CS561: Implementation a LSM-Tree based Key-value Store

Yinan An U59482444
Minghui Yang U49668926
Shun Yao U0503877

Abstract

LSM-tree(Log-structured Merge-Tree) is the underlying data engine commonly used for NoSQL databases. This kind of database can achieve high speed in the scenario of inserting data. It is a kind of data structure for disk, making full use of the order of the disk write performance is much higher than that of random write performance of the properties, the bulk of random write into a one-time order. This report has four parts to demonstrate our work. First part is architecture, which shows the overall structure of the DB and what data structures it contains. Second part introduces all the functions and their logic in detail. Third part is an experiment which contains function testing and performance testing. The last part is the conclusion.

1 Introduction

1.1 Motivation

Most of the NoSQL databases use a variant of LSM to achieve fast speed of inserting. In this report, we implement its key functions and inner data structure to explore why it is so fast. We explore and use different strategies of LSM-tree to make it time and space efficient. To master the implementation of the underlying engine of NoSQL databases can help us better understand its appropriate use scenarios and locate issues when dealing with them in the future.

1.2 Problem Statement and Main Work

Our proposal is to achieve initialization of the database and support read, write, update delete operations. Plus, it should support two compactions strategies to adapt to different scenarios.

Our main works are listed as follows:

1. Implemented a complete LSM-tree based key value store that could support get, put, scan, point/range delete operations, and realized Leveling and Tiering merge policies.
2. The database is durable, and could support multiple databases.

3. To optimize the space and querying time, data files are saved in binary format and are divided into blocks.

4. Use bloom filter, fence pointer and also zones to accelerate the querying process.

5. Do different experiments to evaluate the database performance and merging policies.

2 Architecture

2.1 Overview

The main architecture of our LSM-Tree based key-value store contains one MMTable and several SSTables(Runs). The architecture of a Leveling type database is shown as Figure 1. In memory, we store the MMTable and the metadata(bloom filter, fence pointer and zones) for each SSTable And in disk, we store data files. One SSTable data file is divided into several blocks, and in zones, we have pointers to point to these different file blocks. One SStable belongs to a specific Layer and one Layer may contain several SSTables.

When doing Put operation, we first insert key-value pairs in MMTable, and when the size of MMTable extends a threshold, we will flush all the key-value pairs from MMTable, form a SSTable and store it in the disk. When we add new data into a Layer and the size of this Layer is over threshold, then we will do compaction to compact data with the next Layer.

When doing query operations, we will first search from MMTable, if not the data is not found, then we will check bloom filter, fence pointer and zones to decide if we need to read file blocks and which block we need to read from.

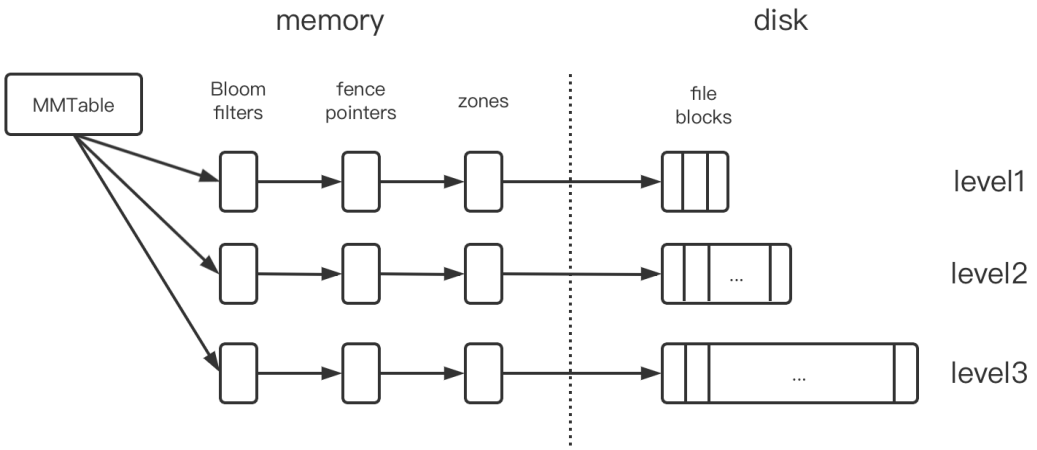

Figure 1

2.2 Data Structure

1. MMTable

We use the map in STL library as the MMTable, it is a red-black tree internally, so it will keep all the key value pairs in key order. We encapsulate the existing add, delete, modify, and query functions in map to support customs data structure.

2. Run(SSTable)

One Run in our implementation is a directory, which contains a metadata file that stores basic information, and a binary-format data file that stores all the key-value pairs. A logical view of SSTable is like Figure 2.

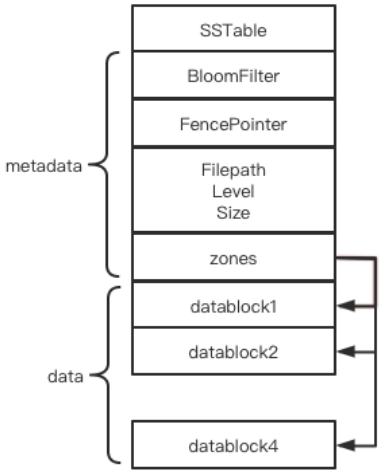

Figure 2

Zone is a data structure that is used to record the min/max key and min/max byte of each block in a data file. When reading an SSTable, we will first read the zones and use the byte number to directly point to the data block that the key value pair may store. It will help to improve the query performance.

Every Run has methods range_query, query, read_disk to give the API support for the database operations.

3. Level & Levels

The design of the Level and Levels is shown in Figure 1. One level contains one(Leveling) or several(Tiering) Runs that are stored. When opening the database, we initialize the database by loading the metadata to the memory, and keep data files in the disk. There are some methods to operate the runs in the level, like addARun, removeARun, getARun, etc. Every level has a threshold value that indicates the

max amount of data that this level can store. The threshold for different levels is exponentially increasing, which means that if the first level threshold is t, then the Nth level threshold is t^N.

Levels is the core component of the database, it consists of several Level objects that form a vector. And there are some methods to operate on the different Levels, like the get and set method, shown as Figure 3.

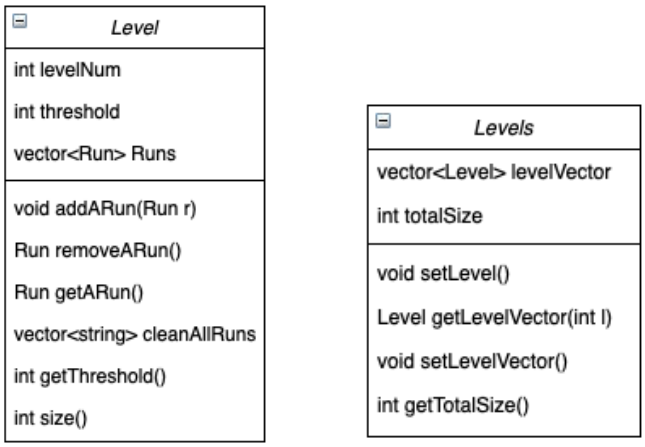

Figure 3

4. Delete Table

Delete table contains a vector that stores all the range delete records. A range delete record contains the min/max key of range delete and the timestamp of the range delete.

3 Implementation

3.1 DB Initialization & Persistence

1. Initialization: When first opening a database, we need to first initialize, which means that we need to create a database structure in memory. The steps are:

1) If the database doesn't exist, then create a new config file with all variables set as default values.

2) Read the config file and construct the Levels structure and initialize all levels. The Run vector at this time is null and needs to be updated in the later steps.

3) Read all the metadata files of all Runs in the data directory and initialize the Metadata of every Run. And then create Runs objects using the metadata and store them in their corresponding level.

4) Read the delete table from the file

2. Persistence: The DB persistence happens in the following circumstance.

1) When flushing the data from MMTable to SSTable: At this time, we need to first create a Metadata object based on the data map, and then partition the data into data blocks and write data blocks

information into zones of that metadata. And then create a Run object based on data and metadata, and store it in files.

2) When closing the database: At this time, we first also need to flush the data from the MMTable and create a Run object and file, like what describes above. Then we need to persist the delete table and update the config file.

3.2 DB Operations

1. Put

1) Put the key value pairs in MMTable. If after insertion, the size of MMTable reaches the threshold, then do the following steps.

2) Flush the data out of MMTable. By doing the DB persistence steps in the above section to store the SSTable in the first level.

3) Do the compaction operation, which will be described later in the fifth part of this section.

2. Get

1) Search if the data is stored in MMTable. If not found, then do the second step, else do the third step.

2) Search from all levels from lower ones to higher ones, first check bloom filter and fence pointer to see if the key value pair is in that level. Then check the zones to find the specific file block that this key value pair may be in, read that file block to get the result.

3) Filter the result by searching from the delete table. If the value has already been deleted from range delete, then return null, else return the result.

3. Scan/Range Scan

1) Search from all the runs of all the levels, do a range query to get all the key value pairs of that level.

2) Search from the MMTable, get all the key value pairs.

3) filter the results by searching from the delete table. If the value has already been deleted, then remove it from the results.

4. Delete/Range Delete

Point Delete

Point Delete operation is the same as the Put operation. We put a special key value pair called tombstone, which has a special tag(visible) to distinguish. The previous key value pairs will be removed during compaction.

Range Delete

We generate a new Record with max/min key and timestamp, and add the record to the delete table.

5. Compaction

Timing: put() and close database. Showed as Figure 4

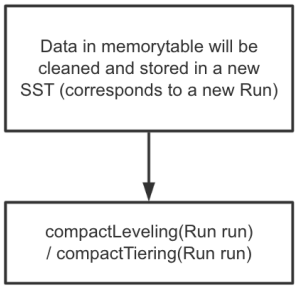

Figure 4

Leveling Compaction

For leveling compaction, each level has just 1 Run, and when doing compaction, data from 2 runs merge together to form a new run, and the level threshold means the number of data pieces in that single run. The compaction will repeat the following steps, shown as Figure 5.

1) Get the data from the current Run r1, merge it with the data from the Run r2 in the next level. Delete the files of r1 and r2.

2) Create a new Run based on the merged data and put it into the next level. And then check if the size of the new Run is over the threshold of that level, if yes, then go back to step 1.

Tiering Compaction

For Tiering compaction, each level has several Runs, and when doing compaction, we will compact all the runs from the current level to form a new Run and add it to the next level. The level threshold here means the number of Runs in that level. The compaction will repeat the following steps. Shown as Figure 6.

1) If the number of Runs in the current level is over the threshold, then get all the data from all the Runs of current level and merge them. Delete files of all the Runs of current level.

2) Create a new Run based on the merged data and put it into the next level. And check if the number of Runs in that level is over threshold, if yes, then go to step 1.

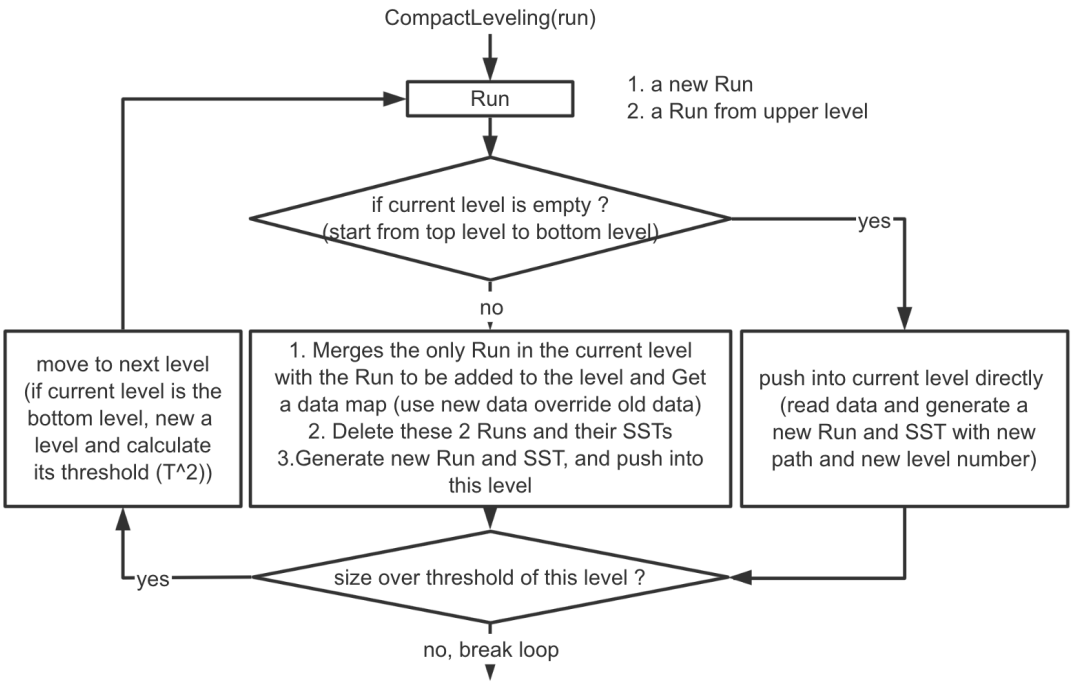

Figure 5

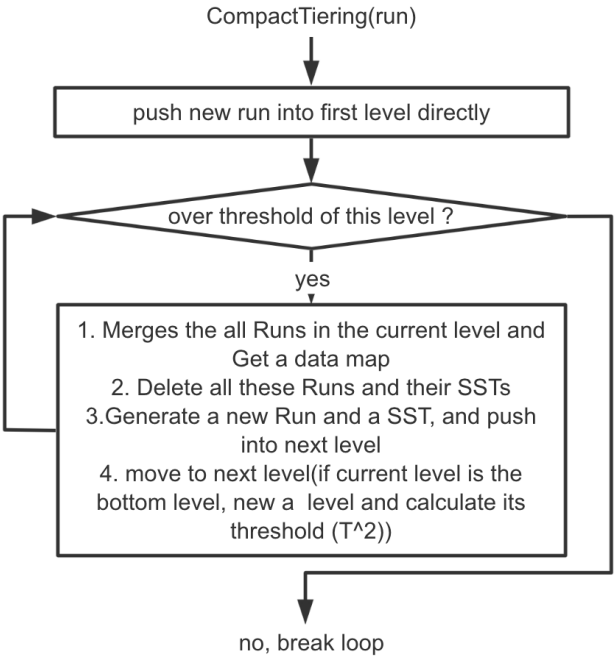

Figure 6

4 Experiment Result

4.1 Experiment Environment

All the below tests and experiments are performed on the 2021 MacBook Pro14, with 8 core M1 pro chip, 16G Memory and MacOS 12.0 operating system. The C++ version is C++17.

4.2 Basic Tests and Performance Test

We design the basic tests and persistence tests to test the basic operations of the database, database persistence and multiple database functions. Shown as Figure 7 and Figure 8.

```
/Users/albertan/Documents/CS561/561-final-project/cmake-build-debug/tests/persistence_test
[==========] Running 2 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 2 tests from PersistenceTest
[ RUN      ] PersistenceTest.BasicOpenClose
[       OK ] PersistenceTest.BasicOpenClose (10 ms)
[ RUN      ] PersistenceTest.DeleteOpenClose
[       OK ] PersistenceTest.DeleteOpenClose (6 ms)
[----------] 2 tests from PersistenceTest (17 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test suite ran. (17 ms total)
[  PASSED  ] 2 tests.
```

Figure 7

```
/Users/albertan/Documents/CS561/561-final-project/cmake-build-debug/tests/basic_test
[==========] Running 6 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 6 tests from DBTest
[ RUN      ] DBTest.IsEmptyInitially
[       OK ] DBTest.IsEmptyInitially (8 ms)
[ RUN      ] DBTest.GetFunctionality
[       OK ] DBTest.GetFunctionality (3 ms)
[ RUN      ] DBTest.PutAndGetFunctionality
[       OK ] DBTest.PutAndGetFunctionality (2 ms)
[ RUN      ] DBTest.DeleteFunctionality
[       OK ] DBTest.DeleteFunctionality (10 ms)
[ RUN      ] DBTest.ScanFunctionality
[       OK ] DBTest.ScanFunctionality (6 ms)
[ RUN      ] DBTest.RangeDeleteFunctionality
[       OK ] DBTest.RangeDeleteFunctionality (8 ms)
[----------] 6 tests from DBTest (41 ms total)

[----------] Global test environment tear-down
[==========] 6 tests from 1 test suite ran. (41 ms total)
[  PASSED  ] 6 tests.
```

Figure 8

We then tried the durable test using 3 million insertion dataset and 3 million operation dataset. When the key distribution of the operation dataset is concentrated, we could finish the benchmark test within 2 and a half hours. Shown as Figure 9.

```
/Users/albertan/Documents/CS561/561-final-project/cmake-build-debug/examples/simple_benchmark benchmark_db.txt -f ../../data/test_3000000_10.data -w ../../data/test_3000000_10_5000.wl
Workload Time 8863683553 us

Process finished with exit code 0
```

FIgure 9

4.3 Experiments

We now performed the influence that zone elements numbers, threshold, BF size, key distribution of database and operation, and different kinds of compaction types make to the database reading and writing performance.

The first experiment is to find the relationship between the first level threshold with the database reading and writing performance. We perform our tests based on Leveling strategy, with test_10000_3 dataset and test_10000_3_2000 operation dataset. The number of elements per zone is 50. The MMTable size is 25. We perform each experiment several times and get the average time, the result is listed as follows Figure 10.

From the results, we could see that there exists a level threshold value that could give the best database performance. When the threshold is small, there are more layers and more SSTables, which means that when doing queries, there are more disk I/O operations. And when the threshold is large, then one level will contain more data, and doing a query on a single level will be costly.

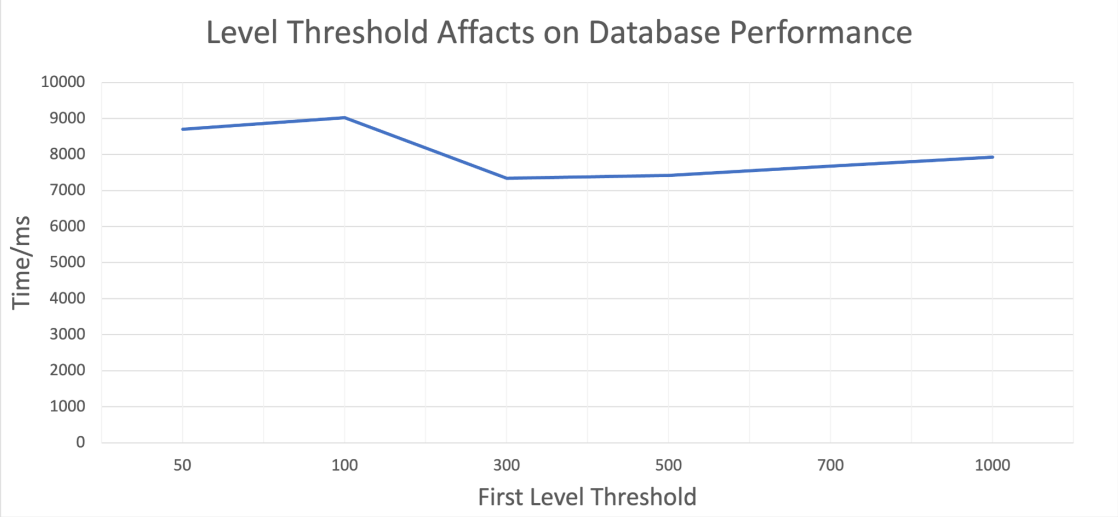

Figure 10

The second experiment is to figure out the relationship between the number of elements per zone with the database reading and writing performance. We performed the experiments based on Leveling and Tiering compaction strategies, with test_10000_3 data dataset and test_10000_3_2000 operation dataset. We set the first level threshold as 25, MMTable size is 25. And the result is shown as Figure 11.

From the results, we could see that there exists a value that could give the best database performance. If the number of elements per zone is small, then there are more zones, which means there are more disk I/O

operations, and if the number of elements per zone is large, then each zone will have more elements, and it will affect the database efficiency.

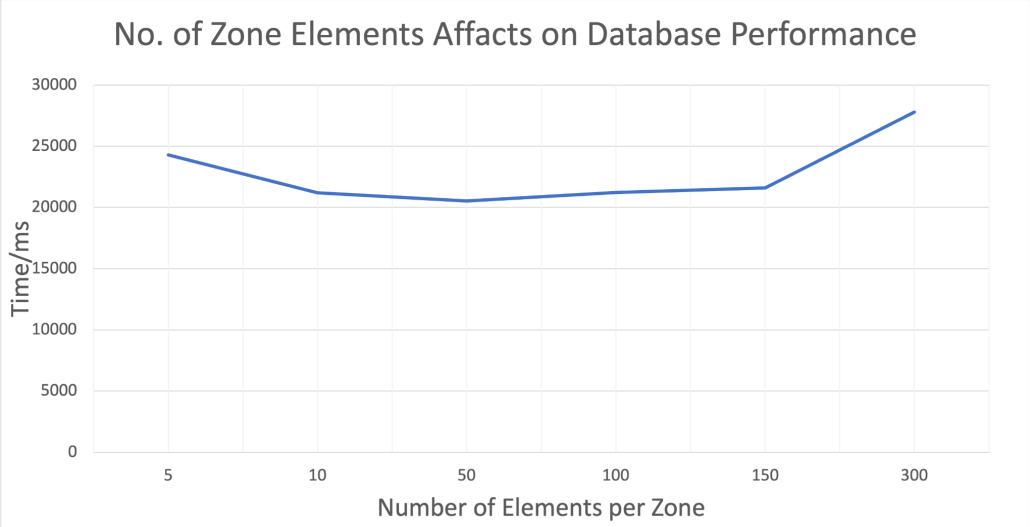

Figure 11

The third experiment is to figure out how the Bloom Filter size affects the database querying performance. We performed the experiments based on Leveling strategy, with test_10000_3 dataset and test_10000_3_2000 operation dataset. We set the number of elements per zone as 50 and the MMTable size as 25, and try the bits per element of Bloom Filter 5 and 10. The result is shown as Figure 12. From the result, we could find that a larger Bloom Filter size will give a better database performance.

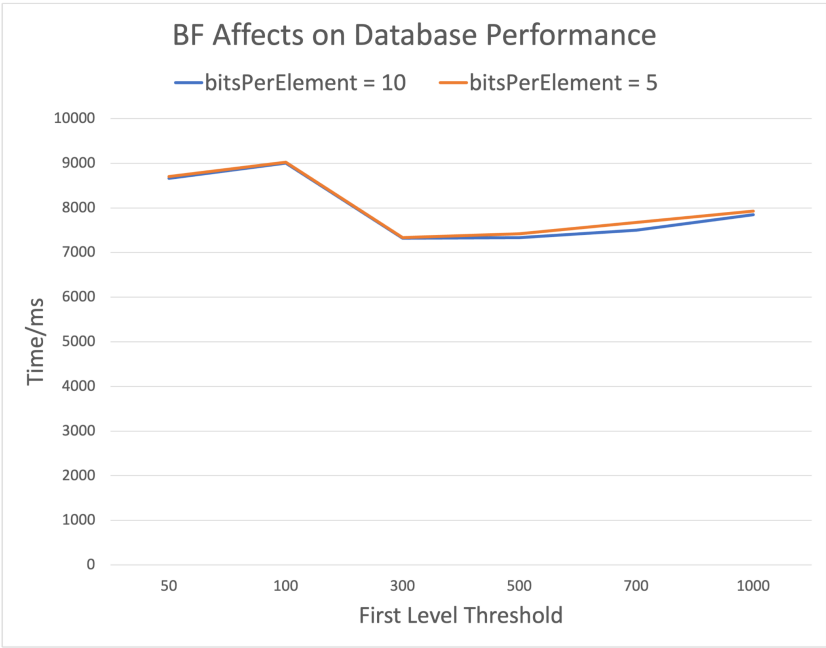

Figure 12

The fourth experiment is to figure out the key distribution effects on database querying performance. We use the Leveling strategy, set the number of elements per zone as 50, first level threshold as 50, MMTable size as 25, and generate 3 operation dataset with different key distributions, test_10000_3_200.wl, test_10000_3_2000.wl, test_10000_3_10000.wl. The result is shown as Figure 13. From the result, we could see that concentrated distribution of keys will benefit the database performance.

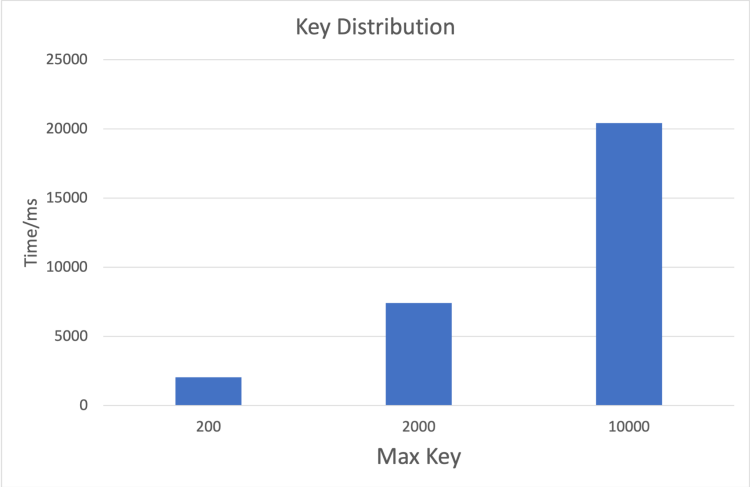

Figure 13

The last experiment is to figure out how different Leveling and Tiering affect the reading and writing performance. Therefore, we generate a writing-heavy dataset and a reading-heavy dataset, and measure the performance of Leveling and Tiering.

Specifically, first we set the first level threshold to 50, number of elements per zone equals to 50, MMTable size as 25, and use test_10000_3_data and test_100_3_200.wl operation dataset(writing dominant), and get the result shown as Figure 14.

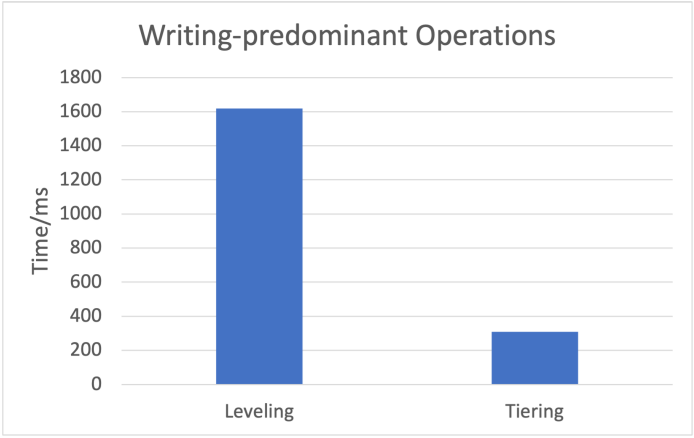

Figure 14

Then, we set the same level threshold, number of elements per zone, and use test_100_3_data and test_10000_3_2000 operation dataset(reading dominant), and get the result shown as Figure 15.

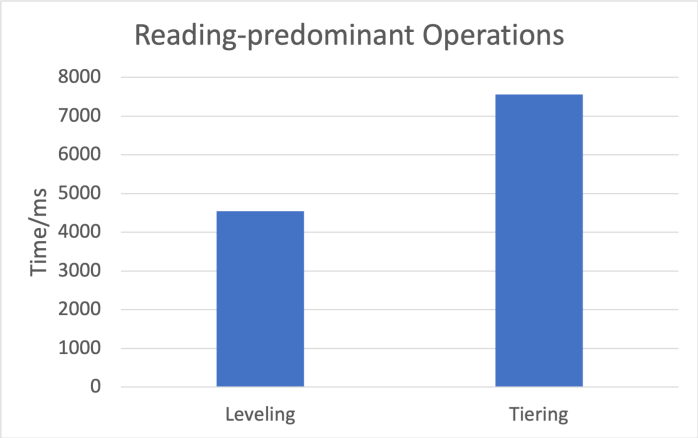

Figure 15

We could find that Tiering is write optimized and Leveling is read optimized.

5 Conclusion

In our implementation of LSM-tree based key value store, firstly, we used the binary file to store the data, which shows great advantages in saving space compared with general text file. Then, we introduced the concept of zones and divided the files into blocks to optimize the query process. We used the 3 million insertion and 3 million operations to test our database and our database shows great robustness and performance.

We tried different tunes from our database. We evaluated the cost trade-offs that different parameters of the database makes to the database performance. We found that there exists a best value for those parameters like "number of elements per zone" and "threshold" that could give the best database performance. And values bigger than that or smaller than that will all give some additional costs. We then showed how a bigger bloom filter will increase the querying speed, and how a concentrated distribution of keys benefits the database performance. Finally, we verified the difference between merge policies. That is, Leveling is reading optimized and Tiering is writing optimized.

usage doc: USAGE.md in the repository.

Project Github: Github address: https://github.com/minghuiyang1998/561-final-project (private, email: yjasmine@bu.edu for permission)