CSE 444 LAB 2

1. Runtimes :
    1. Query 1: 0.71 seconds
    2. Query 2: Achieved 10 tuples in a half hour. Didn't complete in a half hour.
    3. Query 3: Query didn't complete in 30 minutes.

2. This lab consisted of several different components that were built on top of the earlier low level files that we had written in Lab 1. The first thing we implemented in this lab were the operators Filter and Join. Filter and Join are operators that take in a predicate that defines which tuples satisfy the condition that the filter/join is looking for. The Filter operator looks at all the possible values in the child passed to it and only returns those that satisfy the predicate. In a similar fashion, the Join operator looks through both the child elements using a Nested For Join Loop and returns the tuple pair that satisfy the constraint. The operators return the tuples one at a time (or a pair for the Join operator) when fetchNext() is called on the operators. Each of the operators is basically an iterator that returns the computed correct values one at a time when asked.

The Aggregate operators aim at accomplishing the Aggregate components of a SQL Query(Sum, Max, Min, etc Including Group By). The Aggregate consists of two parts - IntegerAggregator and StringAggregator. The IntegerAggregator performs an aggregation operation on a Tuple provided based on the field the aggregation should be done on and the predicate to be applied. The group by is also taken into context to make sure if there is a grouping, the aggregate is computed accordingly. String Aggregator basically has the same functionality as the IntegerAggregator but only works on Count aggregation over Strings. The only difference is on wether the aggregation field is an integer or String. The Aggregate Class can handle which Aggregator to call when and return the results accordingly as it is also an iterator implementation.

The other important component we implemented in this lab is the functionality to insert and delete tuples. This is done as explained. When the user wants to delete a Tuple, they call BufferPool.deleteTuple(). BufferPool.deleteTuple() calls deleteTuple() on the file that contains that tuple, The heapfile deleteTuple method calls the deleteTuple method of the HeapPage. And the heapPage changes the header such that the slot that the tuple was stored in is now available. In a similar fashion, when the user wants to Insert a tuple, the BufferPool.insertTuple() is called which calls the insertTuple method of the respective heapFile which in turn calls the insertTuple method of the heapPage which writes the data onto the page, sets the slot as marked and changes the recordId of the Tuple. If a page isn't available in the file, a new page is written into the file so that it's available to insert tuples. This is how the insert and delete works. Each page that is affected by the insert/delete is added to the BufferPool so it can be written to disk before being evicted and be available to other transactions reading the page.

The BufferPool has a maximum number of pages it can store. When the maximum number of pages has been reached, the Bufferpool flushes a page from it's storage and makes place for the incoming page to be stored. Flushing of a page basically means checking if the page has been altered in any way during the duration that the page was in the BufferPool and if it is "dirty", it is first written in disk by putting the entire pages contents into the place in the file(memory) where the previous page was. Hence, through these different operations, we can now execute most of the SQL Queries (including aggregates and joins!).

3. My page eviction policy in this lab was to remove a random page from the BufferPool. I believed this would make the implementation and computing cost easier than some of the other methods like LRU while not compromising a lot on the operation timing cost. I decided to use the Nested Loop Join for this lab. Some of the other design decision policies I made is to compute the next tuple for a join only when one asked the next instead of precomputing and storing all the possible tuples and returning them one by one when asked.

4. A Unit Test that could be added would be to see if when we insert a tuple into a page, we are actually changing the slot value for where we insert the page. Instead of just checking if the numberOfFreeSpaces decreases, the test could check if the page is indeed put into the first available slot and the slot header value is changed accordingly as soon as something is inserted into the page.

5. No API Changes

6. Feedback : “More Detailed commenting on the Parser would make debugging Query Fails easier since figuring out what went wrong when the Query fails is very tough using just print statements and break points. Other than that, I found the lab very fun!”