- **Exercise 7 Runtime:**

  The run time of three query is 1.42 seconds, 4.73 seconds and 10.38 seconds.

- **What the Lab is About:**

  The first half of the lab helps us learn about the implementation of join, filter, group, and aggregation. I also practiced a lot about the algorithms (for example hash equal join and nested loop join for other join predicate) to optimize these operations. Filter and join are easy to work with. For filter we only need to check if the tuple satisfy the predicate. For join we could just combine two tuples using predicate. The only thing worth noticing for join that I would tell is that we need to be aware that some tuples might have same keys. We have to form the combination between two groups of tuples that has same keys. As for aggregate, we have two types of aggregator, one is interger aggregator which allow all kinds of aggregate including min, max, sum, average, count, the other is the string aggregator which only accepts count predicate. The aggregate operator calls the two types of aggregator based on target field's type. We also have to consider grouping for the aggregate operator if the group field is specified. If group is specified we will need to return aggregation of each group according to the predicate.

  The second half of the lab helps me learn more about write operations to SimpleDB, specifically the operation of inserting and deleting tuples. To insert and delete tuples, we have to consider update our original deployment of the Heappage, Heapfile, and BufferPool. The buffer pool will call heapfile's insert and delete method, mark the modified page dirty and the heapfile will call HeapPage's insert and write method to modify specific page and write to the disk. We also implemented the insert and delete operator which calls the buffer pool for inserting and deleting tuples.

- **Design Decisions**

  For join operator, I used my own version of the hashed join for equal predicate. Hash join is the fastest one pass join we learned in class thus I would like to take advantage. Firstly, I hashed the tuples from first table by their join key. For the group of tuples that have a same key, I store them into an array list and save the array list as value of their key into a hash table. Then for every tuple in the second table, I will get the group of table-one tuples in the hash table by their join key and create combination for each tuple in the group with tuple in the second table. For the other predicate, I have been using the traditional nested loops because hash join doesn't work so well for the ranged data (hash table access could be unclustered). The trade off for using hashed

join for equal operator is that I have used more memory than nested loop join to store the hash table.

For aggregation, I also used the similar technique to hash the tuples by their group field and the values are the aggregate field. In this way, we could make fetching the previously aggregated group super fast and thus speed up the merge process for new tuple. For average predicate, I used two additional hash maps, one is to store the decimal sum of previous tuples (groupAggregate1) from same group, the other is to store the count of the tuples in the group (groupAggregate2). The normal groupAggregate hash map is used to store the final new result that calculated by sum plus new tuple value then divide by count plus 1. Instead of storing average, this way we could make the result more precise since we are rounding it to integer.

For my choice of page eviction policy, I used the concept of LRU(Least Recent Used) cache as an example. I would always evict the least used page and save the most recently used if new page needs to be added but buffer pool are full. The reason for adopting this policy is that we are dealing with pages of table data that stores continuously and would be reading the pages continuously in most of the times. Thus we would be super likely to revisit same page again that we visited most recently to fetch the neighbor tuples, thus I think LRU policy would be a great fit. I implemented this by changing the buffer pool's storaging to a linked hash map which provides method for removing least used pages.

- **Unit Test to Add**

  We don't have simple JUnit Test for delete tuples in HeapPages and Heapfile delete but only have system test for large files. It makes behavior validation a little difficult in the beginning and thus I would have some small example such as delete one tuple from three pages. Also, Join tests only test for equal and greater than operator but not for the rest of operator like less than, less than or equal, etc. We could add some more.

- **Other Declarations**

  I didn't change any of API. I didn't miss any implementation that the lab requires. I don' t have any feedback. I didn't collaborate with anyone.