Industrial Paper

DLC: A New Compaction Scheme for LSM-tree with High Stability and Low Latency

Peiquan Jin
University of Science and Technology of China
Hefei, China
jpq@ustc.edu.cn

Jianchuang Li
University of Science and Technology of China
Hefei, China
lijc@mail.ustc.edu.cn

Hai Long
Huawei Technologies Co., Ltd.
Shenzhen, China
longhai@huawei.com

ABSTRACT

Many big data systems employ LSM (Log-Structured Merge)-tree-based key-value stores, such as RocksDB and Cassandra. LSM-tree has a multi-level data structure and can transform random writes into sequential ones by compaction operations. However, the compaction operations in LSM-tree introduce the read/write amplification issue, which will increase the processing latency and incur throughput drops. In this paper, to eliminate the impact of compaction on the throughput stability and latency of LSM-tree, we propose a new compaction method called DLC (Delay Level-0 Compaction). We notice that workloads like OLTP are periodically high. For example, an electronic business platform may have high requests at noon and night but have few requests in the early morning. When the workload becomes high, many data will be flushed to Level-0 of LSM-tree from memory, which will trigger frequent Level-0 compaction and lower the system's throughput. The main idea of DLC is to delay Level-0 compaction at a high load and resume compaction when the system becomes low-loaded. As the low-loaded system generally has enough free CPU cores and I/O bandwidths, performing the delayed compaction will not affect the system's overall performance. Therefore, we can maintain stable throughput even when the system is high-loaded. To implement DLC, we first define a new I/O estimation model to characterize the workload. Then, we determine whether to delay Level-0 compaction according to the characteristics of the current workload. Moreover, to deal with sustained high workloads, we invent a burst compaction strategy to reduce throughput dropping and present two implementations for the bursty compaction. We implemented DLC on Myrocks and experimentally compared DLC with the original MyRocks and a state-of-the-art scheme called SILK under various OLTP workloads. The results show that DLC outperforms MyRocks and SILK in both latency and throughput stability.

1 INTRODUCTION

LSM-tree (Log-Structure Merge tree)[17] has been widely used in key-value stores, such as RocksDB and Cassandra. LSM-tree maintains a multi-level data structure, and all data in each level are stored using Sorted String Tables (SSTables), in which all key-values are sorted in order. Data are flushed from memory to the SSTables in the first level (Level-0, or L0 for simplicity) through sequential writes. As sequential writes are much faster than random writes, LSM-tree can offer high writing performance. However, when the SSTables in Level-0 exceeds a threshold, LSM-tree performs a compaction operation to merge the SSTables in

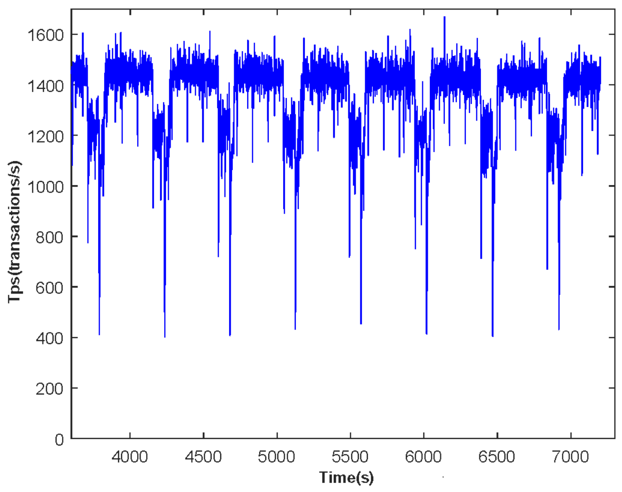


Figure 1: Throughput drops of RocksDB when running on the default OLTP workload generated by *sysbench*.

Level-0 with those in Level-1. If Level-1 also exceeds the threshold, next compaction will also be triggered to merge SSTables in Level-1 with those in Level-2. During a compaction process, both CPU and the I/O bandwidth will be highly used, resulting in the decrease of the overall throughput. To validate the influence of the compaction in LSM-tree, we ran the default OLTP workload in *sysbench*[13] (see Section 6.1 for the detailed settings) on RocksDB and tested the system's throughput. As shown in Fig. 1, there are periodical throughput-drops when the system has been running for a long time, because the system has to perform periodical compaction to merge up-level data into low levels.

The stability of throughput is critical to many applications. For example, an online short-video social network platform like TikTok can not tolerate periodical high latency when playing videos. To improve the throughput stability of LSM-tree, various solutions [3, 6, 16, 18] have been proposed. Among them, the state-of-the-art method is SILK [3], which won the best paper of ATC 2019. The experimental results of SILK showed that it can maintain stable throughput for about 2500 seconds. However, we experimentally found that when SILK kept running for 3500 seconds, it had dramatic throughput drops and the throughput became unstable. This is mainly because the compaction scheduling in SILK can not adapt to the workload changes well.

The compaction operations in LSM-tree are known as background operations (also called internal operations), because they are scheduled on background periodically. A compaction operation consumes a great number of I/O bandwidths because it has to read and write a large amount of data. This is the main reason that affects the throughput stability of LSM-tree. The key challenge to keep throughput stability is how to schedule compaction when the I/O bandwidth is used heavily. An intuitive solution is to reserve some I/O bandwidths for compaction operations, but different levels in LSM-tree have different needs of I/O bandwidths. Therefore, it is hard to reserve appropriate I/O bandwidths for LSM-tree to maintain throughput stability.

Basically, we can roughly divide workloads into two types, namely write-intensive workloads and read-intensive workloads. For write-intensive workloads, the competition for disk I/O between parallel compaction operations is the main reason causing the instability of throughput. In such cases, write stalls may occur and lower the throughput [16]. For this reason, most of existing studies[1, 3, 16, 18] were toward optimizing compaction for write-intensive workloads. However, so far, there is no solution that can offer continuously stable throughput for LSM-tree.

In this paper, we also focus on write-intensive workloads and aim to improve the throughput stability of LSM-tree and lower the processing latency. We propose a new compaction scheme named DLC (**D**elay **L**evel-0 **C**ompaction). DLC aims to achieve three goals. First, it should have a more stable throughput than RocksDB and SILK [3] when running for a long time. Second, it is expected to have lower latency. Third, it should adapt to periodically varying workloads, i.e., the arriving rate of requests is high for a period and then becomes low. The basic idea of DLC is to delay L0[1] compaction at a high load and to resume L0 compaction at a load load. Thus, DLC can work well on workloads with periodically varying workloads[3]. Briefly, we make the following contributions in this study:

(1) We propose a new I/O model to estimate the I/O bandwidth of the current workload effectively and precisely. Our I/O model is inspired by the model proposed by SILK [3], but we devise new estimation functions. Differing from the I/O model of SILK that simply summarizes the data size of all read and write operations, our I/O model introduces two new ideas. First, we remove the data size of write operations, because LSM-tree always writes data to in-memory Memtables, meaning that write operations will not occupy I/O bandwidths. Thus, it is not reasonable to include the written-data size in I/O bandwidth estimation. Second, we divide read operations into *Get* and *Scan* because these two read operations have different I/O costs in LSM-tree. We demonstrate that our model is more accurate than the SILK model and can allocate I/O bandwidths for background compaction more effectively.
(2) Based on the proposed I/O model, we present a new compaction scheme called DLC that delays L0 compaction at a high load and resumes L0 compaction at a load load. We use the new I/O model to characterize the I/O bandwidth need of the current workload, and determine whether the workload is high or low. When the workload is high, we delay L0 compaction. This differs from the compaction schemes in RocksDB and SILK. RocksDB uses a threshold-based compaction scheme and will trigger many compaction operations at a high load, leading to frequent throughput drops. SILK also claims to delay compaction, but it delays the bottom level[2].
(3) Although DLC works well for periodically high workloads, the SSTables in L0 may accumulate under sustained high load, leading to write stalls/stops and serious throughput drops. Thus, we devise a bursty compaction strategy to make DLC suitable for sustained high workloads. We present two implementations for the bursty compaction, namely "resume full compaction" and "resume part compaction". The former is to compact all the SSTables in L0, while the latter is to compact selected part SSTables in L0.
(4) We implemented DLC in MyRocks (MySQL with RocksDB) and evaluated DLC using the *sysbench* tool. We generate various OLTP workloads, including periodically varying workloads, workloads with different read-write ratios, workloads with a long time of a high load, and sustained high workload. We compare DLC with MyRocks and SILK (with the DLC I/O estimation model). The results in terms of throughput and latency show that DLC achieves the best throughput stability and the lowest latency in all experiments.

The remainder of the paper is structured as follows. Section 2 introduces the background and related work. Section 3 presents the I/O estimation model. Section 4 details the DLC strategy. Section 5 discusses the bursty compaction policy. Section 6 reports experimental results. And finally, in Section 7, we conclude the paper and discuss future work.

2 BACKGROUND AND RELATED WORK

2.1 LSM-tree

The basic client operations of LSM-tree are the same as the other NoSQL key-value databases[14, 23], which include *Insert, Delete, Update, Get*, and *Scan*. For convenience, we call *Insert, Delete*, and *Update* as write operations and *Get* and *Scan* as read operations. We take RocksDB as an example to introduce the LSM-tree structure. The RocksDB storage engine mainly consists of two part, Memtable and Immutable Memtables in memory and SSTables in the disk.

LSM-tree uses the Sorted String Table (SSTable) as the basic data structure, which stores key-value pairs in the disk. SSTable is a sorted table which mainly consists of data blocks and meta blocks. Meta blocks store indexes about data block and Bloom filter[5] for read. Data blocks store key-value pairs in sequence for quickly visited by read operations. SSTables are grouped by levels. We call the levels as L0, L1, ..., L_n in short from top to bottom. We call the levels near to the memory as up levels, e.g., L0, and the other levels as low levels. The SSTables in L0 are mainly flushed from Immutable Memtables in memory. The SStables in L1 and low levels are generated by major compaction.

There is one Memtable and one or more Immutable Memtables in memory. Memtable uses *Skiplist* as its structure. Skiplist is a data structure that uses probabilistic balancing. Its algorithms for insertion and deletion are much simpler and significantly faster than equivalent algorithms for balanced trees[19]. We can consider Memtable as an in-memory buffer for inserting, deleting, and updating. When Memtable reaches its capacity threshold, it will be transformed into Immutable Memtable, which cannot be modified by any (write) operations, and a new Memtable will be created for writing new key-value pairs. When a new Immutable Memtable is created, a background thread would be scheduled to flush the Immutable Memtable to disk as one SSTable, in which all the flushed key-value pairs are stored. The flush operation is also called minor compaction.

[1]L_i means Level-i in this paper.
[2]Here, Level-0 is the top level, and Level-i with the biggest i is the bottom level.

With SSTables being accumulated in the same level and reaching the threshold of the level, a major compaction will be triggered (in this paper, we simply use the term "compaction" to represent "major compaction" by default) and may schedule compaction for garbage collection to reduce disk usage and read cost[9][8]. Different levels have different thresholds for compaction, and when the SSTables in L_i or low levels reach the threshold of the level, compaction will be triggered. A compaction operation fetches one SSTable in the level, which triggers the compaction and the SSTables in the next level that have overlapping keys with the SSTable in the up level, then merges sort all key-value pairs in sequence. The merged key-value pairs are then written to new SSTables in the next level. When the flush and compaction of LSM-tree happen in the background, the system can answer write and read requests normally. The new data inserted by write operations can be put into Memtable directly, while read operations will visit Memtable, Immutable Memtables, and SSTables in all levels[9].

2.2 Compaction Optimizations for LSM-tree

2.2.1 Reducing Compaction Cost. There have been a lot of studies toward optimizing the compaction for LSM-tree. One idea is to reduce compaction cost. WiscKey[15] and HashKV[7] separate keys from values to maintain a smaller LSM-tree that contains keys and the pointer to the values, which is effective for large keys and write-intensive workloads. But their optimizations are not suitable for range scans. MonKey[9], Dostoevsky[10] change the structure of LSM-tree by tuning related parameters for different demands, and LSM-Bush[11] proposes a more general structure for more demands. Ahmad and Kemme[1] cope with spike caused by compaction by offloading compaction to a dedicated compaction server, and it solve the cache avalanche by smart warm-up strategy, which is a good method for a distributed database like HBase or Cassandra.

Some people proposed efficient scheduling algorithms to optimize compaction for LSM-tree. bLSM[20] bounds the write latency by using the spring-and-gear merge scheduler, which is not suitable for partitioning structure. Luo and Carey[16] suggest using 95% maximum throughput to run the experiments, which is adjusted to 90% in DLC experiments. dCompaction[18] proposes delayed compaction mainly for low-level compaction, which is a lazy compaction mechanism for write-intensive workloads. Chen et al.[8] uses a priority and fairness mixed compaction scheduling mechanism to reduce write amplification and read amplification.

LDC[6] decreases the tail latency and reduces write amplification by a novel Lower-level Driven Compaction, which is orthogonal to DLC. Most of them concentrate on low-level compaction and pay little attention to up-level compaction.

Other optimizations of compaction include LSbM-tree and TRIAD. LSbM-tree[21] adds one buffer for every level to decrease block cache miss and improve read performance after compaction, which concentrates on reducing cache miss after compaction. TRIAD[2] is designed for skewed workloads. It also delays L0 compaction until there is enough key overlap in L0 to be compacted. However, DLC delays L0 compaction according to the workload.

2.3 State-of-the-Art Optimization: SILK

SILK[3, 4] is the state-of-the-art optimization for the compaction in LSM-tree. In this paper, we mainly focus on improving SILK. Thus, in this section, we briefly introduce the details of SILK.

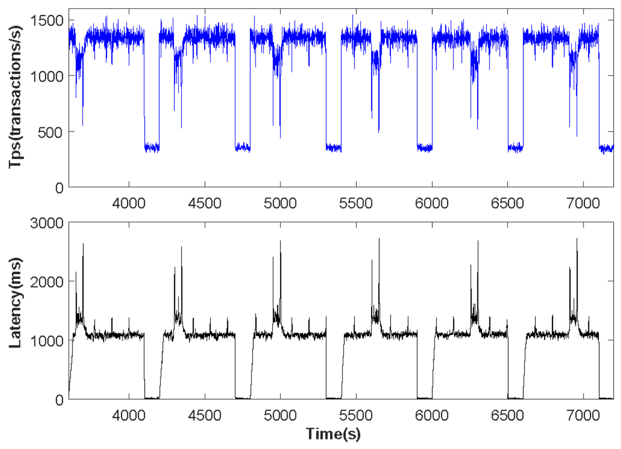


Figure 2: SILK's throughput drops and latency increasing.

SILK proposed an effective way to prevent latency spikes in RocksDB. It used an I/O scheduler for background analysis, paused scheduling low-level compaction when in high loads. The I/O scheduler in SILK can dynamically allocate bandwidth between client operations, so it can allocate more bandwidth to compaction during a low-loaded period.

2.3.1 Compaction of SILK. SILK puts forward two main methods to reduce the compaction cost in RocksDB, namely prioritizing and preempting internal operations. SILK maintains two internal thread pools, one with high priority is for flush and another with low priority is for compaction. According to the SILK I/O scheduler, when the computed bandwidth of client operations exceeds the threshold, SILK will pause compaction. As a result, only up-level compaction can be scheduled and low-level compaction will be delayed during the pausing time. Furthermore, the up-level compaction will be scheduled in the high-priority thread pools, meaning that the compaction may preempt the low-level compaction scheduled in the low-priority thread pool. The preempted compaction will recover compaction work when the thread pool is free. By this way, compaction from L0 can be scheduled along with other compaction paused, and the remaining I/O bandwidths (allocated to flush and compaction) can be allocated to the compaction from L0. When the computed bandwidth of client operations is over the threshold, SILK will resume compaction. More bandwidth will be allocated to internal operations, and parallel compaction can be scheduled. In short, SILK pauses low-level compaction at a high load and resume them at a load load.

2.3.2 Problems. As reported in the original SILK paper [3], SILK can maintain stable throughput for 2000 seconds. However, if a high workload lasts for a longer time than 2000 seconds, SILK will incur dramatic throughput drops and latency increasing. We ran SILK to see its performance and found that SILK has periodical throughput drops and latency increasing after running for more than 3500 seconds, as shown in Fig. 2. This is mainly because after a long-time running, the frequent compaction operations in SILK will consume a large amount of I/O bandwidths. In addition, the I/O analyzer of SILK fails to estimate the workload level accurately. This results in inappropriate compaction scheduling, which will finally trigger write stalls or write stops to delay writing or stop writing to the memory. For example, SILK will resume compaction even when the system runs at a high load.

Although SILK proposed to delay the low-level compaction at a high load, if the up-level compaction cannot finish in time during a load load, SSTables will accumulate in up-levels, which will incur the file retention of LSM-tree [8]. With the increasing of the number of the SSTables to be merged, SILK will finally reach the threshold of write stall or stop, resulting in throughput dropping and latency increasing.

3 I/O ESTIMATION MODEL

In this section, we propose an I/O estimation model inspired by the SILK I/O scheduler [3]. By this model, we can compute the I/O bandwidth taken by client operations more accurately than SILK. Note that the I/O bandwidth estimation is critical to compaction scheduling. If we fail to estimate the bandwidth of the current workload, we may resume compaction at a high load, like SILK, and lead to throughput drops.

3.1 The I/O Estimation Model of SILK

SILK monitors the bandwidth used by client operations and allocates the available I/O bandwidth to internal operations. It realizes its I/O scheduler by setting a separate thread on client load (which is db_bench on RocksDB actually). The I/O scheduler can get actual numbers of client operations what client has accomplished last time interval and computes client's I/O bandwidth according to Eq. 1, where N_{read} and N_{write} are the read times and write times in the last time interval, B_{kv_pairs} is the Bytes of key and value, $T_{interval}$ is the last time interval. SILK sets the limit of total bandwidth $Bandwidth_{limit}$, so it can dynamically allocate left bandwidth to internal operations easily using rate limiter according to Eq. 2, where ε is a small buffer which are not significant enough to adjust internal operation bandwidth.

$$Bandwidth_{client} = (N_{read} + N_{write}) * B_{kv\_pairs}/T_{interval} \quad (1)$$

$$Bandwidth_{internal} = Bandwidth_{limit} - Bandwidth_{client} - \varepsilon \quad (2)$$

3.2 The I/O Estimation Model of DLC

SILK has many restrictions so that it can not be applied to the OLTP workload directly. DLC optimizes the SILK I/O scheduler to make the model suit for the OLTP workload.

The separate thread used by SILK to monitor the bandwidth cannot be used for OLTP workloads.This is because SILK is toward the workloads generated by *db_bench*. But the OLTP workload is generated by *sysbench*. Therefore, we can not monitor the bandwidth of OLTP workloads with the same method as SILK. DLC uses a separate thread on RocksDB to monitor the bandwidth so that it can work on any workloads independently. Other than getting the specific counts of all client operations every time interval, which is proposed by SILK, DLC gets data from *Statistics*, which is a statistical tool provided by RocksDB. In this way, we can get the concrete counts of every client operation. As we can see in Table 1, we can get total counts of client operations according to ticker name (which is added like ticker) from *Statistics*. Based on the SILK's I/O scheduler and the total counts of client operations, we construct the I/O estimation model for DLC. This model is workload-sensitive and it can classify the current load into high or low quickly. This model has two functions, namely computation and analyzing.

3.2.1 Computation. The I/O estimation model of DLC summarizes a universal I/O cost analyzing equation based on Eq. 1 and

Table 1: Some related statistical data in Statistics.

Ticker name	Description
NUMBER_KEYS_READ	Total counts of get (k)
NUMBER_KEYS_WRITTEN	Total counts of write (k, v)
NUMBER_DB_SEEK	Total counts of scan(k_1, k_2)

Eq. 2. DLC computes new I/O bandwidth according to Eq. 3. In contrast to SILK, DLC adds scan s for the client bandwidth estimation because there are some scan operations on OLTP workloads. We also add weights for all client operations so as to estimate the actual I/O cost precisely. It is necessary to add weights for client operations because there exists read amplification[15]. If a *Get* operation is missed in the block cache, it will fetch at least one block from the disk. Thus, one *Get* operation will read more than one key-value pairs on average. Meanwhile, DLC ignores to compute write cost for I/O bandwidths, because write operations insert key-value pairs into memory directly, which do not consume I/O bandwidths. When flush and compaction operations happen, the bandwidth required for writing data to disk belongs to $Bandwidth_{internal}$. Though it is possible to compute the actual value of all weights for client operations, e.g., by a linear programming method, it is inconvenient in practical applications. DLC changes Eq. 3 on the basis of OLTP workloads to make it suitable for other workloads. As OLTP workloads have only one type of transaction with ten *Get* operations, four *Scan* operations, and four *Write* operations, we change Eq. 3 into Eq. 4, N_{trans} is the numbers of the committed transactions in the last time interval. Though there exists delay between operations execution and transaction commits, it is reasonable to assume that $N_{get} = 10 * N_{trans}$ and $N_{scan} = 4 * N_{trans}$. Eq. 4 shows that for OLTP with one type of transaction, all operations in the transaction have the same proportion. We only need to use one operation (*Get* or *Scan*) to compute the real client operation bandwidths. Taking the *Get* operation as an example, we can compute $Bandwidth_{client}$ easily using Eq. 5. DLC allocates the bandwidth to internal operations by using Eq. 2, which is the same as SILK.

$$Bandwidth_{client} = (\omega_{get} * N_{get} + \omega_{scan} * N_{scan}) * \frac{B_{kv\_pairs}}{T_{interval}} \quad (3)$$

$$\begin{aligned} Bandwidth_{client} &= (\omega_{get} * 10 * N_{trans} + \omega_{scan} * 4 * N_{trans}) * \frac{B_{kv\_pairs}}{T_{interval}} \\ &= N_{scan} * (\omega_{get} * 2.5 + \omega_{scan}) * B_{kv\_pairs}/T_{interval} \\ &= N_{get} * (\omega_{get} + \omega_{scan} * 0.4) * B_{kv\_pairs}/T_{interval} \end{aligned} \quad (4)$$

$$Bandwidth_{client} = N_{get} * \omega'_{get}/T_{interval} \quad (5)$$

Though Eq. 5 is easy to compute the actual bandwidth for client operations, there is still a problem that we must compute ω'_{get} every time when we change the workload or the parameters of LSM-tree. It is difficult to change ω'_{get} when running for changeable workloads or auto-tuning LSM-tree[3]. DLC proposes a new idea to compute the actual I/O bandwidth, which is suitable for any workloads and any structures of LSM-tree. As we know, a *Get* operation gets data from the block cache and block cache gets blocks from the disk when a cache miss occurs, so the actual I/O bandwidth is consumed when the block cache gets blocks from the disk. DLC computes the actual I/O bandwidth by

[3]Changeable workload means the proportion of operations can change, auto-tuning LSM-tree means that the parameters of LSM-tree can be changed when running.

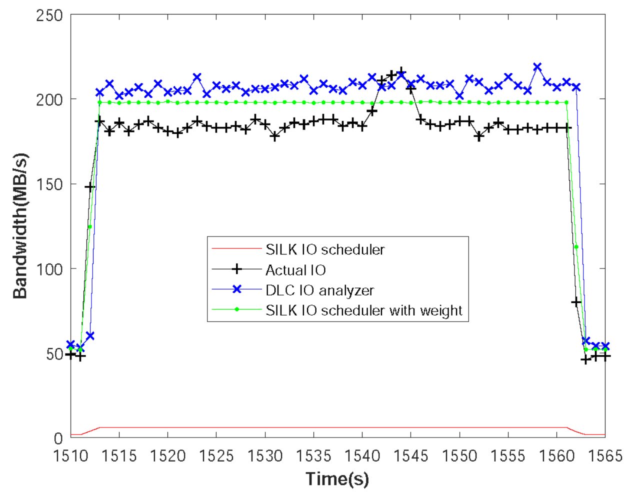


Figure 3: Comparison of various I/O estimation models (using *iostat* on RocksDB).

counting the number of the blocks added to the block cache in the time interval, as we can see from Eq. 6, where γ is the compression ratio of SSTable[4], $Sum_{block\_cache\_added}$ is the number of the blocks added to the block cache, and B_{block} is the bytes within one block.

$$Bandwidth_{client} = \gamma * Sum_{block\_cache\_added} * B_{block}/T_{interval} \tag{6}$$

We conducted an experiment on RocksDB to compare different I/O estimation models with the actual I/O bandwidth. The results are shown in Fig. 3. We can see that the SILK I/O estimation model can not suit for the workload while the DLC I/O scheduler and the SILK I/O scheduler with weights can estimate the actual I/O bandwidth with a little tolerable error, both of which can be applied to our experiments. The DLC I/O scheduler is easier to use than the SILK I/O scheduler with weights.

3.2.2 Analyzing. SILK uses a simple threshold of bandwidth to distinguish between a high load and a load load. It computes $Bandwidth_{client}$ every 10ms and allocates the remaining bandwidth to flush and compaction. DLC adds some parameters for tuning the model, which we can see in Table 2. DLC uses these parameters to distinguish a high load and a load load . For every time_interval, DLC computes the bandwidth of client operations according to Eq. 5. If the bandwidth exceeds the io_high_limit, DLC will regard the workload as a high load. However, low bandwidth is not a sufficient condition of a load load because there are many reasons like flush and compaction that will lead to low bandwidth over a period of time. Thus, even if the client bandwidth is inferior to io_low_limit, we can not judge that it is a low load. DLC uses *softness* to determine the sensitivity of the model. Only when we get several continuous times of low bandwidth can we conclude that the current workload is a low load. By using these parameters, DLC can classify the current workload more correctly than SILK.

[4]Because data in SSTable is compressed while data in the block cache is uncompressed.

4 DESIGN OF DLC

DLC is an improved version of SILK (also an optimization of RocksDB), which optimizes the I/O scheduler of SILK and proposes to delay L0 compaction, which differs from SILK that delays low-level compaction.

4.1 DLC's Compaction Policy

DLC proposes a novel idea to reduce the compaction impact to the throughput stability. The main idea of DLC is to delay L0 compaction[5] at a high load, and to resume L0 compaction when workload is low. DLC uses the I/O estimation model discussed in Section 3.2 to compute the I/O bandwidth, analyze the workload, and decide whether to delay or resume L0 compaction. As we described before, the DLC I/O estimation model can judge the workload more correctly than SILK. First, in the case of fluctuation of workload, DLC sets the time_interval to 1s by default, which can fit most workloads (including OLTP) because many of them also conclude and summarize the statistical data every one second by default. The other parameters in DLC are set according to the workload so that DLC can distinguish the actual high or low workload correctly. Second, on the basis of the actual workload, DLC gives two optional policies for scheduling compaction in a sustained high load, namely delay full L0 compaction or delay part L0 compaction.

To delay full L0 compaction means to delay compaction from L0 to both L0 and L1 under a high load and to resume compaction under a load load. Other than changing the threshold of L0, DLC changes the scheduling mechanism so that it can really delay compaction. Because of the compaction mechanism of RocksDB, up-level compaction will be triggered unexpectedly. Increasing the threshold of L0 can only delay compaction temporarily but the compaction will still be triggered, which is almost uncontrollable. DLC proposes a controllable delay mechanism so that it can delay compaction at a high load and resume compaction at a load load.

To delay part L0 compaction means DLC only delays compaction from L0 to L1. It is a trade-off between read performance and throughput stability. With the accumulation of the SSTables in L0, the read latency will increase and the throughput will decrease. Thus, DLC allows compaction to be scheduled from L0 to L0, which will impact the temporary throughput in a short time but is helpful for future performance. To delay part L0 compaction is suitable for a high load with a relatively long time, which also needs a longer low load to resume compaction. However, this method can not stop L0 compaction fully, so the throughput may drop when compaction from L0 to L0 happens. Therefore, we take "delay full L0 compaction" as the default policy. For both policies, we allow only one low-level compaction under a high load being scheduled.

DLC assumes that there are only a few bandwidths that can be allocated to internal operations at a high load and the flush operations from Immutable Memtable to L0 is unstoppable. Thus, there must be some bandwidths allocated to flush, which may cause a small fluctuation when flush happens. Compared to flush operations and low-level compaction, up-level compaction will cause longer time and bigger fluctuation. The reasons are as follows. First, up-level compaction will merge more SSTables than flush. Second, the key range of up-level compaction is wider than low-level compaction, and the access frequency of the up-level SSTtables is higher than the low-level files[9].

[5]We also call L0 compaction as up-level compaction in this paper.

Table 2: Some parameters added by DLC.

Parameter name	Description	Default Value
io_low_limit	The limitation of io bandwidth to confirm the low load	180MB/s
io_high_limit	The limitation of io bandwidth to confirm the high load	300MB/s
io_limit	The io bandwidth of disk	350MB/s
softness	The sensitivity of DLC to confirm low or high low	3
time_interval	The time interval to compute and allocate the bandwidth	1000ms
ω'_{get}	The modified weight of get operation	16000

The throughput will decrease if up-level compaction is scheduled at high load. Because up-level compaction will occupy a large quantity of I/O bandwidths, client operations cannot get enough bandwidths, yielding the drops of throughput. Thus, DLC delays up-level compaction until the next low load is detected. When there are plenty of bandwidths allocated to compaction at a load load, DLC resumes compaction to make full use of the bandwidth.

4.2 Rate Limiter for DLC

When flush happens, it merge-sorts key-value pairs in all Immutable Memtables (if only one Immutable Memtable, no merge-sort will happen) and writes them to a new SSTable in L0, so flush consumes disk I/O write bandwidth only. But when compaction happens, it first reads related SSTable from disk, merge-sorts them and writes them to a few SSTables in the corresponding level, so compaction consumes both disk I/O write and read bandwidth. Rate limiter is designed to throttle the maximum write speed within a certain limit for lots of reasons. For example, flash writes cause terrible spikes in read latency if they exceed a certain threshold. In other words, the rate limiter is to limit the speed of the data written to the disk. And the rate limiter can be modified for throttling the maximum sum speed of both read and write easily, which is used in DLC. By dynamically allocating left bandwidth is available for flush and low-level compaction but cannot quite effective for up-level compaction because bandwidth is not the only reason for the fluctuation of throughput[1, 21]. The bandwidth exceeds the limit of speed to cause terrible spikes, leads to terrible fluctuation and long latency. So to delay L0 compaction under high load may be the best for OLTP workload to maintain both high throughput and low latency and to resume L0 compaction under a load load to achieve minimal losses of throughput and latency.

5 BURSTY COMPACTION FOR DLC

DLC is mainly designed for the OLTP workload with periodical high and low loads. By delaying L0 compaction at high load and resuming L0 compaction at a load load, we can make full use of the I/O bandwidth with the least throughput loss. However, if the workload becomes continuously high, which is called a sustained high load in this paper, the SSTables in L0 will become more and more, leading to write stalls or stops. When running under a sustained high load, DLC will keep delaying L0 compaction to maintain throughput. The read performance will become worse and the throughput will decrease gradually as time goes. There is no time and bandwidth for compaction at a sustained high load; all read operations amortize the influence of delaying L0 compaction. This problem could be solved when the workload changes into a load load and DLC resumes L0 compaction. However, when running with a sustained high load, the workload will

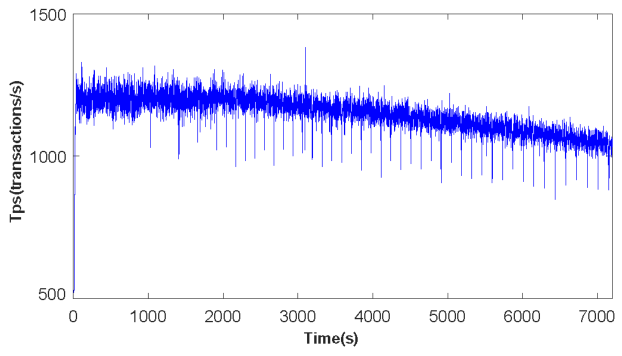


Figure 4: DLC running on the sustained high workload.

not change into a load load, which will gradually affect throughput of DLC, and cause some other unavoidable result such as write stall or write stop[8].

Generally, the system will not always be at a high load. Thus, we can assume that the system's throughput is limited by the number of the accumulated SSTables in L0. However, if the workload keeps high for a long time, we can infer that the system's throughput will finally decrease because more and more SSTables will be accumulated in L0. To verify our analysis, we tested DLC under a sustained high workload and the result is shown in Fig. 4, which shows that the throughput decreases with time. If we do not take any action, the sustained high workload will finally trigger write stalls or stops, which will worsen the performance of DLC.

To make DLC suitable for sustained high workload, we further propose a bursty compaction policy, which can avoid the throughput drops of DLC under the sustained high workload. The idea of bursty compaction is to compact selected SSTables in L0 to avoid the continuous accumulation of SSTables and triggering write stalls or stops. This is implemented by monitoring a threshold representing the number of accumulated SSTables in L0.

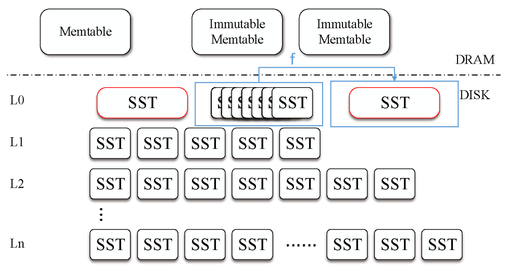


Figure 5: Bursty compaction from Immutable Memtable to L0.

Resume Full Compaction. To resume full L0 compaction is that when the amount of SSTables in L0 gets the threshold, or the total size of SSTables in L0 gets the threshold, DLC stops delaying and resumes compaction from L0 to L1, this cumulatively bursty compaction would consume plenty of time and I/O bandwidth, causing an inevitable degradation of throughput and increase in latency. As we can see in Fig. 4, we could not maintain high throughput all the time and we should schedule compaction in time to keep high throughput and low latency by sacrificing performance for a period of time.

Resume Part Compaction. To resume part L0 compaction is that when the number of the SSTables in L0 flushed from MemTable gets the threshold, DLC resumes compaction from Immutable Memtable to L0. The difference between resume part L0 compaction in the bursty compaction and normal compaction from Immutable Memtable to L0 in MyRocks is that our bursty compaction only merges and sorts the SSTables flushed from Immutable MemTable but normal compaction will merge and sort all SSTables in L0. As compaction will produce a big SSTable that reserves in L0, the bursty compaction does not merge all SSTables to reduce disk I/O bandwidth. Figure 5 shows the idea of the bursty compaction. The SSTable generated from the bursty compaction will not be scheduled again in future bursty compaction.

By controlling the parameters of the threshold, DLC can schedule both two policies easily or only one policy according to the need. Bursty compaction is a new compaction mechanism for sustained high workload, which permits schedule up-level compaction temporarily for future throughput and latency with an inevitable fluctuation for a period of time.

6 PERFORMANCE EVALUATION

6.1 Experimental Setting

We conducted experiments on the Elastic Cloud Server of Huawei Cloud running Linux CentOS 7.6. The server has four Intel Xeon 4-core CPUs with 3.0GHz and 32GB of DRAM. It has one 128GB super-high SSD for storing logs and another 640GB super-high SSD (350MB/s approximately) for storing the MyRocks data.

We use 64 tables in the experiment and each table has 10^7 records. Each key-value pair has a 16B key and a 184B value. We set a 128MB MemTable and only one Immutable Memtable. We set the threshold of L1 to 2GB, the size of SSTable to 64MB, the size-ratio of each adjacent levels to 10, the level of LSM-tree to 5, the block-cache size to 12GB, and the block size to 64KB. The database size is nearly 140GB. We set level0_slowdown_writes_trigger and level0_stop_writes_trigger to 100 both to avoid write stalls or write stops too early.

We compare DLC with two competitors, including MyRocks (MySQL 5.7.26-29 with RocksDB) and SILK. To make the comparison fair, we replace the I/O estimation model of SILK with the DLC I/O estimation model, and we use SILK*(SILK with the DLC I/O estimation model) to indicate this modification.

We use OLTP in *sysbench* [13] as the basic benchmark. The OLTP (Online Transaction Processing) workload in *sysbench* [13] is a SQL workload that can be adjusted to read-intensive or write-intensive. The OLTP workload in *sysbench* has only one type of transaction, which has ten point queries, four range queries, two update queries, one delete query, and one insert query. By using *sysbench*, we can generate OLTP workloads with periodic high and a load loads, which can satisfy most experiments in this paper.

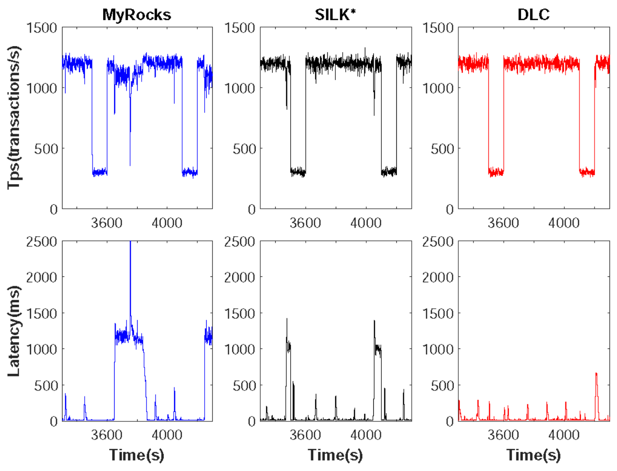


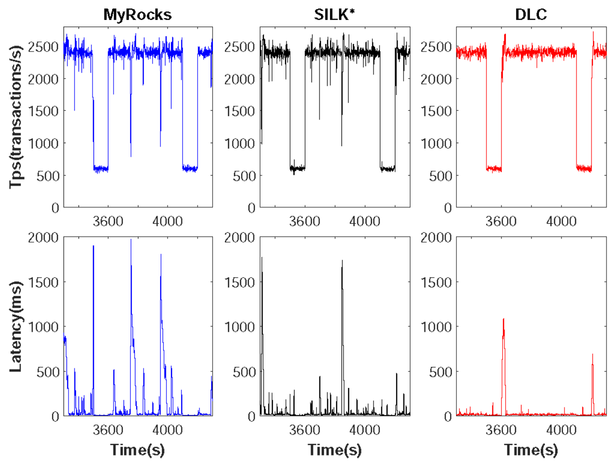
Figure 6: MyRocks, SILK* and RocksDB running on the default OLTP workload.

To make the workload not overwhelm the maximum capacity of the system, in our experiments, we first run the workload to measure the maximum throughput of the system, then we set 90% maximum throughput as the threshold to ensure that the workload will not overwhelm the capability of the system. We mainly evaluate two metrics, namely throughput and latency. We use transactions per second to represent throughput, and use the P99 latency to represent latency. The P99 latency refers to the 99th latency percentile, meaning that 99% of requests(transactions) will be faster than the given latency number, and only 1% of the requests will be slower than the P99 latency. These metrics have also been used in prior work like SILK[3][4].

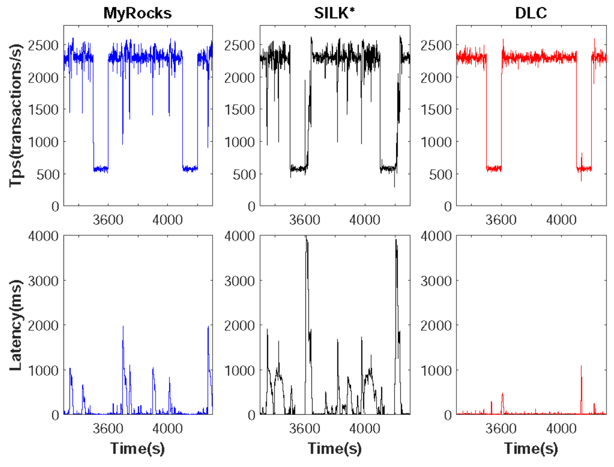
6.2 OLTP with Periodically Varying Workloads

Both DLC and SILK are designed toward periodically varying workloads, i.e., the arriving rate of requests is high for a period and then becomes low. Note that a continuously-high workload will overwhelm the maximum capacity of the system. In this situation, all approaches will fail to keep a stable throughput. On the other hand, most big-data applications like E-commerce platforms have the feature of periodically varying workloads. Another assumption of DLC and SILK is that the workload is write-intensive. This is because only frequent writes can trigger frequent compaction operations, which can be utilized to evaluate the performance of DLC and SILK. How to avoid throughput drops caused by compaction is more challenging that other issues in current LSM-tree-based systems. Although it is important to optimize the read performance of LSM-tree, e.g., under read-intensive workloads, it is orthogonal to this study. An intuitive way to improve read performance is to enlarge the block cache.

To generate appropriate workloads for DLC and SILK, we run the default OLTP workload (each transaction has ten point queries, four range queries, two update queries, one delete query, and one insert query.) in *sysbench* with a high arriving rate for 500s, followed by a low arriving rate for 100s. The high arriving rate is set to 1,200 transaction per second, and the low arriving rate is 300 transactions per second. Note that the two rates and the time period for high/low should be set according to the maximum capacity of the system to be evaluated. In our experiment, the time interval of two up-level compactions is between 400s and



(a) R:W = 5:5



(b) R:W = 1:9

Figure 7: MyRocks, SILK*, and DLC under different read-write ratios.

600s. We list the parameters of the DLC's I/O analyze model on Table 2. We execute the workload on MyRocks, SILK*, and DLC and calculate the throughput and latency continuously. Since at the beginning there is no compaction triggered, we only report the results of all systems between 3,300s and 4,300s. When each system has been running for over 3,000s, we notice that there will be frequent compaction triggered by the insertions of key-value pairs.

6.2.1 Under the Default OLTP Workload. Figure 6 shows the throughput and latency of MyRocks, SILK*, and DLC under the default OLTP workload. In this experiment, the threshold of the maximum capacity of the system is set to 1,200 tps. When the throughput is high (about 1,200 tps in the figure), the system runs at a high load. When the throughput is low (about 300 tps), the system runs at a load load. This is consistent with the periodically varying feature of the workload. MyRocks shows the worst performance. It can not keep stable throughput at a high load because it has to perform up-level compaction at a high load, which will consume additional system resources (I/O bandwidth, CPU, and memory) and lower the throughput. This also leads to the high latency of MyRocks. The throughput stability and latency of SILK* is better than MyRocks, owing to the delay of low-level compaction in SILK*. However, SILK* has a lot of throughput drops during the high-load period. We can see in Fig. 6 that there are serious throughput drops near 3,500s and 4,100s. This is mainly because when the high load runs for a long time, many up-level compactions have been triggered, but SILK* has to perform those up-level compactions even when the system runs at a high load, which also leads to the increasing of the latency of SILK*. On the contrary, DLC exhibits the most stable throughput and the lowest latency compared to MyRocks and SILK*. When the system runs at a high load, DLC can always keep the throughput around 1,200 tps, which is the arriving rate of the high load. When the system runs at a load load, DLC can keep the throughput around 300 tps, which is the arriving rate of the low load. We can see in the figure that DLC has no serious throughput drop. Moreover, the latency of DLC is much lower than others, because it always performs up-level compaction at a load load. In summary, DLC achieves a more stable throughput and higher time performance than its competitors.

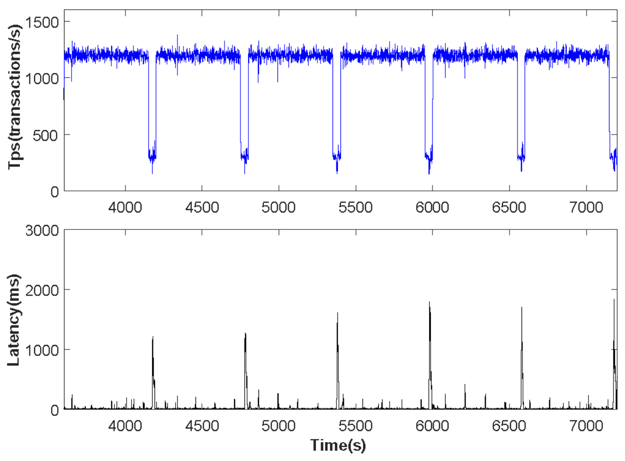
6.2.2 Varying the Read-Write Ratio. In this experiment, we test the performance of DLC under OLTP workloads with different ratios of read and write requests. As DLC is proposed for write-intensive workloads, we prepare two types of OLTP workloads with a read-write ratio of 5:5 and 1:9. Note that in this experiment, we remove the range and update queries from the OLTP workloads to make the workload easier to be generated. When the read-write ratio is set to 5:5, the threshold of the maximum capacity is set to 2,500 tps, which is determined by running the workload before the experiment. When the read-write ratio is 1:9, the threshold is set to 2,350 tps. Figure 7 shows the throughput and latency of MyRocks, SILK*, and DLC under the two read-write ratios. We can see that DLC performs better under the 1:9 read-write ratio, showing that DLC is more efficient for write-intensive workloads. For the workload with the 5:5 read-write ratio, DLC also achieves the best stable throughput and the lowest latency than MyRocks and SILK*.

6.2.3 High Load with a Long Time. Next, we evaluate the performance of DLC under a long period of a high load. This experiment is to show whether DLC can still keep high performance under a long time of a high load. For this sake, we also use the default OLTP workload but shorten the time period of a load load to only 50s, which means that we leave little time for DLC to perform delayed up-level compaction. In addition, we change the time period of a high load to 550s and 1,100, respectively. Consequently, we get two workloads, one is with 550s high load followed by 50s low load, and the other is 1,100s high load followed by 50s low load.

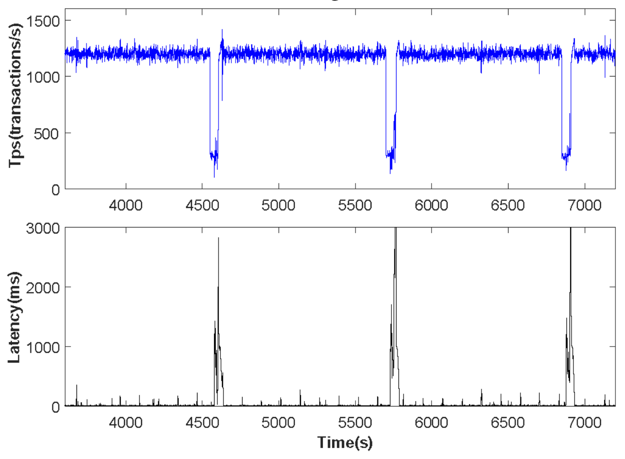
Figure 8 shows the throughput and latency of DLC under the two kinds of workloads. We can see that DLC maintains a stable throughput even when the high-load period increases from 550s to 1,100s, indicating that DLC can adapt to workloads with varying periods of a high load. This also shows that the compaction scheduling cost of DLC is relatively low and DLC can quickly detect the status change of the workload and perform the delayed up-level compaction. Note that DLC has periodic high latency arising under the 1,100s high load, as shown in Fig. 8(b). This is because there are more accumulated SSTables in L0, which cost more time of DLC to complete the compaction.

6.3 Performance of Bursty Compaction

In this experiment, we verify the efficiency of the bursty compaction of DLC. When the workload becomes continuously high

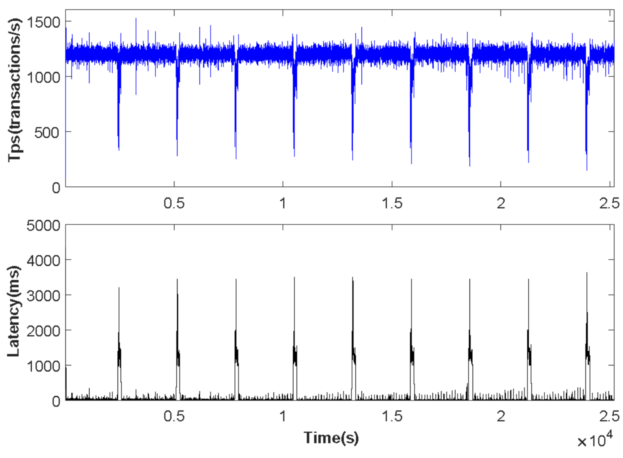


(a) 550s high load

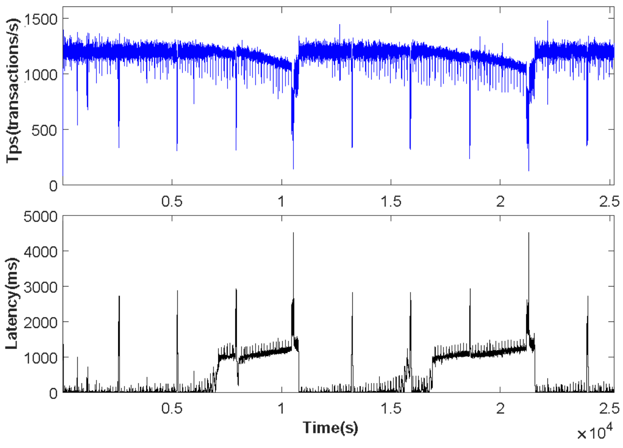


(b) 1100s high load

Figure 8: DLC under high load with a long time.



(a) resume full compaction only



(b) resume part and full compaction

Figure 9: Performance of DLC on a sustained high load.

(which is named sustained high load), the accumulated SSTables in L0 will become more and more, which will worsen the read performance and lower the throughput. DLC monitors the number of the accumulated SSTables in L0, and if the number exceeds a threshold, DLC will perform the bursty compaction to merge selected SSTables in L0 to L1.

To generate a sustained high load, we run the default OLTP workload continuously at the high arriving rate (1,200 tps), and let the system run for a long time to make the number of the SSTables in L0 increase to the threshold (which is set to 20 in this experiment). Figure 9 shows the throughout and latency trend of DLC under a sustained high load. Figure 9(a) shows the result of the "resume full compaction" policy, which is to resume full L0 compaction when the number of SSTables flushed from MemTable exceeds the threshold. Figure 9(b) shows the result of the "resume part and full compaction" policy, which is to resume part L0 compaction when the number of SSTables flushed from MemTable reaches the threshold and to resume full L0 compaction when part compaction has been scheduled for four times.

Figure 9 shows that both the two policies can maintain stable throughput for about 2,550s with a short time of throughput degradation (about 60s for the "resume full compaction" and about 40s for the "resume part compaction"). The "resume full compaction" policy can quickly resume high throughput after bursty compaction, but it has to compact all SSTables in L0, which is time-consuming. We can see in Fig. 9(a) that the latency of the "resume full compaction" becomes extremely high when DLC performs the full compaction. On the other hand, the "resume part compaction" policy only compact selected partial SSTables for maintaining a stable throughput of DLC. Thus, the cost of part compaction is lower than that of full compaction. As shown in the figure, the latency of part compaction is lower than that of full compaction. However, the "resume part compaction" policy sacrifices part of the read performance (read requests still need to read many SSTables in L0), resulting slightly dropping of throughput. To avoid continuous throughput-drops caused by the accumulation of the SSTables in L0 (even after part compaction), the "resume part compaction" in DLC performs full compaction when part compaction has been scheduled for four times. As shown in Fig. 9(b), the throughput slightly drops with time but resume to a high level after four part compactions (each serious drop in the figure indicates part compaction).

6.4 Impact on Read Performance

In this experiment, we measure the impact of DLC on the read performance. Basically, as DLC delays the up-level compaction, there may be accumulated SSTables in L0, which will worsen the read performance under read-intensive workloads.

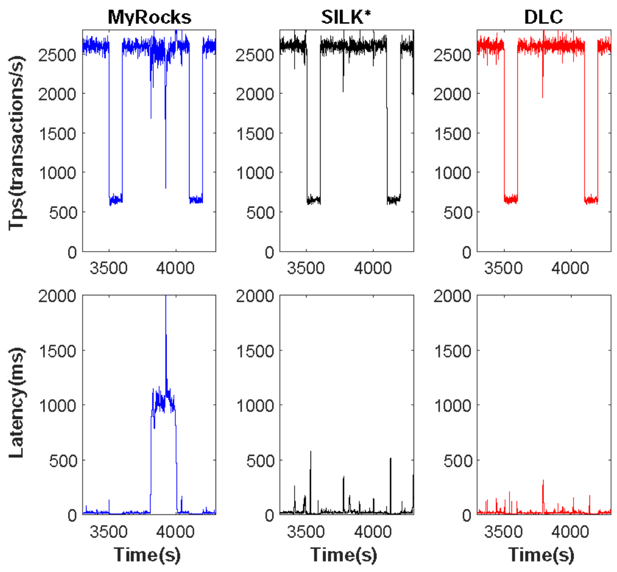


Figure 10: MyRocks, SILK*, and DLC under read-intensive workload with 90% reads and 10% writes.

We first modify the OLTP workload used in Fig. 7 by changing the read-write ratio to 9:1, preparing a read-intensive workload. Then, we run MyRocks, SILK*, and DLC to compare the throughput and latency. The results are shown in Fig. 10. Although DLC shows worse performance compared to its performance under write-intensive workloads (see Fig. 7), it still has comparable throughput stability with SILK*, and its latency is lower than that of SILK* and MyRocks. Thus, DLC can also work for read-intensive workloads.

Further, to measure the number of the SSTables in L0, we conduct an additional experiment to see the change of the number of L0 SSTables in DLC. In this experiment, we use the default benchmark tool *db_bench* in RocksDB and simply run DLC on RocksDB to calculate the number of the SSTables in L0 while DLC is running. We set one thread for inserting key-value pairs and thirty threads to perform *Get* operations. Figure 11 shows the change of the number of the SSTables in L0 as well as the read performance (in terms of QPS, because *db_bench* does not support multi-transaction processing). We can see that the number of L0 SSTables increases with time stably. Note such increase is not a linear function. We explicitly show a part of the enlarged curve in the figure, indicating that the increasing of SSTables is step-wise. This is because only when we flush Immutable Memtable to L0, the number of SSTables in L0 can increase. With the accumulation of the SSTables in L0, QPS slightly decreases while the read latency increases. Figure 12 shows the change of the number of the SSTables in L0 as well as the read performance when we use *Scan* operations and other settings remain unchanged, which shows similar results as Fig. 11.

In summary, DLC is especially suitable for write-intensive workloads, but it can also maintain comparable performance with SILK* under read-intensive workloads. Although the delay of up-level compaction results in the accumulation of the SSTables in L0, DLC can merge them to L1 at a load load or by performing bursty compaction.

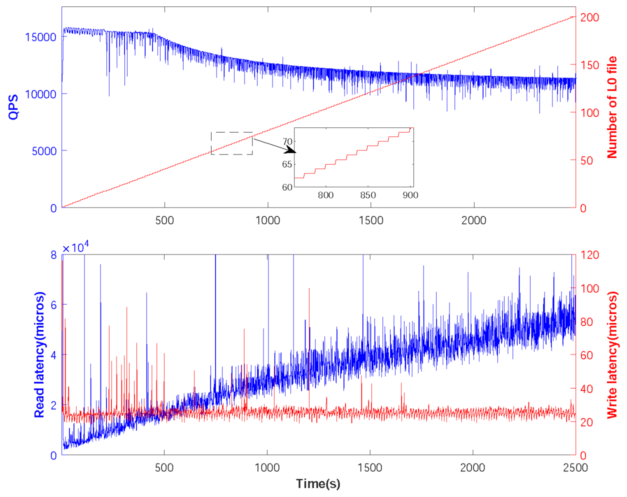


Figure 11: Accumulation of L0 SSTables and its impact on *Get* performance.

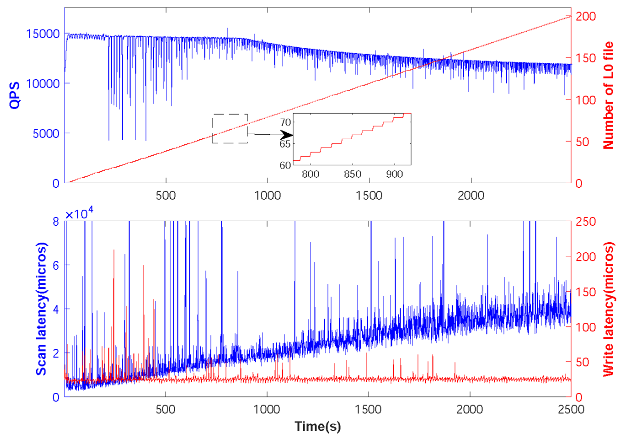


Figure 12: Accumulation of L0 SSTables and its impact on *Scan* performance.

7 CONCLUSION AND FUTURE WORK

LSM-tree has been widely used in many key-value stores, due to its high writing performance. However, the compaction operations in LSM-tree highly impact the throughput of LSM-tree, especially when LSM-tree runs under write-intensive workloads. Prior work has shown that compaction will result in serious throughput drops and increasing in processing latency. In this paper, aiming to provide stable high throughput and low latency, we proposed to delay the L0 compaction in LSM-tree when the system is at a high load and perform the delayed L0 compaction at a load load. With such a mechanism, the system's throughput can maintain a high level at a high load because no up-level compaction will be executed. On the other hand, performing compaction at a load load has little impact on the throughput because the system' resources, including I/O bandwidth and CPU, are not fully used.

Following the idea of delaying L0 compaction, we presented the DLC approach to optimize the compaction scheme in LSM-tree. We first proposed a new model to estimate the I/O bandwidth that is needed by the workload. Based on the I/O estimation model, DLC decided whether to delay the up-level compaction or to perform the delayed compaction. DLC is especially designed for periodically varying workloads, i.e., the arriving rate of requests is high for a period and then becomes low. By scheduling up-level compaction appropriately, DLC can main stable throughput and latency. Further, to solve the problem that the workload is continuously high for a long time, which is called a sustained high load in the paper, we proposed the bursty compaction policy to perform mandatory compaction of the SSTables in L0, so as to avoid the drops of the throughput. We designed two policies to implement the bursty compaction, namely "resume full compaction" and "resume part compaction". The difference between the two policies lies in the range of the L0 SSTables to be compacted.

Finally, we implemented DLC on RocksDB and compared DLC with MyRocks and SILK* (SILK with the DLC I/O estimation model), which is the state-of-the-art optimization of the compaction in LSM-tree. The experimental results under different kinds of OLTP workloads suggest that DLC has the best throughput stability and the lowest latency. We also demonstrated that DLC can achieve comparable performance with SILK* under read-intensive workloads.

In the future, we will consider optimizing the read performance of LSM-tree and building a read/write-optimized tree structure[12]. The current design of DLC is not read-friendly, making it more suitable for write-intensive workloads. We will focus on improving the block cache management scheme[22] and the Bloom filter to reduce the read amplification and block-cache miss in LSM-tree.

ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers for their valuable comments on this paper. We are also grateful to Prof. Jianliang Xu for his constructive suggestions on improving the paper. This work is partially supported by the National Science Foundation of China (No. 62072419 and No. 61672479) and Huawei Technologies Co., Ltd.

REFERENCES

[1] Muhammad Yousuf Ahmad and Bettina Kemme. 2015. Compaction management in distributed key-value datastores. *Proceedings of the VLDB Endowment* 8, 8 (2015), 850–861.

[2] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. 2017. TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores. In *Proceedings of 2017 USENIX Annual Technical Conference (ATC)*. 363–375.

[3] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. 2019. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *Proceedings of 2019 USENIX Annual Technical Conference (ATC)*. 753–766.

[4] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, and Diego Didona. 2020. SILK+ Preventing Latency Spikes in Log-Structured Merge Key-Value Stores Running Heterogeneous Workloads. *ACM Transactions on Computer Systems* 36, 4 (2020), 1–27.

[5] Burton H Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.

[6] Yunpeng Chai, Yanfeng Chai, Xin Wang, Haocheng Wei, Ning Bao, and Yushi Liang. 2019. LDC: A Lower-Level Driven Compaction Method to Optimize SSD-Oriented Key-Value Stores. In *Proceedings of the 35th International Conference on Data Engineering (ICDE)*. IEEE, 722–733.

[7] Helen H. W. Chan, Yongkun Li, Patrick P. C. Lee, and Yinlong Xu. 2018. HashKV: Enabling Efficient Updates in KV Storage via Hashing. In *Proceedings of 2018 USENIX Annual Technical Conference (ATC)*. 1007–1019.

[8] Lidong Chen, Yinliang Yue, Haobo Wang, and Jianhua Wu. 2018. A Priority and Fairness Mixed Compaction Scheduling Mechanism for LSM-tree Based KV-Stores. In *Proceedings of the International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP)*. Springer, 89–105.

[9] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD)*. ACM, 79–94.

[10] Niv Dayan and Stratos Idreos. 2018. Dostoevsky: Better space-time trade-offs for LSM-tree based key-value stores via adaptive removal of superfluous merging. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD)*. ACM, 505–520.

[11] Niv Dayan and Stratos Idreos. 2019. The log-structured merge-bush & the wacky continuum. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD)*. 449–466.

[12] Peiquan Jin, Chengcheng Yang, Christian S. Jensen, Puyuan Yang, and Lihua Yue. 2016. Read/write-optimized tree indexing for solid-state drives. *The VLDB Journal* 25, 5 (2016), 695–717.

[13] Alexey Kopytov. [n.d.]. Sysbench. https://github.com/akopytov/sysbench.

[14] Ruicheng Liu, Peiquan Jin, Xiaoliang Wang, Zhou Zhang, Shouhong Wan, and Bei Hua. 2019. NVLevel: A high performance key-value store for non-volatile memory. In *Proceedings of the 21st IEEE International Conference on High Performance Computing and Communications (HPCC)*. IEEE, 1020–1027.

[15] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2017. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Transactions on Storage* 13, 1 (2017), 5.

[16] Chen Luo and Michael J. Carey. 2019. On Performance Stability in LSM-based Storage Systems. *Proceedings of the VLDB Endowment* 13, 4 (2019), 449–462.

[17] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385.

[18] Fengfeng Pan, Yinliang Yue, and Jin Xiong. 2017. dCompaction: Delayed compaction for the LSM-tree. *International Journal of Parallel Programming* 45, 6 (2017), 1310–1325.

[19] William Pugh. 1990. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM* 33, 6 (1990).

[20] Russell Sears and Raghu Ramakrishnan. 2012. bLSM: a general purpose log structured merge tree. In *Proceedings of the 2012 International Conference on Management of Data (SIGMOD)*. ACM, 217–228.

[21] Dejun Teng, Lei Guo, Rubao Lee, Feng Chen, Siyuan Ma, Yanfeng Zhang, and Xiaodong Zhang. 2017. LSbM-tree: Re-enabling buffer caching in data management for mixed reads and writes. In *Proceedings of the 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 68–79.

[22] Lei Yang, Hong Wu, Tieying Zhang, Xuntao Cheng, Feifei Li, Lei Zou, Yujie Wang, Rongyao Chen, Jianying Wang, and Gui Huang. 2020. Leaper: A learned prefetcher for cache invalidation in LSM-tree based storage engines. *Proceedings of the VLDB Endowment* 13, 11 (2020), 1976–1989.

[23] Zhou Zhang, Peiquan Jin, Xingjun Hao, Ruicheng Liu, Xiaoliang Wang, and Shouhong Wan. 2019. RadixKV: A memory efficient and high performance key-value store. In *Proceedings of the 21st IEEE International Conference on High Performance Computing and Communications (HPCC)*. IEEE, 2774–2781.