Andrew Dawson
1220796

Lab 1 was about setting up the basic infrastructure of our simpleDB. We started by defining some simple classes Tuple, and TupleDesc which provide means for us to deal with records and schemas. Then we moved on to creating the catalog which holds information about the tables in the database. It is through the catalog that we access information about tables such as table names and DBFiles. The catalog is accessed through a global static instance in Database.java. Next we implemented BufferPool. BufferPool is basically just an in memory cache of pages that have been recently read from disk. Operators access pages through BufferPool to take advantage of caching. In BufferPool we had to write the getPage method. This is the method that operators use to get pages. If the requested page has already been cached it is simply returned, otherwise BufferPool uses a DBFile to read a page from disk. Next we worked on HeapFile and the classes associated with it. HeapFile is a type of access methods. Access methods are simply operators that read information from disk. A heap file is an access method in which the data is not stored in any order in disk. A HeapFile is made of many HeapPages. A HeapPage is basically just a collection of tuples. A HeapPage additionally stores some information about if tuples are valid or not - this information is stored in the header of the HeapPage. HeapPage has some functionality to check how many pages are empty, to iterate over the tuples in a page and check if a slot is empty. The primary method we had to write in HeapFile is readPage. The readPage method simply reads a page from disk. Given a PageId, the HeapFile computes the byte offset of that page, creates a byte array of page size length and reads page size bytes into the byte array. Then using this byte array HeapFile is able to construct a new HeapPage and return it. This returned page will be used by BufferPool and be cached for future accesses. HeapFile also has to be able to iterate over all the tuples in it. This is a little tricky for two reasons. Firstly, a heap file is made up of several pages so you have to iterate over the tuples in a page, then change out iterators once you have finished iterating over the current page. Secondly, if the table gets really big not all the tuples can fit in memory so you have to load only a single page into memory at a time rather than the whole DBFile. The last major component that we had to implement was ScanSeq, this is a simple operator that iterates over all tuples in a HeapFile. To my understanding ScanSeq is more or less a wrapper around HeapFileIterator.

There were not very many design decisions to make in lab 1. The two that I did make both revolved around iterators - the HeapPage iterator and the HeapFile iterator. Implementing the HeapPage iterator was a little tricky because we had to iterate over only the valid tuples and remove needed to throw an exception. So in order to handle this I built an array of all the valid tuples, converted it to a list, and returned that list's iterator. This works because the return value of array.asList is an immutable list so when you get the iterator for this the remove operator is unsupported. This was a nice tool that I used to avoid having to make a HeapPageIterator class. The logic for the HeapFile iterator was much more complex. The HeapFile iterator had to keep track of page iterators and swap out the iterators once they had been fully used. The HeapPageIterator extended the AbstractDbFileIterator. This was a useful abstract class

because it provided the next and hasNext functionality (simply dependent upon me implementing readNext()). Within DbHeapFileIterator I made a very small design decision to factor out the logic for switching out the iterator because both open and readNext() used this logic - so by creating a private helper method I was able to reuse code. I also made a design decision not to pass the HeapFile to the DbHeapFileIterator. The DbHeapFileIterator could have used the logic in HeapFile, but I did not want to add this dependency in my code. So instead I simply based the necessary fields to DbHeapFileIterator.

One unit test that could have been added was checking that the right type of error was thrown when HeapPage.iterator.remove() was called. I was throwing the wrong exception by mistake at one point and the tests did not catch this.

I did not make any changes to the API.

I do not think that there are any missing or incomplete sections of my code.

I think I spent about 10 hours in full on the lab. The only thing I found confusing was the pageNumber field of HeapPageId. I did not understand what this field was and once I knew what it was I had to do some digging to figure out if it was based on 0 indexing or 1 indexing. I also had a horrible bug in HeapFile. I was using File instead of RandomAccessFile. This bug was extremely hard to find and I spent the majority of the project time finding it. One thing that went really well was understanding the general architecture of simpleDB. Between the work we did in section, the spec and the class work - understanding how all the pieces fit together went very smoothly.

Overall this lab was enjoyable and reasonable to understand.