Base de datos

1er. Cuatrimestre 2006

Proyecto: Minibase

Informe: Diseño Detallado

27 de octubre de 2006

Número de Grupo:
Nombre del Grupo: ?

Integrantes

Apellido y Nombre	L.U.	Mail
Leandro Groisman	222/03	gleandro@gmail.com
Fernando Rodriguez	XXX/XX	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

2. BufferManager

2.1. Descripción general

Es la estructura encargada de traer páginas de memoria secundaria a memoria principal sin la necesidad de leerlas de la memoria secundaria constantemente. Para este propósito, posee una colección frames, que son los encargados de alojar las páginas en la memoria principal.

Como ya mencionamos, la utilidad 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 paginas 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 alojar una pagina que no se encuentra en el mismo en ese momento. Para realizar este procedimiento se utiliza un algoritmo de remoción de páginas.

La implementación del Buffer Manager utiliza una Interfaz para acceder a la política de remoción(clase Replacer). Minibase presenta distintas implementaciones de esta interfaz para proveer algoritmos FIFO(clase Clock), LRU y MRU.

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. Las páginas cuentan con una marca que indica si han sido modificadas (dirty). 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 DiskSpace Manager) en caso de estar marcada, para evitar que se pierdan los cambios realizados.

2.1.1. Diseño interno, estructuras

El buffer pool es una collección de frames (secuencia de paginas de tamaño fijo de bytes de memoria principal) que son manejadas por el Buffer Manager. Físicamente guardado como un arreglo (bufPool) Adicionalmente, se mantiene un arreglo de bufDescr[numbuf] 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 correspondiente. Una página es identificada por un número de página que es generado por DM cuando la página es alojada, y es único para todas las páginas en la base de datos. El objeto PageId es la encapsulación de un entero.

Se usa una tabla de hash para identificar que páginas de disco ocupan un frame. La tabla de hash está implementada(enteramente en memoria principal) usando un arreglo de pares <numero de pagina, numero de frame>. El arreglo es llamado directorio. Dado un número de página, se puede aplicar la función de hash para encontrar la entrada del directorio que apunta al bucket que contiene el número de frame para esa página, si esa pagina esta en el buffer pool. Si se busca en el bucket y no se encuentra un par conteniendo el número de página, la página no está en el pool. Si se encuentra tal par, este dirá en que frame reside la pagina.

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 esta en el buffer pool, debe hacer lo que sigue:
  1. Elegir un frame para remover la página que contiene, usando las políticas de remocion (Clock, MRU o LRU)
  2. Si el frame elegido para remover tiene una página que ha sido modificada(dirty= true), se deben escribir los datos de la página que el frame contiene a disco, mediante DM.
  3. Se lee la pagina solicitada desde disco(otra vez, llamando a la DM) y ubicarla dentro del frame elegido para remover. El pin count y el dirtybit para el frame son inicializados en 0 y falso respectivamente.
  4. Se borra la entrada para la página vieja en la tabla de hash del Buffer Manager y se inserta una entrada para la página nueva.
- Tambien, se actualiza la entrada para este frame en el arreglo del bufDescr.
- Se pincha(pin) la página requerida incrementando el pin count en el descriptor para este frame.

2.2. Clases principales y sus protocolos principales

2.2.1. 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) sus pin count. El pin count cambia cuando se pincha (pin) o despincha (unpin) una página.

Atributos miembro

```
public boolean dirty
```

Indica si la página que contiene el frame ha sido modificada. true, para frames cuyas páginas han sido modificados, false para frames cuyas páginas no han sido modificadas.

`public PageId pageNo`

El id de la página, o INVALID_PAGE si el frame está vacío.

`public int pincnt`

cantidad de pinches: Si la página está pinchada mas de una vez (pin_cnt >0) no puede ser removida.

Métodos relevantes

`public int pin()`

Incrementa pin_cnt en 1

`public int unpin()`

Decrementa pin_cnt en 1

2.2.2. 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.

Atributos miembro

`public BufHTEntry next`

La próxima entrada.

`public PageId pageNo`

El número de página.

`public int frameNo`

El frame en el que esta la página.

2.2.3. 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

Atributos miembro

```
private static final int HTSIZE = 20
```

El tamaño de la tabla de hash.

```
private BufHTEntry ht{[}] = new BufHTEntry{[}HTSIZE]
```

Cada bucket mantiene una lista enlazada de BufHTEntrys, NULL significa que no hay nada.

Métodos relevantes

```
public boolean insert(PageId pageNo, int frameNo)
```

Inserta una asociación (BufHEntry) entre la página(pageNo) y el frame (frameNo).

Esta asociación es utilizada para indicar que el frame (frameNo) aloja la página (pageNo).

Dicha asociación es insertada en el primer lugar del bucket.

Ej:

b11->b12->b13

b21->b22->b23

.

.

bn1

Al insertar b0 (supongamos que la funcion de hash me da el bucket de b11) queda:

b0->b11->b12->b13

b21->b22->b23

.

.

bn1

```
public int lookup(PageId pageNo)
```

Busca el frame en el que está alojada la página con el id "pageNo".

Si lo encuentra devuelve su número, caso contrario, INVALID_PAGE.

```
public boolean remove(PageId pageNo)
```

Elimina una página de la tabla de hash.

2.2.4. BufMgr

Asigna páginas nuevas en el buffer pool.
Asocia y desasocia páginas con frames (pins y unpins)
Libera las páginas en el frame y usa algoritmos de remoción para reemplazar las páginas. (LRU, MRU o Clock)
pinPage: guarda una página en un frame, si no hay ninguno libre el algoritmo de reemnplazo elige una.
Si esa página contenia info, la guarda. Luego asocia la página al frame.
El que escribe y lee a disco es DM
Atributos miembro

```
private BufHashTbl hashTable = new BufHashTbl();
```

La tabla de hash. Sirve para indicar en que frame está alojada la página.

```
private int numBuffers;
```

Numero total de frames en el buffer pool.

```
private byte{[}]{[}] bufPool; default = byte{[}NUMBUF]{[}MAXSPACE];
```

buffer pool físico(donde se guaran físicamente las páginas del buffer).

```
private FrameDesc{[}] frmeTable; default = new FrameDesc{[}NUMBUF];
```

Un arreglo de descriptores, uno por cada frame. Sirve para indicar si la página que está alojadas en el buffer está siendo utilizada (pin_cnt > 0) y si fué modificada (dirty = true).

```
private Replacer replacer
```

El objeto Replacer, encargado de obtener una página para su remoción implementando una política de remoción particular.
Métodos

```
public BufMgr( int numbufs, String replacerArg )
```

Constructor: Crea un objeto buffer manager estableciendo la política de remoción.
Parámetros:
numbufs número de buffers en el buffer pool.
replacerArg nombre de la política de remoción.(LRU, MRU o Clock)

```
private void privFlushPages(PageId pageid, int allpages)
```

Si all_pages es distinto de 0, escribe todas las páginas válidas y dirty a disco.

Si all_pages es 0, escribe la página pageid a disco (solo si es válida y dirty).

Luego libera el frame.

Este método se puede ejecutar únicamente si no hay páginas asociadas al frame.

Obtengo el id de la página asociada al frame

Obtengo la página del buffer pool

Escribo la página a disco

Elimino la página de la tabla de hash

Libero el frame de la página en el arreglo frmaTable

```
public void pinPage(PageId pinpgid, Page page, boolean
emptyPage)
```

Page_Id_in_a_DB número de página en la base de datos.

page: objeto pagina

emptyPage: true-> página vacía, false-> página no vacía

Pincha una página; esto significa que la página no puede ser removida.

Busca la página en el buffer pool(por medio de la tabla de hash). Si la encuentra, incrementa pin_count del frame en donde está alojada.

Si no está busca un frame mediante la política de remoción para alojarla.

Para esto hace lo siguiente:

Elimina la asociacion página-frame en la tabla de hash

Inserta la nueva asociación en la tabla de hash

Inserta la asociación en la frmeTable

Si tuvo que remover una página para alojar esta, escribe la página vieja que estaba en el frame seleccionado si fué modificada (dirty = true).

Si la página no está vacía (emptyPage=false)la lee de disco usando el metodo apropiado del paquete diskmgr , la inserta en el bufPool y luego incrementa el pin_cpunt del frame en donde está alojada.

```
public void unpinPage(PageId PageIdinaDB, boolean
dirty)
```

globalPageId_in_a_DB numero de página en minibase.

dirty: indica si está marcada como modificada.

Despincha una página especificada por el id de la página.

Decrementa pincount, y si llega a cero, la página queda en el grupo de candidatos para la remoción.

Este metodo debe ser llamado con dirty==true si el cliente ha modificado la pagina.

Si esto sucede, esta llamada debe establecer el dirty bit para este frame.

```
public PageId newPage(Page firstpage, int howmany)
```

firstpage la primer página.
howmany número total de páginas nuevas a alojar.
devuelve el id de la primer página de las páginas nuevas.
Aloja nuevas paginas.
Llama al DM para alojar un conjunto de páginas nuevas.
Busca un frame en el buffer pool para la primer página y la pincha(pin).
Si el buffer está lleno, llama al DM desalojando todas esas páginas y devuelve un error (null).

```
public void freePage(PageId globalPageId)
```

globalPageId el número de página en la Base de Datos.
Borra la página globalPageId.
Si la página estaba en el buffer pool, antes de hacer esto, llama a replacer.free.
Elimina la entrada de la tabla de hash.
Elimina la entrada en el descriptor de frames y luego desaloja la página, llamando a BD.

```
public void flushPage(PageId pageid)
```

Escribe la página pageid a disco, si es válida y fué modificada (dirty=true).
Para hacer esto, llama ejecuta el método write_page de DiskManager.
Luego libera el frame.

```
public void flushAllPages()
```

Escribe todas las páginas válidas y modificadas (dirty=true) a disco.
Luego libera el frame.
Este método se puede ejecutar únicamente si no hay páginas asociadas al frame.

Proyectores y delegados

```
 getNumBuffers-> numBuffers

getNumUnpinnedBuffers->replacer.getNumUnpinnedBuffers

frameTable()->frmeTable

writepage->DB.writepage

readpage->DB.readpage

allocatepage->DB.allocatepage

deallocatepage->DB.deallocatepage
```

2.2.5. Replacer

Esta interfaz se asocia con el algoritmo de reemplazo. Describe si la página de un frame esta asociada(pin), desasociada(unpin) o disponible.

Describe que frame debe ser elegido para su remoción.

Atributos miembro

```
protected BufMgr mgr;
```

Objeto buffer manager

```
protected int head;
```

Manejador de reloj. Esta variable es utilizada por el algoritmo de reloj.

```
protected STATE statebit{[}];
```

Guarda el estado de un frame. este puede ser (Avaible, Referenced o Pinned)

Métodos

```
public void pin( int frameNo )
```

frameNo número de frame de la página.

Devuelve true si todo estuvo bien.

Asocia una página candidata en el buffer pool.

Hace un pin en el frame de la tabla de descriptores de frame del buffer pool.

Establece su estado como Pinned.

———————————— Unpins a page in the buffer pool.

frameNo número de frame de la página.

Devuelve true si todo estuvo bien

Hace un unpin en el frame de la tabla de descriptores de frame del buffer pool.

Establece su estado como Referenced si no tiene ningun pin (pin_count = 0).

```
public void free( int frameNo )
```

frameNo frame number of the page.

Libera y desasocia una página en el buffer pool.

Hace un unpin en el frame de la tabla de descriptores de frame del buffer pool.

Establece su estado como Avaible.

```
public abstract int pickvictim()
```

Debe asociar el frame devuelto

```
public abstract String name();
```

Devuelve el nombre del algoritmo de remoción

```
public int getNumUnpinnedBuffers()
```

Devuelve la cantidad de frames libres (un frame está libre si pin_count = 0)

```
protected void setBufferManager( BufMgr mgrArg )
```

Asocia el el bufMgr con esta clase de remoción.

2.3. Interacción con otros componentes

2.4. Ejemplo de uso

Si se justifica

2.4.1. Diagramas de secuencia

2.4.2. Script de ejemplo

2.5. Evaluación del componente (opinión acerca de la calidad de código, diseño, etc.)

3. DiskManager

3.1. 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.

3.2. Clases principales y sus protocolos principales

3.2.1. 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.

3.2.2. 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.

3.2.3. 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.

3.2.4. Directorio de Archivos

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

- Next Page: PageID de la próxima página del Directorio. En la última página, este valor es INVALID_PAGE (-1). Ocupa 4 bytes.
- Number of Entries: Cantidad de Entries que entran en la página. Ocupa 4 bytes.
- Entry:
- PageID: PageID de la primer página del archivo. Ocupa 4 bytes.
- File Name: Nombre del archivo. Ocupa 52 bytes.
- Number of Pages: 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.

Sería útil aclarar preliminarmente 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.

Constructores

`OpenDB(fname)`

Abre un archivo de base de datos existente con el nombre fname.

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.

`OpenDB(fname, numpages)`

Crea un archivo de base de datos nuevo con nombre fname (si existe, la borra) y de num_pages páginas:

1. Crea un archivo del tamaño correspondiente y lo llena de ceros.
2. Inicializa la primer página de la base de datos utilizando pinPage del Buffer Manager sin escribir en disco. Esto lo hace para poder escribir en memoria.
3. En esta primer página, inicializa la primer página del Directorio (y de la base de datos) donde guardará los nombres (MAX_NAME = 50 bytes de largo + 4b de página donde comienza el archivo + 2b para que quede múltiplo de 4bytes) de archivos lógicos (Heap Files). Para esto, utiliza el constructor de DBFirstPage, que no es otra cosa que un DBHeaderPage (ver Páginas del Directorio).
4. Calcula el espacio que necesitará el Space Map para todas las páginas de la base. Inicializa las páginas necesarias del space map con los bits correspondientes a las páginas ya utilizadas (incluídas las del space map). Véase Mapa de Bits.

`allocatePage(PageID, int x)`

Trata de reservar en el archivo de base de datos x páginas.

1. Busca en el SpaceMap si existen x páginas contiguas en 0.
2. Si las encuentra, devuelve en PageID el número de la primer página e inicializa x páginas en el SpaceMap en 1 para reservarlas. Está de más aclarar que, para hacer todo esto, debe llamar al BufferManager para realizar las operaciones con el SpaceMap en el buffer pool.

Métodos relevantes

`closeDB()`

Lo único que hace es cerrar el archivo único de base de datos.

`DBDestroy()`

Elimina del disco el archivo de la base de datos.

`ReadPage(PageID, Page)`

Lee del disco la página PageID y la guarda en un buffer temporal al que apunta Page. Esto lo realiza con la función seek del filesystem, ya que PageID es un entero que corresponde al orden en que están las páginas en el disco. Por ejemplo, para acceder a la página 3, se realiza un seek(3 * 1024).

`WritePage(PageID, Page)`

De la misma manera en que ReadPage lee una página del disco, esta función la escribe.

`allocatepage(PageID, int n)`

Esta función se encarga de reservar n páginas (n >= 1) contiguas en el archivo de la base de datos, devolviendo la PageID de la primera reservada:

1. Necesita recorrer el Space Map para tratar de encontrar n bits contiguos en 0.
2. Luego de encontrarlos los marca como 1
3. Devuelve la PageID de la primer página de las reservadas.

`allocatepage(PageID)`

Esta función se encarga de reservar 1 página en el archivo de la base de datos, devolviendo su PageID. Llama a la anterior con n=1.

`deallocatepage(PageId, int n)`

Se encarga de marcar con 0 n páginas contiguas desde PageID.

`deallocatepage(PageId)`

Se encarga de marcar con 0 la página PageID.

`addfileentry(fname, PageID)`

Esta función se encarga de agregar una entry en el Directorio de Archivos de la base de datos con fname como nombre del archivo lógico y PageID un puntero a la primer página del archivo.

1. Recorre todas las páginas del directorio hasta encontrar alguna entry que apunte a una página inválida. Esto es gracias a que cada página del directorio se inicializa con entries que apuntan a INVALID_PAGE. Pueden darse dos casos:
   a) Encuentra: Agrega el entry con la funcion setFileEntry de DBHeaderPage.
   b) No encuentra: Debe crear una nueva página del Directorio de Páginas, y luego agregar el entry con la función setFileEntry de DBHeaderPage.

`deletefileentry(fname)`

Esta función se encarga de "borrar" el entry del Directorio de Archivos. Para realizar esto debe buscar por todas las páginas hasta encontrar el entry con ese mismo fname. Una vez encontrado debe llamar a la función setFileEntry con INVALID_PAGE y cualquier string para "anular" el entry. De esta manera al agregar en otro momento otro entry este pueda ser guardado en este lugar.

```
getfileentry(fname)->PageId
```

Esta función se encarga de devolver la primer PageID de un archivo lógico con nombre fname de la base de datos. Debe recorrer todas las páginas del Directorio de Archivos y a cada uno de sus entries; y luego devolver la PageID correspondiente a ese fname.

```
string dbname()
```

Retorna el nombre del archivo de la base de datos.

```
int dbnumpages()
```

Retorna la cantidad de páginas de la base de datos.

```
int dbpagesize()
```

Retorna el tamaño de una página de la base de datos. En este caso es fijo y esta función siempre devolverá 1024 que es el valor de la constante MINIBASE_PAGESIZE.

Aclaración: 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.

3.2.5. 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.

Constructores

```
DBHeaderPage(page, int pageusedbytes)
```

Crea una página Header inicializando los datos que corresponden. Para esto debe poner como INVALID_PAGE la página siguiente, debe calcular la cantidad de entries que va a tener. Como esta clase inicializa los datos de una página culquiera debe calcular la cantidad de entries en base a pageusedbytes. (ver Constructores de DBDirectoryPage y DBFirstPage) que son los que llaman a este constructor con el pageusedbytes correcto. Además de debe inicializar todos los entries con PageID = INVALID_PAGE, ya que de esta manera es como se consideran habilitados para ser usados.

```
DBHeaderPage(page, int pageusedbytes)
```

Crea una página Header inicializando los datos que corresponden. Para esto debe poner como INVALID_PAGE la página siguiente, debe calcular la cantidad de entries que va a tener. Como esta clase inicializa los datos de una página culquiera debe calcular la cantidad de entries en base a pageusedbytes. (ver Constructores de DBDirectoryPage y DBFirstPage) que son los que llaman a este constructor con pageusedbytes correcto. Además de debe inicializar todos los entries con PageID = INVALID_PAGE, ya que de esta manera es como se consideran habilitados para ser usados.

Métodos relevantes

`PageID getNextPage()`

Retorna la PageID de proxima página almacenada en esta página.

`setNextPage(PageID)`

Guarda la PageID de proxima página en esta Página.

`int getNumberOfEntries()`

Retorna la cantidad de entries en esta página.

`setFileEntry(pageID, fname, int pos)`

Guarda la entry en esta página en la posición pos. Con pos calcula el offset correspondiente y escribe desde ahí, primero el pageID y luego fname.

`fname getFileEntry(pageID, int pos)`

Retorna la entry en esta página en la posición pos. Con pos calcula el offset correspondiente y lee desde ahí, primero el pageID y luego fname.

3.2.6. Clase DBFirstPage

Llama al constructor de DBHeaderPage con los datos correspondientes al momento de crear la primer página de la base de datos (página 0) que es también la primer página del Diccionario de Archivos.

Métodos relevantes

`OpenPage(page)`

Inicializa la página con los datos de page.

`num getNumDBPages()`

Permite obtener la cantidad de páginas de la base de datos.

`setNumDBPages(num)`

Permite setear la cantidad de páginas de la base de datos.

3.2.7. Clase DBDirectoryPage

Llama al constructor de DBHeaderPage con los datos correspondientes al momento de crear cualquier página del Diccionario de Archivos menos la primer página de la base de datos (página 0).

Métodos relevantes

`OpenPage(page)`

Inicializa la página con los datos de page.

3.3. Interacción con otros componentes

3.4. Ejemplo de uso

Si se justifica

3.4.1. Diagramas de secuencia

4. HeapFile

4.1. Descripción general

Es un archivo desordenado. Cada registro en el archivo tiene un rid 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 ese registro dentro de dicha 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 se debe mantener un registro de las páginas que tienen espacio libre para implementar la inserción eficientemente.

4.1.1. Diseño interno-estructuras

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

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 información compatible con DataPageInfo. Es decir: 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(id de la página), la cual contiene los registros. 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).

4.2. Clases principales y sus protocolos principales

4.2.1. Heapfile

Atributos miembro ————

```
PageId firstDirPageId;
```

número de página de la página encabezado (header page)

```
int ftype;
```

```
private boolean filedeleted;
```

```
private String fileName;
```

```
private static int tempfilecount = 0;
```

Número de archivo temporal. Lo usa el constructor cuando no se pasa un nombre.

Métodos relevantes

```
private HFPage newDatapage(DataPageInfo dpinfop)
```

Obtiene una nueva página mediante el BM(éste le pide a DM que cree una página y luego la pone en el buffer pool) e inicializa dpinfop para esa página

```
private boolean findDataPage( RID rid, PageId dirPageId,
HFPage dirpage, PageId dataPageId, HFPage datapage, RID rpDataPageRid)
```

Devuelve la página del directorio, la página de datos y el rid que cumplen con lo siguiente:

la página del directorio apunta a la página de datos en dónde se encuentra el rid pasado por parámetro. Las dos págianas pinchadas(pinned)

y true si se encontró el rid, false caso contrario.

Como funciona:

1. Obtiene la primer página del directorio (_firstDirPageId) por medio del BM (pin)
2. Empieza a recorrer las páginas del directorio a partir de currentDirPage.
3. Por cada registro de la página de directorio actual(recordemos que estos registros son del tipo DataPageInfo) lo convierte en tupla (método getRecord de HFPage) y luego obtiene el id de la página que figura en DataPageInfo.
4. Llama a BM para obtener esa página.
5. Si el rid pasado x parámetro 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)

Devuelvo en dirpage y dirPageId de la página del directorio actual.

Devuelvo en datapage y dataPageId de la página de datos actual.

Y en rpDataPageRid el rid del registro.

```
public Heapfile(String name)
```

Si name es null, toma como _fileName "tempHeapFile" + "user.name"(nobre de usuario x defecto) y le concatena un número único (tempfilecount) para que este nombre no se repita.

Si se llamó con name != null, hace: obtiene el id de la primer página del directorio, llamando a get_file_entry del DM. (recordemos que se mantiene una tupla <nom_BD, 1er_pag_del_dir_de_datos> por cada archivo de base de datos>

Si no devuelve nada(nombre inválido), o se llamó con name = null

Se crea _firstDirPageId mediante BM.

Y se agregar la tupla<_fileName, _firstDirPageId> mediante DM

Se guarda (mediante el método unpin de BM) _firstDirPageId con el Id de obtenido anteriormente y con su referencia a páginas anterior y posterior en null.

`public int getRecCnt()`

Devuelve la cantidad de registros.
Por cada página del directorio, comenzando por _firstDirPageId
Obtiene la página por medio del BM (la pincha)
Cada registro en esa página (DataPageInfo), informa la cantidad de registros de cada una de las páginas de datos(dpinfo.recct)
Simplemente va sumando esas cantidades.

`public RID insertRecord(byte{[}] recPtr)`

Busca espacio disponible:
Obtiene cada página del directorio llamando al BM(pin), comenzando por la primera (_firstDirPageId)
Busca por cada DataPageInfo en la página de directorio actual uno cuyo espacio disponible sea >= que la longitud del registro a insertar.
Si no encontró espacio disponible:
Se fija si la página de directorio actual tiene espacio para alojar un nuevo registro DataPageInfo.
Si no hay lugar, busca alguna página de directorio que sí tenga. (recorriendo las páginas de directorios, recordemos que esta estructura es una lista doblemente enlazada de HFPage)
Si ninguna tiene, crea una nueva página de directorio y genera los enlaces correspondientes con la lista.
Pide al BM que aloje una nueva página de datos.
Genera un DataPageInfo para esa página.
Guarda en la página de datos el registro mediante BM(unpin)
Modifica los valores de DatePageInfo para la página de directorio mediante BM(unpin)
Si encontró espacio disponible:
Obtiene la página de datos que indica el DataPageInfo mediante el BM(pinPage)
Inserta el registro, persiste esa página mediante BM(unpin)
Modifica los valores de DatePageInfo para la página de directorio mediante BM(unpin)

`public boolean deleteRecord(RID rid)`

Borra un registro dado su rid.
Busca el registro en la BD. Si no lo encontró, devuelve false
Elimina el registro de la página de datos
Actualiza el DataPageInfo en la página del directorio
Si la cantidad de registros que tiene el DataPageInfo ahora es 0:
Elimina la pagina de datos(BM->freePage)
Elimina el registro de la pagina de directorio.
Si la pagina de directorio no es la cabecera y esta vacia, se borra.
Se actualizan los enlaces de la lista de páginas de directorio.
Devuelve true si pudo borrarlo.

`public boolean updateRecord(RID rid, Tuple newtuple)`

Modifica el registro. Se asume que el registro nuevo debe tener el mismo tamaño que el que se quiere modificar.
rid: registro a modificar
newtuple: nuevo valor
Busca el registro en la BD. Si no lo encontró, devuelve false
Modifica el registro de la página de datos
Saca las páginas(de datos y de directorio) de memoria principal.

`public Tuple getRecord(RID rid)`

Lee el registro de dado un rid
Busca el registro en la BD. Si no lo encontró, devuelve null
Lee el registro de la página de datos

`public Scan openScan()`

Inicializa un scan secuencial

Borra el archivo de la BD
Por cada página del directorio
Borra todas las páginas de datos contenidas en cada DataPageInfo de esa página del directorio
Borra la página del directorio
Borra la entrada para ese archivo; la tupla<_fileName, _firstDirPageId>

`pinPage->BM.pinPage`

`unpinPage->BM.unpinPage`

`freePage->BM.freePage`

`newPage->BM.newPage`

`getfileentry->DM.getfileentry`

`addfileentry->DM.addfileentry`

`deletefileentry->DM.deletefileentry`

4.2.2. DataPageInfo

Esta clase se utiliza para guardar información sobre las páginas de datos (las que realmente contienen los registros). Esta información se guarda en las páginas de directorio(que como dijimos anteriormente, es una lista doblemente enlazada). Entonces, una página de directorio almacena información sobre

muchas páginas de datos como una entrada DataPageInfo. Esta información es la siguiente:

```
int availspace;
```

Espacio disponible

```
int recct;
```

Indica la cantidad de registros en

```
PageId pageId = new PageId();
```

Id de HFPage

4.2.3. HFPage

El diseño asume que los registros se compactan cuando se realiza un borrado.

Se utilizan slots al principio que indican la informacion del registro(longitud, offset, tipo, 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.

Atributos miembro

```
private short slotCnt;
```

número de slots en uso

```
private short usedPtr;
```

offset del primer byte usado por registros en el arreglo de bytes(data)

```
private short freeSpace;
```

número de bytes libres en data[]

```
private short type;
```

an arbitrary value used by subclasses as needed

```
private PageId prevPage = new PageId();
```

referencia a la página anterior

```
private PageId nextPage = new PageId();
```

referencia a la próxima página

```
protected PageId curPage = new PageId();
```

número de esta página.

Métodos relevantes

```
public void openHFpage(Page
apage)
```

Establece: 0 slots en uso, pageNo como página actual, pág anterior y próxima como INVALID_PAGE, espacio libre: MAX_SPACE - DPFIXED, USED_PTR =MAX_SPACE

Los siguietnes métodos obtienen y guardan los atributos de la clase fisicamente en la pagina (el arreglo de bytes data)

```
getHFpageArray()
```

Obtiene los valores del arreglo de bytes(data) y los guarda en los atributos

```
 public void dumpPage()

public PageId getPrevPage()

public void setPrevPage(PageId pageNo)

public PageId getNextPage()

public void setNextPage(PageId pageNo)

public PageId getCurPage()

public void setCurPage(PageId pageNo)

public short getType()

public void setType(short valtype)

public short getSlotCnt()

public void setSlot(int slotno, int length, int offset)

public short getSlotLength(int slotno)

public short getSlotOffset(int slotno)

public int availablespace()

public boolean empty()
```

Dice si la página está vacía.

Para esto se va fijando en cada uno de los slots si está vacío.

`public RID insertRecord ( byte {[}] record)`

Inserta un registro nuevo en la página, devuelve el rid de este registro

1. Se fija si hay lugar disponible para el registro nuevo, comparando el tamaño de este registro con el valor de FREE_SPACE(guardada en la página)

   Si no hay lugar devuelve null

2. Busca un slot marcado como borrado (longitud negativa)

   Si lo encuentra, le resta a FREE_SPACE el tamaño del registro.

   Si no, usa uno nuevo; le resta a FREE_SPACE el tamaño del registro y además aumenta la cantidad de slots en uso (SLOT_CNT)

Modifica la dirección (USED_PTR -= long del registro.)
Inserta la información del slot en la página: numero de slot, longitud del registro, dirección (USED_PTR)
Inserta el registro en la página
Arma el rid con el número de slot y id de la página

`public void deleteRecord ( RID rid )`

Elimina el registro especificado por su rid
Obtiene el número de slot por su rid.
Obtiene el offset, usedPtr y tamaño
Elimina el registro (hace un shift hacia la derecha, de todo lo anterior a ese registro)
Actualiza los offsets de todos los registros anteriores a éste.
Actualiza la ubicación de usedPtr
Incrementa el espacio libre de la página.
Marca el slot como libre

`public RID firstRecord()`

Devuelve el RID del primer registro de la página o null si no hay ningún registro en la página.
Busca el primer slot que no esté vacío.
Arma el rid con el id de la página y el número de slot.

`public RID nextRecord (RID curRid)`

Obtiene el rid del próximo registro de la página o null si no hay otro.
Idem anterior, con la diferencia que:
Obtiene el slot de este registro por medio del rid.
Y comienza a buscar a partir de ese slot, el próximo válido.

`public Tuple getRecord ( RID rid )`

Copia el arreglo de bytes(data) del registro dado un RID en una tupla.

4.2.4. Scan

Atributos miembro

```
private Heapfile hf;
```

El heapfile

```
private PageId dirpageId = new PageId();
```

Id de la página actual del directorio (HFPage)

```
private HFPage dirpage = new HFPage();
```

Pagina actual del directorio (HFPage)
(la pág. apuntada por dirpageId)

```
private RID datapageRid = new RID();
```

rid que apunta al DataPageInfo de la página del directorio

```
private PageId datapageId = new PageId();
```

página id en donde está contenido el rid

```
private HFPage datapage = new HFPage();
```

página en donde está contenido el rid

```
private RID userrid = new RID();
```

rid contenido en la actual pagina de datos(datapage)

```
private boolean nextUserStatus;
```

/** Status of next user status */

Métodos relevantes

```
public Scan(Heapfile
hf)
```

Inicializa el miembro privado hf con el hf pasado por parametro.
Establece el id de dirPageId como el id de la primer página del directorio de hf.
Obtiene esta página mediante el BM y la guarda en dirPage

```
public Tuple getNext(RID rid)
```

Devuelve el proximo registro.
Si no hay un proximo registro en esta pagina (nextUserStatus = false)
obtiene la proxima pagina
Obtiene la tupla correspondiente y guarda en userrid el proximo registro.

`public boolean position(RID rid)`

Ubica el cursor (userrid) en el registro pasado por parametro.

Se fija si la pagina de datos actual coincide con el id de la pag. del rid pasado x parametro.

Si no, busca esa pagina a partir de la primera apuntada por la primer pagina del directorio. Utiliza el metodo (nextDataPage())

Una vez encontrada, comienza a buscar los registros de esa pagina hasta encontrar el que coincida con el pasado por parametro. Utiliza el metodo (peekNext)

`public void closescan()`

Le indica al BM que desaloje la pagina de directorio y la de datos.

`private boolean firstDataPage()`

Establece dirpageId como el id de la primer página de directorios del HF.

Obtiene esta página a partir del BM y la guarda en dirpage

nextUserStatus = true;

Devuelve true si todo estuvo bien

`private boolean nextDataPage()`

Obtiene la próxima página de datos apuntada por datapageRid.

Si la pagina de datos actual es nula y el dataPageId es válido, obtiene y devuelve esta página.

Obtiene el siguiente registro a datapageRid de la pagina de directorio, si es nulo, busca el primer registro de la proxima pag. de directorio si hay.

A partir de este registro, obtiene la pagina de datos.

Guarda los valores nuevos de todos los atributos.

Devuelve true si estuvo todo bien

`private boolean peekNext(RID rid)`

Guarda en rid el proximo registro (establecido en el miembro userrid)

4.3. Interacción con otros componentes

4.4. Ejemplo de uso

Si se justifica

4.4.1. Diagramas de secuencia

4.4.2. Script de ejemplo

4.5. Evaluación del componente (opinión acerca de la calidad de código, diseño, etc.)

5. Catalogo

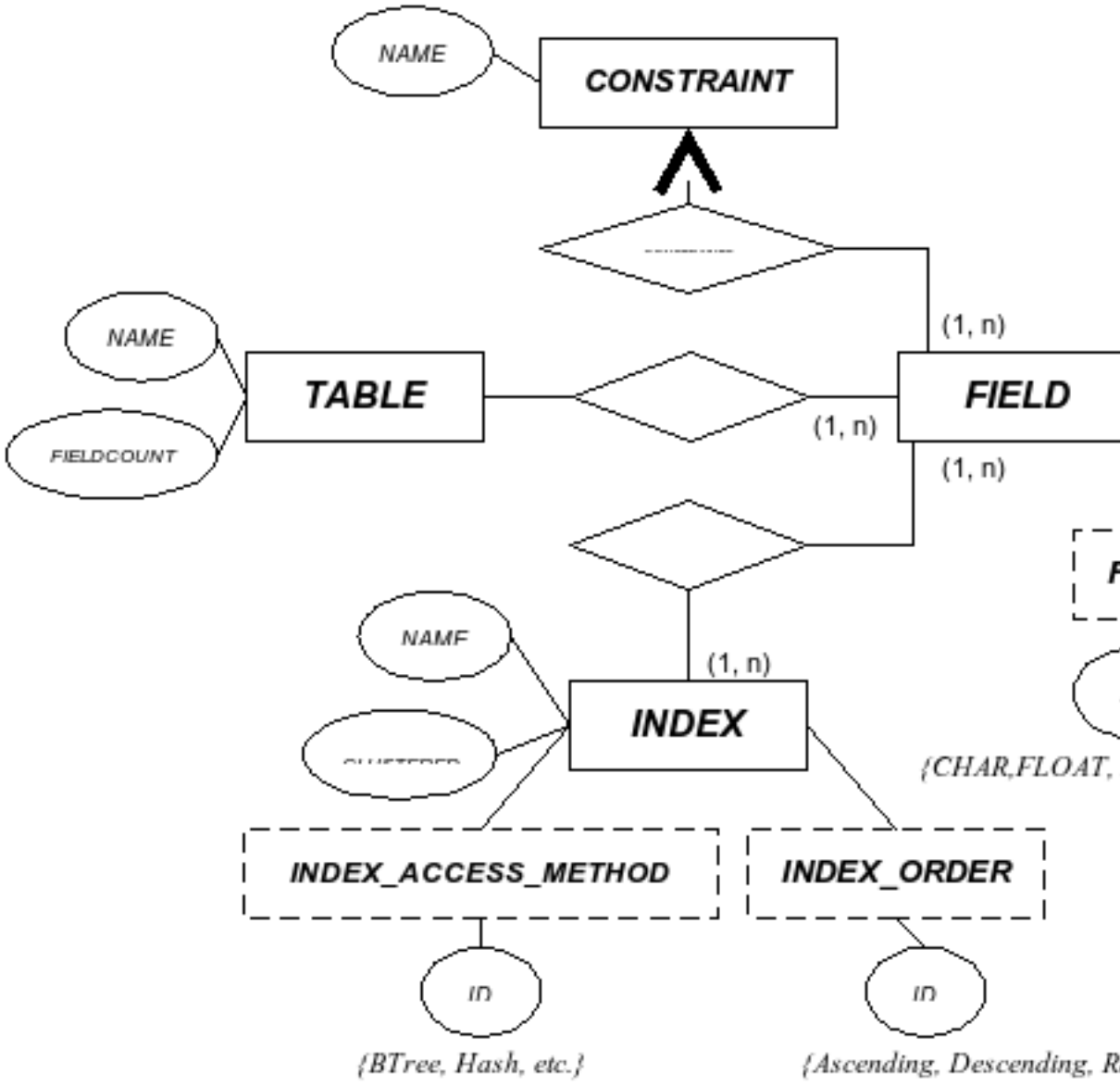
5.1. Descripción general

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

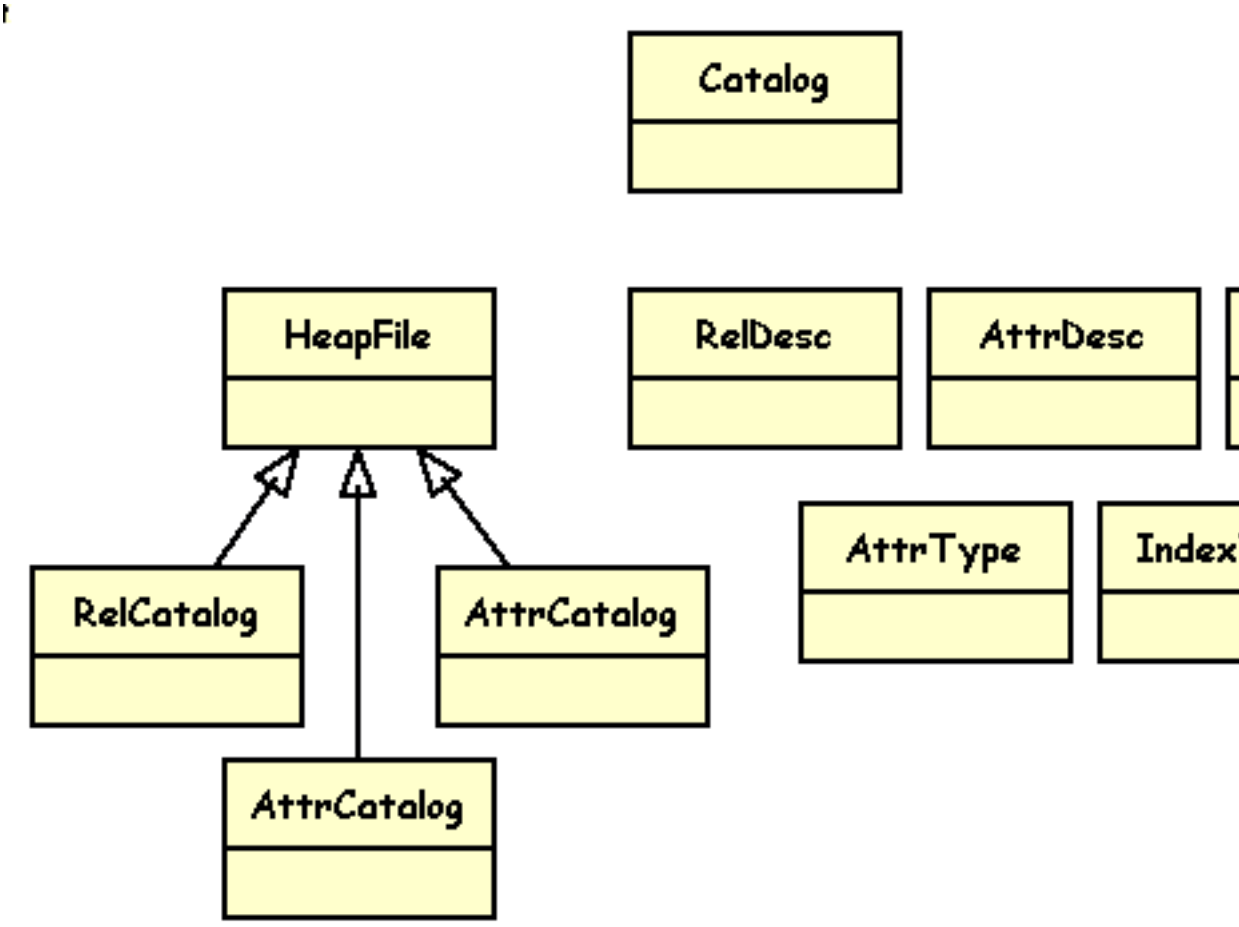
5.2. DER

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



5.3. Clases principales y sus protocolos principales

Las clases principales del catálogo son las siguientes:



5.3.1. Catalog

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

5.3.2. 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.

5.3.3. 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

5.3.4. 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).

5.3.5. AttrType

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

5.3.6. 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)

5.4. attrData

//attrData class for minimum and maximum attribute values

5.4.1. TupleOrder

attrInfo // class used for creating relations

5.4.2. attrNode

5.5. Interacción con otros componentes

Falta?

5.6. Ejemplo de uso

Si se justifica

5.6.1. Diagramas de secuencia

5.7. Evaluación del componente (opinión acerca de la calidad de código, diseño, etc.)

Falta?

6. Iterator

TODO: Revisar los arboles binarios como estan construido y hacer un analisis mas profundo

TODO: Analizar mas profundamente las clases de eval y projection, y en la parte de uso explicar como se introduce una condicion

6.1. 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.

6.2. 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.

6.2.1. 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.

6.2.2. 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.

6.2.3. 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.

6.2.4. 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.

6.2.5. 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.

6.3. 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.

6.4. Ejemplo de uso

TODO: algun ejemplo de uso q muestre claramente los detalles

6.4.1. Diagramas de secuencia

No se justifica

6.5. 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.

7. Index-BTree

7.1. Descripción general

7.2. Clases principales y sus protocolos principales

7.3. Interacción con otros componentes

7.4. Ejemplo de uso

Si se justifica

7.4.1. Diagramas de secuencia

7.4.2. Script de ejemplo

7.5. Evaluación del componente (opinión acerca de la calidad de código, diseño, etc.)