Lab2 Write Up:

Yuanfeng Li

Feb.3rd.2021

1. **Report Runtimes of the queries from Exercise 7**

I had a lot of fun making the hash join function in this practice and I do have a lot to talk about, but making it short, I will give you a quick view of comparison.

Loops/Runtimes	Query 1	Query 2	Query 3
Nested Loop Join	0.3 seconds	0.69 seconds	> 500 seconds
Hash Join	0.19 seconds	0.64 seconds	1.18 seconds

(I putted the screen shots in the appendix's runtime section.)

- There are 4 things to notice:
  1. The hash join only works on equal operations. (Predicate.Op.EQUALS)
  2. The hash Join is more than 500 times faster than the nested loop on equal operations!! Woohoo!!
  3. I could not take the screenshots for nested loop join of query 3 since I accidentally turn off my laptop, so you will not find it in appendix section, but I do remember the runtime is around 530 seconds.
  4. All the runtimes are from the results in my laptop's Linux environment, not on attu3, since I am not a CSE major's student, so I am not 100% sure whether the speed will increase or decrease on attu3.

2. **Describe what this lab was about?**

In Summary I had implemented two major operation function including filter and join, and also the function to modify table such as insert Tuples, and delete Tuples.

I will give you a short intro on each cluster of classes that I developed.

1) Filter and Join

   Filter only returns the tuples that meet the requirement of the predicate by using the compare method in , and join function joins the compare method in field interface, and join method using join both tuples from child 1 and child 2, with the requirement and condition of the Join Predicate.

2) Aggregates
   Three java classes basically did the same thing, they aggregate the tuples, or even group by the tuples and fields with MIN, MAX, SUM, AVG, and COUNT operands.

3) HeapFile Mutability and Insertion and deletion
   HeapFile calls buffer pool to manipulate the tables with insertion and deletion, and heappage do the lowest level job. If we insert and delete a tuple or some tuple, the heapPage and HeapFile would marked as dirty and stored in the bufferpool, then write to disk later for updates.
4) Page eviction
   When we reach the limit of pages that can store in the buffer pool, we need to call flushpage, or flushAllpages in buffer pool to get more memory space in buffer pool. And page eviction is the tool to achieve that goal.

3. **Design decisions**

I will talk about my design in the Join.java first and then the page eviction policy:

For my Join.java:

My design was enlightened by the 3 queries in the optional contest. I was thinking using the nested loop join for all of them at first, but find it was really difficult to finish the 3rd query in short time. To be honest, I was eager to only pass the 3rd queries in short time, so the easiest method I can think of is to use a hash join method only for Equal operations. So you can see the hashjoin method in my Join.java open method, that I used it only when operation is EQUALS. Other than EQUALS, I will still use nested loops since hash function can not solve the greater joins or less joins, etc easily. Due to the short time of hand, I have not developed a merge sort join method which might be able to solve all kinds operations, including greater than or less than, etc.;

My Hash join method was focusing on putting the tuples from child1 into a hashmap, and use their hashcodes of the field value as the keys, and an array of tuple as values. After we put all the child1's tuples in the hashmap, we can iterate through the tuples from child2 and find out the arrays by using the hashcode of child2's field in the map, we can easily find out the arrays of child1 tuples that have the same hashcode with child2's, and then we can join child2's tuple with each child1's tuple in the arraylist and add the combined tuple in a global arraylist of tuple for future uses. I will just introduce the high level of implementation here, but you can find more specific details in my Join.java.

For Eviction Policy:

My design for the evictPage method is quite simple. I will always throw the first page in the keyset() of the cache, since I am using a HashMap for cache, so It does not really matter which page I flush. As long As I can get more memory in cache, it will be good enough.

4. **One example of unit test**

I found it was really difficult to understand how my joins work with exceptions using the current unit tests, so the one example of unit test I would suggest to add would be:

A unit test to test errors that might happen such as TransactionAbortedException of DBExceptions while we are joining tables.

5. **Discuss and justify any changes you made to the API.**
   No changes.

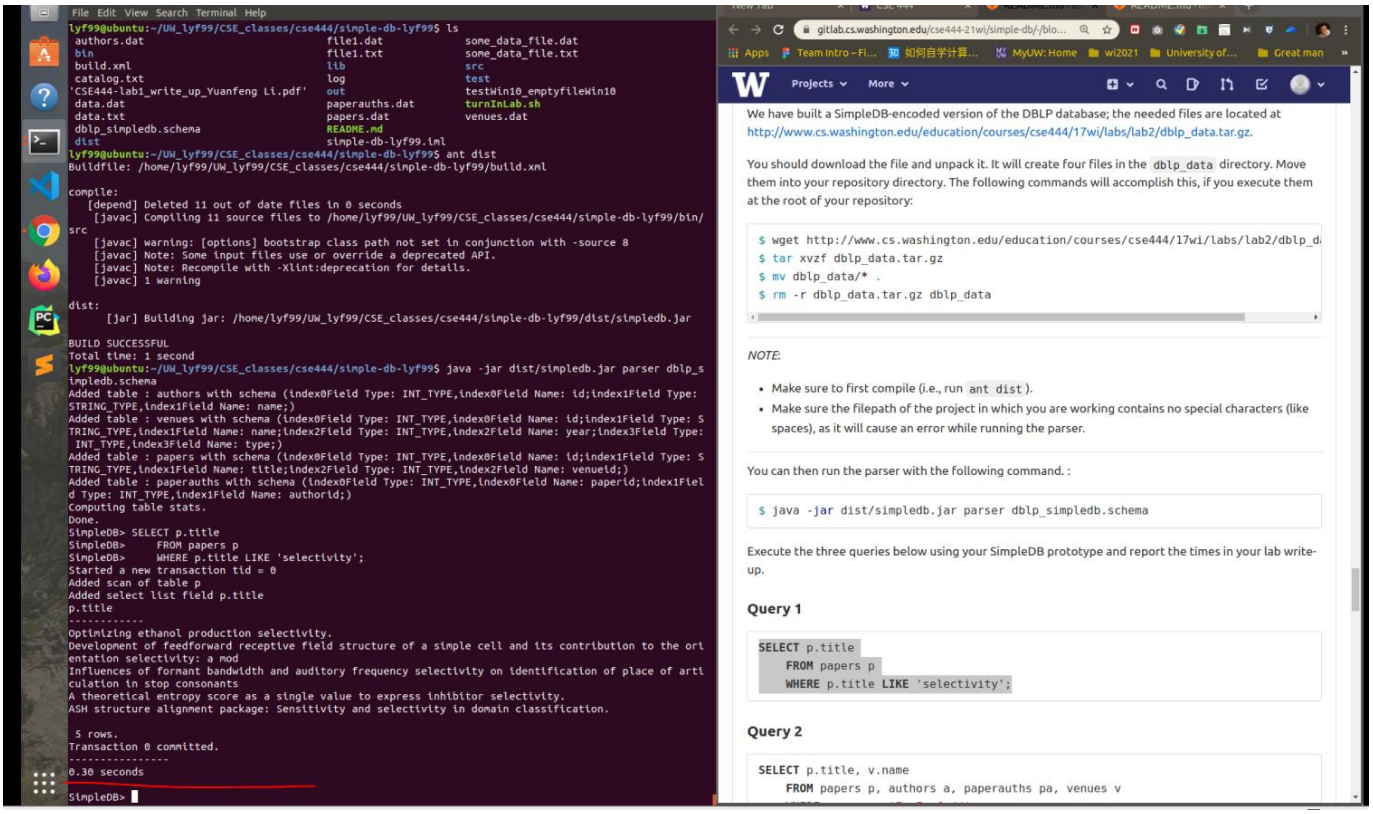
6. **Describe any missing or incomplete elements of your code.**

As I mentioned in design decisions, I would like to develop an sort and merge join method for join.java, but due to the short time in hand, I couldn't really finish it before due date.

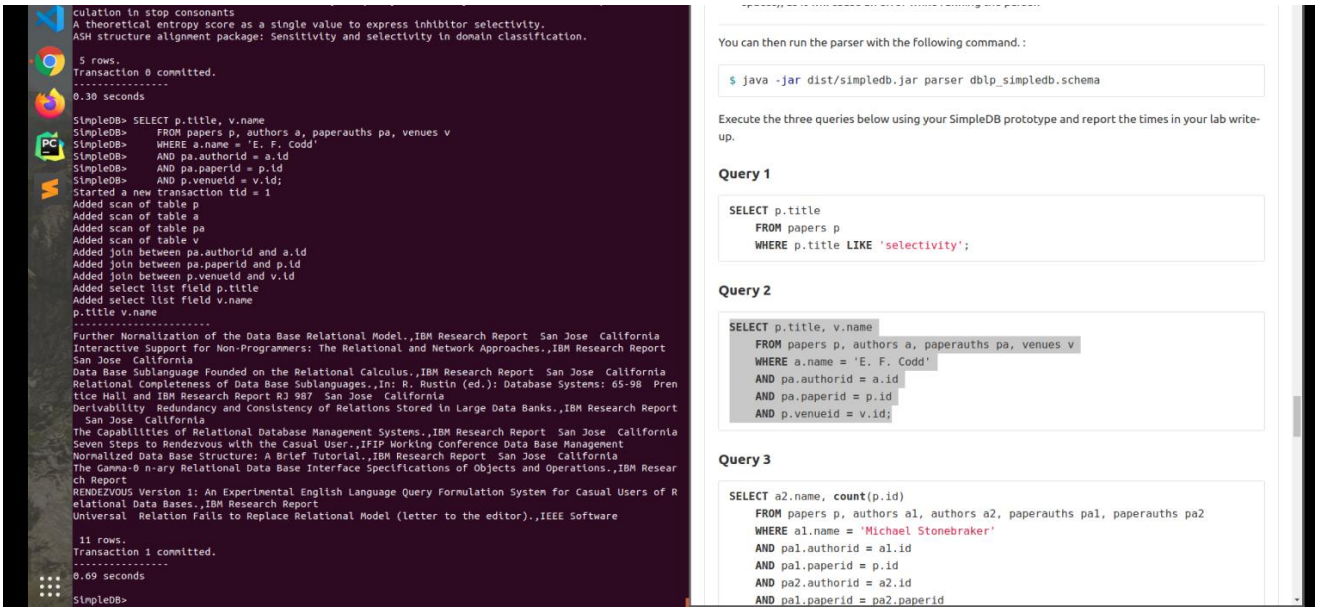
Appendix:

Nested Loop Join:

- Nested loop query 1



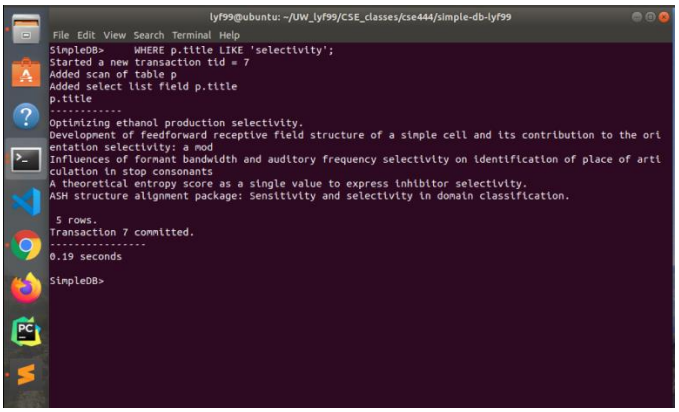
- Nested loop query 2



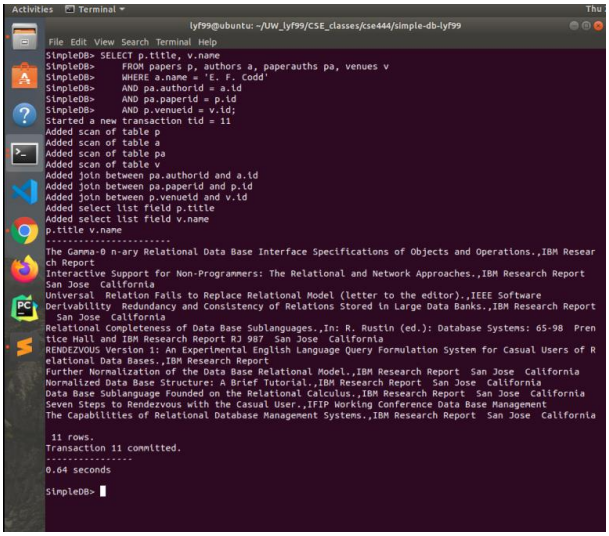
- Nested loop query3: More than 500 seconds!(as mentioned earlier, I forgot to take the screenshots, but I remember it was around 500 seconds)

Hash Join:

Query 1: using hash Join for Equal, and nested loop Join for other than equal. 0.19s



Query 2: 0.64 s



Query 3: 1.18 seconds