Base de datos

1er. Cuatrimestre 2006

Proyecto: Minibase
Informe: Diseño Detallado

14 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 codigo no esta disponible, cuando se pide el codigo de Minibase en la pagina de Ramakrishnan, se recibe una version 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 codigo en profundidad y realizar varios tests caso por caso para poder encontrar todos los problemas.

Como objetivos secundarios estan la creación de un nuevo tipo de Join y de un utilitario para la creación de nuevas relaciones. Tambien se extendio 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 collección 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.
- Tambien, 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 informacion 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, étc) 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 cojunto 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 asignandole 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 (idem).
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, étc). 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, étc)
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.

5.4. Ejemplo de uso

6. Catalogo

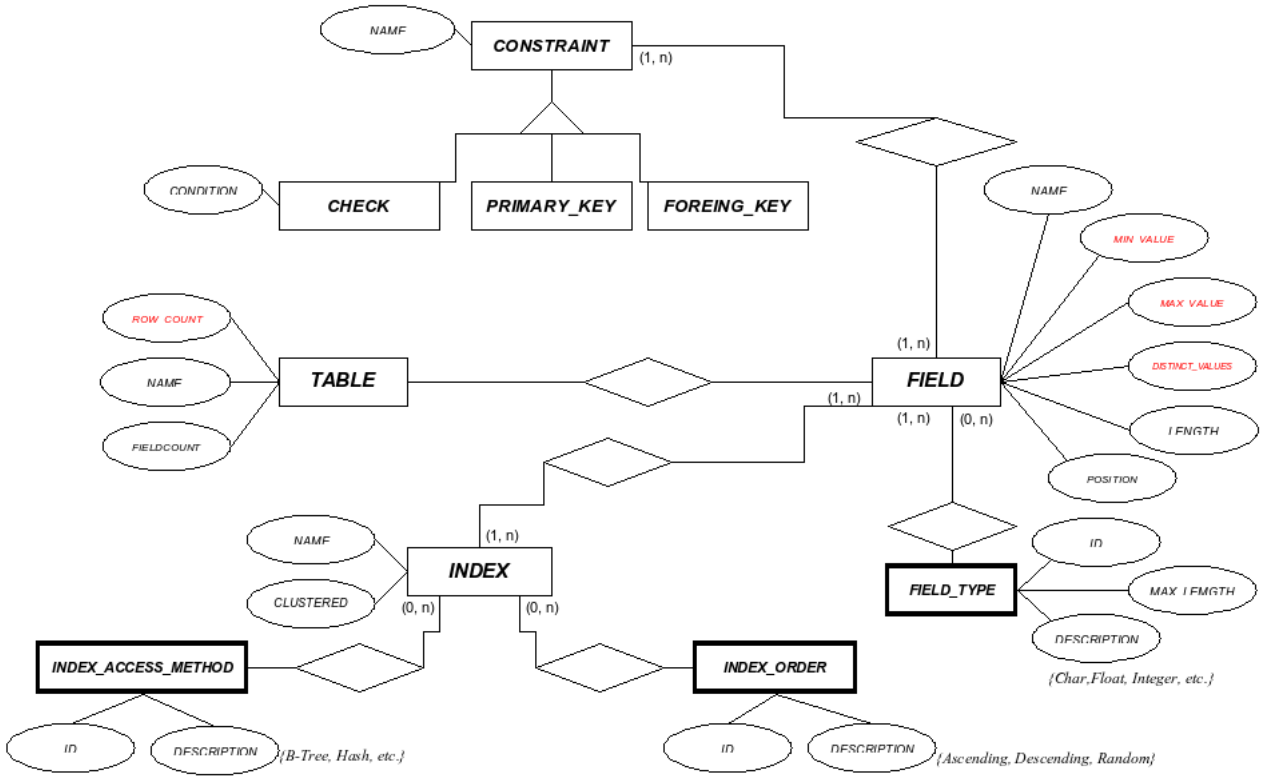
6.1. Descripción general

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

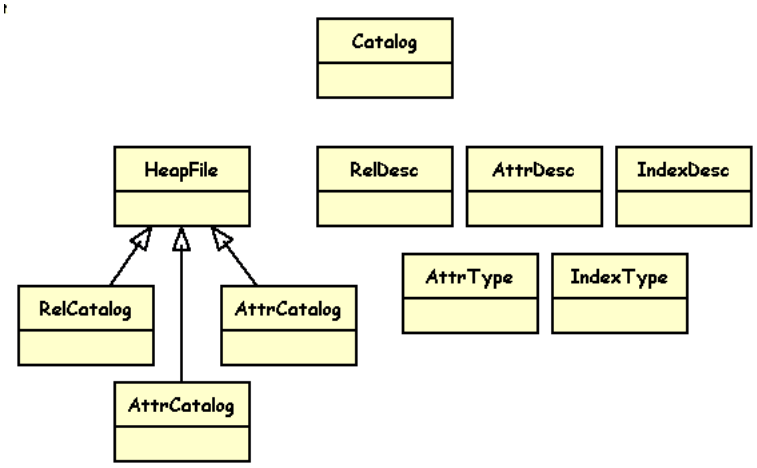
6.2. DER

(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 IndexCatalogs.

6.3.2. RelCatalog, AttrCatalog e IndexCatalogs

Las clases RelCatalog, AttrCatalog e IndexCatalog mantienen información sobre las tablas, los campos y los índices existentes en la base de datos, respectivamente. Estas clases están implementadas como heapfiles (heredan de la clase HeapFile) extendiendo el comportamiento básico de un heapfile con la funcionalidad particular para la administración la información que mantienen.

6.3.3. RelDesc, AttrDesc e IndexDesc

Para administrar la información de tablas, campos 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 (por ejemplo, para crear una nueva relación se debe especificar

6.3.4. 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).

6.3.5. AttrType

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

6.3.6. IndexType

Las instancias de esta clase representan los posibles tipos de índices que pueden crearse sobre una tabla (B-Tree, Hashed). (No creo que sea muy significativo describir estas clases? preguntar)

6.4. attrData

//attrData class for minimum and maximum attribute values

6.4.1. TupleOrder

attrInfo // class used for creating relations

6.4.2. attrNode

6.5. 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.

7. Iterator

7.1. Descripción general

La componente iterator es el punto de acceso a las tablas y a sus registros. Como tal ofrece, tambien, las operaciones basicas sobre tablas, como join, proyeccion y seleccion. La iterfaz usada, como dice el nombre, es la de un iterador, el cual se inicializa en base a otros iteradores o heapfiles y despues 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 abstaida 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 deberian abstraer a otros iteradores.

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

La evaluación de la condicion de seleccion y la proyeccion son proveidas 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 verfican la condicion de join. Tambien se puede realizar una proyeccion en la salida.

7.2.4. SortMerge

SortMerge realiza un join utilizando el algoritmo de merge sort. Como *NestedLoopsJoins* permite realizar una proyeccion 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 implementacion no elimina los registros duplicados.

7.2.5. Sort

Sort permite iterar de una manera ordenada la salida de otro iterador. Utiliza un arbol 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 interaccion con la base de datos (salvo la modificacion 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 deberian 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 deberia ser necesaria para las tareas que realiza.

Respecto a la utilizacion de las otras componentes, *Iterator* se limita a utilizar *Scan* del paquete *Heapfile* para la interaccion con las tablas. Esta clase deberia 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 parametros necesarios (sobre que iteradores o realaciones se accede y cuales son sus atributos, los atributos de salida y las condiciones de selección). Por ejemplo para FileScan tendriamos:

```
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 proyeccion, que simplemente se podria implementar como otro iterador. Tambien resulta incomoda la construccion 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 informacion de un indice. La unica implementación soporta hasta el momento es sobre arboles B+.

8.2. Clases principales y sus protocolos principales

La clase principal es *IndexScan*. Este iterador permite recorrer las tuplas que referencia un indice utilizando *IndexFileScan*. *IndexFileScan* permite iterar sobre un arbol B+.

8.3. Interacción con otros componentes

Index utiliza fuertemente la componente btree, que es una implementación de un arbol B+.

9. Tests

9.1. Descripción

9.2. Resultados obtenidos

9.3. Ejemplos de uso

10. Herramienta de carga de datos

10.1. Descripción

10.2. Ejemplos de uso

11. Conclusiones generales

12. Apéndices

13. Código fuente

14. 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