CS591A1 : Adaptive Range Scan LSMTree

Chengjun Wu
Boston University

Athanasios Filippidis
Boston University

ABSTRACT

In this short paper, we explore the design space of key-value stores that utilize the LSM-tree (log-structured merge-tree) structure. More specifically, we implemented an LSM-tree strongly based in the design conventions that are known and explored in previous work in order to get a better understanding of the basic knowledge that already exists about LSM-trees. Then, we decided to focus on an area that we found to be interesting and not-so-well-explored; the range-scan aspect of an LSM-tree. We show a heuristic approach we came up with that has as a goal to minimize the storage I/Os during a range scan. Moreover, we expose to a user plenty of tunable knobs which allow our implementation to be flexible and adjustable in different workloads and types of usages of our LSM-tree.

1 INTRODUCTION

1.1 Motivation

The main motivation behind this project is twofold. The first part of it is the motivation behind the LSM-tree itself. That is the performance advantages which such a data structure provides to indexed write-heavy data/workloads. We wanted to be able to understand how a system is benefited from that structure and to understand what are the key components of that structure that enable those advantages in performance. The best way to achieve that is by actually building such a data structure/system from scratch. The second thing that motivated us was the nature of LSM-trees. As mentioned before, usually, they are used for OLTP workloads. However, nowadays, more and more applications require some kind of analytical queries to their data as well. Instead of creating a secondary storage that would be optimized for such workloads we wanted to explore whether we can actually enhance the performance of our implementation in the domain of OLAP workloads as well.

1.2 Problem Statement

The main problem that we had to overcome regarding the OLAP performance enhancement was to determine which part of an OLAP query is the most time-consuming and how to optimize that part. We decided that this part is the range scan and more specifically the storage I/Os that are performed during one. OLAP queries usually are some kind of aggregation queries that access a significant portion of data. The common functionality behind every aggregation query is that it involves range scanning. Thus the main problem that we had to face was how to minimize the storage I/Os during range scans. This problem has a lower bound - we know that we cannot do better than accessing every single tuple that exists in the requested range. However, what we needed to do is to avoid accessing as many as possible redundant tuples that either do not exist in that range or have stale values.

1.3 Contributions

Our main contributions are two. Firstly, we provide an implementation of an LSM-tree that due to our design is highly flexible and tunable. We provide plenty of knobs that enable us to change the "shape" of our tree based on the requirements of the expected workload. Secondly, we present a heuristic algorithm based on which we are able to utilize every piece of available metadata in order to minimize the storage I/Os during range scans.

2 BACKGROUND

The original presentation of a Log-Structured Merge-tree[1] was suggesting a disk-based data structure that provides low-cost indexing for files that are expected to experience a high-volume of write-transactions. However, later implementations[2][3][4][5] have explored the design space of this structure towards many different directions. Some of them have explored paths that lead to better performance for read-heavy workloads, some others have explored the trade-offs between write-optimized, read-optimized and memory efficient storages, others have explored the effect of the two main metadata structures (Bloom Filters and Fence Pointers) of an LSM-tree and how those structures can be optimally configured and of course there are a lot of products that are currently being used in the databases industry that are strongly based on the LSM-tree structure and are adjusted to meet the needs of the respective company/product.

3 ARCHITECTURE

At this point we will start analyzing the main components of our architecture. We will also mention the modifications that we have made to various standard LSM-tree components and we will reason behind them.

3.1 caDB

Our implementation follows the standard LSM-tree implementation in many points. First of all, we used C++ for efficiency with stl in order to not waste time in the creation of the simple structures that we use. We use a simple in-memory buffer that receives the operations that a client application sends. We expose a minimal API to our clients which includes the basic operations that are expected to exist in a key-value store. Those are get, put, scan and delete. We have created a simple wrapper around our tree which makes it a database system and we call it caDB. Whenever the buffer of the tree gets full, it is dumped to the first level of our tree. Our levels consist of multiple runs. Each level has a fixed size ratio compared to the previous and the next one. The default value for this is two. For deletions, we utilize the "tombstones" idea that has been suggested in previous work. What this means is that when the client wants to delete a tuple, she just inserts a new tuple with the same key that she wants to delete and with a specific value that the system knows internally it designates a deletion of a value.

Following are some of the pieces of the standard implementation that we have modified or we believe that are core components of it and should be discussed.

3.2 Duplicates removal

One of the main issues that an LSM-tree has to overcome is the duplicate keys. There are several reasons behind the existence of them. Firstly, the client might wanted to update the value of the tuple. In order to do so, she will insert the new value with the same key. This, will lead to the tree having for some time both the fresh and the stale value. Then, as described above, we may have again a similar situation with deletes. We tackle this issue by removing the duplicates during the sorting/merging procedure. At some point, the runs of every level get merged between them when the level reaches some capacity threshold. The merging procedure procedure is responsible in our implementation for merging multiple runs and creating a merged output without any duplicates by always preserving the latest value of every key.

3.3 Bloom Filters

The bloom filters are an essential component of every LSM-tree. They are very useful since they are able with minimal computational and memory overhead to answer whether a key is part of a level or a run. Their main drawback is their false positive probability. While when a bloom filter gives a negative answer we are able to know for sure that this is true, when it gives a positive answer there is a chance that this is false. We provide two tunable knobs that can modify that functionality. Firstly, we provide the option of determining how many bloom filters will exist in every SST file. The SST files are simply binary files that store the data of the levels of the tree in storage. Secondly, we provide a knob that tunes the precision of the filter by adding or removing bits to it. In simple words, the more bits a bloom filter utilizes to represent the existence of a tuple the less likely it is to have false positives.

3.4 Fence Pointers

The fence pointers are again a structure that is commonly used in LSM-trees. In simple words, they are the limits of some virtual blocks of data. Utilizing them, we are able to skip blocks that are not "interesting" to a query and minimize the tuples that will be accessed during its execution. We have implemented our fence pointers in an abstract way. This decision was made because we realized that we may use them in multiple different "levels" in our design. The first spot where we can utilize fence pointers are the SST files. For example, by keeping a fence pointer at the start and one at the end of the file we are able to easily determine whether there is any chance that the tuple we are looking for might actually be in that file. This way we can skip fetching unnecessary files from the storage to memory. Then, in case that our SST files are larger than an operating system "page" we can use fence pointers inside those files as well. The benefit from that would be to avoid fetching the whole file in memory and, instead, to fetch the minimum necessary data.

3.5 Range Scan

As we discussed before, in order to improve the OLAP performance of our implementation, we decided to have an as efficient as possible range scan. The main way to do that, without creating any additional overheads, is by utilizing as much as possible the available metadata that we have. This metadata is available from three different sources. The first one is the results of the scan itself. When a new range scan is initiated we create a bit vector that has a length equal to the maximum length of the result of the range scan. Then, as we are scanning through our tree in order to locate the keys that are part of the requested range, every time we find one of them, we update our bit vector and set this key as found. This way, we keep track of the keys we have already found and we dynamically update our range, every time narrowing it down as much as possible. The second source from which we utilize metadata is the fence pointers. The intuition here is that we use the fence pointers in order to again avoid fetching irrelevant data. So after getting the narrowed down range from the bit vector we use the fence pointers to exclude parts of the range that was requested if we can tell for sure from the fence pointers that those keys are out of the range of the specific file/run. Then, the last source of metadata is the bloom filters. At last, after narrowing down the range even more, we use the available bloom filters. At this point, we just ask the respective bloom filter for each of the remaining values keeping track of the keys we get negative answers for. If those keys are at the start or at the end of our range we exclude them as well and we narrow down the range even more. As someone may imagine, this process may lead to a significant pruning of the range. However, this is strongly correlated to the specific query and the distribution of the data.

3.6 Testing

The implemented functionality was tested in order to ensure correctness. More specifically, we have implemented some simple unit tests that we were using through the development phase during every integration in order to ensure that none of the modifications we were making was a breaking one. After the end of the development phase, we firstly created a tunable workload generator in Python and we then utilized the provided workload generator with the Google test functionality. Both of them, as they create the workload they also keep a log of the operations that are being performed which is then used to verify the correctness of the results.

3.7 Benchmark Explanations

At this point it is useful to mention the system on which our implementation was tested. We used a Linux machine running Ubuntu 18.04.3 LTS in a 4-cores Intel(R) Core(TM) i5-6400 CPU @ 2.70GHz with 8GB RAM and an HDD. Unfortunately, our results were usually counter-intuitive and inconsistent. We believe that our results might have been more reasonable if we knew beforehand about the existence of the Direct I/O feature.

4 RESULTS

Following are some graphs that provide some insights regarding the performance of our implementation. We should mention that for a more accurate benchmarking we should have probably used a dedicated machine that would have no other background tasks running and affecting our measurements.

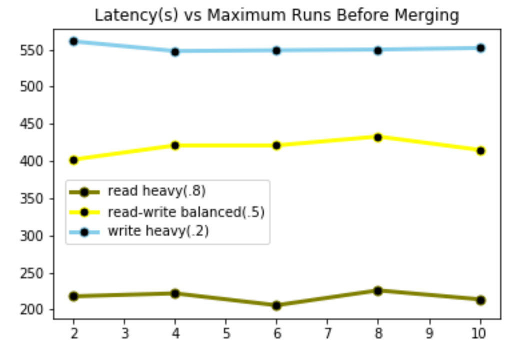

Figure 1: In this diagram, the reader may observe the aggregate latency of our system under three different workloads and how this is affected by the merging policy. The y-axis represents the latency and the x-axis the number of runs we had in each level before merging. The closer we are on the y-axis the closer we are to a Leveling merge policy. On the other hand, the further we are from the y-axis the closer we get from a lazy Tiering to a Tiering policy. The workload for this graph was 500,000 operations. In the read-heavy version 80% of them were reads and 20% writes, in the balanced version they were divided as 50-50% and in the write heavy the reads were the 20% of the workload.

As mentioned earlier, considering that the LSM-tree was originally designed to perform better in write-heavy workloads, this diagram is quite counter-intuitive. We believe that this result may be due to two facts. The first one is that we did not have the time to do all the optimizations we had in mind. The second one is that we did not have the time to explore the complete design space and capabilities of our implementation in order to end up using the optimal configurations for our benchmark. Unfortunately, due to the various different knobs that we have implemented we spent too much time debugging the various corner cases and, at some point, we had to decide whether we will make sure that our system provides formally correctness under any configuration or we will perform extensive benchmarking on it. We believe that creating a credible system is more important since the benchmarking process can be performed in later time as well, thus we invested our time in ensuring that our system performs always correctly.

5 CONCLUSION

This was an extremely interesting project for both of us. We practiced on the design process of such a system and then each one of us invested a lot of time in learning the best programming practices from a performance perspective in C++. Moreover, we both studied papers and discussed multiple ideas that we could integrate in our implementation. Finally, we ended up creating a system that we are really proud from a correctness perspective considering its complexity. As future work, we would love to explore more the bottlenecks of our system and come up with ideas on how to overcome them. A first step towards this direction could be the usage of a skip-list for our in-memory buffer.

REFERENCES

[1] Patrick O'Neil et al, The Log-Structured Merge-Tree (LSM-Tree), https://www.cs.umb.edu/~poneil/lsmtree.pdf
[2] Facebook Open Source, https://rocksdb.org/
[3] Niv Dayan, Stratos Idreos, The Log-Structured Merge-Bush and the Wacky Continuum, https://dl.acm.org/doi/pdf/10.1145/3299869.3319903
[4] Niv Dayan, Stratos Idreos, Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging, https://dl.acm.org/doi/pdf/10.1145/3183713.3196927
[5] Niv Dayan, Manos Athanassoulis, Stratos Idreos, Monkey: Optimal Navigable Key-Value Store, https://dl.acm.org/doi/pdf/10.1145/3035918.3064054