Roger Lee (704018489)
Andrew Shih (704032348)

Write up: CS 143-Lab2

We took the code from our submission from lab 1 and downloaded the additional documents like the spec suggested. In total, we each spent about about 10 hours on this project, for a combined estimated 20 hours.

Filter and Join

For this Filter, it just matches the child iterator with its appropriate predicate to write operate over. Most of this class was straight forward except for the return types of getting the children and setting the children. However, we searched on piazza and found that someone else had had the same question we did and the answer gave us a good explanation on the correct way to implement those functions.

For join, we use the nested-loops join. Join’s fetchNext was slightly difficult to implement because it had to keep track where the previous call left off. We could not just use two simple nested loops, without keeping track whether to call next for the outer loop.

For all classes that extend Operator, we kept forgetting to put super.open(), and super.close(), which caused some headaches when debugging.

Aggregates

Our IntegerAggregator and StringAggregator had an aggregate hashmap and a count hashmap. This design choice was to implement the AVG operator (AVG = SUM / COUNT) , in which the aggregate hashmap would keep track of the sum and the count would be tracked in the count hashmap. The other aggregates were pretty straight forward.

HeapFile Mutability

To insert and delete in HeapPage, the tuple is either inserted or removed respectively and then the page header is updated appropriately.

In HeapFile, insertTuple checks its freemap for pages that have empty slots. It will grab the pid of the first free page it finds or it will create a new empty page if none have free slots. The page is retrieved through the BufferPool and then the tuple is inserted with HeapPage’s insrtTuple function. Freemap is updated. Page is marked dirty.

DeleteTuple retrieves the page through the BufferPool and then deletes the tuple form the Page with HeapPage’s deleteTuple function. Freemap is updated. Page is marked dirty.

HeapFile’s WritePage was implemented similarly to ReadPage except that page data is written to the RandomAccessFile at the specified offset.

The insertTuple and deleteTuple functions in BufferPool just called their counterpart functions in HeapFile.

Insertion and Deletion

Insert and delete were just operator iterators that implement DbIterator. Their implementations were very straight forward and similar to the functions in filter. The main point of confusion was implementing fetchNext() function. We were not allowed to change the header of the function since it was extending the Operator class, while at the same time we had to call the buffer pool’s insertTuple and deleteTuple function. Since both of these functions throws an IOException, we had to catch the exception so that it would not get thrown. And if that exception were to be caught, we simply exit.

Page Eviction

For page eviction in buffer pool, we used the least recently used eviction policy. We implemented this policy using a linked list that acted as a queue. If a page was read into the bufferpool, its hash code is added to the end of the linked list. If the page read is already in the bufferpool, then its hash code is removed from the linked list then added to the end. When the bufferpool is full and needs to read in another page, it evicts the page at the head of the linked list because that is the page that was least recently used page.

FlushPage just simply wrote the page to the disk.
FlushAllPages iterates over the pages in the BufferPool and calls FlushPage on all the dirty pages.