Experience of Implementing SILK in LevelDB

Kai Li, Qi Zhang and Yili Luo

Abstract

Log-Structured-Merge (LSM) key-value store is known to be effective and effcient when processing write-intensive workloads. Existing implementations of LSM key-value stores (e.g., LevelDB, RocksDB) have been widely used and adopted. However, current implementations of LSM incur high tail latency due to the inteference between clients' operations and LSM internal operations, namely the compactions. SILK is a novel technique proposed to solve this problem by cordinating these two kinds of operations. The idea is to throttle the IO bandwidth that can be used by the compaction threads on higher levels during the peak of clients' operations. The prototype of SILK is imelmented as an extension on RocksDB. In this project, we aim to implement the novel technique proposed by SILK in LevelDB. During our implementation, we identified several challenges and proposed an alternative solution based on the essence distilled from SILK, our alternative solution LevelDBSILK demonstrated promising performance improvement when handling write workloads. Currently, our solution performed poorly when handling read-intensive workloads, future work is proposed to improve 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 [?] and RocksDB [?]. However, a recent research paper SILK [?] has shown the current design of LSM incurs high tail latency when handling write-intensive workloads. The root cause of such tail latency is the interference between clients' operations (reads/writes) and LSM internal operations (e.g., compactions). Due to lack of coordination between clients' opetations and internal operations, the compactions may use up the system's IO bandwidth and block the flushing work of memtable and evetually lead to a full memtable which blocks the clients' write operations. To mitigate the tail latency, SILK proposed a novel technology which prioritizes the compactions at the lower levels and limits the IO bandwidth that can be used by compactions at higher levels. The essence of SILK is to minimize the possibility of level 0 becomes full, therefore the memtable can be always flushed to level 0.

The prototype of SILK is implemented based on RocksDB. In this project, we aim to implement the same idea of SILK in LevelDB. However, after examing LevelDB's source code, we identified several challenges. First, unlike RocksDB which adopts multiple threads to do compactions, LevelDB merely adopts a single thread to do compaction. Second, SILK implements two thread queues to schedule those multiple compaction threads and prioritizes the threads at lower levels, we need to implement such two thread queues in LevelDB. Finally, SILK implements a client load minitor which guides SILK to throttle the IO bandwidth used by compaction threads at higher levels.

Given a short period time for this project, it is difficult to achieve the same level of implemtation as SILK in LevelDB. Instead, we choose to implement the essence of SILK which is to minimize the chance that level 0 becomes full during the peak of clients' opetations. To this end, we noticed that there are 2 Hyperparameters in LevelDB which control the size of level 0, we proposed to dynamically tune these Hyperparameters and control the size of level 0. Our proposed solution **LevelDBSILK** works by increasing the Hyperparameters during peak of clients' operations and decreasing it during the low load peroid. We implemented LevelDBSILK and conducted experiments by runing different workloads specified in `db_bench`. The evaluation results show that compared with the native LevelDB, LevelDBSILK improves the write performance by $3X$.

The remainder of this report are orginized as follows. In section **??**, we introduce the preliminaries of our project. Section **??** elaborates on our implementation of SILK in LevelDB. Section **??** evaluates the LevelDBSILK and compares it with LevelDB. Section **??** discuss our current implementaion of LevelDBSILK with future work listed in Section **??**. Section **??** concludes the report.

2 Preliminaries

This section provides some preliminaries of this project.

2.1 SILK's Design

The prototype of SILK is implemented as an extension on RocksDB. SILK proposed a novel IO scheduler which aims to reduce interference between client operations and internal operations. The IO scheduler works as follows.

1. Opportunistically allocating more bandwidth to internal operations during periods of low load.

2. Implementing two thread queues, one for compaction threads at lower levels and the other one for higher levels. It prioritizes compaction threads at the lower levels.

3. Allowing compaction threads at lower levels to preempt that at higher levels, in order to facilitate the compaction work at lower levels.

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

3 Implementation

This section illustrates the challenges we encountered when implementing SILK in LevelDB. We proposed an alternative way to implement the core idea of SILK in LevelDB instead of duplicating the same level implementation of SILK.

3.1 Challenges

As discussed in Section **??**, SILK's IO scheduler is implemented as an extension on RocksDB which schedules multiple compaction threads with different priority. However, when implementing the same idea in LevelDB, an inherent limitation is LevelDB only allows single thread to do compaction.We summarize the challenges of implementing SILK in LevelDB as follows.

1. The first challenge is to implement a multiple threads based compaction in LevelDB.

2. The second challenge is to implement two thread queues with different priority when conducting compaction work.

3. The third challenge is to implement a client load monitor and throttle the IO bandwidth that can be used by the compaction threads.

3.2 Alternative Implementation - LevelDBSILK

Given the time constraint of three weeks, we believe it is difficult to achieve the same level of implementation of SILK. The intuition of SILK is to minimize the possibility that flushing work is blocked by a full level 0 in LSM. In other words, the main purpose of SILK is to minimize the chance that LSM has a full level 0 when processing write-intensive workloads. With the same intuition in mind, we identified an alternative solution called **LevelDBSILK** that achieves the purpose similar to SILK. Recall in **??**, LevelDB adopts two Hyperparameters to control the compaction at level 0. *kL0_SlowdownWritesTrigger* is the threshold of slowing down each write operation by 1 ms, *kL0_StopWritesTrigger* is the threshold of blocking a write operation until the compaction work is finished at level 0. LevelDBSILK achieves the purpose similar to SILK by dynamically tuning these Hyperparameters.

At a high level, during peak of client operations, LevelDBSILK increases these Hyperparameters whenever the current value is hitted by the number of files at level 0. And during the low load peroid of client operations, LevelDBSILK decreases the Hyperparameters until it drop to the initial value.

```
1  if (!bg_error_.ok()) {
2        // Yield previous error
3        s = bg_error_;
4        break;
5      } else if (allow_delay && versions_->NumLevelFiles(0) >=
6      kL0_SlowdownWritesTrigger) {
7        // We are getting close to hitting a hard limit on the number of
8        // L0 files.
9        Log(options_.info_log, "%d level-0 files, waiting 1ms...\n", versions_->NumLevelFiles(0));
10       kL0_SlowdownWritesTrigger *= 2;
11       mutex_.Unlock();
12       env_->SleepForMicroseconds(1000);
13       allow_delay = false;  // Do not delay a single write more than once
14       mutex_.Lock();
15     } else if (!force &&
16                (mem_->ApproximateMemoryUsage() <= options_.write_buffer_size)) {
17       // There is room in current memtable
18       break;
19     } else if (imm_ != nullptr) {
20       // We have filled up the current memtable, but the previous
21       // one is still being compacted, so we wait.
22       Log(options_.info_log, "Current memtable full; waiting...\n");
23       background_work_finished_signal_.Wait();
24     } else if (versions_->NumLevelFiles(0) >= kL0_StopWritesTrigger) {
25       // There are too many level-0 files.
26       Log(options_.info_log, "Too many L0 files; waiting...\n");
27       background_work_finished_signal_.Wait();
28       kL0_StopWritesTrigger *= 2;
```

Listing 1: Source Code of LevelDBSILK

Listing ?? shows the code snippet of Hyperparameters increasing part in LevelDBSILK. In line 10 and 28, we double the Hyperparameter once it is hitted. We defer the Hyperparameters decreasing part to future work.

4 Evaluation

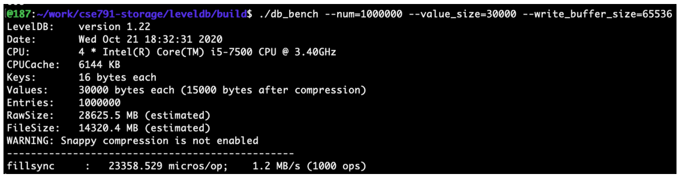
4.1 Experimental Environment

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.

4.2 Reproduce the Problem in LevelDB

The root cause of high tail latency comes from the fact that level 0 is full when handling write-intensive workloads in a short peroid of time. A full level 0 blocks the flush of memtable and eventually leads to a full memtable. In order to reproduce a full level 0, we noticed that in db_bench, there are two options which can be configured to reproduce the desired case, namely *value_size* and *write_buffer_size*. For instance, set a large *value_size* along with a small *write_buffer_size* can fill up the memtable quickly, therefore, flushing happens frequently and can lead to a full level 0 in a short period of time. To this end, we set *value_size* to 30 KB and *write_buffer_size* to 64 KB, which means every 3 write operations can fill up the memtable. We choose a pure-write workload **fillsync** and drive 1000 operations (through option *num*) to reporduce the high tail latency problem.

The experiment result is presented in Figure ??. It can be seen that the amortized latency is 23.359 ms/op and throughput is 1.2 MB/s. Since we did not capture the latency of each write operation, the output does not show how many times the latency spike occurs. However, we can estimate it by counting the occurrence of 'slow down' (line 9) and 'blocking' (line 26) in the log. From the log, we count 607 occurrences of 'slow down' and 164 occurrences of 'blocking'.



```
@187:~/work/cse791-storage/leveldb/build$ ./db_bench --num=1000000 --value_size=30000 --write_buffer_size=65536
LevelDB:    version 1.22
Date:       Wed Oct 21 18:32:31 2020
CPU:        4 * Intel(R) Core(TM) i5-7500 CPU @ 3.40GHz
CPUCache:   6144 KB
Keys:       16 bytes each
Values:     30000 bytes each (15000 bytes after compression)
Entries:    1000000
RawSize:    28625.5 MB (estimated)
FileSize:   14320.4 MB (estimated)
WARNING: Snappy compression is not enabled
------------------------------------------------
fillsync     :   23358.529 micros/op;    1.2 MB/s (1000 ops)
```

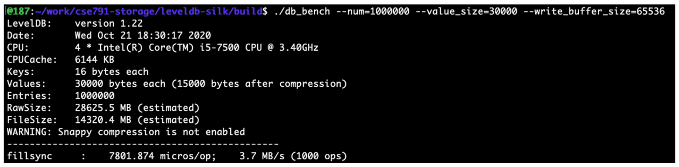
Figure 1: Run fillsync in LevelDB

4.3 Run LevelDBSILK

Under the same setting as described in ??, we test LevelDBSILK with the same workload. The experiment result is presented in Figure ??. It can be seen that the amortized latency is 7.802 ms/op and throughput is 3.7 MB/s. From the log, we count 6 occurrences of 'slow down' and 6 occurrences of 'blocking'. The experiment result shows LevelDBSILK reduces the latency by $3X$ compared with native LevelDB when processing a pure-write workload.

Workloads	LevelDB	LevelDBSILK
fillseq	5.880 ms	5.619 ms
fillsync	9.332 ms	8.293 ms
fillrandom	22.076 ms	4.501 ms
overwrite	37.279 ms	4.628 ms
readrandom	4.769 us	54.319 us
readseq	1.310 us	10.603 us
readreverse	4.132 us	16.178 us

Table 1: Performance comparison of LevelDB and LevelDBSILK



```
@187:~/work/cse791-storage/leveldb-silk/build$ ./db_bench --num=1000000 --value_size=30000 --write_buffer_size=65536
LevelDB:    version 1.22
Date:       Wed Oct 21 18:30:17 2020
CPU:        4 * Intel(R) Core(TM) i5-7500 CPU @ 3.40GHz
CPUCache:   6144 KB
Keys:       16 bytes each
Values:     30000 bytes each (15000 bytes after compression)
Entries:    1000000
RawSize:    28625.5 MB (estimated)
FileSize:   14320.4 MB (estimated)
WARNING: Snappy compression is not enabled
------------------------------------------------
fillsync     :    7801.874 micros/op;    3.7 MB/s (1000 ops)
```

Figure 2: Run fillsync in LevelDBSILK

4.4 Run Multiple Workloads

Under the same setting as described in **??**, we run all the workloads specified in db_bench, including read-intensive workloads and write-intensive workloads and compare the performance of LevelDB and LevelDBSILK. The experiment results are summarized in table **??**.

From the table, it can be found that for pure-write workloads, including *fillseq, fillsync, fillrandom, overwrite*, LevelDBSILK achieves better performance than LevelDB. For pure-read workloads such as *readrandom, readseq, readreverse*, LevelDBSILK achieves worse performance than LevelDB. LevelDBSILK achieves worse read performance due to the larger level 0, because we haven't implemented the Hyperparameters decreasing part and after running those pure-write workloads, LevelDBSILK produces a large number of files at level 0 before executing those pure-read workloads.

5 Discussion

Currently, LevelDBSILK increases the Hyperparameters everytime when the current value is hitted and there is no upper limit on these Hyperparameters. One may concern that these Hyperparameters may increase indefinitely. To prevent it from growing indefinitely, one can add an upper limit to cap these Hyperparameters and the implementation should be simple. However, we believe these Hyperparameters should not grow to a huge number, if this situation happens, it implies that the workloads contain a long-run of write-intensive opetaions and the size of Memtable grows faster than level 0. In this case, even SILK cannot mitigate the tail latency problem.

6 Future Work

For the future work, we plan to implement a client requests monitor as the guidence to dynamically tune the Hyperparameters in order to achieve better performance. Other than the client requests monitor, we will implement the Hyperparameters decreasing part to improve the read performance.

7 Conclusion

In this report, we share the experience of implementing SILK in LevelDB. During the implementation, we identified several challenges due to the inherent difference between RocksDB and LevelDB. Instead of achieving the same level implementaion, we proposed an alternative solution to achieve the purpose similar to SILK. Our proposed solution works by dynamically tuning the Hyperparameters that control the size of level 0 in LevelDB. The evaluation results show that our solution achieves $3X$ improvement than the native LevelDB when processing write-intensive workloads.