NEUStore: A Simple Java Package for the Construction of Disk-based, Paginated, and Buffered Indices

Donghui Zhang *
College of Computer and Information Science
Northeastern University
donghui@ccs.neu.edu

September 14, 2005

1 Introduction

Suppose you are a professor and you want your students to implement several disk-based index structures. This could happen if you want the students in your database class to implement some basic index structures such as the B+-tree and the linear hashing. Alternatively, you may want your Ph.D. students to implement some novel indices for research purposes. We know there are some common features of the disk-based indices.

- Such an index needs to be stored on disk, e.g. as a random access file.
- The file should be paginated. Each disk access reads/writes one disk page.
- A memory buffer should be utilized.

Naturally, you want to give your students a package that implements these common features. The benefit is that the students can focus on implementing the key ideas of the index structures, without being distracted by the tedious, low-level details of file, pagination and buffer.

This paper describes such a package: the **Northeastern University Storage Package (NEUStore)**. It was implemented in Java by the Database Lab in Northeastern University.

The rest of the paper is organized as follows. Section 2 shows how to download the NEUStore package and how to set up the Eclipse development environment. Section 3 provides an overview of the package. Section 4 discusses the three core classes. Section 5 describes a naive heap file and a generic, full-blown heap file to illustrate how to use the NEUStore package. Finally, Section 6 concludes the paper.

2 Download and Setup

The NEUStore package can be downloaded from its official web site –

http://www.ccs.neu.edu/home/donghui/research/neustore

The downloaded file *neustoreroot.tgz*, if extracted, contains one directory *neustoreroot*, and everything else is inside of it.

*Partially supported by NSF CAREER Award IIS-0347600.

It is recommended that you use Eclipse (http://www.eclipse.org/) to manage the package. The remaining part of this section discusses some basics of Eclipse. Feel free to skip them if you choose not to use Eclipse or if you are an experienced Eclipse user.

2.1 Create Project

1. In the *File* menu, click *New*, then select *Project*.
2. Make sure "Java Project" is selected. Click *Next*.
3. Select "Create project at external location".
4. Click *Browse* and select the extracted *neustoreroot* directory.
5. Click *Finish*.

2.2 Support the 'assert' Statement

It is possible that at this point Eclipse complains that it does not understand the 'assert' statement. To solve this problem, you would use "*javac -source 1.4 ...*" if you were using command-line compiler. In Eclipse, what you should do is:

1. In the *Window* menu, click *Preference*.
2. Expand *Java*, and select *Compiler*.
3. Select the tab "Compliance and class files".
4. Uncheck "Use default compliance settings".
5. Make sure that the value is "1.4" at all the following three places: "Compiler compliance level", "Generated .class files compatibility", and "Source compatibility".
6. Click *Apply* and then *OK*.

If you still have problems, you should click the menu item *Project* → *Properties* and follow steps 2 to 6 above.

2.3 Run A Program

Each of the two sub-directories in the *test* directory contains a runnable program. To run a program for the first time, do the following:

1. In the *Run* menu, click *Run*.
2. Select "Java Application", and click *New*.
3. Change the content of the *Name* field to "TestNaiveHeapFile" (or any name you want).
4. Near "Main Class", click *Search* and choose "TestNaiveHeapFile".
5. Click *Apply* and then *Run*.

Note that if you want to run the program again, you can simply click the *Run* icon.

2.4 Use Javadoc

The directory *neustoreroot/doc* contains documents about the public class members of the public classes. These documents were generated using *Javadoc*. In order to generate documents at other protection levels, one needs to know how to use *Javadoc*. Here are the steps.

1. In the *Package* window, select *neustoreroot*. This is an important step because you may have clicked other things in the *Package* window, e.g. you may have double clicked a *.java* source file to view it. If you do not select *neustoreroot*, Javadoc will still run but produce inconsistent result.
2. In the *Project* menu, select "Generate Javadoc".
3. Near "Create Javadoc for members with visibility", choose *Protected*.
4. Click *Next*, and click *Next* again.
5. Check "JRE 1.4 compatibility".
6. Click *Finish*.

To view the generated documents, in the *Package* window, expand *doc*, and then double click *index.html*.

3 Overview of the NEUStore Package

The extracted directory *neustoreroot* contains the following sub-directories:

- **doc:** the document generated using Javadoc.
- **neustore:** the storage package. It contains two sub-directories:
  - **base:** the base of the storage package. It contains definitions of classes *DBIndex*, *DBPage*, *DBBuffer*, and so on.
  - **heapfile:** a generic implementation of the heap file. Here the class *HeapFile* was derived from *DBIndex*, and *HeapFilePage* was derived from *DBPage*.
- **test:** sample test programs. It contains two sub-directories:
  - **naiveheapfile:** test program for the naive heap file. This directory contains a full implementation of a naive heap file. It demonstrates how to design a new index structure by deriving classes from *DBPage* and *DBIndex*.
  - **heapfile:** test program for using *neustore.heapfile*.

The four sub-packages (*neustore.base, neustore.heapfile, test.naiveheapfile, test.heapfile*) and their public classes are shown in Figure 1, where the arrows represent the superclass → subclass relationship.

One may wonder why, while the implementation of the heap file is in the *neustore* folder, the implementation of the naive heap file is in the *test* folder. The reason is that the naive heap file is not meant to be actually used. It is provided only to illustrate how to use the NEUStore package.

4 The Usage of *DBPage*, *DBBuffer*, and *DBIndex*

This section discusses the three most important classes as shown in bold face in Figure 1.

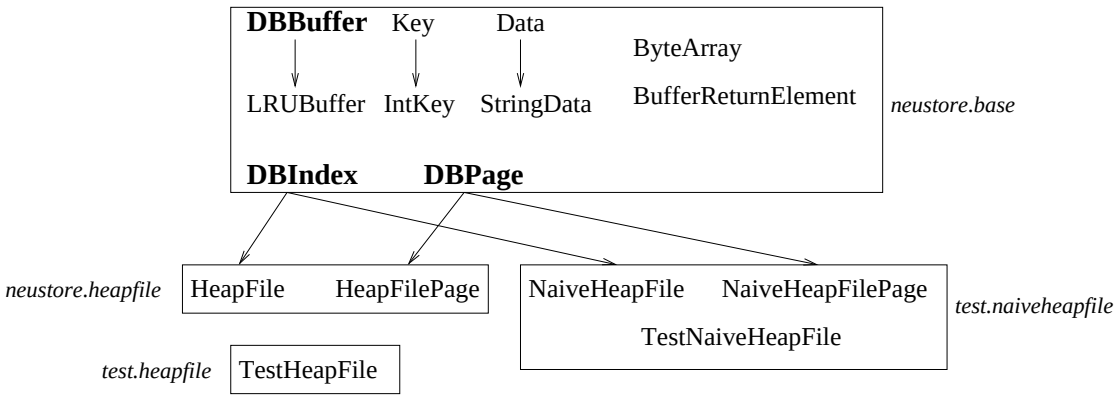

Figure 1: Overview of the NEUStore classes.

4.1 Class *DBPage*

Class *DBPage* is an abstract base class for a memory-version disk page. For example, to implement a B-tree index, one may derive from *DBPage* two sub-classes: *BTreeIndexPage* and *BTreeLeafPage*. A derived class can determine what information to store in this object. For instance, a *BTreeLeafPage* may store a list of records.

Public and abstract member functions of *DBPage*.	
	DBPage(int nodeType, int pageSize) Creates a DBPage.
protected abstract void	read(byte[] page) Reads the content of this DBPage from a byte array.
protected abstract void	write(byte[] page) Writes the content of this DBPage to a byte array.

The constructor of the *DBPage* takes as parameter a node type and a page size.

A disk-version disk page is a byte array. In order to transform a page from its disk version to memory version or the other way around, the class *DBPage* contains two protected abstract functions *read* and *write*. They are to be implemented by the derived classes.

To facilitate the handling of byte arrays, in *neustore.base* we provide a class *ByteArray*. The following code segment illustrates its usage, given a byte array *b*:

```
ByteArray ba = new ByteArray( b, ByteArray.READ );
int firstInt = ba.readInt( );
int secondInt = ba.readInt( );
```

A small issue here is that before calling *DBPage.read* to read the content of an *DBPage* object, one typically wants to know what type of object to generate in the first place. Notice that *DBPage* is an abstract class and thus to generate such an object one has to know the actual derived class. For instance, in a B-tree application, if we get a page from disk (a byte array), we need to know whether to generate a *BTreeIndexPage* object or a *BTreeLeafPage* object, before calling the object's *read* member function.

For this reason, we enforce that in the disk version of every disk page, the first four bytes store the node type – a positive integer chosen by the user. For instance, the user may choose to use 1 to represent an index page and use 2 to represent a leaf page. When a new disk page is read from disk, by examining the first four bytes, the user instantly knows whether this is a leaf page or an index page.

4.2 Class *DBBuffer*

Class *DBBuffer* is an abstract base class for a buffer. It utilizes one random access file as the underlying storage, and organizes the file by disk pages. It is used to support a *DBIndex*.

Public member functions of *DBBuffer*.	
	DBBuffer(String filename, int bufferSize) Opens an existing file.
	DBBuffer(String filename, int bufferSize, int pageSize) Creates a new file.
DBBufferReturnElement	readPage(int pageID) Returns a buffered page.
void	writePage(int pageID, DBPage dbpage) Writes a DBPage to buffer.
void	pin(int pageID) Pins a page.
void	unpin(int pageID) Unpins a page.
void	clearIOs() Resets the I/O statistics.
int[]	getIOs() Reports the I/O statistics.

In the above table we only list the public member functions (but not the abstract members as in the *DBPage* case). The reason is that the users typically do not need to derive a sub-class of *DBBuffer* and therefore do not need to know about the abstract classes. Instead, they can simply use the derived class *LRUBuffer* which is included in the NEUStore package.

There are two constructors, one opening an existing file and the other creating a new file. Both constructors take as input a file name and a buffer size (number of memory pages the buffer has). The difference is that the creation constructor takes an extra parameter: page size. Needless to say, the open constructor does not need it because the page size can be read from the existing file.

Among the remaining member functions, the most useful ones are *readPage* and *writePage*. To read or write disk pages, a user sees only the buffer and not the disk. That is to say, every time the user needs to read a disk page, she should simply ask the buffer for it; and every time the user has changed the content of a page, she should write it to buffer. The buffer takes care of the low-level details. An example of such details is that when a disk page is needed, the buffer will check whether it is in memory or not, and only reads the disk if the page is not currently in memory. Moreover, if a disk page is to be read into memory while the buffer is full, some buffered page should be selected and swapped out.

One question that arises here is: what do we store in the buffer for one disk page? In particular, shall we store a *DBPage* object or a byte array? It seems that we have to choose from efficiency and generality. If we store *DBPage* in the buffer, It seems that the buffer is specific to the application. That is, once the buffer reads in a byte array from disk, it should know which derived class of *DBPage* it corresponds to (e.g. is it a BTreeIndexPage?), in order to create a *DBPage* object from it. As a result, one is forced to derive a new class from *DBBuffer* for every index structure. This is certainly not good since we want the *DBBuffer* class to be general. On the other hand, if we store byte arrays in the buffer, every time the application needs to access a page it will need to generate a *DBPage* object from the byte array. This approach has expensive CPU cost. The challenge here is whether we can achieve both generality and efficiency.

The answer is yes. The idea is that a buffered disk page can be stored as either a byte array or a *DBPage*. In more detail, when a disk page is first read into memory, it is simply stored as a byte array in the buffer. This means that the buffer is general, since it does not need to know which derived class of *DBPage* the disk page corresponds to. When the application receives the byte array, it generates a *DBPage* object. Later

when the application calls *DBBuffer.writePage*, the buffer will store the *DBPage* object being passed, which can be returned to the application once the page is needed again. It can be seen that, besides being general, this approach is also efficient. This is because we only need to translate a byte array to a *DBPage* object once (as long as the page is not swapped out).

An implementation detail is: how to implement *DBBuffer.readPage* such that it can return either a byte array or a *DBPage*? Our answer is to define the following class as the return type of *DBBuffer.readPage*:

```
public class DBBufferReturnElement {
    public boolean parsed;
    public Object object;
    public int nodeType;
}
```

If *parsed=true*, the returned object is a *DBPage* object. Otherwise, the returned object is a byte array and the application should generate a *DBPage* object from it.

The remaining four public member functions of *DBBuffer* are *pin* and *unpin* to pin/unpin a page in memory, and *getIOs* and *clearIOs* to read/reset the statistics on how many I/Os have occurred. Although not as crucial as *readPage* and *writePage*, these member functions may also be needed.

Similar to *DBPage*, the class *DBBuffer* is also abstract. A derived class should implement the following three abstract functions:

- *add* : to add a new page into buffer;
- $find$: to find a buffered page;
- $flush$: to empty the buffer while writing dirty pages to disk.

Once again, an application may choose to use the provided *neustore.base.LRUBuffer*, without deriving a sub-class from *DBBuffer*.

4.3 Class *DBIndex*

The class *DBIndex* is an abstract base class for a disk-based, paginated and buffered index. The user-defined index structure, e.g. a B-tree and a linear hashing, are supposed to be sub-classes of *DBIndex*.

Public and abstract member functions of *DBIndex*.	
	DBIndex(DBBuffer buffer, boolean isCreate) Creates or Opens an index file with a buffer.
int	allocate() Assigns a new pageID.
void	freePage(int pageID) Frees an empty page.
void	close() Saves the header page to disk and flushes the buffer.
protected abstract void	readIndexHead (byte[] indexHead) Reads index head information from a byte array.
protected abstract void	writeIndexHead (byte[] indexHead) Writes index head information to a byte array.
protected abstract void	initIndexHead() Initializes index head information.

The constructor of *DBIndex* takes a *DBBuffer* object. That means the caller who wishes to generate a *DBIndex* object should generate a *DBBuffer* object first. The *DBIndex* object relies on the *DBBuffer* object to manage the disk storage.

An application may call *allocate* to get the pageID of a new page, or call *freePage* to declare a page to be empty. Upon the calling of *allocate*, if there exist some empty page, the pageID of one of them will be returned. Otherwise, a new page at the end of the file is returned.

One implementation detail is that all the empty pages in the file are linked together in a linked list. An empty page contains two integers: 0 and pageID for the next empty page. If it is the last empty page in the file, the next pointer is set to -1. A private class called *DBEmptyPage* is defined as a memory version of an empty page. Of course, the users do not need to know these details in order to use *DBIndex*.

The only other public member function of *DBIndex* is *close*. This should be called before the application terminates, to ensure that the buffered pages are flushed to disk.

Do not forget that the public member functions of *DBBuffer* are also available. For instance, a class derived from *DBIndex* may call *buffer.readPage* and *buffer.writePage*. Here *buffer*, an object of class *DBBuffer*, is a public field of *DBIndex*.

Before discussing the abstract member functions of *DBIndex*, let's first define the term *head information*. The file can be viewed as a consecutive list of disk pages, with equal size. The page at the beginning of the file is the head page. It uses the first OVERHEAD=16 bytes to store four integers: -1, page size, number of non-empty pages, and pageID of the first empty page. The rest (page size - OVERHEAD) bytes of the head page is used to store the user-specified *head information*. For instance, a tree index may store the pageID for the root page here. To manage the head information, a derived class needs to implement the following three abstract functions:

- initIndexHead: called when the index is built, to perform initialization.
- readIndexHead: called when the index is opened, to read the head information from disk.
- writeIndexHead: called right before the index is closed, to write the head information to disk.

5 Case Studies

5.1 The Naive Heap File

To illustrate the usage of the NEUStore package, we have provided the implementation of a naive heap file. It stores a list of integers. The only update is to append an integer at the end of the file. The only search is to tell whether an integer exists in the file or not.

The class *NaiveHeapFile* is derived from *DBIndex*. The class *NaiveHeapFilePage* is derived from *DBPage*. The readers are suggested to go through the source code, and then play with the runnable class *TestNaiveHeapFile* which tests the index.

One can easily identify some **limitations** of the naive heap file:

1. There is only key and no data for a record.
2. No deletion is allowed, and an insertion only goes to the end of the file.
3. The type of the key is fixed to be an integer. People may want to use other keys such as a String. A generic implementation of the heap file should allow the users to specify the type of the key after they get the heap file implementation. Similar for the data part.

HALT! The readers are suggested to put aside the paper temporarily and try to implement, as an exercise, a generic and full-blown heap file, which will be discussed in the next section.

5.2 The Heap File

The package *neustore.heapfile* contains a generic, full-blown heap file. The readers can learn it by studying the source code and modify/run the tester program in *test.heapfile*. Here we discuss some key ideas of it, by addressing the three limitations of the naive heap file.

To address the first limitation is easy. We simply define a private class *HeapFileRecord*, inside class *HeapFilePage*, which contains a key and a data.

To address the second limitation is a little more complex. The thing is, if we allow deletion, some page in the middle of the file will become non-full. Therefore insertions should try to fill the non-full pages instead of always going to the end of the file. What we do is to maintain a double linked list of full pages and a separate double linked list of non-full pages.

The third issue is the most challenging one. How can we implement the heap file first and allow the users to specify the types of key and data later?

What we did was to define two interfaces Key and $Data$, requiring that any record key be belonging to some class derived from Key, and any record data be belonging to some class derived from $Data$. We also provide implementations of $IntKey$ and $StringData$, which are an integer key derived from Key and a variable-length string data derived from $Data$, respectively.

Two important member functions of both Key and $Data$ are:

- `public abstract void read(DataInputStream in):`
  which reads the content of the Key object from an input stream;
- `public abstract void write(DataOutputStream out):`
  which writes the content of the object to an output stream.

These functions enable the transformation between a byte array and an *HeapFilePage* object (which stores a list of [Key object, $Data$ object] pairs).

There remains a challenging issue. Let's say we have read a disk page to memory as a byte array, and have defined a *DataInputStream* on the byte array. Now we want to create a *HeapFilePage* object. We will need to generate a Key object and call *read* function of this object. The problem is: how can we generate the Key object? Remember we are now inside the implementation of the generic heap file, and we have absolutely no idea what class the user may derive from Key.

Our solution to this challenge is that we require the constructor function of *HeapFile* (and also *HeapFilePage*) to take as input a sample key and a sample data. In the above scenario when a new Key object is needed, we call the *clone* member function of Key.

Acknowledgement: Thank Tian Xia for participating in coding.

6 Conclusions

This paper described NEUStore, a Java package that aims to support the development of disk-based, paginated, and buffered index structures. It started with an overview figure with 16 classes. Three core classes were discussed in-depth. As case studies the naive heap file and the heap file were presented. The discussion of the other classes were weaved into the places when they are needed. Some challenges, especially on how to make the implementations generic, were identified and addressed. We believe the NEUStore package can help people understand the basics of developing disk-based index structures and can help save development time.