Base de datos

1er. Cuatrimestre 2006

Proyecto: Minibase
Informe: Diseño Detallado

17 de noviembre de 2006

Número de Grupo:
Nombre del Grupo: ?

Integrantes

Apellido y Nombre	L.U.	Mail
Leandro Groisman	222/03	gleandro@gmail.com
Fernando Rodriguez	516/00	ferrod20@gmail.com
Guillermo Amaral	522/98	Guillermo.AMARAL@total.com
Facioni Francisco	004/04	fran6co@fibertel.com.ar

Instancia	Corrector	Nota
Entrega		
Reentrega		

Comentarios del corrector:

Índice

1. Introducción a Minibase

Minibase es un sistema de administración de bases de datos desarrollado para uso educativo. Si bien la documentación oficial ostenta tener un parser y un optimizador, la implementación sobre la cual trabajamos no implementa estas componentes. Sin embargo, sí tiene un buffer pool, mecanismos de almacenamiento tales como heap files e índices, con estructuras de tipo árbol B+, y un sistema de administración del espacio en disco.

El objetivo de Minibase no es sólo obtener un sistema de administración de bases de datos que sea funcional, sino también tener acceso a los componentes principales para facilitar su estudio. En las siguientes secciones del informe, vamos a analizar esas componentes, y finalmente estudiaremos la ejecución de una consulta en esta DBMS.

1.1. Historia y evolución de Minibase

Ramakrishnan, junto con Mike Carey, quería extender Minirel, que era una pequeña RDBMS creada como un proyecto de curso de universidad. Pretendían hacerlo en C++ y agregarle control de concurrencia, control de recuperación, optimizador de consultas, más métodos de evaluación de operadores relacionales y herramientas de diseño. De allí surgió Minibase.

La mayor parte del código fue escrita por alumnos del curso; distintos grupos se dedicaron a distintas componentes, mientras que otros se ocuparon de integrar las partes. Con el tiempo, el código fue refinado y extendido, y nuevos grupos de personas escribieron utilitarios, entre los que se incluyen herramientas de visualización.

Lamentablemente ese código no esta disponible, cuando se pide el código de Minibase en la pagina de Ramakrishnan, se recibe una versión en Java con problemas de compilación y severamente incompleta.

2. Objetivos

El objetivo principal de este trabajo practico es poner en funcionamiento Minibase. Para ello tuvimos que analizar el código en profundidad y realizar varios tests caso por caso para poder encontrar todos los problemas.

Como objetivos secundarios están la creación de un nuevo tipo de Join y de un utilitario para la creación de nuevas relaciones. También se extendió el catalogo para soportar el atributo Primary Key.

3. BufferManager

3.1. Descripción general

El Buffer Manager es la estructura encargada de traer páginas de memoria secundaria a memoria principal sin la necesidad de leerlas desde la memoria secundaria constantemente. Para este propósito, posee una colección frames, que son los encargados de alojar las páginas en la memoria principal. Este mecanismo permite traer páginas desde el disco y brindarle a las clases superiores los métodos necesarios para mantenerlas en memoria hasta que digan lo contrario.

Las páginas removidas del Buffer serán grabadas en el disco sólo en caso de haber sido modificadas. Para elegir qué página remover se utiliza un algoritmo de remoción de páginas entre las implementaciones de Minibase: Clock, LRU y MRU.

Para marcar cuando una pagina fue modificada, estas cuentan con una marca indicando si han sido modificadas (dirty). Luego, antes de remover la página de memoria, el Buffer Manager la persiste (llamando al DiskSpace Manager) en caso de estar marcada, para evitar que se pierdan los cambios realizados.

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 (pin) mientras se esté operando con ellas y luego se liberarán(unpin) para que sean candidatas para la remoción.

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 DiskSpace Manager para que persista los resultados.

3.1.1. Diseño interno, estructuras

Buffer pool El buffer pool es una colección de frames (secuencia de páginas de tamaño fijo que se encuentran en memoria principal)

Descriptores Adicionalmente, se mantiene un arreglo de descriptores, uno por cada frame. Cada descriptor tiene los siguientes campos:

- numero de pagina (PageId)
- pin count (entero)
- dirtybit (bool)

Esto describe la página que es guardada en el frame. Una página es identificada por un número de página que es generado por DM cuando la página es alojada, y es único en toda la base de datos.

Directorio de búsqueda (tabla de hash) Para identificar que páginas de disco están en el Buffer Manager se utiliza un directorio. Éste consiste en una tabla de hash que se encuentra en memoria principal y que utiliza un arreglo de <numero de pagina, numero de frame>. Dado un número de página, se puede aplicar la función de hash para encontrar el bucket correspondiente a esa página. Luego se recorre el bucket, buscando la página. Si se encuentra, tendrá un frame asociado, caso contrario, la página no está en el buffer pool.

Cuando se solicita una página, el buffer manager debe hacer lo siguiente:

- Buscar en el buffer pool (usando la tabla de hash) para ver si contiene la pagina solicitada.
- Si la pagina no está en el buffer pool:
  1. Elegir un frame en dónde se guardará ésta página, utilizando las políticas de remoción (Clock, MRU o LRU)
  2. Si el frame elegido tiene una página que ha sido modificada(dirty= true), se deben escribir los datos de esa página en el disco, mediante DM.
  3. Se lee la página solicitada desde disco(otra vez, llamando al DM) y se ubica dentro del frame elegido.
  4. Se borra la entrada para la página vieja en el directorio del Buffer Manager (tabla de hash) y se inserta una entrada para la página nueva.
- También, se actualiza la entrada para este frame en el arreglo de descriptores.
- Se pincha(pin) la página requerida.

3.2. Clases principales y sus protocolos principales

3.2.1. FrameDesc

Es una clase que describe un frame. Describe cada página que está en el buffer pool: el número de página en el archivo, si esta marcada como modificada(dirty) y sus pin count. El pin count cambia cuando se pincha (pin) o despincha (unpin) una página.

3.2.2. BufHTEntry

Es el nodo que se utiliza para representar una lista simplemente encadenada, que es guardada en los buckets de la tabla de hash. Guarda información sobre:

- el número de página
- el número de frame en donde está esa página
- y tiene un puntero a la próxima entrada.

3.2.3. BufHashTbl

Una tabla de hash para mantener el seguimiento de las páginas que están en el buffer pool(memoria principal) Sus funciones son insertar, obtener y remover páginas de la tabla de hash.
Función de hash: pageNo.pid % HTSIZE
El tamaño de la tabla es fijo (establecido en 20 por defecto)
Cada bucket mantiene una lista enlazada de BufHTEntrys, NULL significa que no hay nada.
Posee métodos para buscar, insertar y eliminar.

3.2.4. BufMgr

Los atributos miembro son básicamente los mencionados anteriormente más un replacer y el buffer físico(un arreglo en dónde se guardan físicamente las páginas en memoria principal)
Tiene métodos para guardar explícitamente las páginas en disco, para obtener una página y marcarla, para desmarcarla y para eliminarla del disco.

3.2.5. Replacer

Esta interfaz se asocia con el algoritmo de reemplazo. Describe que frame debe ser elegido para su remoción. Minibase implementa esta interfaz con las clases Clock, LRU y MRU.

3.3. Interacción con otros componentes

El Buffer Manager interactúa fuertemente con el Disk Manager, al cual le pide que lea y escriba páginas en disco.

4. DiskManager

4.1. Descripción general

Se encarga de crear, abrir, cerrar o borrar una base de datos, que no es otra cosa que un archivo en el disco. Provee diversos métodos para acceder a las páginas en el disco y manejar el Space Map, que sirve para saber cuáles páginas están siendo utilizadas. También permite mantener el nombre y la ubicación de los archivos lógicos de la base, manejando el Directorio de Archivos.

4.2. Clases principales y sus protocolos principales

4.2.1. Clase Page

Funciona como un buffer para guardar una página en memoria al momento de leer o escribir de disco, o para su utilización temporal.

4.2.2. Clase DBHeaderPage

Se encarga de reflejar el formato de una página cualquiera del Directorio de Archivos y provee ciertas funcionalidades tanto para modificar como para leer los datos de estas páginas.

4.2.3. Clase DB

Esta es la clase que se encarga de ser la interfaz del Disk Manager. Exporta la funcionalidad de crear, abrir, cerrar o borrar una base de datos. La unidad mínima física de lectura y escritura en Minibase es una página y, por lo tanto, el archivo está dividido en varias páginas. El tamaño de una de ellas está dado por una constante, por lo que esta clase no provee un método para inicializarlo; en este caso, es de 1024 bytes.

4.2.4. Space Map

El mapa de bits representa, usando un bit por página, si una página de la base de datos está siendo utilizada o no. Como en Minibase las páginas ocupan 1024 bytes, entonces podremos llevar registro de 8 * 1024 páginas, por cada página que utilice el Space Map. Por supuesto, las páginas usadas por el Space Map son marcadas como usadas.

El bit enésimo significa:

- 0: La página enésima esta libre.
- 1: La página enésima esta utilizada.

Como esta estructura no es una lista enlazada y como la primer página que no pertenece al Space Map en sí es justamente la primera inmediata que le sucede, entonces el Space Map como estructura de datos no puede crecer una vez creado. Por ende, la cantidad de páginas del Space Map es fija, la cantidad de páginas que se pueden reservar queda acotada y el tamaño de la base de datos queda acotado como consecuencia. Por ejemplo, si se desea crear una base de datos de 2000 páginas, entonces se necesitará sólo una página de Space Map para marcarlas.

4.2.5. Directorio de Archivos

El directorio de archivos es una lista simplemente encadenada de páginas, en donde se guarda la siguiente información:

- PageID de la próxima página del Directorio. En la última página, este valor es INVALID_PAGE (-1). Ocupa 4 bytes.
- Cantidad de Entries que entran en la página. Ocupa 4 bytes.
- PageID de la primer página del archivo. Ocupa 4 bytes.
- Nombre del archivo. Ocupa 52 bytes.
- Cantidad total de páginas de la base de datos. Este atributo figura sólo en la primera página del Directorio.

Este formato de página se encuentra reflejado en la clase DBHeaderPage, y es heredado por las clases DBFirstPage, que se encarga del formato de la primer página, y por DBDirectoryPage, que se encarga de las demás páginas del directorio.

Es bueno resaltar que tanto el Directorio como el Space Map son parte del archivo de la base de datos. Y la página 0 siempre es la primera página del Directorio.

4.3. Interacción con otros componentes

Sería útil aclarar que el Disk Manager actúa tanto como productor de datos para el Buffer Manager que como consumidor de éste último. Como productor, recibe pedidos de lectura y escritura de páginas y se las suministra o recibe del BM. Como consumidor, utiliza al BM como cache para las páginas de su estructura interna, es decir, las del Space Map y las del Directorio. Es decir: todas las funciones que recorren el Diccionario de Archivos o el Space Map utilizan al Buffer Manager para trabajar en memoria. Esto lo realizan con las funciones pinPage y unPinPage.

4.4. Ejemplo de uso

Abrir un archivo de base de datos existente con un nombre dado.

1. Abre el archivo con ese nombre para lectura y escritura.
2. Como toda base de datos tiene al menos una página, inicializa la cantidad de páginas en 1. Pide un PinPage al Buffer Manager de la página 0, pues utiliza de entrada el BM como cache. El BM, en consecuencia, le pide al DM, en modo productor que lea la página 0.
3. Una vez leída del disco la página 0, se la da al BM. El BM le devuelve la página al DM (quien actúa nuevamente en modo consumidor del cache). Ahora sí, extrae la cantidad total de páginas verdadera y la actualiza. Luego, hace un UnPinPage de la página 0 y la marca como undirty, ya que no escribió nada en ella.

5. HeapFile

5.1. Descripción general

Es un archivo desordenado. Cada registro en el archivo tiene un rid(record id) único y cada página en el archivo es del mismo tamaño. El rid es una concatenación del id de la página donde está ubicado el registro y el slot en donde se encuentra dicho registro dentro de la página. Además permite la creación de un scan para poder recorrer todos los registros de un archivo heap, repitiendo solicitudes para el próximo registro. También mantiene un registro de las páginas que tienen espacio libre para implementar la inserción eficientemente.

5.1.1. Diseño interno-estructuras

Heapfile mantiene una estructura de información sobre las páginas de datos llamada directorio de páginas Esta estructura está implementada como una lista doblemente enlazada compuesta por páginas del tipo HFPage. Cada entrada en una página del directorio de páginas, contiene la siguiente información: id de la página, espacio libre y la cantidad de registros que esa página contiene. Cada una de estas entradas, apunta a una página de datos, la cual contiene los registros. Entonces, una página de directorio almacena información sobre muchas páginas de datos.

Las páginas de datos del HeapFile también son del tipo HFPage, están implementadas como páginas con slots. Cada una de éstas páginas contiene los slots al inicio, con información sobre los registros(longitud, offset, tipo, etc) registros al final y espacio libre en el medio (si es que hay).

La primer página del directorio de páginas, es la página de cabecera(header page) para todo el heapfile. Para recordar donde está ubicada esta página, se mantiene por medio de DM, un conjunto de tuplas <nom_BD, 1er_pag_del_dir_de_datos>

5.2. Clases principales y sus protocolos principales

5.2.1. Heapfile

Heapfile tiene métodos para devolver eficientemente la cantidad de registros que tiene el HeapFile, para insertar un registro. Al crear un Heapfile, se asegura que el nombre no exista y asignándole un nombre único.

Búsqueda de un determinado registro

1. Empieza a recorrer las páginas del directorio.
2. Por cada registro de la página de directorio actual obtiene el id de la página a la que apunta.
3. Llama a BM para obtener esa página.
4. Verifica si el rid buscado coincide con el id de esta página(recordemos que el rid es una concatenación del id de página y el slot en dónde se encuentra el registro)

Inserción de registros:

1. Empieza a recorrer las páginas del directorio, buscando alguna página de datos que contenga espacio suficiente.
2. Si no encontró espacio disponible: crea una nueva página de datos (por medio del BM), crea una entrada para esta página en el directorio de páginas (ídem).
3. Inserta el registro y modifica los valores de la entrada para esa página en el directorio de páginas.
4. Persiste la página(por medio del BM).

Eliminación de un registro:

1. Busca la página de datos del registro y la página de directorio que apunta a éste.
2. Actualiza la página de directorio.
3. Si la cantidad de registros de la página de datos es cero, la elimina.
4. Borra el registro y modifica los valores de la entrada para esa página en el directorio de páginas.

5.2.2. HFPage

Representa una página dividida en slots. Se utilizan slots al principio que indican la información del registro(longitud, offset, tipo, número de slot, etc). Los registros se guardan al final. Cuando se borra un registro, se elimina el registro físicamente y en la información del slot, se establece su tamaño en un valor negativo. Por lo tanto, el número de slots en uso no se modifica. Esto se hace para no perder los números de rid(recordemos que este número está asociado a la posición del registro slot). El diseño asume que los registros se compactan cuando se realiza un borrado (no los slots). HFPage guarda información sobre el número de slots en uso y el número de bytes libres.

Insertar un registro:

1. Se fija si hay lugar disponible para el registro nuevo.
2. Busca un slot marcado como borrado (longitud negativa)
3. Si no lo encuentra, crea uno nuevo.
4. Inserta la información del slot(número de slot, longitud del registro, offset del registro, etc)
5. Inserta el registro en la página y devuelve su rid (número de pagina concatenado con número de slot)

Eliminar un registro:

1. Obtiene el número de slot por su rid.
2. Elimina el registro (hace un shift hacia la derecha, de todo lo anterior a ese registro)
3. Actualiza los offsets de todos los registros anteriores a éste.
4. Incrementa el espacio libre de la página.
5. Marca el slot como libre (establece la longitud del registro como negativa)

5.3. Interacción con otros componentes

HeapFile interactúa fuertemente con BufferManager, para obtener y guardar las páginas.

6. Catalogo

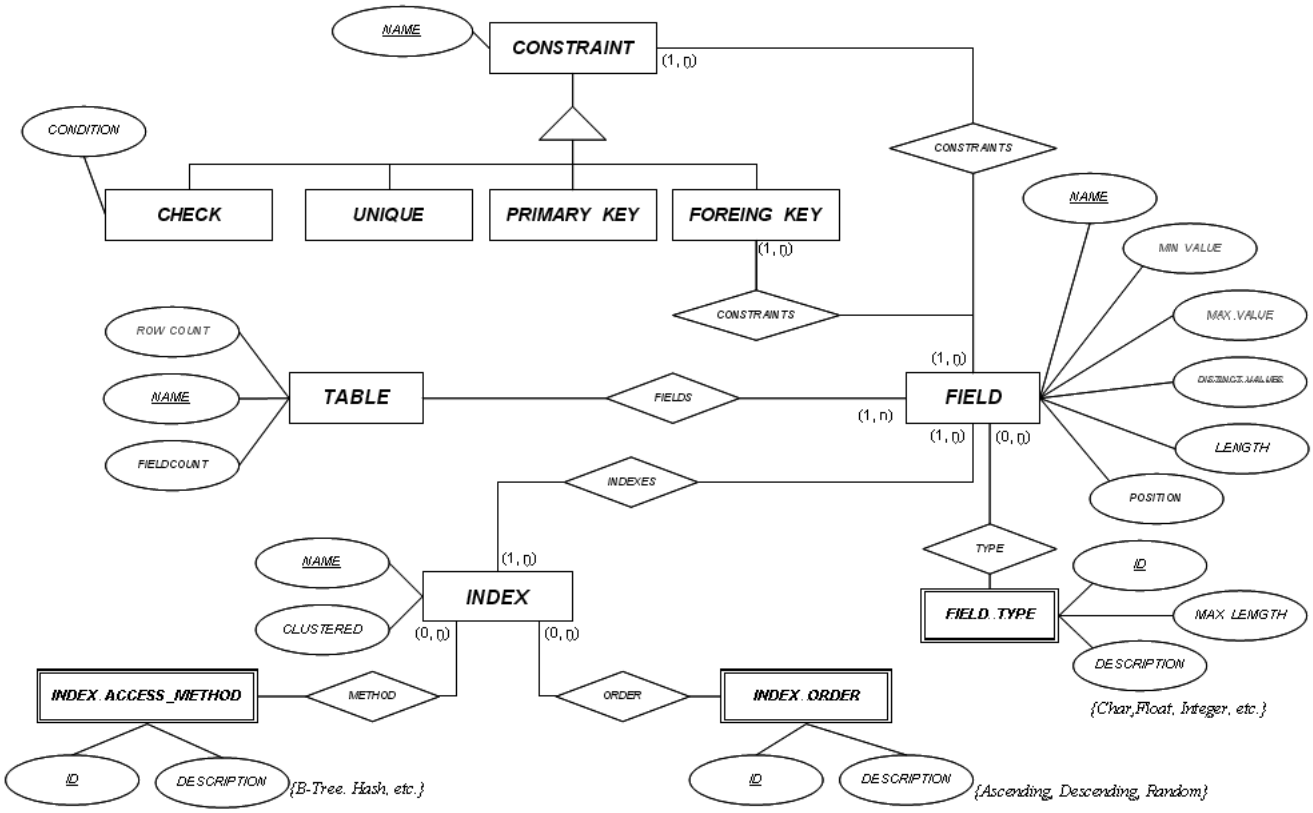
6.1. Descripción general

Este módulo se encarga de administrar el catálogo de la base de datos (ie. donde se mantiene la información de las relaciones o tablas, sus atributos y sus índices) y de proveer las siguientes funcionalidades:

1. Agregar/eliminar una tabla al/del catálogo
2. Agregar/eliminar un índice a/de una tabla
3. Proporcionar información acerca de:
   a) Una tabla
   b) Un atributo en particular
   c) Todos los atributos de una tabla
   d) Un índice en particular
   e) Todos los índices de un atributo
   f) Todos los índices de una tabla
4. Proveer la interfaz para el optimizador

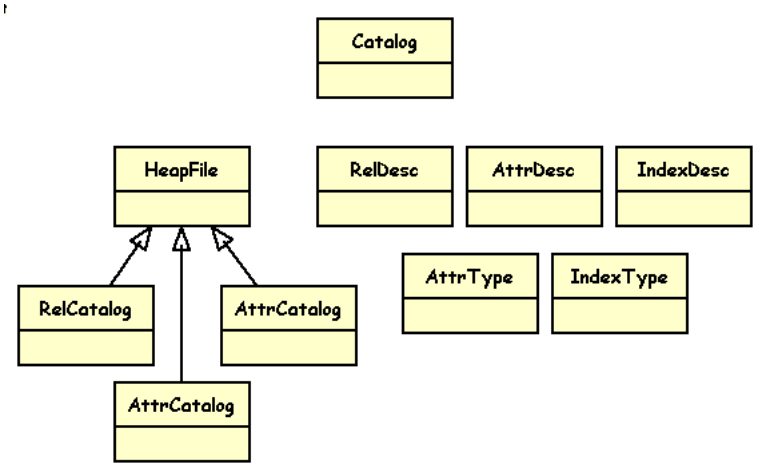
6.2. DER

(Este DER representa el modelo conceptual deseado y no el existente en el Minibase)



6.3. Clases principales y sus protocolos principales

Las clases principales del catálogo son las siguientes:



6.3.1. Catalog

La clase *Catalog* proporciona una interfaz externa para las funcionalidades antes mencionadas, delegando dichas funcionalidades en las clases *RelCatalog*, *AttrCatalog* e *IndexCatalog*.

6.3.2. RelCatalog

La clase *RelCatalog* mantiene la información sobre las tablas existentes en la base de datos. Esta clase está implementada como un heapfile (hereda de la clase *HeapFile*) extendiendo el comportamiento básico de un heapfile con la funcionalidad particular para la administración la información de las tablas. Básicamente, la información de cada tabla se resume en:

- El nombre de la tabla
- La cantidad de atributos que posee
- La cantidad de atributos sobre los cuales existe un índice
- El número total de registros que contiene
- El número de páginas en el archivo de la tabla

6.3.3. AttrCatalog

La clase *AttrCatalog* mantiene la información sobre los atributos de todas las tablas existentes en la base de datos.
De la misma forma que *RelCatalog*, esta clase está implementada como un heapfile agregando la funcionalidad particular para la administración la información sobre los atributos. Esta información se compone de:

- El nombre de la tabla a la que pertenece
- El nombre del atributo
- El offset (en bytes) dentro de un registro
- La posición dentro de los atributos de la tabla
- El tipo de dato
- La longitud máxima
- La cantidad de índices que existen sobre el atributo

- Los valores mínimo y máximo a lo largo de toda la tabla (ie. en función de los valores para el atributo de todos los registros).

Esta clase fue modificada, para el agregado de las PKs. Esto es, se agregó un campo más al catálogo donde se especifica si un atributo de una tabla forma parte (o no) de la PK de la misma. Dicho campo es de tipo integer (1 equivale a verdadero y 0 a falso). Adicionalmente se modificaron todas los métodos involucrados con el manejo de dicho catálogo para agregar el nuevo campo (read_tuple y make_tuple).

6.3.4. IndexCatalog

La clase *IndexCatalog* mantiene la información sobre los índices que existen sobre todas las tablas de la base de datos. Al igual que *RelCatalog* y *AttrCatalog*, esta clase está implementada como un heapfile. La información mantenida sobre los índices es la siguiente: el nombre de la tabla, el nombre del atributo sobre el cuál aplica el índice, el tipo de índice o método de acceso (Hash, B-Tree), el ordenamiento (ascendente o descendente), si es clustered o no, el número de claves (valores) distintas y el número de páginas del archivo de índice.

6.3.5. RelDesc, AttrDesc e IndexDesc

Para administrar la información de tablas, atributos e índices, se utilizan las clases *RelDesc*, *AttrDesc* e *IndexDesc*, las cuales modelan la especificación de una relación, de un atributo y de un índice respectivamente. Mediante estas especificaciones es posible interactuar con los catálogos respectivos (ie. obtener la información de cada catálogo así como modificarla). Debido a la extensión del catálogo para la inclusión de PKs, se extendió la clase *AttrDesc* agregándole una variable para mantener la información sobre si el atributo especificado forma parte (o no) de la PK de la tabla a la que pertenece.

6.3.6. attrInfo

Esta es una clase auxiliar utilizada para la creación de tablas. Las instancias de la misma modelan la especificación de un atributo conteniendo: el nombre del atributo, el tipo de dato y la longitud del mismo.Esta clase también fue extendida para especificar si el atributo es o no una PK.

6.3.7. Clases para manejo de excepciones

Para cada una de los servicios provistos por el catálogo se manejan diversas excepciones tanto de bajo nivel (acceso al heapfile) como de alto nivel (información lógica contenida en los catálogos).

6.3.8. AttrType

Las instancias de esta clase representan los posibles tipos de datos que puede contener un atributo (*Integer*, *Float*, *String*, etc.).

6.3.9. IndexType

Las instancias de esta clase representan los posibles tipos de índices que pueden crearse sobre una tabla (B-Tree, Hashed).

6.4. Interacción con otros componentes

El catalogo utiliza fuertemente el resto de las componentes, ya que es el punto de acceso a la base de datos. Toda acción tiene que ser revisada por el catalogo para poder extraer la información necesaria.

6.5. Ejemplos de uso

6.5.1. Crear y eliminar una tabla

Para crear una tabla se debe proporcionar el nombre y una lista con los atributos que tendrá. Por ejemplo, para crear la tabla sailors con los campos sid (PK), sname y srating se podría hacer lo siguiente:

```
List<attrInfo> attributes = new ArrayList<attrInfo>();
attributes.add(new attrInfo("sid", new AttrType(AttrType.attrInteger), 0 , true));
attributes.add(new attrInfo("sname", new AttrType(AttrType.attrString), 10 , false));
attributes.add(new attrInfo("srating", new AttrType(AttrType.attrReal), 0 , false));
attrInfo[] listInfo = new attrInfo[attributes.size()];
attributes.toArray(listInfo);
SystemDefs.JavabaseCatalog.createRel("sailors", listInfo);
```

Notar que el ante último parámetro que recibe el constructor de *attrInfo* (la longitud) solo se especifica para los atributos de tipo *String* (en el caso de los otros tipos es indistinto, ie. no se tiene en cuenta el valor pasado en dicho parámetro). También se ve en este ejemplo que el método *createRel* se aplica sobre el catálogo (instancia de la clase *Catalog*). Este a su vez, como se comentó anteriormente delegará la creación de la tabla en el catálogo de tablas (instancia de *RelCatalog*). Para eliminar una tabla solo se debe proporcionar el nombre de la misma. Por ejemplo, para borrar la tabla sailors se debería hacer lo siguiente:

```
SystemDefs.JavabaseCatalog.removeInfo("sailors");
```

6.5.2. Crear y eliminar un índice

Para agregar un índice sobre la tabla sailors se debe especificar el nombre de la tabla, el nombre del atributo sobre el que se desea agregar el índice, el tipo de índice (Hash o B-Tree) y la cantidad de buckets (este último parámetro no se utiliza actualmente). Por ejemplo, para agregar un índice de tipo B-Tree sobre el campo sname de la tabla sailors, se podría hacer lo siguiente:

```
SystemDefs.JavabaseCatalog.addIndex("sailors", "sname", new IndexType (IndexType.B_Index), 0);
```

La funcionalidad para borrar el índice aún no fue implementada, pero básicamente sería de la siguiente forma:

```
SystemDefs.JavabaseCatalog.dropIndex("sailors", "sname", new IndexType (IndexType.B_Index);
```

6.5.3. Obtener información sobre una tabla, atributo(s) o índice(s)

Para obtener información sobre una tabla específica bastará con especificar el nombre de la relación y un record donde se pasará dicha información. Por ejemplo:

```
RelDesc relDesc = null;
(2) SystemDefs.JavabaseCatalog.getRelationInfo("sailors", relDesc);
```

Otra alternativa sería interactuar directamente con el catálogo de tablas de la siguiente forma:

```
(1) RelDesc relDesc = ExtendedSystemDefs.MINIBASE_RELCAT.getInfo("sailors");
```

Notar que *SystemDefs.JavabaseCatalog* es el catalogo (instancia de la clase *Catalog*) de la base de datos, que a su vez contienen los catálogos de tablas, atributos e índices (instancias de *RelCatalog*, *AttrCatalog IndexCatalog*). Dichas instancias también son accesibles directamente de la siguiente forma:

```
ExtendedSystemDefs.MINIBASE_RELCAT
ExtendedSystemDefs.MINIBASE_ATTRCAT
ExtendedSystemDefs.MINIBASE_INDCAT
```

Dado que el catalogo (*SystemDefs.JavabaseCatalog*) delega las responsabilidades en sus catálogos particulares, las expresiones (1) y (2) son equivalentes (aunque sería más conveniente acceder siempre a través de *SystemDefs.JavabaseCatalog*). Para obtener información sobre un atributo en particular, por ejemplo el atributo srating de la tabla sailors basta con hacer lo siguiente:

```
AttrDesc attrDesc = SystemDefs.JavabaseCatalog.getAttributeInfo("sailors", "srating");
```

O bien para obtener la información sobre todos los atributos del una tabla:

```
AttrDesc  [] attrDescs = SystemDefs.JavabaseCatalog.getRelAttributes("sailors");
```

Finalmente para obtener la información sobre un índice en particular

```
IndexDesc indexDesc = SystemDefs.JavabaseCatalog.getIndexInfo("sailors", "sid", new IndexType
(IndexType.B_Index));
```

todos los índices de una tabla

```
IndexDesc  [] indexDescs = SystemDefs.JavabaseCatalog.getRelIndexes("sailors");
```

o sobre todos los índices de un atributo

```
IndexDesc  [] indexDescs = SystemDefs.JavabaseCatalog.getAttrIndexes("sailors", "sid");
```

7. Iterator

7.1. Descripción general

La componente iterator es el punto de acceso a las tablas y a sus registros. Como tal ofrece, también, las operaciones básicas sobre tablas, como join, proyección y selección. La interfaz usada, como dice el nombre, es la de un iterador, el cual se inicializa en base a otros iteradores o heapfiles y después se va accediendo elemento por elemento en un orden definido por el iterador.

7.2. Clases principales y sus protocolos principales

La interfaz de iterador abstraída por la clase *Iterator* de la cual heredan todas las clases que ofrecen acceso a un conjunto de registros.

7.2.1. Scan

Scan se decidió tratarla en esta sección, por más que se encuentre en el paquete de *Heapfile* y no herede de *Iterator*, porque su funcionalidad es la de un iterador e implementa la interfaz de *Iterator*.

Este iterador se ocupa de iterar sobre todos los registros de un *Heapfile*, sin ninguna condición.

7.2.2. FileScan

FileScan es simplemente un wrapper de la clase *Scan,* o sea que no ofrece casi funcionalidad adicional a Scan salvo por algunas que se deberían abstraer a otros iteradores.

La funcionalidad adicional que ofrece es la de realizar un selección sobre el *Scan* y realizar una proyección. Estas dos se deberían implementar utilizando iteradores. Notar que si la condición de selección es nula, se itera sobre todos los registros del *Heapfile*.

La evaluación de la condición de selección y la proyección son proveídas por las clases *PredEval* y *Projection*, respectivamente.

7.2.3. NestedLoopsJoins

NestedLoopsJoins permite realizar un join entre un *Iterator* y un *Heapfile* (aunque seria más correcto que fuera entre dos *Iterator*). El algoritmo utilizado es el más simple de los implementados, es un doble ciclo donde, en el cuerpo del ciclo interno, se verifican la condición de join. También se puede realizar una proyección en la salida.

7.2.4. SortMerge

SortMerge realiza un join utilizando el algoritmo de merge sort. Como *NestedLoopsJoins* permite realizar una proyección en la salida.

Para realizar el sort utiliza varias clases auxiliares, como *Sort* (que se encarga de iterar de una manera ordenada un *Heapfile*) y *IoBuf* (un simple wrapper para el *BufMgr* para que trabaje con tuplas).

Esta implementación no elimina los registros duplicados.

7.2.5. Sort

Sort permite iterar de una manera ordenada la salida de otro iterador. Utiliza un árbol binario ordenado para establecer el orden de la salida.

7.3. Interacción con otros componentes

Esta clase es el punto de acceso principal a los registros. En general con esta clase es con la cual se realiza toda interacción con la base de datos (salvo la modificación de las tablas o de los registros).

El paquete utiliza las clases *OBuf* e *IoBuf* para interactuar con el *BufMgr*. Estas clases simplemente permiten a los iteradores comunicarse con el BufMgr usando tuplas. Con lo cual deberían estar definidas en en el paquete de *BufMgr* para maximizar la modularidad del sistema. Esto se debe a que, como esta implementado actualmente, el paquete *Iterator* utiliza directamente funcionalidad del *BufMgr* que no debería ser necesaria para las tareas que realiza.

Respecto a la utilización de las otras componentes, *Iterator* se limita a utilizar *Scan* del paquete *Heapfile* para la interacción con las tablas. Esta clase debería heredar de Iterator para reforzar los protocolos.

7.4. Ejemplo de uso

La utilización de un iterador es bastante simple. Alcanza con inicializarlo con los parámetros necesarios (sobre que iteradores o relaciones se accede y cuales son sus atributos, los atributos de salida y las condiciones de selección). Por ejemplo para FileScan tendríamos:

```
FileScan scan = new FileScan ("relacion.in",atributos,tamañosAtributos, proyeccion,condiciones);
```

Para el acceso iterativo a las tuplas se utiliza simplemente el get_next:

```
Tuple tuple = null;
while((tuple = scan.get_next()) != null){
//Hacer algo con la tupla
}
```

Cuando se termina de utilizar el scan hay que acordarse de hacer un close:

```
scan.close();
```

7.5. Evaluación del componente

Iterator ofrece una interfaz poco practica para el uso frecuente. Mucha funcionalidad esta repetida, como la proyección, que simplemente se podría implementar como otro iterador. También resulta incomoda la construcción de un iterador por la cantidad de estructuras de datos que hay que generar. Igualmente esto se debe a pobre diseño de las clases y su modularización a favor de un estilo que se encuentra, por lo general, en programas escritos en C.

8. Index

8.1. Descripción general

Esta componente ofrece las utilidades para poder crear y acceder a la información de un índice. La única implementación soporta hasta el momento es sobre arboles B+.

La componente BTree contiene la implementación de un índice basado en un B+ Tree. Estos árboles permiten mantener una estructura de datos ordenada con un costo relativamente bajo. En los BTrees se guardaran 2 elementos: El elemento "clave", o sea el campo que define el orden del índice y el RID de la tupla completa. Para más información sobre BTrees (que escapan al scope de este trabajo practico) remitirse a la bibliografía.

8.2. Clases principales y sus protocolos principales

8.3. BTreeFile

Esta clase extiende *indexFile* para representar un índice de tipo *BTree*. Su funciones mas importante son:

Insertar y borrar elementos: Las funciones Insert y Delete se encargan de realizar esta tarea. Aquí se realizan las tareas de rebalanceo del B+ Tree.

Búsqueda de elementos: A través de la función new_*scan* se obtiene un *IndexScan* con el cual se puede iterar un conjunto de valores contiguos en el orden del índice, dado los límites inferior y superior.

También contiene el header del BTree (*BTreeHeaderPage*) con datos como tipo del la clave de ordenamiento, el algoritmo de borrado a utilizar .

8.3.1. IndexFileScan

Es el tipo de iterador que devuelve new_scan de la clase *BTreeIndexFile*. Posee una interfaz simple que consta de get_next y delete_current. Está extendida por *BTFileScan*, que es quien tiene el algoritmo de iteración de árboles B+ Tree.

8.3.2. BT

En esta clase se implementan varias utilidades qué son usadas por los distintos componentes del paquete. El método *keyCompare* esta implementado aquí.

8.3.3. BTSortedPage

Es el tipo de heap file page que se utiliza en los nodos del árbol. Se encarga de la inserción y borrado de elementos dentro del nodo (funciones que son llamadas por el algoritmo de inserción y borrado de BTreeFile). Esta clase es extendida por *BTIndexPage* y *BTLeafPage*, que son los nodos internos y las hojas del árbol respectivamente.

8.3.4. KeyDataEntry

Esta clase contiene los datos que se insertarán en el B+ Tree. En definitiva no es más que una estructura que incluye una *KeyClass* y una *DataClass*. *KeyClass* representa al elemento clave por el cual se ordena el B+ Tree, que puede ser de tipo Integer (*IntegerKey*) o de tipo String (*StringKey*). *DataClass* representa al dato que acompaña a la clave en el B+ Tree. En caso de un nodo interno (*BTIndexPage*) el dato será el *PageId* del nodo hijo y en caso de una hoja del árbol (*BTLeafPage*) será el *RID* de la tupla representada por esa entrada del B+ Tree.

8.4. Interacción con otros componentes

8.4.1. Agregado de elementos al árbol

La función de agregado (*insert*) del *BTreeFile* es llamada en dos circunstancias:

Creado de un índice (*IndexCatalog.AddIndex*): En este caso se utiliza para agregar al índice todas las tuplas existentes en la relación hasta el momento.

Agregado de una tupla a la relación (*Utility.InserRecUT*): En este caso se agrega la nueva tupla al índice.

8.4.2. Borrado de elementos del árbol

La Clase *BTFileScan* (el *IndexFileScan* correspondiente a los índices de tipo *BTree*) implementa *delete_current*, que utiliza el método de borrado de la clase *BTIndexFile*. En la práctica ningún modulo implementa el uso de esta funcionalidad.

8.4.3. Búsqueda de elementos del árbol

En el módulo *IndexUtils* se encuentra la función *BTreeScan* que recibe el *IndexFile* y los límites del rango a iterar, y devuelve el *IndexScan* que corresponde. Este es el método utilizado en *SpeedJoin* para conseguir las tuplas internas que cumplen con el criterio del join.

9. Index Nested Loop Join

9.1. Explicación

Index Nested Loop Join es una estrategia de join que utiliza la existencia de un índice en la relación "interna" para acelerar el tiempo de procesamiento del mismo.

El pseudocódigo de un recorrido completo sobre el Index Nested Loop Join es el siguiente:

```
Para cada tupla de la relacion externa
   Buscar en el indice todos las tuplas de la relacion interna que contengan
   el valor con el cual se hace join
      Procesar(tupla externa, tupla interna)
```

A simple vista es similar a un NestedLoopsJoin. La diferencia principal entre ambos, es que en vez de iterar la relación interna de principio a fin, solo lo hace dentro de los valores que necesita, ya que utiliza el índice para extraer eficientemente dichos límites.

9.2. Implementación

El constructor del join recibe principalmente 5 parámetros (en la implementación real son más, ya que datos que corresponden al mismo parámetro se envían por separado).

- Relación externa: La relación que se iterará. Aquí se acepta en realidad cualquier iterador, por lo que la relación externa podría provenir de un join, proyección, etc.
- Elemento de la relación externa que participará del join
- Relación interna: La relación cuyo índice será utilizado. Cabe aclarar que no puede provenir de otro join o proyección, ya que se necesita acceder al índice directamente y no usando otros iteradores
- Índice a utilizar: El índice de la relación interna
- Proyección a realizar: Se refiere a que campos se enviarán en la tupla de salida y en que orden

En el código las partes mas importantes son:

getnext()

```
(1) KeyClass rightValue = new IntegerKey(outertuple.getIntFld(joincol1));
(2) inner = IndexUtils.BTreescan(rightValue,rightValue,innerIndexFile);
```

Cuando se agotan las tuplas de la relación interna, se avanza iterando la relación externa y luego realizando una búsqueda en el índice de la relación interna. En la primera línea del código se extrae el elemento de la relación externa que será utilizada para el join.

En la segunda, se realiza una búsqueda en el índice de la relación interna. Notar que se envía dos veces el parámetro *RightValue*, ya que los índices permiten escanear rangos de valores aparte de coincidencias exactas.

```
(1) while ((innerdata = inner.getnext()) != null){
(2)    rid = ((LeafData)innerdata.data).getData();
(3)    innertuple= hf.getRecord(rid);
}
```

Este es el código que se encarga de iterar el índice de la relación interna. En la línea 1, se pide al índice el próximo valor a analizar. Luego se extrae del índice el valor RID correspondiente a dicha posición del índice. Por último, se busca en la relación interna la tupla correspondiente al RID devuelto por el índice.

9.3. Pruebas y mediciones.

Para realizar pruebas y mediciones sobre la performance de esta estrategia de join, se creo la clase JoinSpeedTest. En ésta, se realiza de tres formas distintas
la siguiente Query:

```
Find the names of sailors who have reserved a boat and print each name once.
SELECT DISTINCT S.sname
FROM Sailors S, Reserves R
WHERE S.sid = R.sid
```

Éstas tres formas, son las estrategias de join que se encuentran implementadas en NestedLoopsJoin y SortMergeJoin, incluidas en la versión original de Minibase, y Index Nested Loop Join, implementada en el presente trabajo.

9.3.1. Métrica de medición

Para poder comparar las distintas estrategias teníamos varias alternativas.

- Medir el tiempo transcurrido
- Medir la cantidad de llamados a getnext
- Medir la cantidad de accesos a disco

La primer posibilidad era poco practica, ya que al estar realizando otras tareas el sistema operativo mientras corremos los tests, los resultados pueden verse afectados por situaciones ajenas a Minibase.

La última posibilidad es quizás la mejor indicadora del costo del join. Sin embargo, la cantidad de llamados a getnext es una buena aproximación del costo del join, y es bastante más simple de implementar. Es por ello que optamos por esta última opción.

9.3.2. Implementación

Para obtener estadísticas de los getnext llamados durante los distintos joins teníamos 2 opciones.

- Enviar un parámetro por referencia al join y que lo devuelva sumando la cantidad de getnext utilizados.
- Utilizar una variable global a la que cada vez que se realiza getnext es incrementada en 1.

La primera opción es quizás la m'as "elegante", pero lleva a modificar todos los protocolos para obtener una estadística utilizada solo para realizar pruebas. Por lo tanto nos decidimos por la segunda opción. De este modo se agrego a las tres estrategias de join, la linea:

```
SystemDefs.estadisticas++;
```

cada vez que se realiza un llamado a getnext

9.3.3. Análisis teórico de las mediciones

En principio las relaciones a las cuales se les aplicará el join son:

- sailors.in Tiene 25 elementos
- reserves.in Tiene 10 elementos, pero solo 6 sid distintos. Los 10 elementos tienen un correspondiente sid en la tabla sailors.

La parte de eliminación de duplicados es común a los 3 joins, analicemos cuantos getnext deberían hacer aproximadamente.

Eliminación de duplicados
Recibe una relación con 10 elementos. Para ordenarla hace 10 getnext (para obtenerlos). Luego de ello quedan 6 elementos distintos, por lo que al iterarlos se utilizarán 6 getnext más. En total son 16 getnext.

NestedLoopsJoin
Recordemos que itera todas las combinaciones de tupla interna y externa, por lo que debe realizar 10*25 = 250 getnext. Sumando la eliminación de duplicados da 276 getnext

SortMergeJoin
Aquí se iteran completamente las dos relaciones para ordenarlas. Luego se vuelven a iterar para hacer el merge (en realidad es posible que no se iteren las 2 hasta el final debido a que se puede "acabar" una de las dos relaciones antes que la otra, por lo que se termina el merge sin iterar completamente una de las dos). Y por último se eliminan los duplicados. Es en total (25 + 10) * 2 + 16 = 86 getnext.

SpeedJoin
Aquí se itera completamente la relación externa, y de la relación interna solo se iteran las tuplas que coinciden en sid con la tupla externa iterada. Como todas las tuplas de la relación interna (reserves.in) coinciden en sid con solo una tupla de la relación externa (sailors.in que tiene como pk a sid), en total se van a iterar (en algún orden) todas las tuplas de esta relación interna. Sumándole la eliminación de duplicados da 25 + 10 + 15 = 51

9.3.4. Resultado de las mediciones

Al correr las pruebas se obtienen los siguientes resultados.

[extracto del output de la corrida del test]

```
SortMergeJoin hace 87 getnext
SpeedJoin hace 48 getnext
NestedLoopsJoin hace 288 getnext
```

Lo que demuestra en este caso que Index Nested Loop Join es la opción más rápida.
Si en sailors.in hubiese habido sid repetidos, entonces Index Nested Loop Join hubiese iterado más de una vez los valores de la relación interna, por lo que sus estadísticas se hubieran acercado más a las del SortMergeJoin.
Llegado al extremo de que todos los sid de sailors y reserves fueran iguales, Index Nested Loop Join hubiese tardado lo mismo que el NestedLoopsJoin ya que se hubiesen tenido que iterar todas las combinaciones de las dos tablas.

10. Herramienta de carga de datos

10.1. Descripción general

La clase Utility provee de algunas funcionalidades para interactuar con la base de datos sin tener que ocuparse de detalles de bajo nivel. Las tres utilidades provistas por esta clase son las siguientes:

1. Insertar un registro en una tabla
2. Borrar uno (o más) registro(s) de una tabla
3. Cargar un archivo con registros (en formato ASCII) en una tabla

La primera de las funcionalidades inserta un registro en una tabla actualizando los índices existentes sobre dicha tabla. Esta funcionalidad está implementada en en el método *insertRecUT* el cual toma como parámetros el nombre de la tabla en donde se debe insertar el registro y una lista con los nombres de los campos y sus respectivos valores (un array de *attrNode*). Previamente a la inserción del registro, esta función verifica que el tipo de los valores pasados como parámetro coincidan con los tipos especificados en el catálogo para la tabla en cuestión, así como también si se viola la clave primara (de existir sobre la tabla).

La utilidad de carga inserta los registros contenidos en un archivo ASCII donde además se especifican los campos de tabla con sus respectivos tipos y longitudes (en la primer fila). Esta funcionalidad está implementada en el método *loadUT* el cual toma como parámetros el nombre de la tabla y el nombre del archivo que contiene los registros. Para realizar la inserción de los registros utiliza la funcionalidad anterior (con la tabla especificada y los campos de la misma tomados del catálogo). Si la relación no existe se crea, y si ya existe, se introducen los nuevos registros.

Por lo general los motores de bases de datos crean un índice sobre los atributos primary key, en este caso se decidió no hacerlo ya que se trató de reducir los problemas de consistencia con el resto del código. O sea, en Minibase es posible, en ciertas condiciones, crear una relación con atributos pk y sin índice sobre ellos, se prefirió adoptar esa postura.

10.2. Interacción con otros componentes

La utilidad de carga interactúa fuertemente con la componente de catalogo para la inserción de los registros y para la creación de las relaciones.

10.3. Ejemplos de uso

Para cargar una tabla desde un archivo simplemente se utiliza la función *loadUT* del paquete *Utility* de la siguiente manera:

```
Utility.loadUt("nombre relación","nombre archivo");
```

El formato del archivo de carga es el siguiente:

```
atributo1:String30:pk  atributo2:Integer  atributo3:Real
Juan Domingo  1  2.0
```

La separación entre las columnas es el carácter tab, esto permite establecer “Juan Domingo” como atributo1, ya que el tab establece el fin de la columna. En este ejemplo tenemos una tabla con un registro (Juan Domingo, 1, 2.0) en una tabla con un atributo de tipo String, de tamaño 30 y primary key y otros dos de tipo entero y real respectivamente. El tipo puede contener minúsculas y/o mayúsculas. En el caso de String el tamaño se toma el valor que le sigue inmediatamente.

11. Bugs encontrados

Los bugs encontrados en el código de Minibase se deben, casi enteramente, al pasaje de C/C++ a Java.

El error más común que se encontró fue el pasaje por referencia. En C/C++ es posible pasar por referencia un parámetro utilizando el &. Esto en Java no es posible como se pasa a ejemplificar:

```
RelDesc desc = null;
getRelDesc("sailors.in", desc);
```

En este caso se crea un puntero a un objeto RelDesc, y se lo inicializa en null. El comportamiento esperado es que la función getRelDesc ponga en desc la descripción de la relación con el nombre "sailors.in". Esto no sucede porque el puntero desc se pasa por copia a la función, con lo cual desc nunca es modificado por getRelDesc y queda en null. Para corregir este problema se opto por devolver el objeto desc de la siguiente manera:

```
RelDesc desc = getRelDesc("sailors.in");
```

Dando un interfaz más clara.

Otro bug común que se encontró fue la falta de los llamados a close a los scan que se abren. Esto produce que las paginas queden pineadas y no se puedan liberar nunca. Este error, probablemente, se deba a la falta de destructores explícitos en Java (existe finalize pero no se garantiza su ejecución). Con lo cual se introdujeron los close necesarios.

12. Conclusiones generales

En líneas generales, hemos encontrado un código que cumple con los módulos, funcionamiento y estructuras clásicas que describe la bibliografía para una base de datos, en especial el libro de Ramakrishna.

Por lo tanto, podemos afirmar que Minibase posee las características necesarias para utilizarse como una base de datos de estudio.

Sin embargo, hemos observado que Minibase posee un código de muy poca calidad.

Los puntos más básicos y visibles para programar y/o extender funcionalidad en Minibase son los siguientes: la construcción de las clases es incómoda; a tal punto que ciertos constructores toman parámetros redundantes. Incluso existen clases que no construyen totalmente a sus objetos en el constructor, sino que hay que llamar luego a distintos métodos para completar su construcción.

Los métodos para generar querys son extremadamente rudimentarios y no aprovechan en lo más mínimo el paradigma de objetos.

En muchos módulos se utilizan listas encadenadas (BM, HeapFile, étc). Estas listas están implementadas sobre nodos con apuntadores al siguiente. Java, al igual que la mayoría de los lenguajes OO, posee estructuras de datos dinámicas, que implementan de igual manera esta funcionalidad(colecciones, y en su última versión, colecciones genéricas) evitando la ardua tarea de desarrollar y mantener listas compuestas por nodos con apuntadores.

Todo lo anterior se debe en su mayoría a un diseño de clases muy pobre.

A tal nivel que no parece desapropiado conjeturar que Minibase parece haberse migrado del lenguaje C al lenguaje java sin el más mínimo análisis. Obviando completamente los detalles(y no solo detalles) del pasaje de un lenguaje estructurado a otro orientado a objetos, con el consecuente resultado de un código de pésima calidad.

Dejando eso de lado, este trabajo práctico nos ha servido para familiarizamos en profundidad con las ideas y funcionamiento de los módulos principales que componen una RDBMS, en las distintas capas de abstracción. Después de adquirir toda esa información, no parece nada imposible, eventualmente, escribir de cero una RDBMS en parte o en su totalidad.

13. Referencias/Bibliografía

- The Minibase Home Page http://www.cs.wisc.edu/coral/mini_doc/minibase.html
- Página de la materia http://www.dc.uba.ar/people/materias/bd/
- CS432 Assignment 4: Joins (curso de Cornell University, Computer Science Dept.) http://www.cs.cornell.edu/courses/cs432/20

- Minibase Assignment #1: Buffer Manager (curso de Carleton College, Mathematics and Computer Science Dept.) http://www.mathcs.carleton.edu/faculty/dmusican/cs347f03/proj1/
- Project 1: B+ Tree (Computer Science and Engineering) http://www.cse.unsw.edu.au/~cs9315/proj/proj1/spec.html
- Buffer Manager Internal Design (York University) http://www.cs.yorku.ca/course_archive/2003-04/W/4411/proj/bufmgr/
- System Architecture (Iowa State University, College of Liberal Arts and Sciences Department of Computer Science) http://www.cs.iastate.edu
- JavaDocs de Minibase http://www.cs.purdue.edu/homes/aref/Spring2006CS448/prj1doc/overview-summary.html
- B+ tree
  http://www.seanster.com/BplusTree/BplusTree.html
  http://en.wikipedia.org/wiki/B_plus_tree
  http://www.nist.gov/dads/HTML/bplustree.html