Base de Datos

Implementación de algoritmo de ARIES sobre desarrollo de DBMS

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Luciano Leggieri
lleggieri@dc.uba.ar

Julian Berlín
jberlin@dc.uba.ar

Victor Cabas
vcabas@dc.uba.ar

Facundo Pippia
facundomensajes@hotmail.com

Director:
Alejandro Eidelsztein
ae0n@dc.uba.ar

Abstract

ARIES es un algoritmo de recuperación muy popular que usa un enfoque steal / no-force. Comparado con otros esquemas de recuperación, es simple y soporta diferentes grados de granularidad en los bloqueos. Luego de una caída, este algoritmo procede en tres fases para dejar la base de datos en el estado que tenia antes del crash. El objetivo de este trabajo es realizar una implementación de ARIES en un motor de base de datos construido íntegramente en Java.

Keywords: Bases de datos relacionales, ARIES, Transacciones, Recovery Manager

Índice

Índice de figuras

1. Introducción

1.1. Objetivos

El objetivo de este trabajo es construir un motor de base de datos relacional con fines académicos, para poder obtener una herramienta que permita una visión profunda de ***cómo*** funciona un DBMS por dentro. Este proyecto se especializa en mostrar el trabajo de concurrencia transaccional y recuperación del motor. Para la parte de recuperación, este motor se basa en la familia de algoritmos ARIES [ARIES], el cual posee ciertas propiedades deseables en un método de recuperación eficiente y seguro.

Las bases de datos académicas existentes, suelen concentrarse en el funcionamiento del analizador sintáctico, conversión a Algebra Relacional de la consulta, y posterior optimización de la misma. Suelen proveer índices basados en árboles B+, así como diversos métodos para la persistencia de tablas en disco y métodos de paginación en el Buffer Manager. Sin embargo, temas tan importantes como procesar varias consultas de manera simultánea y el sistema de recuperación de la base de datos frente a desastres, suelen ser ignorados por estos motores educacionales. Esto conlleva a que los interesados en aprender el funcionamiento de un DBMS, solo obtengan la teoría en lo que respecta al manejo de transacciones (concurrencia y control de bloqueo de datos) y recuperación de las mismas en caso de una caída imprevista del motor.

1.2. Problema a tratar

Con este proyecto se intenta solventar esta falencia en lo que respecta a motores académicos, proveyendo de un motor de base de datos educacional que muestre, tanto en la implementación como en el uso, a aquellos módulos encargados del manejo transaccional de las consultas a la base (Transaction Manager, Lock Manager) y recuperación de aquellas que hayan terminado pero sus contenidos no estuvieran volcados en un medio persistente (Recovery Manager). Este motor también tiene como finalidad dotar al Departamento de Ciencias de la Computación [DCFCEN] de nuestra facultad [FCEN], de una herramienta para mostrar y enseñar el funcionamiento interno de una base de datos relacional con soporte de transacciones y recuperación.

En la búsqueda de soluciones similares en distintos departamentos de las universidades del mundo, hemos encontrado que aquellas que muestran el funcionamiento de un DBMS, se concentran, como ya fue mencionado, en el análisis y optimización de consultas. Solo se ha visto un trabajo que extiende a una base de datos educacional existente, agregándole soporte de

transacciones y recuperación [MINIREL].

1.3. Nuestro aporte

La contribución de este proyecto es la construcción de un motor de base de datos con fines educativos. Esto nos dio los objetivos de hacerlo simple, documentar su implementación y que sea posible mostrar, cuando se encuentra en uso, qué es lo que va haciendo, así como el estado interno de las estructuras del sistema.

Para la construcción del motor nos hemos basado en lo aprendido durante la cursada de Bases de Datos [BDFCEN], y en la bibliografía de la cátedra [RAMA03], [ULLMAN88] y [BERN87]

Como trabajo durante del curso, se nos pidió que el motor soporte el aborto de transacciones y recuperación de las mismas en caso de caídas, pero el único requisito era que el log de los eventos sea construido utilizando la técnica WAL, por lo que se realizó un sistema simple pero funcional. Luego, como trabajo final, se decidió cambiar este sistema por uno basado en ARIES, aunque manteniendo el fin educativo y de simplicidad en la implementación.

Como se ampliará posteriormente, ARIES nos ha permitido que el Buffer Manager utilice un esquema STEAL / NO-FORCE [RAMA03], el cual simplifica la implementación del mismo. También era requisito que el sistema de bloqueos se encuentre basado en el algoritmo Two Phase Locking. Hemos extendido este requisito y agregamos soporte para los cuatro niveles estándares de aislamiento [SQL92].

El trabajo también contiene un pequeño sistema de índices basados en hash para proveer mayor concurrencia entre consultas paralelas. Estas consultas son escritas en formato SQL estándar, y el motor se encarga de analizar y descomponer la sentencia SQL para luego poder ejecutarla sobre la base de datos.

Para visualizar el funcionamiento del motor, y proveer una manera de ejecutar consultas y ver sus resultados, se dispone de un cliente gráfico [KANONOP], el cual permite simular muchas conexiones al servidor, con fines de poder ejecutar varias sentencias de manera concurrente. Este cliente también informa del estado de los objetos bloqueados y liberados durante el transcurso de cada transacción, y muestra los eventos del log correspondientes a cada operación dentro de las mismas.

Para el diseño del motor nos hemos desviado de los clásicos algoritmos procedurales mostrados en los distintos papers, para intentar un acercamiento orientado a objetos. Esto incluye el uso de patrones de diseño y una arquitectura modular para un mayor entendimiento de cada componente, así como permitir modificaciones de las mismas por separado sin necesidad de cambiar a las demás.

2. Estado del arte

Actualmente hay una inmensa variedad de DBMS tanto con fines industriales como educativos. En esta sección se muestran los proyectos y sistemas mas relevantes cuyas implementaciones nos parecieron interesantes sobre todo lo que respecta al tratamiento de transacciones y la recuperación.

2.1. DBMS para propósitos académicos y de investigación

Entre los motores de base de datos educativos más importantes se encuentran:

MINIBASE, es una base de datos de objetivo educacional. Tiene implementados el Buffer Manager, índices B+ y Disk Space Manager. No tiene implementada la parte de concurrencia, transacciones y recuperacion del sistema [MINIBAS].

MINIREL es una RDBMS multi-usuario simple que se basa en ARIES para el manejo del log y recuperación. Implementa la cola del log con una sola página que mantiene en memoria compartida. La lectura y escritura de la misma es mutuamente excluyente. Tiene limitaciones de baja concurrencia en el Log Manager y asume crash simples provocados por el propio sistema [MINIREL].

LEAP es una base de datos con soporte multiusuario que usa como lenguaje de consultas el Álgebra Relacional (base teórica para lenguajes como SQL) que maneja también concurrencia y transacciones [LEAP].

System R es una base de datos construida como un proyecto de investigación en el IBM San Jose Research en los años '70. Su sistema de recuperación se basa en el protocolo DO-UNDO-REDO, el cual divide a las páginas de la base en dos modelos: *nonshadowed* y *shadowed*. En el primero no hay recuperación automática frente a una caida del sistema o el aborto de una transacción; en el segundo, se guardan versiones de las páginas modificadas por si es necesario restaurarlas a un estado anterior. Mas detalles pueden encontrarse en [GRAY80].

University Ingres, Berkeley Ingres o Ingres89 es la versión original de Ingres desarrollada en UC Berkeley durante la década del 70; la primera implementación de un sistema de administración de base de datos relacional. Usa *QUEL* (QUEry Language) como DML, y el funcionamiento y la confiabilidad son solamente justos a adecuado. Este sistema multiusuario brinda una vista relacional de datos, soporta dos niveles altos de sublenguajes de

datos no procedural, y funciona como una colección de procesos usuario sobre sistema operativo UNIX. Para la recuperación se utitiza un archivo temporal por proceso, el cual se borrará o se ejecutará nuevamente si es que el proceso actual ya había comenzado [STONE76].

2.2. DBMS para propósitos comerciales

DERBY es una DBMS comercial y gratuita, implementada en JAVA. Basa el manejo de log de transacciones y recuperación en los algoritmos de ARIES. Tiene algunas diferencias con el estándar ARIES:

- En vez de guardar el pageLSN con la página, guarda un número de versión de la misma.
- No guarda una tabla de páginas sucias en el checkpoint. Durante el checkpoint solamente fuerza las paginas a disco.
- El Undo empieza en el LSN actual al momento de comenzar el checkpoint.
- En en el reinicio del motor, Derby hace el Redo o el Undo según el que tenga LSN mas chico para empezar.
- Usa transacciones internas en vez de Nested Top Level Actions para separar los cambios estructurales de las operaciones normales.
- Derby omite la fase de análisis en el reinicio del motor ya que no lo requiere por la forma en la que hace los checkpoints.

Mas información sobre Derby en [DERBY]

ALTIBASE es un motor de base de datos relacional de alta performance y tolerancia a fallas que maximiza el uso de la memoria principal para las operaciones logrando una gran performance. Está pensada principalmente para dispositivos móviles. Un Log Flush Thread se encarga de enviar el log a disco sin interferir con las transacciones activas. El log se escribe en múltiples archivos para mejorar eficiencia en la recuperación. También separa las transacciones en distintos niveles de durabilidad, usa un buffer de memoria y otro archivo de mapeo a memoria, ambos como buffer de log. En los niveles más altos, un thread sincroniza con un archivo de log en disco. En los niveles más bajos, no garantiza la durabilidad del commit ya que en éstos el estado de commit de la transacción se pone antes de ser escrito en el archivo de log. Tambien provee distintos niveles de logging basándose en la importancia entre la performance de la transacción y la consistencia de los

datos [ALTIBAS].

SQL Server tambien usa ARIES en su Recovery Manager, junto con el protocolo WAL. Para que una transacción asegure su estado de commit todos sus registros de log deben estar escritos en disco. Tiene un enfoque NO FORCE. Soporta locking a nivel registro, rango, página o tabla. La tasa de transacciones se incrementa usando transacciones pequeñas [KB230785].

Tanto [ARIES] como [ARIESRH] mencionan la implementación de los algoritmos ARIES en la familia de productos DB2 de IBM.
DB2 guarda imágenes del espacio de tablas y va a la última estable que tenía (en vez de tener que ir al primer registro de log y rehacer toda la operación). DB2 usa entradas en la tabla SYSCOPY para identificar y localizar la copia más reciente a tomar para restaurar. Entonces aplica los cambios a los datos siguiendo la secuencia del log. QUIESCE utility es una utilidad para establecer un punto de consistencia para un table space o un conjunto de table spaces logicamente relacionados. REPORT RECOVERY recopila info de SYSIBM.SYSCOPY y SYSIBM.SYSLGRNX (que son parte del directorio de DB2) para brindar información importante para la recuperación. Hay dos tipos de metodos de recovery: *to current (al más reciente punto de consistencia)* es el estándar para errores de hardware o software; *to a specific point-in-time (a un punto específico)* se aplica mayormente a errores de aplicación [BRUNI02].

Oracle es uno de los motores de base de datos relacionales mas importantes de la industria y es usado a gran escala por muchas compañías y multinacionales. Para su recovery manager (RMAN) utiliza una tecnología llamada *Flashback*, la cual consiste en un conjunto de métodos para recuperar la integridad de la base de datos luego de producido un crash o un error humano. Entre las características mas importantes que brinda la tecnología Flashback se encuentra la posibilidad de generar consultas a versiones antiguas del esquema de objetos, generar consultas de la información histórica de la base de datos y realizar un auto-reparación de información lógica corrupta; todo esto mientras la base de datos se encuentra online. Por otro lado no hay mucha información disponible acerca de qué métodos y enfoques utiliza Oracle en su Recovery Manager [ORACLE].

3. Diseño

3.1. Modelo propuesto

El sistema sigue el protocolo Cliente - Servidor, donde toda la complejidad del motor de base de datos se encuentra en el servidor. El cliente es una aplicación liviana encargada de la comunicación con el usuario, enviando los pedidos de consultas al servidor, esperando por el resultado y mostrando al mismo por pantalla.

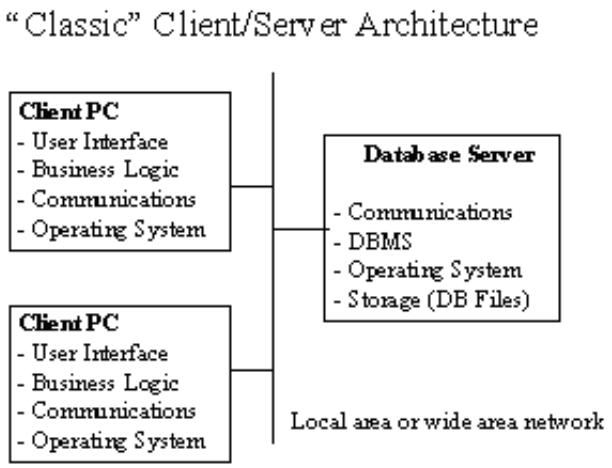

Figura 1: Arquitectura del modelo Cliente / Servidor

Este modelo permite una mejor abstracción y separación de la funcionalidad propia del motor, con aquella encargada de la interfaz al mundo exterior. Siguiendo un esquema de envío de mensajes, ambos programas pueden ejecutarse tanto en la misma máquina como en computadoras distintas, sin perder ninguna clase de utilidad.

3.2. Arquitectura del servidor

Teniendo en cuenta los fines académicos del trabajo, se decidió separar cada componente del servidor en módulos bien diferenciados e independientes, tratando de maximizar la cohesión y tener un bajo acoplamiento. Cada modulo respeta una interfaz, la cual es usada por aquellos que lo acceden, desligándose así de cómo está implementado cada uno. Esto permite poder modificar a futuro la implementación de un modulo sin que ello afecte

a los restantes, mientras se respeten las interfaces establecidas en el sistema. El enfoque esta basado en un diseño orientado a objetos, haciendo uso de diferentes patrones de diseño para poder facilitar el entendimiento de cada modulo y proveer de soluciones estándar. Cada objeto tiene un propósito específico y diferenciable, lo que permite un mayor entendimiento del desarrollo de los módulos y funcionamiento del motor.

En la figura 2 muestran los módulos e interrelaciones existentes en el sistema:

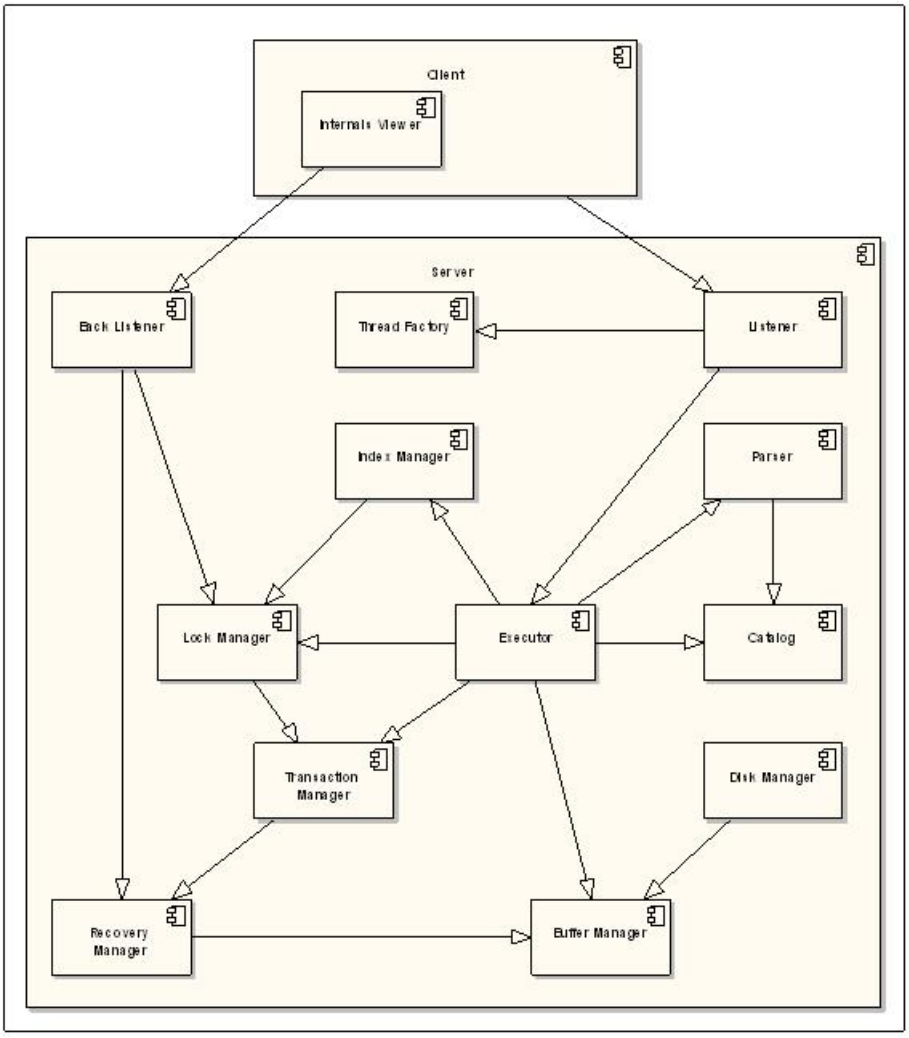

Figura 2: Arquitectura del Servidor Kanon

A continuación se explican con mayor detalle aquellas componentes que tienen mayor relevancia con el objetivo de este trabajo. En cada una se pun-

tualiza la trascendencia que tienen los algoritmos de ARIES sobre la misma, enfocando su funcionalidad principal en la sección del Recovery Manager.

3.3. Disk Space Manager

El DiskSpace Manager es el encargado de guardar las páginas en un medio de almacenamiento persistente, en nuestro caso mediante archivos en disco duro dentro de un directorio ya preestablecido.

Las páginas se guardan en formato binario. La elección del mismo se debe a:

- La necesidad de implementar índices, o sea que las páginas no sólo contienen información de las tablas.
- Por ARIES, ya que en principio, el algoritmo guarda en los eventos de log aquellos cambios de los registros de las tablas a nivel de bytes.
- Por similitud a un motor de base de datos real.

El formato utilizado en un principio había sido XML, y estaba justificado por el hecho de tener un formato más legible sin necesidad de transformación alguna. Esto es, se podía abrir una página, con cualquier visor de texto, y saber cómo era su contenido; útil para fines académicos. El formato XML también fue usado en el log de eventos para la recuperación del sistema, el cual fue cambiado por el log de ARIES.

También hay que destacar que al no realizar la conversión XML, el Disk Manager toma una lógica mucho más simple, pues sólo debe estampar el arreglo de bytes representantes de una página provistos por el Buffer Manager en un archivo en disco. Y viceversa para la lectura.

Este manager se completa con métodos de creación y borrado de página:

- Crear de una nueva página. A partir de las páginas existentes se obtiene cuál será el número de la nueva página. La implementación está optimizada para tablas que no necesitan ningún orden en particular sobre sus registros.
- Borrar una página. Borra el archivo de la página del disco persistente.

3.4. Buffer Manager

Para el manejo de las páginas en el Buffer Manager se utiliza el esquema **steal / no-force**. Con **no-force**, al no requerir que la página se guarde en disco estable para cada commit, esta escritura se puede realizar una vez luego que todas las transacciones hayan modificado la página, reduciendo considerablemente la cantidad de E/S. Esta mejora se combina con el enfoque **steal**, el cual permite que una página que haya sido modificada por una transacción en curso, sea guardada en disco y eliminada de la memoria (para hacer lugar a otra página), aumentando la capacidad de la memoria virtual del motor. Luego si esos cambios deben ser deshechos, entrará en juego el proceso de rollback propuesto por ARIES, leyendo el archivo de log para revertir las modificaciones.

Otra posibilidad más simple de implementar es el esquema **no-steal / force**, pero el mismo posee determinadas desventajas. **No-steal** asume que todas las páginas modificadas por las transacciones en curso pueden estar fijas en el pool de páginas del Buffer Manager lo cual es una asunción no realista, pues limita la cantidad de transacciones que se pueden estar ejecutando de manera concurrente y el tamaño de las mismas. Con **force**, si una misma página es modificada por muchas transacciones seguidas, ésta se escribe en disco a medida que las transacciones van haciendo commit.

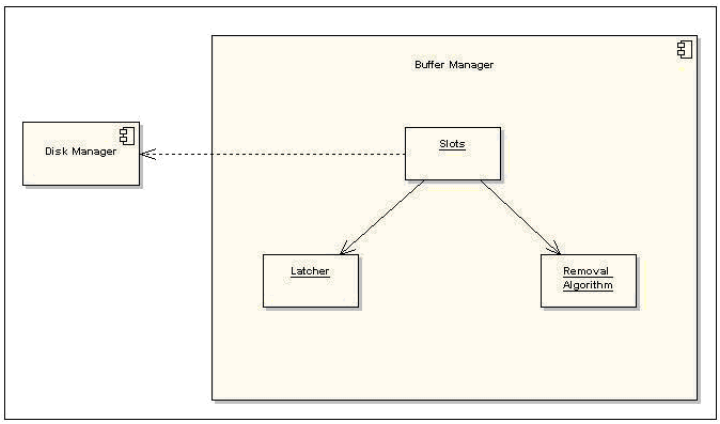

Figura 3: Arquitectura del Buffer Manager de Kanon

Otro punto importante tratado por ARIES es el control de acceso a las páginas. Las páginas disponen de tres tipos de bloqueos:

- **pin**: Sirve para que el algoritmo de remoción utilizado por el Buffer Manager no deseche una página que esté siendo usada por una o varias

transacciones. Cuando una transacción necesita utilizar una página, la fija en el slot. En este trabajo, el manejo de pines corre por parte del submódulo Pin Manager. Para cada página un contador indica por cuántas transacciones está siendo usada. Se incrementa al acceder y se decrementa cuando es liberada.

- **latch**: Sirve para insertar un nuevo registro y para que el RecLSN se mantenga ordenado con respecto a las operaciones sobre la página. Esa variable apunta al último evento del log de ARIES que realizó una modificación en la página. También, por simplicidad, va a estar representado por un Latch Manager propio. Se obtiene un latch al modificar la página (insert, update o delete), y se libera luego de escribir la entrada del evento en el log de ARIES. No se obtiene ningún latch al leer.
- **lock**: Bloqueo estándar de páginas hecho por el Lock Manager principal. Puede ser utilizado para optimizaciones (si una transacción tiene bloqueados una gran porcentaje de registros pertenecientes a una misma página, le conviene bloquear toda la página). En este trabajo este tipo de bloqueo no va a ser realizado.

La documentación de ARIES muestra los pasos a seguir para evitar deadlock entre lock de registros y latch de páginas (esto aplica también a las páginas de los índices):

1. **update / delete**: Se bloquea el registro de manera exclusiva y luego se obtiene un latch la página. Si alguien ya tenia el latch de esa página, por el funcionamiento del algoritmo, se sabe que no va a pedir el lock de algún registro bloqueado antes de liberar tal latch.
2. **insert**: Se obtiene un latch de la página y se obtiene el ID del nuevo registro. Entonces, se procede a bloquear el ID de manera exclusiva para insertar el registro. Este bloqueo es CONDICIONAL. Si falla, se libera el latch y se intenta bloquear el ID de manera exclusiva INCONDICIONAL. Una vez que se obtenga ese lock, se pide de nuevo un latch de la página y se verifica que ese ID no haya sido usado (justo antes de bloquearlo INCONDICIONAL). Si fue usado, se libera el lock de tal ID y se vuelve repetir toda la operación.

Cuando se desea acceder o modificar una tabla, se le pide al Buffer Manager que traiga las páginas correspondientes a memoria, en caso de no encontrase allí con anterioridad. Estas páginas serán marcadas mientras se esté operando con ellas y luego se liberarán para que sean removidas en caso de necesitar más memoria.

El Buffer Manager contiene métodos para obtener una página, liberarla, crearla, borrarla, saber si se encuentra en memoria y guardar las páginas que fueron modificadas. Para la mayoría de ellos, luego de realizar las acciones necesarias se llama al Disk Space Manager para que persista los resultados.

3.5. Ejecutor

El Ejecutor tiene como objetivo, llamar al analizador para descomponer la sentencia SQL en partes más fáciles de procesar; luego realizar la ejecución y devolver el resultado de la consulta propiamente dicha. Esto se realiza desde el Servidor, quien obtiene las sentencias que son ejecutadas por el cliente.

El hecho de existir manejo de índices, hace que la ejecución de cláusulas WHERE (tanto en el select, update como delete) intente acceder por ellos. En el caso de no ser posible, entonces recién ahí se procede a recorrer toda la tabla.

Por simplicidad, no se contemplan joins (no se realiza producto cartesiano).

Las sentencias que soporta la aplicación son las siguientes:

```
INSERT INTO tabla (col1, col2...) VALUES (valor1, valor2...);
INSERT INTO tabla (col1, col2...) SELECT...;
UPDATE tabla SET col1 = expresion WHERE expresionWhere;
SELECT col1, expresion1... FROM tabla WHERE expresionWhere;
DELETE FROM tabla WHERE expresionWhere;
CREATE TABLE tabla (col1 NUMERIC/CHAR(XX)...);
DROP TABLE tabla;
BEGIN TRANSACTION;
COMMIT TRANSACTION;
SAVEPOINT nombre;
ROLLBACK nombre;
ROLLBACK TRANSACTION;
CRASH;
CHECKPOINT;
ISOLATION nivelAislamiento;
```

El nivel de aislamiento puede ser Read Uncommited, Read Commited, Repeatable Read y Serializable. Más información sobre los mismos aparece en la sección correspondiente al Lock Manager.

3.6. Recovery Manager

El Recovery Manager es el modulo encargado de proveer robustez a una base de datos. Su función consiste en darle las propiedades de atomicidad y durabilidad a las transacciones del motor [RAMA03].

Entre las técnicas disponibles para realizar este cometido, las más conocidas son Shadow Paging [SHAWIK], [RAMA03] y Write Ahead Logging [WALWIK], [RAMA03]. Uno de los objetivos de este trabajo es mostrar el funcionamiento de un sistema de recuperación basado en los algoritmos de ARIES [ARIES]. Estos utilizan una estrategia WAL, la cual es menos costosa en términos de memoria.

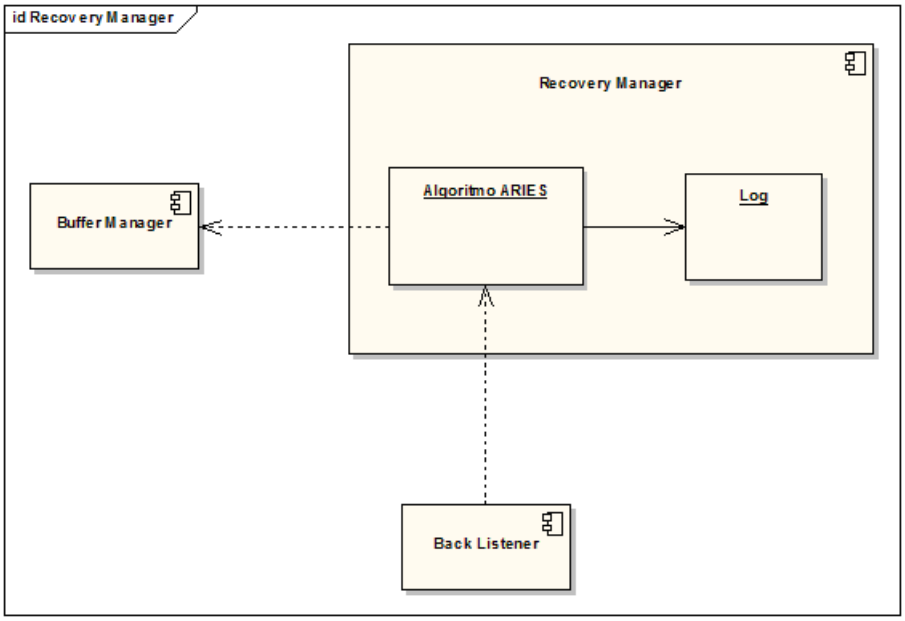

Figura 4: Arquitectura del Recovery Manager de Kanon

Cada evento que ocurre en la base de datos, se guarda en un log, indicando qué evento es y parámetros asociados al mismo para poder rehacer dicha operación o deshacerla en caso de ser necesario. El archivo de log es de formato creciente, y debe ser guardado en medio persistente cada vez que una transacción hace commit, o cuando se realiza un checkpoint de la base de datos [RAMA03].

Cuando se modifica una página, ya sea de una tabla o de un índice, el estándar de ARIES recomienda que se guarde un evento indicando qué página fue modificada y cuáles bytes cambiaron dentro de la misma. Teniendo en cuenta los fines académicos de este proyecto, se decidió separar la operación

de modificación en valores más lógicos. Se indica si se trata de la inserción de un registro, modificación o borrado del mismo, y de la misma manera para los registros de los índices. Esto permite conocer mejor el funcionamiento del motor, ya que una operación de inserción se verá reflejada en el log con un evento de inserción, junto al identificador del nuevo registro y los valores correspondientes a cada columna de la tabla afectada (y el agregado de los registros correspondientes a cada índice asociado).

Siguiendo las referencias de ARIES, cada evento tiene un identificador propio, el cual es basado en su posición dentro del archivo de log. Luego, cada transacción toma nota del último evento realizado, y cada página sabe el identificador del último evento que realizó una modificación sobre la misma. Cuando una transacción realiza commit se toma de este hecho con un evento de commit, y cuando termina (ya sea por commit o un aborto), se marca una finalización de ésta en el log. Cuando se realiza un aborto de una transacción, se deshacen los cambios hechos en orden inverso, y los eventos CLR toman de esto, por si luego hay que volver a deshacer esos cambios.

Cuando ocurre una falla y se cae el sistema, al volver a iniciarse, es necesario que aquellas modificaciones que no fueron guardadas de manera persistente sean rehechas, para que no se pierdan estos cambios. También es necesario que las transacciones que se encontraban en curso durante la caída y no hicieron commit deshagan sus cambios, para que parezca como si nunca hubieran existido. Para esto ARIES propone que la recuperación se divida en tres fases: ANALISIS, UNDO y REDO y explica los pasos a seguir en cada fase. Este modulo realiza la recuperación de la misma manera, y se detalla cuando ocurre cada paso. Sin embargo, se decidió desviarse de la implementación procedural propuesta, por una orientada a objetos.

Para asegurarse que los cambios hechos por las transacciones estén en una memoria persistente, existe el concepto de Checkpoint. Éste fuerza la escritura del log a disco así como de aquellas páginas modificadas desde la última vez que fueron escritas. Las bases de datos suelen tomar un Checkpoint de manera periódica. Pero para este motor se decidió no hacer eso, para poder ver el funcionamiento del sistema de recuperación en caso de una falla. Sin embargo, si se agregó la funcionalidad de poder realizar un Checkpoint de manera manual, y también de poder forzar una caída del sistema de manera manual (ambas con sentencias SQL propias de esta base de datos).

De la misma manera que en los demás módulos, se agregó el soporte para savepoints y transacciones anidadas. Las modificaciones para soportar a esta última se realizaron según los algoritmos de ARIES/NT [ARIESNT]. Como los savepoints son entidades lógicas de una transacción, no fue nece-

sario ningún cambio al sistema de recuperación para soportarlos.

Hay eventos que pueden ocurrir dentro de una transacción para los que es deseable que no se deshagan, aunque la misma sea abortada. Para realizar esto se implementaron las Nested Top Actions [ARIES] cuya funcionalidad es justamente ésta y en ARIES se explica qué cambios hay que cometer para soportarlas.

3.7. Transaction Manager

Este modulo tiene la finalidad de administrar la ejecución de todas las transacciones en curso. Tiene una estructura la cual almacena todas las transacciones que se están corriendo sobre el Server. Para un cliente dado se puede ver si actualmente tiene una transacción en curso. Ya que por cada cliente, se va a tener una thread del servidor ejecutando sus operaciones. Las operaciones de la transacción de un cliente no interfieren con las transacciones de los otros clientes.

Una transacción corresponde a un thread, éstas no pueden ser suspendidas y resumidas en otro thread.

El servidor ejecuta concurrentemente transacciones de cada conexión con un cliente. Cada conexión se asocia a un thread del servidor y para cada uno de estos threads puede haber una transacción en curso o ninguna.

Para cada sentencia (línea de comando que envía el cliente) se sabe si esta debe ser corrida dentro de una transacción o no. Por ejemplo, las instrucciones DML deben serlo, pero para la instrucción que realiza un checkpoint no es necesario. Luego, aquellas que lo necesiten van a ser tratadas por el ejecutor como una transacción propia si no existía ninguna en curso.

Se tomó esta decisión para que en casos donde una instrucción simple trabaje sobre un conjunto de datos, efectúe todas sus acciones o ninguna.Por ejemplo al hacer un update con un set, se van a modificar todos los registros en cuestión o ninguno.

Lo que hace el ejecutor es fijarse si hay una transacción en curso, cuando recibió la sentencia, si no la hay la crea (a lo que llamamos Transacción Automática) , ejecuta la sentencia y luego se fija en un flag si la transacción donde ejecuto la instrucción es automática o no, si lo es tiene que finalizarla.

Es posible crear transacciones explicitas del cliente cuando se envíe un Begin Transaction, seguido por una serie de sentencias y finalizando con un Commit o Rollback, y también se tendrán transacciones implícitas cuando

el cliente envíe comandos simples como Insert ... el ejecutor se va a encargar de ejecutar esta sentencia como Begin Transaction, luego Insert ... y finalizara con un Commit (o un Rollback en caso de haberse lanzado alguna excepción).

Las transacciones tienen un número como identificador, el cual va siendo incrementado atómicamente. Además, al inicio del sistema, se verifica el log para empezar con el siguiente valor al numero mas alto de las transacciones del log para que los eventos de las nuevas transacciones no se confundan con las ya grabadas.

Las propiedades ACID de las transacciones se obtienen de la siguiente manera:

- Atomicidad: Cuando ocurre un error o se desea abortar la transacción, el Recovery Manager basado en ARIES, a través de su log, se encarga de realizar el rollback y restaurar las modificaciones hechas por cada sentencia dentro de la transacción a su estado anterior.

- Consistencia: Como se comenta en [RAMA03] en la sección 18.1.1, los usuarios del sistema son responsables de mantener la consistencia de las bases de datos afectadas. Este motor no soporta restricciones de integridad, como claves primarias (o unicidad en los valores de las columnas) y claves foráneas.

- Aislamiento: El Lock Manager se encarga de las garantías de serialización de las transacciones que se ejecutan de manera concurrente, y también se encarga de los bloqueos para aquellas que desean acceder o modificar objetos (registros, tablas) al mismo tiempo.

- Durabilidad: Así como con la atomicidad, el Recovery Manager se encarga de la recuperación de las transacciones activas al momento de ocurrir una caída del sistema. También se encarga de rehacer las acciones de aquellas transacciones cuyos cambios no fueron guardados en un almacenamiento no volátil.

En la sección del Lock Manager se comenta sobre el interlineado de transacciones y el Schedule formado por las mismas.

En vistas de dar una mayor profundidad a los temas de transacciones, se han agregado dos extensiones de ARIES para proveer de mayor funcionalidad a este motor, sin dejar de lado los fines educacionales del mismo:

3.7.1. Transacciones anidadas

En [ARIESNT] se comenta como dar soporte de transacciones anidadas a un sistema de recuperación basado en ARIES, indicando los cambios en el sistema de log y en las tablas de transacciones, se decidió realizar tales cambios en este motor académico para que futuros interesados puedan conocer funciones avanzadas de una base de datos concurrente. Los cambios en el Transaction Manager implican que para cada thread, no existe una referencia a una transacción, sino una lista de transacciones, la lista se encuentra ordenada de la transacción de mas alto nivel a aquella mas reciente y profunda en el anidamiento.

Cuando se crea una transacción anidada, ésta hereda el nivel de aislamiento de la transacción padre (y claro está, se ejecutan en el mismo thread).

El método de abortar la transacción fue reemplazado por dos métodos, uno aborta la última transacción (La más anidada), volviendo a la transacción padre, y el otro aborta todas las transacciones del thread. El método de commit realiza la operación solo para la transacción mas anidada, y luego el log se encarga de unir esa operación de commit con la transacción padre, por si esta luego realiza un rollback. A la transacción padre se le actualiza su último LSN con esta operación de commit de la hija.

En las secciones del Lock Manager y Recovery Manager se comentan los cambios hechos para que ambos soporten transacciones anidadas.

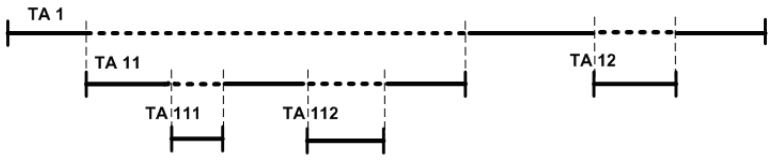

Figura 5: Transacciones anidadas según el tiempo de ejecución

Cabe destacar que cuando se crea una transacción hija, la transacción padre es suspendida, pues todas las sentencias nuevas del thread corresponderán a la transacción hija hasta que esta aborte o haga commit. Luego, no es necesario ocultar los cambios de la transacción hija al padre pues éste se

encuentra suspendido, y una transacción no puede tener más de una transacción hija en el mismo instante.

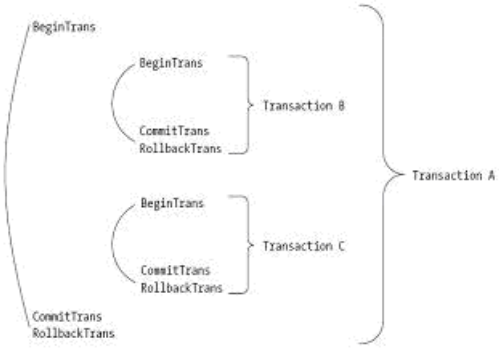

Figura 6: Transacciones anidadas como operaciones

3.7.2. Savepoints

Un savepoint permite establecer un punto significativo dentro de una transacción y permite deshacer los cambios hechos desde ese punto en adelante (siempre y cuando no se haya terminado la misma). El uso de estos savepoints está expuesto a la interfaz de usuario en forma de sentencias como se muestran en el apartado del analizador.

Para el usuario, los savepoints toman nombres amistosos. Luego estos se asocian con el ultimo LSN (evento de log, ver apartado de ARIES) de la transacción en curso. Si establece un savepoint con un nombre ya tomado, el anterior es borrado. Entre distintas transacciones puede haber savepoints de igual nombre. Esto incluye transacciones anidadas, pues no se puede volver a un savepoint de una transacción padre (No se encuentra soportado).

Aprovechando la capacidad de savepoints, cuando se ejecutan sentencias dentro de una transacción explícita, antes de cada una se marca un savepoint de manera automática, así si ocurre una excepción en la ejecución de la misma, se realiza un rollback hasta ese savepoint. Esto excluye excepciones por dead lock, pues ahí se abortan todas las transacciones del thread, para liberar los locks que tuvieran tomados.

3.8. Index Manager

Este modulo fue creado en vistas de mejorar la concurrencia entre transacciones así como la performance de las sentencias que consultan las tablas.

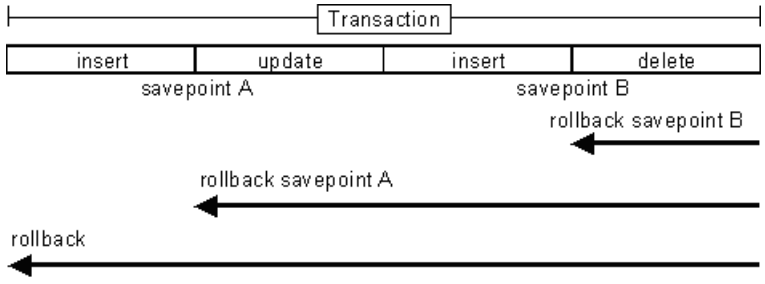

Figura 7: Uso de savepoints

La finalidad es que cuando se realice una consulta con una expresión donde se iguale una columna con un valor, se va a poder usar el índice correspondiente de la columna. Luego se van a recorrer los registros de la tabla que existan en ese índice, evitando la necesidad de recorrer todos los registros, y poder lograr mayor concurrencia en el caso que hubiera otra transacción que realice otra consulta utilizando una expresión tal que los registros representados en su índice no se encuentren compartidos con la primer transacción (ejemplo, una expresión donde se iguala la misma columna pero con otro valor tal que su número de hash es distinto).

Como los datos de las tablas no se encuentran ordenados, éstos índices son del tipo *unclustered*, y como existe una entrada para cada registro de la tabla (según el hash que corresponda), también son del tipo *densos*.

Es necesario mencionar que este motor no tiene implementadas claves primarias ni restricciones de unicidad en las columnas de una tabla. Suelen ser conocidos entonces como índices secundarios. En el capítulo 8.4 de [RAMA03] se explican y comparan las diferentes propiedades que puede tener un índice.

Para cada índice se utilizan buckets con overflow. (Explicación de índices hash con overflow en el capítulo 10 de [RAMA03]). Todos los valores posibles de la columna son asignados a un numero hash, y se agrega una entrada en el bucket que corresponda a tal par [columna, hash]. Al no ser dinámico se puede dar el caso que una entrada de hash tenga muchos buckets, pero aun así va a ser mas concurrente que recorrer toda la tabla. Además, el objetivo del trabajo práctico es mostrar una implementación y funcionamiento del sistema ARIES. Estos índices son solo un agregado para mejorar un poco la concurrencia y proveer nivel aislamiento serializable sin tener que bloquear

toda la tabla.

El índice hash también es útil para las tablas del catalogo, pues cuando se desea obtener una tabla no es necesario recorrer toda la tabla de tablas buscándola.

El sistema de ARIES nos va a permitir que los índices sean tan transaccionales como las tablas comunes. Se agregaron los eventos (lógicos) de actualización de índices en el log para realizar REDO y UNDO de los mismos en caso de ser necesario, y se agregó al sistema de Lock Manager bloqueo sobre los índices (ver más detalles en la sección correspondiente de dicho administrador). Por último, cuando se modifica algún Bucket, de la misma manera que las páginas, se toma un Latch para que darle orden a las modificaciones concurrentes.

3.9. Lock Manager

Este modulo tiene como objetivo administrar los bloqueos tanto de lectura como de escritura. Se utiliza un bloqueo pesimista [RAMA03] en vistas de mostrar su diseño, implementación y uso de manera educativa, y por ser más simple que el mantenimiento de versiones que suelen tener los bloqueos optimistas.

En este diseño se tomo la decisión de implementar el protocolo de 2PL, sobre el esquema RLOCK / WLOCK, o sea sobre bloqueos ternarios. En principio se pensó tomando en cuenta las condiciones de los capítulos 18 y 19 de [RAMA03]. Tiene soporte para bloqueo de granularidad fina (a nivel de registro) lo que permite una mayor concurrencia entre transacciones.

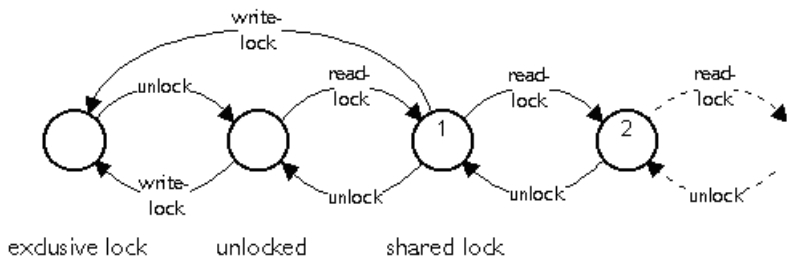

Figura 8: Modelo de Two phase locking

Se han implementado los cuatro niveles estándar de aislamiento transaccional: READ UNCOMMITTED, READ COMMITTED, REPEATABLE

READ y SERIALIZABLE [SQL92], [RAMA03]. El bloqueo de elementos dentro de una transacción se adhiere al protocolo 2PL estricto para los primeros dos niveles [S2PLWIK] y al protocolo 2PL riguroso para los últimos dos niveles [R2PLWIK]. Según se muestra en [BERNTS] y en [FRACCR], se mantiene la correctitud semántica, y los distintos niveles muestran un tradeoff entre performance (menos bloqueos / mayor concurrencia) e inconsistencias posibles en el aislamiento de las transacciones.

El Lock Manager está diseñado de la siguiente manera:

Su estructura cuenta con métodos para bloquear un elemento (se utiliza el ID del elemento), para desbloquearlo, y para saber si un elemento se encuentra bloqueado.

Cuando se desea bloquear un elemento también se especifica si se desea un bloqueo exclusivo o compartido. En nuestro esquema, el bloqueo exclusivo es utilizado para bloquear objetos para su posterior escritura, y el bloqueo compartido para bloquear elementos que van a ser leídos. Estos métodos se ejecutan de manera sincronizada. Esto es, sólo una transacción (thread/conexión) puede estar ejecutándolo a la vez. Fue necesario pues las estructuras donde se guardan los Locks son compartidas por todas las conexiones.

Al igual que con el resto de los módulos relacionados con transacciones, éste administrador fue luego modificado para dar soporte a transacciones anidadas y Savepoints.

Cuando se aborta hasta un Savepoint, se desean liberar aquellos Locks obtenidos luego del mismo. Para ello, se conoce cuál fue el último LSN antes de crear un Lock en la transacción, y existe un método que libera aquellos Locks mayores al LSN relacionado con el Savepoint.

Cuando se crea una transacción hija, esta hereda los locks de sus ancestros. Cada Lock guarda qué transacción lo creó. Pero el algoritmo para saber si un elemento ya se encuentra bloqueado por una transacción itera también por aquellos pertenecientes a transacciones ancestras. Cuando una transacción hija es abortada, todos sus locks son liberados de la misma manera que fuera una de nivel alto, pero cuando realiza commit, todos sus locks no son liberados sino delegados a la transacción padre. Luego, a medida que vayan haciendo commit se irán pasando hasta llegar a la de más alto nivel.

Un caso especial es cuando una transacción ancestra tiene un Lock compartido sobre un elemento, y la descendiente desea actualizar ese Lock a uno exclusivo (para realizar una modificación o borrado del elemento). En

este caso ambos Locks se mantienen en las estructuras, para que el bloqueo exclusivo sea liberado en caso de un aborto, o sea delegado al padre en caso de commit. Si el padre era quien poseía el Lock compartido, el mismo será actualizado a exclusivo (como ya había un bloqueo exclusivo, se sabe que ninguna otra transacción posee bloqueos compartidos sobre el elemento afectado).

3.9.1. Deadlock

Para el tratamiento de DeadLocks, se optó por usar los algoritmos de prevención en vez de detección una vez que ocurrieron. Esto es para mantener la simplicidad del trabajo. Como trabajo futuro se podría implementar un algoritmo basado en deadlock detection. Se diseñó una interfaz, la cual es usada por el administrador cada vez que se desea bloquear un elemento, para verificar si puede haber un conflicto entre la transacción que desea bloquear con aquellas dueñas de Locks sobre el elemento en cuestión.

4. Implementacion

A continuación se describen los temas de implementación, separados por módulos para una mejor comprensión.

4.1. Plataforma

El lenguaje elegido para desarrollar el RDBMS fue Java dada la gran cantidad de plugins existentes para hacer interfaces graficas, así como la simplicidad de su lenguaje, poder que resulta propicio para la realización de trabajos académicos de este estilo. Utilizamos la versión de Java 5.0 que trae mejoras con respecto a las anteriores versiones para simplificar la programación de la aplicación en los aspectos de diseño y concurrencia. También destacamos el haber hecho la implementación usando Patrones de Diseño, entre los que podemos mencionar: Factory, Strategy, Decorator, Singleton y Abstract factory [GAMMA95].

Entre los plugins utilizados, usamos un CVS remoto para subir las fuentes y ad-ministrar las versiones de los diferentes módulos [CVSSSL]. Usamos el Jigloo para crear la interfaz grafica del cliente. [JIGLOO]

Para ciertas funcionalidades del sistema utilizamos paquetes especializados, a saber:

- ZQL: paquete que provee análisis de sentencias SQL y la creación de estructuras Java que corresponden a las mismas [ZQL].

- SC: librería que realiza el coloreo de palabras clave de una sentencia SQL para facilitar la lectura de la misma en el cliente [SYNCOL].

- Commons-Collections: conjunto de diferentes estructuras de datos e implementaciones especializadas (Conjunto, Lista, Mapa) [COMCOL].

El sistema sigue el protocolo Cliente - Servidor, en donde el motor de base de datos actúa en un modo pasivo, escuchando por un puerto a que se conecte un cliente. Éste le va haciendo pedidos al servidor, el cual los procesa y devuelve resultados.

Ambos programas, al inicializar, abren puertos de escucha utilizando los métodos provistos por Java y delegando al mismo los detalles de conexión.

El intercambio de mensajes se hace a través de estos Sockets abiertos, y utilizan un formato de texto simple. Los pedidos del cliente hacia el servidor serán las sentencias SQL escritas por el usuario, mientras que los resultados en sentido inverso son los mensajes de respuesta del servidor. Esta respuesta puede ser:

- un mensaje de información explicando el resultado de la sentencia, cuando esta se ejecuta de manera exitosa.
- un texto conteniendo los resultados en un formato de tabla cuando la sentencia es una consulta.
- un mensaje con una descripción de error cuando ocurre uno al procesar la sentencia.

Existen también dos puertos traseros en el servidor, los cuales sirven para darle información al cliente administrador sobre el estado de las estructuras de Lock, y de los eventos que van siendo guardados en el Log. Éstos son provistos para que los usuarios puedan saber, en tiempo real, los eventos ocurridos dentro del servidor y poder entender los mecanismos del mismo.

4.2. Disk Space Manager

Cada página ha sido implementada de manera tal que contiene un número fijo de re-gistros, y un número fijo de bytes (64KB). La cantidad de registros depende de las columnas de la tabla, es decir, dependiendo de la cantidad de columnas y tamaño de las mismas. Este formato de página hace que no existan registros guardados en más de una página, o sea, cada página guarda una cantidad entera de registros. Esto trae la simplicidad en el motor de transacciones, ya que los eventos de un registro siempre corresponden a una sola página, pero trae la contrariedad de no poder crear tablas cuya longitud de registro sea mayor al tamaño especificado de una página. Esta decisión fue tomada con fines de simplificar el motor de transacciones. El tamaño de cada página va a variar dependiendo si se aprovechan los 64KB o, por el hecho de no cortar registros entre 2 páginas, si es menor al mismo.

La nomenclatura elegida para el nombramiento de los archivos en disco es la siguiente:

- NumeroTabla.NumeroPagina para identificar un bloque que representa a una página. No se utiliza nombre de tabla porque, como el motor es case sensitive (diferencia los nombres en mayúscula y minúscula), esto

ocasionaba problemas para quellas plataformas en donde el sistema de archivos I/O es case insensitive.

- NumeroTabla.NumeroColumna.NumeroHash.NumeroBucket para los buckets de los índices.

Las páginas guardadas en binario, se mantienen en los slots del Buffer Manager, y son transformadas en objetos entendibles por el motor al momento de la lectura y escritura de registros. Al no haber tipo de datos de longitud variable, revisando en el catálogo la tabla de columnas es posible saber cuál es la longitud de cada registro correspondiente a una determinada tabla.

Para saber si un registro se encuentra libre u ocupado, se pensaron dos maneras:

1. guardar al principio de la página un conjunto de bits, en donde si se encuentra encendido el n-ésimo bit, quiere decir que el n-ésimo registro está ocupado.
2. que el primer bit de cada registro sea un valor de verdad que indique si el mismo está libre o no.

La manera (a) tiene la ventaja que es fácil saber si una página está vacía o llena, pues todos los bits se encuentran consecutivos y si están todos desactivados o activos indican (respectivamente) los estados mencionados. Por otro lado, si en el log de ARIES se guardan los cambios a nivel de bytes, entonces es conveniente que el indicador se encuentre junto con el resto del contenido del registro, como en la manera (b), para guardar en un solo evento los bytes cambiados por la modificación del registro. Como los eventos del log son lógicos, se decidió utilizar la opción (a). Esta política está implementada en la clase ArregloBits.

Además, se crearon métodos que convierten los elementos Java, que representan los tipos soportados por el motor, a arreglos binarios de tamaño fijo, determinado por el tipo, y que contienen una representación del valor del elemento. Luego se guardan de manera consecutiva estos arreglos de bytes, respetando el orden según las columnas de la tabla. Notar que no se guarda ninguna información respecto a qué tipo es o la longitud del arreglo de bytes, pues esta información se obtiene al consultar los tipos de las columnas de la tabla correspondiente.

4.3. Buffer Manager

En el momento de realizar el análisis previo a la implementación de este proyecto, hemos observado que en la mayoría de los casos iba a existir una

pequeña cantidad de páginas con un enorme número de referencias. Por este motivo es que, en nuestro diseño, decidimos incluir una estructura que nos permita mantener una cierta cantidad de páginas en memoria sin la necesidad de leerlas de la memoria secundaria constantemente. Esta estructura encargada de traer páginas de memoria secundaria a memoria principal es el Buffer Manager, el cual para este propósito, posee una colección de páginas llamadas **Frames** (también denominadas **Buffer Pool**). La cantidad de Frames en el pool del Buffer Manager es configurable.

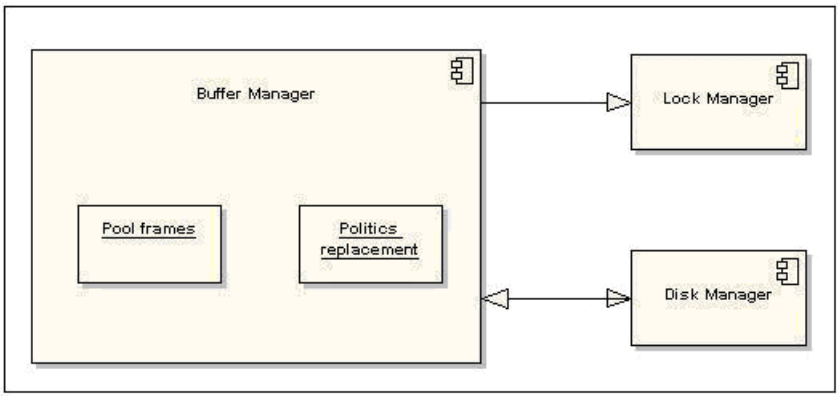

Figura 9: Implementación del Buffer Manager de Kanon

El pool de Frames está implementado con un mapa que tiene el ID de una página como clave y la página propiamente dicha como valor. Este mapa se encuentra sincronizado, pues es accedido por varios threads, y sincronizar las operaciones principales del Buffer Manager era demasiado costoso. El tamaño del mapa es configurable, de manera de poder realizar comparaciones entre distintos tamaños para conocer diferencias de performance al utilizar más memoria en el pool.

Como ya mencionamos, la tarea fundamental del Buffer Manager es traer páginas de disco y brindarle a las clases superiores los métodos necesarios para mantenerlas en memoria hasta que estos digan lo contrario. Sin embargo estas páginas liberadas solo serán removidas del Buffer y grabadas nuevamente en el disco, en el caso de haber sido modificadas, cuando no existan más frames libres en el Buffer y se solicite alocar una página que no se encuentra en el mismo en ese momento.

Para realizar este procedimiento utilizamos un algoritmo de remoción de páginas. La implementación del Buffer Manager utiliza una Interfaz para acceder a la política de reemplazo. Distintas implementaciones de esta interfaz proveen algoritmos **FIFO**, **LRU**, **MRU**, **LFU**, **MFU** y **remoción al azar**.

Cada uno de ellos tienes sus ventajas y desventajas según el procedimiento que se esté realizando. Por ello se recomiendan usar híbridos (por ejemplo, una mezcla entre **LFU** y **LRU**). Sin embargo, en vistas de mantener el trabajo simple y entendible, no se han implementado algoritmos avanzados de reemplazo de páginas. La política de reemplazo a utilizar por la aplicación es configurable.

Cada página tiene la propiedad de saber si esta sucia o no, es decir, si se modificó desde la última vez que se guardó en disco. Con esta información, necesaria para el funcionamiento de ARIES, se utiliza la llamada **Dirty-Pages table**, la cual provee información sobre las páginas que han sido modificadas durante el uso normal de la base de datos. Esta tabla también es usada cuando se recupera el sistema luego de una caída. Cada entrada de la tabla consiste de dos campos: PageID y RecLSN. Durante el procesamiento normal, cuando una página se fija en un slot del Buffer Manager para modificarla, se registra el LSN del próximo evento a ser guardado en el log (o sea, el LSN del fin del log). Este valor indica desde qué punto en el log pueden haber modificaciones sobre la pagina que probablemente no se encuentren en la versión de almacenamiento estable de la misma (útil para saber hasta qué punto se encontraba actualizada la página en caso de suceder una caída). Los contenidos de esta tabla son incluidos en el registro de checkpoint. Luego, cuando se recupera el sistema y se lee dicho registro, se recrea esta tabla y es modificada durante la fase de Análisis. El menor valor de RecLSN en la tabla indica el punto de partida para la fase de Redo al momento de recuperar el sistema.

Las páginas cuentan con una bandera que indica si han sido modificadas. La misma es marcada cuando se realiza una inserción, actualización o remoción de un registro de tal página. Luego, antes de remover la página de memoria, el Buffer Manager la persiste (llamando al Disk Space Manager) en caso de estar marcada, para evitar que se pierdan los cambios realizados.

4.4. Ejecutor

El Server va leyendo cada sentencia que le llega desde el cliente. Por cada una debe: analizarla, ejecutar el comando correspondiente y devolver el resultado o error.

Una de las principales características de nuestras clases XQL (las cuales son una translación directa de las clases provistas por el analizador), es que se pueden agregar nuevos componentes fácilmente sin tener que realizar modificaciones a la clase. Toda clase XQL implementa XStatement con sus 2 métodos: `zqlToXql()` y `execute()`. El primero permite dado un objeto

de la clase ZQL (la devuelta por el analizador) copiar y procesar sus componentes a su par en XQL. El método `execute()` realiza la ejecución propia de cada statement. De esta forma obtenemos un diseño de código mucho más escalable, permitiendo su modificación fácilmente. Esta implementación es tomada del patrón de diseño Strategy [GAMMA95], cada XQL sabe que hacer, por lo que el ejecutor se desliga de esos detalles.

Desde el Servidor, se debe verificar si ya hay una transacción abierta o si se debe crear una transacción de manera implícita para la sentencia que se está analizando. Una transacción tiene dos formas de iniciarse, de manera explícita o implícita. De manera explícita, el cliente solicita mediante una sentencia (BEGIN TRANSACTION) que desea abrir una transacción, y para cerrar la transacción el cliente debe indicar si aborta o realiza commit del conjunto de sentencias. De manera implícita, el cliente envía una sentencia sin tener abierta una transacción. Para este caso, el servidor inicia una transacción para esa sentencia, y una vez ejecutada esa transacción procede a realizar commit sin necesidad de confirmación por parte del cliente (en el caso de que se haya lanzado una excepción, se aborta la transacción).

4.5. Recovery Manager

Para la implementación orientada a objetos del módulo, existen entidades que representan a cada evento posible de los existentes en el log, y a medida que se va leyendo el mismo, estos objetos son creados y dentro de cada uno existe un código propio para las tres fases correspondientes. El propósito de la fase de Análisis es saber cuáles transacciones se encontraban en curso cuando se produjo la caída, y que páginas contenían modificaciones que no fueron guardadas en disco. La fase de Redo se encarga de reproducir las modificaciones de todas las transacciones involucradas en aquellas páginas no persistidas, y la fase de Undo consiste en luego deshacer los cambios por las transacciones que no habían hecho commit. Entonces, los tres ciclos principales en el modulo se encargan de leer el log y crear los eventos, y cada evento sabe qué hacer según el ciclo que se encuentre. Esto permite una mejor compresión del algoritmo de recuperación, así como también permite realizar modificaciones al mismo o a eventos en particular sin que ello afecte a las demás componentes que interactúan en el proceso.

Los eventos disponibles en este proyecto para el modulo de recuperación son:

- Evento de inserción de un registro en una página de una tabla: contiene la transacción a la que pertenece, el LSN del evento anterior en la

transacción, el identificador del registro insertado y los valores de cada columna de la tabla afectada.

- Evento de modificación de un registro en una página de una tabla: contiene la transacción a la que pertenece, el LSN del evento anterior en la transacción, el identificador del registro modificado, los valores originales de las columnas modificadas, y los nuevos valores que ocuparán las mismas en ese registro.

- Evento de borrado de un registro en una página de una tabla: contiene la transacción a la que pertenece, el LSN del evento anterior en la transacción, el identificador del registro borrado y los valores originales de las columnas de la tabla para ese registro.

- Evento CLR de inserción de un registro en una página de una tabla: contiene la transacción a la que pertenece, el LSN del evento anterior en la transacción, el identificador del registro insertado y un conjunto de UndoNextLSN (modificación hecha para el soporte de transacciones anidadas, detallada más adelante en el texto).

- Evento CLR de modificación de un registro en una página de una tabla: contiene la transacción a la que pertenece, el LSN del evento anterior en la transacción, el identificador del registro modificado, los valores originales de las columnas afectadas antes por la modificación y un conjunto de UndoNextLSN (modificación hecha para el soporte de transacciones anidadas, detallada más adelante en el texto).

- Evento CLR de borrado de un registro en una página de una tabla: contiene la transacción a la que pertenece, el LSN del evento anterior en la transacción, el identificador del registro borrado, los valores originales de las columnas del registro y un conjunto de UndoNextLSN (modificación hecha para el soporte de transacciones anidadas, detallada más adelante en el texto).

- Evento de inserción de un registro en una página de un índice: contiene la transacción a la que pertenece, el LSN del evento anterior en la transacción, el identificador del registro-índice insertado y el identificador del registro de la tabla referenciado.

- Evento de borrado de un registro en una página de un índice: contiene la transacción a la que pertenece, el LSN del evento anterior en la transacción y el identificador del registro-índice borrado.

- Evento CLR de inserción de un registro en una página de un índice: contiene la transacción a la que pertenece, el LSN del evento anterior en la transacción, el identificador del registro-índice insertado y un

conjunto de UndoNextLSN (modificación hecha para el soporte de transacciones anidadas, detallada más adelante en el texto).

- Evento CLR de borrado de un registro en una página de un índice: contiene la transacción a la que pertenece, el LSN del evento anterior en la transacción, el identificador del registro-índice borrado, el registro de la tabla al cual referenciaba y un conjunto de UndoNextLSN (modificación hecha para el soporte de transacciones anidadas, detallada más adelante en el texto).
- Evento de commit de una transacción: contiene la transacción a la que pertenece, el LSN del evento anterior en la transacción y un conjunto con los identificadores de los registros bloqueados al momento del commit.
- Evento de rollback de una transacción: contiene la transacción a la que pertenece y el LSN del evento anterior en la transacción.
- Evento de fin de una transacción: contiene la transacción a la que pertenece y el LSN del evento anterior en la transacción.
- Evento de Begin Checkpoint: no contiene ningún parámetro. Es el evento referenciado por el registro maestro al realizarse un Checkpoint, y a partir del cual se empezará a leer el log cuando se inicia el sistema.
- Evento de End Checkpoint: contiene una lista con las transacciones en curso al momento del Checkpoint. Para cada transacción se toma su identificador, último LSN, estado, conjunto de UndoNextLSN y registros bloqueados al momento de la colecta. Tambien contiene una lista con las páginas que no fue ron persistidas. Para cada página se toma su identificador y LSN del último evento que la modificó.

Las modificaciones de transacciones anidadas mantienen el esquema orientado a objetos ya diseñado. Se agrega el evento que vincula al commit de una transacción hija con la transacción padre, para que en el caso de tener que realizar un Redo o Undo de la última, también se lo haga de la primera. Este evento se llama CHILD-COMMITTED, toma el identificador de la transacción hija y el último LSN de la misma, y es insertado como un evento de la transacción padre. También se realizan los cambios mencionados en el algoritmo para que una transacción no tenga un solo UndoNextLSN sino un conjunto de ellos durante el aborto, y se irá tomando el de mayor LSN para seguir un orden cronológico inverso.

Para agregar el soporte de Nested Top Actions, se agregó el evento sugerido de DUMMY-CLR. Cuando se comienza con una NTA dentro de una transacción, se toma nota del último LSN de la misma, y luego al finalizar,

se escribe el evento DUMMY-CLR. Éste referencia a la LSN mencionada, para que si la transacción es deshecha, los eventos correspondientes a la NTA sean salteados (y no deshechos).

4.6. Transaction Manager

El Transaction Manager cuenta con dos mapas de transacciones. Uno asocia cada transacción con su identificador, para poder obtenerlas rápidamente a partir del mismo. El segundo mapa asocia cada thread con las transacciones en curso. Como se mencionó anteriormente, una lista es usada para las tomar nota de las transacciones anidadas que pueda haber.

Existen métodos para consultar si hay una transacción en curso para un thread en particular (sin parámetros asume que se pregunta por el thread que ejecuto el método), para iniciar una transacción en un thread, para abortarla (tanto los dos métodos de aborto mencionados anteriormente, como un método para abortar una transacción hasta un savepoint dentro de la misma) o realizar commit. Para el inicio o fin de cada transacción, el transaction manager se comunica con el lock manager para libere los locks correspondientes (si es necesario) y con el Recovery Manager para que se escriban los eventos de fin de transacción en el mismo.

La profundidad de las transacciones anidadas solo está limitada por la memoria del sistema.

Estructura de una Transacción:

- Su identificador único: para asignarle un id único a cada transacción se utiliza un numero que va creciendo monotónicamente.
- El estado de la misma (en curso, abortada o terminada).
- El thread donde se ejecuta la transacción.
- Un campo timestamp de inicio que marca el momento en el cual empezó la transacción, usado para los algoritmos de Prevención de Dead-Lock.
- El LSN del último evento guardado en el log correspondiente a la transacción.
- El conjunto de UndoNextLSN. Complementando lo dicho en la bibliografía de ARIES con el soporte para transacciones anidadas, el Undo Next LSN es el LSN del próximo evento a ser leído durante el rollback

de la transacción. Si durante el rollback se incluyen transacciones hijas de ésta, sus respectivos Undo Next LSN se irán guardando en este conjunto, y para el próximo evento a abortar, se elige el de mayor número.

- La transacción padre de esta, en caso de haber una.

4.7. Index Manager

Se crea un índice para cada columna de cada tabla. No existen las sentencias DDL de manejo de índices (CREATE INDEX, DROP INDEX, etc.)

Se usa el índice correspondiente al primer elemento del WHERE que sea una igualdad. Para mayor información, consultar la API correspondiente a la ejecución de sentencias que contienen consultas (SELECT, UPDATE Y DELETE). Se recorren solo los registros obtenidos por el índice asociado, o todos los registros de la tabla en caso de no poder usar ningún índice. Formato de un bucket: una lista con los ID de los registros cuyo valor en la columna especificada concuerda con el hash del bucket.

Hay un arreglo de bits que indican los lugares de la lista libres (pueden quedar huecos en el medio de la lista, pero nos evitamos reordenar y el iterador va a ser inteligente y saltea los huecos) Cuando se llena un bucket se crea otro (de la misma manera que una página).

El iterador que devuelva el índice va a ser de aquellos registros cuyo valor de hash en la columna concuerde con el hash del valor especificado. O sea, Si pido aquellos registros cuya columna 1 sea igual a "valor1z "valor2"tiene el mismo hash que "valor1.[en]tonces los índices me van a dar no solo aquellos registros cuyo valor en la columna sea "valor1"sino también los que tienen "valor2". Una optimización podría ser guardar el valor propio en el Bucket, eso haría que solo se devuelvan los registros de valor "valor1z aumentaría la concurrencia.

El numero de hash va a ser modulo 8 para evitar que haya muchos buckets con un solo elemento.

4.8. Lock Manager

En la implementación del Lock Manager se usaron 3 estructuras de datos:

Un mapa con ID de clave y un conjunto de Locks por valor:

Este mapa permite saber qué Locks existen actualmente para un elemento dado (especificado según el ID). Pueden existir varios locks compartidos para un elemento, pero si existe un bloqueo exclusivo, va a ser único en el conjunto, salvo que haya locks compartidos pertenecientes a transacciones ancestras de aquella que tiene el bloqueo exclusivo.

Un mapa con ID de clave y una cola de pedidos como valor:

Este mapa guarda las transacciones encoladas (usando un orden FIFO) que desean adquirir un Lock para un elemento dado (especificado según el ID). De esta manera se evita inanición al querer bloquear un objeto.

La estructura de un pedido de bloqueo que se encola contiene los siguientes datos:

- La transacción que realiza el pedido.
- El thread perteneciente a esa transacción (y que será suspendido mientras no se consiga bloquear el objeto)
- Valor que indica si el bloqueo deseado es exclusivo o compartido.

Un mapa por cada Transacción con ID de clave y Lock como Valor:

Este mapa guarda los Locks adquiridos por cada transacción. Es usado para obtener un acceso más rápido a los Locks de la misma, y para mantener la consistencia del sistema de bloqueos. Cuando se desea bloquear un objeto puede ocurrir que:

1. No se encuentre bloqueado: En este caso, se procede al bloqueo efectivo del elemento. En caso de haberse realizado un bloqueo compartido, el administrador comprueba si la cola de conexiones en espera para bloquear el mismo elemento no está vacía, y en ese caso de toma la primer conexión en espera y le avisa que proceda con su bloqueo (el cual va a ser exitoso en caso de ser un bloqueo compartido y va a volver a esperar en caso de ser uno exclusivo).
2. Se encuentre ya bloqueado por la misma conexión (ya sea de la misma transacción o de alguna ancestra): El bloqueo se deja tal cual estaba, salvo el caso que hubiera un bloqueo compartido y ahora se desee uno exclusivo. Entonces se procederá a hacer una actualización del Lock, pero sólo después de que otras conexiones que también tuvieran bloqueos compartidos sobre el mismo objeto los hayan liberado.

3. Se encuentra ya bloqueado por otra transacción (una o varias): Aquí de nuevo se divide en casos si se desea un bloqueo compartido o exclusivo.

    a) Si se desea un bloqueo compartido y los bloqueos existentes también son compartidos: Si no existe ningún pedido de bloqueo exclusivo encolado, entonces se realiza el bloqueo y se agrega al conjunto de Locks de ese ID. Si existe alguno, éste pedido se encola al final, esto es para que el pedido de bloqueo exclusivo que se encuentra encolado no sufra de inanición.

    b) Si el bloqueo existente es exclusivo, se suspende la transacción hasta que tal bloqueo sea liberado. Cabe notar que por la cola FIFO, no va a llegar ningún pedido de bloqueo en el medio, salvo que el mismo sea una actualización de algún bloqueo existente compartido al modo exclusivo.

    c) Si se desea un bloqueo exclusivo, entonces se encola el pedido. Se espera hasta que todos los Locks sobre el elemento sean liberados, así como que ocurran todos los bloqueos encolados con anterioridad.

Al desbloquear un objeto, si lo que se tenía sobre tal objeto era un bloqueo exclusivo (se consulta el mapa local para cada conexión para averiguar cuál era el Lock sobre el objeto), entonces luego de desbloquearlo se consulta la cola de espera de tal elemento para que la siguiente conexión proceda a la adquisición del Lock sobre el objeto.

Si se intenta desbloquear un elemento no bloqueado por la conexión, se lanzará una excepción. Para poner en espera a las conexiones que desean adquirir un bloqueo sobre un elemento ya bloqueado, se utilizan los métodos de suspensión de Threads provistos por Java 1.5.

También se ha implementado un decorador (Del patrón de diseño Decorator [GAMMA95] del administrador el cuál guarda en un registro cada vez que se desea bloquear o desbloquear un elemento. Este registro luego sirve para mostrar como queda el Historial de eventos de bloqueo durante un determinado tiempo.

Cuando se le pide una tabla al Catálogo, éste la decora con una implementación que realiza el bloqueo y desbloqueo de elementos de manera automática, evitando agregar esta complejidad al ejecutor. Antes de ejecutar las operaciones de inserción, actualización y eliminación, se bloquea el elemento a modificar de manera exclusiva, y si no existe ninguna transacción en curso, se desbloquea luego de la operación.

En caso de haber una transacción activa, los bloqueos y desbloqueos se realizaran según el nivel de aislamiento correspondiente:

- READ UNCOMMITTED: No se realiza un bloqueo compartido antes de realizar una lectura. Se realizan bloqueos exclusivos antes de modificaciones o inserciones. Estos bloqueos son liberados cuando se termina la transacción (o delegado a la transacción padre si existe).
- READ COMMITTED: Se realizan bloqueos compartidos antes de realizar lecturas. Estos son liberados inmediatamente luego de la misma. Se realizan bloqueos exclusivos antes de modificaciones o inserciones. Estos bloqueos son liberados cuando se termina la transacción (o delegado a la transacción padre si existe).
- REPEATABLE READ: Se realizan bloqueos compartidos antes de realizar lecturas y exclusivos antes de modificaciones o inserciones. Todos los bloqueos son liberados cuando se termina la transacción (o delegado a la transacción padre si existe).
- SERIALIZABLE: Ídem anterior, pero además, en una lectura se bloquea el índice correspondiente o toda la tabla en caso de no utilizarse ninguno, para evitar "lecturas fantasmas".

En todos los niveles, al realizar inserciones se bloquean todos los índices de las columnas de la tabla (o en caso de modificaciones, los índices afectados), para que las lecturas de nivel SERIALIZABLE sepan sobre estas modificaciones o inserciones y se eviten las mencionadas "lecturas fantasmas".

Se han implementado distintos algoritmos de prevención, como ser Wound - Wait, Wait - Die, Caution Waiting, y una implementación simple que indica que nunca ocurre DeadLock. Al levantar el servidor se puede elegir qué algoritmo será utilizado. Por omisión se toma Caution Waiting.

Algunas de estas implementaciones utilizan la fecha de comienzo de las distintas transacciones involucradas para decidir cuál va a ser la víctima.

4.8.1. Deadlock

Para el tratamiento de DeadLocks, se optó por usar los algoritmos de prevención en vez de detección una vez que ocurrieron. Esto es para mantener la simplicidad del trabajo. Como trabajo futuro se podría implementar un algoritmo basado en deadlock detection. Se diseñó una interfaz, la cual

es usada por el administrador cada vez que se desea bloquear un elemento, para verificar si puede haber un conflicto entre la transacción que desea bloquear con aquellas dueñas de Locks sobre el elemento en cuestión.

5. Pruebas sobre el sistema

El objetivo de las pruebas fue principalmente verificar la correctitud en toda la funcionalidad soportada por el servidor y el cliente.

A continuación, ponemos en detalle una de la pruebas realizadas para observar como se va modificando el log a medida que se ejecutan las sentencias.

5.1. Pruebas de funcionamiento

Prueba de inserción:
La siguiente prueba muestra el resultado de ejecutar un insert sobre una tabla existente:

```
insert into proveedores values('Sancor', 'Cordoba 5400', 46532456);
```

El Log registra los siguientes cambios:

3976	INSERT	IdTx: 4	PrevLSN:0	
	proveedores	Pag: 0	Reg: 0	
		Col: 0		Sancor
		Col: 1		Cordoba 5400
		Col: 2		46532456
4472	PREPARE	IdTx: 4	PrevLSN:4356	
	Lock:	proveedores	Pag: 0	Reg: 0
4540	END	IdTx: 4	PrevLSN:4472	

El primer evento corresponde a la inserción de un registro a la tabla proveedores de la transacción 4, con LSN 3976 crea el registro 0 en la pagina 0. También se muestra en el log el contenido en cada columna.

Luego se hace un insert de registros de índices por cada columna, los cuales no se visualizan en el log. Por eso el LSN 4472 tiene PrevLSN:4356 y no 3976.

El evento PREPARE, indica el commit de la transacción y muestra los locks obtenidos por la misma.

El evento END, muestra el cierre de la transacción.

El resultado de ejecutar un select sobre la tabla proveedores muestra lo siguiente:

nombre	**direccion**	**telefono**
Sancor	Cordoba 5400	46532456

Prueba de rollback:

Muestra el resultado de ejecutar una transacción explicita en la cual se realiza la inserción de dos registros sobre la tabla proveedores y luego se hace un rollback.

```
begin transaction;
insert into proveedores values('Parmalat', 'Corrientes 2400', 46843221);
insert into proveedores values('La Serenisima', 'Callao 1030', 44327000);
rollback transaction;
```

El log registra los siguientes cambios:

4604	INSERT	IdTx: 6	PrevLSN:0	
	proveedores	Pag: 0	Reg: 1	
		Col: 0		Parmalat
		Col: 1		Corrientes 2400
		Col: 2		46843221
5100	INSERT	IdTx: 6	PrevLSN:4984	
	proveedores	Pag: 0	Reg: 2	
		Col: 0		La Serenisima
		Col: 1		Callao 1030
		Col: 2		44327000
5596	ROLLBACK	IdTx: 6	PrevLSN:5480	
Lectura: 5596	ROLLBACK	proxLSN:5616		
Lectura: 5480	INSERT_INDEX	proxLSN:5596		
Lectura: 5364	INSERT_INDEX	proxLSN:5480		
Lectura: 5248	INSERT_INDEX	proxLSN:5364		
Lectura: 5100	INSERT	proxLSN:5248		
5868	CLR_INSERT	IdTx: 6	PrevLSN:5784	
		4984		
	proveedores	Pag: 0	Reg: 2	
Lectura: 4984	INSERT_INDEX	proxLSN:5100		
Lectura: 4868	INSERT_INDEX	proxLSN:4984		
Lectura: 4752	INSERT_INDEX	proxLSN:4868		
Lectura: 4604	INSERT	proxLSN:4752		
6196	CLR_INSERT	IdTx: 6	PrevLSN:6112	
		0		
	proveedores	Pag: 0	Reg: 1	
6272	END	IdTx: 6	PrevLSN:6196	

La operación begin inicializa el PrevLSN en 0. Los dos primeros eventos corresponden a la inserción de dos registros en la tabla proveedores.Luego se muestra el evento de ROLLBACK de la transacción 6 y se empieza a

recorrer la misma leyendo el log desde la última operación hasta la primera (aquí se muestra los eventos sobre los índices).

Para cada inserción abortada se escribe en el log el evento CLR_INSERT (según lo especificado por ARIES). Este muestra la transacción a la que pertenece, los UndoNextLSN, y al final el registro que se esta removiendo. Los eventos CLR_INSERT_INDEX no son mostrados en la pantalla del log, para mejor lectura.

Y por último se muestra el evento END de finalización de la transacción 6.

6. Conclusiones

Entre las cosas más destacables de haber realizado este trabajo fue la gran cantidad de cosas que pudimos aprender durante el transcurso del proyecto sobre lo que es un RDBMS y los aspectos relacionados con la recuperación a un estado consistente luego de una caída.

Llegamos a la conclusión de que la familia de algoritmos de ARIES para la recuperación y bloqueo es un método muy ingenioso y eficaz que es muy usado actualmente en la industria por muchos de los motores relacionales de uso comercial y también otros para uso académico. Este método encara con precisión todos los problemas reales de concurrencia y recuperación que están presentes en los RDBMS actuales que se usan ámbitos industriales.

Aprendimos de un sistema como ARIES, lo implementamos y nos dimos cuenta del buen funcionamiento y que tan complejo puede llegar a ser el manejo de los problemas de concurrencia y recuperación. En un principio creímos que este método sólo era de aplicación académica pero luego vimos que era ampliamente utilizado y en varios casos extendido o personalizado de maneras interesantes (como paralelización de la recuperación o uso de versionado en las páginas).

Este trabajo va servir de ayuda para comprender y observar el funcionamiento de un RDBMS ya que fue pensado con fines académicos reflejando de la forma más sencilla posible el funcionamiento interno del motor hacia el usuario. Mostramos como se implementa, desde cero, ARIES con un modelo académico simple, modular, orientado a objetos, donde quien esté interesado en el tema puede ver el funcionamiento de ARIES en su totalidad para una mayor comprensión.

La metodología que plantea el paper de ARIES se acopla bien con temas complejos como anidamiento de transacciones, recuperación de caídas o savepoints.

7. Trabajo futuro

En este trabajo construimos un motor de base de datos relacional con propósitos académicos Que utiliza el método de ARIES para el manejo de transacciones y el aspecto de la recuperación ante caídas imprevistas del sistema.

Esta implementación no contempla algunas temas que son muy interesantes en la actualidad como las transacciones distribuidas y transacciones de larga duración; cuestiones abiertas y de investigación en muchas universidades.

Una de las principales ideas a futuro es extender este trabajo implementando una versión distribuida de ARIES (como por ejemplo, D-ARIES) y que preserve todas las ventajas de ARIES como partial rollbacks , repeatable history, etc.

Otro aspecto a mejorar podría ser la etapa de Análisis que se hace durante la recuperación. El cambio consistiría en paralelizar algunas de las etapas de recuperación de ARIES para ganar eficiencia, otra forma de optimizar sería hacer que en la etapa de Redo, éstos sean selectivos, es decir evitar rehacer transacciones que luego se van a deshacer en la etapa de Undo. De esta manera se lograría ganar eficiencia en la parte de recuperación.

Mientras se hacen las fases de recuperación, que vaya escuchando nuevos pedidos para que sea más eficiente el arranque.

Soportar media recovery (soporte para copias de seguridad y recuperación frente a errores ocurridos si se rompe el disco).

Deadlock detection en vez de deadlock prevention. Detalles sobre deadlock prevention se pueden encontrar en la secciòn 19.2.2 de [RAMA03].

Agregar un conector estándar JDBC (acceder a la DBMS con un cliente JDBC).

Mejorar el sistema de índices en base a la mejoras descrip-tas en [ARIESIM], [ARIESKVL] y [ARIESLHS].

Agregar soporte para suspender una transacción y luego reanudarla y seguir ejecutándola.

Referencias

[ALTIBAS] Jung, Kwang-Chul - Lee, Kyu-Woong *Implementing Storage Manager in Main Memory DBMS ALTIBASE*

[ARIES] Mohan, C. et al *ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging*

[ARIESIM] Mohan, C. - Levine, F. *ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging*

[ARIESKVL] Mohan, C. *ARIES/KVL A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes*

[ARIESLHS] Mohan, C. - Rothernlel, K. *ARIES/LHS: A Concurrency Control and Recovery Method Using Write-Ahead Logging for Linear Hashing with Separators*

[ARIESNT] Mohan, C. - Rothernlel, K. *ARIES/NT: A Recovery Method Based on Write-Ahead Logging for Nested Transactions*

[ARIESRH] Mohan, C. *Repeating History Beyond ARIES*

[BDFCEN] Fundamentals of Databases Course
http://www-2.dc.uba.ar/materias/bd

[BERNTS] Bernstein, Arthur J. et al *Using Transaction Semantics to Increase Performance*

[BERN87] Bernstein, P. - Hadzilacos, V. - Goodman, N. *Concurrency Control and Recovery in Database Systems, Addison Wesley. 1987*

[BRUNI02] Bruni, P. et al *Application Recovery Tool for IMS and DB2 Databases A Data Recovery Guide*

[COMCOL] Jakarta Commons Collections Homepage
http://jakarta.apache.org/commons/collections/

[CVSSSL] An SSL connection method for CVS (sserver)
http://home.arcor.de/rolf_wilms/cvsssl/cvsssl_help.html

[DCFCEN] Computer Science Department
http://www.dc.uba.ar

[DERBY] DERBY Homepage
http://db.apache.org/derby/papers/recovery.html

[FCEN] Facultad de Ciencias Exactas y Naturales
http://www.fcen.uba.ar

[FRACCR] Franklin, Michael J. *Concurrency Control and Recovery*

[GAMMA95] Gamma, E. et al *Design Patterns, Elements of Reusable Object-Oriented Software*

[GRAY80] Gray, J. et al *The Recovery Manager of the System R Database Manager*

[JIGLOO] JIGLOO Homepage *http://www.cloudgarden.com/jigloo/*

[KANONOP] Leggieri, L. - Berlin, J. - Cabas, V. - Pippia, F.
KANON dbms Operations manual

[KB230785] Microsoft Knowledge Base
http://support.microsoft.com/kb/q230785

[LEAP] LEAP Homepage *http://leap.sourceforge.net/*

[MINIBAS] MiniBase Homepage
http://www.cs.wisc.edu/coral/mini_doc/project.html

[MINIREL] Minirel Homepage
http://www.ceid.upatras.gr/courses/minibase/minibase-1.0/documentation/html/minibase/logMgr/report/main.html

[ORACLE] Oracle technology network
http://www.oracle.com/technology/deploy/availability/htdocs/Flashback_Overview.htm

[RAMA03] Ramakrishnan, R. - Gherke, J. *Database Management Systems, 3rd Ed. Mc Graw-Hill, 2003*

[R2PLWIK] Rigurous Two-Phase Locking
http://en.wikipedia.org/wiki/Rigorous_two-phase_locking

[SHAWIK] Shadow Paging *http://en.wikipedia.org/wiki/Shadow_paging*

[SQL92] Isolation Levels
http://en.wikipedia.org/wiki/Isolation_(computer_science)

[STONE76] Stonebraker, M. - Wong, E. - Kreps, P. - Held, G. *The Design and Implementation of INGRES, ACM Transactions on Database Systems, 1976*

[SYNCOL] JAPISOFT JSyntaxColor
http://www.japisoft.com/syntaxcolor/

[S2PLWIK] Strict Two-Phase Locking
http://en.wikipedia.org/wiki/Strict_two-phase_locking

[ULLMAN88] Ullman *Principles of Database and Knowledge Base Systems, Computer Science Press, 1988*

[WALWIK] Write-Ahead Logging
http://en.wikipedia.org/wiki/Write_ahead_logging

[ZQL] ZQL Homepage *www.experlog.com/gibello/zql/*