CSE 444: Lab 2
Aakash Srazali | Cherry Winarto

1. Query runtimes:
    - Query 1: 0.73
    - Query 2: Didn't finish (we waited ~2 hours)
    - Query 3: Didn't finish (we waited ~3 hours)

2. In the initial stage of the lab, we primarily focused on incorporating select, join, and aggregate operations. Subsequently, in the second part, we successfully completed the implementation of table modifications involving tuple deletions and insertions. Completion of this lab enables the modification of tables in SimpleDB, and the ability to handle SQL queries that involve the basic operations.

    SimpleDB consists of two predicates namely, predicate: operator used to perform unary comparisons between fields of tuples during a Filter operation, and JoinPredicate: operator used to compare fields of two tuples using a predicate during a Join operation.

    The Filter Operator effectively functioned as a select operator within the relational algebra framework. Filter consists of a relational selection, which filters out tuples based on a specific predicate. It requires a Predicate and an OpIterator as inputs. The Predicate determines the criteria for a tuple to pass the filtering condition, while the OpIterator is used to iteratively fetch tuples until one satisfies the conditions set by the Predicate. Once a tuple satisfies the conditions, it is subsequently returned. By utilizing this operator, we were able to merge two tuples from separate relations based on a join condition, thereby accomplishing join operations.

    Join consists of a relational join process that combines tuples based on a specified predicate. It takes in a Predicate and a pair of OpIterators as inputs, each corresponding to a relation for joining. The Predicate determines which tuple pair meets the join criteria. Both pairs of OpIterators iteratively obtain pairs of tuples until one satisfies the conditions outlined by the Predicate and then the tuple pair is returned. The implementation of Join employs a simple nested loop join technique for matching tuples, which is likely the reason why Query 2 and Query 3 show very long performance times (see design decision discussion below).

    Aggregates are operators handling integer and string aggregations, including column-based grouping and operations like min, max, sum, avg, and count. The Aggregate operator acts as an iterator calculating aggregates over a single column with or without groupings, returning new tuple(s) representing aggregate values. Based on the column type, either an IntegerAggregator or a StringAggregator is used. Aggregators hold the logic for computing aggregates over int/string fields, with the StringAggregator limited to 'count' operations.

    In the subsequent phase of the lab, we established all the essential components required for inserting and deleting tuples from a table. This was made feasible in SimpleDB through the implementation of the Insert and Delete operators. These operators directly invoke the BufferPool's methods for tuple insertion and deletion, which in turn utilize the HeapFile's corresponding operations. This hierarchical structure enables us to exercise precise control over the actions taking place in memory. For instance, we can determine when to write data to disk by marking a tuple as "dirty." Additionally, we possess the ability to choose eviction strategies for pages when memory resources are depleted.

3. For **Predicate.java, JoinPredicate.java and Filter.java**, there were no design choices to be made. We implemented a usual object class for Java. For **Join.java**, we added a method to

merge tuples together in Tuple.java. We referenced lecture pseudocode and used a nested loop join. The problem that comes up is that it is a slow join so Query 2 and 3 takes a very long time to finish. We might have to change this to a hash join in the future.

We chose to use a HashMap for **StringAggregator** and **IntegerAggregator**. StringAggregator was implemented using one HashMap, which maps the group-by fields to count of tuples processed. If our group-by field has no grouping, we simply keep track of the number of tuples processed using an int variable to save on space.

IntegerAggregator, on the other hand, was implemented using two HashMaps – one maps the group-by field to the aggregated/accumulated value, while the other maps the group-by fields to count of tuples processed. We need the extra HashMap because, unlike StringAggregator which only handles COUNT, IntegerAggregator also handles AVG, MIN, MAX, and SUM. Therefore, we also need to track the aggregated value of these group-by fields. Similar to StringAggregator, if the group-by field has no grouping, then their group-by field will be represented with a null key in the HashMaps. For **Aggregate**, we used the StringAggregator and IntegerAggregator class as an abstraction. As such, it mainly served as a wrapper class for those other classes.

For **BufferPool**, we chose a randomized page eviction policy because it was easier and cheaper to implement than other policies like LRU policy. While LRU does provide good performance, since pages that have been used in the past few instructions are likely to be used again in the next few instructions, it is expensive and will take us a longer time to implement in practice. On the other hand, random replacement eliminates the overhead cost of tracking pages and performs better than other policies, such as FIFO.

4. A unit test that could be added would be a check if we are flushing the pages correctly. This is especially important because we want to make sure that dirty pages are written to disk before we evict them. InsertTest and DeleteTest both deal with dirty pages, but do not check if they are written properly. We actually ran into a bug while implementing this and weren't aware that our dirty pages weren't getting written to disk. We also notice that the EvictionTest also does not check whether pages are written to disk, even though flushing pages is part of the eviction policy because we were passing EvictionTest despite failing BufferPoolWriteTest which we eventually solved.

5. We did not make any changes to the API.

6. We didn't have any missing or incomplete pieces to our code.

7. We do not have any feedback on this assignment, we thoroughly enjoyed it.