Ming Hill
CS 660
Professor Athanassoulis
28 Nov 2023

PA3 Write Up

Worked with Abdelazim

For our Filter class, there weren't many design decisions that we had to make. For the smaller methods (open, close, rewind, etc) we just called the parent operator function before calling on the fields themselves. This is also true when we implemented those functions for the other classes in this assignment. Other than that, for the fetchNext() method, we decided that the simplest way to approach this problem was to just iterate through the child iterator, until we found a tuple that passed the filter, and then returned it.

For the join class, one decision that was pivotal in completing the fetchNext() method was to create a t1 field. The t1 field represented an optional tuple object, which would store the current tuple from child1. For the fetchNext() method, we used a nested loop, matching a given t1 to a t2. We added this line "t1 != std::nullopt || child1->hasNext()" for the conditional of the outer loop, which would make sure that we would always have a child1, and if not, we would just return a nullopt. Then by iterating through child2, we can find the element that would match with t1, and return the tuple. When child2 is at the end of the iteration, we know then to iterate child1 to the next value therefore, we would rewind child2 back to the beginning and iterate child1 one iteration. With this logic, we would be able to match and join each element from both groups. This is the same logic that we used for HashEquiJoin.

For our IntegerAggeregate, a couple of small design decisions were made to optimize the implementation of the methods. These included adding a hashMaps field to the IntegerAggerate class (count, groupSum), which we could refer to in mergeTupletoGroup. The count map represents the number of counts per grouping, if there is a grouping, and groupSum represents the total value of each grouping. For mergeTupleIntoGroup, we first identify if there is a grouping or not, and set that as our groupbyField variable. Then by using that field variable, we can increase the count in our count hashmap of the respective grouping if there is one. Then by using case matching on the aggregate operators, we implemented the correct operations for updating groupSum. For next() in IntegerAggregatorIterator, we get the key from the open() method, which points us to the current key in the count hashmap. This key is the same for both count and groupSum hashmaps, therefore we can retrieve the respective values for both hashmaps. Then once again, similar to mergeTupletoGroup, we utilize case matching on the aggregate operator to decide which operation to use and store the result in the resultValue variable. Finally, depending on whether there is a grouping or not, we create a tuple with one field, which is just the aggregate value, or a tuple with

two fields, one being the grouping and the second being the value. Finally, we increment our current iterator to point at the next key in the count hashmap before returning our newly created tuple.

For our Aggregate class, the fetchNext() method is related to the IntegerAggreagtor class by fetching the next tuple and returning it. Additionally, getTupleDesc is dependent on the grouping field. Depending on whether there is a grouping field or not, the TupleDesc will contain a group and aggregate column, or only an aggregate column.

For the Delete/Insert classes, our implementation for their respective fetchNext() is very similar. We just iterate through the child and keep track of the number of tuples that we insert/delete, before returning a new tuple that that value as the first field.

We did not make any significant changes to the API, the only thing that we changed was inserting an unordered map to the constructor of agregateIterator.

We did not miss or have any incomplete elements of code.

This assignment was less intense and time-consuming than PA2 for my partner and me. Many of the earlier methods we had to implement were pretty straightforward (Filter, Insert, Delete). We had 2 main issues when trying to complete the code. The first one was the fetchNext() method for the join operator. When we initially tried developing the method, we had a major logic mistake, instead of matching up each element from the first group (t1) to every matching element in the second group (t2), we would completely iterate over t1 after the first match. The second issue we had was with the IntegerAggregate file. Given the minimal instructions, it was difficult at first to understand what we were supposed to implement, and when we figured that out, the logic behind the implementation took a while to figure out. Specifically, we were confused about how the IntegerAggregatorIterator was tied to IntegerAggregate, as well as how to implement next() in the iterator class and fetchNext() in the IntegerAggregate class.

For this project, I worked with Abdel from our class. For 90% of the project, we worked together, especially on more of the difficult problems such as IntegerAggregator and fetchNext() for join. Other than that, we worked separately on the last two classes, Delete and Insert, however, after implementing both of them, we went over them together to get an understanding of the logic behind our implementations.