CS2550 Project Phase 1 Report

TEAM 1

1. Sequential Part:

We have several main classes for baseline: myPTA, Buffer, Table, Page, Record

myPTA is for reading and processing the operation from the script. It is the entrance of the program. It also has a hashmap<String, Table> *tables* which stores all the tables instances. We can know if a table exists by checking this hashmap.

For each table we will create a folder naming with the table name. Within each table(directory), we create a txt file for each page starting with the index 0 and name it with the index (e.g. page0.txt). Records are recorded in pages one record per line and are inserted sequentially. When the size of records in one page has reached the size of a page, the next inserting record will be added to a new page with index added by 1.

We use a buffer as the read/write cache. The data structure we use for buffer is LinkedHashMap where the key is tableName+page_No and value is an instance of Page. It can remove least recently used (LRU) entry when size of LinkedHashMap is greater than maxsize user specified. Before removing it, we will write this page back to disk for consistency.

When a reading operation (associated with a table) is processing, we check all the pages from index 0 of that table one by one until we find the wanted record. For each page to be processed, if the page is already in the buffer, just process it (There is an ArrayList<Record> attribute *records* storing all records in Page). Otherwise, we will load the specified page from disk (implement by reading that txt file) to Buffer and then process it.

When an erasing operation comes, similarly we check pages from index 0 of that table one by one. If we don't find the record, just return with reporting. Otherwise, we delete that record from *records* of that Page, and if this page is not the last page, we push the index of this page to Stack<Integer> attribute *freeSpace* of that Table. When inserting a record in this table, we will pop out the element on the top of the stack and insert this record to the page with this top index.

When a writing operation comes, similarly we check pages from index 0 of that table one by one. If we find the record with equal key, then just update this record in the page. Otherwise, it means it is an insert. We will first check *freeSpace* in Table: If it is empty, then find if the last page is full. If it is not full, just append the record to the last page. Otherwise, create a new page with index added by 1 and add to this page. If *freeSpace* is not empty, then pop out the element on the top of the stack and use it to be the index of the page where we insert the record sequentially to that page.

When a deleting table operation comes, we remove that table instance in hashmap *tables* and if that table existed before, we delete that directory on the disk. There would not be an issue for the possibility of having some page of that deleted table in Buffer, because when

the operation is read or write, we will first check if this table exists in *tables*. If not, the operation is aborted and there would be no possibility of dirty read.

Test Cases and anticipate result:

// for R	Expected results:
R X 5	aborted
W X (5, John, 412-111-2222)	create table and success insert
R X 5	(5, John, 412-111-2222)
R X 6	not exist
// for W	
W X (5, Ben, 412-111-2222)	update successfully
R X 5	(5, Ben, 412-111-2222)
W X (6, Ann, 412-222-1111)	insert successfully
R X 6	(6, Ann, 412-222-1111)
// for multiple update	
W X (6, Jack, 412-222-1111)	update successfully
W X (6, Mike, 412-222-1111)	update successfully
R X 6	(6, Mike, 412-222-1111)
// for E	
E X 7	not exist
E X 6	success
R X 6	not exist
// for D	
D X	success
R X 6	aborted
W X (6, Ann, 412-222-1111)	create table and success insert
R X 6	(6, Ann, 412-222-1111)

// for M

M Y 412	aborted
W Y (10, Mary, 412-333-3333)	create table and success insert
W Y (11, Paul, 412-4444-4444)	insert successfully
M Y 412	(10, Mary, 412-333-3333) (11, Paul, 412-4444-4444)

Logging: (page size = 2, buffer size = 3)

R X 5

The table does not exist, the read is aborted.

W X (5, John, 412-111-2222)

Create T-X P-0

Swap in T-X P-0

Written: 5, John, 412-111-2222

R X 5

Read: 5, John, 412-111-2222

R X 6

The table X doesn't have record with ID = 6

W X (5, Ben, 412-111-2222)

Written: 5, Ben, 412-111-2222

R X 5

Read: 5, Ben, 412-111-2222

W X (6, Ann, 412-222-1111)

Written: 6, Ann, 412-222-1111

R X 6

Read: 6, Ann, 412-222-1111

W X (6, Jack, 412-222-1111)

Written: 6, Jack, 412-222-1111

W X (6, Mike, 412-222-1111)

Written: 6, Mike, 412-222-1111

R X 6

Read: 6, Mike, 412-222-1111

E X 7

There is no record with ID = 7 in Table X

E X 6

Erased: X 6

R X 6

The table X doesn't have record with ID = 6

D X

R X 6

The table does not exist, the read is aborted.

W X (6, Ann, 412-222-1111)

Create T-X P-0

Swap in T-X P-0

Written: 6, Ann, 412-222-1111

R X 6

Read: 6, Ann, 412-222-1111

M Y 412

The table does not exist, show user with area code is aborted.

W Y (10, Mary, 412-333-3333)

Create T-Y P-0

Swap in T-Y P-0

Written: 10, Mary, 412-333-3333

W Y (11, Paul, 412-444-4444)

Written: 11, Paul, 412-444-4444

M Y 412

MRead: 10, Mary, 412-333-3333

MRead: 11, Paul, 412-444-4444

Runtime: 12

2. LSM-tree Strategy:

1 record placement, file organization, and memory management strategies

ID is primary key of each record and records are sorted by ID and stored line by line in files. Each file contains a unique id at first line and followings are records. A file is considered as a SStable. When initialize database, user needs to specify a base directory to store files. Each file is stored under a folder, naming levelx, which indicates which level SStable is at in LSM-Tree. Level folder is under table name folder and table name folder is under base directory. In general, files are stored as Basedir/tableName/levelx/files.

Tables are managed by a hashmap. Each table has a corresponding LSM-Tree and LSM-Tree stores meta data of this table in memory. Each LSM-Tree has a memtable and information of all its SStables. Memtable has size of k and uses a hashmap to store records and ids are keys. It contains at most k-1 records in memory and will flush to disk when kth record is written. SStable information is stored in two parts, level0 and other levels. Each SStable has a corresponding object in memory, which contains location of actual file in disk, number of records, a unique id and largest and smallest ids of records it has. When a memtable flush to disk, memtable's id becomes SStable's id. Level0 is a simple array of SStables with size 4. All other levels are stored in an array of treeSet. TreeSet is a sorted

datastucture and SStables are sorted by smallest id. Each treeSet is corresponding to s level.

Read Cache is shared with all tables. Read Cache is implemented by a LinkedHashMap<String, MemTable>. LinkedHashMap can remove least recently used (LRU) entry when size of LinkedHashMap is greater than maxsize user specified. Since it is a read cache, writing back to disk si is not needed. The only time an entry could be invalid is during compaction. When performing compaction, LSM-Tree will drop all SStables that are involved in compation from read cache. This will be described more detailed with compaction.

2. Operations

Write:
First, LSM-tree will write record to memtable. If memtable is full, records in memtable will sorted, wirtes to a file and add a new SStable to level0. After that, memtable will be empty and able to receive new records. Then LSM-tree checks if it needs compaction and compaction only happens when level 0 is full. Compaction involves two parts compact and compact other levels.

When compacting level 0, LSM will pick oldest SStable in level0 as the chosen one and find all SStables that overlap with chosen one in level0. Then SStables in level0 can be combined and get a final range of ids. Then LSM will choose SStables in level 1 that overlap with final range. Those SStables are all SStables needed for compaction level0 and others will remain unchanged. In this process, LSM will pick at most 14 SStbales(4 in level0 and 10 in level1). Then compaction function will load chosen SStables to memory and put all records to a large set to remove duplicate. Every time LSM-tree load SStables from disk, it will first check whether they exist in read cache by check their unique ids and only read from disk when they are not in cache. SStables are loaded one by one and for each record add <id, SStable> to map in the descending order of age. Level1 SStables are older than level1 and level0 SStable can be easily order by age. In this way, newer values will overwrite old values and perform correct compaction. Then map is sorted by id and are written to files. We write by order of ids. We get an id, load its SStable from buffer or disk, find record and write to new SStable. When size of SStable is over max size, we switch to a new SStable. Also, when current SStable overlaps with more than 10 SStables in next level, we switch to new a new SStable.

After compacting level0, level1's size increases, which may trigger compaction. If level1 size is bigger than maximum size, it needs to be compacted. LSM will pick only one SStable in level1+ in single compaction. We choose a SStable according to SStable we chosen from last compaction. This SStable is the one which is the smallest one whose smallest key is greater than last chosen's greatest key. In first compaction, we will choose smallest SStable and if such SStable does not exist, we will also choose the first one. Then we find all SStables that overlap with chosen one. Then we follow the same procedure as compacting

level 0. Since we only allow a SStable overlap with at most 10 SStables, we will load at most 11 SStables. We keep check size of next level until every level is satisfied.

Erase:
For Erase, it is mostly same as write. We write a dummy record with phone number as “delete”, The only difference is before we insert dummy record, we first read id to check if record exist. If it does not exist, we do nothing. Dummy record will not be compact to a new oldest level. For example, currently LSM-tree have level3 and a SStable in level3 contains dummy records. When that SStable is chosen as the SStable compacting to level4 and level4 is currently empty, dummy records will be dropped.

Delete:
Delete will recursively delete all files and folders under base directory and LSM object will be garbage collected.

M:
Simple full scan

Read:
Read a record is similar to a full scan. LSM-tree will first search in memtable, level0 and level1+. In level0, LSM-tree will search SStable in ascending order of age. **It will only load a SStable from disk if key we search is between smallest and largest values of the SStable.** Once we find one record’s id equal to key, we will stop search and return record. If the first record we found is marked as delete, we return record not found. will In different conditions like multiple updates, deleted records and deleted tables, read will still be correct.

Multiple updates:
If multiple updates are not flushed to SStable, newer update will overwrite older values in hashmap. Thus, two updates cannot be in same SStable. If two updates are both in level0 and different SStable, we will read newer SStable first and return. Similarly, we will visit level 0 before level1+ and read the latest update.

deleted records:
Only if the latest update is deletion will LSM-tree consider a record is deleted. Thus, deleted records can be considered as a update and we have shown we can read correctly under multiple updates.

deleted tables:
Tables are managed by a hashmap. Every time reading from a table, we will check if table name exist in hashmap. Thus, if we found a table is not in hashmap, we will know it is not be created or has been deleted.

Test Cases and anticipate result:

// for R	Expected Results
R X 5	aborted
W X (5, John, 412-111-2222)	create table and success insert
R X 5	(5, John, 412-111-2222)
R X 6	not exist
// for W	
W X (5, Ben, 412-111-2222)	update successfully
R X 5	(5, Ben, 412-111-2222)
W X (6, Ann, 412-222-1111)	insert successfully
R X 6	(6, Ann, 412-222-1111)
// for multiple update	
W X (6, Jack, 412-222-1111)	update successfully
W X (6, Mike, 412-222-1111)	update successfully
R X 6	(6, Mike, 412-222-1111)
// for E	
E X 7	not exist
E X 6	success
R X 6	not exist
// for D	
D X	success
R X 6	aborted
W X (6, Ann, 412-222-1111)	create table and success insert
R X 6	(6, Ann, 412-222-1111)

// for M

M Y 412	aborted
W Y (10, Mary, 412-333-3333)	create table and success insert
W Y (11, Paul, 412-4444-4444)	insert successfully
M Y 412	(10, Mary, 412-333-3333) (11, Paul, 412-4444-4444)

Logging: (SSTable size = 2, buffer size = 3)
R X 5
The table does not exist, the read is aborted.

W X (5, John, 412-111-2222)
Written: (5, John, 412-111-2222)

R X 5
Read: (5, John, 412-111-2222)

R X 6
The table X doesn't have record with ID = 6

W X (5, Ben, 412-111-2222)
Written: (5, Ben, 412-111-2222)

R X 5
Read: (5, Ben, 412-111-2222)

W X (6, Ann, 412-222-1111)
Written: (6, Ann, 412-222-1111)
create L-0 K X-5 X-6

R X 6
Swap in L-0 K X-5 X-6
Read: (6, Ann, 412-222-1111)

W X (6, Jack, 412-222-1111)
Written: (6, Jack, 412-222-1111)

W X (6, Mike, 412-222-1111)
Written: (6, Mike, 412-222-1111)

R X 6
Read: (6, Mike, 412-222-1111)

E X 7

Erased: X 7

E X 6
Erased: X 6

R X 6
The table X doesn't have record with ID = 6

D X
Deleted: X

R X 6
The table does not exist, the read is aborted.

W X (6, Ann, 412-222-1111)
Written: (6, Ann, 412-222-1111)

R X 6
Read: (6, Ann, 412-222-1111)

M Y 412
The table Y does not exist, show user with area code is aborted.

W Y (10, Mary, 412-333-3333)
Written: (10, Mary, 412-333-3333)

W Y (11, Paul, 412-4444-4444)
Written: (11, Paul, 412-4444-4444)
create L-0 K Y-10 Y-11

M Y 412
Swap in L-0 K Y-10 Y-11
MRead: (10, Mary, 412-333-3333)
MRead: (11, Paul, 412-4444-4444)
Runtime: 30

3. Comparison

10000 records
B: Buffer Size
S: SSTable Size
P: Page Size
Efficiency (throughput): records per second

Sequential	B = 22 P = 10	B = 22 P = 100	B = 22 P = 200	B = 12 P = 200
write	5.53	48.43	73.32	93.90
read	4.77	43.42	102.01	96.71

LSM-tree	B = 20 S = 10	B = 20 S = 100	B = 20 S = 200	B = 10 S = 200
write	1544.40154	2423.65487	2452.18244	2797.98545
read	4755.11175	4079.96736	3189.79266	3210.27287

Analysis:

We can clearly see from the result that when buffer size get larger, both methods appears a higher efficiency on READ. This is obvious because we can directly access a data unit (a page or a SSTable) in buffer.

When the data unit gets larger, we will get higher efficiency in both methods. This is because the smallest I/O unit's capacity is inverse proportional to I/O operation (read and write disk) times.

LSM Tree read is optimized by using Tree structure to store SStable information of each level. For each level1+ , records we search can only exist in one SStable and we can use floor function to find SStable efficiently. If floor function returns null, we can skip to the next level. All these operations happen in memory and we only need to load at most one SStable each level.

We can see that LSM Tree method is significantly faster than Sequential method. This is because in sequential method, every time we write a new record in table we need to firstly loop through the whole table to check if the primary key exists. However, in LSM tree method, we only need to insert the new record to MemTable.

4. Contributions

For the Sequential part, who implemented the Buffer, the Table, the Page, the Record, the read operation
For the LSM-tree strategy, who implemented the memtable, the SSTable, the compaction algorithms

Guo: Fangzheng Guo
Zhu: Zhiben Zhu
Yao: Yuan Yao

Sequential:

Buffer	Table	Page	Record	Data Manager	Logging	Operations	Testing	*Initialization
Guo, Zhu	Guo	Zhu	Zhu	Guo, Zhu	Guo	Guo, Zhu	Guo	Guo

LSM-tree:

Memtable	Read cache	Compaction	Data Manager	Logging	Operations	Testing
Yao	Yao	Yao	Guo, Zhu	Guo	Yao	Yao, Zhu

Report:

Sequential	LSM-tree	Comparison
Zhu	Yao	Guo