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.

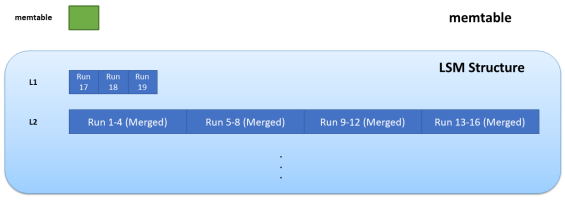

Figure 1: Memory Layout

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.

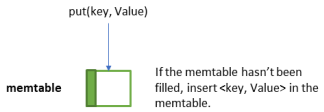

Figure 2: Write Case 1: Adequate Space in the memtable

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.

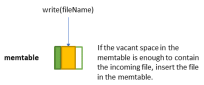

Figure 4: Put Case 1: Adequate Space in the memtable

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.

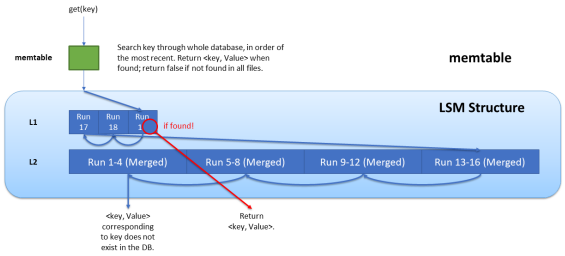

Figure 6: Get Policy

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. In particular, we count deleted entries—ones with the visibility turned off—as a valid entry to return as part of the scan as opposed to the scan showing only visible entries.

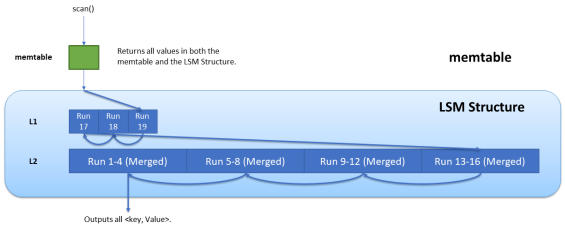

Figure 7: Bulk Scan

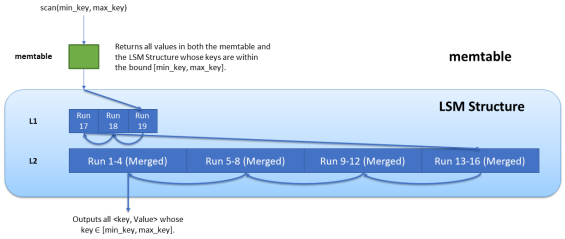

Figure 8: Range Scan

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.

3.2.6 TemplateDB Setup. In *TemplateDB*, we have modified a few functions in order to help with our implementation. *operation.hpp* and *operation.cpp* files contain codes and strings that represent the types of operations that a certain data file may have. These are mostly handled in the *main.cpp*, where we test our base implementation. These operations include **put**, **get**, **scan**, and **delete** operations. At the current milestone, we have handled **get** and **delete** operations.

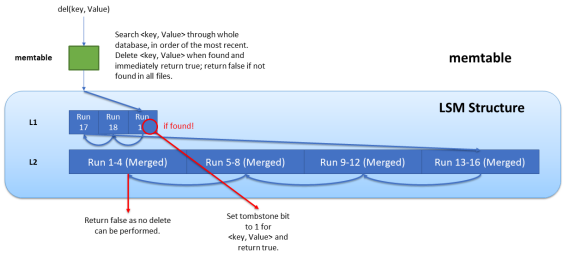

Figure 9: Point Delete

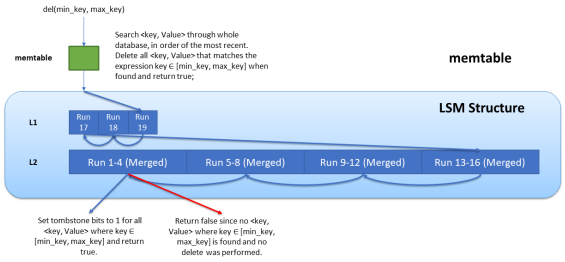

Figure 10: Range Delete

In *db.hpp* and *db.cpp*, we've built upon the functions that have already been given to us. In adding a *Levels* class to *db.hpp*, we account for the multiple levels of the LSM-Tree and use the variable *numFiles* to represent the number of runs per level. We also use the variable *fileNames* to represent the actual runs of each level.

In adding the *newfiles* function to *db.cpp*, we've created files that represent the runs of each level. We have also removed the *open* function in *db.cpp* as we have modified our *write_to_file* to open files once that function is called. One functionality of *write_to_file* is to add a header to the top of the file, signaling the number of rows as well as the minimum and maximum keys; another *write_to_file* functionality is where it takes the data from the *.data* files that have been previously generated and write them to each file. The delete function that we have modified currently sets the visibility of the inputted key to 0, to signify that the entry has been deleted from the file.

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.

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 EXPERIMENTATION

Upon building the code, we sought to test out the functions that we've built to compare how accurate and efficient our implementation of the LSM-Tree actually is to what we expect it to be according to our logical design.

4.1 Writing Performance

Point Inserts (Put). Our point inserts perform according to our expectations. The experiments show either a minimal processing time simply because it appends to the empty tail of the memtable or an astronomical processing time owing to the fact that the memtable is full and needs to be merged and flushed to the next level which, may or may not trigger a cascade of merges/flushes within the LSM-Structure before the processed memtable can be flushed and the incoming entry can be inserted.

Bulk Inserts (Write). With bulk-inserts, we insert in increments of files instead of entries. When the engine inserts the file into the memtable, it checks to see whether there are entries that extends beyond the full capacity of the memtable. If there are, it merges the full memtable, flushes it to the next level, and checks to see whether there are redundant entries again, and so on, until all remaining entries fit inside the memtable. The performance, again, hinges on how many merges and flushes the engine needs to perform in order to complete the insert.

TIERING (100,000 rows)

	Number of Inserts				
	100	1000	10000	100000	1000000
Dimensions	(s)				
100	0.072	0.048	0.284	3.384	38.469
1000	0.009	0.046	0.584	7.232	91.45
10000	0.024	0.163	2.944	38.345	520.638
100000	0.064	1.386	25.411	343.14	?

Figure 11: Bulk-Insert for Tiering

LEVELING (100,000 rows)

	Number of Inserts				
	100	1000	10000	100000	1000000
Dimensions	(s)				
100	0.054	0.051	0.554	7.723	82.637
1000	0.014	0.082	1.171	16.737	199.633
10000	0.019	0.363	6.811	98.15	1203.36
100000	0.072	3.084	60.44	884.98	?

Figure 12: Bulk-Insert for Leveling

4.2 Reading Performance

Point Queries (Get). Our point queries also perform according to our conceptual design. The experiments show a linear increase in processing time as we expected because the query searches in a linearly hierarchical manner. It performs the search from the most recent data file to the least recent and whenever the data is found, the observed entry will immediately get returned. For leveling, the query searches from the memtable to L1, then to L2, and so on. Tiering follows the same query paradigm as leveling but for each LSM-Structure level traversed, the engine queries from the last file of the level back to the first file.

Range Queries (Range Scan). For the range queries in both our tiering and leveling trees, we tested on an LSM tree that contained 100,000 inserts. When the workload contained just 1 range scan query with a max key of 200,000, it ran for about 12.304 seconds in our tiering tree and 14.487 in our leveling tree. There doesn't seem to be a large time difference between leveling and tiering when it comes to searching for the appropriate values that the range query was looking for. With a workload of 10 range scan queries, tiering was shown to take a bit longer, with a time of 149.42 seconds, compared to leveling's 119.83 seconds. This was evident

when it came to running a workload with 50 scan queries. The range scan on the leveling LSM tree ran for 628.51 seconds, compared to tiering's 725.1 seconds. Leveling's shorter running time can be attributed to the fact that it is read-optimized. Unfortunately, when testing out greater workloads, we experienced some timeout issues, preventing us from seeing further results.

Number of Reads (Get Queries)	1	10	50	100
Time (s)	5.837	43.719	286.44	569.67
Number of Scan Queries	1	10	50	
Time (s)	12.304	149.42	725.1	

Figure 13: Point and Range Queries for Tiering

Number of Reads (QUERIES)	1	10	50	100
Time	4.281	32.2	283.45	540.29
Number of Scan Queries	1	10	50	
Time (s)	14.487	119.83	628.51	

Figure 14: Point and Range Queries for Leveling

Full Queries (Full Scan). Full queries do not attempt to analyze any information within an entry and, instead, blindly returns all entries present in the database. This idea makes full scan comparatively fast as it is performed without any intermediate arithmetic or bitwise-logic calculations. Hence, in exponential scale, time grows linearly according to the increase in file size, in the same manner for tiering as it is for leveling.

4.3 Delete Performance

Both point-deletes and range-deletes are simply point-queries and range-queries with an added process of modifying the tombstone bit (*visible* variable) from 1 to 0, which is always of a constant, minimal duration. Therefore, the empirical trend of time taken for processing of deletes follows the same trajectory as that of reads.

Number of Delete(QUERIES)	100
Time	1381.053
Number of Mix (QUERIES)	100
Time	423.69

Figure 15: Delete and Mix Queries for Tiering

Number of Deletes (QUERIES)	100
Time	1201.691
Number of Mix (QUERIES)	100
Time	205.639

Figure 16: Delete and Mix Queries for Leveling

5 CONCLUSION

As such, while there are many obstacles and sacrifices we've made along the way, we suppose that our attempt at implementing an LSM-Tree as an actual database concludes with a satisfactory success. Most of what we've implemented ended up working in ways we expected them to work; there are only some regions where we couldn't quite tune it to work in such ways and that, we profess, are our shortcomings. Overall, there are still many avenues which are open to improvement, one of which we notice is the algorithms by which to perform bulk processes (bulk-inserts, bulk-queries, and bulk-deletes) as prioritizing some facet of data different from the order with which we implemented may yield better performance for some particular workload.

Aside from the performance our implementation proves, we have learnt a great deal of in this project. Experience-wise, we have learnt about the intricacies of the LSM-Tree in ways we couldn't have possibly fathom by simply reading related papers and discussing them in class. This project has been an important learning experience on database systems for us and we will be able to approach our programming career with prudence in respect to the pitfalls and benefits we've learnt so far. We also came to understand that not all logical designs can be translated into a code implementation with ease; we have drawn the logical designs out and talked each other through the steps in code in which to traverse but the act of coding the designs out still prove challenging for us, with many hours spent debugging and tweaking code until the behavior complies with our expectations. In addition, we realized the importance of communicating with the client who, in this case, are the professor and the teaching assistants. There are aspects of the design that we were unsure in which way to proceed and the solution to that is simply meeting up with the client to discuss the minute details to ensure that the design we're proceeding with meets with the client's objective. We also believe that this project serves as a great stepping-stone into the world of database systems as, with it, we've learnt the database fundamentals required for careers in this field.

In all, this project has been an inspiration to all of us to pursue further endeavors in database systems and we sincerely believe that our work reflects that.

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