A Dynamic Approach to Tune LevelDB's Hyperparameters for Better Write Performance

Kai Li, Qi Zhang and Yili Luo

Abstract

Existing implementations of LSM key-value stores (e.g., LevelDB, RocksDB) incur high tail latency due to the interference between clients' operations and LSM internal operations, namely the compactions. In this project, we aim to solve this problem by introducing a dynamic approach to tune the hyper-parameters related to the compaction work in LSM. At the high level, our approach increases the hyper-parameters when facing write-intensive workloads (correspondingly allows much more SSTable files at level 0) and decreases the hyper-parameters when facing read-intensive workloads (correspondingly allows much less SSTable files at level 0). Specifically, we built a workload monitor to collect the most recent clients' operations, by calculating the read-to-write ratio, the workload monitor is able to indicate the current workload's feature. To dynamically tune the hyper-parameters, we leverage feedback-control algorithms such as AIMD and MIMD. We implemented our approach atop LevelDB. Compared with native LevelDB, our implementation, LevelDBSILK demonstrated improved write performance without degrading the read performance.

1 Introduction

Log-Structured-Merge (LSM) tree is designed to improve the performance when processing write-intensive workloads. The most two popular implementations of LSM key-value store are LevelDB [4] and RocksDB [3]. However, current implementations incur high tail latency when handling write-intensive workloads. The root cause comes from the interference between clients' operations (reads/writes) and LSM internal operations (e.g., compactions). The internal operations may use up the system's IO bandwidth and thus block the flushing work of memtable and evetually lead to a full memtable which blocks the clients' write operations. To mitigate this problem, we propose a novel technology which dynamically tunes the Hyperparameters related to the compactions. We identified 2 Hyperparameters in LevelDB which control the number of SSTable files at level 0, our approach **LevelDBSILK** works by increasing these Hyperparameters (which correspondingly allows more files at level 0) when facing write-intensive workloads and decreasing them (allow much less files at level 0) when facing read-intensive workloads. Specifically, we built a workload monitor to track the most recent clients' operations, including both read and write operations. By calculating the read-to-write ratio, the workload monitor is able to tell the current workload's feature whether it is read-intensive or write-intensive. Once we obtained the current workload's feature, we feed the feature into the Hyperparameters adjustment component to tune the Hyperparameters accordingly. In the Hyperparameters adjustment component, we implemented both Additive Increase Multiplicative Decrease (AIMD) and Multiplicative Increase Multiplicative Decrease (MIMD) algorithms. We implemented our approach on top of LevelDB and evaluated it by runing different workloads specified in `db_bench`. The evaluation results show that compared with the native LevelDB, our approach LevelDBSILK can improves the write performance by $3X$ without degrading the read performance.

The remainder of this report are orginized as follows. In section 2, we introduce the preliminaries of our project. Section 3 elaborates on our implementation of SILK in LevelDB. Section 4 evaluates the LevelDBSILK and compares it with LevelDB. Section 5 concludes the report.

2 Preliminaries

This section provides some preliminaries of this project.

2.1 LevelDB's Compaction

To implement SILK in LevelDB, the first step is to understand how LevelDB conducts the compaction work. LevelDB adopts two levels of compactions, namely minor compaction and major compaction. Figure 1 shows the Function Call Graph (FCG) of compaction in LevelDB. We elaborate on these two compactions as follows.

LevelDB adopts a single thread to do compaction. Once a write operation starts, it will call the *Write* function, inside *Write*, *MakeRoomForWrite*

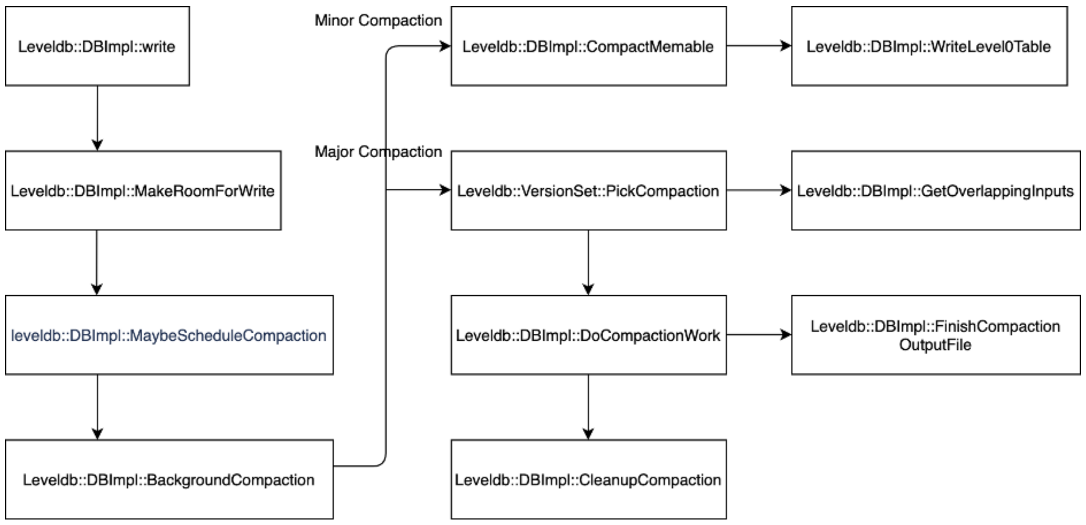
Leveldb::DBImpl::write
Leveldb::DBImpl::MakeRoomForWrite
leveldb::DBImpl::MaybeScheduleCompaction
Leveldb::DBImpl::BackgroundCompaction
Minor Compaction
Major Compaction
Leveldb::DBImpl::CompactMemable
Leveldb::DBImpl::WriteLevel0Table
Leveldb::VersionSet::PickCompaction
Leveldb::DBImpl::GetOverlappingInputs
Leveldb::DBImpl::DoCompactionWork
Leveldb::DBImpl::FinishCompaction OutputFile
Leveldb::DBImpl::CleanupCompaction

Figure 1: Compaction Function Call Graph in LevelDB

will be triggered. *MakeRoomForWrite* implements a write control mechanism using two Hyperparameters: *kL0_SlowdownWritesTrigger* (8 by default) and *kL0_StopWritesTrigger* (12 by default). If the number of files at level 0 reaches to *kL0_SlowdownWritesTrigger*, LevelDB will slow down each write operation by 1ms. If the number of file reaches to *kL0_StopWritesTrigge*, all writes will be blocked.

MaybeScheduleCompaction is the entry of the actual compaction work. Three conditions can trigger this function to be executed.

1. *manual_compaction_ != NULL*, user triggers the compaction manually.

2. *imm_ != NULL*, the Memtable is not NULL pointer and needs to be dumped to SSTable. This refers to the minor compaction.

3. *versions_−>NeedsCompaction*, some level needs compaction according to the compaction score. This refers to the major compaction.

Then the thread will continue to invoke *Bacground-Compaction* and start minor compaction and major compaction based on the above conditions. For a minor compaction, it will be triggered as long as the following two conditions hold simutaneously: 1) memtable exceeds 4MB (defined by *options_.write_buffer_size*); 2) *immutable == NULL*, the Memtable will be rotated to an immutable Memtable first which is further dumped to an SSTable and inserted to the disk.

For a major compaction, it will be triggered by one of following four conditions: 1) Manual trigger; 2) *allowed_seek* used up, every file has a seek limitation, once a seek miss happens, this number will be decreased by one and once it s used up, the compaction will start; 3) The numner of files at level 0 exceeds *kL0_CompactionTrigger* (4 by default); 4) Size at level i (i>0) exceeds 10^i MB.

2.2 Feedback Control

A Feedback Control System is a system which tends to maintain a prescribed relationship of one system variable to another by comparing functions of these variables and using the difference as a means of control. The Feedback control template for the network congestion is defined as:

$$w(t{+}1) = \begin{cases} a_I + b_I * w(t) & \text{congestion not detected, Increase} \\ a_D + b_D * w(t) & \text{congestion detected, Decrease} \end{cases}$$

From the template, we can see there are multiple combinations of feedback control methods with parameters a and b. Based on the existing consequence [2], only additive-increase/multiplicative-decrease (AIMD) and multiplicative-increase/multiplicative-decrease (MIMD) can achieve efficiency. Therefore, we only consider these two algorithms in our dynamic tuning approach.

2.2.1 AIMD

The Mathematical Formula for AIMD [1] is defined as follows:

$$w(t+1) = \begin{cases} w(t) + a & a > 1 \\ w(t) * b & 0 < b < 1 \end{cases}$$

b is typically 0.5.

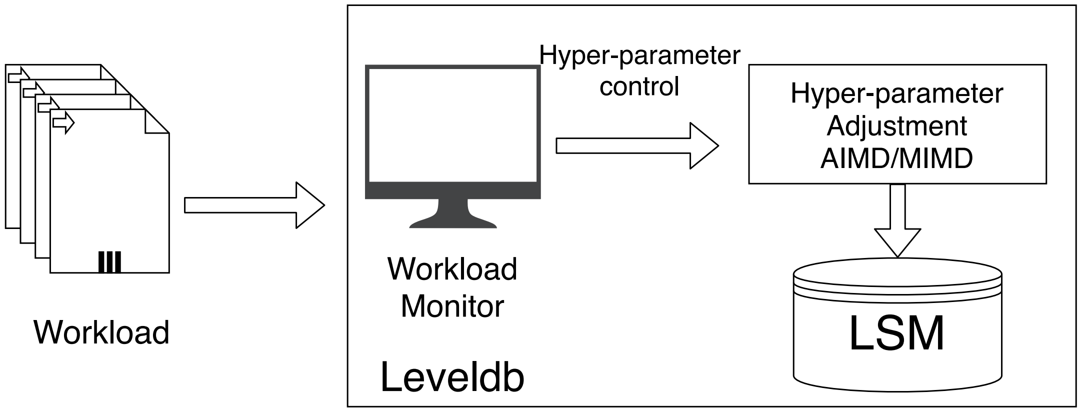

Figure 2: LevelDBSILK System

2.2.2 MIMD

The Mathematical Formula for MIMD is defined as follows:

$$w(t+1) = \begin{cases} w(t) * b_I & b_I > 1 \\ w(t) * b_D & 0 < b_D < 1 \end{cases}$$

In the later section, we will implement these two feedback-control algorithms and compare their performance.

3 Implementation

This section illustrates the implementation details of LevelDBSILK. The system architecture is shown in Figure 2. We implemented a workload monitor to track the most recent clients' operations. The pseudocode is shown in listing 1. We defined three terms to track the clients' most recent operations, namely long/middle/short terms. We calculate the read-to-write ratio for each term respectively and assign each ratio with different weights. Finally, the workload's feature is derived by comparing the weighted ratio with a threshold. If the weighted ratio equals to or larger than the threshold, the current workload is read-intensive, otherwise, the workload is write-intensive. Eventually, we feed the derived workload feature to the Hyperparameters adjustment component which employs feedback-control algorithms to tune the Hyperparameters. .

```
class Statistics {
 public:
  Statistics();
  ~Statistics();

  uint64_t longterm_length;
  uint64_t midterm_length;
  uint64_t shortterm_length;

  vector<int> vect_l(longterm_length);
  vector<int> vect_m(midterm_length);
  vector<int> vect_s(shortterm_length);

  float longterm_weight;
  float midterm_weight;
  float shortterm_weight;

  float boundary; // larger than this value is
    write-intensive, smaller than it would be
    read-intensive
  // TODO set a safe boundary window to avoid
    sharp turn-around

  float CalculateFeature(); // long/mid/short
    term stats collectively decide the workload
     feature

  void AWrite();
  void ARead();
  bool Reset() override;

 private:
  uint64_t longterm_read;
  uint64_t longterm_write;

  uint64_t midterm_read;
  uint64_t midterm_write;

  uint64_t shortterm_read;
  uint64_t shortterm_write;
}
```

Listing 1: Workload Monitor

4 Evaluation

4.1 Experimental Environment

In this project, we conducted experiments on a machine equipped with Intel 4-core i5-7500 CPU of 3.40 GHz with 8 MB cache, a 8 GB RAM and 256 GB disk. We ran the workloads speficied in db_bench, including read-intensive workloads and write-intensive workloads. For

Workloads	LevelDB	LevelDBSILK (AIMD)	LevelDBSILK (MIMD)
fillseq	6.657 ms	6.681 ms	6.626 ms
fillrandom	20.378 ms	8.922 ms	6.797 ms
overwrite	27.774 ms	10.727 ms	7.446 ms
readrandom	3.686 us	3.251 us	3.238 us
readseq	0.498 us	0.391 us	0.383 us
readreverse	3.342 us	2.651 us	2.643 us

Table 1: Performance comparison of LevelDB and LevelDBSILK

AIMD, the additive increase factor is fixed at 2. For MIMD, the multiplicative increase factor is fixed at 2. In both two algorithms, the multiplicative decrease factor is fixed at 2.

4.2 Run LevelDBSILK

When executing db_bench, we set options *num*, *value_size* and *write_buffer_size* to be 2000, 32768 and 65536 respectively. We compare the performance of LevelDBSILK with LevelDB by running the experiment three times and reporting the average latency. For LevelDBSILK, we run both AIMD and MIMD feedback control algorithms. The experiment results are summarized in table 1.

From the table, it can be seen that for pure-write workloads, including *fillseq, fillrandom, overwrite*, LevelDBSILK improve write performance by $3X$ compared with native LevelDB, and for different feedback control algorithms, MIMD works better than AIMD in handling write-intensive workload. For pure-read workloads including *readrandom, readseq, readreverse*, LevelDBSILK achieves similar read performance to LevelDB. The two feedback control algorithms have similar read performance.

5 Conclusion

This work proposed an adaptive approach to improve the write performance in LevelDB by dynamically tuning the Hyperparameters. We built a workload monitor to track clients' operations and infer the current workload's feature, we aslo built a Hyperparameters adjustment component which leverages existing feedback-control algorithms to tune the Hyperparameters. The evaluation results show that our solution achieves $3X$ write performance improvement than the native LevelDB without degrading the read performance.

References

[1] Additive increase/multiplicative decrease. Additive increase/multiplicative decrease — Wikipedia, the free encyclopedia, 2020. [Online; accessed 04-December-2020].

[2] CS268, Berkeley. Lecture 4: Tcp congestion control. [Online; accessed 04-December-2020].

[3] Facebook. Rocksdb. `https://github.com/facebook/rocksdb`.

[4] Google. Leveldb. `https://github.com/google/leveldb`.