CS561 : Log Store Merge (LSM) Tree System Project

Adit Mehta
Boston University
U98200191

Amara Nwigwe
Boston University
U57410823

Huda Irshad
Boston University
U59175841

Satha Kitirattragarn
Boston University
U01186635

ABSTRACT

The foundations of nearly all modern database systems concern the quintessential key-value database paradigm. Two staple implementations thereof—logs and sorted arrays—provide either efficient reads or updates while critically sacrificing the other quality. As a compromise, Log-structured merge (LSM) trees—along with tiering and leveling techniques—provide a configurable key-value environment, tailoring to the exact workload requirements, i.e. how read-heavy or update-heavy will the workload tend to be. This paper aims to explain our teams implementation of the LSM tree and the decisions we made in the process.

As far as architecture, we explain our thought processes when it comes to implementing our write, delete, and get policies. We have built on the given *TemplateDB*'s multiple helper classes and functions like *write_to_file* and the **get** function in order to help us with our implementation.

1 INTRODUCTION

1.1 Motivation

In the realm of computing technology, there has always been an insatiable demand from clients of all technological sectors for faster and more reliable database systems for data management. Traditional key-value designs are stored as logs or sorted arrays. Logs store values linearly and sequentially, which results in minimal update cost at the great expense of read cost. On the other hand, sorted arrays store values in organized chunks, which results in minimal read cost but at the high expense of update cost. To compromise between these two extremes, log-structured merge trees are proposed, where concepts of tiering and leveling will allow users/clients to tune their key-value engine in relation to read/update cost more minutely to their desired workload. Such granular control on in this LSM-Tree design motivates us to attempt to implement it in practice for experimentation and better understanding.

1.2 Problem Statement

We will be implementing an LSM-Tree design in a simulated environment using C++ and benchmarking how it performs quantitatively under a set of circumstances. These will be determined by various tunable factors such as merging policy, compaction design and data threshold for merging/flushing on each LSM level.

2 BACKGROUND

LSM-Trees were invented in 1996 to satisfy the read/write performance gap left by logs and sorted arrays, the only available key-value stores known then. An LSM-Tree is composed of levels of log-structured data layout—a linear, contiguous array of data slots—where the lower level contains more slots that the upper level. When data first enters the LSM-Tree database, data is stored at the highest level L0. When L0 becomes filled, it "flushes" its whole log to the lower level L1 by copying the log onto L1 and empties its log for further writes; when L1 becomes filled with L0 flushes, it flushes to L2, and so on. When an search query is issued to the tree, it searches from the uppermost level L0 first; if the desired data isn't found there, the tree moves on to search in L1, and so on. Therefore, the levels store data of different recentness—under the assumption that more recent data are likely be more frequently read/updated than less recent one—and the size discrepancy between levels reflects the idea that less recent data are of a greater quantity than the more recent one.

3 ARCHITECTURE

3.1 On-board Memory and Main Memory Layout

In our general memory layout, the on-board memory is designed to hold a specific size as defined in our database write function. In our conception, the query engine will continually append data that inserts unique keys within the on-board memory. When the on-board memory reaches its capacity, that memory will be placed as a run in the first level of the main memory (L0). If runs already exist in that level, then the run will be placed at the tail of the level so that it can be acknowledged as the most recent data. We've implemented our engine such that the ratio of runs to a single Sorted String Table (SSTable) is two to one. We've configured the SSTable to contain a header which provides the metadata consisting of the number of elements as well as its minimum and maximum keys. All files representing an SSTable will be preallocated; the header will tell the system whether the SSTable is unused. We wrote the engine such that the header reading -1 as the number of elements and 0 as the minimum and maximum key indicates that the contents of the SSTable is empty.

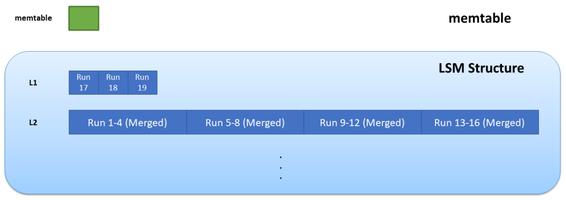

Figure 1: Memory Layout

3.2 Operational Policies

The following segments will be database functions we've implemented: **write** (bulk-insert), **put** (point-insert), **get** (point-query), **scan** (range-query and bulk-query), and **delete** policies (point-delete and range-delete). They comprise the fundamental operations expected of an LSM store.

3.2.1 Write Policy. When inserting values, our design dictates that the tree will always write to the on-board memory table. In the case of the LSM-Tree, its maximum capacity is set to 50 inserts. This choice serves to ease experimentation and debugging in the current milestone of the project, which we will further modify to be tunable in subsequent development timeline. Once the on-board memory table has reached its maximum capacity, it will write its contents to a run to the level below, L1.

Writing to memory. When writing to memory, our method includes writing all of the data according to the instructions of the data file. The writes are simply appended to the end of the file and the memtable is therefore updated. Regardless, we've capped a maximum capacity under which data can be written. Once the memtable is full, it is to be flushed to one level below. In our case, it would be flushed to the first run of the second table.

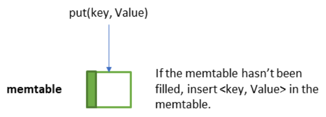
put(key, Value)
memtable
If the memtable hasn't been filled, insert <key, Value> in the memtable.

Figure 2: Write Case 1: Adequate Space in the memtable

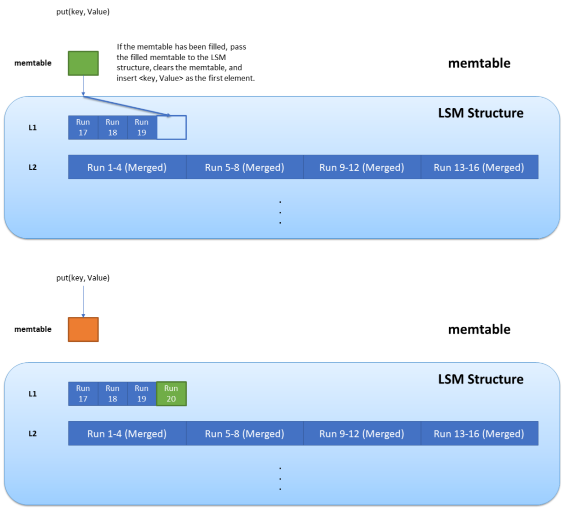

Figure 3: Write Case 2: Inadequate space in the memtable

3.2.2 Put Policy. The put policy involves inserting a single <key,value> pair into the database by attempting to insert it into the memtable first. If it's unsuccessful, the LSM engine writes the memtable to the LSM database according to the storage scheme, clears it, and inserts the <key,value> pair into the now-empty memtable.

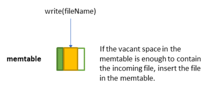
write(fileName)
memtable
If the vacant space in the memtable is enough to contain the incoming file, insert the file in the memtable.

Figure 4: Put Case 1: Adequate Space in the memtable

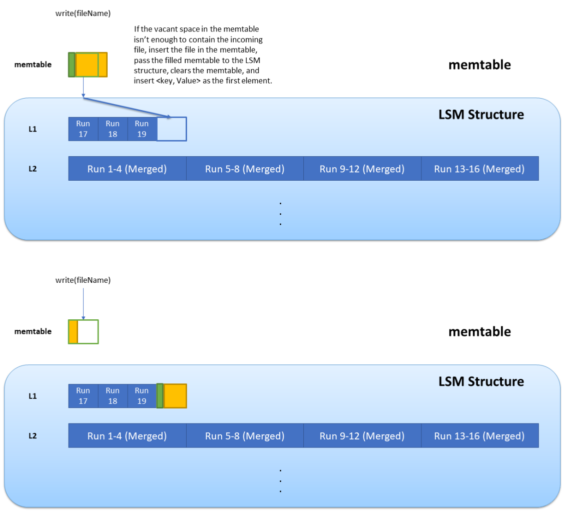

Figure 5: Put Case 2: Inadequate space in the memtable

3.2.3 Get Policy. We implemented the get policy as a function with a key as its input which can read files to obtain the desired value should the key was not found in the memtable. This function will sequentially go through each run and search until there is a call to our *write_to_file*, upon which the engine will initiate scanning for that value and writing it to the designated file.

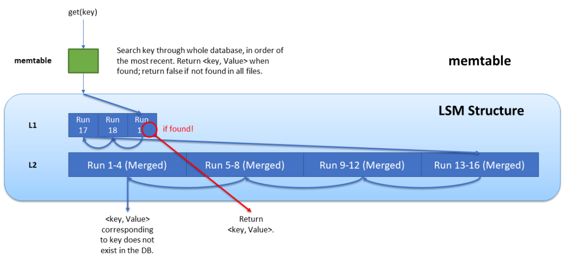
get(key)
memtable
Search key through whole database, in order of the most recent. Return <key, Value> when found; return false if not found in all files.
memtable
LSM Structure
L1
Run 17
Run 18
Run 19
if found!
L2
Run 1-4 (Merged)
Run 5-8 (Merged)
Run 9-12 (Merged)
Run 13-16 (Merged)
<key, Value> corresponding to key does not exist in the DB.
Return <key, Value>.

Figure 6: Get Policy

3.2.4 Scan Policy. The scan policy entails reading elements from the memtable and the LSM database. (****Ray: Delete "and the LSM database" if we're only reading from the memtable.****) We implemented the scan policy in two modes: an argument-less function which will perform the total scan and output all values found in the memtable and the LSM database and a function with the minimum key *min_key* and the maximum key *max_key* and output all values within this bound it's been given.

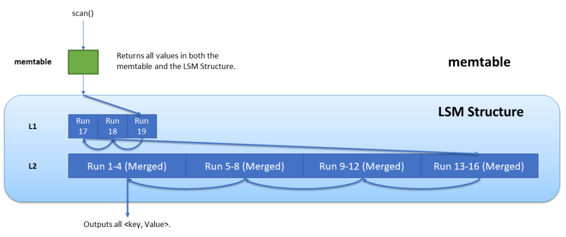

Figure 7: Bulk Scan

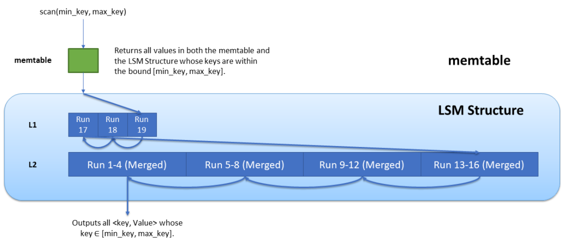
scan(min_key, max_key)
memtable
Returns all values in both the memtable and the LSM Structure whose keys are within the bound [min_key, max_key].
memtable
LSM Structure
L1
Run 17
Run 18
Run 19
L2
Run 1-4 (Merged)
Run 5-8 (Merged)
Run 9-12 (Merged)
Run 13-16 (Merged)
Outputs all <key, Value> whose key ∈ [min_key, max_key].

Figure 8: Range Scan

3.2.5 Delete Policy. We've modelled the delete policy in such a way that each key-value pair will be allocated a tombstone-bit which will tell the user if the value entry has been expired. Deletion is not applied to all the existing key-value pairs under the same key, but, rather, to a key-value pair that is in the on-board memory table—the uppermost level—because it is the most recent value among pairs of the same key. If the key exists under the on-board memory table when the delete is requested, then the key-value pair is removed from the on-board memory table. If the key does not exist in the on-board memory, then the key is to be paired with a null value and the accommodating tombstone-bit is to be written as true, denoted by the integer "1". Vainly writing the tombstone as true for the old values will not be a concern since the LSM-Tree is designed to read for the most recent version by checking the most recent runs first and runs will not reach those deprecated values. In addition, as compaction takes place with the key-value pair coupled with the true tombstone-bit being the most recent, this version of the key-value pair will survive the compaction.

3.3 Storage Schemes

As we've described before, an LSM-Tree is born out of the attempt at compromising the benefits and cost of logs and sorted arrays. When there is incoming dataset, a log simply append the dataset at its tail without processing it in any way, benefiting write cost while suffering heavily on read cost. In the same situation, a sorted array will always merge the incoming dataset with its present dataset, resulting in low read cost but high write cost. As such, the tiering scheme mimics logs with its simple appending of the dataset flushed from the previous level while the leveling scheme mimics sorted arrays with merge being performed on the flushed dataset with the dataset extant in the level.

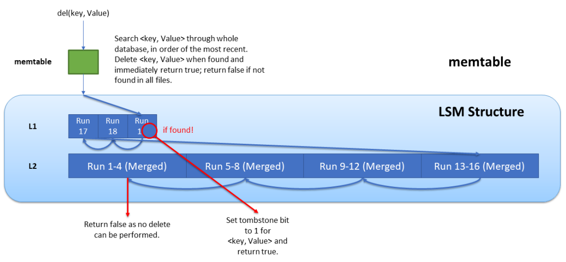
del(key, Value)
memtable
Search <key, Value> through whole database, in order of the most recent. Delete <key, Value> when found and immediately return true; return false if not found in all files.
memtable
LSM Structure
L1
Run 17
Run 18
Run 1
if found!
L2
Run 1-4 (Merged)
Run 5-8 (Merged)
Run 9-12 (Merged)
Run 13-16 (Merged)
Return false as no delete can be performed.
Set tombstone bit to 1 for <key, Value> and return true.

Figure 9: Point Delete

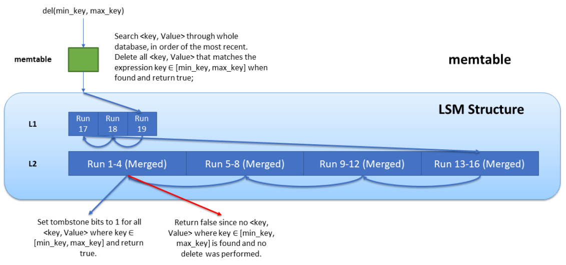
del(min_key, max_key)
memtable
Search <key, Value> through whole database, in order of the most recent. Delete all <key, Value> that matches the expression key ∈ [min_key, max_key] when found and return true;
memtable
LSM Structure
L1
Run 17
Run 18
Run 19
L2
Run 1-4 (Merged)
Run 5-8 (Merged)
Run 9-12 (Merged)
Run 13-16 (Merged)
Set tombstone bits to 1 for all <key, Value> where key ∈ [min_key, max_key] and return true.
Return false since no <key, Value> where key ∈ [min_key, max_key] is found and no delete was performed.

Figure 10: Range Delete

We implemented both of these schemes, with sorting on the randomized integer keys pertaining to the each set of values as our merging policy. The following is our implementation designs.

3.3.1 Tiering. Tiering is an LSM design described in two scenarios when there is an incoming dataset to be inserted: if the current level the dataset is to be inserted is not full, the dataset is simply appended to the end of the level; otherwise, the dataset extant in the level is merged together, flushed to its subsequent level, and the current level empties itself of the contents, and let the incoming dataset inserted into it.

3.3.2 Leveling. Leveling is an LSM design where, when there is an incoming dataset to be inserted: if the current level is not full, the incoming dataset is appended to the level and merged with the dataset already extant in the level; if the current level is full, the dataset in the current level will simply be flushed to the next level (as it has been merged in a previous operation), empties itself of the contents, and let the incoming dataset inserted into it.

3.3.3 Tunable Parameters. There are several design parameters which we engineered to be tunable in order to make our experimentation more flexible and holistic. These parameters are as follows:

(1) *tablesize*: The size of the memtable
(2) *numFiles*: The number of files in each level
(3) *tiering*: A boolean variable dictating either Tiering or Leveling

The first parameter fixates the storage capacity of the memtable which directly translates to how many data elements can be issued to it before it has to traverse into the LSM levels in order to retrieve data. For it to be too little in size will make the memtable insufficient for storing an adequate number of "hot"—or, frequently accessed—data and demands the engine to enter LSM levels more than necessary; for it to be large in size will make sacrifice the benefits of the LSM structure and reverting the design back to the log structure.

The second parameter dictates the maximum number of "conceptual" files in the level. In Leveling, even if each flush from a previous level will get merged with the extant dataset, always resulting in 1 file in the level, we conceptually consider the number of files in the level to be n as designated by this parameter in order to use n to calculate the size of the level it corresponds to.

As such, both parameters also dictate the size of the files in LSM's levels since each level's size is in increments of the memtable's size. For instance, each file flushed to L1 from the memtable is the same size as the memtable total size and each file flushed to L2 is the same size as L1's total size, equating to $(tablesize) \times (numFiles)$. In this way, the size of each level's file can be calculated with the following formula:

$$levelSize = (tablesize) \times (numFiles)^{(current\ level)}$$

4 NEXT STEPS

We are working on fine-tuning our operational policies working in tandem with our LSM engine. The write, put, and delete policies function as expected but the scan and get policies prove rather difficult to engineer despite several attempts at debugging. However, the expectation is for it to be resolved by the date of the live demonstration session in the upcoming Tuesday, where-after we will report on our LSM engine's performance under the "Experimentation" section in the final report. It will be on the writing/reading performance in detailed description as well as accompanying comparison graphs for visual clarity.

REFERENCES

[1] Sattam Alsubaiee, Michael J. Carey, and Chen Li. 2015. LSM-Based Storage and Indexing: An Old Idea with Timely Benefits. In *Second International ACM Workshop on Managing and Mining Enriched Geo-Spatial Data* (Melbourne, VIC, Australia) *(GeoRich'15)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/2786006.2786007

[2] Burton H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (1970), 422–426. http://dl.acm.org/citation.cfm?id=362686.362692

[3] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2018. Optimal Bloom Filters and Adaptive Merging for LSM-Trees. *ACM Trans. Database Syst.* 43, 4, Article 16 (dec 2018), 48 pages. https://doi.org/10.1145/3276980

[4] Chen Luo. 2019. LSM-based Storage Techniques: A Survey. *VLDB Journal* 29 (2019), 393–418. https://arxiv.org/abs/1812.07527

[5] Chen Luo. 2020. Breaking Down Memory Walls in LSM-Based Storage Systems. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2817–2819. https://doi.org/10.1145/3318464.3384399