-Yoshita Buthalapalli yrb3@pitt.edu

Github link:

https://github.com/yoshita120/CS2550-DBMS-project

PHASE 2:

Random read mode has been implemented. More details at the end of the report under Transaction manager heading.

Scheduler:

Consists of :

- RecoveryManager.java
- Scheduler.java
- LockTable.java
- LockManager.java

Recovery manager has abort function that reads the log file backwards and reverts the write and update operations for LSM. We could not implement recovery for Sequential disk due to time limitations.

LockTable.java has the code for acquirelock,releaselock and detecting deadlock using the provided py file.

The locks we used are T-WL,T-RL,P-WL,T-MREAD.

P is for process and it has only one lock i.e P-WL. This is because processes are read uncommitted, hence they don't need read locks. And a process can read any data even if it is locked.

For reading, the lock table each row has <item name, locks granted list, waiting list>.

Item name is table name for table lock and tablename_userid for row lock.

If a transaction is writing, then another transaction cannot issue do Mread. It must wait until the transaction releases the file. Hence we lock the file with intention to write.

The scheduler works with LSM, we couldn't test with seq file organization.

Executing this project:

We are using python file provided for detecting deadlocks. Hence the location of python must be specified in the project. Please make the following change to run the project locally:

In LockTable.java, line 224:

```
            ProcessBuilder processBuilder = new
ProcessBuilder("C:/Users/Yoshita/AppData/Local/Programs/Python/Python39/python.exe", "C:/Users/Yoshita/Desktop/CS2550-DBMS-project-1/DeadlockDetector.py");
```

Update the location of python.exe. and the location of the DeadlockDetector.py code which we have provided in the repository.

Delete any tables from previous execution like X.txt files as the code just appends to existing files.

To execute the project we can run the following commands:

-bash-4.1$ javac *.java

-bash-4.1$ java Main

Now enter the required input like disk block size etc. Once execution finishes, the output will be stored in outputs folder.

Sample arguments: in this case the output log will be stored in outputs/test_consurrency_normal_Test1.txt

```
Enter testcase directory:
tests/concurrency/normal/Test1
Choose readMode roundrobin or random: Enter rr or r
rr
Choose a memory strategy to implement - Sequential File Strategy 'sfs' or LSM Strategy 'lsm':
lsm
Enter cacheCapacity in bytes (should be a multiple of diskblocksize):
48
Enter Diskblocksize in bytes (should be a multiple of 12):
24
Enter SSTableCapacity (number of diskblocks in a sstable):
2
```

The Sequential file disk strategy can be tested better by uncommenting the main() section in SeqFileStrategy.java

The LSM file disk strategy can be tested better by uncommenting the main() section in LSMStrategy.java

- *We have already run the program on test cases and the outputs are stored in outputs folder. If you want to run the same test case again then the log file in output folder gets appended..hence you the output folder files can be deleted to test and check logs freshly.*

PHASE 1:

Disk Manager:

We are assuming that every record has a fixed size of 12 bytes. (user_id, user_age, satisfaction_score)

Sequential File Strategy: SeqFileStrategy.java

INPUT:	*disk block size* - in bytes, it should be a multiple of 12 *Cache capacity*- in bytes, it should be a multiple of diskblock size
OUTPUT:	log.txt in the current working directory tablename.txt files in the current working directory

The terms Page and disk block have been used interchangeably in the following text.

Index access method used: **ISAM**<user_id,pagenumber>: key is the first user_id of each page.

Cache: It is implemented using a LinkedHashMap. We are using write through cache.

Cache get: Everytime a page is accessed from the cache, it is removed from the cache and placed at the back of the cache list.

Cache put: Everytime a page is put in to cache, it is inserted at the back. If the cache is already full, then the front of the cache is removed. This way LRU is replaced.

Functions:

pagetable: stores the

pagetablerecord: stores the tablename,cacheindex, bool incache

INSERT: *seqWrite(tablename,record)*	• Create tablename.txt if it does not already exist. • Create ISAM index structure for this table if it doesn't already exist (say indexlist). • Fetch the pagetable for this tablename(say X). • As this should be a sequential ordered file, find the page number of the disk block where we need to insert this user_id. The page number is found using the ISAMlist. (If we should insert the record in the end and if the last page is full, then create new page. Add it to the ISAM list). • Refer pagetable to check if this page is in cache. If not in cache then swapin.If cache is full, then we used LRU to replace a page. Update the cacheindex for this page in the pagetable after swapping in. • Read the records in this page and insert the new record in correct position. • Update the new page in cache and in the tablename.txt • Update the ISAM list if the new record was inserted in the beginning of the page.

	• If the page was full even before inserting the new record, then we need to insert the leftover record in the next suitable page/block. So, we user recursion to insert the leftover record. seqWrite(tablename,leftoverrecord)
UPDATE	• Get page number from the ISAMlist. • Check if this page is in the cache using the pagetable. If not in cache, then swap in. • Check all the records in this page and update the appropriate record. • Replace the old page with this new page in the tablename.txt file. • Update the old page with new page in the cache.
READ	• Abort if tablename.txt does not exist • Get pagenumber from ISAM list. • Check in pagetable if this page is in cache or not. • If not in cache then swapin and update pagetable. • Read all records in this page to find the required record. • Log and return the record.
MREAD	• SwapIn all pages one by one into the cache. (no need to swapin if the page is already in cache). • Compare age/score as requested. • Add all the matched records to ArrayList • Log in log.txt and Return the arraylist
G	• Optimized to save the time of reading every record. • For every tablename, we store a Gnode object. • For every insert/update to a table, we update the Gnode.min,Gnode.max and GNode.avg accordingly. • So when user requests G X min; we can simply return get the gnode for table X and return Gnode.min

LSM-tree Strategy:

INPUT:	Cachecapacity, diskblocksize: same as above Sstablecapacity: number of diskblocks that a SStable in level zero can hold Number of SSTables in level zero
OUTPUT:	log.txt /lsmtables: all SSTables are added to this folder. Create this blank folder before executing lsm code.

Cache: It is implemented using LinkedHashMap. It can store (int)Cachecapacity/diskblocksize number of blocks. LRU is used to replace old blocks.

Memtable is a TreeMap<userid,record>: Each tablename has its own memtable. Used a treemap to eliminate duplicate keys in memtable.

All the memtables are store in another HashMap, where tablename is the key.

SSTable is stored in disk as a .txt file. For example: L0X1.txt is the first SSTable for tablename X at level 0.

As we go down the levels, the number of SStables reduces to $1/10^{th}$ the above level. And the size of SSTable doubles. In the final level, there will be only one Large SSTable.

Functions:

INSERT: writeLSM(tablename,record)	• Get the memtable for the tablename from the map if exists. Else, create a new memtable for this tablename. • Add the record to the memtable where userid is the key • Step 3: Check if the memtable is full. If it is not full then return. If yes, then we need to flush, so first check the number of SSTables in level 0. • If the Level 0 is full then call startCompaction function. • Now, to flush the memtable, create a level 0 SSTable. • Store the memtable records in blocks. Add the blocks to ArrayList and store this arraylist object into the respective level 0 SStable .txtfile.
UPDATE updateLSM(..)	• Call readLSM for the user_id to fetch the record. • Update the score in this record. • Now, add this record to the memtable. Update the cache block if it has this record. Check if memory table is full(continue step3 from the above cell).
READ	• First check in cache for the user_id. • If not in cache, then check in memtable • If not in memtable then start searching from level zero to finallevel. • In each level, start searching from latest SSTables to older ones because we want to read the latest updated record only. • For each SSTable, read it blockwise. Fetch each block to cache and read. • Stop once you find the record
MREAD	• Check the memtable first • then start searching from level zero to finallevel. • In each level, start searching from latest SSTables to older ones because we want to read the latest updated record only. • We want to read only the latest record for each user_id. Hence use a HashSet to store all the user_id's of the records that were read. • For each SSTable, read it blockwise. Fetch each block to cache and read. • An arraylist of results is returned.
G	Same as sequential file strategy.

<u>Transaction Manager:</u>

INPUT:	Script File, Strategy Strategy: SequentialFileStrategy(sfs) and LSMStrategy(lsm) are implemented.
OUTPUT:	On the terminal

Both concurrent reading methods has been implemented: a Round Robin which reads one line from each file at a time in turns. When the Main file is run, the user is asked to select the type of concurrent reading to proceed with(currently only RoundRobin). When the user enters “rr”, transaction manager will read the transactions in round-robin method. “r” for random works.

The transaction manager will first iterate all the files under the script directory and generates 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 according to the reading mode enter by the user at the beginning. i. roundRobin(fileList, transactionBuffer, transactionMode)

After running this method, the transactionList is iterated to read the operations and perform them accordingly. In each transaction, the type of instruction is read from the first character, the table is the second character. Based on the type of operation in the current transaction, the necessary values are extracted and passed to the respective strategy function as selected by the user in Main class. Based on this, either sfs or lsm functions as described above in Sequential File Strategy and LSM strategy are called by passing the read instructions from the current transaction. The outputs of these file strategies in stored in log.txt in outputs folder.

roundrobin	roundRobin(fileList, transactionBuffer, transactionMode) The method creates 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. • The fileList is iterated when the transactionMode is not empty. bufferReader in the transactionBuffer is used to map one new read line from the file. • If the new line is not null, then store the new line in the list created. 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.
Random	random(List fileList, Map transactionBuffer, Map transactionMode, long randomSeed, int maxLines) The method creates 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.

	• We create a random number generator using the given seed. While the transactionMode is not empty, generate a random integer that is larger or equal to 0 and smaller than the size of ransactionMode, 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. While the readRow is larger 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. At last it calls the scheduler execute method.