PA3 Write-up

Team members: Wei-Hsiang Lin and Alex Wong

1. Describe any design decisions you made with a little more details.

1.1. Filter

We instantiate Filter class by providing a Predicate and DbIterator class. The Predicate class contains information about the operation and value to use for comparison. The DbIterator class contains the data to apply the filter to. To iterate through the Filter class, we can use its fetchNext() method. It is simple. We use a while-loop to traverse all data in the DbIterator class and return the next data that passes/matches the Predicate class. The match is performed using the predicate's filter method.

1.2. Join

1.2.1. Join.java

The Join class takes in two DbIterator classes and a JoinPredicate. The goal is to join the two DbIterator classes. To implement fetchNext(), we used a double-while loop. Assuming m and n is the length of the two DbIterator class, the runtime complexity is O(mn). We can see that this is not very efficient, and therefore we provide a better solution in the HashEquiJoin class.

1.2.2. HashEquiJoin.java

To implement fetchNext(), we use a join algorithm called "HashJoin". There are two phases: build and probe. In the build phase, we start with a hash table and add the first DbIterator's data into the hash table. In the probe phase, we read the second DbIterator's data and find matching data from the hash table. In an ideal situation, we will not run into hash collisions and achieve a runtime complexity of O(m+n). However, the worse situation is we will run into hash collisions every single time and get a runtime complexity of O(mn).

1.3. Aggregate

1.3.1. IntegerAggregator.java

To implement mergeTupleIntoGroup(Tuple tup), the IntegerAggregator class maintains a hashtable to store all aggregate-related data; including min, max, sum, count and average. The first step is checking if the hash table has the key already. If it does not, we set all aggregate-related data based on the first data we have and insert it into the hash table. If it does have the key, we simply append the data based on the type of aggregate data. For computing average specifically, we don't aggregate the value into the average. Instead, we first aggregate sum and count, and then calculate average by dividing sum by count.

1.3.2. StringAggregator.java

Similar to IntegerAggregator.java, we maintain a hash table but only keep data about the count of a particular word/string.

1.4. Heap File Mutability

1.4.1. HeapPage

To implement insertTuple(Tuple tup), we first check if there are available slots in the current page. If there is none, we return a DbException. If there is space, we loop each spot to find an available one. Then, we insert the tuple to the available slot on this page, mark the slot as used and add recordId to the tuple (to make deletion easier later on).

To implement deleteTuple(Tuple tup), we iterate all used tuples on this page and find the tuple we want to delete by matching tuple's recordId (which we add to Tuple during insertTuple step). The actual step to delete the Tuple is simply marking the slot as free; so the next Tuple will just replace this slot.

1.4.2. HeapFile

To implement insertTuple(TransactionId tid, Tuple tup), we maintain an arraylist of dirty pages. Then, we try to iterate pages in our heap file to find one with an empty slot. Once this page is found, we call HeapPage class' insertTuple(Tuple tup) to insert the tuple to the page. However, if there are no available slots in all pages in the Heap File, then we create a new page and add the Tuple into it. Then, we write the page to disk. In the entire process, we are keeping track of dirty pages as well.

deleteTuple(TransactionId tid, Tuple t) is very similar to insertTuple(TransactionId tid, Tuple tup). Instead of calling Heap Page's insertTuple(..), we call Heap Page's deleteTuple(..). If we are unable to locate the page, then we throw an exception.

1.5. Insert & Delete

1.5.1. Insert

To implement fetchNext(), we maintain a boolean to keep track of whether this instance has already been executed. If it has been, we return null. If it hasn't been, we insert the DbIterator's data into the BufferPool. While doing this, we keep count of how many data we have inserted and return this count as a Tuple.

1.5.2. Delete

The Delete class works almost identical to the Insert class. The only difference is the Delete class deletes data from BufferPool, rather than inserting it.

2. Discuss and justify any changes you made to the API + Describe any missing or incomplete elements of your code.

3. Describe how long you spent on the assignment, and whether there was anything you found particularly difficult or confusing.

Spent 2.5 days each.

4. Collaboration

In this assignment, Wei-Hsiang and I (Alex) discussed each part's underlying implementation before coding and testing. In terms of code, Wei-Hsiang was in charge of 1.1 to 1.2 and I was in charge of 1.3 to 1.4 and writing the report.