Lab 2
Ivan Zachary - ivanb13
Phuoc Huynh - phuynh08

Query 1: 0.45 seconds
Query 2: 8 rows and unable to finish after running 40 minutes
Queyr 3: unable to finish after 1 hour

Lab 2 Write Up:

Exercise 1:

This part of the lab provides methods for SimpleDb to filter the tuple of a that met the condition and join the tuple between tables. We use Predicate to get the tuples that have field value equals to the condition. JoinPredicate is used to know whether 2 tuples can join based on the join predicate. In Filter, we iterator through a table with OpIterator and apply the Predicate to each tuple, only return the tuple that satisfies the predicate in Filter constructor. Similar to Filter, Join will iterator through each possible pair of tuples that satisfies the JoinPredicate and return one tuple a time in next() function.

Exercise 2:

This part of lab 2 handles the aggregate operator that aggregates over tuples received from the child operator for query plans that perform groupBy's and/or compute aggregates. In our simple DB implementation, we have 2 possible data types that we can perform an aggregate on, a string or integer field. With this operator, our db implementation is able to perform: GroupBy, Sum, Count, Avg, Min, and Max. But, before handling the aggregate operator, we first had to implement a class for string and integer aggregates, which both implement the aggregator interface, since the operations for aggregating between the two data types are different. In both cases for string and integer aggregates, the way our db aggregates fields is by continually merging new tuples from the child operator into the proper existing aggregate.

For aggregating over integer fields and string fields, the constructor for both aggregators, will receive the field to be aggregated on and the type of that field, as well as what operation to compute on the field, such as max or count. Additionally, the constructor will be told whether or not there be grouping on this field along with aggregating. For the integer aggregator, it allows for the following operations: GroupBy, Sum, Max, Min, Avg and Count. On the other hand, our string aggregator can only perform the groupby operation and count. For the integer and string aggregator class, after the aggregate operation, an iterator is given that allows the user to iterate over the group aggregate results.

Now, for the aggregate operator itself, it uses the integer and string aggregator class as helper classes to perform the aggregate operations. Within the constructor of the operator, like the helper classes, it is given the field and field type of which to perform the aggregate over. Using the type of the field passed in, the constructor of the operator knows which helper class to use. Additionally, the operator is also told from the

constructor whether or not there will be grouping involved. Once constructed, the aggregate operator will read in tuples from its child operator and pass these tuples into the helper classes to perform the groupby, if present in the query plan, and necessary supported aggregate operations for that field's type.

Exercise 3: In this part of the lab, in order to add/delete a row into table, we use BufferPool to insert tuple into HeapFile. First, we need to getPage() into the Buffer Pool and use insert/delete method in HeapFile to insert/delete a tuple. We iterate though each Page in HeapFile to get a empty slot in a Page in order to insert into file, if there no empy Page, we need to create an empty Page. Similar for delete a tuple, we use BufferPool to get each Page() inside the HeapFile until a Page that contains the tuple. HeaPage provides the methods to insert/delete a tuple inside a Page, we only need to iterate all the positions inside a Page, and check whether a position is empty by isSlotUsed(), flip its value by markSlotUsed(). Whenever we modify a Page in BufferPool, we need to mark it as dirty Page, we need to write a dirty Page back to disk before flush it.

Exercise 4:

In lab 2, we also implemented the operators for insert and delete queries. These operators work such that, for any delete/insert query plan, these operators will be the top most operator of the plan and receive tuples from the child operators that will be either inserted or deleted from their respective table, reflecting these changes within the pages on disk containing the tuples for this table and using the BufferPool's tuple insert and delete methods to do so. Additionally, at the end of the insertion and deletion process of these tuples, the operator will return a single field tuple that reports back the amount of affected tuples.

For both the insert and delete operators, the operator will receive the tuples to be inserted or deleted from their child operator, the transaction Id for the query that is performing these operations, and the table for which these tuples should be inserted into or deleted from. Once the operator is constructed and opened, it will begin to iterate through the tuples passed to it by the child operator and continually call, based on the operation, either the buffer pool tuple insert or delete methods on all the tuples being iterated over. When calling the tuple insert and delete buffer pool methods, the bufferpool class will retrieve the heap file associated with the table Id of the table containing these to be inserted/deleted tuples and insert or delete these specified tuples from their respective pages in the heap file accordingly.

At the end of the insertion or deletion process, for a successful insert/delete, the changes will be reflected and written back to disk, marking changed pages as "dirty", and a single field tuple of the count of affected tuples will be returned

Exercise 5: Our Buffer Pool has limited capacity, therefore we need a eviction policy to evict a Page whenever it gets full. We implemented Least Recently Used algorithm, we used deque data structure, the most recently used will be put at the end of the deque by addLast() method, we evict the first item in the deque that is not dirty Page, we reserve our dirty page until the transaction is completed and flush it to disk. Once we flushed

dirty pages to disk, it becomes not dirty and we can evict it. Whenever a Page is used in BufferPool, we will move it to the end of deque as the most recently used. For flushPage, we use writePage of HeapFile to write data in that Page to actual file on disk.

Potential Unit Test To Be Added:

An example of a possible unit test that could be added is a test that checks to see after the operations for the insert and delete operators are finished that tuples cannot be continually added or deleted afterwards using the operators fetchNext() method. In our implementation of the operators, after the operator finished its inserts/deletes, the set of tuples given by the child operator are marked as read, indicating that the tuples have already been properly dealt with and inserted/deleted from the respective table and pages in the bufferpool. The only way to continue using the operator to insert or delete tuples is by calling rewind(). With the current unit tests for the insert and delete operators, it does not check to see that the insert/delete operations cannot be repeated once already executed without calling rewind().