EURECOM

Sophia Antipolis

Database Management Systems

Lab 2 Report

Group 17

AGARWALLA, Yash

SENANE, Zineb

HAKAM, Mouad

25 January 2022

We used the Lab 1 solution provided by the professor to work on, as we were not satisfied with our codes in the first lab.

Prediacte.java : is a simple file, easy to implement. It contains a constructor with some getter and setter methods. It contains also another method that compares the field of a given tuple to the one specified in the constructor using the operand attribute and Field's compare method. This file took us 30 minutes.

JoinPredicate.java: Very easy too, contains constructor and some getter methods. It applies the predicate and makes also uses Field's compare method to compare two tuples through their fields. This file took us 30 minutes.

In filter.java, We started with initializing the predicate and child in a constructor. Then we completed two getters "getPredicate()" and "getTupleDesc()". "Open", "close" and "rewind "were easy functions to implement. For "fetchNext", We ran a loop till the child operator has no more tuples, checked if the tuples were filtered using the predicate value, and returned the tuple if it was not filtered. However, we were getting errors in FilterTest. After checking, we realized that we were not considering the situation that if there was no more tuple. We added a return null statement in the end to fix it. It made our file run without any errors. This file took us 1 hour.

Join.java was almost like filter.java, however, there was an important function called "fetchNext". Here, we firstly looped around the first child tuple. We moved to the next child tuple if it was null. Then we iterated the second child which followed the same condition of null. If there was a match in the tuples, we create a combined tuple with the values of both child tuples. We ran two loops for entering the value for each tuple.
We were able to pass the PredicateTest, JoinPredicate, FilterTest, JoinTest, system tests: FilterTest and JoinTest; all passed successfully without any errors. This file took us 30 minutes.

IntegerAggregate.java : A very complex file, it took a lot of time to implement(around 7h), we had to implement the constructor, mergeTupleIntoGroup where we had to make the difference between the GROUP_BY case and the NO_GROUPING case. And the most challenging and time-consuming part was the programming of the five SQL aggregates (SUM, AVG, COUNT, MIN, MAX) for String form and Integer form data. At the end of the file, we had to program a method that create an OpIterator over group aggregate results.

StringAggregate.java : An equally complex file as the previous one, but this time it was easier to do everything since COUNT is the only SQL aggregate that String accepts and it has the same structure of IntegerAggregate.java

Aggregate.java: this is like an interface class that is implemented once in the IntegerAggregator and once in the StringAggregator files. By grouping and performing the aggregated column it creates an instance of one of the previous classes depending on the type of the aggregated value, and by specifying if there is any grouping (null or the field type of the grouping field). This file took us 2h.
IntegerAggregatorTest, StringAggregatortest, AggregateTest, system test:AggregateTest all passed successfully.

HeapPage.java: fortunately in this file we don't have to deal with the constructor of the class, but we had to implement IsSlotUsed, Isdirty, markSlotUsed, getNumEmptySlot .. it was quite easy but the methods deleteTuple, InsertTuple were quite tricky: for deleteTuple we had to get the Tuple number, get the heappageId where it's stored, then check if the Tuple belong to this heappage and not empty and set it to null. in the case of insertTuple we had to check if the slots are used before adding the tuple, overall this file took around 5h to do in total because HeapPageTest.java unit test refuse to pass as a consequence of futile errors in the code.

HeapFile.java: we first implemented only the writePage method as mentioned in the lab description. The tricky part here is to calculate the offset. First of all, we get the Page number using getPageNumber() method through its Id and we check if it's valid. Then we do random access to the dbfile in order to write at the arbitrary offset, which is calculated by multiplying the page number and page size in the bufferpool. With this random access file we seek and write the data contained on this page and we close it. After running the tests and getting errors we realized that we need to implement insertTuple and deleteTuple methods as well. For the InsertTuple method, we get all the pages and we store them in a heappage. Then we check if this heappage contains a free slot if yes we insert the tuple into it and we add the concerned page to our list and we return the list of the pages to where the tuple was inserted. If there is no empty slot for all the pages we create a new one using the output streaming and we load an empty page to it then we close it. Finally, we load it into memory by creating a new heappage and we insert the tuple to it. We add it to our list and we return it.

For the deleteTuple method, we look for the pages that contain the specified tuple t through the getPage() bufferpool's method. Once we get this heappage we delete the tuple using the deleteTuple heappage's method and we add it to our list. Finally, we return the list of modified pages. HeapFileWriteTest.java unit test passed successfully after fixing the problems due to insertTuple and deleteTuple methods. Overall this file took us around 4 hours to complete and fix the errors.

The functions "insertTuple" and "deleteTuple" of the bufferpool file were extremely difficult to implement, mainly due to the bugs present in previous files (heapPage and heapFile). We started with writing a huge code where we get the database file for the respective table id then we iterated over each page from the page list, marking them dirty if they were dirtied with that transaction and added them to cache. For Delete, we get the database file according to the tuple's table id, and we do the same, i.e., mark dirty if it was dirtied and put it into the cache. However, we got an error (delete) in the tests "BufferPoolWriteTest". We did many changes, but nothing was fixing the error. So, we tried to change previous files. To fix it, we created the table from the beginning and initialized it with type int. After this, we were successfully able to pass BufferPoolWriteTest without any errors. This file took us also around 5 hours.

Insert.java: Simple file, with constructor, getter, and setter methods, and necessary methods for databases/memory management(close open rewind). It contains a fetchNext method that inserts the tuples read from child into a specified table by its id. It's an iterating process through the child that uses insert bufferpool's method and increment the counter. Finally, it returns a 1 field tuple containing the number of the inserted tuples. This file took us 1h 30minutes.

Delete.java: It's similar to the insert file but during the iterating process, it deletes the tuples instead of inserting them compared to the insert file. This file took us 30 minutes.

The unit test of insert passes successfully however it's not the case for the simpletest Insert. We get the error thrown by the exception we generated if TupleDesc of child differs from the table into which we want to insert. During the debugging phase we looked at the variables after the call of the line in the test file generating this error and realized that the sequential scan has a null TupleDesc, we didn't figure out how to fix this problem, but once we delete this verification the test passes successfully (normally the test should pass by this verification).
The delete simple test passes successfully.

Finally, we had to implement the flushPage method in bufferPool.java, it wasn't challenging we have to check first if the page exist in the hashmap (in the bufferpool) and then check if it was marked as dirty or not. And finally, write the page into the file (in the disk). And also evictPage() method where we had to discard a page from the buffer pool and flush the page to disk to ensure dirty pages are updated on disk. We made the choice to create a synchronized block where the synchronization parameter is the hashmap of pages to atomize its execution, if we arrived at the maximum number of pages in the buffer we will choose randomly a pageId and flush the page corresponding to this pageId and then remove this page from the hashmap otherwise we directly remove it. These methods took around 2h to be implemented.

We didn't make any changes to the API.