Query Runtimes (Exercise 7):

Query 1: 0.90 seconds

Query 2: Did not finish after 60 minutes

Query 3: Did not finish after 60 minutes

Description of Lab 2:

Lab 2 flushes out a couple of different aspects of SimpleDB. One of these aspects is additional operators. The first operation is `Filter` which allows us to take a stream of tuples and reduce them to only those that satisfy a specific condition (`Predicate`). It achieves this by iterating over the child's tuples and only keeping the ones that match the `Predicate`. The `Predicate` itself just checks whether the condition it stores holds for the tuple that it is given.

The second operation is `Join` which enables combining of two streams of tuples matching them on a certain condition (`JoinPredicate`). To achieve this we use a nested loop join that iterates over the outer stream of tuples while checking them against all of the tuples of the inner stream with the `JoinPredicate`. The `JoinPredicate` itself compares a pair of tuples to see if they should be joined given the join attribute.

The third operation is `Insert` which allows us to insert a stream of tuples into the database. Prior to its inclusion the testing harness was responsible for generating all of the data. Insert walks through its child's tuples and inserts all of them into the database though the `BufferPool` since the `HeapPages` that it writes to should now be in the cache.

The fourth operator is delete and it mirrors insert but allows us to remove a stream of tuples from our database instead.

The fifth operator is `Aggregate` so that we can summarize statistics about a stream of tuples. This `COUNT`, `SUM`, `AVG`, `MIN`, `MAX`, and will be extended in future labs to support additional operations. Since SimpleDB supports two types of `Fields` (i.e. `IntField` and `StringField`) there are two underlying `aggregators` that are aptly named (`IntAggregator` and `StringAggregator`) for each case. Overall the aggregation merges all of the tuples into the stream either all together or by group. This occurs in the opening of the operator so that all calls to `next`() return complete summaries of either the entire stream of tuples or the given group.

Outside of operations Lab 2 improves the functionality of the `BufferPool` by allowing for flushing and eviction of pages. This is made possible by the addition of insert and deletion support throughout the database. With write support we are able to flush pages (write them to disk) and evict pages (flush then delete). Without these additions the pool would eventually fill up and prevent all other pages from being interacted with.

The changes to insert and delete also carry through to the `HeapPage` and `HeapFile` that we built on during this lab. The `HeapFile` was expanded to handle page writing which is essential to

(and utilized by) the `BufferPool` to flush pages. Similarly, it also supports tuple specific operations that access the underlying `HeapPages` which are updated accordingly. The `HeapPages` hold the tuples and have been extended to delete and insert them individually by changing the `header` and `tuples` representation. Additionally, they now track whether they have become `dirty`. This is useful in the `BufferPool` to know which pages need to be flushed and which ones are up to date.

Design decisions:

- `Filter` walks through each `Tuple` of the `child` until it finds one that fits the criteria then it returns it. The position is stored by the `child` operator so that it does not duplicate return values.
- `Join` utilizes a nested loop join that merges matching tuples from both relations while rewinding the inner relation so that it can be compared to each outer tuple.
- `IntegerAggregator` utilizes three maps with `Field→Integer` mapping: `aggregate`, `groupCounts`, and `groupSums`. This is so that we can support all of the current operations, keep a running average, and support the future sum-count and sum-count-average operations.
- `StringAggregator` utilizes one map with `Field→Integer` mapping since it only supports the count operation.
- `Aggregate` performs the aggregation when opened so that the iterations can return the correctly aggregated values.
- `HeapFile` utilizes a read/write `RandomAccessFile` to seek the proper position for writing pages. Furthermore, it utilizes a `FileOutputStream` to append a new `HeapPage` to the end of its underlying file when all are full during a tuple insertion.
- `HeapPage` stores additional fields to track whether it is dirty and the transaction that made it so (if applicable). It also streams through the tuples to find the first empty location when inserting a new tuple. This is more overhead but prevents losing track of freed locations after deletion.
- `BufferPool` functions as a wrapper around `HeapFile` when inserting and deleting tuples. It uses the `Database` singleton to access the specific `HeapFile` it is altering.
- The `Delete` and `Insert` operators walk through all of their `child`'s tuples returning a single summarizing `Tuple` of the number changed.
- Our page eviction policy was to remove the first `Page` in the `BufferPool`. This method has a straightforward implementation and is similar to random `Page` eviction since `HashMaps` are not in insertion order.

Example of a unit test that could be added to improve the set:

One unit test that could be added to improve the testing suite would be to verify that the `HeapFile#deleteTuple()` works properly. For example, a unit test that asserts that pages that we return from the call to `deleteTuple()` are the proper pages to be returned.

Changes you made to the API: None.

Describe any missing or incomplete elements of your code: None.

Additional Feedback: None.