

La función $sp_k(n)$ establece una marca *savepoint* etiquetada con el valor *n* por la transacción *i*. En el código se establecen dos *savepoints*, por las transacciones Ti y Tj respectivamente. Si las versiones del objeto *x* generadas por ambas transacciones son diferentes, entonces el monitor de transacciones selecciona a la transacción "culpable" como la candidata a llevar a cabo un aborto parcial. En el ejemplo, la transacción Tj debe

realizar un aborto parcial hasta el *savepoint* definido por la función `sp_j(1)`. La figura 4-19 ilustra este escenario.

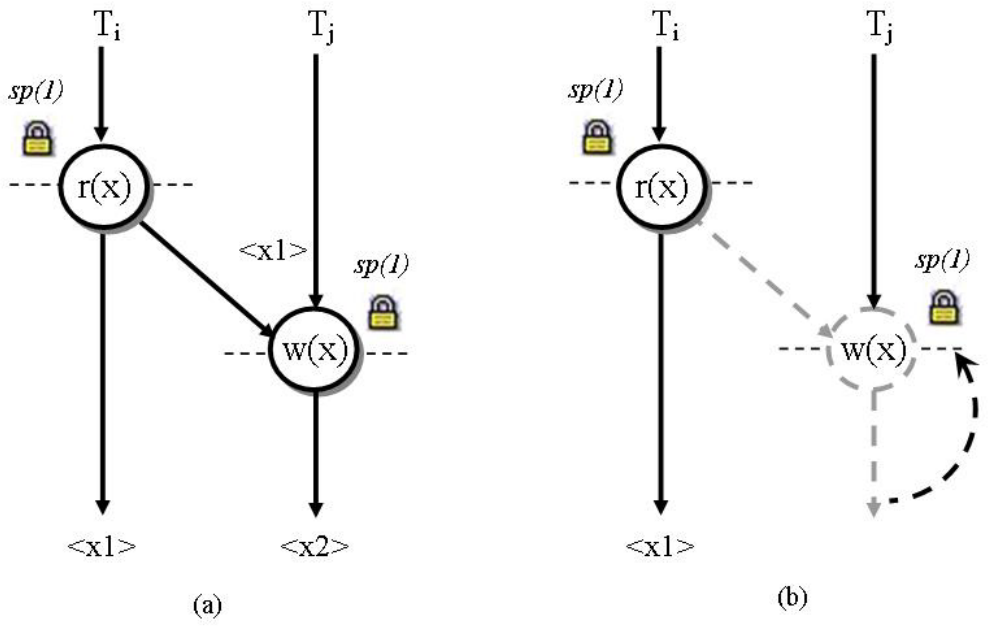



**Figura 4-19** Evitando lecturas sucias.

En la figura 4-19a, se ilustra el escenario donde se crea la mala dependencia lectura sucia. Las versiones creadas en ambas transacciones son diferentes, por lo tanto se requiere abortar la escritura realizada por la transacción Tj. En la figura 4-19b se puede apreciar un aborto parcial hasta el *savepoint 1* dentro de la transacción Tj. Este aborto cancela la versión <x2> generada por la escritura realizada en Tj.

### 4.6.2 Dependencia Write-Read

La dependencia *write-read*, define la relación que existe entre dos transacciones concurrentes que compiten por el mismo recurso. En este escenario, una transacción Ti realiza una escritura a un objeto *x* generando una versión nueva del objeto. Posteriormente una transacción Tj lleva a cabo una lectura al mismo objeto obteniendo la última versión creada por Tj.

Esta dependencia no es considerada anómala, pero puede ocasionar malas dependencias en los siguientes casos:

1. Si Tj aborta, entonces la lectura de Ti se convierte en una *lectura sucia.*

2. Si Tj aborta y Ti relee el objeto *x,* se obtiene una *lectura no repetible.*

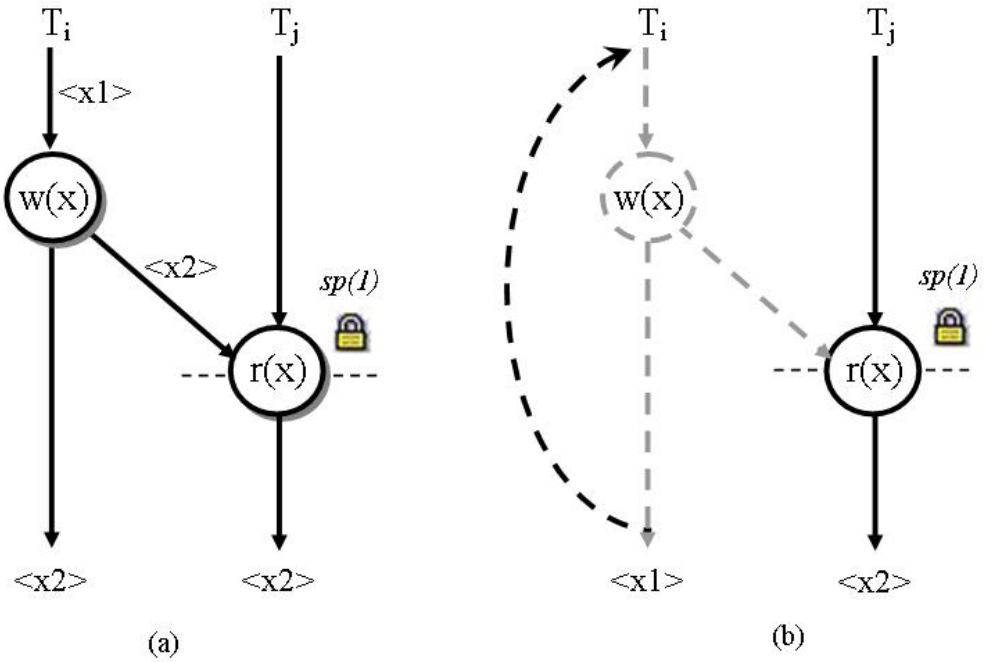



**Figura 4-20** Problemas en la dependencia write-read.

En la figura 4-20a se ilustra la dependencia *write-read*. Hasta ese momento, las versiones leídas por ambas transacciones son consistentes.

Esto es debido a que la transacción Tj ha leído la última versión generada por la transacción Ti. Sin embargo, si Ti debe abortar por alguna falla, regresará a la versión original de *x* antes de la escritura. La figura 4-20b muestra este escenario, las versiones de ambas transacciones son inconsistentes, por lo tanto Tj debe llevar a cabo un aborto parcial hasta el *savepoint 1* y reintentar la lectura al objeto x para obtener una versión consistente del objeto *x* (ver figura 4-21).

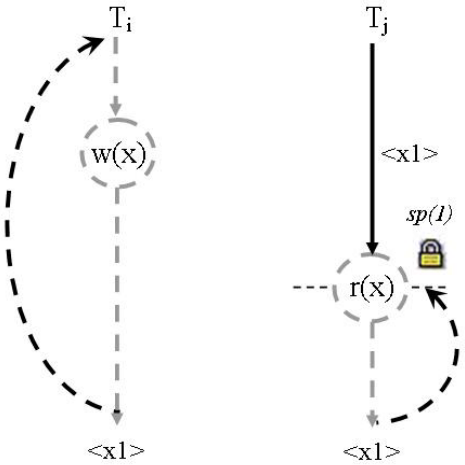



**Figura 4-21** Evitando los efectos de aborto en dependencias write-read.

# Capítulo 5 Pruebas y resultados

En este capítulo se presentan las pruebas realizadas y los resultados obtenidos del desempeño del MTA con el modelo de control de concurrencia extendido para transacciones anidadas cerradas/abiertas.

## 5.1 Diseño de las pruebas

A continuación se describe el formato de las pruebas a realizar. Se cuentan con dos escenarios de ejecución de transacciones concurrentes:

1. a través de la API-JDBC de Java, y
2. bajo el contexto del **Monitor de Transacciones Anidadas (MTA)** utilizando el modelo de transacciones anidadas abiertas (*Opened Nested Transactions*: ONT)**.**

### 5.1.1 Transacciones concurrentes con la API JDBC de JAVA

Bajo este contexto, se ejecutarán programas escritos en Java Standar Edition 5, utilizando la API JDBC. Cada transacción se conecta a un servidor de bases datos (en este caso Microsoft Sql Server) utilizando el nivel de aislamiento por defecto *READ UNCOMMITTED*. Con este nivel de aislamiento las transacciones concurrentes son capaces de leer sólo valores que ya han sido confirmados por otras transacciones. Este nivel de aislamiento garantiza la ausencia de "**lecturas sucias**", pero reduce el rendimiento de las aplicaciones debido al bloqueo de los objetos en proceso, y haciendo posible la presencia de "**abrazo mortal**".

### 5.1.2 Transacciones concurrentes utilizando MTA

En este contexto, se ejecutan aplicaciones escritas en Java las cuales se conectan a un Monitor de Transacciones Anidadas (MTA), utilizando el modelo de Transacciones Anidadas Cerradas/Abiertas (TAC/TAA). En este caso se utiliza el modelo de TAA, debido a que estas hacen commit una vez que han terminado con éxito, sin esperar a que la transacción padre decida hacer commit, en el caso de que decida hacer rollback, las transacciones terminadas con éxitos (durables) tiene que efectuar un proceso de compensación. Así las transacciones en este modelo liberan los recursos una vez que han terminado con éxito. Las transacciones concurrentes relajan el nivel de aislamiento al menos restrictivo READ UNCOMMITTED. Bajo este nivel pueden suceder malas dependencias (lecturas sucias, lectura no repetibles, y fantasmas). Para resolver los efectos de inconsistencias se llevan a cabo abortos parciales (ver Capítulo 4). De esta manera se consigue tener más transacciones terminadas con éxito.

Se llevarán a cabo el procesamiento de un grupo de transacciones (Ts) en este ejemplo se describe una historia compuesta de 5 transacciones con las siguientes instrucciones:

| T1 | T2 | T3 | T4 | T5 |
|---|---|---|---|---|
| <T1, bt, null><br><T1, r, loc><br><T1, ct, null> | <T2, bt, null><br><T2, w,LOC><br><T2, w,LOC><br><T2, ct, null> | <T3, bt, null><br><T3, r,loc><br><T3, ct, null> | <T4, bt, null><br><T4, w,LOC><br><T4, ct, null> | <T5, bt, null><br><T5, w,LOC><br><T5, ct, null> |

Donde <t, op, ob> define una triupla constituida por: (t) que define a la transacción actual, (op) la operación que se lleva a cabo y (ob) el objeto que se está procesando.

- bt : begin transaction
- ct: commit
- rt: rollback
- r: lectura
- w: escritura

La ejecución de este grupo de transacciones, será lanzada usando tres escenarios diferentes:

**Ejecución 1.** Ejecutar las transacciones utilizando el nivel de aislamiento por defecto del DBMS (READ COMMITTED), donde cada transacción sólo puede acceder a un recurso que haya sido previamente confirmado.

**Ejecución 2.** Ejecutar las Ts utilizando el MTA con el nivel de aislamiento READ COMMITTED.

**Ejecución 3.** Ejecutar las Ts utilizando el MTA con el nivel de aislamiento READ UNCOMMITTED.

## 5.2 Resultados

En la ejecución no. 1, se espera que las transacciones presenten tiempos de espera para acceder a aquellos objetos que han sido bloqueados y no han sido confirmados. La figura 5-1 muestra un grafo de dependencias donde las operaciones bloqueadas son serializadas para evitar malas dependencias.

```
<T5RC, bt, null>
  <T5RC,w,LOC>
 <T3RC,bt, null>
 <T4RC,bt, null>
 <T1RC,bt, null>
 <T2RC,bt, null>
 <T5RC,ct, null>
  <T1RC,r,loc>
 <T1RC,ct, null>
  <T4RC,w,LOC>
 <T4RC,ct, null>
  <T3RC,r,loc>
 <T3RC,ct, null>
  <T2RC,w,LOC>
  <T2RC,w,LOC>
 <T2RC,ct, null>
```

**Figura 5-1** Serialización de la historia.

En la ejecución no. 2, se esperan los mismos resultados en la ejecución no. 1, con la excepción de que ahora se obtiene un monitoreo de las transacciones, como se muestra en la figura 5-2.

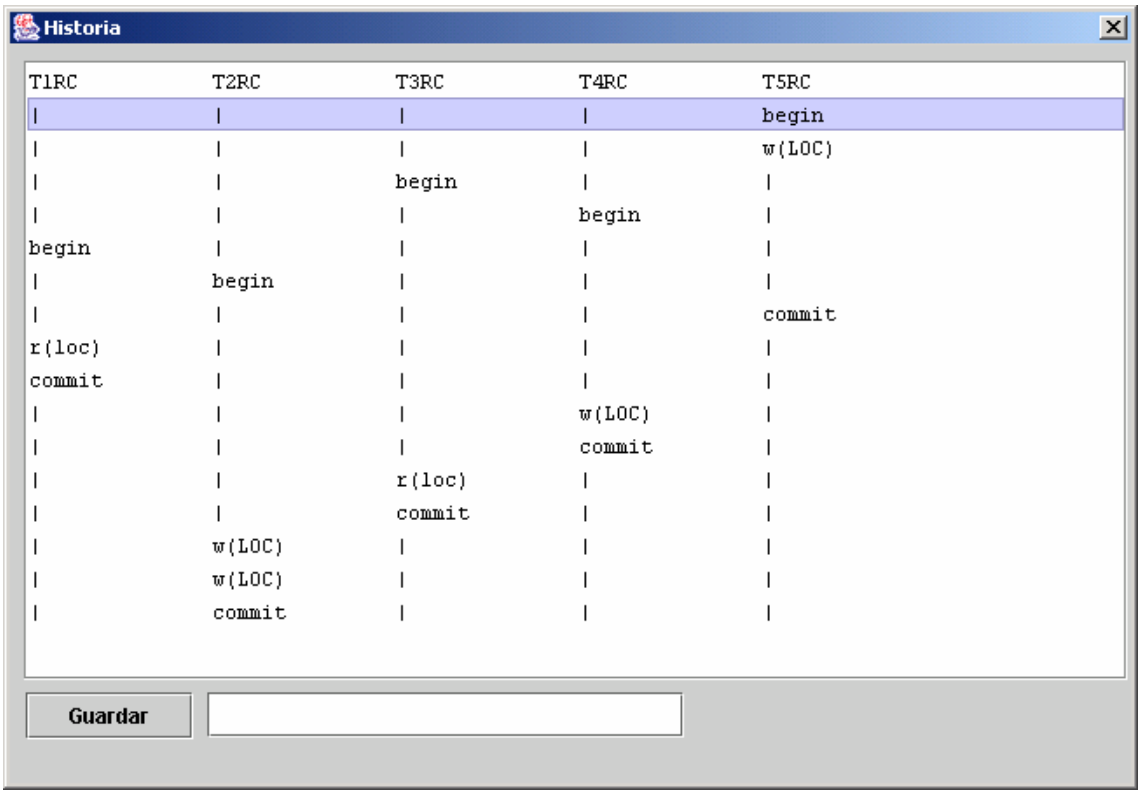



**Figura 5-2** Monitoreo de las Ts con el nivel de aislamiento READ COMMITTED.

En la ejecución 3, se relaja el nivel de aislamiento en las Ts a READ UNCOMMITTED, con el objetivo de mejorar el rendimiento de la aplicación, en el caso de que sucedan malas dependencias entonces se lleva a cabo una cancelación de las Ts en

conflicto. La figura 5-3 muestra un grafo de dependencias con una ejecución donde se obtienen los mismos resultados que en los casos de las figuras 1 y 2. El área sombreada ilustra a las transacciones T1 y T3 que pudieron acceder a un objeto bloqueado por T5 si tener que esperar hasta que T5 haya confirmado, mejorando el rendimiento de estas Ts.

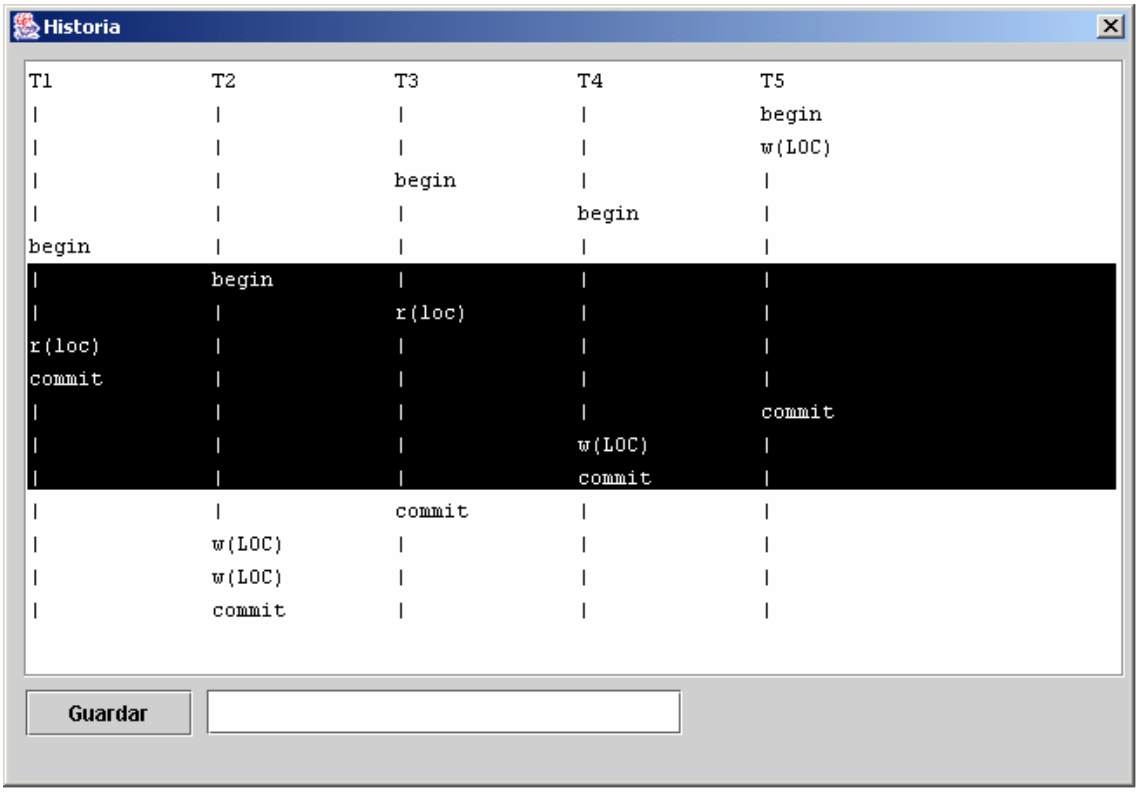



**Figura 5-3** Transacciones T1 y T3 acceden a un objeto bloqueado por T5.

### 5.2.1 Pruebas con 50 transacciones

**1. Utilizando JDBC.** En este tipo de pruebas, se utiliza el API JDBC estándar de Java para llevar a cabo la ejecución de las transacciones.

- **Transacciones Concurrentes**: 50
- **Número de operaciones por transacción**: 10 (5 escrituras 5 lecturas)
- **Nivel de aislamiento**: **READ COMMITTED** (default) ya que con este nivel se garantiza la consistencia de las lecturas.

**2. Utilizando el MTA.** Se utiliza para monitorear las operaciones críticas entre las transacciones y resolver las malas dependencias.

- **Transacciones Concurrentes**: 50
- **Número de operaciones por transacción**: 10 (5 escrituras 5 lecturas)

- **Nivel de aislamiento**: ***READ UNCOMMITTED***, si se presentan malas dependencias el MTA las resuelve enviando mensajes de aborto o aborto parcial a las transacciones afectadas, garantizando la consistencia de los datos.
- **Modelo**: *OPEN NESTED TRANSACTIONS* (ONT)

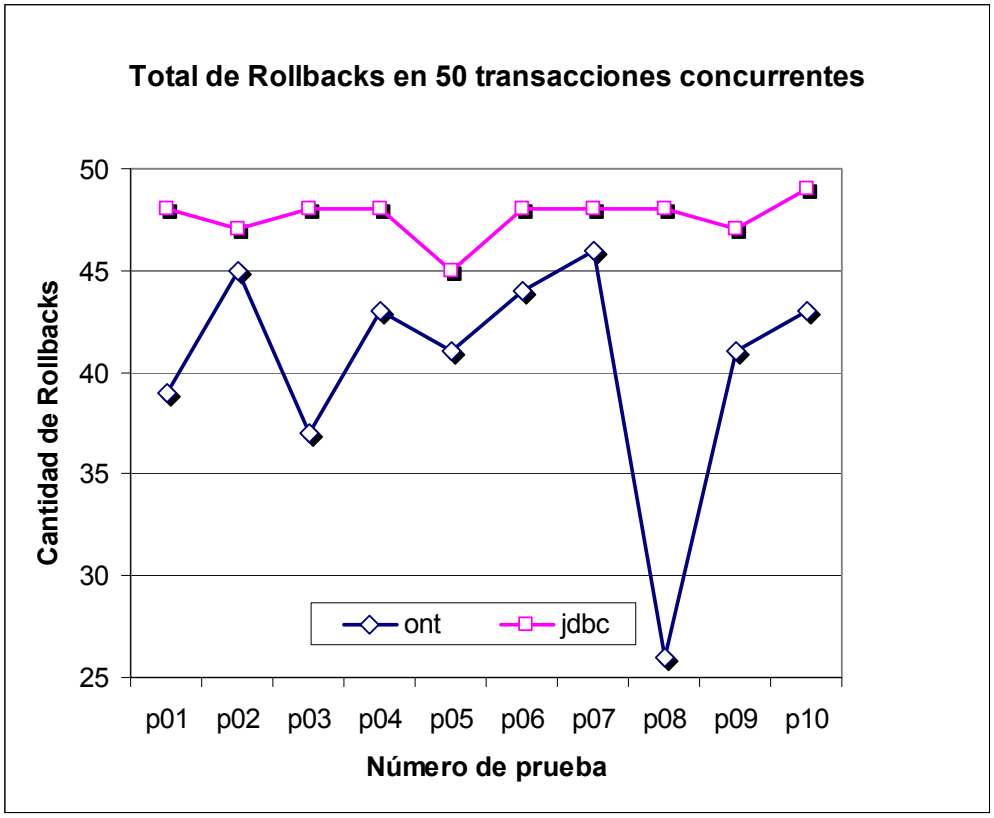



**Figura 5-4** Cantidad de cancelaciones ejecutando 50 transacciones concurrentes.

En la figura 5-4 se ilustran los resultados de las pruebas realizadas en ambos escenarios (JDBC y ONT), las gráficas ilustran la cantidad de transacciones canceladas. Utilizando el puente estándar de Java para conexiones a bases de datos JDBC se obtiene un promedio del 95% de cancelaciones. Utilizando el modelo ONT con un nivel de aislamiento relajado se obtiene un promedio de 81% cancelaciones, lo cual representa una mejora del 14% en transacciones terminadas con éxito.

### 5.2.2 Pruebas con 100 transacciones

1. Utilizando JDBC

- **Transacciones Concurrentes**: 100
- **Número de operaciones por transacción**: 10 (5 escrituras 5 lecturas)
- **Nivel de aislamiento**: **READ COMMITTED** (default) ya que con este nivel se garantiza la consistencia de las lecturas.

2. Utilizando el MTA

- **Transacciones Concurrentes**: 100
- **Número de operaciones por transacción**: 10 (5 escrituras 5 lecturas)
- **Nivel de aislamiento**: **READ UNCOMMITTED**, si se presentan malas dependencias el MTA las resuelve enviando mensajes de aborto o aborto parcial a las transacciones afectadas, garantizando la consistencia de los datos.
- **Modelo**: OPEN NESTED TRANSACTIONS (ONT)

Resultado de las pruebas medidas en función a la cantidad de abortos:

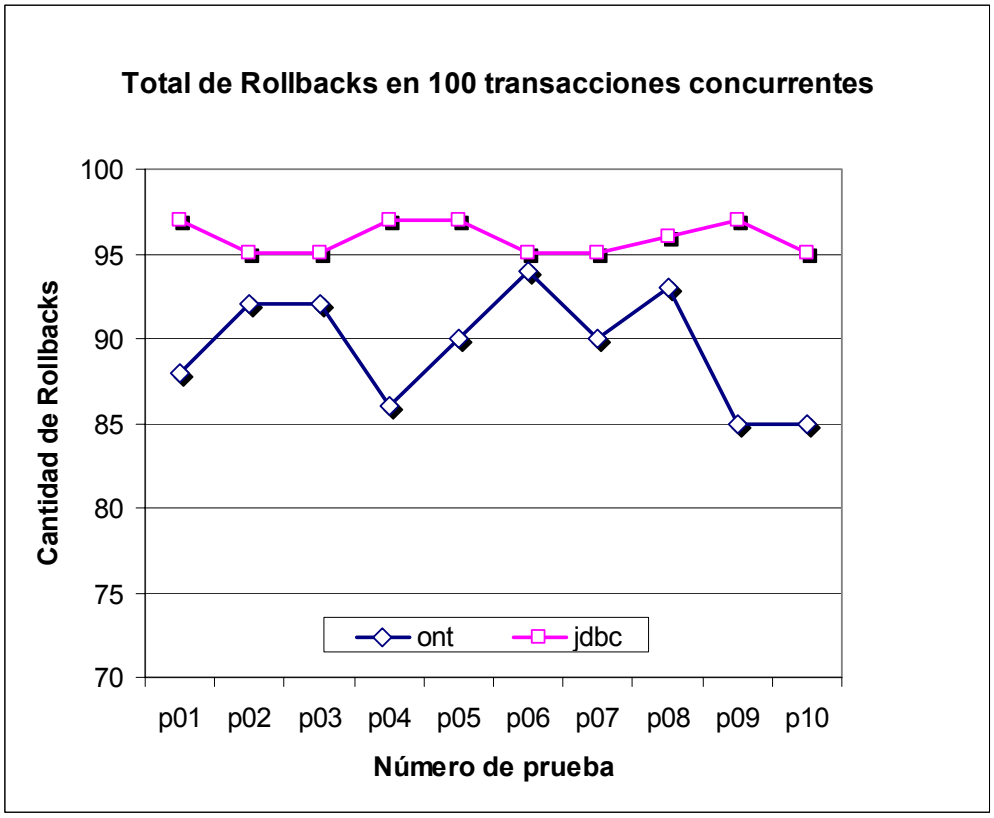



**Figura 5-5** Cantidad de cancelaciones ejecutando 100 transacciones concurrentes.

La figura 5-5 ilustra la cantidad de transacciones canceladas en ambos escenarios (JDBC y ONT) con un total de 100 transacciones concurrentes ejecutando 10 operaciones de lectura y escritura cada una. El promedio de cancelaciones bajo el modelo ONT es del 90%, mientras que utilizando JDBC es del 96%, lo cual representa una mejora del 6% utilizando el modelo ONT a través del MTA.

### 5.2.3 Pruebas con 500 transacciones

1. Utilizando JDBC

- **Transacciones Concurrentes**: 500
- **Número de operaciones por transacción**: 10 (5 escrituras 5 lecturas)

- **Nivel de aislamiento**: **READ COMMITTED** (default) ya que con este nivel se garantiza la consistencia de las lecturas.

2. Utilizando el MTA

- **Transacciones Concurrentes**: 500
- **Número de operaciones por transacción**: 10 (5 escrituras 5 lecturas)
- **Nivel de aislamiento**: **READ UNCOMMITTED**, si se presentan malas dependencias el MTA las resuelve enviando mensajes de aborto o aborto parcial a las transacciones afectadas, garantizando la consistencia de los datos.
- **Modelo**: OPEN NESTED TRANSACTIONS (ONT)

Resultado de las pruebas medidas en función a la cantidad de abortos:

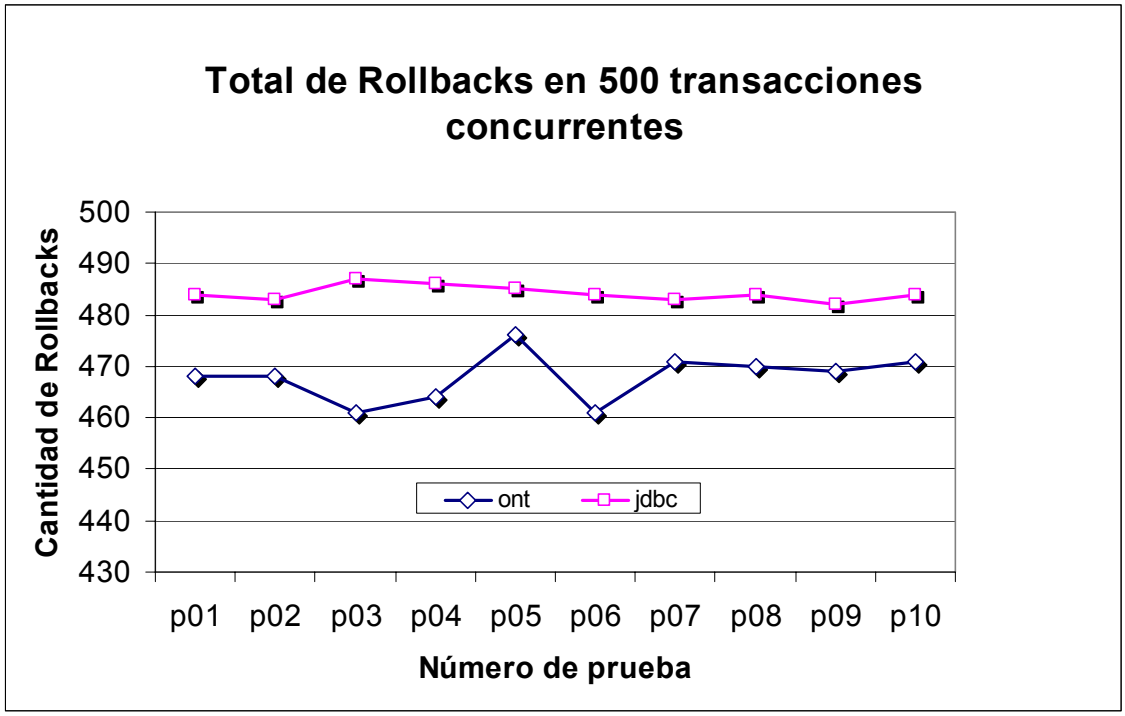



**Figura 5-6** Cantidad de cancelaciones ejecutando 500 transacciones concurrentes.

La figura 5-6 ilustra la cantidad de transacciones canceladas en ambos contextos (JDBC–ONT) con un total de 500 transacciones concurrentes ejecutando 10 operaciones de lectura y escritura cada una. El promedio de cancelaciones bajo el modelo ONT es del 93.58%, mientras que utilizando JDBC es del 96.84%, lo cual representa una mejora del 3.26% del modelo ONT contra JDBC.

### 5.2.4 Pruebas con 500 transacciones con procesador dual

Se desarrollaron las mismas pruebas descritas en la sección 5.2.3 pero en un sistema con dos procesadores (multi-procesador), obteniendo mejoras en los rendimientos de las transacciones concurrentes como se muestra en la figura 5-7.

El promedio de cancelaciones con el modelo ONT es de 431.10 (86.22%) contra 471.30 (94.26%) del obtenido utilizando JDBC. Esto representa una mejora en el rendimiento de las transacciones concurrentes del 8.04% contra el 3.26% obtenido en un sistemas uni-procesador.

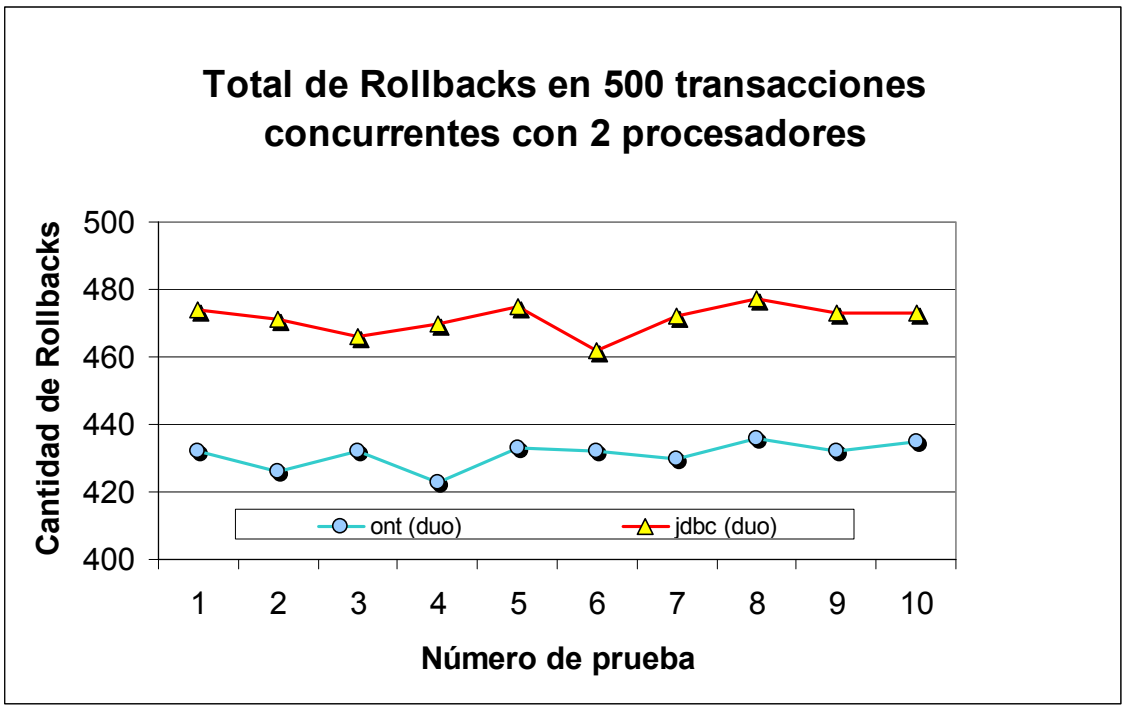



**Figura 5-7** Ejecución con dos procesadores.

En la figura 5-8 se ilustra la comparación del rendimiento entre un sistema uni-procesador y un multi-procesador. El porcentaje de cancelaciones utilizando el modelo ONT con un procesador es del 93.58% y con dos procesadores se obtiene un 86.22%, esto representa una mejora del 7.36% bajo el mismo modelo ONT. Utilizando JDBC con un procesador se obtiene un 96.84% de cancelaciones y 94.26% de cancelaciones en un sistema de dos procesadores, esto representa una mejora del 2.58% bajo JDBC.

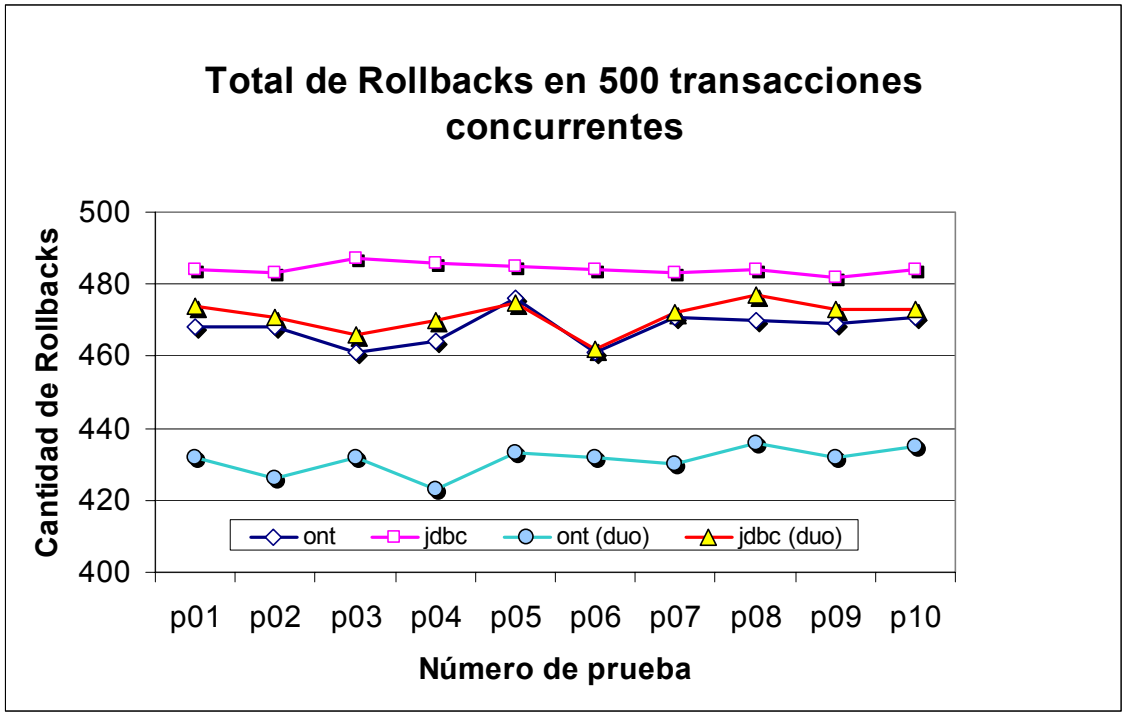



**Figura 5-8** Comparación de resultados entre en un sistema uni-procesador y un multi-procesador.

Por lo tanto se mejora el rendimiento de las transacciones utilizando el modelo ONT de un sistema uni-procesador a un multi-procesador, que usando JDBC. La figura 5-9 ilustra los rendimientos de estos escenarios, como se puede observar el rendimiento de las transacciones tiende a mejorar bajo ONT aumentando la cantidad de procesadores en el sistema que con JDBC.

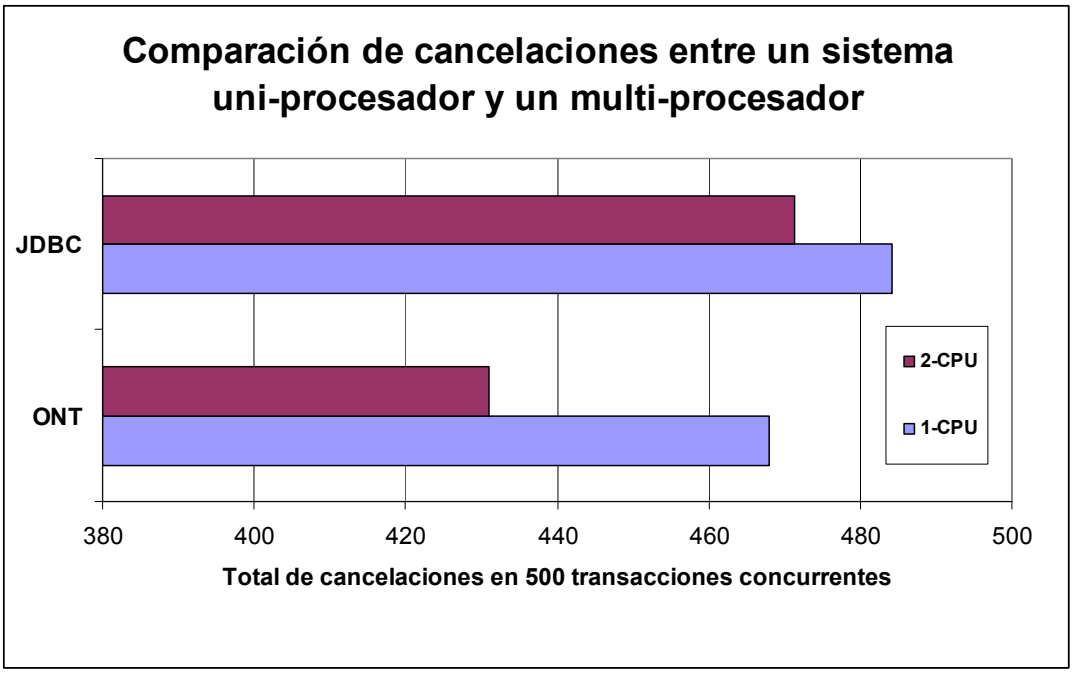



**Figura 5-9** Promedio de cancelaciones entre sistemas uni-procesamiento y multi-procesamiento.

## 5.3 Prueba de correctez

Cada una de las transacciones ejecutadas en ambos contextos (JDBC – ONT), van creando un archivo log con las operaciones realizadas en orden cronológico, pudiendo identificar que operaciones en conflicto tuvieron que ser serializadas, y cuales transacciones fueron abortadas. La siguiente historia representa una muestra de las operaciones llevadas a cabo en la prueba número 1, ejecutada con 50 transacciones concurrentes bajo el modelo ONT.

```
[1160387141546 11:45:41.546] <t11, bt, null>
[1160387141593 11:45:41.593] <t11, w, cant10>
[1160387141968 11:45:41.968] <t8, bt, null>
[1160387141968 11:45:41.968] <t8, try w, cant10>
[1160387142500 11:45:42.500] <t15, bt, null>
[1160387142500 11:45:42.500] <t15, try w, cant10>
[1160387144640 11:45:44.640] <t28, bt, null>
[1160387149703 11:45:49.703] <t28, r, cant10>
[1160387254390 11:47:34.390] <t15, rt, null>
[1160387249281 11:47:29.281] <t11, ct, null>
[1160387284421 11:48:04.421] <t8, rt, null>
[1160387449953 11:50:49.953] <t28, ct, null>
```

Esta historia es generada por el monitor MTA, donde se incluye la hora en que fue ejecutada la operación y una triupla de la forma <tx, operación, objeto>, donde:

- “tx” representa el id de la transacción;
- “operación” identifica el tipo de operación ejecutada pudiendo ser ‘bt’ (begin transaction), ‘ct’ (commit transaction), ‘rt’ (rollback transaction), ‘try r’ (intento de lectura), ‘try w’ (intento de escritura), ‘w’ (escritura exitosa) y ‘r’ (lectura exitosa).
- “objeto” representa el objeto afectado por “operación”.

En la figura 5-8 se ilustra un grafo de dependencias de la anterior historia, donde se cancelaron un total de 39 transacciones usando ONT contra 48 con JDBC, de una cantidad de 50 transacciones ejecutadas concurrentemente, lo cual representa una mejora a la tolerancia a fallas del 18%.

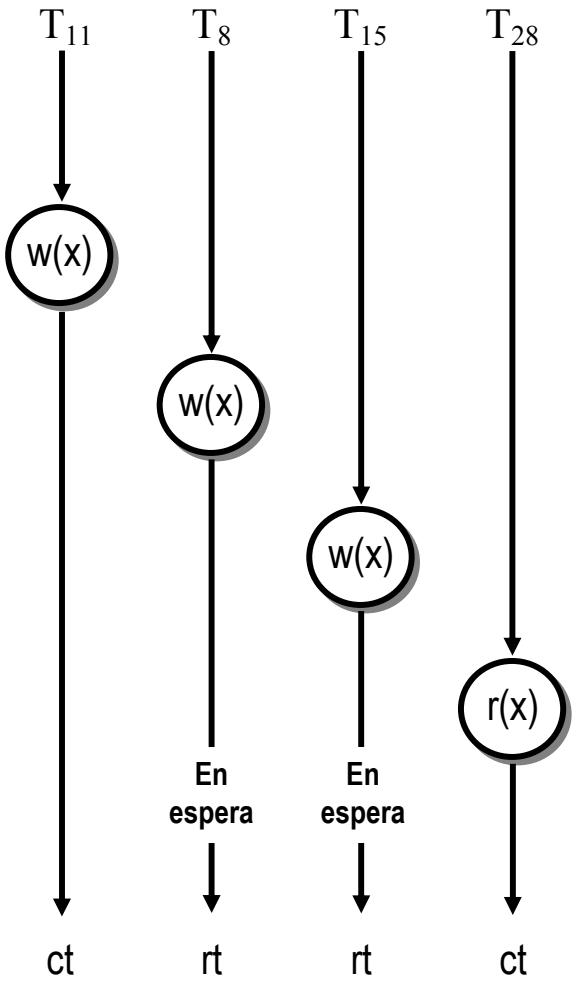



**Figura 5-10** Serialización de operaciones en conflicto.

El MTA preserva la consistencia de los datos, serializando las operaciones en conflicto.

# Capítulo 6 Ventajas y aplicabilidad

## 6.1 Tolerancia a defectos (*Fault tolerance*)

**Con el modelo CCxTA se consigue el efecto del protocolo MVCC donde las lecturas no bloquean a las escrituras y viceversa, pero con un control y resolución de malas dependencias**. CCxTA controla estas malas dependencias consecuencia de relajar el nivel de aislamiento de las transacciones concurrentes, a través del MTA. El MTA es capaz de verificar la consistencia de los objetos afectados concurrentemente y así deshacer los efectos de malas dependencias tales como: *lecturas-sucias*, *lecturas no-repetibles* y *fantasmas*. A través de la emisión de mensajes a las transacciones involucradas en una mala dependencia. De tal manera, cada transacción recibe además una indicación de hasta que punto hacer un rollback (*savepoint*) para evitar perder el trabajo realizado con éxito antes de la inconsistencia. Por tales efectos, el modelo es más "tolerante a defectos".

## 6.2 Más transacciones terminadas con éxito

Gracias al mecanismo de control de malas dependencias, las transacciones son capaces de eliminar los efectos de estas, a través del cancelaciones parciales (ver sección 4.5), es decir, cancelan sólo aquellas operaciones ejecutadas desde aquella operación que presentó una mala dependencia. El resto de las operaciones se conserva gracias al mecanismo de *savepoints*, eventualmente las operaciones canceladas son intentadas nuevamente (redo). Con este mecanismo una transacción que obtenga un mensaje de malas dependencias puede preservar el resto de su trabajo sin tener que cancelarlo totalmente, y así se consiguen mayor número de transacciones terminadas con éxito.

## 6.3 Disminución de fallas

Relajar el nivel de aislamiento de las transacciones al nivel menos restrictivo (*READ UNCOMMITTED*) se consigue que las lecturas no sean bloqueadas por las escrituras y viceversa (tal y como sucede en el protocolo MVCC) por lo tanto, al no existir bloqueos, se disminuye la presencia de "abrazos mortales", sin embargo, estos no son eliminados totalmente debido a que puede suceder la dependencia write-write (ver sección 2.4) la cual no es posible evitar el bloqueo, puesto que es la mala dependencia más crítica conocida como *lost-update*. De hecho, esta mala dependencia nunca sucede en un ambiente de transacciones concurrentes aún con el nivel de aislamiento menos restrictivo.

## 6.4 Comparación con otros modelos de control de concurrencia

El uso de protocolos de control de concurrencia basados en bloqueo, aumenta la posibilidad de fallas tales como "abrazos mortales" así como una disminución en el rendimiento de las aplicaciones. En la sección 5.2 si ilustran los resultados obtenidos al ejecutar un grupo de transacciones concurrentes basados en el modelo de bloqueo del DBMS Microsoft Sql

Server contra el mismo grupo de transacciones ejecutadas bajo el monitoreo del MTA, obteniendo un porcentaje del 14% de más transacciones terminadas con éxito utilizando MTA para 100 transacciones concurrentes. La figura 6-1 ilustra en color claro la cantidad de abortos y en color oscuro la cantidad de transacciones terminadas con éxito.

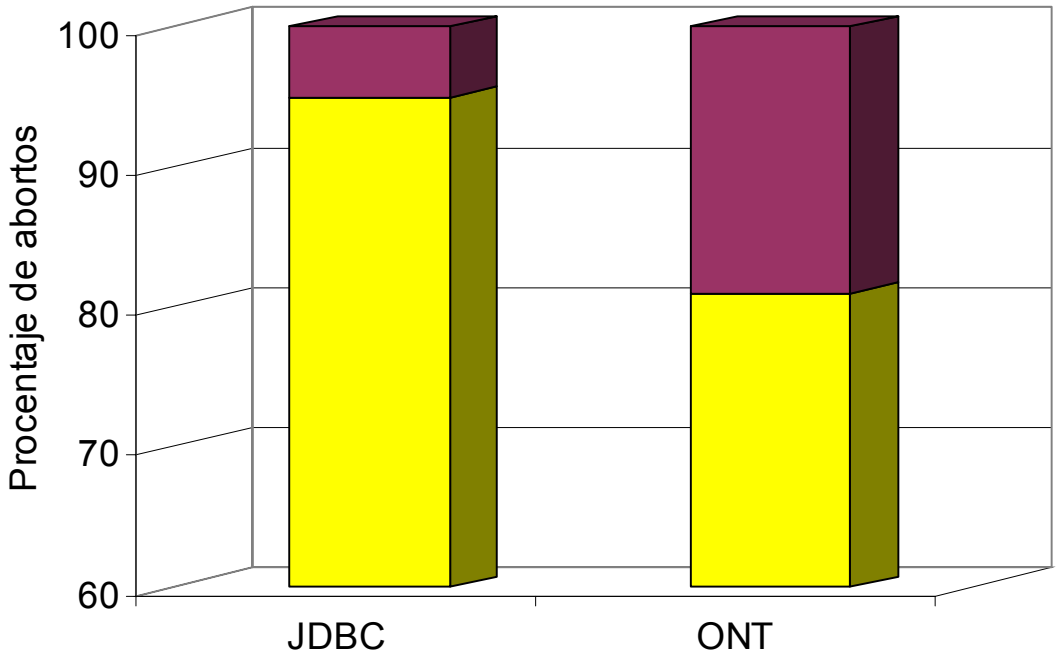



**Figura 6-1** Comparación de CCxTA vs. JDBC.

Existe una discusión entre los beneficios y desventajas de los protocolos pesimistas (basados en bloqueo) y los optimistas (MVCC). En [25] se describe un análisis comparativo sobre el protocolo MVCC utilizado en Oracle 9i y el protocolo basado en bloqueo utilizado por IBM DB2.

# Capítulo 7 Aplicaciones

A continuación se describen las aplicaciones que se pueden llevar a cabo con el modelo de control de concurrencia extendido para transacciones anidadas.

## 7.1 Transacciones Anidadas Móviles

Con el modelo Transacciones Anidadas Móviles (TAM) [13] es posible coordinar el procesamiento de transacciones en grupos de dispositivos móviles. TAM está basado en el modelo de TA [19], donde el control de las sub-transacciones -*unidades lógicas de trabajo*- es más flexible y robusto. Así, es posible procesar transacciones sobre grupos de dispositivos móviles sin acceso a redes cableadas o a Internet. Una ventaja adicional es que la distribución del procesamiento en sub-transacciones en cada dispositivo móvil, permite la confirmación o cancelación de manera independiente de los demás. Por lo tanto, las sub-transacciones confirmadas, no se ven afectadas por fallas o por espera de otras aún en proceso [19]. El objetivo principal de modelo TAM es minimizar la pérdida de trabajo realizado por una transacción cuando se utilizan dispositivos móviles, debido a las características inestables de estos ambientes.

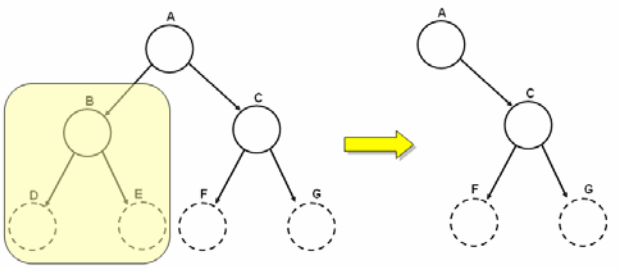



**Figura 7-1** Validación de A, a pesar del aborto de B.

La figura 7-1, muestra el escenario donde una transacción A (Transacción Raíz) se divide en las transacciones B, D, … , G. Si la transacción B (y sus respectivas sub-transacciones D y E) fallan, A puede validar con sólo el trabajo realizado por las sub-transacciones C, F y G. Sin embargo, no todos los problemas pueden ser resueltos usando el modelo de TAM. Este modelo es muy útil para aquellos procesos donde la lógica del problema puede ser distribuida en diferentes dispositivos, cada uno de estos procesando una sub-transacción.

## 7.2 Workflow transaccional

En las organizaciones se genera la información necesaria para el funcionamiento de sus diferentes procesos. Cuando la información generada es usada en un lugar distinto al que la produce, se debe tener en cuenta que los documentos hacen un recorrido dentro de la empresa, antes de llegar a su destino final; esto es conocido como flujo de trabajo (*workflow*).

Tradicionalmente, el principal inconveniente del manejo de documentos ha sido la dificultad para distribuirlos con un mismo formato, según la información que contengan, a las dependencias de la empresa que requieran dichos artículos. Las redes de área local (intranet) han creado medios donde se pueda superar este obstáculo, haciendo que casi cualquier información sobre la red este disponible en cualquier punto de la empresa. Debido a que la intranet permite una buena gestión de documentos por medio de una sede Web, se puede aprovechar esta infraestructura para crear nuevas herramientas para los administradores de redes, que ahora deben enfrentar consideraciones sobre el acceso, control y la autoría de los documentos (ver figura 7-2).

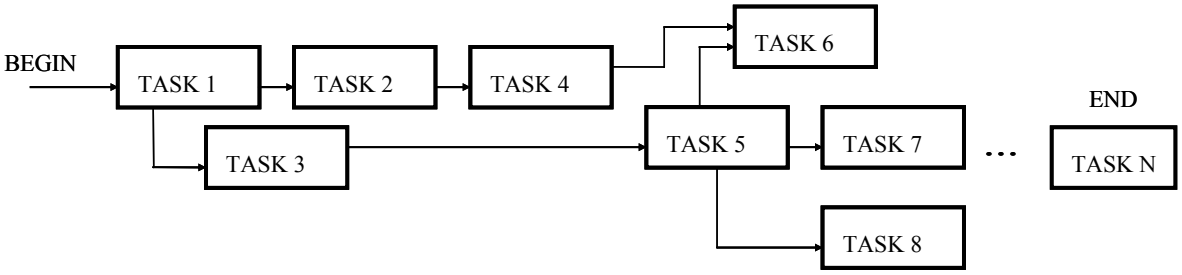



**Figura 7-2** Grupo de tareas bajo el contexto de workflow.

Para mejorar la tolerancia a fallas de este grupo de procesos se define: *Workflow Transaccional.* Donde el objetivo es proveer a los procesos de un entorno transaccional y sus propiedades ACID. La figura 7-3 ilustra su arquitectura.

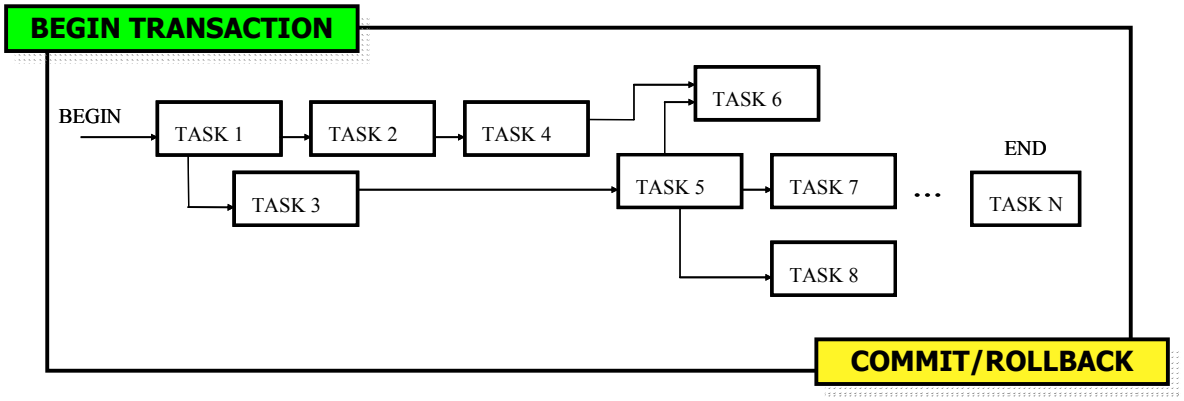



**Figura 7-3** Arquitectura de workflow transaccional.

Sin embargo, la propiedad Atomicidad representa una desventaja, debido a que si una de las tareas falla el resto de las tareas se tendrán que cancelar, como se ilustra en la figura 7-4.

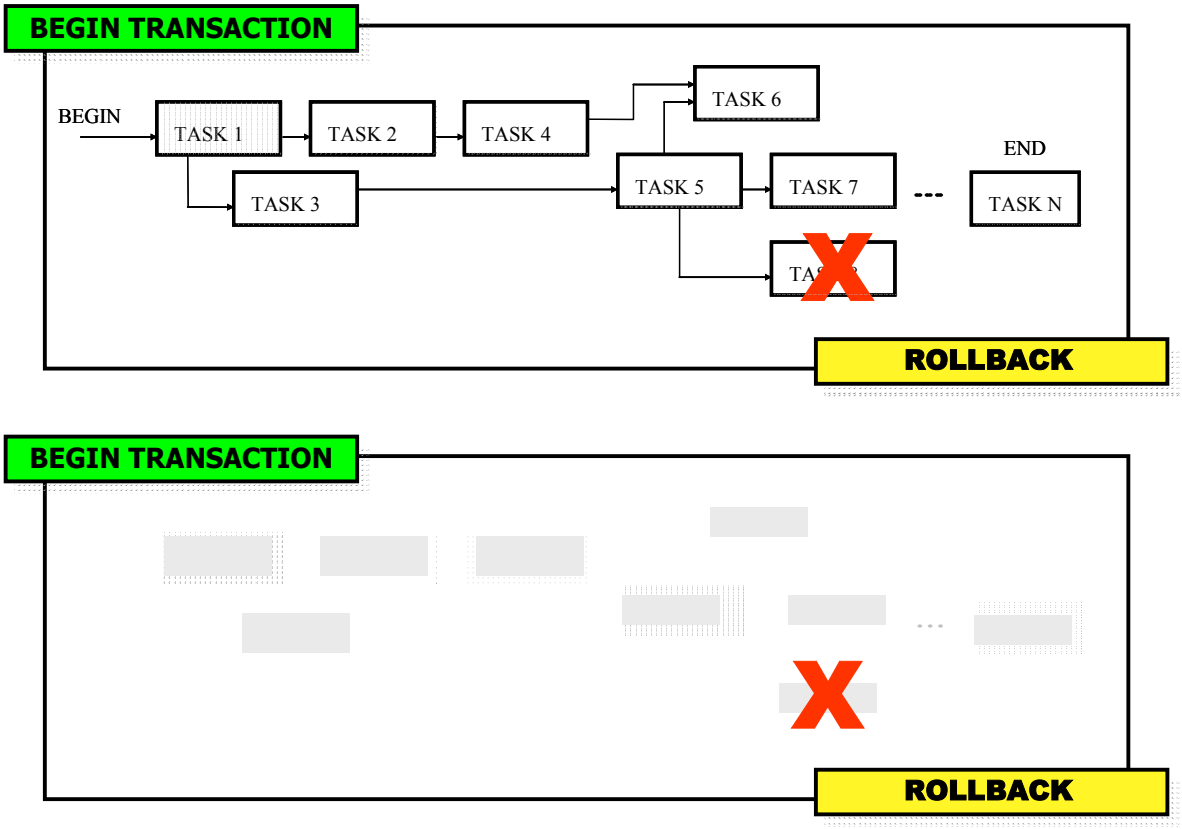



**Figura 7-4** Cancelación de todo el proceso debido a la Atomicidad.

Por lo tanto, se propone Workflow Transaccional Anidado. Con este modelo ahora el workflow se rige bajo las reglas del modelo de transacciones anidadas (Cerradas/Abiertas) y bajo las políticas del control de concurrencia extendido CCxTA. La figura 7-5 ilustra este escenario. Si uno de los procesos falla, no es obligatorio cancelar el resto de los procesos que previamente han terminado con éxito. Para esto se van creando marcas (con *savepoints*) que indican puntos de consistencia en el flujo. Por lo tanto, ahora es posible cancelar sólo un grupo de procesos hasta una marca dada, y no el total de los procesos del workflow.

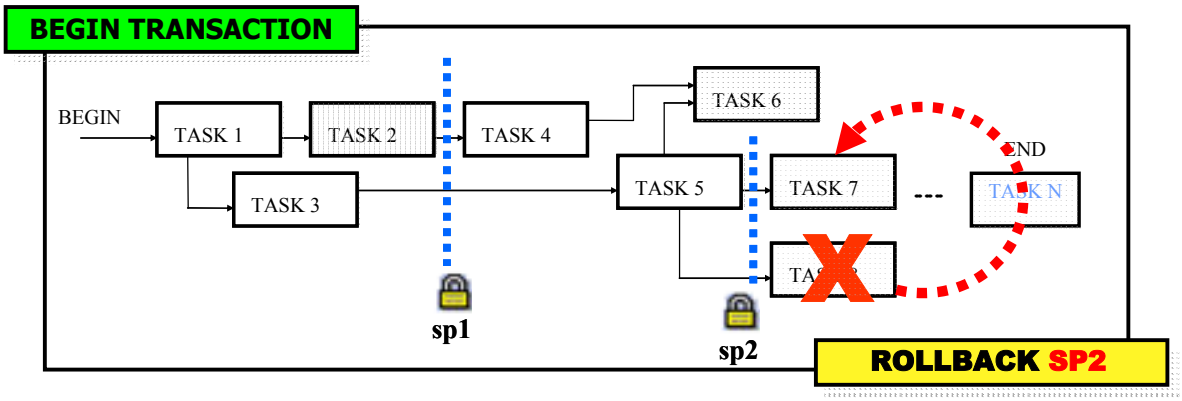



**Figura 7-5** Ejecución de una cancelación parcial.

# Aportaciones y conclusiones

En este trabajo se presentó el análisis, diseño e implementación de un modelo de Control de Concurrencia para Transacciones Anidadas, CCxTA, tanto Cerradas como Abiertas, el cual permite a transacciones concurrentes utilizar datos no confirmados sin caer en inconsistencias. La implementación es mediante un Monitor que administra la presencia de malas dependencias y asegura la consistencia de los datos al final de la ejecución de las transacciones concurrentes.

- **Aportaciones de modelo CCxTA**

1. Las transacciones concurrentes pueden trabajar con datos no confirmados, preservando la consistencia de los datos al final de ser ejecutadas.

2. Uso del nivel de aislamiento read uncommitted, el más relajado, preservando la constancia final de los datos (no se reportan en la literatura tal tipo de modelos).

**3.** Los lectores no bloquean a los escritores y los escritores no bloquean a los lectores.

   - *Lectura:* la disponibilidad de los objetos es compartida a todos los niveles y ramas del árbol de transacciones, tal que las transacciones concurrentes obtienen un bloqueo compartido (*shared lock: slock*) sobre el mismo objeto.

   - *Escritura:* se relaja el nivel de aislamiento al menos restrictivo *read uncommitted,* para permitir el acceso a objetos previamente bloqueados.

**4.** Como consecuencia de 1., 2. y 3., se disminuye el tiempo de espera de las transacciones concurrentes, y como consecuencia aumenta el número de transacciones terminadas con éxito.

**5.** Con CCxTA, basado en un protocolo de control de concurrencia pesimista (2PL), se consiguen los efectos de los protocolos optimistas –tal como MVCC.

El Monitor implementa el CCxTA y administra los objetos procesados, vigila su estado y detecta estados inconsistentes. Cuando surgen malas dependencias el monitor envía mensajes de cancelación, parcial la mayoría de las veces, a cada una de las transacciones involucradas, y mantiene la consistencia de los datos.

- **Aportaciones del Monitor que implementa el CCxTA**

1. Se administran con robustez los efectos de inconsistencia de las malas dependencias.

2. Incluye la creación de un *API* para ejecutar transacciones bajo los modelos de Transacciones Anidadas Cerradas y Abiertas.

3. Buen rendimiento al conseguir aumentar el número de transacciones terminadas con éxito, respecto al JDBC.

4. Disminución de la presencia de abrazos mortales al relajar el nivel de aislamiento.

- **Aportaciones y conclusiones respecto a aplicabilidad**

Áreas relevantes donde puede darse gran aplicabilidad del modelo CCxTA y del Monitor es en las llamadas **computación móvil y computación ubicua**, las cuales frecuentemente operan sobre redes inalámbricas y son más proclives a cierta inestabilidad por su propia naturaleza. Empleando el monitor de TAs es posible que las transacciones ejecutadas desde dispositivos tales como lap-tops, PDAs y teléfonos móviles sean terminadas con éxito y en mayor número.

Otra área de aplicabilidad es en el **Flujo de Trabajo** (*WorkFlow)*: al agrupar los procesos de trabajo en un contexto transaccional y deshacer los efectos de trabajos no concluidos mediante cancelaciones parciales, pueden preservarse las tareas que hayan sido concluidas en su totalidad. Así, se preservan mayor cantidad de trabajo confirmado y verificado, y por lo tanto se hace más eficiente la conclusión de procesos exitosos y supervisados.

# Glosario de términos

**2PL** *Two Phase Locking*: Protocolo de control de concurrencia basado en bloqueos, donde existen dos etapas: etapa 1, adquisición de bloqueos y etapa 2, liberación de los mismos.

**API** Una API (del inglés *Application Programming Interface* - Interfaz de Programación de Aplicaciones) es el conjunto de funciones y procedimientos (o métodos si se refiere a programación orientada a objetos) que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción.

**DBMS** (*Database Management System*) Los sistemas de gestión de base de datos son un tipo de software muy específico, dedicado a servir de interfaz entre la base de datos, el usuario y las aplicaciones que la utilizan. Se compone de un lenguaje de definición de datos, de un lenguaje de manipulación de datos y de un lenguaje de consulta.

**DSN** *Data Source Name* (Nombre de origen de datos), representa una fuente de datos configurada por el usuario para conectarse a una Base de datos. Donde el usuario tiene que especificar una serie de información que permitan al Controlador saber con qué fabricante(s) se tiene que conectar y la cadena de conexión que tiene que enviarle a dicho fabricante(s) para establecer la conexión con la fuente de datos ODBC accedida por el proveedor en cuestión.

**JDBC** Es el acrónimo de *Java Database Connectivity*, un API que permite la ejecución de operaciones sobre bases de datos desde el lenguaje de programación Java independientemente del sistema de operación donde se ejecute o de la base de datos a la cual se accede utilizando el dialecto SQL del modelo de base de datos que se utilice.

**ODBC** *Open DataBase Connectivity*: es un estándar de acceso a Bases de Datos desarrollado por *Microsoft*, el objetivo es acceder a cualquier DBMS, esto se logra al insertar una capa intermedia entre la aplicación y el DBMS. El propósito de esta capa es traducir las consultas de datos de la aplicación en comandos que el DBMS entienda.

**PDA** *Personal Digital Assistant*, (Asistente Personal Digital) es un dispositivo de mano originalmente diseñado como agenda electrónica (calendario, lista de contactos, bloc de notas y recordatorios) con un sistema de reconocimiento de escritura.

**SQL** (*Structured Query Language*) es un lenguaje declarativo de acceso a bases de datos relacionales que permite especificar diversos tipos de operaciones sobre las mismas. Una de sus características es el manejo del álgebra y el cálculo relacional permitiendo lanzar consultas con el fin de recuperar información de interés de una base de datos.

# Referencias

[1] ANSI X3.135-1992, “American National Standard for Information Systems – Database Language – SQL”, Nov 1992.

[2] Barbará, Daniel. “Mobile Computing and Databases – A Survey”. IEEE Transactions on Knowledge and Data Engineering, vol. 11, No. 1, January-February 1999.

[3] Berenson Hal, Bernstein Philip A., Gray Jim, Melton Jim, O'Neil, Patrick E. O'Neil Elizabeth J. “A Critique of ANSI SQL Isolation Levels”. SIGMOD Conference 1995: pp. 1-10. 1995.

[4] Bernstein P.A. and Goodman N., “Multiversion Concurrency Control-Theory and Algorithms,” ACM Trans. Database Systems, vol. 8, no. 4, pp. 465-483, Dec. 1983.

[5] Bertino Elisa, Catania Barbara, Vinai A. “Transaction Models And Architectures”. Encyclopedia of Computer Science and Technology, Vol.38, Marcel Dekker, pp.316-400. 1998.

[6] Date C. J. and White C. J. “An Introduction to Database Systems”, 7a. ed., Addison-Wesley. 2001.

[7] Dijkstra, Edsger W. “The structure of the ‘THE’-multiprogramming system”. Communications of the ACM. Volume 11, Issue 5. pp 341 – 346. ISSN: 0001-0782. May 1968.

[8] Doucet A. and Gançarski S. and Leon C. and Rukoz M. “Estrategias para verificar restricciones de integridad globales en multi base de datos con transacciones anidadas”, in Proc. XXVI Conferencia Latinoamericana de Informática, CLEI'2000, Mexico City. 2000.

[9] Dunham Margaret H., Vijay Kumar. “Impact of Mobility on Transaction Management”. International workshop on data engineering for wireless and mobile access. pp 14-21. 1999.

[10] Forman H., George, Zahorjan, John. “The Challenges of Mobile Computing”. Computer Science & Engineering, University of Washington, March 9, 1994.

[11] Gama L. A., Alvarado M. “Transacciones para Cómputo Móvil: presente y perspectiva futura”. Revista Digital Universitaria, Vol. 3 No. 4. ISSN: 1607-6079. 2002. http://www.revista.unam.mx.

[12] Gama Luis A., Alvarado Matías. “Concurrency control for Read-Only in Mobile Nested Transactions: Wireless Nomadic Teams within Oil Framework.” 2nd. Workshop on Intelligent Computing in the Petroleum Industry ICPI 2003. August, 2003.

[13] Gama Luis A., Alvarado Matías. "Mobile Nested Transactions for Nomadic Teams." *Special Issue on Intelligent Computing for Pretoleum Industry. Expert Systems with Applications,* Matias Alvarado, Leonid Cheremetov and Francisco Cantu (Eds.). Vol. 26, No. 1. 2004.

[14] Gama-Moreno Luis A., Alvarado Matías. "Mobile Groups based on Nested Transactions". In Alvarado, Matias; Sheremetov, Leonid proeedings of 1st Workshop of Intelligent Computing in the Petroleum Industry ICPI'02. ISBN: 968-489-013-3. November, 2002.

[15] Gray Jim, Reuter Andreus. "Transaction Processing: Concepts and Techniques." Morgan Kaufmann Publishers, Inc. ISBN 1-55860-190-2. 2003.

[16] Kumar V., Dunham M. H. "Defining Location Data Dependency, Transaction Mobility and Commitment", Technical Report 98-CSE-01, Department of Computer Science and Engineering, Southern Methodist University, February. 1998.

[17] Lee Victor C. S., Lam Kwok-Wa. "Optimistic Concurrency Control in Broadcast Environments: Looking Forward at the Server and Backward at the Clients". MDA: pp. 97-106. 1999.

[18] Madria Sanjay Kumar, Bhargava Bharat K. "A Transaction Model for Mobile Computing". IDEAS: pp. 92-102. 1998.

[19] Moss J. E. B. "Nested Transactions: An Approach to Reliable Distributed Computing". MIT Press, Cambridge, MA. 1985.

[20] Murthy V. K. "Seamless Mobile Transaction Processing: Models, Protocols, and Software Tools". Proceedings of the Eighth International Conference on Parallel and Distributed Systems (ICPADS'01). pp. 0147. 2001.

[21] Oracle. "Berkeley DB Reference Guide: Degrees of Isolation". White Paper. 2004. (http://www.oracle.com/technology/documentation/berkeley-db/db/ref/transapp/read.html)

[22] Özsu M. T. and Valduriez P. "Principles of Distributed Database Systems". 2nd edition, Prentice-Hall, Inc., ISBN 0-13-691643-0, pp. 381-401. 1999.

[23] Pitoura E., Bhargava B. "Maintaining Consistency of Data in Mobile distributed Environments". Proceedings of the 15 th International Conference on Distributed Computing Systems, Vancouver, Canada, May 30-June 2, 1995.

[24] Rito Silva António, Pereira João, Alves Marques José. "A Framework for Heterogeneous Concurrency Control Policies in Distributed Applications". IEEE Proceedings of the 8th International Workshop on Software Specifications and Design. March 1996.

[25] Rokytskyy, Roman. “Firebird and Multi Version Concurrency Control”. White Paper. IBM Corporartion. 2002.

[26] Satyanarayanan, M. “Fundamental Challenges in Mobile Computing”. School of Computer Science, Carnegie Mellon University. 1996.