myPTA Phase 2 Documentation

- Transaction Manager and Scheduler

team 5

For the Scheduler:

a. A detailed description of the design and implementation of the concurrency control strategies under different execution modes, including the compatibility table, the locking mechanisms for each operation, and so on.

b. A detailed description of the design and implementation of the deadlock detection and handling mechanisms.

c. A detailed description of the design and implementation of the recovery mechanisms under transaction abortion.

For the Transaction Manager:

a. A detailed description of the design and implementation of the operation reading mechanisms.

   Two methods of concurrent reading have been implemented: a Round Robin (which reads one line from each file at a time in turns) and a Random (which reads from files in random order, and reads a random number of lines from each file). When the user runs the LSM code, he will be asked to enter what kind of reading mode he wants, if he chooses the round-robin mode (enter “roundRobin”), the transaction manager will read all the transactions using round-robin method, if he chooses random mode(enter “random”), then he will be asked to enter the random seed he wants to use to generate the random number and the maximum lines he wants to read in each random run, and then the system will read all the transactions randomly.

   The transaction manager will first iterate all the files under the script directory(including the files in the subdirectory) and generate a fileList contains the name of these files. For each file, the transaction manager will use a java bufferedReader to read the first line of it and check whether the type of this script is transaction or process. Then store the filename and its java bufferedReader as a key-value pair in a hashmap named transactionBuffer, store the filename and its script type as a key-value pair in a hashmap named transactionMode. After that, the transaction manager will pass the fileList, transactionBuffer, and transactionMode to the roundRobin() function or the random() function according to the reading mode enter by the user at the beginning.

i. roundRobin(fileList, transactionBuffer, transactionMode)
    1. create two empty lists, transactionList and processList, the first one is used to store the content of transactions, the second one is to store the content of processes.
    2. while the transactionMode is not empty, iterate the fileList, for each file in the list, use the bufferReader it stored in the transactionBuffer map to read one new line from this file. If the new line is not null, then store the new line in the list created in step 1, that is, if the reading mode of this file stored in transactionMode map is “transaction”, then store the new line with its filename in the transactionList, otherwise, store the new line with its filename in the processList, if the new line is null, then the reading of this file is finished, remove its key-value pairs from the transactionBuffer and transactionMode, remove its name from the fileList, and continue the while loop.
    3. pass the transactionList and the processList to the scheduler.

ii. random(fileList, transactionBuffer, transactionMode, randomSeed, maxLines)
    1. create two empty lists, transactionList and processList, the first one is used to store the content of transactions, the second one is to store the content of processes.
    2. create a random number generator using the given seed.
    3. while the transactionMode is not empty, generate a random integer that is larger or equal to 0 and smaller than the size of transactionMode, use this random integer as an index to select a file from the fileList. For each selected file, generate a random integer named readRow that is larger or equal to 0 and smaller than the given parameter maxLines, this integer indicates how many lines should be read in this file in this run.
        a. While the readRow is lager than 0, use the bufferReader this file stored in the transactionBuffer map to read one new line from this file. If the new line is not null, reduced readRow by 1, and store this new line in the list created in step 1, that is, if the reading mode of this file stored in transactionMode map is “transaction”, then store the new line with its filename in the transactionList, otherwise, store the new line with its filename in the processList, if the new line is null, then the reading of this file is finished, remove its key-value pairs from the transactionBuffer and transactionMode, remove its name from the fileList, and break this inner while loop.
    4. pass the transactionList and the processList to the scheduler.

For the Testing:

a. For each normal test case, state clearly the transaction setup, the expected output with reasons, and the actual output. If the actual output is different from the expected output, analyze the reason.
   i. Concurrency Controll:
      For this part, I can use the same test files just like in deadlock detection, as the files in deadlock detection includes the problem about write after read, read after write and write after write. By using the test cases in deadlock detection I can know the correctness of the system. But my teammates do not have the enough time to finish all the part, so I cannot give out the test result.

   ii. Deadlock Detection:
      Normal Tests: For deadlock, there are 10 combinations for the operations, and operation R is not conflict to M operation, so there are 9 conflict pairs. In our code, all the detections with M operation have bug, so here I will test the 6 of the conflict paris, there are: R-D, R-E, R-W, W-D, W-E, E-D.

      For the R-D part, I have 2 script files: 1 and 2, the contents of these two files are the same as below:
      B0
      R A 1
      D A
      C
      The expected result is that there is a deadlock, as when file 1 wants to drop table A, it cannot get the lock because file 2 wants to read the table, and file 2 cannot grant the lock with the same reason. And the test result is in the github repository, we can see the result from the log.txt file, there are the history of the execution and the deadlock detection information, and the statistic data.The result shows that deadlock detected in history, because there is one edge from 1 to 2, and another edge from 2 to 1.

      For the R-E part, I have 2 script files: 1 and 2, the contents of these two files are the same as below:
      B0
      R A 1
      E A 1
      C
      The expected result is that there is a deadlock, as when file 1 wants to erase value 1 from table A, it cannot get the lock because file 2 wants to read the table, and file 2 cannot grant the lock with the same reason. And the test result shows that there is a deadlock, as we find the circle in the graph.

      For the R-W part, I have 2 script files: 1 and 2, the contents of these two files are the same as below:

```
B0
R A 1
W A (1, Thalia, 412-111-1324)
C
```

The expected result is that there is a deadlock, as when file 1 wants to write table A, it cannot get the lock because file 2 wants to read the table, and file 2 cannot grant the lock with the same reason. And the test result shows that there is a deadlock as we expect.

For the W-D part, I have 2 script files: 1 and 2, the contents of these two files are shown below:

```
B0                              B0
W B (1, Hey, 423-000-0000)      W A (1, Hey, 423-000-0000)
D A                             D B
C                               C
```

The expected result is that there is a deadlock, as when file 1 wants to delete table A, it cannot get the lock because file 2 wants to write the table, and file 2 cannot grant the lock because file 1 wants to write the table B. And the result shows that there is a deadlock as we expect.

For the W-E part, I have 2 script files: 1 and 2, the contents of these two files are shown below:

```
B0                              B0
W B (1, Hey, 423-000-0000)      W A (1, Hey, 423-000-0000)
E A 1                           E B 1
C                               C
```

The expected result is that there is a deadlock, as when file 1 wants to erase table A, it cannot get the lock because file 2 wants to write the table, and file 2 cannot grant the lock because file 1 wants to write the table B. And the result shows that there is a deadlock as we expect.

For the E-D part, I have 2 script files: 1 and 2, the contents of these two files are shown below:

```
B0                              B0
W B (1, Hey, 423-000-0000)      W A (1, Hey, 423-000-0000)
E B 1                           E A 1
D A                             D B
C                               C
```

The expected result is that there are 2 deadlocks, as when file 1 wants to erase table A, it cannot get the lock because file 2 wants to write the table, and file 2 cannot grant the lock because file 1 wants to write the table B. And another deadlock is that when file 1 wants to delete table A, it cannot get the lock as file 2 is erasing the value 1 in table A, and file 2 cannot get the lock for deleting table B for the same reason. The result shows that there are deadlocks, and we can see that the detector finds two edges from 1 to 2 and from 2 to 1. this is the same as we expect.

iii. Recovery:
For this part, I can use the same test files just like in deadlock detection, as the files in deadlock detection will cause deadlock problems, and then after the deadlock happens, I can use those cases to test whether the function works correctly. But my team does not finish this part, so I cannot give out the result of the test.

b. For each benchmark test case, state clearly the transaction setup, the execution result(e.g., if the program crashed), and the efficiency measures in throughput.

i. Concurrency Controll:
I can also use the benchmark generator to generate the benchmark we need here, as those benchmark includes all problems of write after read, read after write and write after write.

ii. Deadlock Detection:
For the benchmark mode, you can go to the benchmark.java to run it, and type in conflict pair you want to test: RD, RE, RW, WD, WE, ED, and then you need to type in the loop size of the generator (the first loop size decides how many big deadlock loops do you want, and the second loop size is how long each deadlock loop is). And then you will get those script files in the scripts folder. Then you can go test the deadlocks.

For the R-D part, I set all the loop sizes to 10, so there are 100 files. And the loop size of the deadlock is 10, so file 2 rely on file1, file 3 rely on file2, and finally file 10 rely on file 1. One of the test file is shown below:

```
B0
R A 1
D A
C
```

And there are 10 loops like this, I will change the name of the table for each loop. After building up the benchmark, I use roundRobin to test the result, and the sizes of buffer are all 4. As can be seen from the result, the detector finds out the deadlock, and for the statistic part, we can see the number of transactions, the percentage for each operation, and the execution time and average response time. The average response time for this is 0.173 ms.

For the R-E part, the settings for the benchmark is the same as R-D part, I just use the E operation instead of D operation. One of the test file is shown below:

```
B0
R A 1
E A 1
C
```

And the result shows that the deadlock is detected, and the average detection time is 0.157ms.

For the R-W part, the setting for the benchmark is the same as R-D part, one of the test file is shown below:

```
B0
R A 1
W A (1, Thalia, 412-111-1324)
C
```

and the result shows that the average response time is 0.133ms.

For the W-D part, you need to set the conflict mode to WD, and each test file will first write on a table, then delete another table, for example, file 1 will write on table A, and delete table B; and then file 2 will write on table B, and delete table C. And finally, the last file will delete table A, so there is a long loop deadlock. And between the big loops, the difference is that I change the key value of the table file wants to write. Here I set the outer loop size to be 10 again, and the inner size to 9. Because the system will frist handle file 1, 10 and 100, so if you want to build up a deadlock loop, the inner loop should be less than 10. One of the test file is shown below:

```
B0
W A (1, Thalia, 412-111-1324)
D B
C
```

And the transaction manager settings are the same, the result shows that there are deadlocks, and the average response time is 0.167ms.

For the W-E part, the benchmark and manager settings are the same with W-D part, and one of the test file is shown below:

```
B0
W A (1, Thalia, 412-111-1324)
E B 1
C
```

And the result shows that there are deadlocks, and the average response time is 0.159ms.

For the E-D part, I will first let the file to write on a table, and it will then erase this tuple, and then going to delete another table. For example, for file 1, it will write on table A with tuple a, and going to delete this tuple a. Finally it will delete table B. And file 2 will write on table B with tuple b, and then delete this tuple, and then erase table C. And finally the last file is going to delete table A, those files will form a big loop. Here I also set the outer loop size to be 10 and the inner size to be 9. One of the test file is shown below:

```
B0
W A (1, Thalia, 412-111-1324)
E A 1
D B
C
```

And the manager settings are the same. The result shows that there are deadlocks, and the average response time is 0.161ms.

iii. Recovery:
For this part, I can use the benchmark generator to generate the files we want here, as after the deadlock detection, we can test the efficiency of the recovery system.

Others

a. A detailed description of the contribution of each team member based on the implemented components.
   i. Transaction Manager: Ziyi Huang
   ii. Scheduler: Sudeep, Antony
   iii. Testing: Lingrui Ouyang