Implementing the LSM tree in C++

John C. Merfeld Stathis Karatsiolis

April 29th, 2019

Code can be found at https://github.com/johncmerfeld/LSMTree

1 Introduction

For our final project, we set out to implement an LSM tree in C++. An LSM tree is a data structure supporting the creation of indexed key-value stores expecting a high volume of writes, such as a transaction log. Clearly, we may need to examine the log at a moment?s notice, and so it needs an index in order to support efficient reads. Ordinarily, though, keeping an index over a key-value store in real-time is problematic due to the extra disk I/O incurred while working to maintain it. Every transaction incurs the overhead of modifying the structure of the log?s index.

The LSM tree?s key solution to this problem (which we will discuss in more detail below) is to hold a collection of such updates in main memory for as long as possible. Only when memory becomes full of updates is that ?run? written to the disk. Thus, the cost of updating the tree is amortized across many updates, but a logarithmic structure is still maintained for reads.

After reading multiple papers about LSM trees and how they can be tuned, we have three design goals for our implementation: supporting a high volume of real time updates, maintaining reasonable read performance, and utilizing memory-resident metadata to improve lookup speed. C++ was a natural choice for this implementation because of its close relationship to the hardware and low overhead. In building an LSM tree, we hope to gain insight into why they offer the benefits they do, as well as a first-hand look at how different parameters affect the performance of a real data system.

2 Background

Our work is of course heavily inspired by the original 1996 paper on log-structured merge trees [1]. This paper introduces the fundamental idea of a memory-resident structure merging batched updates into a disk-resident tree directory, and a general design heuristic of avoiding random-access IO. Other important considerations from this initial work are the relative

irrelevance of the memory-resident data?s exact structure, and the difficulties incurred by searching the entire depth of the tree, especially for range queries.

To incorporate some optimizations on reading the disk tree, we also took inspiration from the 2017 Monkey paper [2]. Although this paper puts heavy emphasis on flexibility (namely, how the merge policy at different levels of the disk tree can be tuned to a given workload, and how memory-resident lookup structures can have different semantic guarantees at different layers), we were more modest in what we used in our implementation. In particular, we attempted to make the most out of two pieces of memory-resident metadata ? bloom filters and fence pointers ? to enable a degree of data skipping, especially during point queries. Although we do not claim to contribute novel extensions of these methods in our implementation, we believe the design discussed below offers an easy-to-understand baseline onto which further optimizations could be added in order to meet our stated goals. At present, we are working towards a full implementation of the Monkey paper?s ?tiered? vs. ?leveled? versions of the LSM tree, but the reader should know that our disk tree structure as of now uses a tiered scheme, meaning multiple un-merged runs exist at each level.

3 Design

Our LSM tree implementation stores pairs of key-value integers. By this we mean that both the keys and the values are of type integer. Although this choice was made for simplicity, it would not be difficult to further abstract the key-value pairs and enable them to store alternate data types, at least as values. Additionally, each key can only correspond to a single value. Again, this is a design choice made for simplicity; other implementations of LSM trees take different approaches to handling one-to-many mappings of their keys (CITE). A key-value pair forms an object of type Entry. An entry also contains a boolean variable that will indicate if this entry is a deletion. We explain below how this boolean is used to handle deletes in the LSM tree.

Although the idea of a many-level tree is a useful abstraction, there is only one real division in our implementation: that between data resident in memory and data resident on disk. In memory, some number of entries are stored in an array class called a memory run. The disk data consists of groups of entries stored in distinct data files called DiskRuns. The memory-resident LSM Tree a 2-dimensional list of pointers to these DiskRuns. The order in which they are traversed, and the progressively increasing file sizes (with decreasing metadata granularity) simulate the file-directory geometry discussed in the original LSM Tree paper.

Of course, the highest-order question is how our LSM tree handles writes. Before walking through the flow of data, though, we must outline the metadata that the LSM tree maintains to ensure efficient reads. Metadata is stored in memory and recorded for each run.

Each RunMetadata object contains 4 pieces of data. The first is a bloom filter, a deterministic hashing structure that can indicate in constant time whether a value is present in the run. The second is a set of fence pointers, structures which hold the highest and lowest values stored in the run, enabling efficient data skipping. Finally, the number of entries in

the run and the name of the disk file storing it are included as well.

To understand how our design addresses the goals outlined above, we will now walk through the LSM tree?s key external functions and how they are implemented. The most important method, of course, is insert(). When an entry is inserted, it is immediately added to the next position in the memory run Thus, the cost of most insertions is only that of appending to an array. When the memory run reaches capacity, the process of writing it to disk kicks off. The run is sorted, its Entries inserted into a bloom filter, and its highest- and lowest-value keys recorded in a fence pointer. The bloom filter and fence pointer are added to a new RunMetadata object, which also receives a fresh filename. The array of entries is written to a new file, and the run?s counter is reset to 0. Again, we see how this design contributes to the goal of efficient writes, because the array can be written on a continuous sequence of disk pages instead of a separate access for each one.

Thankfully, the way that Entries are created and merged means that deletions and updates have the same functional interface as insertions. The LsmTree class has not update() method. To update the value of a key, it can simply be inserted again. And although there is a method to remove keys from an LSM tree, under the hood, this simply inserts an entry into the memory run with its deletion variable set to True. When runs are merged to make new disk levels, these updates and deletes take the place of older inserts of the same key.

Finally, the get() and getRange() functions search the LSM tree for points or ranges in the data, utilizing the metadata to avoid reading from the disk as much as possible. First, get() scans the run in memory in case the query value was inserted recently. If that key?s entry cannot be found in memory, the metadata object for each of the disk levels is scanned in turn. If its bloom filter indicates that the key might be present in the run and its fence pointer does not deem it out of range, then the run is loaded from disk into memory and scanned for the key. We are especially aided by the LSM tree?s temporal partitioning here: if the first entry we find with the search key is a delete, we can cease our search immediately. The getRange() method follows a similar logic, except a memory run of results is maintained throughout the search and merged with the matching keys of all disk runs whose fence pointers overlap with the query range.

Thus, the LSM tree is clearly not optimized for reads alone. There is no global index mapping out where certain entries can be found in the data. Instead, small local structures are maintained to make reads as fast as possible while still assuming that the default workload will consist mostly of inserting, updating, and deleting. The further ?down? the tree an entry travels, the less frequently it is read from the disk for continued sort-merging. Again, this is as it should be. The data entries most likely to be updated in real applications are those most recently inserted, so it is appropriate to make the access path to reading them as short as possible [3].

4 Evaluation

Our experimental evaluation of the tree is ongoing, but we will discuss our preliminary results. Our primary workload was a sequence of one million writes and 50,000 reads. 100,000

of the inserts were updates, and another 100,000 were deletes. We chose this workload because it utilizes multiple facets of the LSM tree?s API while keeping the emphasis on our primary design goal of write optimization.

The test script itself reads a CSV file with workload instructions into an array. It then starts a timer, reads through the array and performs every operation in it, then stops the timer. Separate timers are used for reading and writing. Whenever possible, the average result from multiple runs is used in the below figures. The upshot of our experimental

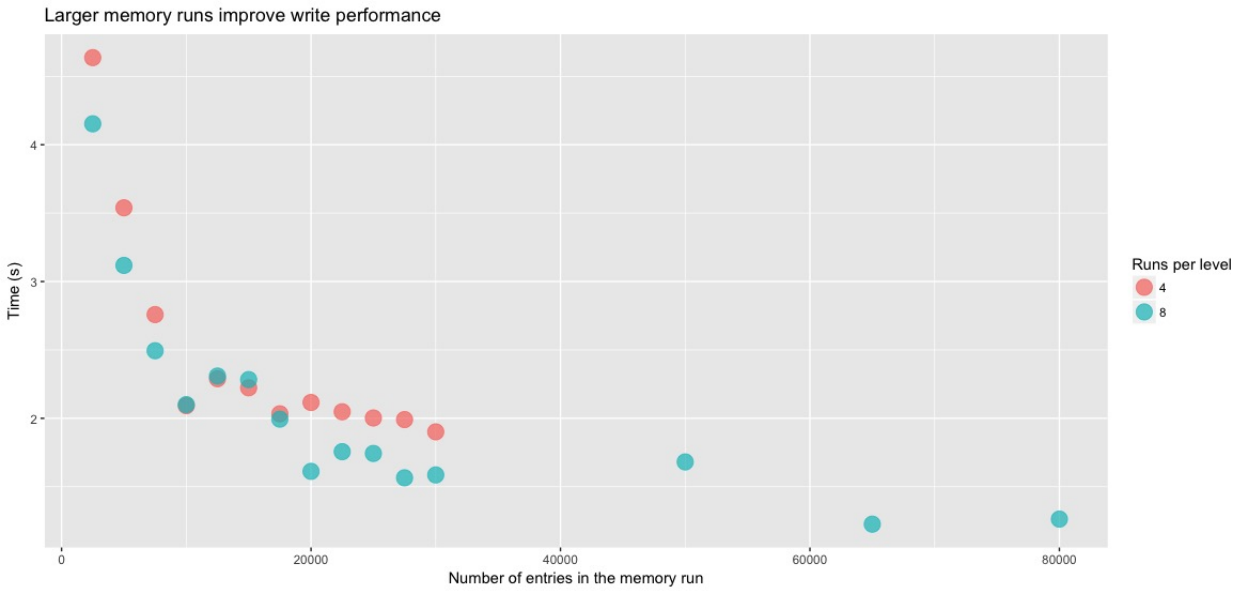

Figure 1: Write performance for different memory run and level sizes

results is shown in Figure 1, a clear link between memory run size and write performance. In general, it also appears that having more runs per disk tree level improves performance.

Furthermore, the tree clearly performs better on writes than reads. Notice how a good-sized memory run can handle a million inserts in under a second. Reads are a much slower affair, as Figure 2 indicates. This is due to the frequent need for multiple disk I/Os for a read, compared to the memory-only nature of most writes.

5 Conclusion

For this project, we set out a simple design goal: build a data system in C++ inspired by the LSM tree that supports real time updates without completely sacrificing read efficiency. The result was a structure that seeks to balance large, batched disk merges with a sufficiently descriptive set of memory-resident metadata. We use a simple array of key-value pairs to hold insertions, updates, and deletes in memory for as long as possible. Additionally, auxiliary lists of bloom filters and fence pointers enable significant data skipping to reduce lookup

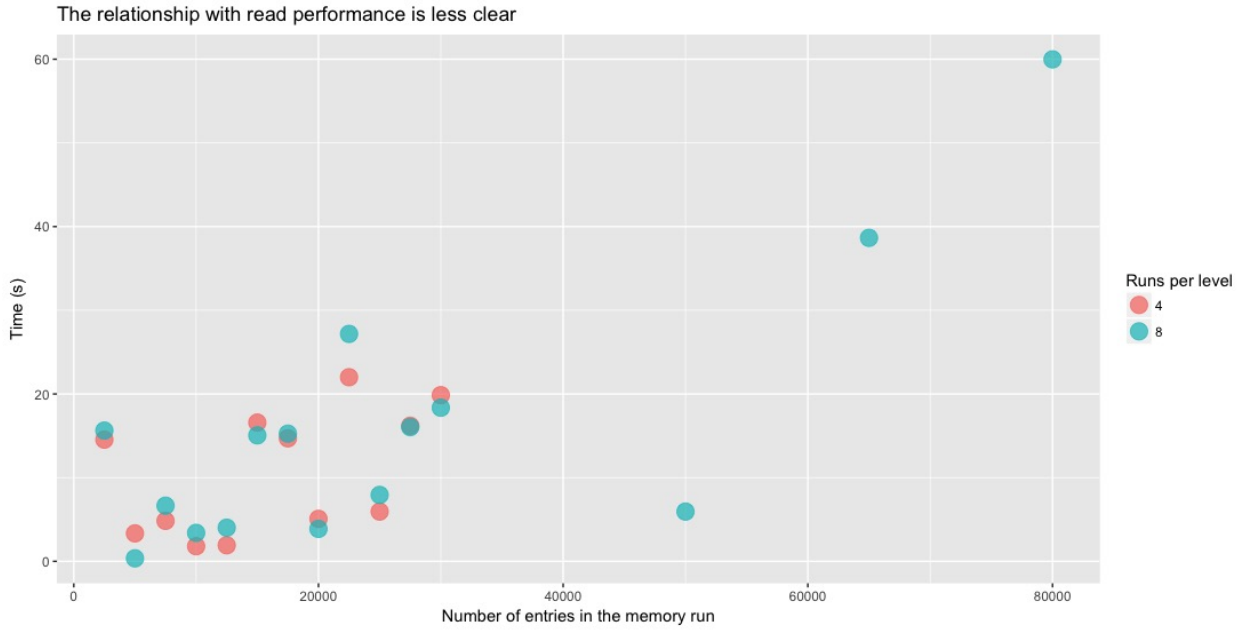

Figure 2: It is difficult to tease out any strong correlations in these data. Optimizing read performance within our structure, perhaps with a leveled LSM tree, will be a priority going forward

costs. Our experimental evaluation found a correlation between the size of the memory run and the write performance of the tree.

Our future work will include gathering more rigorous experimental results and fully implementing a leveled variation of the tree. We also hope to introduce of parallelism into the system, possibly by batching updates, going trough the batch, and assigning tasks to different threads.

Both group members took part in the design of system. Mr. Karatsiolis built the disk-based backend of the LSM tree, got the system through the debugging process, and ran the experiments. Mr. Merfeld built the tree API and most of the memory-resident data structures, including operations performed on runs, and took the lead on the report and presentation.

Sources

[1] P. E. O?Neil, E. Cheng, D. Gawlick, and E. J. O?Neil. The log-structured merge-tree (LSM-tree). Acta Informatica, 33(4):351?385, 1996.

[2] Niv Dayan , Manos Athanassoulis , Stratos Idreos, Monkey: Optimal Navigable Key-Value Store, Proceedings of the 2017 ACM International Conference on Management of Data, May 14-19, 2017, Chicago, Illinois, USA

[3] D. Beaver, S. Kumar, H. C. Li, J. Sobel, P. Vajgel, and F. Inc. Finding ?a needle in

haystack: Facebook?s photo storage. In OSDI, 2010.