lab2 write up

Query running

1

```
5 rows.
Transaction 0 committed.
6.30 seconds
```

2, 3

I didn’t get the chance to finish query 2 & 3. I ran query 2 on my own computer for about 40 mins and query 3 for 3 hours but still couldn’t the finish the result. I’m not sure if there’s some bugs in my join implementation (since I’m able to finish query 1) . I’ve written some simple unit test for my join function but it works fine. Maybe it’s because I’m using the most simple nested loop join? But it shouldn’t take that long time to finish query according to #242 in Ed. I’ll try to see if I can implement Hash join before Lab3.

Lab description

This lab we implement the basic operators: **Filter, Join, Aggregate and Insert and Delete.** The most important job of each operator is that they're responsible for return the fetchNext() tuple, which is the result of all the downward children chain. Since they’re chained with each other, their class format are similar. But in order to perform different functionality in a query with similar format we also implement several “helper” class to embedded into those operators. Such as: **Predicate, JoinPredicate, IntegerAggregator, StringAggregator.** Also, in order to perform Insert and Delete, we need to implement disk modification method. I personally think the modification logic in between HeapFile, HeapPage and BufferPool is the most difficult part in this lab. I’ll talk about my understanding later. Lastly we're asked to implement the eviction policy for Buffer Pool. I use the “Most-recent-used-stay" rule for my buffer pool.

There’s not too much design in Filters and Predicate. One is that, I think you guys can mention the compare order for filter() method.

For Join and JoinPredicate. I’m using the nested-loop join. Iterate through each inner child and every time the inner child iterator hit the last slot, move the outer child iterator to next(). Using JoinPredicate filter out the valid join result.

For IntegerAggregator, I divided into “Grouping” and “No-grouping” based on the initialize filed gbField. If it’s “No-grouping” the Aggregator only need to initialize two single value field: result, count. one is to count the number of values and one is for record the result. Every time we merge new tuples, we only need update those two field. If it’s “Grouping”, we initialize two hashMap: resultMap and countMap. countMap responsible for map each gbField with its frequency. resultMap is responsible for map each gbField with the aggregate result(for COUNT it’s the same map). So when we implement Iterator(), we simply just retrieve the result of map.iterator(). One thing is that we need to use (double) to record the result of AVG, otherwise it won’t get the correct result.

For tuple modification part. It’s kind of hard for me at first to understand the logic between each calls between HeapPage, HeapFile and BufferPool. My understanding is that: Every time we want to modify a tuple, fetchNext() will call on BufferPool (**I think this is a important point to know before we actually implement Heapfile and HeaPage. Maybe it would be better to put the implementation of Delete and Insert operator before the heapInsert and heapDelete part?**) Then the BufferPool will find the DbFile corresponding to this tuple modification and call on HeapFile.delete() or HeapFile.Insert(). Then DbFile will try to find the corresponding HeapPage to finish the modification job. The important thing here is that we'll access the HeapPage via Buffer Pool. It's kind of unintuitive at first since we're already using the BufferPool to call on DbFile at first place and now we call back on it? I think the reason we do that is **we don't want to directly make the modification on disk when we do insertion and deletion. We only want to keep track the modification pages(dirty) and then write them once they got evicted. So we want the BufferPool to always cache with the latest used page.** When we access the page via BufferPool it automatically cached in it. If we directly use HeapFile and HeapPage modify the tuple on disk, the bufferPool won't be able to know and it's clearly lead to bad performance. After understanding this, the implementation is fairly simple. One disk modification we have to done is that if there’s no room for a new tuple, we'll have to create a new page and append it on disk(even though we don't have to update the new tuple on it) just to let DbFile and BufferPool know there’a new page appended.

One different thing to notice for Delete and Insert operator is that. It only return the result tuple once in its (open-close or rewind) lifecycle. Also, no matter the child.hasNext() or not it’ll return this onetime result `If !child.hasNext() return {0}` so I use a boolean field hasReturned to indicate whether this operator has returned result in this lifecycle or not.

The eviction policy design I use is similar with solution in Leetcode 146. I simply use a **Queue<PageId.hashCode()>** data structure to record the “most-recent-used" pages. Every time a page been accessed, we put the PageId to the end of the queue to make it “most-recent". Every time the page number out of limit we're trying to evict a page, we retrieve the page at the top of the queue. Then we flush it(If it's dirty) and discard it from the Queue and BufferPool Map.

Unit Test

I think we could add unit tests on time of join algorithm. So we can know how efficient our join algorithm.

we could add noGrouping Max, Min, Count, Sum test

Also, I think the HeapFile and HeapPage unit test in lab1 can not detect some edge cases bugs. I find my implementation on HeapPage.iterator() and HeapFile.iterator() with bugs until this lab. It would be better if we add the case: **fullPage → emptyPage → fullPage → emptyPage → emptyPage** to lab 1 unit test.