CruiseDB: An LSM-Tree Key-Value Store with Both Better Tail Throughput and Tail Latency

Junkai Liang
Key Laboratory of DEKE, MOE, China
School of Information, Renmin University of China
Beijing, China
liang_jk@ruc.edu.cn

Yunpeng Chai
Key Laboratory of DEKE, MOE, China
School of Information, Renmin University of China
Beijing, China
ypchai@ruc.edu.cn

2021 IEEE 37th International Conference on Data Engineering (ICDE) | 978-1-7281-9184-3/20/$31.00 ©2021 IEEE | DOI: 10.1109/ICDE51399.2021.00094

***Abstract*—Due to excellent performance, LSM-tree key-value stores have been widely used in various applications in recent years. However, LSM-tree's inherent batched data processing approach makes it suffer from poor SLA behaviors, such as a very unstable throughput and high tail latency. Unlike the I/O isolation or prioritization methods that cannot solve the SLA problem thoroughly, we have designed and implemented a new SLA-oriented LSM-tree KV store, i.e., CruiseDB, to solve both the essential and the direct SLA problems of LSM-tree KV stores by introducing an adaptive admission mechanism and improving the LSM-tree structure. According to reliable estimation of the service capacity of the LSM-tree, CruiseDB adaptively admits only an appropriate number of user requests to enter the LSM-tree memory buffer in unit time and removes the internal roadblocks of the request processing, with the advantages of preventing the write stall phenomenon, which leads to SLA declines. CruiseDB can promote the guaranteed throughput by 2.08 times on average compared with the state-of-the-art LSM-tree or B-tree KV stores.**

***Index Terms*—LSM-tree, key-value, SLA, tail throughput, tail latency**

I. Introduction

Key-value (KV) data are suitable to support many types of applications due to simplicity, reliability, and high adaptability, such as big data management [1], social networking [2], graph processing [3], and machine learning [4]. The percentage of writing operations in many applications has increased rapidly in recent years [5]. Due to the much higher write performance compared with traditional techniques, the log-structured merge (LSM) tree KV stores are therefore widely used, e.g., in BigTable [6], HBase [7], Cassandra [8], CockroachDB [9], and TiDB [10].

Motivation. However, the Service Level Agreement (SLA) behaviors (e.g., the stability of the system throughput and the tail latency) of LSM-tree KV stores are typically low, because of the inherent batched data processing manner of the LSM-tree (see Section III-A for the detailed evaluation). In LSM-tree KV stores, when the memory buffer is full or one of the layers is so large that the performance is severely influenced, the LSM-tree must perform a *write stall*, which means forcing the user requests to slow down or even to be suspended and is the *direct* reason of the SLA declines. But the *essential* reason for the low SLA lies in that the LSM-tree always puts the arrived user requests into the memory buffer much more than its realistic I/O processing capacity, so a periodical slow-down or stop is needed to digest the excessive requests in the memory.

Although there is a lot of average-performance-oriented optimization work [11]–[21] on the software or hardware-software co-design aspects for LSM-tree KV stores, only a few solutions have been suggested to concentrate on promoting SLA. And the current SLA-aware solutions, e.g., Auto-tuned RocksDB [22] and SILK [23], [24], only isolate or prioritize the users' requests and the background operations, which cannot solve both the above *essential* and *direct* SLA problems of LSM-tree KV stores thoroughly.

Solution. In this paper, therefore, we have designed and implemented an SLA-oriented LSM-tree KV Store, i.e., CruiseDB, which can make the real-time system throughput much more stable and achieve lower tail latency based on the similar average performance. Motivated by the admission mechanisms [25], [26] to improve quality-of-service (QoS), CruiseDB introduces an Adaptive Admission module to reliably estimate the service capacity of the LSM-tree and to prevent from accepting excessive user requests (i.e., solving the *essential* SLA problem). Furthermore, we have proposed an improved Simplified Practical LSM-tree (SP-LSM-tree) structure to make the inner data processing of the LSM-tree more fluent and efficient (i.e., reducing the write stalls that cause the *direct* SLA problem).

CruiseDB was implemented based on a widely used industrial grade open-source LSM-tree KV store, i.e., RocksDB [5] developed by Facebook. The evaluation based on the famous KV benchmark YCSB [27] shows that CruiseDB can promote the 99-percentile tail throughput (see Definition III.3 in Section III-A) by 2.08 times on average compared with the state-of-the-art LSM-tree or B-tree KV stores for both the point accessing and the range query scenarios. At the same time, CruiseDB can reduce the maximum 99-percentile tail latency per second by 19.6% $\sim$ 58.9%.

The contributions of this paper include:

1) We have performed a comprehensive and in-depth assessment and review of LSM-tree KV stores' SLA issues. Also, we have discovered some misunderstandings about improving SLA, e.g., simply isolating or prioritizing I/O operations, which cannot fix the SLA problems or even make them worse.

978-1-7281-9184-3/21/$31.00 ©2021 IEEE
DOI 10.1109/ICDE51399.2021.00094

2) We have suggested an important SLA metric, i.e., *Tail Throughput*, to measure both the performance and the stability of the system throughput. It is more comprehensive to combine both tail throughput and tail latency to evaluate the SLA behaviors of systems.

3) We have designed and implemented an SLA-oriented LSM-tree KV store, i.e., CruiseDB, which can achieve a much higher tail throughput and lower tail latency compared with the state-of-the-art LSM-tree or B-tree KV stores.

4) CruiseDB has been implemented based on an industrial grade open-source LSM-tree KV store, i.e., RocksDB, which has around 300,000 LOC. Therefore, CruiseDB is of great practical importance to be adopted in enterprise environments with both high performance and high SLA.

The rest of this paper is organized as follows. The background of the LSM-tree is introduced in Section II and Section III provides an assessment and in-depth analysis of its SLA behavior. The SLA enhancement methodology in our CruiseDB and the contrast with existing solutions are described in Section IV, accompanied with comprehensive statements in Section VI and V, respectively, on the two main components of CruiseDB, i.e. Adaptive Admission and Simplified Practical LSM-Tree. The following Section VII introduces the CruiseDB implementation details, while the evaluation is given in Section VIII. Finally, after the related work presented in Section IX, we conclude this paper in Section X.

II. Background of LSM-Tree KV Stores

A. Principles of LSM-Tree KV Stores

Definition II.1. ***(LSM Tree)*** *A data structure consisting of multiple data sets, i.e., C_i ($0 \leq i \leq N$), where C_i is a data structure in order and $sizeof(C_i) < sizeof(C_{i+1})$.*

Definition II.2. ***(Compaction)*** *The behavior of merging all or part of the data in C_i to the next data set C_{i+1} and maintain the order of all the data in C_{i+1} after merging.*

Definition II.3. ***(Write Amplification)*** *The phenomenon that the practical I/O amount is larger than that from users, which comes from two aspects in an LSM-tree: 1) Writing the same data several times into C_0, C_1, C_2, etc., successively. 2) When merging C_i to $C_i + 1$, additional I/Os are introduced because we have to load C_{i+1} into the memory and then write all the sorted data into the new C_{i+1} in order to make data sorted.*

The structure of an LSM-tree is shown as Fig. 1. When a new written key-value pair arrives, it will be inserted into the first and the smallest set C_0. Since C_0 is very small, the sorting behavior is typically very rapid after inserting the new data. Then the data will be placed in C_1, C_2, etc., respectively, through compaction operations, which keeps the data ordered to accelerate reading and also can decrease the storage usage by merging different versions of the same data (i.e., the same key in a key-value store).

For a read request, we may need to read several times from an LSM-tree, because the target data may be located in any level of the LSM-tree. We normally read C_0 first, and then C_1, C_2, etc., and we do not need to read more when any of the data set contains the target data, since the upper level contains the more recent version of the same data.

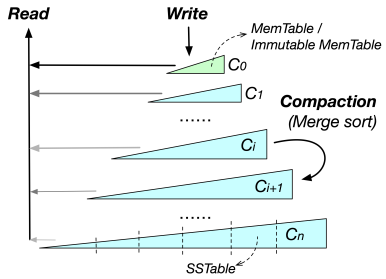

Fig. 1. The structure of an LSM-tree and its read/write request processing.

Definition II.4. ***(MemTable)*** *C_0 usually locates in memory; it is called* MemTable*. All the data in a MemTable is in strict order for fast reading.*

Definition II.5. ***(Immutable MemTable)*** *A* MemTable *has limited capacity; when it is full, it will be transformed into an* Immutable MemTable*, which cannot be modified and is waiting to be written into persistent storage.*

Definition II.6. ***(SSTable)*** *A data block or file that preserves key-value pairs in strict order of keys stored in I/O devices; it is typically part of one level in an LSM-tree.*

Because each level in an LSM-tree may become very large, in practical KV systems, each level is usually divided into multiple *SSTables* to read data conveniently and limit the coverage area of a compaction process.

B. Memory-I/O Boundary in Practical LSM-Tree KV Stores

In practical LSM-tree KV stores, DRAM is fast, but it is volatile and has higher costs and limited storage capacity; I/O devices (e.g., Flash-based SSDs and magnetic disks) are relatively cheaper, larger, and non-volatile, but their performance is relatively lower than DRAM. Therefore, how to use the two kinds of storage devices well is important, especially how to process the memory-I/O boundary for an LSM-tree.

Generally, C_0 in an LSM-tree is usually placed in DRAM, since data are written into C_0 first and we also read C_0 first when processing a read request. Due to the restricted capacity of DRAM, the other levels (i.e., $C_1 \sim C_N$) are normally located in the I/O devices. If we just adopt such a common-sense data distribution for an LSM-tree plotted as Fig. 1, a serious issue will occur, i.e., user requests will sometimes be blocked due to the lack of memory. For example, because of the LSM-tree write amplification phenomenon, when we perform a compaction operation between C_0 and C_1 in order to release the occupied memory space, a lot of slow I/O operations have to be performed. The I/O operations are much slower than the memory access, so they will block users from writing data.

Therefore, today's practical LSM-tree (i.e., **P-LSM-tree**) KV stores (e.g., RocksDB [28]) have introduced some major changes based on the standard LSM-tree structure: 1) As Fig.

2 illustrates, multiple MemTables or Immutable MemTables are typically deployed in memory; multiple users can write data into MemTables in parallel. Of course, key-value pairs in each MemTable are sorted, but there is not a guaranteed order among the data from different MemTables. 2) A particular Level 0 (i.e., L_0) is designed to be located in the I/O devices. L_0 consists of several *SSTables*, which are the results of Immutable MemTables' persistence. Thus, similar to the MemTables in the memory, there is no global order for key-value pairs among different *SSTables* of L_0; only the data within a single *SSTable* are sorted. In this case, no write amplification will be caused by the persistence of Immutable MemTables to L_0, and the process will be significantly boosted to release the precious memory space in time. This new action is called *Flush*.

Definition II.7. ***(Flush)*** *The process of writing an in-memory Immutable MemTable into the LSM-tree's Level 0 (L_0) located in I/O devices; the process will NOT cause write amplification.*

However, the introduction of L_0 in practical LSM-tree KV stores also brings a side effect, i.e., the read amplification is aggravated in an LSM-tree. The reason lies in the lack of order between the *SSTables* in L_0. So we cannot quickly locate the target key; instead we have to read all the *SSTables* in L_0 one by one. Consequently, the maximum number of *SSTables* in L_0 is strictly limited (e.g., 20). If the number of L_0's *SSTables* is close to or exceeds the threshold, for the sake of guaranteeing read efficiency, we have to limit the writing throughput or even stop writing entirely.

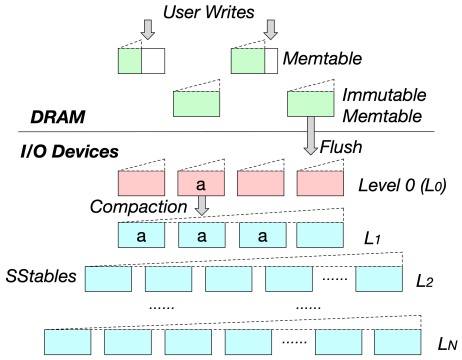

Fig. 2. The structure of typical practical LSM-tree (**P-LSM-tree**) KV stores.

III. SLA Analysis of LSM-Tree KV Stores

Although the average performance of LSM-tree KV stores is good, their SLA face severe challenges. In this section, we will provide an SLA evaluation in Section III-A and an in-depth analysis of the SLA challenges in Section III-B.

Definition III.1. ***(SLA)*** *The Service Level Agreement (SLA) refers to a commitment that a service provider guarantees some particular aspects of the service (e.g., quality, performance, responsibilities, availability, etc.) to a client.*

Definition III.2. ***(Tail Latency)*** *The tail latency is the small percentage of response time that is close to the longest compared with the overall response time.*

A. SLA Evaluation

We evaluated the SLA metrics of a commonly used LSM-tree KV store, i.e., RocksDB [28], powered by the famous YCSB benchmark [27]. The number of parallel threads of YCSB is configured to 100.

Figs. 3 and 4 show the fluctuations of the throughput over each second and the 99-percentile tail latency over each second, respectively, under a YCSB A-variant workload (i.e., 50% *insert* and 50% *get*). Obviously, the throughput drops dramatically and then returns to the original value, and so on. And the 99-percentile tail latency also tends to have regularly increasing and decreasing changes that almost keep the same pace with the throughput variations. The results indicate that the typical LSM-tree KV store (e.g., RocksDB) cannot maintain a stable system throughput and stable tail latency after the memory buffer is fully filled in the first phase.

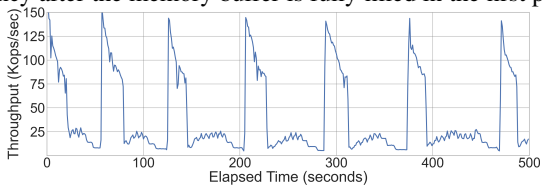

Fig. 3. The throughput fluctuation of RocksDB under YCSB A-variant.

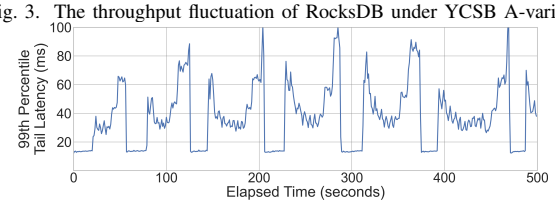

Fig. 4. The tail latency fluctuation of RocksDB under YCSB A-variant.

Definition III.3. ***(Tail Throughput)*** *The tail throughput means the small percentage of the lowest throughput, out of the throughput values over all the small time periods during the service time, reflecting the guaranteed throughput of any small time period that a service can promise to users.*

Similar to the concept of the tail latency, we propose a new metric, i.e., the *tail throughput*, to measure the SLA of key-value stores in the aspect of throughput stability. We assume the time window of the throughput calculation is Δt (e.g., one second), the total service time is T, which contains N time windows of Δt, and the throughput of each time window is $th_i, i \in [0, N)$. If the $P\%$ (e.g., 99%) highest throughput among the N ones is considered as the guaranteed throughput for users, the tail throughput during the service time T, i.e., Th_{tail}, can be calculated as Eq. 1.

$$\begin{aligned} L &= descending\_array(\{th_i\}), i \in [0, N); \\ Th_{tail} &= L[\lceil N \times P\% \rceil] \end{aligned} \tag{1}$$

As Fig. 3 plots, the 99-percentile tail throughput of the measured 500 seconds is only 5329.9 ops/sec, while the average throughput of the 500 seconds is up to 41241.6 ops/sec, reaching a gap of 7.74 times. Therefore, although the LSM-tree KV stores have good average performance, their SLA metrics are poor, including both the tail throughput and the tail latency. This makes the traditional LSM-tree KV stores bring great troubles to the upper applications.

B. SLA Problem Analysis

In this part, we will give an in-depth analysis of the SLA problems of the traditional LSM-tree KV stores. The sharp throughput decline and the high tail latency when processing writing requests are triggered by the following two reasons.

1) The direct reason is the *write stalls* that occur when the memory buffer is full or the read amplification is too high. Recall Section II-B that data are first written into *MemTables* in the memory and then flushed to Level 0 located in I/O devices for the LSM-tree. When the memory buffer is full and we cannot flush the Immutable MemTables in time, there will be no memory space left for accepting new data. In this case, we have to stall the writing requests, i.e., *write stall*, causing a rapid decline of the throughput and an increment of the tail latency (see Figs. 3 $\sim$ 4).

The other case that may also lead to *write stalls* happens when there are too many *SSTables* in Level 0. Since there is not a global order for all the *SSTables* in Level 0, we have to read them one by one when processing a reading request, i.e., with linear complexity. If L_0 is too large, the read amplification will not be acceptable in practical applications, most of which have the performance and SLA requirements for both reading and writing. In this case, the incoming writing requests should also be blocked and the compaction will be performed to reduce the size of Level 0.

2) The essential reason is the absence of the admission mechanism. Although the direct reason for the SLA declines is the *write stalls* caused by the lack of memory or the too large Level 0, the essential reason for the periodical changes of both throughput and tail latency lies in the fact that too many writing requests are permitted to enter the LSM-tree KV store and this exceeds the system's processing capability. When the excess written data have been accumulated, the system has to stall the writing requests for a while to "digest" these written data through the compaction.

In fact, the essential reason for the poor SLA is more important compared with the direct problem. Without an appropriate admission, even if the write stall phenomenon is optimized in the LSM-tree, SLA will also significantly decline when the system is overloaded.

IV. Methodology of Designing an SLA-Oriented LSM-Tree KV Store

Aiming at solving both the *essential* and the *direct* SLA problems of LSM-tree KV stores, we have designed and implemented a new SLA-oriented LSM-tree KV store, i.e., CruiseDB, which can supply a stable throughput and low tail latency for service tenants. In this section, the overview of CruiseDB will be given in Section IV-A, and some further comparisons with current SLA-aware LSM-tree KV solutions will be elaborated in Section IV-B.

A. Overview of CruiseDB

The objective of CruiseDB is an SLA-oriented in-storage KV store with the support for complete KV functions including both point accessing and range queries. Furthermore, among the various SLA metrics, a stable throughput (i.e., high *tail throughput*) is the most important for the majority of applications, because LSM-tree KV stores are usually performed as the underlying data storage for many applications and the low throughput will block the running of the upper applications seriously.

The principle of designing an SLA-oriented LSM-tree KV store is to solve all the SLA problems discussed above including the *essential* problem of accepting too many user requests and the *direct* problem of *write stalls*. An SLA-oriented system follows the "Liebig's law", i.e., a barrel's volume is decided by the shortest plate that consists of the barrel. Only solving part of the SLA problems may not have obvious effects on improving SLA. We need to detect all the main problems that cause the decline of SLA and give a complete solution to eliminate all the main negative factors.

According to this principle, we've designed a new SLA-oriented LSM-tree KV store called *CruiseDB* to achieve both higher tail throughput and lower tail latency with similar average performance compared with the performance-first LSM-tree KV stores. Two key components are designed in CruiseDB, i.e., the Adaptive Admission and the Simplified Practical LSM-tree (SP-LSM-tree), which two solve the *essential* and the *direct* SLA problems respectively.

Traditionally, an LSM-tree KV store tends to accept more user writes into the memory buffer than its practical processing capability, i.e., the I/O devices' bandwidth to perform the foreground and the amplified background I/Os, for larger write batches and higher processing efficiency. However, the way to stop the excessive data input, i.e., *write stall*, has great negative effects, i.e., reducing the throughput seriously.

This is the ***Essential SLA problem*** of LSM-tree KV stores. Only accepting a reasonable amount of user writing requests gracefully is the right solution to the *essential* SLA problem, i.e., we need an accurate and effective admission control mechanism for LSM-tree KV stores. However, the inner status of an LSM-tree is extremely complicated and flexible, and the features of user access also change all the time, so it is difficult to estimate the accurate service capacity of an LSM-tree KV store. Therefore, we have proposed an intelligent and adaptive admission strategy to solve this problem, which will be elaborated in Section V.

In order to solve the ***Direct SLA problem***, we have made an improvement based on the structure of practical LSM-trees, i.e., SP-LSM-tree, to cooperate with the adaptive admission scheme and eliminate the *write stalls*. First, SP-LSM-tree removes L_0 introduced in P-LSM-tree and the corresponding *flush* operations thoroughly, simplifying P-LSM-tree and eliminating the *write stalls* caused by L_0. Accordingly, SP-LSM-tree introduces a larger memory buffer because the data output may be slower due to removing L_0. In the whole, by eliminating the *flush* operations, SP-LSM-tree has positive effects on the overall performance since it reduces the practical I/O amplification in total. The design details and discussions about SP-LSM-tree can be found in the following Section VI.

TABLE I
COMPARISON OF EXISTING SOLUTIONS TO THE ESSENTIAL AND THE DIRECT SLA PROBLEMS OF LSM-TREE KV STORES.

Solution	Principle	Essential Problem	Direct Problem	
			Memory	L_0
TRIAD	Keeping popular KV pairs longer in the memory buffer to merge overwritten data more	×	✓	×
Auto-tuned RocksDB	I/O bandwidth isolation between user requests and the background *flush* / *compaction*	×	×	×
SILK	Prioritized I/Os (i.e., user reqs > *flush* > L_0 to L_1 *compaction* > other *compaction*)	×	✓ / ×	✓ / ×
CruiseDB	Adaptive admission for *Essential SLA Problem* and SP-LSM-tree for *Direct SLA Problem*	✓	✓	✓

B. Comparison with Existing SLA-Aware Solutions

Although many approaches have been proposed to promote the performance of LSM-tree KV stores, there are only a few existing solutions aiming at improving SLA (e.g., TRIAD [29], Auto-tuned RocksDB [22], and SILK [23], [24]). In this part, we will give a detailed analysis of the SLA-improvement effects of these solutions based on the above two kinds of SLA problems (i.e., *Essential Problem* and *Direct Problem*).

Table I summarizes the principles of these current solutions and their contributions to solving the *Essential* and *Direct Problems* of SLA. In Table I, the symbol ✓ indicates the solution is effective for the SLA problem in this aspect, and the symbol × represents no effects or even negative effects.

TRIAD leverages skew for frequently-accessed data in the memory to reduce the amount of I/O operations due to more write hits in MemTable. This strategy is equivalent to using larger effective memory space. Thus TRIAD can put off the *write stalls* caused by the full memory buffer. As Table I indicates, TRIAD helps alleviate the pressure of the memory, but does not contribute to L_0 and the *Essential Problem*.

Auto-tuned RocksDB is a variant of RocksDB aiming to protect the processing speed of foreground user requests by setting a throughput limit for background operations (i.e., *flush* and *compaction*). Furthermore, the throughput limit is set automatically according to the known performance of underlying I/O devices and the throughput requirements of users. Because the foreground requests have independent I/O bandwidth and will be less blocked by the background I/Os, the tail latency can be improved to some extent. However, Auto-tuned RocksDB cannot solve either the *Essential* or the *Direct SLA Problems*; it usually has a slower speed to clear the data in memory and L_0 due to lower I/O bandwidth for *flush* and *compaction* in most cases. Therefore, Auto-tuned RocksDB cannot alleviate the serious throughput fluctuations of LSM-tree KV stores (see Section VIII-B for more experimental results).

SILK introduces a new I/O scheduler with multiple I/O priorities, i.e., user requests > *flush* > L_0 to L_1 *compaction* > other *compaction*. The I/O priority setting is beneficial to releasing space of the memory buffer and L_0; this is good for solving the *Direct SLA Problem*. However, because user requests have the highest priority, more writing data are allowed to enter the system, which may be much higher than the system's processing capability and worsen the *Essential SLA Problem*. Meanwhile, both the *flush* and the L_0 to L_1 *compaction* may be slowed down due to processing more user requests; this is also negative for the *Direct SLA Problem*. Therefore, SILK cannot improve SLA effectively compared with original RocksDB (see VIII-B for more).

V. ADAPTIVE ADMISSION

The objective of the admission in CruiseDB is to appropriately limit the speed of data input. Although the memory buffer can endure a very high throughput of data writing when it has free space, the practical data processing capability is determined by the processing speed of the amplified I/O operations due to the LSM-tree's *compaction*. If the admission is too loose, the LSM-tree will consume all the memory buffer space quickly, which leads to a standstill of the system or a too large LSM-tree level causing much higher I/O amplification rate and lower performance. On the contrary, although a too strict admission may produce high SLA, it will lead to a waste of system processing capability.

Due to the complexity of LSM-tree KV stores and the constant changes of user-access features, it is difficult to measure or calculate an accurate and appropriate stable system throughput limit for an LSM-tree KV store. Therefore, in CruiseDB, we have introduced a feedback controller-based adaptive admission mechanism to update the limit dynamically according to the latest system status (i.e., Section V-A). The method of estimating LSM-tree KV stores' processing capability will be introduced in Section V-B; the specified admission control is operated based on the token-bucket method [30] and will be presented in Section V-C.

A. Feedback Controller-based Adaptive Admission

The I/O processing capability of an LSM-tree KV store is not a stable value. For example, when the number of the LSM-tree's levels increases, the I/O amplification may also increase, leading to a lower throughput; when the ratio between reading and writing changes, the overall user throughput will also change. Therefore, the adaptive admission should adjust the user writing limit dynamically to get it close to the practical real-time processing capability of the LSM-tree.

CruiseDB follows the classical control theory to perform an adaptive admission. As Fig. 5 shows, the utilization rate of the LSM-tree memory buffer, i.e., its watermark ranging from 0% to 100%, is chosen as the target of the feedback controller. The ideal target of the controller is to achieve a 50% watermark, which has enough free space to endure the most sudden changes of system status and also has good resource utilization. The reason why we do not adopt a target watermark higher than 50% lies in that the LSM-tree is complex and its response to adjustments is slow in many conditions. When the target watermark is too high, even if we lower the data input speed, the watermark may keep increasing for some time and cause the *write stall*.

Fig. 6 plots the detailed work procedure of the feedback controller we use in CruiseDB. First, we utilize a variant of

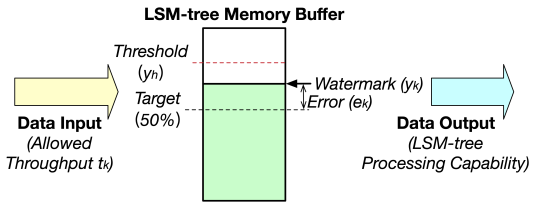

Fig. 5. The adaptive admission adopts the watermark of the LSM-tree memory buffer as the adjustment target.

the Proportional-Integral-Derivative (PID) fixed gain controller [31]. The adaptive admission module sets the allowed writing throughput at the time point $k-1$, i.e., t_{k-1}, of the LSM-tree, and we observe the corresponding watermark of the memory buffer in the next time point k, i.e., y_k. The error e_k between y_k and the target value 50%, i.e., $y_k - 50\%$, is passed to the adaptive admission module (i.e., the feedback controller), which will calculate the new user throughput limit t_k according to Eq. 2.

$$t_k = \begin{cases} t_{k-1}(1 - K_1 e_k - K_2(e_k - e_{k-1})), & y_k < y_h \\ T_c, & y_k \geq y_h \end{cases} \quad (2)$$

Besides the last round throughput limit t_{k-1}, the generation of t_k is also affected by the error e_k (i.e., $y_k - 50\%$) and the trends of error changes (i.e., $e_k - e_{k-1}$). In Eq. 2, K_1 and K_2 are positive weights. For example, when y_k is larger than 50% and maintains growth, both $K_1 e_k$ and $K_2(e_k - e_{k-1})$ are positive, so we will set a tighter throughput limit than the last period to lower the memory buffer watermark. The impacts of different K_1 and K_2 settings are evaluated in Section VIII-F.

The initial value of t_k can be set to the average throughput practically measured in the initial phase without an admission. And if the watermark gets too high (e.g., larger than 80%), in order to avoid the *write stall*, we do not use the feedback controller to determine the next-round throughput limit, but set it to a conservative estimation of the system performance, i.e., T_c, which is determined in Section V-B.

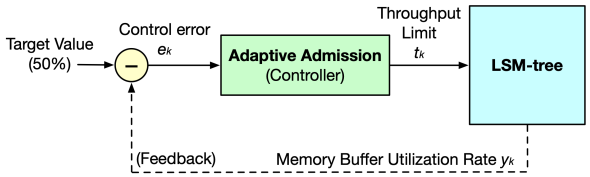

Fig. 6. Diagram of the feedback controller used in the adaptive admission.

B. Conservative Estimation of LSM-tree Processing Capacity

Essentially, the performance of LSM-tree KV stores is determined by the I/O processing. In order to give a conservative estimation of the LSM-tree's throughput, some important metrics of the system status need to be recorded in the past time period, including the reading percent (i.e., P_r) and the writing percent (i.e., P_w), the total read amplification rate when reading data from the LSM-tree (i.e., R_{RA}), the total write amplification rate of writing data into all LSM-tree levels (i.e., R_{WA}). Supposing the read and the write throughput of the I/O device is T_R^{IO} and T_W^{IO}, respectively, we can get Eq. 3 and calculate T_c, because in unit time, three kinds of I/Os will be accomplished including the amplified user reading, the amplified write operations in compaction, and the related read ones in compaction of the same amount.

$$\frac{T_c \cdot P_r \cdot R_{RA}}{T_R^{IO}} + \frac{T_c \cdot P_w \cdot R_{WA}}{T_W^{IO}} + \frac{T_c \cdot P_w \cdot R_{WA}}{T_R^{IO}} = 1 \quad (3)$$

So the ratio of reading and writing amounts practically happened on the I/O device is $(P_r \cdot R_{RA} + P_w \cdot R_{WA}) : (P_w \cdot R_{WA})$. We know that many new storage media do not have symmetric reading and writing performance, e.g., Flash, Phase-Change Memory, etc. Therefore, we should get a reading and write performance mapping for different reading/writing ratios beforehand through some measurements on the specified I/O device. Each time we need to calculate T_c according to Eq. 3, we can get relatively accurate values of T_R^{IO} and T_W^{IO} from the mapping.

C. Token-Bucket-based Admission Control

The adaptive admission is executed based on the token-bucket method. As Fig. 7 plots, a new Request Waiting Queue (RWQ) for temporal storage and a token-bucket have been added in CruiseDB. All the user requests are put into RWQ first; after consuming one token, one request will be processed by the LSM-tree KV store. In addition, the tokens are generated evenly at the speed of the throughput limit t_k, which is dynamically determined by the adaptive admission.

When there are no tokens left in the bucket, some user requests will be suspended in RWQ for a while, i.e., for less then $1/t_k$ second. But this mechanism can avoid the case of a full memory buffer, in which the upcoming requests will be blocked for a long time and both high average latency and high tail latency will be caused. Besides, the additional overhead introduced by RWQ and the token-bucket is negligible, because their processing is very simple.

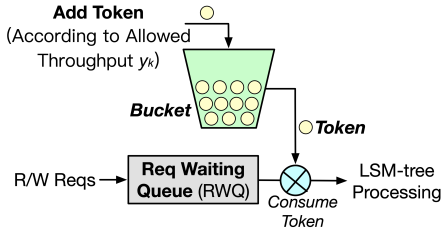

Fig. 7. Diagram of the token-bucket-based admission control.

VI. Simplified Practical LSM-Tree

Although the above adaptive admission can avoid too many user requests entering the KV store, the SLA may also decline if the inner execution of the LSM-tree is not smooth and efficient. The most important thing is to prevent the *write stalls* caused by either lack of memory or too many *SSTables* in L_0. Therefore, in CruiseDB, we've reconstructed the LSM-tree to solve the two specified problems of L_0 and memory, respectively introduced in Section VI-A and VI-B.

A. Solving the SLA Problem Caused by L_0

L_0 is not a part of a standard LSM-tree due to the lack of an order among different *SSTables* in this level; it is introduced by practical LSM-tree KV stores to accelerate the recycling

of the memory buffer space (i.e., through *flush* without write amplification). Because it is lacking in an order, the size of L_0 is strictly limited to avoid too slow reading requests. In CruiseDB, we argue that L_0 is unnecessary and even harmful; it should be removed from the LSM-tree.

We have proposed a new simplified practical LSM-tree, i.e., **SP-LSM-tree**, plotted as Fig. 8. First, L_0 is eliminated in SP-LSM-tree, so *write stalls* cannot be triggered by L_0. Second, since there is no L_0, the *flush* operations between the memory buffer and L_0 are also eliminated. Thus the data flow between the memory buffer and the I/O devices carries out through *compaction*, the same as that between different LSM-tree levels located in I/O devices.

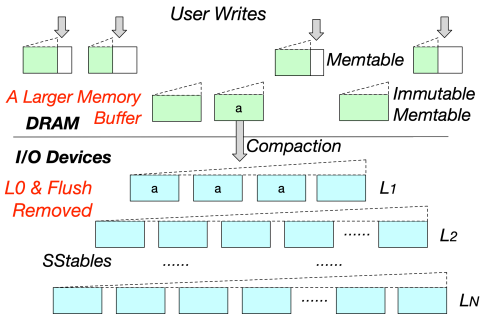

Fig. 8. The structure of the simplified practical LSM-tree (SP-LSM-tree).

Of course, unlike *flush*, *compaction* operations introduce write amplification. However, this will not impede the procedure of releasing the memory buffer space; instead, the practical I/O processing capability of SP-LSM-tree is higher than P-LSM-tree from an overall perspective. Fig. 9 gives the data traffic models between the memory buffer and the I/O devices of both P-LSM-tree and SP-LSM-tree.

Theorem VI.1. *The I/O processing rate, i.e. the reasonable data input speed, of SP-LSM-tree is higher than P-LSM-tree.*

Proof (sketch): We assume the underlying I/O devices' bandwidth is B_{IO} and the accumulated I/O amplification rate of writing data from L_0 (for P-LSM-tree) or memory (for SP-LSM-tree) to the last level is R_A. For P-LSM-tree, assuming the users' writing data amount is W_1, F should be equal to W_1 without any write amplification. Thus the total I/O amount is $F + R_A \times W_1$, and the consumed time t to accomplish all the I/O operations should be $(F + R_A \times W_1)/B_{IO}$.

For SP-LSM-tree, the flush overhead F is eliminated since L_0 is removed. Assuming the users' writing data amount is W_2, the total I/O amount is $R_A \times W_2$, less than P-LSM-tree's one. Although writing in-memory data into L_1 triggers write amplification, the amplified I/O amount is already counted in $R_A \times W_2$. Therefore, in the same consumed I/O time (i.e., t), the writing data amount of SP-LSM-tree (i.e., W_2) should be larger than that of P-LSM-tree (i.e., W_1). □

B. Solving the SLA Problem Caused by the Memory Buffer

The adaptive admission mechanism of CruiseDB only allows a reasonable amount of user requests to enter the LSM-tree according to the practical processing capability. Thus in order to avoid the *write stalls* caused by the memory buffer,

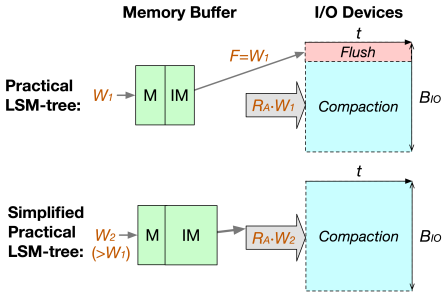

Fig. 9. The data traffic models of P-LSM-tree and SP-LSM-tree.

we should optimize the data output scheme and the temporal storage of the memory buffer as follows.

*1) **The data output optimization of the memory buffer**:* According to Theorem VI.1, we know that the average I/O processing capability of SP-LSM-tree is higher than P-LSM-tree. However, the practical execution order of the I/O operations is also an important factor to influence the data output speed of the memory buffer. If the *compaction* I/Os between the memory buffer and L_1 are not performed in time, the memory buffer can also be exhausted.

In CruiseDB, we have implemented a prioritized I/O scheduler to arrange the execution order of all the I/Os in the system. The sensitive I/O operations that are on the critical path of processing requests or free memory directly, such as foreground user requests, log writing, and the *compaction* I/Os between the memory and L_1, have higher priority. More implement details can be found in the *I/O Scheduler* part of Section VII.

Although the I/O priority of the *compaction* from the memory to L_1 is high, we do not always choose it as the *compaction* task first of all in CruiseDB. Instead, the level whose practical size is the farthest away from the expected one is chosen first for *compaction* to reduce its size. Specifically, if a single LSM-tree level is more than 5 times larger than its expected size, the *compaction* of this level should be scheduled prior to that between the memory and L_1. This can avoid the too large I/O amplification rate or the decline of the average I/O processing capability of the system, ensuring a high data output speed.

*2) **Enlarging the memory buffer**:* For SP-LSM-tree, a larger memory buffer is required compared with P-LSM-tree. Because the data output of the memory buffer is performed through *compaction* with write amplification instead of *flush*, SP-LSM-tree needs larger temporal storage space due to a relatively slow memory-recycling speed.

First, a larger memory buffer is reasonable for today's hardware configurations of typical servers. Traditional LSM-tree KV stores are usually designed to work with a small DRAM configuration to match various kinds of applications and environments. For example, the default configuration of RocksDB only demands four *MemTables* or *Immutable MemTables*, i.e., 256 MB totally. In CruiseDB, we enlarge the maximum number of allowed *MemTables* and *Immutable MemTables* to 12 at most, i.e., 0.75 GB, which is not a burden for today's mainstream servers.

Second, there are many benefits when a larger memory buffer is deployed: (1) A larger memory buffer can promote the average performance of a KV store and avoid the decline of SLA by temporally storing more user requests in the memory and merging them into I/O devices later when the system is relatively idle. (2) A larger memory buffer can enhance the fault tolerance of the admission since any admission strategy cannot guarantee an absolutely accurate speed limit. (3) Another benefit of a larger memory buffer is to increase the reading performance due to more memory hits.

In addition, we need to store more WAL data in persistent devices for the data in the memory buffer in case of crashes and data loss. The introduced additional overhead is a larger WAL file in I/O devices, which is almost negligible.

VII. System Implementation

CruiseDB was implemented based on RocksDB [28] by adding or modifying more than 1000 LOC in C++, including an *I/O Scheduler*, a *Request Rate Limiter* and an *Adaptive Controller*. We chose RocksDB as the base of CruiseDB because it is an excellent and widely-used industrial grade LSM-Tree KV store. The source code of CruiseDB is now available at https://github.com/YunWorkshop/CruiseDB.

I/O Scheduler. In RocksDB, there are two priority levels for the background operations, i.e., the higher one for *flush* and the lower one for *compaction*. However, both of them use the same POSIX write API for disk I/Os and the I/O stacks of current mainstream operating systems (e.g., Linux) do not support setting the I/O priority for individual I/O requests, so the practical execution order of the I/O requests in the operating system is not guaranteed. Furthermore, as *compaction* may create a mass of I/O requests in a short time, the I/Os of *flush* which are supposed to have higher priority may be blocked when encountering the *compaction* I/Os. To solve this problem, we've added an extra I/O scheduling layer in CruiseDB to take over all the disk I/Os and re-schedule them according to the priority settings of CruiseDB.

CruiseDB has three I/O priority levels. The first level is for foreground user requests and WAL. The second one is for I/Os produced by *compaction* between memory and L_1. The third is for other *compaction* I/Os, which are scheduled only when there are no higher-level I/O requests or the waiting time has reached a threshold to avoid starvation. In addition, the granularity of *compaction* I/Os in CruiseDB is much reduced, i.e., the *compaction* I/Os are generated gradually, so the higher-priority I/Os will not be blocked by a mass of *compaction* I/Os in the I/O stack of the operating system.

With the support of I/O priority, we can improve the parallelism of *compaction* operations for higher performance. RocksDB usually has only one thread for *compaction* in order to guarantee the priority of *flush* operations. However, in CruiseDB, as L_0 and *flush* operations have been removed, *compaction* operations are always executed in parallel to improve the total efficiency. In this case, the above I/O priority setting is necessary to prevent the *compaction* between memory and L_1 from being blocked.

Request Rate Limiter. In CruiseDB, we've added a *Request Rate Limiter* which tracks and limits the rate of user requests according to the decision of the following *Adaptive Controller* module. We attach a *Request Rate Limiter* to each column family which corresponds to an LSM-Tree instance. The *Request Rate Limiter* is implemented based on the token-bucket method.

A request can take a token away immediately if there are remains in the bucket. Otherwise, it joins a queue guarded by a mutex and waits until the bucket is refilled. The bucket is refilled periodically with tokens calculated by the following *Adaptive Controller*. After that, the request at the head of the waiting queue is waked up by condition variable and then it is processed and wakes the next request up. The *Request Rate Limiter* has little effect on performance.

Adaptive Controller. In CruiseDB, we've added an *Adaptive Controller* to estimate the system's processing capability and calculate the refill rate of the *Request Rate Limiter*. It has two core modes: the Feedback mode to automatically adjust the refill rate according to the memory buffer watermark and the Conservative mode to calculate a conservative rate when the memory buffer watermark is already too high.

The *Adaptive Controller* collects several metrics to estimate its processing capability, including the average throughput, the memory buffer watermark, the reading/writing percentage, and the read/write amplification. The overhead of collecting data is small enough not to affect overall performance.

VIII. Evaluation

A. Experimental Setup

Our experiments were run on a server coupled with two Intel Xeon 2.2 GHz Silver 4114 processors, an PCIe SSD (i.e., Dell Express Flash PM1725a 1.6TB), and CentOS release 7.7.1908 (Linux 3.10.0). The memory for OS usage is limited to 8 GB by constructing the left space as a RAM disk.

We compared CruiseDB with some state-of-the-art KV storage systems constructed based on sorted indexes (e.g., LSM-tree and B tree) and supporting range queries, including RocksDB [28], Auto-tuned RocksDB [22], SILK [23], [24], and WiredTiger [32] based on B tree. Since CruiseDB requires to enlarge the memory buffer, for a fair comparison, the memory buffer size of all the other comparison targets were increased to the same size.

All experiments were run through YCSB-C [33], a C++ version of YCSB [27]. We improved it to support all the tests for RocksDB (RDB), Auto-tuned RocksDB (AT-RDB), SILK, WiredTiger, and CruiseDB, and to be capable of recording and outputting the tail latency per second and the tail throughput, which both are utilized to evaluate SLA.

YCSB was run in 100 threads in all the cases except for SILK, as multi-threading is not supported in SILK.

The YCSB workloads were composed of *insert* and *get* in point accessing cases, or *insert* and *scan* in range query cases. One *scan* operation covered 5 key-value pairs on average.

Each key-value pair had a key of 16B and a value of 1KB. In the evaluation, 10 million KV pairs were first inserted to a

KV store, i.e., the initial status of the KV store was with about 10 GB of data, and then tens of GB of data were processed.

B. Overall Results

Figs. 10 and 11 exhibit the 99-percentile tail throughput of CruiseDB and all the other systems by running 500-second YCSB workloads after inserting 10 million key-value pairs first to the KV store.

In Fig. 10, the used YCSB workloads cover a wide range of *get operation* ratios from 5% to 95%. We can get the following observations from the results:

1) CruiseDB achieves the highest tail throughput among all the KV stores in all the cases, i.e., 2.08 times higher than the second highest on average.

2) Among the LSM-tree-based solutions, AT-RDB can usually promote the tail throughput compared with original RDB through the adaptive I/O isolation between the foreground and the background I/Os; SILK is better than RDB and AT-RDB when the write percentage is no less than 50%, and the performance of SILK drops fast when the read percentage is large than 50%. The reason lies in that the static I/O priority in SILK leads to slow execution of *compaction* I/Os, so the data in the LSM-tree are not as well organized as other approaches, which slows down the speed of processing the following reading requests.

3) The B-tree solution WireTiger achieves the second highest tail throughput under almost all the workloads, because the throughput of the B-tree-based KV stores is more stable than the LSM-tree-based ones.

Fig. 11 shows the results of the range query performance when the percentage of the *scan operation* ranges from 5% to 95%. The tail throughput results are very similar to the above ones. CruiseDB also has the highest tail throughput, and the other KV stores have similar results compared with Fig. 10. Because the performance of SILK is too low when the read percentage is larger than 50%, it is not used for comparisons in the following experiments.

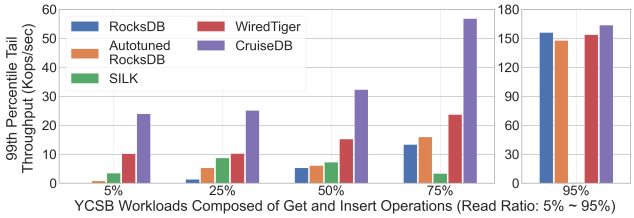

Fig. 10. Tail Throughput comparison under multiple YCSB workloads with different ratios of point accessing.

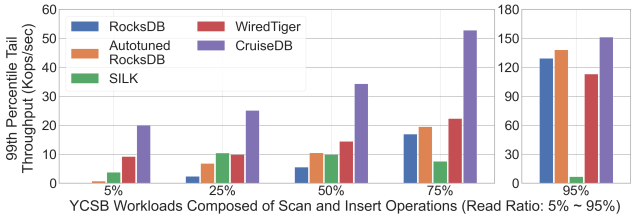

Fig. 11. Tail Throughput comparison under multiple YCSB workloads with different ratios of range queries.

C. Throughput and Tail Latency Fluctuations

Fig. 12 plots the fluctuations of throughput and 99-percentile tail latency over each second during the 500-second test under a YCSB A-variant workload (i.e., 50% *insert* and 50% *get*) for RDB, AT-RDB, WiredTiger, and CruiseDB.

First, in the aspect of the throughput, traditional LSM-tree KV stores (i.e., RDB and AT-RDB) lead to serious throughput fluctuations. The B-tree-based KV store (i.e., WiredTiger) has a much more stable throughput, but its average throughput (i.e., the green line in the figure) is obviously lower than the LSM-tree-based KV stores'. And WiredTiger's throughput shows a significant decreasing trend during the test, because the B-tree has to perform more reading and writing operations as its depth grows. Although CruiseDB is based on an LSM-tree, its throughput is much more stable (i.e., around the green line in a smaller range), indicating the adaptive admission and the SP-LSM-tree structure of CruiseDB are very effective to improve the SLA.

Second, the tail throughput (i.e., the orange line) of CruiseDB is the highest of all, 2.12 $\sim$ 6.06 times higher than the others; its average throughput is higher than WiredTiger and RDB, and only a bit lower than AT-RDB.

Finally, the tail latency fluctuations are basically the reverse of the throughput changes. The fluctuations of CruiseDB and WiredTiger are much slighter than those of RDB and AT-RDB. CruiseDB reduces the maximum tail latency (i.e., the orange line) from 108. 25 $\sim$ 109.0 ms to 44.8 ms significantly compared with RDB and AT-RDB, and it is also lower than WiredTiger (i.e., 55.7 ms). The average tail latency (i.e., the green line) of CruiseDB is 34.1 ms, better than RDB and WiredTiger, and is very close to AT-RDB (i.e., 32.0 ms).

In summary, compared with existing LSM-tree KV stores, CruiseDB can mitigate the fluctuations of both throughput and tail latency significantly, and thus promote the tail throughput (i.e., the guaranteed throughput for users) and the worst tail latency greatly. Compared with existing B-tree KV stores, CruiseDB has better performance thoroughly, i.e., both higher tail throughput and average throughput, both lower maximum and average tail latency.

D. Analysis of Adaptive Admission

Fig. 13 plots the specific progress of adjusting the dynamic throughput limit of CruiseDB according to the real-time memory watermark under the YCSB A-variant workload. The adaptive admission mechanism of CruiseDB adjusts the throughput limit every 0.4-million KV operations. This figure shows the curve comparison of both the throughput limit and the memory buffer watermark in CruiseDB. It's obvious that the two curves have a strong correlation. The throughput limit is increased when the watermark is low and is decreased when the watermark is high. At the same time, it can be found that the curve of the throughput limit is rather smooth, which means the adjustments are not excessive. Note that the memory buffer watermark has always been maintained around 50%, which is the adjustment target of the *Adaptive Admission* mechanism of CruiseDB.

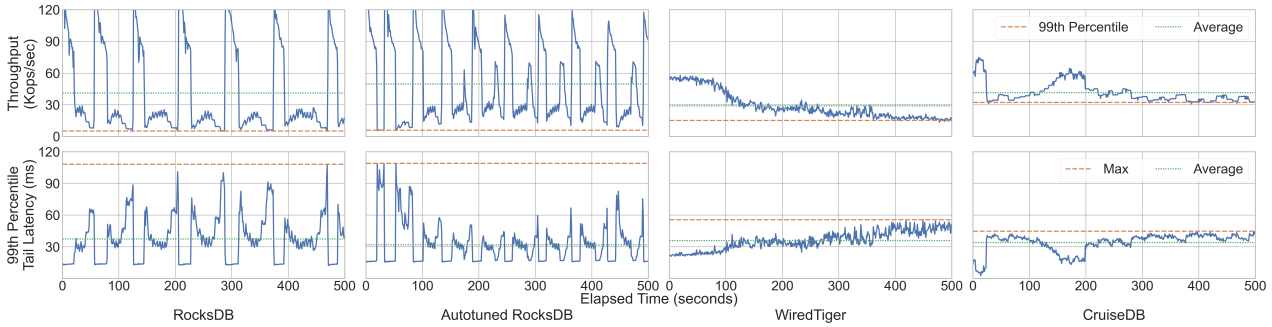

Fig. 12. Comparison of the fluctuations of the throughput and the 99th percentile tail latency under YCSB workload A-variant.

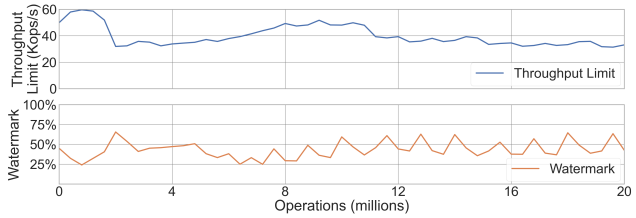

Fig. 13. Curves of the throughput limit and the memory buffer watermark.

E. Improvement of SP-LSM-tree

Fig. 14 exhibits the amount of data written to the underlying SSD in P-LSM-tree and the simplified practical LSM-tree we proposed (i.e., SP-LSM-tree) under the YCSB A-variant workload. After processing the same operations, the writing amount of SP-LSM-tree is significantly reduced compared with P-LSM-tree, e.g., a 18.7% reduction after 20 million operations are processed. It clearly shows that the write amplification in SP-LSM-tree is alleviated. The reason is that SP-LSM-tree eliminates L_0 and *flush* operations without introducing more *compaction* operations.

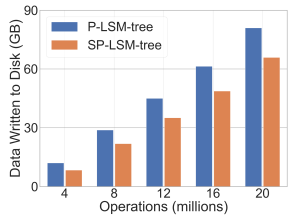

Fig. 14. The writing amount of P-LSM-tree and SP-LSM-tree.

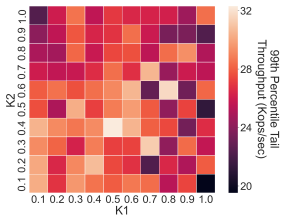

Fig. 15. Tail Throughput of CruiseDB with different K_1 and K_2.

F. Impacts of the settings of K_1 and K_2

The weights K_1 and K_2 in CruiseDB's adaptive admission are usually set empirically (see Eq. 2). In order to find the impacts of different K_1 and K_2 settings, we performed a set of 500-second experiments with different K_1 and K_2 ranging from 0.1 to 1.0, respectively, under the YCSB workload A-variant after loading 10 million 1KB key-value pairs.

As Fig. 15 plots, when K_1 and K_2 are both under 0.6, the tail throughput can maintain a high value (i.e., 28575.9 ops/sec on average). On the contrary, when K_1 or K_2 is larger than 0.6, the tail throughput decreases obviously, because the throughput admission is adjusted overly in these cases. Therefore, the general advice is to choose relatively small values for K_1 and K_2; we can also choose a good setting based on a preliminary evaluation. In all the other experiments of this paper, K_1 and K_2 are set to 0.5 and 0.4, respectively.

G. Scalability Experiments

In this part, we evaluated the performance of the KV stores under different scales, i.e., 20 $\sim$ 80 million key-value operations performed under the YCSB A-variant workload. Figs. 16 and 17 illustrate the 99-percentile tail throughput and the 99-percentile tail latency per second of CruiseDB, RDB, AT-RDB, WiredTiger, and a common-sense static admission strategy, i.e., RocksDB with a fixed throughput limit to the average throughput in the unlimited case.

Tail Throughput. First, as Fig. 16 shows, CruiseDB keeps the tail throughput at a very high level under different data scales. Second, the static admission solution and AT-RocksDB achieve a very similar tail throughput which is better than RocksDB without admission, especially under the larger data scales. It means that the admission is useful in improving the tail throughput. However, the simple static admission still has a large gap compared with the adaptive admission used in CruiseDB; they only achieve a 17.6% to 25.8% tail throughput of CruiseDB. Finally, as a B-tree KV store, WiredTiger's tail throughput drops quickly as the data scale increases. On the contrary, the performance decline of the LSM-tree KV stores due to the deeper tree structure is much slower compared with WiredTiger, except for RocksDB.

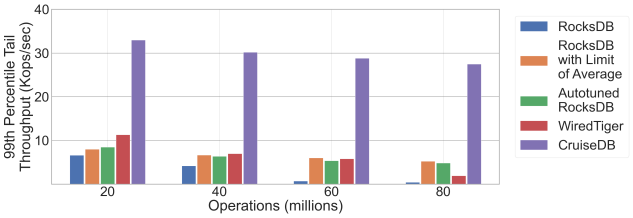

Fig. 16. The 99-percentile tail throughput under different scales.

Tail Latency. It is shown in Fig. 17 that under all the data scales, CruiseDB is always the system that achieves the lowest tail latency. Although RocksDB without admission performs better under the scale of 20 million operations, because the *write stall* is not severe in this phase, RocksDB with static admission and AT-RocksDB achieve lower tail latency under larger data scales. The performance of WiredTiger similarly gets worth as the data scale increases, leading to the worst tail latency in the 80-million case.

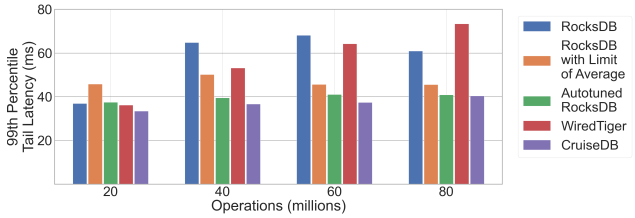

Fig. 17. The 99-percentile tail latency under different scales.

H. Memory Footprint

The real-time memory consumption of CruiseDB under the YCSB workload A-variant was tracked with *pidstat* for 500 seconds and shown in Fig. 18. It contains all the memory consumption of the system including the memory buffer, the cache of hot blocks, bloom filters, metadata, etc. The peak memory footprint of CruiseDB is less than 1.7 GB, which is a small number for modern servers. Fig. 18 also exhibits that the memory consumption goes up and down but not exceeds the memory limitation and CruiseDB maintains a high memory utilization rate of the given memory space, which means the adaptive admission in CruiseDB is very effective.

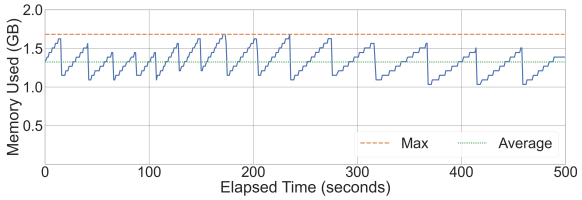

Fig. 18. The memory footprint of CruiseDB.

IX. Related Work

The existing approaches for optimizing LSM-tree KV stores can be divided into the following categories:

Improved Compaction Mechanisms. The most important performance challenge of LSM-tree KV stores comes from the I/O amplification caused by the *compaction* actions. The I/O amplification of *compaction* lies in both the single data merging action between two adjacent layers and merging the same key-value pair level-by-level in the LSM-tree.

Therefore, some of the existing solutions were designed to reduce the I/O amplification rate of single data merging action, e.g., LDC [11], which was a new lower-level-driven *compaction* manner to reduce the I/O amplification by accumulating more upper-level data in one round of *compaction*. Some other ones reduced the level count that key-value pairs have to be written into through delaying or merging some *compaction* actions, e.g., dCompaction [12], Light-weight compaction [13], and PebblesDB [14].

Separating Keys and Values. WiscKey [15] separated the storage of keys and values, using an LSM-tree to manage keys and the pointers to corresponding values, and appending values into logs. Because the values were not involved in *compaction*, this solution had the benefits of reducing the total system I/O amounts. These kinds of solutions were very effective for the applications with large objects, but they were not suitable for small-object cases.

Utilizing Bloom Filters Well. Bloom filters can reduce many unnecessary data reading from I/O devices by indicating a target key is not in a data set accurately with small storage space overhead. But the accuracy of a bloom filter is highly related to its size. Monkey [16] focused on LSM-tree's bloom filters for performance acceleration. The frequently accessed upper levels of the LSM-tree adopted a larger bloom filter for higher accuracy while the lower levels only demanded a smaller bloom filter. This strategy could improve the overall efficiency of bloom filters in the LSM-tree.

Adaptive Schemes. Since different *compaction* manners and LSM-tree shapes are suitable for different applications, some adaptive schemes were proposed to promote adaptability and performance. For example, CuttleTree [17] was an adaptive LSM-tree, whose shape could be auto-tuned according to the run-time statistics of workload patterns. Dostoevsky [18] introduced a hybrid merge policy to remove superfluous merging adaptively. ALDC [19] could adjust the key parameter of *compaction* according to the writing percentage of user accesses to promote the overall performance.

Co-design of Software and Hardware. LOCS [20] was designed to exploit the parallelism of customized open-channel SSDs and to optimize the scheduling and dispatching policies, which could improved the performance. GearDB [21] introduced a new *compaction* method called *Gear Compaction* to eliminate the overhead of on-disk garbage collection especially for the new host-managed shingled magnetic recording drives and thus improved the *compaction* efficiency.

SLA-oriented Optimization. Only a small proportion of existing solutions were designed aiming to promote the SLA of LSM-tree KV stores. Auto-tuned RocksDB [22] and SILK [23], [24] were designed to lower the tail latency of the LSM-tree KV store. And some performance-oriented solutions had some extra benefits of reducing the tail latency to some extent [11], [19]. Different with existing SLA-oriented solutions that focus on optimizing the tail latency only, CruiseDB we proposed aims to ensure a high guaranteed throughput (i.e., tail throughput) first and reduce the tail latency as well, i.e., solving the LSM-tree SLA problems thoroughly.

The tail latency reduction is an important optimization objective for many systems. For example, RStore [34] was designed to reduce the tail latency for the in-memory KV store by fully utilizing multi-cores and thus reducing the waiting time. And FASTER [35] also could reduce the tail latency through parallel accessing by proposing the concurrent hash index and the concurrent log-structured record structure. TPC [36] could split big requests into small ones and serve them in parallel to reduce the tail latency in interactive services.

In fact, lower SLA may be caused by many reasons, and the *write stall* is the most important obstacle to the SLA guarantee of LSM-tree KV stores. CruiseDB solves this problem by both an appropriate admission and improvement of the LSM-tree structure. The ideas of more parallel processing and splitting big requests into small ones from the existing approaches are orthogonal to CruiseDB and can be integrated in CruiseDB to reduce the tail latency further in the future.

X. Conclusion

Although the average performance of LSM-tree key-value stores is good, they generally result in a sharply fluctuating system throughput and high tail latency because of the lack of admission and the inherent structure. Therefore, an SLA-oriented LSM-tree KV store, i.e. CruiseDB, has been designed and implemented to thoroughly solve the SLA problems of the LSM-tree by introducing an adaptive admission and optimizing its structure. With a marginal average performance loss, CruiseDB can promote the guaranteed throughput significantly and reduce the tail latency at the same time.

Acknowledgment

This work is supported by the National Key Research and Development Program of China (No. 2018YFB1004401), National Natural Science Foundation of China (No. 61972402, 61972275, and 61732014). The corresponding author is Yunpeng Chai (ypchai@ruc.edu.cn).

References

[1] Feifei Li. Cloud-native database systems at alibaba: Opportunities and challenges. *Proceedings of the VLDB Endowment*, 12(12):2263–2272, 2019.

[2] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64. ACM, 2012.

[3] Belén Vela, José María Cavero, Paloma Cáceres, Almudena Sierra-Alonso, and Carlos E Cuesta. Using a nosql graph oriented database to store accessible transport routes. In *EDBT/ICDT Workshops*, pages 62–66, 2018.

[4] Ashish Kumar Gupta, Prashant Varshney, Abhishek Kumar, Bakshi Rohit Prasad, and Sonali Agarwal. Evaluation of mapreduce-based distributed parallel machine learning algorithms. In *Advances in Big Data and Cloud Computing*, pages 101–111. Springer, 2018.

[5] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. Optimizing space amplification in rocksdb. In *CIDR*, 2017.

[6] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: a distributed storage system for structured data. *TOCS*, 26(2):15–15, 2008.

[7] Apache hbase, 2017. http://hbase.apache.org/.

[8] Avinash Lakshman and Prashant Malik. Cassandra:a decentralized structured storage system. *Acm Sigops Operating Systems Review*, 44(2):35–40, 2010.

[9] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1493–1509, 2020.

[10] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. Tidb: a raft-based htap database. *Proceedings of the VLDB Endowment*, 13(12):3072–3084, 2020.

[11] Yunpeng Chai, Yanfeng Chai, Xin Wang, Haocheng Wei, Ning Bao, and Yushi Liang. Ldc: a lower-level driven compaction method to optimize ssd-oriented key-value stores. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 722–733. IEEE, 2019.

[12] Feng-Feng Pan, Yin-Liang Yue, and Jin Xiong. dcompaction: Speeding up compaction of the lsm-tree via delayed compaction. *JCST*, 32(1):41–54, 2017.

[13] Ting Yao, Jiguang Wan, Ping Huang, Xubin He, Qingxin Gui, Fei Wu, and Changsheng Xie. A light-weight compaction tree to reduce i/o amplification toward efficient key-value stores. In *MSST*, 2017.

[14] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *SOSP*, pages 497–514. ACM, 2017.

[15] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. In *FAST*, pages 133–148, 2016.

[16] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal navigable key-value store. In *SIGMOD*, pages 79–94. ACM, 2017.

[17] Nicholas Joseph Ruta. *CuttleTree: Adaptive Tuning for Optimized Log-Structured Merge Trees*. PhD thesis, Harvard University, 2017.

[18] Niv Dayan and Stratos Idreos. 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*, pages 505–520, 2018.

[19] Yunpeng Chai, Yanfeng Chai, Xin Wang, Haocheng Wei, and Yangyang Wang. Adaptive lower-level driven compaction to optimize lsm-tree key-value stores. *IEEE Transactions on Knowledge and Data Engineering*, 2020.

[20] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. An efficient design and implementation of lsm-tree based key-value store on open-channel ssd. In *EuroSys*, page 16. ACM, 2014.

[21] Ting Yao, Jiguang Wan, Ping Huang, Yiwen Zhang, Zhiwen Liu, Changsheng Xie, and Xubin He. Geardb: a gc-free key-value store on hm-smr drives with gear compaction. In *17th USENIX Conference on File and Storage Technologies (FAST'19)*, pages 159–171, 2019.

[22] Auto-tuned rate limiter, 2017. https://rocksdb.org/blog/2017/12/18/17-auto-tuned-rate-limiter.html.

[23] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. Silk: Preventing latency spikes in log-structured merge key-value stores. In *2019 USENIX Annual Technical Conference (ATC'19)*, pages 753–766, 2019.

[24] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. Silk+ preventing latency spikes in log-structured merge key-value stores running heterogeneous workloads. *ACM Transactions on Computer Systems (TOCS)*, 36(4):1–27, 2020.

[25] Edward W Knightly and Ness B Shroff. Admission control for statistical qos: Theory and practice. *IEEE network*, 13(2):20–29, 1999.

[26] Meisam Mirahsan, Gamini Senarath, Hamid Farmanbar, Ngoc Dung Dao, and Halim Yanikomeroglu. Admission control of wireless virtual networks in hethet nets. *IEEE Transactions on Vehicular Technology*, 67(5):4565–4576, 2018.

[27] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *SOCC*, pages 143–154, 2010.

[28] Under the hood: Building and open-sourcing rocksdb, 2017. http://goo.gl/9xulVB.

[29] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. Triad: Creating synergies between memory, disk and log in log structured key-value stores. In *2017 USENIX Annual Technical Conference (ATC'17)*, pages 363–375, 2017.

[30] Puqi Perry Tang and T-YC Tai. Network traffic characterization using token bucket model. In *IEEE INFOCOM'99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No. 99CH36320)*, volume 1, pages 51–62. IEEE, 1999.

[31] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of grid computing*, 12(4):559–592, 2014.

[32] Wiredtiger, 2020. http://www.wiredtiger.com/.

[33] Ycsb-c, 2020. https://github.com/basicthinker/YCSB-C.

[34] Lucas Lersch, Ivan Schreter, Ismail Oukid, and Wolfgang Lehner. Enabling low tail latency on multicore key-value stores. *Proceedings of the VLDB Endowment*, 13(7):1091–1104, 2020.

[35] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. Faster: A concurrent key-value store with in-place updates. In *Proceedings of the 2018 International Conference on Management of Data*, pages 275–290, 2018.

[36] Myeongjae Jeon, Yuxiong He, Hwanju Kim, Sameh Elnikety, Scott Rixner, and Alan L Cox. Tpc: Target-driven parallelism combining prediction and correction to reduce tail latency in interactive services. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 129–141, 2016.