Lab2 answers

In this lab, I've implemented several operators and added the page eviction method to `BufferPool`. The operators include `Filter`, `Join`, `Aggregate`, `Insert` and `Delete`. All of these operators extend from `Operator`, the abstract class which implements `DbIterator`. So all of these operators can iterate over tuples and return the results as what the `Predicate` assigned.

Filter

The `Filter` operator implements the `SELECT` query. It has one child operator and one `Predicate`. It can return all tuples that match its `Predicate` from its child operator.

Join

The `Join` operator has two child operators. It can return all tuples from its children that match the `JoinPredicate`. The joined tuple list are obtained once the `Join` operator is opened. In `Join.open()`, it will traverse all the `(t1, t2)` pairs with `t1` from `child1` and `t2` from `child2`; if the `JoinPredicate` is satisfied, it will create a new joined tuple using `mergeTuple()` and add it to the list. To cut down the time complexity, the Hash Join is used for `EQUALS` predicate. That is, first to iterate over `child1` and use a `HashMap` to store all the `(Field, List<Tuple>)` pairs. Specifically, the key is the `Field` and the value is a list of tuples that hold this `Field`. Then we iterate over `child2`, and for each tuple `t`, if its field is recorded in the map, we do the join for `t` and all the tuples stored in that key. For other predicate operations, the nested loop join is used. The outer loop iterators over `child1`, and for each tuple `t1`, it has a inside loop to iterate over `child2` to get each tuple `t2`, and do the join for each `(t1, t2)` pair if they match the `JoinPredicate`.

Aggregate

The `Aggregate` implements the `GROUP BY` query, but only for a single field, so it only has one child operator. Also, we use `Aggregator` to keep the aggregation results. Since we only have two fields: `int` and `String`, we have `IntegerAggregator` and `StringAggregator` to track results of the corresponding field that is aggregated over . For `IntegerAggregator`, there are different types of aggregation, some only need to keep track one aggregate value, such as `MAX` ,`MIN` and `COUNT`, but some need two, such as `SUM_COUNT` and `AVG`, they have to keep track of both sum and count values. Hence, two hashmaps are used to keep track the results, one is `groups`: its key is the `Field` and its value is the results depending on the aggregation type; the other is `counts`: its key is `Field` and its values is the counts of this field. So it will iterates over all the tuples in the assigned group field and update both `counts` and `gruops`. If the group is not specified, it will choose the first field as the group field. After the iteration, the aggregation results can be calculated based on these two hashmaps and the aggregation type. For `StringAggregator`, it only has `COUNT` aggregation, so it is pretty much the same as the one in `IntegerAggregator`.

Insert and Delete

These two operators involve the changes in `HeapPage`, `HeapFile` and `BufferPool`. They has only one child operator and a `tableId` to tell which file these inserted/deleted tuples located. For these two operators, they iterate over the tuples in their child, and call `BufferPool` to do the insertion/deletion. After that, they return the number of inserted/deleted tuples.

In `BufferPool`, it calls the heap file methods to do the insert/delete; in `HeapFile`, it actually call the `HeapPage` do to the insert/delete, and return the list of changed pages to `BufferPool`, such that the `BufferPool` can update the cached pages by update the hashmap called `pageMap` and mark the changed pages as dirty.

The insertion/deletion operations are actually implemented in `HeapPage`. For deletion, `HeapPage` will find the slot of this tuple and mark the slot as unused. For insertion, `HeapPage` will find the first unused slot and insert the tuple and then mark this slot as used. And for `HeapFile`, it will find the first page that has empty slot for insertion; if not, it will create a new page at the end of current pages and call this new page to insert the tuple. After that, this new page will be written to disk by `HeapFile`.

Page eviction

For page eviction, I choose the Least Recently Used (LRU) policy. To be convenient, I use `LinkedHashMap` to cache all the pages in the `BufferPool`. When a page is planned to be cached but the pool is full, the `pageMap.keyset().iterator()` can get the `PageId` by the order that they be added into the pool. If the page to be evicted is dirty, we should flush it to disk and overwrite its original version. And then we discard this page by remove it from the `pageMap`. After eviction, `BufferPool` has place to cache the planned page.

For `flushPage`, `BufferPool` call the heap file method of `writePage` to write it back to disk, and then mark this page as not dirty. For `writePage` method, I use the `RandomAccessFile` class, similar to what I did for `readPage`. From the page number and page size, `RandomAccessFile` can seek the start address for writing this page, and then it can write this page (`page.getPageData()`) to disk.

Query Runtimes

> - query 1: 5 rows, 0.29 seconds;
> - query 2: 11 rows, 0.66 seconds;
> - query 3: 7 rows, 1.18 seconds.

mark: These runtimes change for different executions.