Readme for Project 2 CSC 540

Unity Ids:

Neetish Pathak (npathak2)
Bijo Joseph (bjoseph4)
Pratyush Gupta (pgupta9)
Mohit Satarkar (mmsatar2)

List of Files Changed

Task 1
BasicBufferMgr.java
BufferMgr.java
Buffer.java

Task 2
LogFormatter.java (New)
LogMgr.java (Modified)
Buffer.java (Modified)
Page.java (Modified)
SimpleDB.java (Modified)

Instructions for testing

Task 1:
File name: BufferTest0.java in simpledb.test
First make change in SimpleDB.java, BUFFER_SIZE = 4
Run - BufferTest0.java as a java application

Initial Set Up:
Make 4 blocks and pin blocks 0,1,2
4 blocks are used since the test case is designed for explicitly pinning three blocks and one additional block is for the logManager's buffer
Tests:

1. Start Test pin
2. Start Test unpin
3. Start Test number of available Buffers
4. Start Test buffer reallocation

5. Start Test buffer Abort Exception

Test flow:

1. Pin 3 blocks to buffers and check the pin count.
2. Pin a block again and check its pin count, it should have increased by 1
3. Unpin the a block until its pin count reduces to 0 and check pin count
4. List all the buffers available to check if any buffers are available
5. Try to pin an new block, and check if replacement works
6. Try to pin a new block and check if a BufferAbort Exception is thrown
7. Try to unpin buffer until pin count is zero, and then pin, no exception should be thrown

The output of BufferTest0.java should show which tests have passed.

Task 2: Log Management

1. Use **BufferTestLog.java** for testing the Log Management module (the file can be found in simpledb/test folder)
2. The Module provides four test cases (test1, test2, test3, test4)
3. They are invoked from the main function
4. **Please uncomment the function that needs to be tested and run the main file**
5. **Contents of the log buffer will be printed on console. Terminate the application and look at the simpledb1.log for the flushed values**
6. Below is summary of the test-cases conducted and their result

The log file called simpledb1.log is generated when running the program and can be found in your_home_directory/simpleDB

In order to see the result of each test case individually, delete the simpledb1.log file and do the test. Otherwise, new logs will be appended to the existing log file for the subsequent test cases.

Notes

The log buffer content after the test run is also shown. Since garbage values also get written from other modules when writing to the log files (eg. tx), we have shown some snapshots about where the written value is in the buffer

Also, we have used
Object[] rec = new Object[]{SETINT, txnum, blk.fileName(),blk.number(), offset, val};;
SimpleDB.logMgr().append(rec);)
because
eventually calling append from the Buffer SetInt function maintains the consistency in which other modules write to the logs. It is necessary for recovery later. As recovery manager expects

values with certain FLAGS (eg. SETINT, SETSTRING, START etc) to be present otherwise, it was throwing weird exceptions when we just tried to dump int value directly.

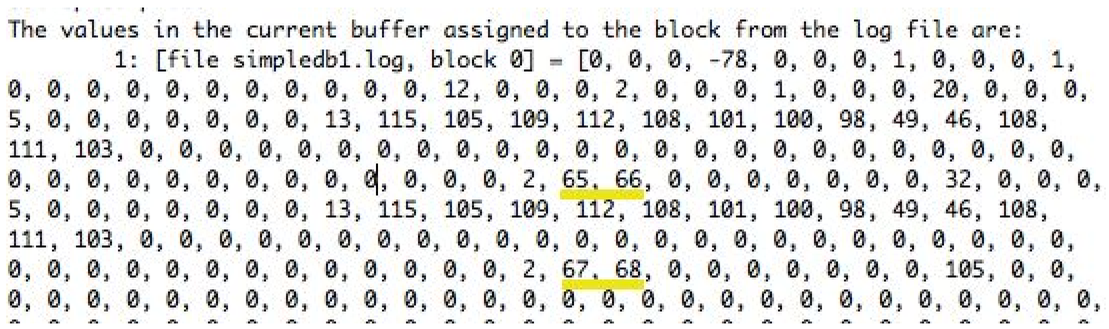
We can make some changes in recovery manager to contain that. However, we assumed that to be out of scope of the project. So, we focussed on LogMgr module.

The application needs to be stopped manually before running a new test

Example tests Summary

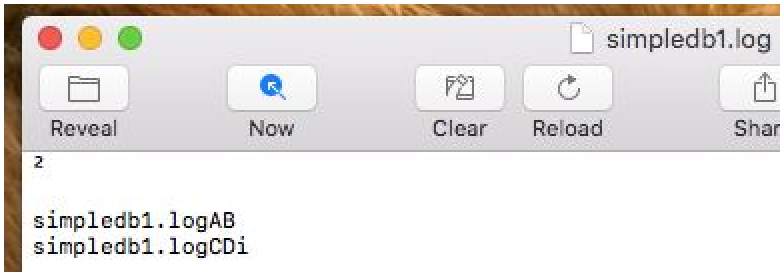
Eg: test1
```
        /*Test 1*/
        /*Write AB and CD in the logbuffer using the setString Function
         *Since logbMgr takes care of the offsets and txnum and lsn, we just make those values 0
here and pass the val appropriately
         * */
        lbuf.setString(0, "AB",0,0);
        lbuf.setString(0,"CD",0,0);
        printLogBuffer(bmgr);
```

Output:
The values 65,66 and 67,68 for AB and CD respectively get written in the buffer
As shown



```
The values in the current buffer assigned to the block from the log file are:
        1: [file simpledb1.log, block 0] = [0, 0, 0, -78, 0, 0, 0, 1, 0, 0, 0, 1,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 12, 0, 0, 0, 2, 0, 0, 0, 1, 0, 0, 0, 20, 0, 0, 0,
5, 0, 0, 0, 0, 0, 0, 0, 13, 115, 105, 109, 112, 108, 101, 100, 98, 49, 46, 108,
111, 103, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 65, 66, 0, 0, 0, 0, 0, 0, 0, 32, 0, 0, 0,
5, 0, 0, 0, 0, 0, 0, 0, 13, 115, 105, 109, 112, 108, 101, 100, 98, 49, 46, 108,
111, 103, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 67, 68, 0, 0, 0, 0, 0, 0, 0, 105, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

Also, the values get written in the log file simpledb1.log



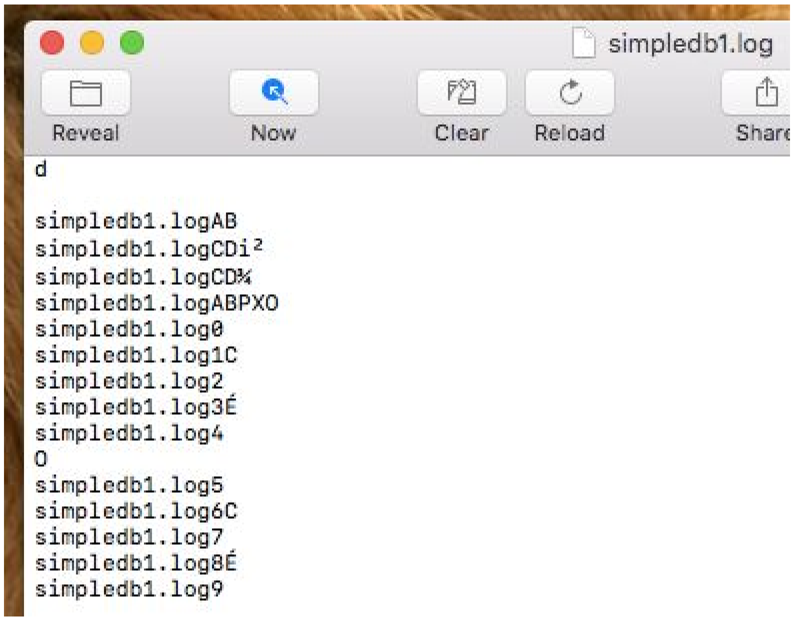
```
z

simpledb1.logAB
simpledb1.logCDi
```

As can be seen, AB and CD gets written in simpledb1.log

test2

```
 int cnt = 1;
        while(cnt-- > 0) {
                for(int x=48; x <= 57; ++x) {
                        lbuf.setInt(0, x, 0, 0);
                }
        }
```

Writing 0-9 in log buffer and it is flushed to log file thereafter.
Here is a snapshot from the logFile.



```
d

simpledb1.logAB
simpledb1.logCDi²
simpledb1.logCD¾
simpledb1.logABPXO
simpledb1.log0
simpledb1.log1C
simpledb1.log2
simpledb1.log3É
simpledb1.log4
O
simpledb1.log5
simpledb1.log6C
simpledb1.log7
simpledb1.log8É
simpledb1.log9
```

As can be seen, 0-9 is written in the extreme right.

We did two other tests as well for writing large chunk of data so that buffer unpin and new log block allocation works properly. The test cases are given in the test file.

We see content getting written to buffer and eventually log files in those cases.