**Final Report**

**Project: Implementation of LSM-Tree Based Key-Value Store**

Team: Jiangshan Luo, Ruidong Duan

1. **Introduction**

Our motivation is to gain a better understanding of how an LSM-tree store works and the specific methodology underneath it, and, at the same time, to improve our programming skill. To achieve that, we implemented a basic LSM-tree key-value store that uses leveling, which is coded in C++. By doing so, we deeply understand lots of details in LSM-tree, its workflow, as well as some coding issues such as the implementation of a bloom filter, I/O, and data system configuration.

1. **Background**

Our system design is inspired by the papers of the original LSM-tree store[ The Log-Structured Merge-Tree (LSM-Tree)] and Monkey[ Monkey: Optimal Navigable Key-Value Store], an optimized LSM-based key-value store. Specifically, the major techniques we use in our experiment are bloom filter, vector, hash function, and file I/O.

The core technique of LSM-tree is to take advantage of continuous storage space to lower the system reading latency. One of the key features it has is the memory buffer. When an entry is to be inserted, the LSM-tree firstly pushes it into the buffer, and flush it into disk only after the buffer is full. In disk(s), as data grows larger, we merge and sort them into bigger runs, so that we can use binary search to speed up lookups. Moreover, bloom filters are used to avoid wasting time in finding non-exist entries, where some hash functions are needed. We use vectors to store buffer, bloom filters, and to move data blocks.

1. **Design**

**Architecture Design**

The system we implemented uses only “leveling” for the better reading performance. It includes basic components of an LSM-tree: a buffer, Bloom filters and fence pointers in main memory, and runs (data blocks) in secondary storage. In terms of the system design, there are mainly four classes which contain the needed data structures and APIs: buffer, BloomFilter, LSM, and DiskRun. The visualization of our design refers to Figure 1.

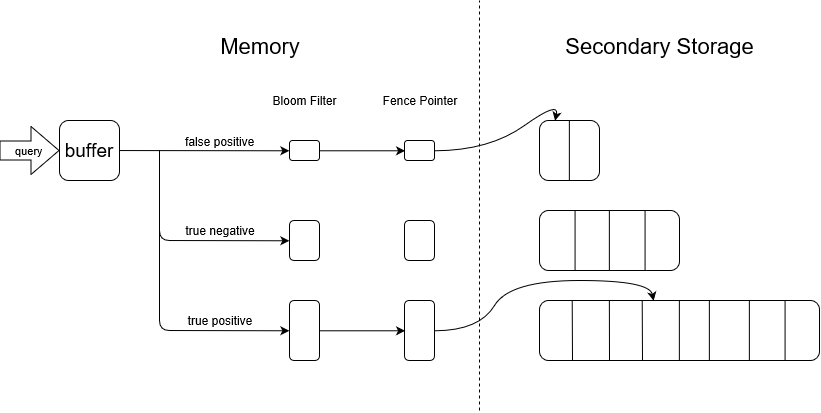


Figure 1. System Design

**API**

|  |  |
| --- | --- |
| **Classes** | |
| class **LSM**{  Buffer buffer[];  DiskRun\* diskrun[];  BloomFilter\* filter[];  LSM(int, int, int, int, float);  void insert(K, V);  vector range(K, K);  void delete\_key(K);  Pair lookup(K);  }  class **buffer**{  vector buffer;  buffer(int);  void insert(Pair);  vector range(K, K);  Pair\* lookup(K);  void push ();  } | class **bloom\_filter**{  vector filter;  BloomFilter(int, double);  void addkey(K);  bool contain(K);  void clear();  }  class **DiskRun**{  string path;  K fencepointer[];  Pair\* lookup(K);  vector rangeSearch(K, K);  void merge(vector);  }  Struct Pair{  K key;  V value;  } |

1. **Evaluation**

**Experimental Setup**

For experimentation, we use a machine with a 1TB SATA HDD, 8 GB DDR4 main memory, and six 2.20 GHz cores with 9 MB L3 Cache. The operating system we use is Windows 10 Home Edition where the experiment is run on an NTFS partition.

Default setting: the size of an entry is 8B, the size ratio is 2, the page size is 4KB, and the false positive rate is 0.1.

**Experimental Workloads**

There are several experiments that we use to evaluate our system.

* We test the system’s capability with increasing data volume. To achieve that, we run the test multiple times with the same (default) system configuration, each time using more data entries.
* And also, we want to know how the entry size of our system could affect the overall performance. Using a similar set up as the first experiment, we keep the data volume constant this time, instead, we gradually increase the entry size for each test and evaluate the performance changes.
* Thirdly, we try to find out how the system handles the tradeoff between update and lookup, by changing the ratio size from 2 to 16.
* The fourth aspect is to test throughput performance under different workloads. In this case, we will use various workloads with different lookups/updates ratios.

Figure 2 Figure 3

Figure 4 Figure 5

From figure 2 and 3, we can see that as the number of entries and entry size grows, the lookup latency does not obviously increase, which is probably caused by the existence of the bloom filters and the fence pointers. From figure 4, it is shown that by changing ratio size, we can do some tradeoffs between lookup and update latency. In figure 5 shows that the ratio of lookups presented in workload slightly influences the system throughput where the growth of lookups will decrease the overall query performance of the system. As a matter of fact, these results of the evaluation well fits in with our expectation.

1. **Conclusion**

We implemented an LSM-tree key-value store which can do basic operations update, delete, lookup, and range search. During the experiment, we research and build bloom filter, buffer, and runs in disks. From the evaluation, we show that our system works properly and has a decent performance. In conclusion, we achieved our goal, to research and build a functional LSM-tree store, and also to improve our programming skill.

1. **Changes after initial submission**

First, we would like to have some clarifications for our design since there were several questions raised during the presentation. In terms of reading the disk runs, our program first checks the bloom filters (point lookup), and if finds the existence, it will search through the fence pointers of this specific level which is stored in the memory. After we find which pages should the data belong, we allow the program to access the data file and directly jump to the pages we need in the system. More specifically, this will be done by I/O stream scanning and loading pair by pair, instead of loading the entire file in to memory.

Secondly, we also added the additional experiment results for our LSM-tree, and includes some of our thought for them.

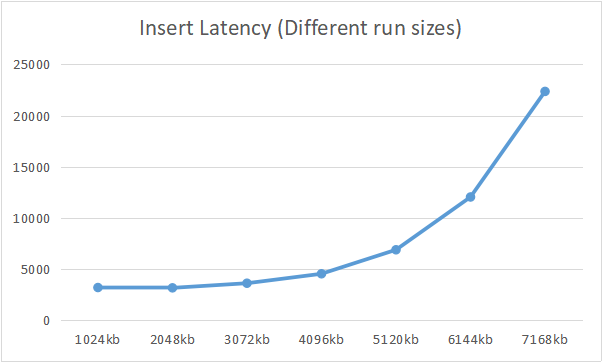
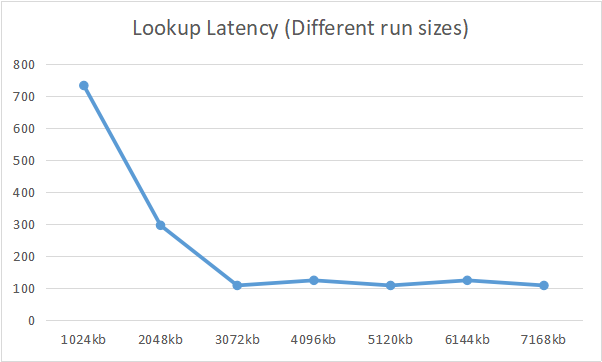
 

Figure 6 Figure 7

One experiment that we had is on how the size of the run, or numbers on element in a run, could possibly influence the system performance. In the experiment, we used run size ranging from 1024kb to 7168kb and evaluation the update/lookup latency, which is shown in Figure 6 and 7. Similar to Figure 4, where we test the size ratio, there is a tradeoff between update/lookup while we shifting the run size, as more and more time is needed to merge and sort the run.

While the other experiment we did is to test the effectiviness of our Bloom Filters. However, there are new questions remained when we adjust the false positive rate as our professor suggested. Figure 10 shows the original result we have for this part, where we gradually increased the FP rate from 0.1 to 1. The result seems reasonable since the lookup latency increases and the insert latency declines, which we have discussed during the presentation.

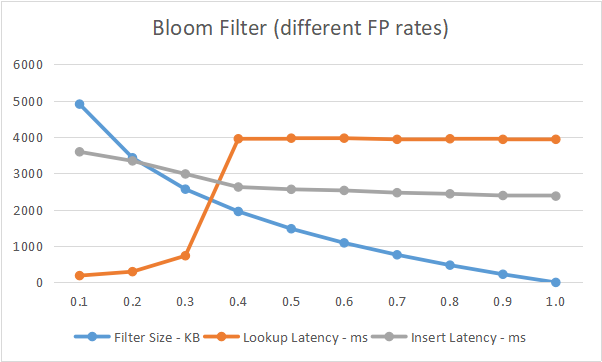


Figure 8 Figure 9

Figure 10 Figure 11

However, we met some issues while doing the additional testing where we set the FP rate from 0.001 to 0.01 (0.1% - 1%). As we can see in Figure 9, 10 and 11, even though the size of the filters and the insert latency are decreasing as before, we never get the increasing lookup latency anymore. Instead, the lookup latency slightly drops down during the experiment, and then increases when the FP rate is beyond 0.1. Our theory is that the size of filters are so large, when we have low FP rates, that it costs more time for checking an element in them (extra memory, extra hash times). And when increased the FP rate from 0.001 to 0.1, we actually eased the function cost and save time.

For the Bloom Filter, unfortunately, we are using the representation of vector<boolean> which is eight times larger than an actual array of bits. Thanks for the presentation feedback that now we know the protential drawback of using the vectors of boolean. If time permitted, this would be one part of the optimization we shall try.