Andrew Dawson
1220796

At a high level this lab had two parts: implementing a couple of operators and expanding BufferPool to include page eviction. The operators that we implemented were: Filter, Join, Aggregate, Insert and Delete. All these operators implement the abstract Operator class. The Operator class implements DbIterator. All Operators are simply DbIterators - meaning they iterate over tuples. Operators are distinct from each other based on how they generate the tuples returned. All operators have one or two child DbIterators - which they use to generate the tuples they return.

The Filter operator implements a selection. Filter has a single child and a Predicate it uses to filter. It returns all tuples from its child that satisfy the given predicate.

The Join operator implements a join using a JoinPreciate and two children (child1 and child2). On Join.open(), Join finds all pairs of tuples t1 from child1 and t2 from child2 such that t1 and t2 join. HashJoin is used for equality joins otherwise Nested Loop Join is used. In terms of implementation the goal of both HashJoin and Nested Loop Join is to find all joining tuples and add them to a list of joined tuples. HashJoin generates this list by building an in-memory hashtable of the tuples in child1, then iterates over the tuples in child2 and queries the hashtable for matches. I choice to use a Map<Field, List<Tuple>> for the hashtable. The key is the join field and the value is all the tuples that have that join value. The Nested Loop Join generates the joined tuple list by iterating over all tuples in child1, t1 and for each t1 iterating over all tuples t2 in child2, checking if t1 and t2 match and if so adding the match to the joined list. Once we have found all matching tuples and put them into a joined list (either with Nested Loop Join or HashJoin) we can use this list's iterator as the underlying data structure for Join to generate tuples.

The Aggregate operator implements aggregation using a single child iterator and an Aggregator. An Aggregator is responsible for keeping track of an aggregate value computed based on tuples that have been merged into it. Aggregate will construct either an IntegerAggregator or a StringAggregator, depending on which type of data it is aggregating over. Then for every tuple t in the child, merge t into the constructed aggregator. Aggregators also provide an iterator method that returns a DbIterator over the aggregate results. This iterator is used by the Aggregate operator to get the tuples that will be returned. The high level implementation of both IntegerAggregator and StringAggregator is a hashtable holds mappings between a group and an aggregate value. The representation is complicated by two factors. First, Aggregators are not required to have a grouping which is problematic if using a map where keys can't null. Second, most aggregate values are a single value such as MAX but AVERAGE requires two values - a count and a sum, therefore we cannot simply store an int for all aggregate types. In order to address these issues I used a pair of ints class. Aggregate values that only need one value to be represented simply use the first int in the pair, while average can use the second int in the pair. In order to address the case of no groupings I added a field that keeps track of a single aggregate value without groups. In the case of groups I use a map from group to aggregate value.

The next two operators we had to implement were Insert and Delete. In order to implement these we had to do work in several places: HeapPage, HeapFile, BufferPool, Insert and Delete. In HeapPage inserting and deleting simply requires updating the header to reflect the state of the deleted or inserted tuple. In HeapFile we had to implement deleteTuple and insertTuple. These methods take a tuple insert or delete it and return a list of all pages that were dirtied by the modification. A dirtied page is a page that is different in memory than it is on disk. Inserts and Deletes must go through BufferPool so that pages that are touched can be added to the cache for faster future access. In BufferPool we had to implement insertTuple and deleteTuple. Both of these methods simply access the correct DbFile, use that file to perform the operation, get back a list of pages affected and add those pages to the cache. The Insert and Delete operator classes are very simple. They use the BufferPool to insert or delete all the child tuples, count how many tuples were inserted or deleted and this becomes the result they return. It is key that these Operators use the BufferPool otherwise the pages will not get cached.

After implementing all the above operators we had to implement a page eviction policy. The BufferPool has a limited number of pages it can hold. So when we have touched more pages than the BufferPool can hold we need to evict some pages to make space. I selected a Least Recently Used, LRU, eviction policy. This policy makes since because pages that have not been accessed for a while are unlikely to be accessed again soon. In order to implement this I changed my BufferPool to use a LinkedHashMap. A LinkedHashMap iterates over buckets in the same order they were added. So the first bucket added is the first bucket to be returned by the LinkedHashMap iterator. This makes implementing LRU very simple. Every time we need to add a page to BufferPool (for insert, delete or read page) we check to see if our BufferPool is at full capacity. If it is at fully capacity than we evict the first page returned by the LinkedHashMap.iterator (which is the least recently used page). The advantage of using a LinkedHashMap is it handles ordering for us and it still provides constant access.

One of the biggest holes in the unit testing was delete was not tested fully. For example, HeapPage.delete was not tested. So I would have added a unit test that checks that the header of HeapPage gets updated corrected when a tuple is deleted. I did not make any changes to the API. I do not think that there are any missing or incomplete parts of my code.

I spent about 20 hours on this lab. Most of my time was spent debugging. I finished lab early, got all the tests passing, but when I ran the queries they did not run. Even queries like select * from data; didn't run. It took me 8 hours to figure out that the problem was not with my code but with something about running on windows. Once I ran on attu it worked fine. That was a painful debugging experience but I learned a lot through it. On the whole I did not find this lab that difficult or confusing. I enjoyed working on it and selecting a page eviction policy.

Query Runtimes

query 1: 5 rows, .84 seconds
query 2: 11 rows, 2.3 seconds
query 3: 7 rows, 3.08 seconds