Daniel Yao 004-182-464

Ming Lu 904-035-039

CS143 Lab 2 write-up

Overall this lab took my partner and me about a week of time to complete. We did not change any APIs, but we did add an additional class to aid the implementation of the lab. We decided to follow the suggested steps while implementing this lab. No part of the lab is particularly difficult or confusing. However debugging of the lab did take some time.

The rest of this write up will discuss in detail how we implemented the 12 classes (Predicate.java, JoinPredicate.java, Filter.java, Join.java, IntegerAggregator.java, StringAggregator.java, Aggregate.java, HeapPage.java, HeapFile.java, Insert.java, Delete.java, BufferPool.java).

IntegerAggregator.java

Implementing this class took quite some time when compared to the others. My general approach to solving this problem was to create 2 different cases, one when we have grouping, one when we do not. I used a HashMap relation when there is grouping and hashed the names of columns in a tuple to an ArrayList of fields (Used to hold all the fields that group together). For the case when there is no grouping I created a single ArrayList of fields. Furthermore, I created an ArrayList of tuples which is used to store all the tuples that match the desired aggregate criteria. For the case when there is no grouping I created 5 variables, _count, _sum, _avg, _min, and _max to keep track of the total results whenever we merge a new tuple. This allows for less overall calculations in the long run. In the case of grouping, I recalculated the desired aggregate result by looping through every matched element.

StringAggregator.java

After implementing IntegerAggregator.java implementing this class did not take as long because this class can be thought as a sub-version of IntegerAggregator.java. I used similar approaches in terms of data structures used for implementation. Instead I only had to implement count operator, as the other ones would not make sense for Strings.

Aggregate.java

This class was the most simple to implement among the trio of classes related to aggregation. This class is mainly composed of setter/accessor methods. The two methods that took a little more thinking is open() and the constructor where I had to consider whether or not to create a IntegerAggregator or a StringAggregator. I created a variable of the superclass of the two types and assigned an instance of one or the other type depending on the type of aggregation field.

BufferPool.java

Implementing this class took some time as well (mainly due to debugging…). The eviction policy we implemented in this class is LRU. We did this by using Java’s PriorityQueue data structure, because it is a heap structure that can insert/log/delete and keep track of a priority value for each of its members. In addition, we have 2 HashMaps one that maps pageIds hash codes to pages, and the other maps TransactionId to an ArrayList of PageIds. (We implemented the case for get page when passed a transaction ID) The harder methods in this class to implement were getPage(), flushPage(PageId pid), InsertTuple, and DeleteTuple. Implementing the classes on their own was not too complex, but integrating them to work with our LRU replacement policy through the use of a PriorityQueue took some time from debugging. We could not pass the Eviction test because we had an ArrayList of pages inside HeapFile.java which held onto pages after we deleted them from the BufferPool, but we fixed it immediately. Implementing writePage() inside HeapFile.java was straightforward, we created an instance of a random access variable with the "FILE" we're given, and wrote to it whenever, by first calculating an offset and then writing the size of a page starting from that offset.

Predicate.java, JoinPredicate.java

These classes are very simple. Just had to store the field nums, operator, and the operand(for predicate.java) as well. Operator.compare was used for the filter functions.

Filter.java

The filter was implemented by having the DbIterator continuously give out tuples and returning a tuple that satisfies the given predicate. Having to use super.close() gave us some problems here.

Join.java

The main functionality of join was implemented as follows. For every tuple from the outer DbIterator(rewind after every pass), check the given predicate with every tuple from the inner DbIterator until the predicate is evaluated to true.

HeapPage.java

Dirty's were set true by the buffer pool if a page is modified. Delete was done by iteration through the tuples on the page until the equivalent tuple of the given tuple is found. Insert worked similarly but different in the it searches for an empty slot on the page and then updates the recordID of the tuple insert to reflect its new location.

HeapFile.java

Insert was done by iterating through the pages of this file until a page.insert succeeds. Removal is easier as the recordId of the tuple will point to the right page to begin with. Write pages is very similar to read pages which is already implemented. We ran into a problem of pages not begin deleted after being evicted because we kept page reference here, so that had to be fixed.

Insert.java, Delete.java

These two weren't difficult. The overall structure of these two classes were very similar to Filter.java and Join.java, which were already implemented. Additionally, the main functionally was done as follows: for every tuple in the given DbIterator, call the insert/remove function of the buffer pool.