PA 3 Writeup - Abdelazim Lokma

For this programming assignment, I worked with Ming Hill in order to solve all the challenges presented to us. Work was done in person throughout the duration of this assignment, as a result, all the coding was done side by side with Ming, either on my laptop, or on his laptop. At the end of each session, the person with the most up to date codebase would push their changes to our shared GitHub Repository so that we can both access and review our work without needing the other persons device. Due to the fact that the Delete and Insert classes were nearly identical, we decided to discuss the general logic for both classes after working on them separately. However, all other classes, especially IntegerAggregator and Join were done together.

Filter Class:

We first Started with the filter class, not only because it was the first exercise of the programming assignment, but also because it is one of the most fundamental operations used in table modification. The filter class is responsible for filtering out any tuples (from the input table) that do not match the predicate (that is also inputted). It inherits its characteristics from the Operator class, which is used to implement functions that are shared across many of the table modification operators. As a result, implementing the functions for this class was fairly simple. For most of the functions (open, close, rewind), we simply had to call their corresponding parent function, and any of the corresponding children's inbuilt function. However, for the FetchNext function, we had to make sure that the iterator passed into the Filter instance (which represents an iterator for the table to be filtered) gets us the next valid tuple matching the predicate condition.

Join & HashEquiJoin Classes:

These two classes are fundamentally similar in their logic and implementation. Due to the requirements given to us as part of the Programming Assignment, we could implement the logic for both classes pretty much identically. Beginning with the getTupleDescriptor function, we needed to merge the tuple descriptors of both of the tables to be joined. This process was made easy due to the `TupleDesc::merge()` function, allowing us to return a TD that describes the resulting joined table. Fetching the Field Names of the columns that we were basing the join on was also as simple as calling the `TupleDesc::getFieldName()` function. The `HashEquiJoin::fetchNext` function iterates through tuples from the two input relations. It updates pointers to tuples from the left and right tables, when a match is found, it creates a new tuple by combining the fields of the matching tuples. We had to create a class variable t1, that would save the position of the outer relations iterator, in the case where we return a tuple, this was done to ensure that successive calls to the `HashEquiJoin::fetchNext` function returned the next tuple in the join table, rather than always resetting the iterator after every return.

Aggregate class:

The `Aggregate` class is responsible for performing aggregation operations on the tuples retrieved from its child iterator. The function of `fetchNext` method is to read tuples from the child iterator. Notably, the `getTupleDesc` function dynamically generates the resulting tuple descriptor based on the specified grouping and aggregation fields, if the aggregation does not use a grouping column, then we simply return a one field TupleDescriptor with the aggregation value, otherwise we return a two field TupleDescriptor for the group name and the aggregation values for each group.

IntegerAggregatorClass:

The `IntegerAggregator` class is responsible for aggregating integer values based on grouping and aggregation criteria. The `mergeTupleIntoGroup` function processes incoming tuples, updating count and aggregate sum information for each group, this information is stored in two unordered maps, one is responsible for storing the calculations (be it a sum for each group, or the min/max of a group), and the other is responsible for storing the sizes of each group. These data structures are a part of the `IntegerAggregatorIterator` class, which also includes an iterator variable to fetch information from both unordered maps. Should the aggregation have no groupings, each unordered map would store only one key value pair. `IntegerAggregatorIterator` class uses its unordered map iterator to handle opening, checking for next tuples, retrieving the next tuple, rewinding, obtaining the tuple descriptor, and closing the iterator. Focusing on our implementation of the `next()` function, we implemented the logic for how information is extracted from the unordered maps to create a tuple that contains the group's name, and the aggregation associated with that group. For example, if the aim was to get an average value for each group then we would simply need to divide the sum of that group (for the aggregation field) by its group size, both of which are retrieved from the unordered hash maps that were generated by `mergeTupleIntoGroup` . We also return adjust the tuple based on if we are using a group by column or not, returning a 2 field tuple in if we are, or a 1 field tuple containing the aggregation value if we are not.

Insert & Delete Classes:

The `Insert` & `Delete` classes facilitate the insertion and deletion of tuples into a specified table. The `fetchNext` function for both of them iterates through tuples from its child iterator with the same logic, inserting or deleting each tuple (respectively) into the database's buffer pool associated with the given transaction and table ID. After all tuples have been inserted/deleted, the `fetchNext` function returns a tuple containing a count of how many tuples were inserted/deleted. The rest of the functions in these two classes involve manipulating the functions defined in the DbIterator class, so that the tables can be scanned from this class.