**Programming Assignment #3**

**Writeup**

**Name:** Ankur Ankit

**BUID:** U68673460

**Predicate.java:** This file only had some getter/setter methods and made use of Field's compare method

**JoinPredicate.java**: This file was also very simple and made use of Field's compare function

**Filter.java:** The only nontrivial function here was fetchNext, which iterated through the child iterator, applying the predicate filter function to see the selection.

**Join.java:** The join is implemented as a simple nested loops join, which is slow, but was easiest to implement and for me to understand. The join is done by iterating through the values of the column of the first join field and using the predicate to determine if it should join based on the join field on the right. If so, a new TupleDesc is made and the two corresponding tuples are merged.

**IntegerAggregator.java, StringAggregator.java:** The IntegerAggregator was implemented using a HashMap, mapping the group to values. The mergeTupleIntoGroup function would add the new tuple values into the map in constant time, which easily keeps track of things like Sum and Count. For Min and Max, the values currently in the map Ire compared against them to see if they would be replaced. The slightly trickier one to deal with was Average since you can't compute it until you know the total number of elements in the group. For that, a sum and a count Ire tabulated using 2 hash maps and then the average was calculated in the iterator. The iterator is then implemented by iterating through the HashMap, creating the appropriate tuples, adding them to an ArrayList, and returning the list's iterator. The StringAggregator was implemented in the same way but only supports count.

**Aggregate.java:** Aggregate uses the Aggregators written above by grouping and doing the aggregate calculations for all the tuples in the child iterator. Its iterator then uses the Aggregator's iterator to get the results of a Group By query.

**HeapPage.java:** insertTuple does a linear search of the page through the header using isSlotUsed to find a free space and then inserts a tuple into that slot, updating the header accordingly. deleteTuple gets the recordId from the tuple to be deleted, removes it from the tuples array, and updates the header.

**HeapFile.java:** insertTuple looks linearly through the file for a free slot; if it does not find one, creates a new page, inserts the tuple into that page (with HeapPage's insertTuple) and appends it to the file. deleteTuple determines the pageId from the tuple to be deleted and uses HeapPage's deleteTuple function to delete the tuple.

**Insert.java, Delete.java**: fetchNext is implemented by iterating through the child iterator and using the BufferPool insert/deleteTuple function (described later). A counter is used to keep track of how many items Ire inputted or deleted and then is returned as a one-element tuple.

**BufferPool.java:** insert and deleteTuple make use of HeapFile.java's insert and deleteTuple functions. They get a list of modified pages from those functions, then mark them as dirty and update them in the cache.

**Page Eviction Policy:** I iterate through the cache and evict the first page I can flush to disk. Since the spec did not require any special algorithm, this was the simplest thing to do and will just evict (probably) one of the first pages returned by the cache's iterator, which is a HashMap. Though simple to implement and easy to understand, it is not optimized for performance at all and relies on what order Java's HashMap iterator traverses through the BufferPool.

Changes to API: None Ire made.

Missing elements: All the unit/system tests passed.