Making LSM-Tree-based Key-Value Store Practical and Efficient for Multi-Tenant Serverless Cloud Databases

YINGJIA WANG, The Chinese University of Hong Kong, Hong Kong
CAIXIN GONG, Alibaba Group, China
GUOYUN ZHU*, Alibaba Group, China
SHENG WANG, Alibaba Group, Singapore
ZHENGHENG WANG, Alibaba Group, China
HUAN LIU, Alibaba Group, China
JUNZHI SHI, Alibaba Group, China
WU QIN, Alibaba Group, China
WEI ZHANG, Alibaba Group, China
FEIFEI LI, Alibaba Group, China
MING-CHANG YANG*, The Chinese University of Hong Kong, Hong Kong

Cloud databases revolutionize data processing and storage by providing on-demand and scalable services housed on the cloud infrastructure. Multi-tenancy and serverless are two key tenets transforming the cloud database architecture, offering significant cost savings and simplified user management. However, we found that cloud databases using LSM-tree-based key-value stores as the storage engine face a crucial conundrum when adopting this promising multi-tenant serverless architecture. Specifically, LSM-tree-based key-value store encounters a critical dilemma between maintaining performance service-level agreements (SLAs) for tenants and over-subscribing storage bandwidth for high cost-efficiency.

In this paper, we present FlexEngine[1], a novel LSM-tree-based key-value store, which for the first time enables the practical and efficient adoption of LSM tree in multi-tenant serverless cloud databases. FlexEngine introduces a series of designs to navigate the above dilemma, including a two-level (i.e., partition- and node-level) I/O admission control framework and a two-stage compaction deferral mechanism. We implement FlexEngine on a commercially-deployed RocksDB and perform comprehensive experiments on both production traces and micro-level workloads. The experimental results demonstrate that FlexEngine can significantly improve the capability to over-subscribe storage bandwidth, which leads to high cost-efficiency, while still promising consistent performance SLAs for users.

[1] FlexEngine has already been integrated in Tair Serverless KV [8], a key-value cloud database from Alibaba Cloud.

* The corresponding authors are Ming-Chang Yang and Guoyun Zhu.

Authors' Contact Information: Yingjia Wang, The Chinese University of Hong Kong, Hong Kong, yjwang22@cse.cuhk.edu.hk; Caixin Gong, Alibaba Group, China, caixin.gcx@alibaba-inc.com; Guoyun Zhu, Alibaba Group, China, zongdai@alibaba-inc.com; Sheng Wang, Alibaba Group, Singapore, sh.wang@alibaba-inc.com; Zhengheng Wang, Alibaba Group, China, zhengheng.wzh@alibaba-inc.com; Huan Liu, Alibaba Group, China, qianyi.lh@alibaba-inc.com; Junzhi Shi, Alibaba Group, China, junzhi.sjz@alibaba-inc.com; Wu Qin, Alibaba Group, China, qinwu.qw@alibaba-inc.com; Wei Zhang, Alibaba Group, China, zwei@alibaba-inc.com; Feifei Li, Alibaba Group, China, lifeifei@alibaba-inc.com; Ming-Chang Yang, The Chinese University of Hong Kong, Hong Kong, mcyang@cse.cuhk.edu.hk.

Conference acronym 'XX, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

ACM Reference Format:
Yingjia Wang, Caixin Gong, Guoyun Zhu, Sheng Wang, Zhengheng Wang, Huan Liu, Junzhi Shi, Wu Qin, Wei Zhang, Feifei Li, and Ming-Chang Yang. 2018. Making LSM-Tree-based Key-Value Store Practical and Efficient for Multi-Tenant Serverless Cloud Databases. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX).* ACM, New York, NY, USA, 25 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

The market for *cloud databases* has expanded rapidly in recent years with the launch of services by numerous public cloud providers, including Amazon AWS, Microsoft Azure, Google Cloud, Alibaba Cloud, IBM Cloud, Oracle Cloud, and so on. Unlike traditional on-premises databases, cloud databases offer excellent advantages to users by eliminating substantial upfront capital expenditures on hardware infrastructure while minimizing operational overhead. In addition, built-in high availability, disaster recovery, and seamless global deployment capabilities benefit users without requiring specialized in-house expertise.

Multi-tenancy and *serverless* are two key tenets recently driving the re-architecture of cloud databases. *Multi-tenancy* consolidates multiple tenants sharing the same set of physical resources while maintaining their logical isolation [18, 19, 23, 33, 41, 50, 64]. This aggregated architecture further enables cloud database providers to *over-subscribe* resources, since most real-world tenants have sporadic and low average resource usage [19, 22, 50]. As a result, multi-tenancy can significantly enhance resource utilization and lower the costs for both users and providers. On this basis, *serverless* further simplifies user-level infrastructure management by automatically provisioning, scaling, and optimizing resources on demand, allowing them to focus solely on data operations while only paying for the actual usage [19, 22, 23, 33, 40, 42, 50, 53, 71].

Figure 1 demonstrates the typical architecture of multi-tenant serverless cloud databases, where multiple tenant partitions operate simultaneously in a virtual machine within a node and share the resources in it (see Section 2.1). However, as we move toward the multi-tenant serverless architecture, we found that cloud databases using *LSM-tree-based key-value store* as the storage engine face a severe dilemma, i.e., failing to reconcile *performance service-level agreements (SLAs) for tenants* with *storage bandwidth over-subscription, which is crucial for cost-efficiency.* More specifically, existing LSM tree designs fall short of supporting performance SLAs and storage bandwidth over-subscription simultaneously, let alone pursuing a high degree of storage bandwidth over-subscription for high cost-efficiency.

This dilemma fundamentally stems from the asynchronous way in which LSM tree handles background activities (see Section 2.3). It periodically invokes bandwidth-hungry compaction processes to reclaim invalid data and accelerate read access. As a result, even a few tenants with low user write loads can easily lift the total storage bandwidth usage very high, e.g., when multiple tenants invoke compactions simultaneously. This frequently exhausts the available storage bandwidth, surges user-perceived operation latency, and even causes service unavailability from time to time.

Such a dilemma delivers us two challenges to overcome. The first is to *embrace both performance SLAs and storage bandwidth over-subscription, feasibly.* A straightforward approach to this is to configure each tenant partition with a *maximum available bandwidth*, which should be sufficient to guarantee performance SLAs while also stabilizing the total usage of the system bandwidth not too high [33]. For better resource utilization, this maximum available bandwidth should also be tuned dynamically, e.g., reduced if the actual user loads are low, so that more tenant partitions can be co-located in a node (i.e., achieving over-subscription in fact). However, our study reveals that *auto-tuned rate limiter* [1], the state-of-the-art bandwidth limiting mechanism in RocksDB, can severely break performance SLAs, since it can not respond timely to sudden user load growth, which is prevalent for real-world tenants (see Section 3.1). Therefore, auto-tuned rate limiter can not practically achieve storage bandwidth over-subscription, otherwise performance SLAs are damaged.

The second challenge is to *achieve a high degree of storage bandwidth over-subscription, which significantly improves cost-efficiency.* The key challenge behind high storage bandwidth over-subscription is *maintaining performance SLAs during storage bandwidth shortages.* Even if the maximum available bandwidth of a partition can be limited, when multiple partitions exhaust their bandwidth (e.g., invoke bandwidth-hungry compactions) simultaneously, storage bandwidth shortages inevitably arise. To make matters worse, as the over-subscription ratio of the storage bandwidth increases, the shortfalls of storage bandwidth also rise, thereby placing higher demands on LSM-tree-based key-value stores to sustain performance SLAs. Nonetheless, we found that existing LSM tree designs fail to address this issue satisfactorily, i.e., sustain performance SLAs during storage bandwidth shortages, especially with a high degree of storage bandwidth over-subscription (see Section 3.2). Specifically, they are either inapplicable in the context of cloud databases (e.g., CruiseDB [44] and ADOC [69]) or not effective enough (e.g., Calcspar [72] and SILK [21]). Thus, existing approaches fail to allow providers to configure a high over-subscription ratio in practice.

In this paper, we present *FlexEngine*, a novel LSM-tree-based key-value store. FlexEngine is uniquely positioned in addressing the above two challenges, and for the first time makes LSM tree practically and efficiently adopted in multi-tenant serverless cloud databases. FlexEngine incorporates a *two-level (i.e., partition- and node-level) I/O admission control framework* and a *two-stage compaction deferral* mechanism. To address the first challenge, FlexEngine innovatively designs the partition-level I/O admission control as a foreground/background separating bandwidth management architecture. Foreground tasks, such as user reads and WAL writes, have fixed-reserved bandwidth to ensure timely processing, while background activities, including flushes and compactions, have dynamically-tuned bandwidth to maximize the overall cost-efficiency. To address the second challenge, FlexEngine introduces node-level I/O admission control to holistically schedule I/Os in a node to secure better latency. In addition, a two-stage compaction deferral mechanism is proposed to prevent write stalls in LSM tree, even with highly over-subscribed storage bandwidth.

We summarize our contributions in this paper as follows:

- To the best of our knowledge, we for the first time investigate building LSM-tree-based multi-tenant serverless cloud databases. In particular, we uncover a critical dilemma between performance SLAs and cost-efficiency (from storage bandwidth over-subscription) in this new architecture.
- We present a novel LSM-tree-based key-value store, namely FlexEngine, to navigate this dilemma with a series of designs on I/O admission control and compaction deferral. FlexEngine augments LSM-tree-based cloud databases to be practically and efficiently evolved into the multi-tenant serverless architecture, embracing *both* consistent performance SLAs and *high* cost-efficiency simultaneously.
- We implement FlexEngine based on a commercially-deployed RocksDB [32] and evaluate FlexEngine with both production traces and micro-level workloads. Our experimental results show that FlexEngine can promise consistent performance SLAs for users while significantly improving the storage bandwidth over-subscription capability. We will open-source some production traces along with the evaluation benchmark tool on GitHub for public use[2].

[2] https://github.com/yingjia-wang/ServerlessKVBench

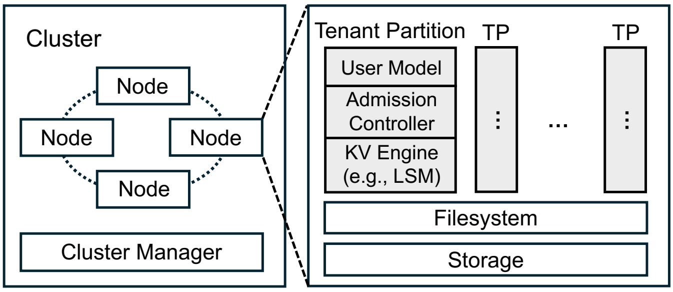


Fig. 1. **The typical architecture of multi-tenant serverless cloud databases.**

In the remainder of this paper, we first present the background in Section 2 and motivation in Section 3. We then elaborate on the designs of FlexEngine in Section 4. The evaluation, related work, and conclusion are introduced in Section 5, 6, and 7, respectively.

2 Background

2.1 Multi-Tenant Serverless Cloud Database

Multi-tenancy is a key principle in cloud databases to achieve high cost-efficiency, consolidating tenants sharing the same physical resources while maintaining their logical isolation [18, 19, 23, 33, 41, 50, 64]. The prevalent adoption of multi-tenancy is predicated on the reality that most tenants have low resource utilization on average. Therefore, it is achievable for the total resources promised to tenants to exceed the physical capacity, which is known as *over-subscription* [19, 22, 41]. This greatly enhances the overall resource utilization, considerably reducing cloud database deployment costs and also delivering more competitive prices to the customers.

The rapid development of serverless computing has fostered a new *serverless* architecture for cloud databases [19, 22, 23, 33, 40, 42, 50, 53, 71]. Serverless databases emphasize leveraging cloud elasticity to dynamically allocate resources based on real-time workload demands, eliminating the need for manual provisioning. By automatically handling scaling, maintenance, and optimization, they significantly reduce the operational burden on users. Furthermore, the pay-as-you-go pricing model ensures high cost efficiency, as users are charged only for actual resource consumption, making serverless databases particularly advantageous for variable or sporadic workloads.

The architecture of multi-tenant serverless cloud databases is demonstrated in Figure 1. User data physically resides in sharded *clusters*, each composed of a set of *nodes* where database tenants are accommodated. Each cluster has a cluster manager to make decisions in placing tenants into nodes, balancing loads across nodes, handling failovers, and so on [50]. Each database tenant is partitioned for performance scalability and further replicated to multiple nodes for high availability. In each node, numerous *partitions* from different tenants are consolidated in a virtual machine, sharing the resources (e.g., storage, CPU, memory, and network) within it. Looking into each tenant partition, an *admission controller* is necessary to enforce a per-partition resource limit, minimizing performance interference between partitions and enabling fast performance scale-ups. In practice, this is accomplished by restricting the maximum number of *read capacity units (RCU)* and *write capacity units (WCU)*. Each capacity unit represents one read or write operation for an item up to a fixed size (e.g., 4KB for RCU and 1KB for WCU supported in our product).

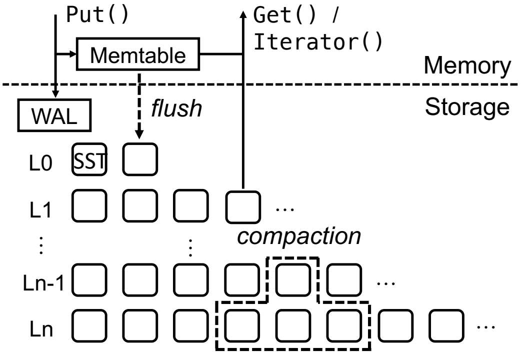


Fig. 2. **The architecture of log-structured merge (LSM) tree.**

2.2 Resource Violation

When resources in a node are over-subscribed, it is indeed possible that the total demanded resources of tenant partitions are close to or even surpass the physical capacity, resulting in *resource violations* [41]. Resource violations, however, are crucial to avoid, as the shortage of resources can degrade service quality and even cause service unavailability. In practice, a node is typically viewed as facing resource violations if the total requirement for a resource exceeds a high-watermark threshold (e.g., 80%). Afterwards, a few resource-hungry tenant partitions in this node are selected and migrated to other nodes, and this procedure continues until the total requirement drops below the high-watermark threshold [33]. Here, configuring a high-watermark threshold for tenant migrations, rather than referring to the exact physical capacity (i.e., 100%), is because tenant migration is not instantaneous [41]. This can allow a period of time for migrations to complete before the resource runs out and the service is impacted.

Serverless databases essentially guarantee the elasticity of services, i.e., can complete auto-scaling through tenant migrations within anticipated times. To achieve this, the time for tenant migrations must be bounded, and in practice, by two principles. First, data migration speed far exceeds the per-partition data input speed. Second, the maximum size of each partition is limited. It is also worth noting that the migration does not take place on the resource-violated node, but on those where the replicas of migrated partitions are located, to avoid further exacerbating the resource shortage. When the migration is finished and a new partition replica is successfully built, the bandwidth traffic is taken over to the new node, after which the resource-violated node can terminate the previous partition and release its resources.

2.3 Log-Structured Merge (LSM) Tree

Log-structured merge (LSM) tree [51] has served as the key-value storage backbone of many data services, powering systems from embedded edge devices to distributed databases handling large-scale workloads. Numerous industry-level LSM-tree-based key-value stores have been introduced and adopted, such as RocksDB [13], LevelDB [12], Cassandra [10], HBase [11] and TiKV [14].

The architecture of LSM tree is demonstrated in Figure 2. Specifically, LSM tree organizes data files in a multi-level architecture for exceptional write performance while maintaining good read performance. At its core, newer data resides in lower levels (e.g., L0), while older data is gradually cascaded down to higher levels (e.g., Ln). Fresh key-value pairs from write operations are initially buffered and batched in an in-memory component called *memtable*. As a memtable grows and reaches its capacity threshold (e.g., 64MB by default in RocksDB), it undergoes a transition to the immutable state, making room for a new memtable to receive incoming writes. The frozen memtable then enters the *flush* process, where its key-value pairs are sorted and persisted to storage as a *sorted string table (SST)*. When the number or the total size of SSTs exceeds the limit of a level, LSM tree invokes the *compaction* process, which selects SSTs in that level and merges with all key-overlapping SSTs from the next lower level. The merged new SSTs are stored in the lower level, while the old ones are deleted. The compaction process has several crucial functions, including eliminating redundant or obsolete data to reduce storage footprint and maintaining a balanced hierarchical structure for read performance.

LSM tree handles user read operations and write-ahead logging (WAL) *synchronously*, while processing flushes and compactions in the background *asynchronously*. User read operations (e.g., `Get()` or `Iterator()`) start with checking in-memory memtables that contain the most recent data. If the requested data is not found in memtables, the search proceeds to SSTs from the lowest (i.e., L0) to the highest (i.e., Ln), ensuring that the most up-to-date version of data is first located. If the requested data is not cached in memory, data must be fetched from storage immediately and synchronously. On the contrary, when a user-written key-value pair arrives (e.g., via `Put()`), it is acknowledged as soon as it is written to WAL and memtable, leaving further data reorganization procedures such as flushes and compactions in the background.

3 Motivation

Multi-tenant serverless cloud databases have gained ever-rising traction in both academia and industry, due to their superior advantages in cost savings and user operational simplicity. Nevertheless, as we evolve our LSM-tree-based cloud database toward the multi-tenant serverless architecture, our product development experience uncovers a critical dilemma between maintaining consistent performance SLAs and achieving cost-effective storage bandwidth over-subscription.

This dilemma manifests as two step-by-step challenges for us: preserving performance SLAs while (1) achieving storage bandwidth over-subscription *feasibly* and (2) achieving storage bandwidth over-subscription *to a high degree*. In the following, we introduce the reasons behind these challenges and our insights that motivate our designs.

3.1 The Feasibility of Achieving Storage Bandwidth Over-Subscription

Since LSM tree handles background activities in an asynchronous way, even a few tenant partitions with low write loads can easily lift the total storage bandwidth usage to very high levels (e.g., when multiple partitions invoke compactions simultaneously). This significantly challenges the promise of performance SLAs, as the resulting frequent bandwidth shortages can substantially elevate I/O latency, which further cascades to upper-level services and is disastrous to user experiences.

To address this problem, one straightforward approach is to configure each partition with a *maximum available bandwidth*, which should be sufficient for guaranteeing performance SLAs while also effectively suppressing the total bandwidth not too high [33]. For better resource utilization, this maximum available bandwidth should also be adjusted dynamically, i.e., taking into account the actual loads at runtime. For example, if a partition is chronically underloaded, the bandwidth allocated to it can be reduced, so that more partitions can be co-located in this node to improve the overall cost-efficiency.

The automatic bandwidth limiting mechanism of RocksDB, *auto-tuned rate limiter* [1], is exactly based on the above idea. It employs the token bucket algorithm [2] to control the bandwidth usage of each RocksDB instance (see Figure 4). The read/write I/Os can only be submitted when they get enough tokens, and tokens are replenished at a specific rate based on the current bandwidth. In response to the ever-changing runtime loads, auto-tuned rate limiter self-adaptively adjusts the bandwidth according to the times tokens are drained during recent token replenishment.

The detailed logic of bandwidth tuning is shown in Equation 1. During initialization, the bandwidth is configured to a specified maximum value (i.e., BW_{max}) and adjusted between this value and its 5% according to $T_{drained}$. Here, *T_drained* is a statistic recorded at runtime, which represents the drained time of tokens during recent token replenishment (e.g., every 100 times by default). Therefore, *T_drained* ranges from 0 to 100, inclusive of both boundaries. Specifically, if tokens are not drained once, the bandwidth is directly set to 5% of BW_{max}. If the drained times of tokens are more than 90 or less than 50, the bandwidth is increased by 5% or decreased by about 5%, respectively. Otherwise, the bandwidth remains unchanged until the next decision time (i.e., after 100 times token replenishment).

$$\mathrm{BW}_{\mathrm{new}} = \begin{cases} 0.05 \times \mathrm{BW}_{\mathrm{max}} & T_{drained} = 0 \\ \max\left(\frac{100}{105} \times \mathrm{BW}_{\mathrm{current}}, 0.05 \times \mathrm{BW}_{\mathrm{max}}\right) & 0 < T_{drained} < 50 \\ \mathrm{BW}_{\mathrm{current}} & 50 <= T_{drained} <= 90 \\ \min\left(\frac{105}{100} \times \mathrm{BW}_{\mathrm{current}}, \mathrm{BW}_{\mathrm{max}}\right) & 90 < T_{drained} <= 100 \end{cases} \tag{1}$$

However, this bandwidth tuning strategy can severely break performance SLAs, as it can not respond quickly to sudden user load growth, which is prevalent for real-world tenants **(Challenge #1)**. This indicates that LSM-tree-based cloud databases can *not* employ auto-tuned rate limiter to achieve storage bandwidth over-subscription feasibly.

We have identified two primary limitations of the existing bandwidth tuning strategy. First, auto-tuned rate limiter employs a *unified* bandwidth control for both foreground and background tasks, which does not consider the synchronous processing nature of user reads and WAL writes. As a result, if a long period of low bandwidth usage has suppressed the partition's maximum available bandwidth to a low watermark, abruptly arriving user reads and WAL writes can be unexpectedly blocked.

Second, auto-tuned rate limiter refers to the recent bandwidth usage statistics to tune the bandwidth in a fine-grained manner, but this inevitably suffers from a *slow* tuning speed. For instance, if the bandwidth should be adjusted from the minimum (i.e., 5% of BW_{max}) to the maximum (i.e., BW_{max}), it requires as long as 62 seconds (i.e., $\ln(1/0.05)$ / $\ln(1.05)$).

3.2 Beyond the Feasibility: Achieving High Storage Bandwidth Over-Subscription

The primary obstacle to high storage bandwidth over-subscription is maintaining performance SLAs in the face of storage bandwidth shortages. Here, storage bandwidth shortage refers to the total storage bandwidth demands of tenant partitions exceeding the physical limit in the node. Even if a partition can have a maximum available bandwidth, storage bandwidth shortages unavoidably occur when multiple partitions exhaust their bandwidth (e.g., invoke compactions) simultaneously, especially with a high over-subscription ratio that exacerbates the bandwidth shortfalls.

Storage bandwidth shortages jeopardize performance SLAs in two aspects: *high latency* and *write stalls*. On the one hand, outstanding I/Os that are not processed in time can *severely inflate user-perceived latency*. On the other hand, a long period of bandwidth shortfalls (due to consistently high user write loads across multiple partitions) can lead to *write stalls* in LSM tree. This is because the actual bandwidth available to each partition is no longer sufficient to digest its compactions (especially L0-to-L1), which accumulates SSTs in L0 and throttles write operations (i.e., WCUs), even if the service remains active for users.

To prevent storage bandwidth shortages, current practice relies on setting a high-watermark threshold for a node, and once reaching this threshold, some bandwidth-hungry partitions are migrated to other nodes until the system bandwidth usage falls below the threshold. However, with a high over-subscription ratio, storage bandwidth shortages usually occur even before tenant migrations are finished (e.g., hundreds of seconds, see Section 2.2).

While prior works have extensively explored various design spaces of LSM tree, they can not satisfactorily address this issue, i.e., sustain performance SLAs until the end of tenant migrations (during storage bandwidth shortages), even with highly over-subscribed bandwidth **(Challenge #2)**. For example, Calcspar [72] explores how to improve the latency during storage bandwidth shortages. It proposes to control the submission of I/Os to storage, ensuring that the required bandwidth of submitted ones is always within the maximum storage bandwidth. Since I/Os can be quickly serviced by storage, Calcspar achieves stably low latency. However, Calcspar can not prevent write stalls, and according to our experiments on production traces, as high as 15% of user WCUs are blocked unexpectedly (see Section 5.2).

Speaking of reducing write stalls, a few existing works, such as CruiseDB [44], ADOC [69], and SILK [21], propose mitigating strategies, but they are either inapplicable or insufficient. Specifically, CruiseDB [44] proposes to reduce the operations that can be permitted into the LSM tree, which is unsuitable in our case, as this explicitly leads to user service unavailability. ADOC [69] increases the number of compaction threads when detecting write stalls to improve the chance of L0-to-L1 compaction being assigned a thread and also decrease the flush speed. ADOC is not usable in our scenario because of its passive decision-making strategy, which already results in service downtime before taking action. In addition, solely adding compaction threads can not quickly eliminate write stalls, while it is important for us to revive user services as soon as possible.

SILK [21] proposes to schedule I/Os from different tasks according to the priority of user reads, flushes, and compactions. More specifically, SILK prioritizes guaranteeing sufficient bandwidth for user reads, while opportunistically allocating remaining bandwidth for other activities (i.e., flushes and compactions). For compactions, SILK further enables higher-level compactions other than L0-to-L1 (which have minimal effects on read performance in a short period) to be suspended when the available bandwidth of non-user operations is restricted.

However, SILK has two key drawbacks. First and most importantly, SILK, originally designed for single-tenant databases, can not be simply migrated to the multi-tenant serverless architecture due to the lack of a holistic control framework across all partitions. Specifically, each partition manages its own SILK instance as well as the maximum available bandwidth, but due to storage bandwidth over-subscription, the actual available bandwidth of a partition may vary, making the partition-level control of SILK not effective in use. Second, from the technical perspective, although suspending higher-level compactions except L0-to-L1 is useful in delaying the onset of write stalls, we found that the effectiveness of preventing write stalls is limited. This is because, since L1-to-L2 compactions are suspended, SSTs in L1 are quickly accumulated, which continuously raises the I/O amplification of L0-to-L1 compactions and makes it slower and slower. As a result, SSTs in L0 are quickly amassed and cause write stalls.

3.3 Summary

In conclusion, existing LSM-tree-based key-value stores can *not* be directly applied to multi-tenant serverless cloud databases. On the one hand, the existing partition-level bandwidth limiting strategy (i.e., auto-tuned rate limiter) employs a unified and recent-usage-driven mechanism to adjust the bandwidth, which fails to respond timely to abrupt user load growth. On the other hand, existing LSM-tree-based key-value stores can not sustain performance SLAs until the end of tenant migrations (during storage bandwidth shortages), especially with a high over-subscription ratio.

On tackling the above issues, we are motivated to design a new key-value store, reviving LSM tree to be served practically and efficiently in multi-tenant serverless cloud databases.

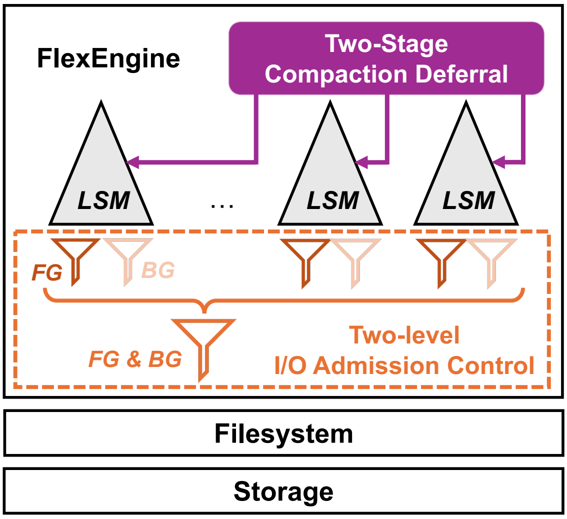

FlexEngine
Two-Stage Compaction Deferral
LSM
...
LSM
LSM
FG
BG
FG & BG
Two-level I/O Admission Control
Filesystem
Storage

Fig. 3. **Overview of FlexEngine.** FG: Foreground I/Os. BG: Background I/Os.

4 Designs of FlexEngine

4.1 Overview

In this paper, we present *FlexEngine*, a novel LSM-tree-based key-value store. FlexEngine harmonizes guaranteed performance SLAs with high cost-efficiency, and further significantly improves the capability to over-subscribe storage bandwidth.

The architectural overview of FlexEngine is demonstrated in Figure 3. FlexEngine first introduces a *two-level I/O admission control framework*, at the levels of *tenant partition* and *node*. The I/O rate limiters at both levels employ the *token bucket algorithm* [2] to control the bandwidth usage, while incorporating multi-priority queues to prioritize critical tasks. However, the principles and goals of the two levels are completely different. At the partition level, FlexEngine innovatively separates foreground I/Os with user reads and WAL writes from background I/Os with flushes and compactions, controlling their maximum available bandwidth individually. Specifically, FlexEngine employs *fixed-reserved* maximum bandwidth management for foreground I/Os to ensure timely processing even during sudden user load growth, while using *dynamically-tuned* maximum bandwidth management for background I/Os to maximize the overall cost-efficiency **(for Challenge #1)**. In contrast, at the node level, FlexEngine regulates I/Os from all partitions, ensuring that the required bandwidth of submitted I/Os is always within the maximum storage bandwidth, thereby achieving consistently low latency **(for Challenge #2)**.

In addition to the two-level I/O admission control framework, FlexEngine further introduces a *two-stage compaction deferral* mechanism. Together with node-level I/O admission control that secures better latency, this mechanism aims to postpone the occurrence of write stalls long until the end of tenant migrations, effective even with highly over-subscribed storage bandwidth **(for Challenge #2)**. The key observation is that the time for tenant migrations is *finite*, and thus, we can *temporarily* accumulate data in low levels (i.e., L1 at the first stage and L0 at the second stage) of the pyramid-like structure of LSM tree, and revert it to the normal layout in the background after the finish of tenant migrations. In this way, the system bandwidth usage can be dramatically

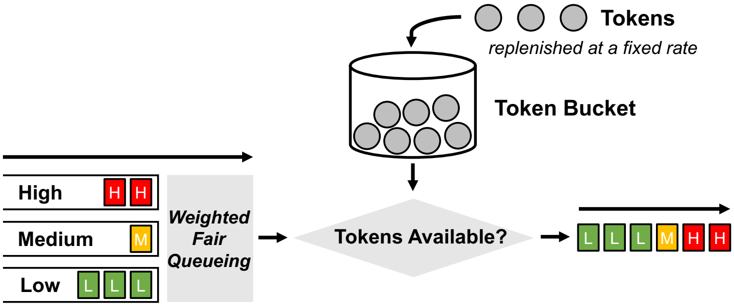


Fig. 4. **Overview of token bucket algorithm with multi-priority queues (e.g., high, medium, and low).**

lowered, within the over-subscribed physical limit for a much longer period without degrading performance SLAs.

In the following, we introduce the detailed designs of two-level I/O admission control and two-stage compaction deferral in Section 4.2 and 4.3, respectively. We conclude with several meaningful discussions in Section 4.4.

4.2 Two-level I/O Admission Control

In this section, we first introduce the basics of the token bucket algorithm as well as the integration with multi-priority queues in Section 4.2.1. We then present the detailed designs of partition-level and node-level I/O admission control in Section 4.2.2 and 4.2.3, respectively.

4.2.1 Basics of Token Bucket Algorithm with Multi-Priority Queues. The *token bucket algorithm* is a classical method for rate limiting and traffic shaping [2]. As shown in Figure 4, it operates by maintaining a bucket of tokens, where each token grants permission to process a unit of work (e.g., a byte of data). Tokens in the bucket are replenished at a fixed rate, and incoming tasks must consume a corresponding number of tokens to proceed. If there are not sufficient tokens available, tasks must be postponed. By enforcing a controllable rate, the token bucket mechanism guarantees system responsiveness while also providing system stability.

The token bucket algorithm can further support *multi-priority queues*, which enables differentiated token allocation based on task importance. In this model, multiple queues, each assigned a priority level, compete for tokens from the same shared bucket. A common implementation of this is based on *weighted fair queueing (WFQ)* [3], which grants higher-priority queues more tokens to ensure critical tasks can be processed in time, while also giving lower-priority queues a small amount of tokens without starving them. By blending the awareness of task importance, the token bucket algorithm becomes more practical in meeting real-world system requirements and further improves overall efficiency.

For example, RocksDB maintains three priority queues competing for tokens, representing that I/Os come from user reads and WAL (high-priority), flushes (medium-priority), and compactions (low-priority), respectively. When new tokens are replenished, RocksDB determines an order of different queues for allocation during this round, using a method similar to flipping a coin. Specifically, RocksDB first randomly selects between "high" and "medium and low", with "high" having 10× higher probability. After determining the order between "high" and "medium and low", RocksDB selects between "medium" and "low", with "medium" also having 10× higher probability. Once the order of different queues is decided, RocksDB prioritizes processing tasks in the earlier

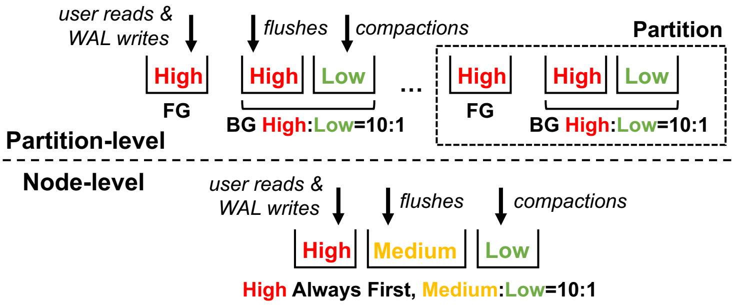


Fig. 5. **Overview of two-level I/O admission control framework.** FG: Foreground I/Os. BG: Background I/Os.

queues during this round, and only allocates tokens to later queues when earlier ones have no tasks remaining.

4.2.2 Partition-level I/O Admission Control. The goal of partition-level I/O admission control is to ensure performance SLAs for tenant partitions while effectively stabilizing the system bandwidth usage. To address the challenge of sustaining performance SLAs during sudden user load growth, FlexEngine assigns each partition two I/O rate limiters, which control the maximum available bandwidth of *foreground* and *background* I/Os individually. Both rate limiters employ the simple yet efficient token bucket algorithm, as described in Section 4.2.1. Specifically, incoming I/Os must acquire enough tokens (each corresponding to a byte) from the bucket before being submitted to storage. If there are not enough tokens yet, the I/Os must be deferred.

However, they are different in the following two aspects. First, a key question is how to determine the token replenishment rate, i.e., configure the maximum available bandwidth at runtime. To this end, FlexEngine employs distinct strategies for them, taking into account their different processing natures (i.e., synchronous or asynchronous). Specifically, FlexEngine employs *fixed-reserved* maximum bandwidth management for foreground I/Os to ensure timely processing, while using *dynamically-tuned* maximum bandwidth management for background I/Os to maximize the overall cost-efficiency. The improvement on cost-efficiency here comes from the fact that the maximum available background bandwidth can be dynamically reduced if the user write load is constantly low. Thus, more partitions can co-exist in the node to enhance the resource utilization.

Second, they have different setups of multi-priority queues. As shown in Figure 5, the rate limiter for foreground tasks only holds *one* priority queue, i.e., both user reads and WAL writes have the same priority. On the contrary, the rate limiter for background tasks maintains *two* priority queues, with the high-priority queue for flush I/Os and the low-priority queue for less-critical compaction I/Os. Same as the default configuration in RocksDB [5], the weighted ratios between high- and low-priority queues are 10:1.

We then introduce the detailed bandwidth tuning logic of foreground/background I/O rate limiters:

- *Foreground I/O Path.* The maximum available bandwidth for foreground I/Os is always reserved according to the sum of two components. One is for *user reads*, which is obtained by the maximum number of RCUs per second ($\text{RCU}_{\text{max\_num\_psec}}$) multiplied by the maximum supported RCU size ($\text{RCU}_{\text{max\_size}}$) and the read amplification factor (RAF). The other is for *WAL writes*, which is calculated by the maximum number of WCUs per second ($\text{WCU}_{\text{max\_num\_psec}}$) multiplied by the maximum supported WCU size ($\text{WCU}_{\text{max\_size}}$).

$$\begin{aligned}\text{Foreground BW}_{\text{max}} = &\ \text{RCU}_{\text{max\_num\_psec}} \times \text{RCU}_{\text{max\_size}} \times \text{RAF} \\ &+\text{WCU}_{\text{max\_num\_psec}} \times \text{WCU}_{\text{max\_size}}\end{aligned} \tag{2}$$

As we have introduced in Section 2.1, each capacity unit (i.e., RCU or WCU) represents a read or write operation for an item up to a fixed size (i.e., $\text{RCU}_{\text{max\_size}}$ or $\text{WCU}_{\text{max\_size}}$). $\text{RCU}_{\text{max\_num\_psec}}$ and $\text{WCU}_{\text{max\_num\_psec}}$ are per-partition limits, which enforce the resource usage of each partition to minimize performance inference between partitions.

On the other hand, RAF represents the multiples of actual disk read I/O sizes to each RCU item size (e.g., 4KB). A showcase of how to derive the upper bound of RAF is introduced as follows. Assume we have four levels of SSTs from L0 to L3. Since SSTs in L0 and L1 are usually cached in memory for high access performance, each RCU requires reading at most two SSTs (i.e., one from L2 and the other from L3). Since an SST is ordered internally, the address space of each RCU is mapped to at most two logical blocks (i.e., 4KB*2=8KB) because each data block in the SST may not be logical-block-aligned. As a result, the upper bound of RAF is 4 (i.e., 2 SSTs*8KB/4KB RCU=4) in this case.

- *Background I/O Path.* The maximum available bandwidth for background I/Os is dynamically tuned based on the average per second loaded data volume from WCUs ($\text{Data}_{\text{avg\_psec}}$) multiplied by the background I/O amplification of LSM tree (LSM-BA).

$$\text{Background BW}_{\text{max}} = \text{Data}_{\text{avg\_psec}} \times \text{LSM-BA} \tag{3}$$

We note that $\text{Data}_{\text{avg\_psec}}$ is not the same as $\text{WCU}_{\text{max\_num}}$ multiplied by $\text{WCU}_{\text{max\_size}}$ used in the foreground I/O path. Specifically, $\text{Data}_{\text{avg\_psec}}$ represents the actual loaded data volume, while users typically invoke a much smaller number of WCUs than $\text{WCU}_{\text{max\_num\_psec}}$, and a WCU may contain a much smaller data volume than its supported $\text{WCU}_{\text{max\_size}}$.

By default, $\text{Data}_{\text{avg\_psec}}$ is adjusted every minute and recorded as the average amount of loaded data per second during this period. For example, if a partition is loaded 120MB of data in total during the last minute, $\text{Data}_{\text{avg\_psec}}$ is set to 120MB/60s=2MB/s for the current minute.

On the other hand, LSM-BA defines the multiples of disk I/O sizes (including both reads and writes) to $\text{Data}_{\text{avg\_psec}}$. A showcase of how to derive the upper bound of LSM-BA is introduced as follows [6]. By default, RocksDB sets the size of L0 to be equal to that of L1, and for higher levels starting from L1, the size multiplier of adjacent levels is 10. If a partition currently has four levels (i.e., L0 to L3), the write amplification is 1(L0)+2(L0-to-L1)+11(L1-to-L2)+11(L2-to-L3)=25. Since SSTs in L0 and L1 are usually cached in memory, the read amplification is 10(L1-to-L2)+11(L2-to-L3)=21. The L1-to-L2 read amplification is 10 (not 11) because the associated SST in L1 is cached, so we only read 10 SSTs in L2 from storage. Thus, the upper bound of LSM-BA is 46 (i.e., 25+21=46) in this case.

4.2.3 Node-level I/O Admission Control. The objective of node-level I/O admission control is to achieve consistently low latency even during storage bandwidth shortages. To this end, FlexEngine further integrates a node-level I/O rate limiter, which regulates I/Os from all partitions holistically. It also employs the token bucket algorithm, but incorporates *three* priority queues to synergize with the awareness of I/O importance in partition-level partners.

As shown in Figure 5, the high-, medium-, and low-priority queues keep I/Os from foreground tasks (i.e., user reads and WAL writes), flushes, and compactions, respectively. Regarding the maximum available bandwidth (which decides the token replenishment rate), the node-level rate limiter configures it to the maximum storage bandwidth, ensuring that the required bandwidth of submitted I/Os is always within the maximum storage bandwidth. In this way, dispatched I/Os

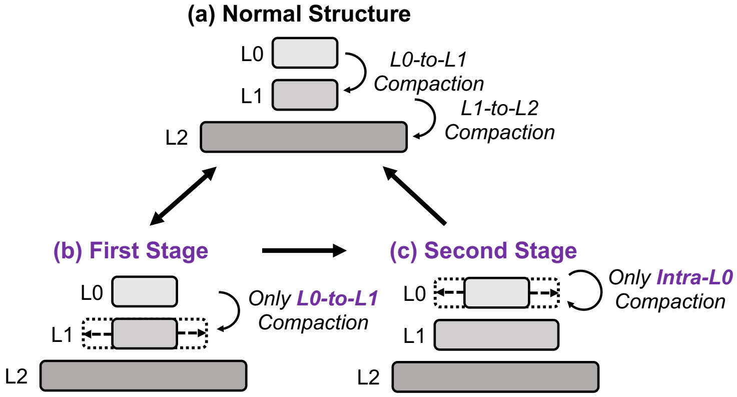


Fig. 6. **Structural changes of LSM tree when two-stage compaction deferral occurs.**

(especially those critical from user reads) can be quickly served, which substantially improves the user-perceived latency. As for how to allocate tokens between queues, the node-level rate limiter always prioritizes high-priority foreground I/Os if they exist, making sure these I/Os that have fixed-reserved bandwidth in partitions are not unexpectedly blocked. For medium- and low-priority I/Os, their weighted ratio is still 10:1 by default.

Finally, it is worth noting that the node-level I/O rate limiter manages all partitions in this node at *any* moment, especially taking into account the partition move-ins and move-outs. For example, if the total storage bandwidth required by all partitions compared to the physical capacity is low, new partitions can be added to this node by the cluster manager, and vice versa.

4.3 Two-Stage Compaction Deferral

To sustain performance SLAs during storage bandwidth shortages, employing only node-level I/O admission control is *not* sufficient. This is because, although node-level I/O admission control rescues the latency, a long period of bandwidth shortfalls (due to consistently high user write loads across multiple partitions) can result in write stalls, which disastrously disrupts users' active services. More specifically, in the event of storage bandwidth shortages, the actual available bandwidth of each partition is no longer sufficient to digest its compactions (especially L0-to-L1), yielding SSTs in L0 to accumulate quickly. If the number of SSTs in L0 surpasses the threshold of write stalls (e.g., 20 in RocksDB[3] [4]) before tenant migrations are finished, performance SLAs are compromised.

In response to this, FlexEngine proposes a *two-stage compaction deferral* mechanism, the goal of which is to temporarily suppress the I/O amplification (as well as the bandwidth requirement) during the period of tenant migrations, so as to prevent the occurrence of write stalls. In the following, we first introduce its detailed rules in Section 4.3.1, and then discuss its implications on system memory usage and read performance in Section 4.3.2.

4.3.1 Transition Flow. When detecting storage bandwidth violations (i.e., the system bandwidth usage surpasses the high-watermark threshold), as shown in Figure 6, FlexEngine enters the *first* stage: *proactively suspending compactions other than L0-to-L1 in some partitions.* By suspending these high-level compactions, the I/O amplification as well as the bandwidth requirement is drastically

[3] RocksDB partially stalls and completely stalls new key-value pairs when there are 20 and 32 SSTs in L0, respectively. In this paper, write stall indicates the first case because even partial data throttling is not acceptable for users.

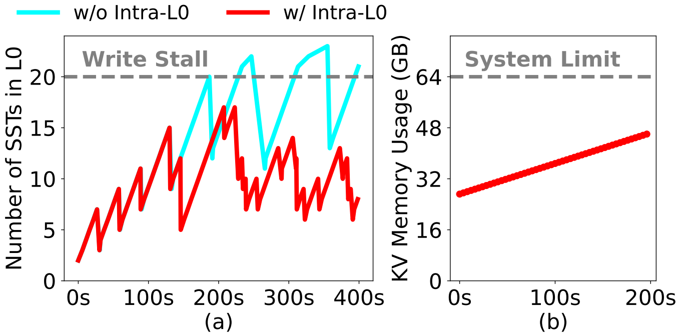


Fig. 7. **Runtime number of SSTs in L0 and application-level (KV) memory usage when two-stage compaction deferral occurs.** For (a), we extend the time to 400s (about twice the tenant migration time) to better observe the effects of transitioning to intra-L0 compactions.

lowered. This is because the I/O amplification of background activities now is only from flushes and L0-to-L1 compactions.

However, suspending L1-to-L2 as well as higher-level compactions inevitably amasses SSTs in L1, and because more and more SSTs in L1 are wrapped in L0-to-L1 compactions, the I/O amplification of L0-to-L1 compactions is continuously increased. This constantly slows down the digestion of L0-to-L1 compactions, leads to a growing number of SSTs in L0, and eventually causes write stalls. Our experiments in Section 5.3.2 show that only the first stage can not prevent write stalls and maintain performance SLAs for long (i.e., until tenant migrations are finished) with a high degree of storage bandwidth over-subscription. Particularly, write stalls can occur when the over-subscription ratio is only 2.

Hence, once the number of SSTs in L0 reaches 60% of the write stall threshold (12 by default), FlexEngine moves into the *second* stage, in which *only intra-L0 compactions are performed and L0-to-L1 compactions are also suspended.* Specifically, FlexEngine continuously monitors the number of SSTs in L0, and when it is at least four, FlexEngine can invoke an intra-L0 compaction. Each intra-L0 compaction picks multiple SSTs from L0, merges them into one larger SST, and still places the new SST in L0. Multiple intra-L0 compactions can be invoked concurrently because the newly generated SSTs remain in L0 where keys across different SSTs are unordered. One noteworthy point for concurrent intra-L0 compactions is that, to consistently maintain the chronological order of SSTs for read correctness, FlexEngine selects current SSTs for intra-L0 compactions always in contiguous batches, where a batch indicates a set of SSTs with contiguous chronological order.

By transitioning from L0-to-L1 compactions to intra-L0 compactions, FlexEngine avoids merging with the already massive data in L1, thereby effectively suppressing the I/O amplification again. The effect of such a transition is clearly visualized in Figure 7a, which depicts the number of SSTs in L0 at runtime when the second stage is enabled or not (more experimental setups in Section 5.1 and 5.3.2). We can observe that, starting around 131s, intra-L0 compactions (i.e., red line) constantly convert multiple SSTs in L0 into a larger one, which suppresses the number of SSTs in L0 again and again to prevent the occurrence of write stalls. Conversely, if not transitioning to the second stage (i.e., cyan line), write stall quickly occurs at only about 187s.

Despite the fact that the second stage is not a panacea and the I/O amplification is also continuously growing as the times of intra-L0 compactions rise, we found that it is effective enough. Particularly, our experiments in Section 5.3.2 demonstrate that FlexEngine with two-stage compaction deferral can sustain performance SLAs even when the over-subscription ratio is 3 and all partitions exhaust their maximum available bandwidth for background I/Os. Moreover, as shown in Section 5.4, if the ratio of write-intensive partitions in a node is not too high (e.g., 50%), the supported over-subscription ratio of FlexEngine without compromising performance SLAs is as high as 7. It is also worth noting that, when the system storage bandwidth becomes sufficient again (either because the user loads become low or tenant migrations are finished), FlexEngine resumes suspended compactions in the background and restores LSM tree to the normal pyramid structure.

4.3.2 Implications on System Memory Usage and Read Performance. Since the lowest-level SSTs (e.g., L0 and L1 in our environments) are pinned in memory for high access performance, FlexEngine increases *application-level* memory usage when two-stage compaction deferral occurs, as more data is temporarily accumulated in L0 and L1. However, it is worth noting that such memory expansion due to larger L0 and L1 is finite and controllable, and does *not* lead to potential out-of-memory (OOM) issues. This is because in serverless cloud databases, the time for tenant migrations is effectively bounded (see Section 2.2). Therefore, once partitions with high write loads are migrated out, system bandwidth usage returns low, allowing deferred compactions to be resumed. As a result, the duration of two-stage compaction deferral is also bounded. Given the limited data input speed (via the maximum number of WCUs per second and the maximum WCU size, see Section 2.2), the temporarily accumulated data in L0 and L1, which is the cause of application-level memory expansion, remains bounded as well.

Figure 7b clearly illustrates the changes in application-level memory usage when two-stage compaction deferral occurs. The results are measured based on the setups in Section 5.1 and 5.3.2, where there are 32 partitions co-located and each partition continuously receives 3MB/s data input. We can see that FlexEngine itself occupies increasingly more memory as compaction deferral proceeds, and the memory peak compared to the start is close to the total data input of all partitions during this period (i.e., 32*3MB/s*200s=19.2GB). Specifically, FlexEngine occupies up to about 72% of the system memory (i.e., 64GB). However, it is worth noting that, in production environments where the system memory of ECS instances, the number of partitions, and the per-partition data input speed are appropriately coordinated, this predictable memory spike of FlexEngine can be safely controlled by the provider.

Speaking of read performance, overall, the two-stage compaction deferral mechanism incurs minimal overhead because short-term deferral of high-level compactions has minimal effect on user read performance [21, 72]. Specifically for L0 where its SSTs have overlapped key ranges, the read performance is also well secured due to the following two main reasons. First, SSTs in L0 are pinned in memory for high access performance. Second, the number of SSTs in L0 is continuously suppressed, either by L0-to-L1 compactions (at the first stage) or intra-L0 compactions (at the second stage).

4.4 Discussions

Architectural choices of multi-tenant LSM trees. As shown in Figure 3, we allocate each tenant partition an independent LSM tree. The key reason behind this is reliability, which is crucial for cloud database products. Specifically, by operating separate LSM trees for each partition, we can configure flexible and independent data backup strategies for customers. Another plausible approach we have considered is allocating each partition a column family, rather than an LSM-tree-based RocksDB instance. The primary drawback of this approach is that all column families share and compete for the same write-ahead logging (WAL), thereby encountering a severe write performance bottleneck.

Generalizing FlexEngine into other LSM-tree-based key-value stores. The designs of FlexEngine, which center on I/O rate-limiting and compaction deferral (or management), can be non-intrusively generalized to other LSM-tree-based key-value stores because these two components are fundamental for the LSM tree structure. Specifically, I/O rate-limiting is crucial for system performance stability, which coordinates the intrinsic bandwidth competition between foreground and background tasks. On the other hand, compaction management with different strategies is also essential, since it allows users to achieve their expected balance between write amplification, read amplification, and space amplification.

How the decoupling process architecture for foreground and background tasks of FlexEngine could benefit other DBMS. Overall, our proposed architecture works if temporarily deferring background tasks does not notably affect user experiences, which can be met in many DBMS. Meanwhile, the more resources required by background tasks, the greater benefits our proposal can offer. We further elaborate with the following two examples. For DBMS checkpointing, users often specify their anticipated checkpointing time window, so our proposal works if the deferral time is within the specified time window. While for Postgres' vacuum [15], deferring its garbage collection may slow down user queries, so a more meticulous in-context investigation is required to secure user experiences, similar to how we design FlexEngine for LSM trees.

5 Evaluation

5.1 Experimental Setups

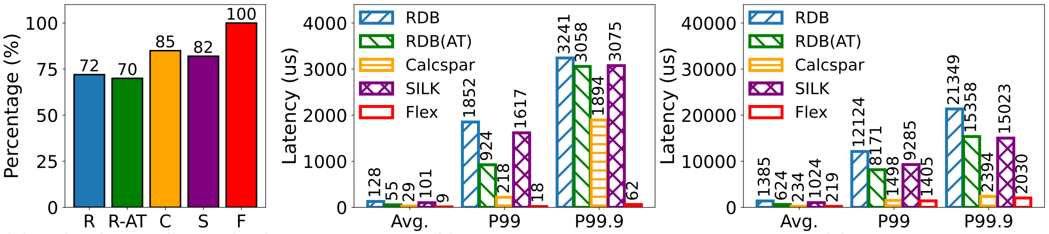
FlexEngine is implemented based on a commercially-deployed RocksDB [13], and runs in an ECS virtual machine from Alibaba Cloud (type: ecs.g5.4x.large) with 16 vCPUs and 64GB memory. The storage device is an Alibaba Cloud Enterprise SSD (ESSD) [7], and its performance setups are detailed in each section below.

We compare FlexEngine with four representative state-of-the-art LSM-tree-based key-value stores, including (vanilla) RocksDB [13], RocksDB with auto-tuned rate limiter [1], SILK [21], and Calcspar [72]. The designs of auto-tuned are introduced in Section 3.1, while the principles of SILK and Calcspar are shown in Section 3.2. In all evaluated stores, the write buffer size is set to a relatively small 16MB, ensuring that the memory usage of the multi-tenant system not too high. For each partition, there are a maximum of 4 write buffers, with both L0 and L1 triggering compactions with the next levels when their sizes reach 64MB.

5.2 Production Workload Performance

In this section, we first evaluate FlexEngine with our production workload, and our goal is to examine its effectiveness in real-world deployments. Specifically, we collect a node-level production trace from our deployment environments for one week, and replay it on FlexEngine as well as four compared state-of-the-art works. We focus on two primary metrics in relation to performance SLAs: (1) *the ratio of normally-serviced WCUs*, which indicates the ratio of WCUs *not* unexpectedly suspended by write stalls, and (2) *latency* of WCUs or RCUs on average (Avg.), at 99th percentile (P99), and 99.9th percentile (P99.9). In particular, only a ratio of 100% for normally-serviced WCUs is satisfactory for the storage engine, while for latency, the lower the better.

The results are demonstrated in Figure 8, with (a), (b), and (c) showing the ratio of normally-serviced WCUs, WCU latency, and RCU latency, respectively. These metrics are presented as the average value of all partitions in the node. From Figure 8a, we can first observe that only FlexEngine is capable of serving WCUs from all partitions normally, thanks to the meticulous designs on both two-level I/O admission control framework and two-stage compaction deferral mechanism. In contrast, all other approaches, i.e., RocksDB, RocksDB with auto-tuned, Calcspar, and SILK, suffer from varying degrees of write blocking from 15% to 30%, which severely sacrifices user service availability.



(a) Ratio of Normally-Serviced WCUs (b) WCU Latency (c) RCU Latency

Fig. 8. **Production workload performance in terms of the ratio of normally-serviced WCUs and WCU/RCU latency.** R/RDB: RocksDB. R-AT/RDB(AT): RocksDB with auto-tuned. C: Calcspar. S: SILK. F/Flex: FlexEngine.

Moreover, as shown in Figure 8b, FlexEngine achieves the best WCU latency, significantly lower than the second-best work Calcspar in average, P99, and P99.9 by 69.0%, 91.7%, and 96.7%, respectively. All other approaches fail to avoid the occurrence of write stalls and incur the tail latency of WCUs as high as several milliseconds. Regarding the RCU latency in Figure 8c, FlexEngine also attains the best RCU latency, similar to Calcspar but significantly outperforming other schemes. Here, the reason Calcspar also achieves good RCU latency is that both FlexEngine and Calcspar holistically control I/Os from all partitions and prioritize the processing of user reads. We finally note that the superior latency of FlexEngine on both WCUs and RCUs is exhibited across all partitions without showing significant deviations.

5.3 Performance Breakdown Analysis

In this section, we individually investigate the design effectiveness of FlexEngine via breakdown experiments. In Section 5.3.1, we explore if partition-level I/O admission control can maintain performance SLAs during sudden user load growth (Challenge #1). Then in Section 5.3.2, we study if node-level I/O admission control and two-stage compaction deferral can sustain performance SLAs during the end of tenant migrations, with a high storage bandwidth over-subscription ratio (Challenge #2).

5.3.1 When User Load Suddenly Increases. To examine the performance of FlexEngine when user load suddenly increases, we first preload a partition with 10M WCUs (each for a 1KB key-value pair) and then run a workload on it. This workload runs for 40 seconds: it initially has no traffic to the partition for the first 10 seconds, and then constantly calls 3K RCUs per second with Zipfian distribution for the remaining 30 seconds. We compare FlexEngine to the state-of-the-art work in the field: RocksDB with auto-tuned I/O rate limiter [1].

Figure 9a demonstrates the number of serviced RCUs per second at runtime. We can see that, starting from the 10th second, FlexEngine can serve all the RCUs with no blocking. This mainly contributes to the foreground/background separated maximum bandwidth management in partition-level I/O admission control. More precisely, the foreground user reads have a reserved bandwidth for use, thereby achieving non-blocking RCU processing as well as excellent latency.

On the contrary, auto-tuned can only service 29.2% of RCUs at the 10th second. This is because, as introduced in Section 3.1, a long time of low bandwidth utilization has suppressed the maximum available bandwidth of auto-tuned to the lowest watermark. To make matters worse, RocksDB with auto-tuned can only admit 3K RCUs per second after the 38th second, which means that it suffers from high user-perceived latency for as long as 28 seconds. We also depict the average, P99, and

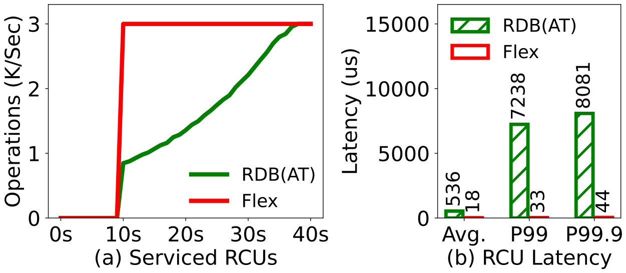


Fig. 9. **Runtime number of serviced RCUs and latency under a workload with suddenly increased user read loads.** RDB(AT): RocksDB with auto-tuned. Flex: FlexEngine.

P99.9 latency of RCUs in Figure 9b, which clearly reveals that, compared to auto-tuned, FlexEngine substantially improves the average, P99, and P99.9 latency by 96.6%, 99.5%, and 99.5%, respectively.

5.3.2 When Storage Bandwidth Shortages Occur. To study the performance of FlexEngine when storage bandwidth shortages occur, we co-locate 32 partitions in a node and first preload each with 10M WCUs (each for a 1KB key-value pair). After that, we employ a workload that continuously inserts 3K WCUs per second into all partitions (representing consistently high user loads), and the workload runs until all tenant migrations are finished. According to Section 4.2.2, each partition is configured with 60MB/s maximum available bandwidth for background tasks, obtained by multiplying the average per second loaded data volume from WCUs (i.e., $\text{Data}_{\text{avg\_psec}}$, 3K*1KB=3MB/s) and the LSM background I/O amplification (i.e., LSM-BA, 20 measured in this experiment).

To enhance the comprehensibility of experiments, we also evaluate under *different degrees of storage bandwidth shortages*. This is achieved by configuring the *maximum ESSD bandwidth*[4], minus the fixed-reserved bandwidth for foreground tasks, below the total required background bandwidth of partitions above (i.e., 32*60MB/s= 1920MB/s). For simplicity, in the following, we define the ratio of maximum ESSD bandwidth minus foreground reserved bandwidth to the total required background bandwidth of partitions as *background over-subscription ratio (BOSR).* In this section, BOSR is set to 1.5, 2, 2.5, and 3, with higher BOSRs evaluated in the next section.

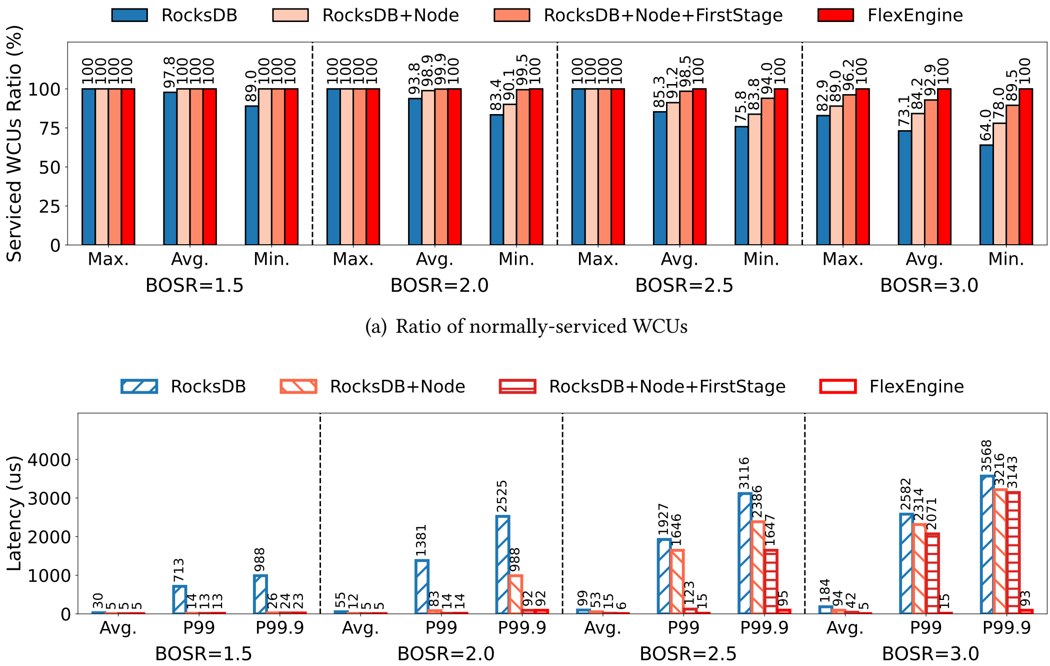
We compare FlexEngine with RocksDB as well as two new schemes for ablation experiments:

- *RocksDB+Node*, which indicates that only node-level I/O admission control is applied based on the vanilla RocksDB.
- *RocksDB+Node+FirstStage*, which represents that node-level I/O admission control and the first stage of two-stage compaction deferral (i.e., only defer higher-level compactions other than L0-to-L1 and not transition to intra-L0 compactions) are employed based on the vanilla RocksDB.

The results are demonstrated in Figure 10. Figure 10a shows the ratio of normally-serviced WCUs with maximum, average, and minimum values across the partitions (i.e., Max., Avg., and Min.). We can first see that FlexEngine can satisfactorily serve all WCUs in all partitions with BOSRs ranging from 1.5 to 3, while other schemes suffer from different degrees of service unavailability.

Specifically, the vanilla RocksDB performs the poorest: it is unable to serve WCUs in some partitions even with a BOSR of 1.5 (i.e., 2.2% on average in partitions and up to 11.0% in a partition); at a BOSR of 3, an average of 26.9% WCUs from partitions can not be admitted. *RocksDB+Node* acts a little better but also starts to block some WCUs at a BOSR of 2 (i.e., 1.1% on average in partitions

[4] The maximum ESSD bandwidth is typically positively correlated with the storage capacity [7, 9, 66]. For example, in our used ESSD, the maximum bandwidth is half of the storage capacity (in GBs) plus 120, or 4000, whichever is smaller. Therefore, we can vary maximum ESSD bandwidth simply via adjusting the storage capacity.



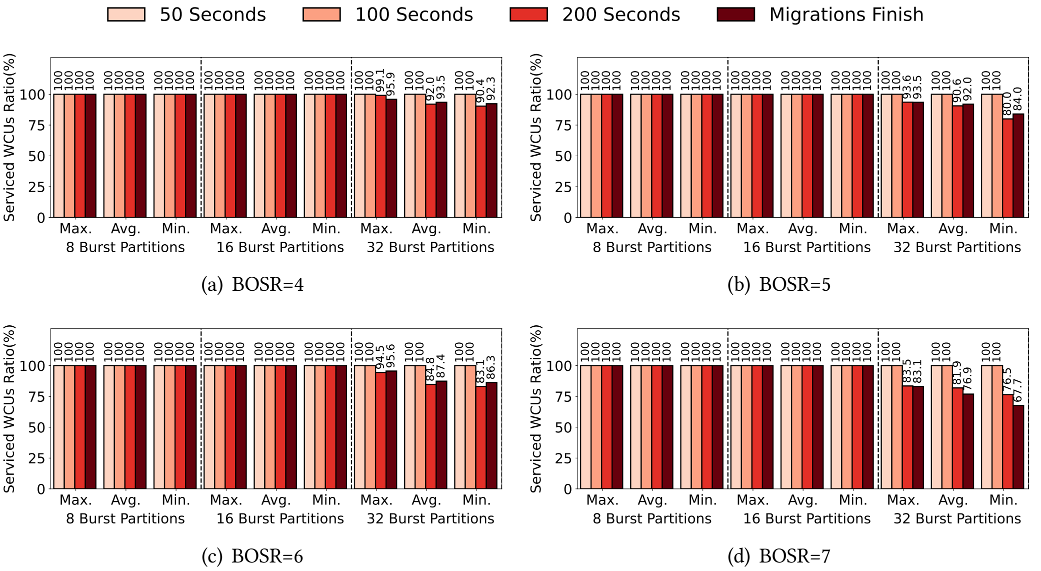
(a) Ratio of normally-serviced WCUs

(b) WCU latency

Fig. 10. **Performance in the ratio of normally-serviced WCUs and WCU latency under a workload with consistently high write loads. There are 32 partitions co-located in a node, and their total required background bandwidth is background over-subscription ratio (BOSR) times the current maximum ESSD bandwidth minus the fixed-reserved bandwidth for foreground tasks. For the ratio of normally-serviced WCUs, we present the maximum, average, and minimum values across the partitions. For the latency, we show the average, P99, and P99 values, which have been averaged across the partitions.**

and 9.9% at most in a partition), and at a BOSR of 3, an average of 15.8% of WCUs from partitions are unserviceable. When the first stage of the two-stage compaction deferral mechanism is also added (i.e., *RocksDB+Node+FirstStage*), the ratio of WCUs that can not be normally served is further reduced; at a BOSR of 3, an average of 7.1% of WCUs can not be admitted.

In Figure 10b, we further depict the latency of these evaluated schemes, where the average, P99, and P99.9 latency are presented. We can notice that FlexEngine significantly improves the latency in whether average, P99, or P99.9, all to sub-100 microseconds. Specifically, in comparison to the vanilla RocksDB, at a BOSR of 1.5, FlexEngine achieves 83.3% lower average latency, 98.2% lower P99 latency, and 97.7% lower P99.9 latency, respectively. When BOSR is increased to 3, the latency improvements are also substantial: the average latency is reduced by 97.3%, the P99 latency is reduced by 99.4%, and the P99.9 latency is reduced by 97.4%, respectively. Compared to the second-best scheme *RocksDB+Node+FirstStage*, FlexEngine achieves similarly good latency with BOSRs of 1.5 and 2 because both of them do not encounter write stalls yet. While at a BOSR of 3, FlexEngine reduces the average latency, P99 latency, and P99.9 latency by 88.1%, 99.3%, and 97.0%, respectively.



(a) BOSR=4

(b) BOSR=5

(c) BOSR=6

(d) BOSR=7

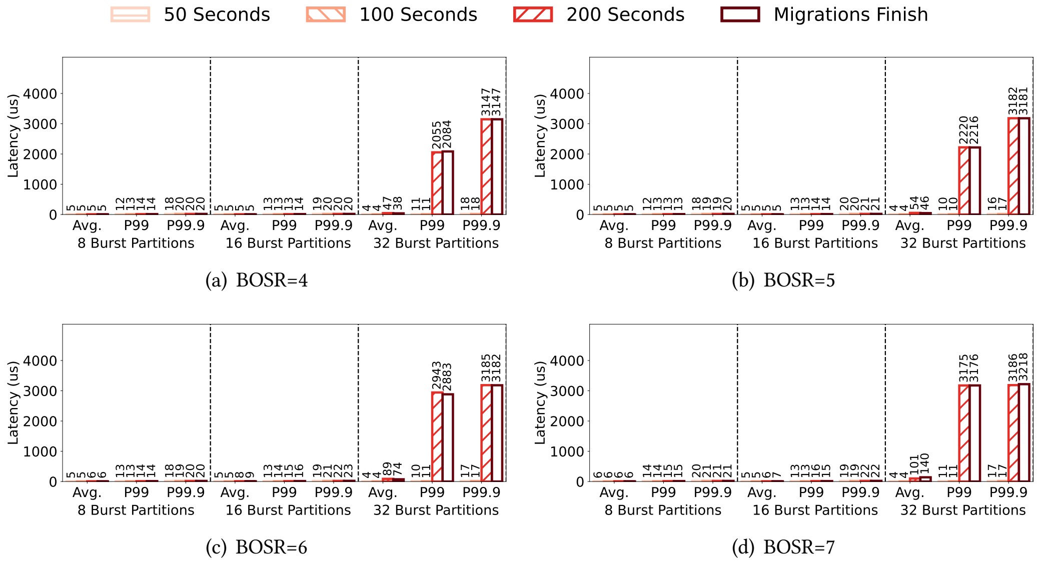
Fig. 11. **Performance in the ratio of normally-serviced WCUs with different background over-subscription ratios (BOSR), different ratios of burst partitions, and different durations to experience high user write loads. For the ratio of normally-serviced WCUs, we present the maximum, average, and minimum values across the partitions. 8, 16, and 32 burst partitions indicate that the ratios of burst partitions are 25%, 50%, 100%, respectively.**

We draw two conclusions from the above results and analysis. First, node-level I/O admission control and each stage of two-stage compaction deferral contribute to a certain degree to sustaining performance SLAs, in both the ratio of normally serviced WCUs as well as the latency. Second, FlexEngine, with these techniques incorporated, can effectively maintain performance SLAs, even when BOSR is 3 and all partitions consistently reach their highest write loads.

5.4 Performance with High Storage Bandwidth Over-Subscription

In this section, we aim to shed further light on the potential of FlexEngine in supporting high storage bandwidth over-subscription and improving cost-efficiency. To this end, we further enrich the experiments from three aspects: (1) configuring different BOSRs from 4 to 7, (2) evaluating different *ratios of burst partitions*, i.e., 25%, 50%, and 100%, and (3) testing different *durations to experience high user write loads*, i.e., 50s, 100s, 200s, and until the finish of tenant migrations (over 200s, described next).

Same as in Section 5.3.2, we co-locate 32 partitions in a node and preload each with 10M WCUs (each for a 1KB key-value pair). Each partition also has 60MB/s maximum available bandwidth for background tasks, and the total required background bandwidth is still 1920MB/s. Given the setups of BOSR, the maximum ESSD bandwidth minus foreground bandwidth now ranges from 640MB/s (at a BOSR of 3) to a mere 274MB/s (at a BOSR of 7). For each burst partition, the workload inserts 3K WCUs per second. While for each non-burst partition, the workload inserts $3K/BOSR$ WCUs per second. This is to control the actual required bandwidth of the non-burst partition exactly the same as the over-subscribed bandwidth. Assume the partition migration speed on average is 50MB/s, it requires over 200 seconds to completely migrate a 10GB partition (i.e., 10M*1KB WCU)



(a) BOSR=4

(b) BOSR=5

(c) BOSR=6

(d) BOSR=7

Fig. 12. **Latency with different background over-subscription ratios (BOSR), different ratios of burst partitions, and different durations to experience high user write loads. For latency, we show the average, P99, and P99 values, which have been averaged across the partitions. 8, 16, and 32 burst partitions indicate that the ratios of burst partitions are 25%, 50%, 100%, respectively.**

that still receives data inputs. With different durations to experience high user write loads (i.e., 50s, 100s, 200s, and until the finish of tenant migrations), the workload stops at the specific times and we record the metrics.

Figure 11 and 12 demonstrate the performance in the ratio of normally-serviced WCUs and the latency, respectively. First, from the dimension of *ratios of burst partitions*, we can see that FlexEngine can satisfactorily sustain performance SLAs with 50% burst partitions (i.e., there are 16 burst partitions), even when BOSR is as high as 7. Second, from the dimension of *durations to experience high user write loads*, FlexEngine can maintain performance SLAs if the high user write loads last for no more than 100 seconds, even when BOSR is as high as 7. Third, the results do reveal the limitations of FlexEngine, which may potentially sacrifice performance SLAs due to write stalls when the majority of partitions are burst and user write loads are consistently high for a long time (e.g., 200 seconds). This is because the proposed two-stage compaction deferral mechanism greatly lengthens the time to accumulate data in L0 and L1, but it can not always keep accumulating. When the I/O amplification of the second stage reaches the threshold that the highly over-subscribed storage bandwidth can not tolerate, write stalls unavoidably happen.

Nevertheless, we believe FlexEngine is competent enough to be deployed in real-world multi-tenant serverless cloud databases for the following two reasons. First, as shown in Section 5.3.2, FlexEngine can effectively maintain performance SLAs when BOSR is 3 and even all partitions consistently reach their highest write loads. Second, as shown in this section, FlexEngine can also sustain performance SLAs when BOSR is as high as 7 if the ratio of write-intensive partitions in the node is not too high, such as 50%.

6 Related Work

Multi-tenant serverless cloud databases. Several works explore the architectural designs of multi-tenant serverless cloud databases. Cao et al. propose a disaggregated cloud-native database architecture in Alibaba PolarDB Serverless with remote memory/storage pools and optimizations to match local performance while enabling multi-dimensional elasticity [23]. Poppe et al. introduce how Microsoft Azure proactively predicts database pause/resume patterns to minimize cold-start delays while optimizing resource reclamation thresholds in serverless SQL databases [53]. Zhang et al. introduce how PolarDB Serverless achieves seamless migration via transaction migration policy and read scale-out through strong consistency on secondary nodes, enabling uninterrupted scaling, low-overhead instance migration, and consistent high-performance reads [71]. Barnhart et al. introduce how Aurora Serverless balances high host utilization with elastic scaling via spare-capacity-aware placement, live migration, and token-bucket rate control [22].

In contrast with these works, we uncover a dilemma when the multi-tenant serverless cloud database architecture meets LSM tree as its storage engine, and propose a new LSM-tree-based key-value store to navigate this dilemma.

LSM tree. LSM tree has a vast design space with hundreds of tuning knobs and has been extensively studied in the past [58, 60]. Representative directions include optimizing point and range querying [27, 28, 30, 43, 49, 70], optimizing storage data layout [26, 36, 37, 46, 56, 65, 67], optimizing compactions [17, 31, 45, 59, 61, 68], and workload-aware performance tuning [29, 38, 47, 48].

Different from the existing works that are based on single-tenant data stores, to the best of our knowledge, FlexEngine is the first to investigate how to design an LSM-tree-based key-value store in the multi-tenant serverless cloud database.

Rate limiting. There are numerous classical algorithms for rate limiting, such as token bucket algorithm, leaky bucket algorithm, fixed window algorithm, and sliding window algorithm, each with its own pros and cons [16]. Among them, FlexEngine (as well as RocksDB) opts for the token bucket algorithm mainly due to its responsiveness to traffic bursts through accumulated tokens, which is crucial for synchronous tasks.

Unlike rate limiting within an application, a body of research optimizes distributed rate limiting (DRL) in cross-node network environments with high communication overhead [24, 35, 55, 63]. By distributing rate-limiting decisions across multiple nodes, DRL achieves high scalability but also faces challenges in accuracy and responsiveness to traffic changes. Moreover, there is also a series of studies on rate limiting dedicated to sharing [20, 39, 52, 62] and scalability [25, 34, 54, 57] in cloud datacenters.

7 Conclusion

Over the past few years, multi-tenant serverless cloud databases have experienced an unparalleled surge in popularity. Yet there have been no prior works showing if and how to build an LSM-tree-based cloud database in this promising architecture. In this paper, we reveal a critical trade-off between performance SLAs versus storage bandwidth over-subscription for cost efficiency, and present FlexEngine to navigate this dilemma. Experimental results show that FlexEngine can promise consistent performance SLAs for users while significantly improving the storage bandwidth over-subscription capability.

Acknowledgments

We sincerely thank the anonymous reviewers for their constructive feedback, which substantially helps us improve this paper.

References

[1] 2017. Auto-tuned Rate Limiter in RocksDB. https://rocksdb.org/blog/2017/12/18/17-auto-tuned-rate-limiter.html.

[2] 2017. Wikipedia of Token Bucket Algorithm. https://en.wikipedia.org/wiki/Token_bucket.

[3] 2017. Wikipedia of Weighted Fair Queueing. https://en.wikipedia.org/wiki/Weighted_fair_queueing.

[4] 2021. Write Stalls in RocksDB. https://github.com/facebook/rocksdb/wiki/Write-Stalls.

[5] 2022. Rate Limiter in RocksDB. https://github.com/facebook/rocksdb/wiki/rate-limiter.

[6] 2023. RocksDB Tuning Guide. https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide.

[7] 2025. Alibaba Cloud ESSD. https://www.alibabacloud.com/help/en/ecs/user-guide/essds.

[8] 2025. Alibaba Tair Serverless KV. https://www.alibabacloud.com/help/en/redis/product-overview/tair-serverless-kv/?spm=a2c63.p38356.help-menu-26340.d_0_0_2.7015bac8Fx96uA&scm=20140722.H_2871463._.OR_help-T_intl~en-V_1.

[9] 2025. Amazon AWS ESSD. https://docs.aws.amazon.com/ebs/latest/userguide/ebs-volumes.html.

[10] 2025. Apache Cassandra. https://cassandra.apache.org/_/index.html.

[11] 2025. Apache HBase. https://hbase.apache.org/.

[12] 2025. Google LevelDB. https://github.com/google/leveldb.

[13] 2025. Meta RocksDB. http://rocksdb.org/.

[14] 2025. PingCAP TiKV. https://tikv.org/.

[15] 2025. Postgres' Vaccum. https://www.postgresql.org/docs/current/sql-vacuum.html.

[16] 2025. Rate Limiting Algorithms - System Design. https://www.geeksforgeeks.org/system-design/rate-limiting-algorithms-system-design/.

[17] Wail Y Alkowaileet, Sattam Alsubaiee, and Michael J Carey. 2019. An LSM-based Tuple Compaction Framework for Apache AsterixDB (Extended Version). *arXiv preprint arXiv:1910.08185* (2019).

[18] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, et al. 2019. Socrates: The new sql server in the cloud. In *Proceedings of the 2019 International Conference on Management of Data*. 1743–1756.

[19] Pankaj Arora, Surajit Chaudhuri, Sudipto Das, Junfeng Dong, Cyril George, Ajay Kalhan, Arnd Christian König, Willis Lang, Changsong Li, Feng Li, et al. 2023. Flexible Resource Allocation for Relational Database-as-a-Service. *Proceedings of the VLDB Endowment* 16, 13 (2023), 4202–4215.

[20] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. 2011. Towards predictable datacenter networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*. 242–253.

[21] 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 *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 753–766.

[22] Bradley Barnhart, Marc Brooker, Daniil Chinenkov, Tony Hooper, Jihoun Im, Prakash Chandra Jha, Tim Kraska, Ashok Kurakula, Alexey Kuznetsov, Grant McAlister, et al. 2024. Resource Management in Aurora Serverless. *Proceedings of the VLDB Endowment* 17, 12 (2024), 4038–4050.

[23] Wei Cao, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, et al. 2021. Polardb serverless: A cloud native database for disaggregated data centers. In *Proceedings of the 2021 International Conference on Management of Data*. 2477–2489.

[24] Lilong Chen, Xiaochong Jiang, Xiang Hu, Tianyu Xu, Ye Yang, Xing Li, Bingqian Lu, Chengkun Wei, and Wenzhi Chen. 2024. CMDRL: A Markovian Distributed Rate Limiting Algorithm in Cloud Networks. In *Proceedings of the 8th Asia-Pacific Workshop on Networking*. 59–66.

[25] Zhongjie Chen, Yingchen Fan, Kun Qian, Qingkai Meng, Ran Shu, Xiaoyu Li, Yiran Zhang, Bo Wang, Wei Li, and Fengyuan Rent. 2025. ScalaTap: Scalable Outbound Rate Limiting in Public Cloud. In *IEEE INFOCOM 2025-IEEE Conference on Computer Communications*. IEEE, 1–10.

[26] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2020. From {WiscKey} to bourbon: A learned index for {Log-Structured} merge trees. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 155–171.

[27] 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*. 79–94.

[28] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2018. Optimal bloom filters and adaptive merging for LSM-trees. *ACM Transactions on Database Systems (TODS)* 43, 4 (2018), 1–48.

[29] 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*. 449–466.

[30] Niv Dayan and Moshe Twitto. 2021. Chucky: A succinct cuckoo filter for LSM-tree. In *Proceedings of the 2021 International Conference on Management of Data*. 365–378.

[31] Niv Dayan, Tamar Weiss, Shmuel Dashevsky, Michael Pan, Edward Bortnikov, and Moshe Twitto. 2022. Spooky: granulating LSM-tree compactions correctly. *Proceedings of the VLDB Endowment* 15, 11 (2022), 3071–3084.

[32] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. Rocksdb: Evolution of development priorities in a key-value store serving large-scale applications. *ACM Transactions on Storage (TOS)* 17, 4 (2021), 1–32.

[33] Mostafa Elhemali, Niall Gallagher, Bin Tang, Nick Gordon, Hao Huang, Haibo Chen, Joseph Idziorek, Mengtian Wang, Richard Krog, Zongpeng Zhu, et al. 2022. Amazon {DynamoDB}: A scalable, predictably performant, and fully managed {NoSQL} database service. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 1037–1048.

[34] Yongchao He, Wenfei Wu, Xuemin Wen, Haifeng Li, and Yongqiang Yang. 2021. Scalable on-switch rate limiters for the cloud. In *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*. IEEE, 1–10.

[35] Xiang Hu, Tianyu Xu, Lilong Chen, Xiaochong Jiang, Ye Yang, Liming Ye, Xu Wang, Yilong Lv, Chenhao Jia, Yongwang Wu, et al. 2025. Distributed Rate Limiting under Decentralized Cloud Networks. *IEEE Transactions on Mobile Computing* (2025).

[36] Haoyu Huang and Shahram Ghandeharizadeh. 2021. Nova-LSM: a distributed, component-based LSM-tree key-value store. In *Proceedings of the 2021 International Conference on Management of Data*. 749–763.

[37] Stratos Idreos and Mark Callaghan. 2020. Key-value storage engines. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2667–2672.

[38] Stratos Idreos, Niv Dayan, Wilson Qin, Mali Akmanalp, Sophie Hilgard, Andrew Ross, James Lennon, Varun Jain, Harshita Gupta, David Li, et al. 2019. Design Continuums and the Path Toward Self-Designing Key-Value Stores that Know and Learn.. In *CIDR*.

[39] Keon Jang, Justine Sherry, Hitesh Ballani, and Toby Moncaster. 2015. Silo: Predictable message latency in the cloud. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. 435–448.

[40] Ram Kesavan, David Gay, Daniel Thevessen, Jimit Shah, and C Mohan. 2023. Firestore: The nosql serverless database for the application developer. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 3376–3388.

[41] Arnd Christian König, Yi Shan, Tobias Ziegler, Aarati Kakaraparthy, Willis Lang, Justin Moeller, Ajay Kalhan, and Vivek Narasayya. 2022. Tenant placement in over-subscribed database-as-a-service clusters. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2559–2571.

[42] Guoliang Li, Haowen Dong, and Chao Zhang. 2022. Cloud databases: New techniques, challenges, and opportunities. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3758–3761.

[43] Yongkun Li, Chengjin Tian, Fan Guo, Cheng Li, and Yinlong Xu. 2019. {ElasticBF}: Elastic bloom filter with hotness awareness for boosting read performance in large {Key-Value} stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 739–752.

[44] Junkai Liang and Yunpeng Chai. 2021. CruiseDB: An LSM-tree key-value store with both better tail throughput and tail latency. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 1032–1043.

[45] Yuhong Liang, Yingjia Wang, Tsun-Yu Yang, Matias Bjørling, and Ming-Chang Yang. 2025. ZonesDB: Building Write-Optimized and Space-Adaptive Key-Value Store on Zoned Storage with Fragmented LSM Tree. *ACM Transactions on Storage* 21, 2 (2025), 1–24.

[46] 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 (TOS)* 13, 1 (2017), 1–28.

[47] Chen Luo and Michael J Carey. 2019. On performance stability in LSM-based storage systems (extended version). *arXiv preprint arXiv:1906.09667* (2019).

[48] Chen Luo and Michael J Carey. 2020. Breaking down memory walls: adaptive memory management in LSM-based storage systems. *Proceedings of the VLDB Endowment* 14, 3 (2020), 241–254.

[49] Siqiang Luo, Subarna Chatterjee, Rafael Ketsetsidis, Niv Dayan, Wilson Qin, and Stratos Idreos. 2020. Rosetta: A robust space-time optimized range filter for key-value stores. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2071–2086.

[50] Vivek Narasayya and Surajit Chaudhuri. 2022. Multi-tenant cloud data services: state-of-the-art, challenges and opportunities. In *Proceedings of the 2022 International Conference on Management of Data*. 2465–2473.

[51] 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.

[52] Lucian Popa, Gautam Kumar, Mosharaf Chowdhury, Arvind Krishnamurthy, Sylvia Ratnasamy, and Ion Stoica. 2012. FairCloud: Sharing the network in cloud computing. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*. 187–198.

[53] Olga Poppe, Qun Guo, Willis Lang, Pankaj Arora, Morgan Oslake, Shize Xu, and Ajay Kalhan. 2022. Moneyball: proactive auto-scaling in Microsoft Azure SQL database serverless. *Proceedings of the VLDB Endowment* 15, 6 (2022), 1279–1287.

[54] Sivasankar Radhakrishnan, Yilong Geng, Vimalkumar Jeyakumar, Abdul Kabbani, George Porter, and Amin Vahdat. 2014. {SENIC}: Scalable {NIC} for {End-Host} rate limiting. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. 475–488.

[55] Barath Raghavan, Kashi Vishwanath, Sriram Ramabhadran, Kenneth Yocum, and Alex C Snoeren. 2007. Cloud control with distributed rate limiting. *ACM SIGCOMM Computer Communication Review* 37, 4 (2007), 337–348.

[56] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 497–514.

[57] Ahmed Saeed, Nandita Dukkipati, Vytautas Valancius, Vinh The Lam, Carlo Contavalli, and Amin Vahdat. 2017. Carousel: Scalable traffic shaping at end hosts. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 404–417.

[58] Subhadeep Sarkar and Manos Athanassoulis. 2022. Dissecting, designing, and optimizing LSM-based data stores. In *Proceedings of the 2022 International Conference on Management of Data*. 2489–2497.

[59] Subhadeep Sarkar, Kaijie Chen, Zichen Zhu, and Manos Athanassoulis. 2022. Compactionary: A dictionary for LSM compactions. In *Proceedings of the 2022 International Conference on Management of Data*. 2429–2432.

[60] Subhadeep Sarkar, Niv Dayan, and Manos Athanassoulis. 2023. The LSM design space and its read optimizations. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 3578–3584.

[61] Subhadeep Sarkar, Dimitris Staratzis, Zichen Zhu, and Manos Athanassoulis. 2022. Constructing and analyzing the LSM compaction design space (updated version). *arXiv preprint arXiv:2202.04522* (2022).

[62] Alan Shieh, Srikanth Kandula, Albert Greenberg, Changhoon Kim, and Bikas Saha. 2011. Sharing the data center network. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*.

[63] Rade Stanojevic and Robert Shorten. 2009. Load balancing vs. distributed rate limiting: an unifying framework for cloud control. In *2009 IEEE International Conference on Communications*. IEEE, 1–6.

[64] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1041–1052.

[65] Yingjia Wang, Tao Lu, Yuhong Liang, Xiang Chen, and Ming-Chang Yang. 2025. Reviving In-Storage Hardware Compression on ZNS SSDs through Host-SSD Collaboration. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 608–623.

[66] Yingjia Wang and Ming-Chang Yang. 2025. The Unwritten Contract of Cloud-based Elastic Solid-State Drives. In *2025 62nd ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–7.

[67] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. 2015. {LSM-trie}: An {LSM-tree-based}{Ultra-Large}{Key-Value} store for small data items. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 71–82.

[68] Ting Yao, Jiguang Wan, Ping Huang, Xubin He, Fei Wu, and Changsheng Xie. 2017. Building efficient key-value stores via a lightweight compaction tree. *ACM Transactions on Storage (TOS)* 13, 4 (2017), 1–28.

[69] Jinghuan Yu, Sam H Noh, Young-ri Choi, and Chun Jason Xue. 2023. {ADOC}: Automatically Harmonizing Dataflow Between Components in {Log-Structured}{Key-Value} Stores for Improved Performance. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. 65–80.

[70] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2018. Surf: Practical range query filtering with fast succinct tries. In *Proceedings of the 2018 International Conference on Management of Data*. 323–336.

[71] Yingqiang Zhang, Xinjun Yang, Hao Chen, Feifei Li, Jiawei Xu, Jie Zhou, Xudong Wu, and Qiang Zhang. 2024. Towards a Shared-Storage-Based Serverless Database Achieving Seamless Scale-Up and Read Scale-Out. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 5119–5131.

[72] Yuanhui Zhou, Jian Zhou, Shuning Chen, Peng Xu, Peng Wu, Yanguang Wang, Xian Liu, Ling Zhan, and Jiguang Wan. 2023. Calcspar: A {Contract-Aware}{LSM} Store for Cloud Storage with Low Latency Spikes. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 451–465.

Received July 2025; revised October 2025; accepted November 2025