TurboEngine: A Cost-Effective LSM-Tree-based Storage Engine for Multi-Tenant Serverless Cloud Databases

Yingjia Wang[1], Caixin Gong[2], Yuhong Liang[3], Zhisheng Hu[1], Huan Liu[2], Guoyun Zhu[2]✉, Wenchao Zhou[2], Wei Zhang[2], Feifei Li[2], Dingwen Tao[4], and Ming-Chang Yang[1]✉

[1] *The Chinese University of Hong Kong* [2] *Alibaba Group* [3] *Great Bay University* [4] *ICT, CAS*

{yjwang22,zshu23,mcyang}@cse.cuhk.edu.hk yhliang@gbu.edu.cn taodingwen@ict.ac.cn

{caixin.gcx,qianyi.lh,zongdai,zwc231487,zwei,lifeifei}@alibaba-inc.com

ABSTRACT

Multi-tenant serverless cloud databases increasingly rely on storage bandwidth over-subscription to improve cost-effectiveness, but LSM-tree-based storage engines face a fundamental challenge: sustaining tenant performance SLAs under high over-subscription ratios. We identify that the core capability demanded during bandwidth shortages is to reliably bound the number of SSTables (SSTs) in a search path while maintaining low write amplification (WA). However, existing solutions lack both principled control over searched SSTs and systematic WA optimization.

We present TurboEngine, a cost-effective LSM-tree-based storage engine for multi-tenant serverless cloud databases. TurboEngine advances from our prior work, FlexEngine, through two key designs. First, TurboEngine places primary partitions on cloud local SSDs to ensure low latency for performance-critical user reads; meanwhile, it places secondary partitions on cloud elastic SSDs to support fast snapshot-based tenant migration. Second, TurboEngine introduces the concept of "LSM-in-LSM", temporarily reorganizing the top levels of the LSM tree into a small, self-contained LSM tree during storage bandwidth shortages, so as to provide principled bounds on the number of searched SSTs to safeguard scan performance while analytically minimizing WA. Extensive experiments show that TurboEngine reduces deployment costs by 22.5% with a 1:1 read-to-write traffic ratio, and sustains 6× storage bandwidth over-subscription without compromising tenant performance SLAs.

PVLDB Reference Format:

Yingjia Wang, Caixin Gong, Yuhong Liang, Zhisheng Hu, Huan Liu, Guoyun Zhu, Wenchao Zhou, Wei Zhang, Feifei Li, Dingwen Tao, and Ming-Chang Yang. TurboEngine: A Cost-Effective LSM-Tree-based Storage Engine for Multi-Tenant Serverless Cloud Databases. PVLDB, 20(4): XXX-XXX, 2026.

doi:XX.XX/XXX.XX

PVLDB Artifact Availability:

Artifacts not disclosed.

1 INTRODUCTION

The rapid growth of *cloud databases* has fundamentally reshaped today's data management landscape, with modern architectures increasingly centered on two defining paradigms: *multi-tenancy* and *serverless*. Multi-tenancy consolidates multiple logically isolated tenants onto shared physical resources [14, 15, 19, 34, 45, 59, 75, 80], enabling providers to over-subscribe resources, i.e., commit more resources to tenants than are physically available. Since most tenants exhibit sporadic and low average utilization, *over-subscription* becomes a key strategy for cost-effectiveness, significantly increasing resource utilization and reducing deployment cost [15, 17, 45]. Building on this, the serverless model further abstracts infrastructure management by automatically provisioning and scaling resources on demand, allowing users to focus solely on data operations under pay-as-you-go pricing [17, 19, 34, 59, 103].

Nevertheless, as cloud databases adopt this promising architecture, those employing *LSM trees* as the storage engine face a critical challenge: *while ensuring consistent performance SLAs for tenants, it is difficult to achieve high storage bandwidth over-subscription, which is crucial for cost-effectiveness* [80]. The root cause of this challenge lies in the unique *compaction* processes inherent to LSM trees, which run asynchronously in the background and consume a significant amount of bandwidth due to intensive data reorganization. Consequently, when multiple partitions trigger compactions simultaneously, system-wide storage bandwidth shortages easily occur and jeopardize tenant performance SLAs. This forces providers to adopt very conservative over-subscription, thereby severely limiting cost-effectiveness [80].

FlexEngine [80] represents the state-of-the-art step toward addressing this challenge, first establishing that storage bandwidth over-subscription is non-negotiable and proposing I/O admission control and compaction deferral designs. At its core is a *two-level I/O admission control framework* that makes LSM trees practical. The key idea is twofold. First, at the partition level, FlexEngine manages performance-critical foreground I/Os (e.g., user reads and WAL writes) and bandwidth-hungry background I/Os (e.g., flushes and compactions) separately, reserving fixed bandwidth for the former and dynamically tuning bandwidth for the latter. Second, at the node level, FlexEngine unconditionally prioritizes foreground I/Os and ensures the total bandwidth of submitted I/Os does not exceed the physical limit, thereby guaranteeing consistently low latency. In addition, FlexEngine defers high-level compaction (starting from L1-to-L2) when storage bandwidth is insufficient and tenant migration is in progress, so as to conserve bandwidth and maintain tenant performance under moderate bandwidth over-subscription. Unfortunately, FlexEngine has limited effectiveness: it not only causes a significant performance degradation for scan-intensive tenants, but also consumes too much bandwidth, limiting the ability for over-subscription (see Section 3.2).

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.

Proceedings of the VLDB Endowment, Vol. 20, No. 4 ISSN 2150-8097.
doi:XX.XX/XXX.XX

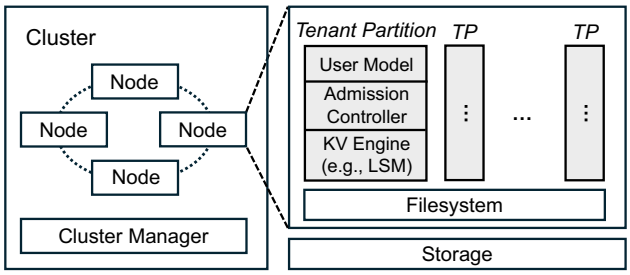


Figure 1: The typical architecture of a multi-tenant serverless cloud database.

Based on our in-depth investigation, we point out that the *core capability* demanded of LSM-tree-based storage engines here is to *reliably bound the number of SSTs in a search path while incurring low write amplification (WA)*[1] *during storage bandwidth shortages*. The rationales behind this are explained as follows. On the one hand, when bandwidth is scarce, even though high-level compactions are already suspended [16, 80, 105], L0-to-L1 compactions still lag behind and inevitably accumulate SSTs in L0. Since L0 SSTs are not sorted, an increase in the number of L0 SSTs can significantly degrade scan performance, as each scan requires locating and merging all SSTs with overlapping keys. On the other hand, since reorganizing L0 SSTs requires compactions that consume bandwidth, lower WA means less bandwidth is needed to keep the number of L0 SSTs bounded. This allows the system to tolerate wider bandwidth deficits without violating performance SLAs, thereby supporting higher over-subscription.

In this paper, we present *TurboEngine*[2], a cost-effective LSM-tree-based key-value storage engine for multi-tenant serverless cloud databases. TurboEngine introduces two key designs that jointly achieve the above core capability, pushing cloud database cost-effectiveness to a significantly higher level.

First, TurboEngine *co-deploys cloud local SSDs (LDs) and cloud elastic SSDs (EDs) within a node, storing primary partitions on LDs and secondary partitions on EDs, respectively.* This approach is supported by two observations. First, our performance analysis of RocksDB across the two storage types reveals a key read-write asymmetry: ED incurs 3–6× higher read latency than LD, yet delivers comparable write latency (see Section 4). Second, in the multi-replica model, secondary partitions are essentially write-only, since they process only replicated writes and typically do not handle user reads. These two observations indicate that placing secondary partitions on EDs incurs negligible performance loss while unlocking a key advantage: the *snapshot functionality* provided by Elastic Block Storage (EBS). Snapshot-based tenant migration is significantly faster than network-based tenant migration (e.g., ~50 seconds vs. ~200 seconds in our environments to migrate a 10GB partition). Since the duration of tenant migration largely determines the maximum duration of bandwidth shortages (see Section 2.2), a shorter migration window means a shorter bandwidth shortage, which in turn reduces the extra data accumulated in an LSM tree and the associated WA, thereby enabling higher bandwidth over-subscription.

[1]In this paper, WA refers to the write amplification at the LSM tree level.
[2]TurboEngine has been integrated into Alibaba Tair Serverless KV [8].

Second, TurboEngine introduces the concept of *"LSM-in-LSM"*: when storage bandwidth becomes insufficient, it *temporarily reorganizes L0 and L1 into a small, self-contained LSM tree, called sub-LSM.* Unlike FlexEngine's two-stage ad-hoc conversion (see Section 3.2), sub-LSM, thanks to its inherent LSM tree structure, can provide principled guarantees on the number of searched SSTs and WA. More specifically, the level count L of sub-LSM is set to match the normal-state searched SST count, to ensure that scan performance is comparable to that under normal conditions. Since the maximum extra data volume during storage bandwidth shortages is known a priori in serverless databases (see Section 2.2), TurboEngine applies the compaction WA formula derived in Spooky [29] to analytically solve for the optimal size ratio T that can minimize WA.

We summarize our contributions in this paper as follows:

- We reframe the cost-effectiveness challenge of LSM-tree-based multi-tenant serverless cloud databases by identifying the fundamental goal: during storage bandwidth shortages, the LSM-tree-based storage engine must reliably bound the number of SSTs in a search path while incurring low WA. This formulation advances the focus from ad-hoc write stall avoidance to principled performance guarantees, providing a clearer foundation for systematic storage engine design.
- We first characterize the performance of cloud local SSDs (LDs) and cloud elastic SSDs (EDs) from the LSM tree perspective, uncovering a key read/write asymmetry. This finding reveals heterogeneous cloud block storage as an underexplored yet impactful design dimension for LSM-tree-based cloud databases.
- We propose TurboEngine with two key designs: (1) LD-ED co-deployment, a heterogeneous-storage optimization that leverages both low latency of LDs and snapshot-based fast migrations of EDs, and (2) sub-LSM-tree-based compaction deferral, which replaces ad-hoc compaction heuristics with a structurally grounded approach, thereby providing a principled bound of searched SST count and WA minimization.
- We implement TurboEngine based on a commercially-deployed RocksDB [12] and extensively evaluate it with both production traces and micro-level workloads. Experiments show that TurboEngine reduces deployment costs by 22.5% with a 1:1 read-to-write ratio, and supports up to 6× storage bandwidth over-subscription without compromising tenant performance SLAs.

In the remainder of this paper, we first introduce the background in Section 2 and the motivation in Section 3. Next, in Section 4, we analyze RocksDB's performance on LD and ED. In Section 5, we detail the designs of TurboEngine. The evaluation, related work, and conclusion are presented in Sections 6, 7, and 8, respectively.

2 BACKGROUND

2.1 Multi-Tenant Serverless Cloud Database

Multi-tenancy is a foundational principle in cloud databases to achieve high cost-effectiveness, enabling the consolidation of multiple tenants that share the same physical resources while maintaining their logical isolation [14, 15, 19, 34, 45, 59, 75]. The widespread adoption of multi-tenancy is motivated by the observation that most tenants exhibit low average resource utilization. This makes it feasible for the total resources committed to tenants to exceed the physical capacity, which is known as *over-subscription* [15, 17, 45].

Over-subscription significantly improves overall resource utilization, thereby reducing deployment costs and enabling more competitive pricing for customers.

The rapid evolution of serverless computing has given rise to a new *serverless* architecture for cloud databases [15, 17, 19, 34, 43, 46, 59, 61, 103]. Serverless databases leverage the elasticity of the cloud to allocate resources dynamically in response to real-time workload demands, eliminating the need for manual provisioning. By automating scaling, maintenance, and optimization, these systems substantially reduce the operational burden on users. Moreover, their pay-as-you-go pricing model ensures cost-efficiency by charging only for actual resource consumption, making serverless databases particularly suitable for variable workloads.

Figure 1 illustrates the typical architecture of a multi-tenant serverless cloud database. User data is physically stored in sharded *clusters*, each comprising a set of *nodes* that host database tenants. Each cluster is managed by a cluster manager responsible for tenant placement, load balancing, failover handling, and other administrative tasks [59]. For performance scalability and high availability, each tenant is partitioned, and each *tenant partition* is replicated across multiple nodes. Within a node, numerous partitions from different tenants are consolidated inside a virtual machine, where they share resources such as storage, CPU, memory, and network. At the tenant partition level, an *admission controller* enforces per-partition resource limits to minimize performance interference and enable rapid scaling. In practice, this is achieved by capping the maximum number of *read capacity units (RCUs)* and *write capacity units (WCUs)*. Each CU here corresponds to a read or write operation of an item up to a fixed size.

2.2 Resource Violation

When resources in a node are over-subscribed, the total resource demand from tenant partitions may exceed the physical capacity, leading to *resource violations* [45]. Such violations must be avoided, as resource shortages can degrade service quality or even cause service unavailability. In practice, a node is considered resource-violated when the total demand for any resource exceeds a predefined high-watermark threshold (e.g., 80%). Once this occurs, a subset of resource-intensive tenant partitions on the affected node is selected and migrated to other nodes. This process repeats until the total resource demand falls below the threshold. The use of a high-watermark threshold, rather than the absolute physical capacity (i.e., 100%), accounts for the non-instantaneous nature of tenant migration. This allows migrations to be completed before resources are fully exhausted and service is impacted.

In multi-tenant serverless databases, elasticity is a core service guarantee: when workloads shift or node-level resource violations occur, the system must complete auto-scaling through tenant migration before user service is affected. This makes tenant migration time a first-order constraint. To make a migration converge while the partition continues receiving writes, the full-data migration speed must be significantly higher than the per-partition write rate; otherwise, incremental writes may accumulate faster than they are copied. Since migration speed is bounded by the underlying storage and network stack, production systems further keep migration time bounded by restricting the maximum size of each tenant partition, rather than allowing a partition to grow arbitrarily large as in standalone database deployments. Notably, migrations are not performed on the resource-violated node itself, but on nodes hosting the replicas of the partitions being migrated, so as not to exacerbate local resource contention. Once migration completes and a new partition replica is successfully established, traffic is redirected to the new replica. The original node can then release the resources held by the previous partition.

2.3 Log-Structured Merge (LSM) Tree

The *log-structured merge (LSM) tree* [60] has established itself as a foundational component in key-value storage, underpinning a wide spectrum of data systems ranging from embedded edge devices to large-scale distributed databases. Numerous production LSM-tree-based key-value stores have been adopted, including RocksDB [12], LevelDB [11], Cassandra [9], HBase [10], and TiKV [13].

The LSM tree organizes data files across multiple levels to achieve high write throughput while maintaining reasonable read performance. In this hierarchy, newly written data is initially placed in L0 and progressively compacted into deeper levels over time. Incoming writes are first buffered and grouped in an in-memory component known as *memtable*. Once a memtable reaches its capacity threshold, it becomes immutable, allowing a new memtable to take its place for subsequent writes. The immutable memtable is then *flushed* to storage as a *sorted string table (SST)* after sorting its key-value pairs. When the number or total size of SSTs in a given level exceeds a predefined threshold, the LSM tree triggers a *compaction*. This process selects SSTs from that level and merges them with all overlapping SSTs from the next deeper level. The resulting new SSTs are written to the next level, while the obsolete ones are discarded. Compaction serves multiple essential purposes: it eliminates redundant or outdated data, reduces storage overhead, and sustains a balanced multi-level structure conducive to efficient reads. However, due to the nature of compactions, the total amount of data written to storage is greater than the actual amount of data written by the user (typically dozens of times more), which consumes a significant amount of storage bandwidth.

The LSM tree handles user reads and write-ahead logging (WAL) *synchronously*, whereas flushes and compactions are performed *asynchronously* in the background. A read operation (e.g., point read or scan) first searches in-memory memtables, which hold the most recent updates. If the target key is not found there, the lookup proceeds through SSTs from L0 to the deepest level, ensuring the latest version is retrieved. In cases where the required data is not cached in memory, it is fetched directly from storage in a synchronous manner. In contrast, a write operation is acknowledged as soon as the key-value pair is stored in both WAL and the active memtable, with subsequent tasks such as flushes and compactions deferred to the background.

2.4 Cloud Block Storage

For cloud database providers, cloud block storage is a widespread choice for persisting tenants' data, and it typically offers two options: *cloud local storage* and *elastic block storage (EBS)*. Table 1 summarizes the comparison between LD and ED.

Table 1: Comparison of cloud local SSD and elastic SSD.

Property	LD	ED
I/O latency	Low (near-physical)	High (3–6× for LSM tree reads)
Data durability	Ephemeral	Persistent
Snapshot support	No	Yes
Performance model	Bundled with VM	Pay-per-use
Cost model	Bundled with VM	Pay-per-use

Cloud local SSDs (LDs), offered by cloud local storage, refer to storage volumes that are provisioned directly from physical SSDs attached to compute servers. Their performance and capacity are coupled to the compute VM configuration, with costs bundled into VM pricing. Thanks to low-overhead virtualization, LDs can deliver near-physical I/O latency (e.g., with only a few microseconds drop [93]). However, data stored on LDs is ephemeral, i.e., no longer retained beyond the lifecycle of the user's VM. This indicates that data is lost upon VM termination or in unexpected events such as hardware failures.

In contrast, EBS provides *cloud elastic SSDs (EDs)*, which offer network-attached volumes decoupled from compute servers, enabling on-demand scaling of both performance (e.g., bandwidth and IOPS) and storage capacity. EDs enable high availability and reliability thanks to multi-replica redundancy within EBS, and also support rapid snapshotting for efficient backups, recovery, and migrations [102]. However, due to the inherent network and layered software processing overhead within EBS, EDs, even with the most advanced types, suffer from much higher I/O latency than LDs [83]. We provide further experiments and analysis on performance from the LSM tree perspective in Section 4.

3 MOTIVATION

Multi-tenant serverless cloud databases have garnered increasing attention in both academia and industry, owing to their superior cost savings and operational simplicity. Nevertheless, *evolving LSM-tree-based cloud databases toward the multi-tenant serverless architecture is challenging for high cost-effectiveness* [80].

In this section, we dissect the root cause of this challenge (Section 3.1), analyze the limitations of existing solutions (Section 3.2), and present the key insights that drive our design (Section 3.3).

3.1 Dissecting the Challenge

Essentially, the LSM tree prioritizes rapid data writes, deferring subsequent data reorganization to asynchronous flushes and compactions performed in the background. However, due to high write amplification of compactions, this can cause sharp spikes in system-wide storage bandwidth usage, even if only a few partitions receive moderate user write loads.

A common solution to this is to cap each partition's bandwidth usage with a *maximum available bandwidth* limit, which helps meet performance service-level agreements (SLAs) while also stabilizing the overall system storage bandwidth [34]. However, even with such bandwidth caps, *storage bandwidth shortages* can still arise when multiple partitions saturate their allocated bandwidth simultaneously. Here, storage bandwidth shortage is defined as a state where the total storage bandwidth demand from partitions exceeds the physical storage bandwidth.

Storage bandwidth shortage can easily jeopardize performance SLAs for tenants and even cause service unavailability. Thus, cloud database providers must resolve this as quickly as possible, typically through *tenant migration*. In serverless databases, the duration of tenant migration is inherently bounded to ensure service elasticity (see Section 2.2). Therefore, for LSM-tree-based storage engines, the goal is to carefully utilize the scarce bandwidth within this finite period to safeguard tenant performance.

To achieve this, we point out that the core capability required is to *reliably bound the maximum number of SSTs in the search path while requiring low write amplification (WA) for this process.*

On the one hand, restricting the maximum number of "searched" SSTs is essential for tenant performance, particularly for scans. The primary obstacle stems from L0, where, due to insufficient bandwidth, L0-to-L1 compactions cannot be timely completed, which causes the number of L0 SSTs to proliferate. Since L0 SSTs are mutually unordered, scan operations must seek and merge all SSTs with overlapping key ranges, thereby significantly degrading latency. On the other hand, any data reorganization during storage bandwidth shortages consumes limited bandwidth, and the efficiency here is governed by WA. A reorganization approach with low WA that can bound the number of L0 SSTs enables the system to tolerate wider bandwidth deficits without performance violations, thereby enabling higher over-subscription in practice.

3.2 Why Existing Solutions Fall Short

Unfortunately, existing research on LSM trees does *not* achieve the aforementioned capability. Some studies [16, 80, 105] suspend high-level compactions from L1-to-L2 onward to concentrate scarce bandwidth on the most critical flushes and L0-to-L1 compactions. SILK [16] and RocksDB I/O rate limiter [6] further prioritize flushes over L0-to-L1 compactions. Another direction is to prevent latency spikes during storage bandwidth shortages: Calcspar [105] and FlexEngine [80] propose holistically managing I/Os that can be submitted to storage, ensuring that the bandwidth required for submitted ones always remains within the maximum.

FlexEngine [80] is the closest work to our expected capability, and to our knowledge, is the only LSM-tree-based storage engine for multi-tenant serverless cloud databases. FlexEngine establishes a two-level (i.e., partition- and node-level) I/O admission framework, with the above techniques well incorporated (see Section 5.3). Moreover, when storage bandwidth becomes insufficient, FlexEngine proposes a *two-stage compaction deferral* mechanism.

Specifically, in the *first* stage, all compactions except those from L0 to L1 are suspended. However, pausing higher-level compactions causes SSTs to accumulate in L1, which in turn slows down L0-to-L1 compactions and accumulates SSTs in L0. Once the number of L0 SSTs reaches an empirical value (i.e., 60% of the write stall threshold[3]), FlexEngine transitions to the *second* stage: L0-to-L1 compactions are also suspended and only intra-L0 compactions are

[3]When L0 has 20 and 32 SSTs by default, RocksDB partially stalls and completely stalls new writes [5]. In this paper, write stall refers to the former, as even partial throttling is unacceptable for users.

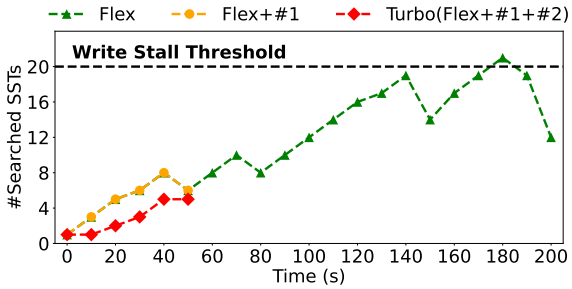


Figure 2: Runtime number of searched SSTs after storage bandwidth shortage occurs. #1/#2: Insight #1/#2. For `Flex` and `Flex+#1`, we count the number of L0 SSTs plus one from the ordered L1. For `Turbo`, we count the number of L0 SSTs in sub-LSM plus the number of other levels in sub-LSM. Section 6.4 describes detailed experimental setups (BOSR=6).

performed. Here, each intra-L0 compaction merges a batch of L0 SSTs into one larger SST still residing in L0.

FlexEngine is unable to meet the expected capability for the following two reasons. First, FlexEngine focuses exclusively on preventing write stalls, which occur in RocksDB when there are as many as 20 SSTs in L0. However, even a few more SSTs than under normal conditions cause a notable scan performance drop (see Section 6.4). Second, FlexEngine does not offer a principled WA minimization. The transition between its two stages is simply governed by a fixed empirical threshold.

3.3 Key Insights

In this paper, we present two key insights that jointly drive the cost-effectiveness of LSM-tree-based multi-tenant serverless cloud databases to a significantly higher level. Figure 2 provides an illustration of how these two insights contribute: compared to the state-of-the-art work FlexEngine (i.e., `Flex`), the first insight reduces the tenant migration time (i.e., `Flex+#1`) and the second further suppresses the number of searched SSTs (i.e., `Flex+#1+#2`).

Insight #1: Exploiting the Heterogeneity of Cloud Block Storage to Reduce Shortage Duration. The choice of cloud block storage has a non-trivial impact on the performance and cost of LSM-tree-based storage engines, yet previous research has paid little attention to this.

As introduced in Section 2.4, cloud block storage is typically divided into two categories: *cloud local SSD (LD)* and *cloud elastic SSD (ED)*. LD supports near-physical low latency, but forces cloud database providers to adopt a lengthy network-based tenant migration process. By contrast, while ED bears higher latency, it allows us to leverage the EBS-level snapshot functionality to quickly migrate tenants (e.g., ~50 seconds vs. ~200 seconds in our environments to migrate a 10GB partition). As the duration of storage bandwidth shortages is bounded by the tenant migration time (see Section 3.1), ED has less extra data than LD to reorganize, given the same user write speed. This means that the bandwidth required for reorganization while sustaining performance SLAs is lower, thereby supporting higher storage bandwidth over-subscription.

Given this, we aim to embrace the best of both worlds of LD and ED, motivated by two key observations. First, our performance analysis in Section 4 reveals that, compared to LD, deploying RocksDB on ED results in a significant drop in read performance, but write performance remains similar. Second, in the multi-replica model, secondary partitions are inherently write-only (i.e., without user read traffic). These two observations indicate that placing secondary partitions on ED results in negligible performance loss, yet would offer the key advantage of accelerating tenant migration through snapshots, thereby enabling high bandwidth over-subscription and improving deployment cost-effectiveness.

Table 2: Comparison of FlexEngine and TurboEngine.

Dimension	FlexEngine	TurboEngine
Storage layout	LD only	LD + ED
Migration	Network-based	Snapshot-based
Admission control	Partition & node	Partition & device
Compaction deferral	Ad-hoc two-stage	Principled sub-LSM

Insight #2: Introducing "Sub-LSM" Structure for Principled Searched SST Bounds and WA Minimization. To bound the maximum number of searched SSTs with low WA requirements, we propose reusing the concept of LSM tree and introducing *sub-LSM*, which temporarily reorganizes L0 and L1 as a small, self-contained LSM tree during storage bandwidth shortages.

Just like an LSM tree, the sub-LSM inherently offers principled guarantees on both searched SST bounds and WA minimization. On the one hand, the number of levels naturally limits the number of SSTs in a search; since every level is internally ordered, only one SST needs to be searched within each level. By setting the number of levels to match the normal condition, we can ensure that scan performance does not degrade from normal. On the other hand, since the maximum extra data volume during storage bandwidth shortages is known a priori in serverless databases (i.e., determined by the enforced user write speed and the duration of tenant migration, see Section 2.2), the optimal size ratio between levels that minimizes WA can be analytically solved [29].

These two mutually-reinforcing insights form the foundation of *TurboEngine*, our proposed LSM-tree-based key-value storage engine for multi-tenant serverless cloud databases. Table 2 summarizes the key differences between FlexEngine and TurboEngine. But before proceeding, it is essential to understand the performance implications of storing tenants on different cloud block storage.

4 PERFORMANCE CHARACTERIZATION OF ROCKSDB ON CLOUD LOCAL SSDS AND CLOUD ELASTIC SSDS

In this section, we characterize the performance of RocksDB [12] on both cloud local SSD (LD) and cloud elastic SSD (ED), focusing on whether storing tenants on EDs can negatively impact their performance and, if so, to what extent. Here, LD is a 2TB SSD bundled with ECS (see Section 6.1) and ED is a 1TB Alibaba Cloud PL3 ESSD (detailed parameters in [7]).

We evaluate RocksDB (one instance) with diverse workloads, including (1) write, (2) point read, (3) scan, (4) mixed point read and write (1:1), and (5) mixed scan and write (1:1). For point reads and scans, we consider both Zipfian and uniform access distributions. We preload 10M 1KB key-value pairs into the instance. For workloads involving scans (i.e., `Scan` and `Mixsw`), the DB instance receives 0.3K RCUs per second, and for other workloads, it receives

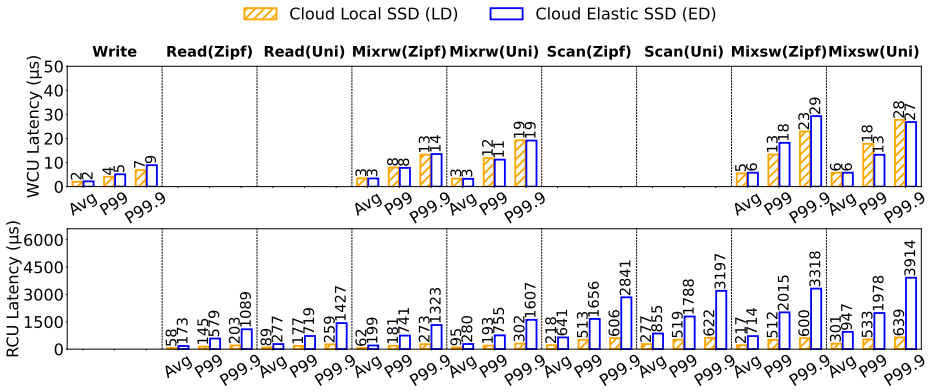


Figure 3: RocksDB's performance on LD and ED under diverse micro-level workloads. Mixrw: mixed point read and write (1:1). Mixsw: mixed scan and write (1:1).

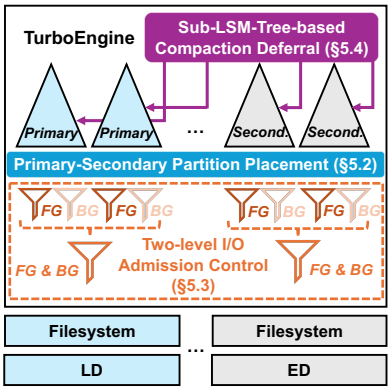


Figure 4: Overview of TurboEngine. FG/BG: foreground/background.

3K RCUs or WCUs per second (each CU is for a 1KB key-value pair). We configure a 2GB block cache and use direct reads to bypass the effect of kernel page cache. We evaluate the default WAL mode where writes are flushed to page cache but not immediately synced to storage; this strikes a practical balance between performance and process-level crash consistency.

Figure 3 shows the latency results on average, at 99th percentile (P99), and at 99.9th percentile (P99.9), measured when the workload runs for 200 seconds. For workloads involving two types of operations, we report their latency separately. Overall, we can observe significant performance divergence between LD and ED across workloads. For workloads involving reads (whether point reads or scans), the latency on ED is consistently and significantly higher. Specifically, the average, P99, and P99.9 latency of ED is 3.0–3.3×, 3.2–4.1×, 4.7–6.1× those of LD, respectively. However, for the write workload, the latency of LD and ED is similar.

The reasons behind this discrepancy are explained as follows. Reads in an LSM tree are processed synchronously, mostly small in size, meaning that the inherent overhead of EBS (e.g., network and software processing [102]) is fully exposed to host software, resulting in significantly higher latency on ED. The factor by which ED degrades the operation latency is essentially consistent with that observed in the device-level I/O latency (measured using `iostat`). In contrast, writes in an LSM tree are buffered in memory first (e.g., 1MB by default) and flushed to storage asynchronously; the WAL likewise defers persistence to page cache without synchronous I/Os. Consequently, although write I/O latency on ED is significantly higher than on LD (e.g., 2.7× higher on average and 1.8× higher at P99 in the write workload), this performance gap is masked by the fully asynchronous LSM write path[4].

5 DESIGNS OF TURBOENGINE

5.1 Overview

We present *TurboEngine*, a cost-effective LSM-tree-based key-value storage engine for multi-tenant serverless cloud databases. TurboEngine is designed to enhance multi-tenant serverless cloud databases that use LSM trees as their storage engines, significantly improving their cost-effectiveness. Figure 4 illustrates the overview of TurboEngine, which incorporates three key designs:

- (1) *Primary-secondary partition placement* (Section 5.2). TurboEngine co-deploys cloud local SSDs (LDs) and cloud elastic SSDs (EDs) within a node, storing primary partitions on LDs and secondary partitions on EDs, respectively.
- (2) *Two-level I/O admission control* (Section 5.3). A hierarchical rate-limiting framework, adapted based on [80], manages available bandwidth at both partition and device levels.
- (3) *Sub-LSM-tree-based compaction deferral* (Section 5.4). When storage bandwidth becomes insufficient, TurboEngine temporarily reorganizes L0 and L1 into a small, self-contained LSM tree, called sub-LSM.

5.2 Primary-Secondary Partition Placement

In database systems, the *multi-replica mechanism* is a key technique for improving data reliability and service availability. It achieves data redundancy by storing multiple copies of the same data across different nodes, thereby ensuring that even if a node fails, data will not be lost and the system will continue to provide services to users. For instance, in the three-replica model, each tenant partition consists of one *primary* partition and two *secondary* partitions, each residing on a different node. To maintain consistency among replicas, a data replication protocol is required to synchronize user-level write traffic. But unlike write traffic, only the primary partition handles read traffic to ensure strong read consistency.

With the multi-replica model, TurboEngine proposes to *store primary partitions on LDs and secondary partitions on EDs, respectively.* This allows TurboEngine to offer low read latency while enabling rapid tenant migration through snapshots, thereby enhancing cost-effectiveness. Note that tenant migration does not occur on the node that is already bandwidth-constrained, but rather on that hosting a *secondary* replica. With snapshot-based migration, there are generally four main steps: (1) generating a snapshot (within seconds, available for use before fully persisting in object storage service), (2) creating a new ED based on the snapshot (within seconds), (3) mounting the new ED on the new node and finalize full synchronization (within seconds), and (4) do incremental synchronization. Compared to LD's network-based migration, the full synchronization time (i.e., steps 1 to 3) is significantly shortened.

[4]Only valid with the default WAL mode or not using WAL [4]; if WAL is frequently fsync'ed, there is still a difference in write performance between LD and ED.

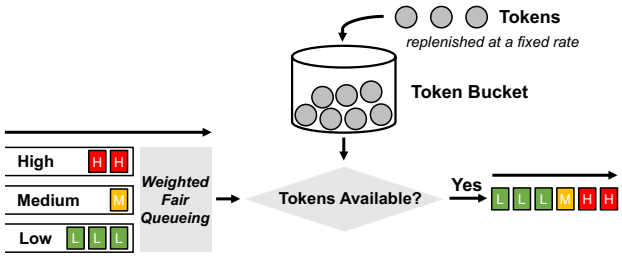


Figure 5: Overview of token bucket algorithm with multi-priority queues (e.g., high, medium, and low).

Migrating partitions using ED snapshots changes the migration granularity *from partition to device*. However, TurboEngine only rebuilds partitions that have been chosen for migration. To guarantee highly stable migration times, TurboEngine further introduces an optimization technique, namely *multi-ED deployment*, which evenly distributes secondary partitions in a node across multiple EDs. For example, our evaluated ECS (see Section 6.1) supports up to 16 EDs simultaneously.

Finally, TurboEngine does *not* store primary partitions on EDs, even when they are write-intensive, for two reasons. First, from the storage device perspective, ED still incurs a substantial read performance overhead compared to LD (see Section 4), which inevitably causes service quality variances for user RCUs. Second, from the DBMS perspective, even if the read and write traffic of many tenants exhibits cyclical behaviors (e.g., daily, weekly, or day-night cycles), which makes it possible to periodically migrate write-only primary partitions to ED, the overhead associated with fine-grained monitoring and frequent tenant migration during traffic shifts would largely offset the further cost-effectiveness gain.

5.3 Two-Level I/O Admission Control

In this section, we first introduce the basics of the token bucket algorithm and its integration with multi-priority queues (Section 5.3.1). We then describe the designs at the levels of partition and device (Sections 5.3.2 and 5.3.3).

5.3.1 Basics of Token Bucket Algorithm with Multi-Priority Queues. The *token bucket algorithm* is a classic method for rate limiting and traffic shaping to ensure both system responsiveness and stability [2]. As shown in Figure 5, this algorithm works by maintaining a token bucket, where each token grants permission to process a work unit (e.g., a byte of data). Tokens in the token bucket are replenished at a fixed rate, and incoming tasks must consume the corresponding number of tokens before they can continue execution. In practice, the token bucket algorithm is often combined with *multi-priority queues*, enabling differentiated token allocation based on task importance, such as through the *weighted fair queuing (WFQ)* algorithm [3].

5.3.2 Partition-level I/O Admission Control. The goal of partition-level I/O admission control is to stabilize system-side bandwidth usage while ensuring tenant performance SLAs. The key challenge here is that the LSM tree processes user reads and WAL writes synchronously (see Section 2.3). However, with RocksDB's existing rate-limiting mechanism (i.e., auto-tuned [1]), if prolonged low-bandwidth usage has reduced the partition's maximum available bandwidth to a low level, a sudden surge in user reads and WAL writes can be unexpectedly blocked.

Therefore, inspired by FlexEngine [80], TurboEngine configures two rate limiters for each partition, managing the maximum available bandwidth for *foreground* and *background* I/Os separately. Both rate limiters are based on the token bucket algorithm but employ different bandwidth management principles.

Foreground I/Os (e.g., user reads and WAL writes) have *fixed-reserved* maximum bandwidth, derived from two components. The one for user reads is obtained by the maximum number of RCUs per second multiplied by the maximum supported RCU size and the read amplification factor. The other for WAL writes is calculated by the maximum number of WCUs per second multiplied by the maximum supported WCU size. The fixed-reserved bandwidth ensures that latency-critical I/Os are never unexpectedly throttled. In contrast, background I/Os (e.g., flushes and compactions) have *dynamically-adjusted* maximum bandwidth, calculated from the average per second loaded data volume from WCUs multiplied by the LSM tree's background I/O amplification. Thus, if the user write traffic remains low, maximum background bandwidth is reduced accordingly, allowing more partitions to be co-located within a node to improve cost-effectiveness. Same as RocksDB [6], the weighted ratio between flushes and compactions is set to 10:1.

5.3.3 Device-level I/O Admission Control. The goal of device-level I/O admission control is to ensure that tenants consistently experience low latency, even during storage bandwidth shortages. The key challenge here is that, when storage bandwidth becomes insufficient, the storage side cannot process I/Os in a timely manner. Thus, unconditionally issuing I/Os without a holistic awareness of I/O priorities may cause latency spikes for critical user reads.

Thus, TurboEngine follows the same principles as Calcspar [105] and FlexEngine [80], but adapts its scope of operation from the node level to the *device* level. TurboEngine integrates a rate limiter for each storage device (i.e., LD or ED) to regulate I/Os across all partitions on that device. This device-level rate limiter only unconditionally issues foreground I/Os to ensure they are not accidentally blocked. For the background flushes and compactions, their weighted ratio is also 10:1.

5.4 Sub-LSM-Tree-based Compaction Deferral

Although the two-level I/O admission control framework manages available bandwidth at both partition and device levels, it alone cannot sustain performance SLAs when storage bandwidth is insufficient. To this end, TurboEngine introduces a *sub-LSM-tree-based compaction deferral* mechanism, in which *sub-LSM* is a temporary top-level reorganization layout exclusively activated during storage bandwidth shortages, rather than a steady-state merge policy.

As illustrated in Figure 6, when a storage bandwidth violation is detected (i.e., the system-wide bandwidth usage exceeds the high-watermark threshold), TurboEngine performs three coordinated actions in bandwidth-intensive partitions: (1) Suspend compactions from L1-to-L2 onward to conserve scarce bandwidth, as short-term delays of these compactions have a negligible impact on read performance [16, 80, 105]; (2) Clear the top levels (i.e., L0 and L1) in two steps: compacting L0 and L1 into L1, and then compacting L1 and L2 into L2; the first step quickly eliminates L0 SSTs to prevent

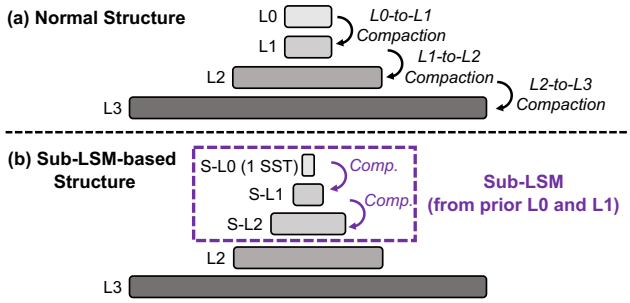


Figure 6: Overview of sub-LSM-tree-based compaction deferral mechanism.

latency spikes; (3) Reorganize L0 and L1 into a small, self-contained LSM tree, namely *sub-LSM*, which now takes over all subsequent flushed data and performs compactions according to its own internal structure.

The key to this mechanism is how to determine the structure of sub-LSM. First, regarding the level count L, sub-LSM places one SST in its L0, so L is normally equal to the maximum number of SSTs in the search path. This is because, just like an LSM tree, each level within sub-LSM is internally sorted; thus, during a search, at most one SST needs to be checked at each level. Therefore, setting L to be no greater than the normal-state searched SST count guarantees that scan performance during shortages is no worse than under normal conditions. For example, in RocksDB with default settings, L0-to-L1 compaction triggers when L0 contains 4 SSTs. So on average, L0 holds about 2 SSTs, and L1 contributes one SST to a search, yielding approximately 3 searched SSTs normally. Accordingly, L is set to 3 in this configuration.

With a fixed L, the other important parameter is the size ratio T between adjacent levels. As shown in Equation 1, a prior work, Spooky [29], has derived the formula for calculating compaction WA in the partial merge model:

$$\text{Compaction WA} = \frac{T+1}{2} \cdot L \tag{1}$$

Since compaction WA increases monotonically with T, minimizing WA is equivalent to obtaining the smallest T that allows sub-LSM to accommodate all extra data during storage bandwidth shortages. The critical background is that, in serverless databases, the maximum extra data volume V is known a priori: it is bounded by the maximum user write speed multiplied by the tenant migration duration (see Section 2.2). For example, assume the maximum number of WCUs per second is 3K, the maximum size of WCU is 1KB, and snapshot-based tenant migration lasts for 50 seconds, the maximum amount of extra data is 3MB/s*50s=150MB. In practice, since tenant migration is typically completed before the worst-case scenario described above occurs, TurboEngine conservatively sets 1/3 of V (i.e., $V/3$) as the target capacity and seeks the minimum value of T that can accommodate the data in L.

This mechanism has minimal impact on user read performance for three reasons. First, the maximum number of searched SSTs is principally bounded to be no greater than the normal-state count (as ensured by L), thereby safeguarding scan latency. Second, each SST is equipped with a bloom filter, so point reads are largely unaffected by the temporary restructuring. Third, the LD-ED co-deployment model significantly reduces tenant migration time, resulting in only a very small amount of temporary data in sub-LSM, which can be properly cached in system memory.

It is also worth noting that when system storage bandwidth becomes sufficient again (whether because user loads decrease or tenant migration is finished), TurboEngine resumes the suspended compactions and restores the LSM tree to its normal structure. The data accumulated in sub-LSM, as a whole, is merged back into L1 through next L0-to-L1 compactions, after which high-level compactions from L1-to-L2 proceed to rebalance the entire tree.

Table 3: Average latency in μs of loading 10M key-value pairs (setups in Section 6.4) under different replication policies. `Turbo(LD)`/`Turbo`: secondary partitions on LDs/EDs.

Deployment	Async. (default)	Semi-sync.	Sync.
`Turbo(LD)`	2.8	1451	1539
`Turbo`	2.8	1454	1537

5.5 Discussions

Architectural choices of multi-tenant LSM trees. There are typically two architectural approaches for building multi-tenant LSM trees: the instance-based approach and the column family-based approach. We choose the first approach and allocate a separate LSM tree to each tenant partition for the following reasons: (1) reliability, which is critical for cloud database products, by running a separate LSM tree for each partition, we can configure flexible and independent data backup strategies for users; (2) performance, in the second approach, all column families share and compete for the same WAL, leading to severe write performance bottlenecks.

Extending ED to general shared storage backend. The rationale behind our heterogeneous LD-ED co-deployment model is to combine the low-latency benefit of cloud local storage with the fast tenant migration capability of cloud elastic storage. As a result, our approach can also be extended if the remote storage is a general shared storage backend, such as a distributed file system.

Performance across different replication policies. TurboEngine, particularly its LD-ED co-deployment, delivers good performance across different replication policies, whether asynchronous (commonly used for high-performance scenarios and required by most of our customers), semi-synchronous, or synchronous. For reads, the replication policy has negligible impact, as user reads are always served by the LD-backed primary with low latency. For writes, the key difference is whether a user write waits for acknowledgments from secondary partitions. Under asynchronous replication, user writes depend only on the LD-backed primary. Under (semi-)synchronous replication, user writes additionally wait for secondary acknowledgments; in these cases, the dominant cost lies in network communication rather than the secondary's local path, so placing secondaries on EDs does not sacrifice performance compared to the best-performing LD-only deployment. Table 3 empirically confirms this, showing similar average load latency between `Turbo(LD)` and `Turbo` across all three policies.

Recovery performance under crash scenarios. The choice of cloud block storage has different implications on recovery performance depending on the crash scenario. If a secondary fails, or both primary and secondary fail simultaneously, LD and ED have comparable impacts on recovery performance. This is because the first

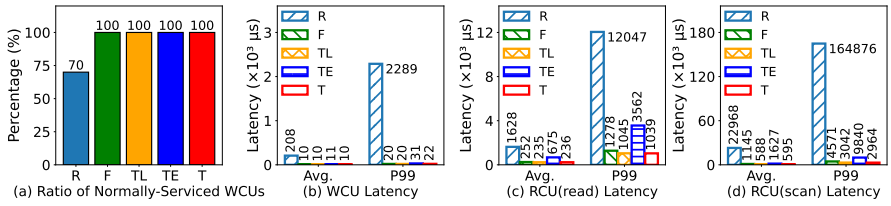


Figure 7: Production workload performance in terms of the ratio of normally-serviced WCUs and WCU/RCU latency. R: RocksDB (with LD). F: FlexEngine (with LD). TL: TurboEngine with LD. TE: TurboEngine with ED. T: TurboEngine with co-deployed LD and ED. In (b)-(d), bar labels are in μs (rather than 1000μs).

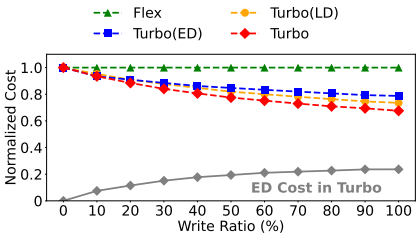


Figure 8: Normalized deployment cost under different write ratios. `Flex`: FlexEngine (with LD). `Turbo(LD/ED)`: TurboEngine with LD or ED.

case does not affect the primary's availability, and the second case directly triggers a database restart whose recovery procedure is almost identical regardless of the underlying storage type. However, if only the primary fails and a primary-to-secondary switchover is triggered, the new primary temporarily has degraded read performance (until an LD-backed instance is rebuilt and becomes the primary again). This problem shares the same root cause as the well-known post-failover cold-start problem in production, including cold caches, insufficiently warmed-up execution resources, and, in this case, the higher read latency of ED (see Section 4). To mitigate this degradation, we also employ strategies analogous to those for cold-start problems: provisioning additional resources for affected tenant partitions, including storage bandwidth, memory, CPU, and intra-/inter-command parallelism.

6 EVALUATION

6.1 Experimental Setups

TurboEngine is implemented based on a commercially-deployed RocksDB [12], and runs in an ECS virtual machine from Alibaba Cloud (type: ecs.i4g.8xlarge) with Intel Xeon (Ice Lake) Platinum 8369B processors, 32 vCPUs, and 128GB memory. The ECS comes with a 2TB LD, which supports up to 3GB/s read bandwidth and 2GB/s write bandwidth. This ECS also supports the optional addition of up to 16 EDs, with the maximum total bandwidth at 1GB/s.

6.2 Production Workload Performance

In this section, we first examine TurboEngine's performance in production deployments. We replay a collected three-day node-level trace on TurboEngine (i.e., `Turbo`), the vanilla RocksDB [12] (i.e., `RDB`), and the state-of-the-art FlexEngine [80] (i.e., `Flex`). To provide more insights, we also evaluate two comparison groups, named `Turbo(LD)` and `Turbo(ED)`, indicating that tenant partitions in TurboEngine are only stored in LDs and EDs, respectively.

To ensure a fair comparison, all groups under evaluation share the same ECS instance type and the same available storage bandwidth (i.e., capped at 1GB/s via device-level rate-limiting, aligned with the maximum ED bandwidth to ensure a fair bandwidth budget across all groups), and they differ in whether I/Os are served by LD (i.e., `RDB`, `Flex`, and `Turbo(LD)`), ED (i.e., `Turbo(ED)`), or both LD and ED (i.e., `Turbo`). In all evaluated groups, we configure both `write_buffer_size` and `target_file_size_base` (i.e., targeted SST size) to 16MB, keeping the multi-tenant system memory acceptable in practice. Thus, the maximum sizes of L0 and L1 are both 64MB. The size ratio between levels over L0 is 10. We also set `max_subcompactions` to 4 to accelerate L0-to-L1 compactions. These settings are also applied to experiments hereafter.

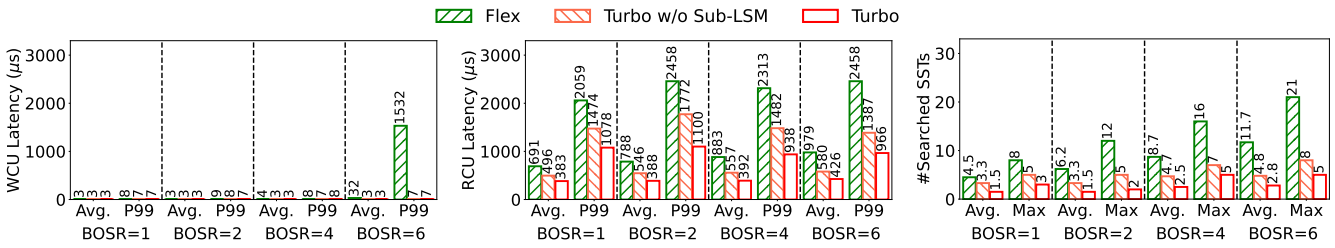
Figure 7a shows the ratio of normally-serviced WCUs (i.e., WCUs not experiencing write stalls), while (b)–(d) show the latency of both WCUs and RCUs. To clearly assess scan performance, we report RCU latency for point reads and scans in (c) and (d) separately. Overall, we can see that `Turbo` serves all WCUs normally without stalls, achieves low WCU latency, and delivers notably better RCU latency. In particular, compared to the state-of-the-art `Flex`, `Turbo` significantly improves scan performance thanks to the combined effect of LD-ED co-deployment and sub-LSM-tree-based compaction deferral. Specifically, for scan-based RCUs, the average latency and P99 latency are improved by 48.0% and 35.2%, respectively. `Turbo(ED)` has notably worse RCU latency across both access types: for point-read-based RCUs, the average and P99 latencies are elevated to 2.9× and 3.4×, respectively, and for scan-based RCUs, the values are increased by 2.7× and 3.3×. The main reason is that primary partitions are stored in ED, directly exposing ED's high read latency to users. `Turbo(LD)` generally performs similarly to `Turbo`, but its deployment cost is much higher due to its lower achievable storage bandwidth over-subscription, as analyzed in the next section.

6.3 Deployment Cost Optimization

In this section, we investigate how TurboEngine optimizes our deployment cost. For this, we develop a simulator that can estimate cost savings based on specific node environments, tenant loads, and deployment setups such as the storage bandwidth over-subscription ratio. Here, the cost refers to the sum of two main contributors: (1) ECS-associated fees and (2) ED fees (if equipped). Note that the price of LD is included in the ECS fee. As in the previous section, we also evaluate TurboEngine that equips only LD or ED (i.e., `Turbo(LD)` and `Turbo(ED)`).

Figure 8 shows the results under varying write ratios, normalized based on the cost of `Flex`. We can see that, compared to `Flex`, `Turbo` achieves substantial cost savings, especially under high write ratios. When the write ratio is 50% (i.e., the read-to-write ratio is 1:1), the cost is reduced by 22.5%, and when the entire node consists of write-only tenants, cost savings reach as high as 32.4%.

The main reason why cost optimization becomes more effective as the write ratio increases is that the bandwidth reserved for read traffic cannot be over-subscribed (see Section 5.3.2). This is because LSM tree reads are synchronous, and sudden spikes in



(a) WCU latency (b) RCU (scan) latency (c) Searched SST count

Figure 9: Performance under a mixed workload consisting of scans and sustained high-intensity writes. `Flex`: FlexEngine. `Turbo w/o Sub-LSM`: TurboEngine without sub-LSM-tree-based compaction deferral. `Turbo`: TurboEngine. There are 16 partitions co-located in a node. The background over-subscription ratio (BOSR) is defined as the ratio of the total required background bandwidth of all partitions to the actual device-level available bandwidth. Here, for `Flex` and `Turbo w/o Sub-LSM`, searched SST count refers to the number of L0 SSTs plus one from the ordered L1; for `Turbo`, it refers to the number of L0 SSTs in the sub-LSM plus the number of other non-empty levels in the sub-LSM. For both latency and searched SST count, the lower the better.

read traffic would cause user-perceived latency to skyrocket if read bandwidth were already over-committed. Consequently, a higher write ratio means that a smaller fraction of the total bandwidth is locked for read reservations, leaving a larger share available for over-subscription of write-related background I/Os, which in turn allows more partitions to be co-located within the same node. Another interesting finding is that the higher the write ratio, the higher the proportion of ED costs of the total; with 100% write ratio, about 35% of the costs are from EDs.

When comparing the three setups for TurboEngine, we can see that co-deploying LD and ED is important for further optimizing the cost. Particularly, when the write ratio is 50%, the cost of `Turbo` is 4.6% and 9.2% lower than that of `Turbo(LD)` and `Turbo(ED)`, respectively. Compared to the second-best `Turbo(LD)`, the cost advantage of `Turbo` stems from placing secondary partitions on EDs, which unlocks snapshot-based fast migration. Faster tenant migrations shorten the duration of storage bandwidth shortages, reducing the amount of data that needs to be reorganized during shortages, and thereby enabling a higher over-subscription ratio within the same bandwidth budget.

6.4 Performance Breakdown Analysis

In this section, we conduct a detailed breakdown analysis to explore the performance impact of TurboEngine's two key designs: LD-ED co-deployment and sub-LSM-tree-based compaction deferral. We deploy 16 primary partitions on LD and preload each with 10M key-value pairs (16B key and 1KB value). Next, we apply a mixed workload consisting of 0.03K scans and 3K inserts per second, continuing until tenant migration is finished. The scans follow a Zipfian distribution with an average scan length of 3.

As introduced in Section 5.3.2, foreground I/Os (including scans) have fixed-reserved bandwidth for timely processing, while background I/Os (e.g., flushes and compactions) have dynamically-tuned bandwidth based on user write speeds to improve overall cost-effectiveness. Therefore, we adjust the maximum available background bandwidth for all partitions through device-level rate limiting, so as to reflect varying degrees of storage bandwidth over-subscription. Meanwhile, the partition-level maximum available background bandwidth remains unchanged.

Specifically, each partition is allocated a maximum available background bandwidth of 90MB/s, which is sufficient to ensure that the LSM tree structure remains stable given the current data volume and user write speed. This value is obtained by multiplying the user write speed (i.e., 3K*1KB=3MB/s) and the provisioned background I/O amplification of the LSM tree (i.e., 30). At the device level, the maximum available background bandwidth is up to the total required background bandwidth of partitions above (i.e., 16*90MB/s= 1440MB/s). For simplicity, we define the ratio of the total required background bandwidth of partitions to the actual available bandwidth as *background over-subscription ratio (BOSR)*. We vary BOSR from 1 (indicating that storage bandwidth is not over-subscribed) to 6, so the device-level maximum available background bandwidth ranges from 1440MB/s (BOSR=1) to 240MB/s (BOSR=6).

We compare TurboEngine (i.e., `Turbo`) with FlexEngine (i.e., `Flex`) and an ablation variant `Turbo w/o Sub-LSM`, which replaces sub-LSM-tree-based compaction deferral with FlexEngine's two-stage compaction deferral while retaining LD-ED co-deployment. `Flex` relies on network-based tenant migration but both `Turbo` variants can use snapshot-based tenant migration. Given the data volume in this section (roughly 10GB per partition), we set the tenant migration time for the network-based method to 200 seconds and for the snapshot-based method to 50 seconds.

Figure 9 demonstrates the results, with (a), (b), and (c) depicting WCU latency, RCU latency, and searched SST count, respectively.

Scan latency (Figure 9b). We can first see that `Turbo` achieves the lowest scan latency across all BOSRs and remains remarkably stable: the average latency ranges from 383μs (BOSR=1) to 426μs (BOSR=6). This highlights that, compared to the scenario without storage bandwidth over-subscription, the latency fluctuation with 6× over-subscription is only 11.2%, demonstrating `Turbo`'s ability to maintain scan performance during storage bandwidth shortages. On the contrary, the performance of `Flex` degrades significantly as BOSR increases: the average latency rises from 691μs at BOSR=1 to 979μs at BOSR=6, with a deterioration as high as 41.7%.

The improvement of `Turbo` comes from two complementary sources, which are exactly TurboEngine's two key designs. First, LD-ED co-deployment alone (comparing `Turbo w/o Sub-LSM` with `Flex`) reduces the average scan latency by 28%–41% across all BOSRs. This is because the much shorter tenant migration reduces

Table 4: Inter-tenant performance of TurboEngine at BOSR=6. The "Group" column format is N (*insert ops/s, scan ops/s*), where N is the number of tenants in the group. CV denotes the coefficient of variation (std/mean) of the metric across all tenants within the group.

Workload	Group	BG Thpt. MB/s	Op	Avg Lat. μs: Min–Max (CV%)	P99 Lat.
Mixed (uniform)	16 (3K,30)	11.0–12.0 (2.0)	Insert	2.7–2.9 (1.9)	7.1–8.1 (3.6)
			Scan	414–460 (2.6)	872–1069 (6.3)
Mixed (skew scan)	4 Hot (3K,100)	11.5–12.2 (2.7)	Insert	3.0–3.2 (2.3)	7.6–8.4 (4.7)
			Scan	381–391 (1.2)	900–1009 (5.3)
	12 Cold (3K,10)	11.2–12.2 (2.8)	Insert	2.8–3.2 (3.6)	7.1–8.3 (4.7)
			Scan	432–482 (3.3)	870–1165 (9.4)
Skew Insert	4 Hot (10K,–)	41.0–43.0 (2.4)	Insert	2.1–2.3 (3.3)	4.7–5.3 (5.5)
	12 Cold (1K,–)	3.0–3.0 (0.0)	Insert	3.2–3.8 (5.0)	8.2–9.7 (4.9)

the duration of storage bandwidth shortages as well as the amount of L0 SST accumulation, thereby mitigating scan performance degradation. Second, sub-LSM further reduces the average scan latency by 23%–33% (comparing `Turbo` with `Turbo w/o Sub-LSM`) by effectively limiting the number of searched SSTs within its internal hierarchical structure, thereby further securing scan performance.

As for the P99 latency, the improvements of `Turbo` are even more pronounced. Specifically, compared to `Flex`, `Turbo` achieves 48%–61% lower P99 latency across all BOSRs. When BOSR is 6, the P99 scan latency drops from 2458μs to 966μs, less than 1ms.

Searched SST count (Figure 9c). The aforementioned trend in scan latency is primarily attributed to the searched SST count, i.e., the number of SSTs in the search path. Here, for `Flex` and `Turbo w/o Sub-LSM`, searched SST count refers to the number of L0 SSTs plus one from the ordered L1; for `Turbo`, it refers to the number of L0 SSTs in the sub-LSM plus the number of other non-empty levels in the sub-LSM. We exclude discussing L2 and deeper levels because they are the same across the three configurations.

For `Flex`, searched SST count grows rapidly as BOSR rises. When BOSR is 6, the average reaches 11.7, and the maximum reaches 21. This far exceeds the scan degradation point (i.e., 3 under normal conditions, see Section 5.4) and even approaches the write stall threshold (i.e., 20). As previously shown in Figure 2, the runtime L0 SST accumulation pattern closely matches this trend. Since a scan must examine all SSTs with overlapping keys, this cumulative effect directly exacerbates scan latency. `Turbo w/o Sub-LSM` reduces the searched SST count by shortening the shortage duration. For example, when BOSR is 6, the average is 4.8 and the maximum is 8. However, as BOSR increases, the count continues to rise.

In contrast, `Turbo` well bounds the searched SST count: even at BOSR=6, the average is only 2.8 and the maximum is 5. This is precisely the effect of the sub-LSM structure: by reorganizing flushed data into a small leveled LSM tree, each level is internally sorted. At most one SST per level needs to be examined during a scan, providing a principled bound on searched SST count.

WCU latency (Figure 9a). At lower BOSR values, all three configurations exhibit similarly low write latency, averaging 3–4μs. This confirms that neither LD-ED co-deployment nor sub-LSM introduces additional overhead in the write path. However, when BOSR is 6, `Flex` experiences severe write stalls: its average WCU latency skyrockets to 32μs, and the P99 value reaches as high as 1532μs. In contrast, both `Turbo` and `Turbo w/o Sub-LSM` maintain

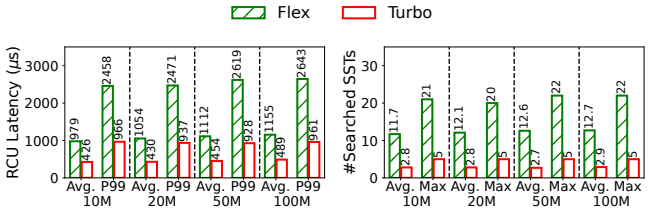


Figure 10: Performance scalability under varying per-partition data volumes. The experimental setup and metric definitions match those of Figure 9, with BOSR of 6.

stable write latency, thanks to the significantly shortened tenant migration time.

The above results and analysis demonstrate that both designs contribute complementary benefits to sustaining tenant performance under high storage bandwidth over-subscription. Even at a BOSR as high as 6, `Turbo` maintains stable performance SLAs, with average scan latency fluctuating by only 11.2% and P99 scan latency remaining below 1ms.

6.5 Performance Stability and Fairness Analysis

As a multi-tenant system, performance stability and fairness across tenants are of particular importance, for which we provide further insights in this section. We focus this analysis on the highest-contention setting in our evaluation: the 16-tenant workload used in Figure 9 under BOSR=6. We also evaluate two tenant-skewed settings, one for inserts and the other for scans, with their parameters detailed in Table 4. As the user read/write traffic of new workloads varies, the foreground/background bandwidth is also adjusted proportionally according to the setups in Section 6.4.

Stability. We use the coefficient of variation (CV) of inter-tenant latency to measure stability. Overall, we can see the latency CV is consistently low under all three workloads, confirming that TurboEngine achieves good performance stability across tenants. The average latency CV ranges from 1.2% to 5.0%. The P99 latency CV is slightly higher, ranging from 3.6% to 9.4%, but remains below 10%.

Fairness. We use the CV of inter-tenant background throughput to measure fairness. As we can see, the throughput CV is only up to 2.8% under all three workloads, confirming that tenants within the same group (defined as those with the same workload setups, see the caption of Table 4) under all three workloads can compete fairly for bandwidth. Moreover, in the skewed insert workload, hot tenants do not starve cold tenants, only using bandwidth proportional to their user write traffic (10:1). We exclude foreground bandwidth because it is fixed-reserved (see Section 5.3).

6.6 Performance Scalability Analysis

Finally, we examine whether `Turbo` retains its scan-performance advantage as both the per-partition data volume and the depth of the main LSM tree increase. We follow the workload settings in Figure 9 with BOSR of 6, and vary the preload across 10M (L0–L4, 5 levels), 20M (L0–L4, 5 levels), 50M (L0–L4, 5 levels), and 100M (L0–L5, 6 levels) key-value pairs per partition.

As shown in Figure 10, `Turbo` keeps its average and P99 scan latencies within 426–489μs and 928–966μs across this tenfold increase in data volume. The corresponding CVs across the four data volumes are only 6.4% and 1.9%, respectively. In contrast, `Flex` exhibits

average and P99 scan latencies of 979–1155μs and 2458–2643μs, respectively. The searched SST count (defined in the caption of Figure 9) supports this consistent advantage. For `Flex`, the average searched SST count ranges from 11.7 to 12.7, and its maximum reaches 20–22; `Turbo` averages only 2.7–2.9 and never exceeds 5.

7 RELATED WORK

Multi-tenant serverless cloud databases. A growing body of work addresses the architecture and resource management of multi-tenant serverless cloud databases. For example, PolarDB Serverless [19] decouples CPU, memory, and storage into independent pools over RDMA, using cache invalidation, optimistic locking, and index-aware prefetching. This enables elastic, fine-grained resource scaling, eliminates fate-sharing, and achieves fast failure recovery while maintaining comparable performance. Azure SQL Database Serverless [61] proactively predicts pause/resume patterns to reduce cold-start delays and uses logical pauses to avoid short ineffective pauses, balancing high quality of service with operational cost. Aurora Serverless [17] achieves predictable elasticity and high host utilization via live migration-based instance repacking, token bucket rate limiting, and distributed reactive control. It balances seamless scaling with cost efficiency by deliberately unbalancing host loads and co-designing hypervisor, OS, and database engine. Abase [41] is a multi-tenant NoSQL serverless database that tackles workload diversity and dynamism in large-scale clouds via cache-aware isolation, predictive autoscaling, dual-layer caching, and multi-resource rescheduling. CockroachDB Serverless [72] introduces cluster virtualization by separating SQL and KV layers into distinct processes. This enables multi-tenant isolation, sub-second scaling from zero, responsive autoscaling, dynamic session migration, and active-active multi-region deployments, solving cost and elasticity challenges for transactional databases.

These efforts collectively advance multi-tenant serverless databases along dimensions such as resource disaggregation, elastic scaling, cold-start mitigation, and multi-tenant isolation. However, they have not addressed the cost-effectiveness dilemma faced by the LSM-tree-based multi-tenant serverless cloud databases, which is the focus of this paper.

LSM-tree-based key-value store. The LSM tree has a vast design space and has been extensively studied in the past [54, 64, 65]. Representative directions include: (1) compaction policies [20, 24–27, 29, 39, 58, 62, 66, 67, 94], (2) storage data layout such as key-value separation [23, 33, 38, 47, 51, 81, 89, 99], (3) workload-aware and cost-model-based tuning [27, 40, 50, 52, 53, 57, 68, 79, 97], (4) filtering and caching [21, 24, 28, 35, 36, 44, 48, 55, 74, 76–78, 86, 87, 92, 98, 104], and (5) emerging hardware, distributed, and cloud adaptations [18, 30–32, 37, 38, 42, 49, 63, 70, 71, 80, 96, 100, 101, 105].

While most of the above works target single-store or single-tenant settings, FlexEngine [80] is the only existing work designed specifically for multi-tenant serverless cloud databases. Our TurboEngine advances beyond it by providing a bounded searched SST count with low WA, thereby achieving consistent tenant performance SLAs even with high storage bandwidth over-subscription.

Looking inside the system, sub-LSM bears some technical overlap with hybrid leveled-tiered policies, such as lazy leveling [26]. However, sub-LSM differs from lazy leveling in several key aspects. First, regarding *scope*: sub-LSM is a transient top-level reorganization confined to L0 and L1, activated only during storage bandwidth shortages, whereas lazy leveling is a steady-state merge policy across non-largest levels and the entire LSM lifecycle. Second, regarding *optimization formulation*: sub-LSM minimizes WA subject to a strict search-path bound and a known data volume ceiling from serverless, whereas lazy leveling targets a general write-read-space tradeoff without capacity constraints. Third, regarding *intra-level structure*: sub-LSM still employs leveling (one sorted run per level with exponentially increasing capacities), whereas lazy leveling employs tiering in all non-largest levels (multiple sorted runs of comparable sizes per level).

Cloud block storage. Several works have documented the advancements in cloud local storage [90, 93] and elastic block storage (EBS) [22, 56, 69, 73, 85, 88, 102]. From a performance characterization perspective, Wang et al. [82, 83] characterize the performance of Amazon AWS and Alibaba Cloud EDs, revealing counter-intuitive performance traits versus LDs. However, these studies focus on evaluation at the raw device level and do not reveal our observed read-write asymmetry from the perspective of the LSM tree.

Heterogeneous cloud storage for key-value stores. A few studies explore heterogeneous cloud storage for key-value stores. For example, Mutant [95] dynamically places SSTs in an LSM tree across different cloud storage types based on access frequency, pursuing seamless cost-performance trade-offs without time-consuming data migration. ROCKSMASH [91] improves cost-efficiency and read performance of LSM trees when deploying both cloud local storage and EBS via compaction-aware caching and space-efficient metadata. MirrorKV [84] balances compaction and query performance of LSM trees on hybrid cloud storage, combining EBS and object storage, via virtual split, mirrored LSM-trees, and cache-guided hot data retention. Cosine [20] self-designs optimal key-value engines using analytical I/O and learned CPU models across cloud local storage, EBS, and object storage, solving cloud cost-performance trade-offs for diverse workloads.

These works treat heterogeneous cloud storage as a data placement problem within a single store, such as balancing cost and performance through tiering. In contrast, TurboEngine leverages this heterogeneity across multi-tenant stores: primary and secondary partitions are strategically deployed in cloud local storage and EBS, respectively, thereby achieving both low read latency and fast snapshot-based migration.

8 CONCLUSION

We present TurboEngine, a cost-effective LSM-tree-based storage engine for multi-tenant serverless cloud databases. We identify that the core capability demanded during storage bandwidth shortages is to reliably bound the number of SSTs in a search path while incurring low write amplification. To this end, TurboEngine introduces two mutually-reinforcing designs: LD-ED co-deployment and sub-LSM-tree-based compaction deferral. Experiments show that TurboEngine reduces the deployment cost by 22.5% with a 1:1 read-to-write traffic ratio, and sustains 6× storage bandwidth over-subscription without compromising tenant performance SLAs.

ACKNOWLEDGMENTS

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

REFERENCES

[1] 2017. Auto-tuned Rate Limiter in RocksDB. Retrieved November 22, 2026 from https://rocksdb.org/blog/2017/12/18/17-auto-tuned-rate-limiter.html

[2] 2017. Wikipedia of Token Bucket Algorithm. Retrieved November 22, 2026 from https://en.wikipedia.org/wiki/Token_bucket

[3] 2017. Wikipedia of Weighted Fair Queueing. Retrieved November 22, 2026 from https://en.wikipedia.org/wiki/Weighted_fair_queueing

[4] 2020. WAL Performance in RocksDB. Retrieved November 22, 2026 from https://github.com/meta/rocksdb/wiki/WAL-Performance

[5] 2021. Write Stalls in RocksDB. Retrieved November 22, 2026 from https://github.com/meta/rocksdb/wiki/Write-Stalls

[6] 2022. Rate Limiter in RocksDB. Retrieved November 22, 2026 from https://github.com/meta/rocksdb/wiki/rate-limiter

[7] 2025. Alibaba Cloud ESSD. Retrieved November 22, 2026 from https://www.alibabacloud.com/help/en/ecs/user-guide/essds

[8] 2026. Alibaba Tair Serverless KV. Retrieved November 22, 2026 from https://www.alibabacloud.com/help/en/redis/product-overview/tair-serverless-kv-redis-compatible-edition

[9] 2026. Apache Cassandra. Retrieved November 22, 2026 from https://cassandra.apache.org/_/index.html

[10] 2026. Apache HBase. Retrieved November 22, 2026 from https://hbase.apache.org/

[11] 2026. Google LevelDB. Retrieved November 22, 2026 from https://github.com/google/leveldb

[12] 2026. Meta RocksDB. Retrieved November 22, 2026 from https://rocksdb.org/

[13] 2026. PingCAP TiKV. Retrieved November 22, 2026 from https://tikv.org/

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] Laurent Bindschaedler, Ashvin Goel, and Willy Zwaenepoel. 2020. Hailstorm: Disaggregated compute and storage for distributed lsm-based databases. In *Proceedings of the twenty-fifth international conference on architectural support for programming languages and operating systems*. 301–316.

[19] 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.

[20] Subarna Chatterjee, Meena Jagadeesan, Wilson Qin, and Stratos Idreos. 2021. Cosine: a cloud-cost optimized self-designing key-value storage engine. *Proceedings of the VLDB Endowment* 15, 1 (2021), 112–126.

[21] Guanduo Chen, Zhenying He, Meng Li, and Siqiang Luo. 2024. Oasis: An optimal disjoint segmented learned range filter. *Proceedings of the VLDB Endowment* 17, 8 (2024), 1911–1924.

[22] Xinqi Chen, Weidong Zhang, Zhongyu Wang, Erci Xu, Xiaolu Zhang, Dong Wu, Junping Wu, Ruiming Lu, Yaheng Song, Chaolei Hu, et al. 2026. Scheduling Cloud Block Storage Proactively and Reactively with Omar. *work* 25 (2026), 26.

[23] 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.

[24] 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.

[25] 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.

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

[27] 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.

[28] 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.

[29] 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.

[30] Chen Ding, Kai Lu, Quanyi Zhang, Zekun Ye, Ting Yao, Daohui Wang, Huatao Wu, and Jiguang Wan. 2025. DFlush: DPU-Offloaded Flush for Disaggregated LSM-based Key-Value Stores. *Proceedings of the ACM on Management of Data* 3, 3 (2025), 1–28.

[31] Chen Ding, Jian Zhou, Kai Lu, Sicen Li, Yiqin Xiong, Jiguang Wan, and Ling Zhan. 2024. D2comp: Efficient offload of lsm-tree compaction with data processing units on disaggregated storage. *ACM Transactions on Architecture and Code Optimization* 21, 3 (2024), 1–22.

[32] Siying Dong, Shiva Shankar P, Satadru Pan, Anand Ananthabhotla, Dhanabal Ekambaram, Abhinav Sharma, Shobhit Dayal, Nishant Vinaybhai Parikh, Yanqin Jin, Albert Kim, et al. 2023. Disaggregating rocksdb: A production experience. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–24.

[33] Zhuohui Duan, Hao Feng, Haikun Liu, Xiaofei Liao, Hai Jin, and Bangyu Li. 2025. {AegonKV}: A High Bandwidth, Low Tail Latency, and Low Storage Cost {KV-Separated}{LSM} Store with {SmartSSD-based}{GC} Offloading. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*. 321–335.

[34] 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.

[35] Navid Eslami, Ioana O Bercea, and Niv Dayan. 2025. Diva: Dynamic range filter for var-length keys and queries. *Proceedings of the VLDB Endowment* 18, 11 (2025), 3923–3936.

[36] Navid Eslami and Niv Dayan. 2024. Memento filter: A fast, dynamic, and robust range filter. *Proceedings of the ACM on Management of Data* 2, 6 (2024), 1–27.

[37] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. 2019. X-Engine: An optimized storage engine for large-scale E-commerce transaction processing. In *Proceedings of the 2019 International Conference on Management of Data*. 651–665.

[38] 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.

[39] Andy Huynh, Harshal A Chaudhari, Evimaria Terzi, and Manos Athanassoulis. 2024. Towards flexibility and robustness of LSM trees. *The VLDB Journal* 33, 4 (2024), 1105–1128.

[40] 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*.

[41] Rong Kang, Yanbin Chen, Ye Liu, Fuxin Jiang, Qingshuo Li, Miao Ma, Jian Liu, Guangliang Zhao, Tieying Zhang, Jianjun Chen, et al. 2025. ABase: the Multi-Tenant NoSQL Serverless Database for Diverse and Dynamic Workloads in Large-scale Cloud Environments. In *Companion of the 2025 International Conference on Management of Data*. 471–484.

[42] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. Redesigning {LSMs} for nonvolatile memory with {NoveLSM}. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 993–1005.

[43] 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.

[44] Eric R Knorr, Baptiste Lemaire, Andrew Lim, Siqiang Luo, Huanchen Zhang, Stratos Idreos, and Michael Mitzenmacher. 2022. Proteus: A self-designing range filter. In *Proceedings of the 2022 International Conference on Management of Data*. 1670–1684.

[45] 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.

[46] 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.

[47] Yongkun Li, Zhen Liu, Patrick PC Lee, Jiayu Wu, Yinlong Xu, Yi Wu, Liu Tang, Qi Liu, and Qiu Cui. 2021. Differentiated {Key-Value} storage management for balanced {I/O} performance. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 673–687.

[48] 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.

[49] 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.

[50] Junfeng Liu, Haoxuan Xie, and Siqiang Luo. 2025. [Extended Version] ArceKV: Towards Workload-driven LSM-compactions for Key-Value Store Under Dynamic Workloads. *arXiv preprint arXiv:2508.03565* (2025).

[51] 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.

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

[53] 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).

[54] Chen Luo and Michael J Carey. 2020. LSM-based storage techniques: a survey: C. Luo, MJ Carey. *The VLDB Journal* 29, 1 (2020), 393–418.

[55] 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.

[56] Rui Miao, Lingjun Zhu, Shu Ma, Kun Qian, Shujun Zhuang, Bo Li, Shuguang Cheng, Jiaqi Gao, Yan Zhuang, Pengcheng Zhang, et al. 2022. From luna to solar: the evolutions of the compute-to-storage networks in alibaba cloud. In *Proceedings of the ACM SIGCOMM 2022 Conference*. 753–766.

[57] Dingheng Mo, Fanchao Chen, Siqiang Luo, and Caihua Shan. 2023. Learning to optimize lsm-trees: Towards a reinforcement learning based key-value store for dynamic workloads. *Proceedings of the ACM on Management of Data* 1, 3 (2023), 1–25.

[58] Dingheng Mo, Siqiang Luo, and Stratos Idreos. 2025. How to grow an lsm-tree? towards bridging the gap between theory and practice. *Proceedings of the ACM on Management of Data* 3, 3 (2025), 1–25.

[59] 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.

[60] 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.

[61] 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.

[62] 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.

[63] Yuanming Ren, Siyuan Sheng, Zhang Cao, Yongkun Li, and Patrick PC Lee. 2026. Holistic and Automated Task Scheduling for Distributed {LSM-tree-based} Storage. In *24th USENIX Conference on File and Storage Technologies (FAST 26)*. 203–219.

[64] 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.

[65] 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.

[66] Subhadeep Sarkar, Tarikul Islam Papon, Dimitris Staratzis, and Manos Athanassoulis. 2020. Lethe: A tunable delete-aware LSM engine. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 893–908.

[67] 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).

[68] Zhitao Shen, Shiyu Yang, Weibo Chen, Kunming Wang, Yue Li, Jiabao Jin, Wei Jia, Junwei Chen, Yuan Su, Xiaoxia Duan, et al. 2025. TierBase: A Workload-Driven Cost-Optimized Key-Value Store. In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. IEEE, 571–584.

[69] Junyi Shu, Kun Qian, Ennan Zhai, Xuanzhe Liu, and Xin Jin. 2024. Burstable Cloud Block Storage with Data Processing Units. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 783–799.

[70] Hui Sun, Jinfeng Xu, Xiangxiang Jiang, Guanzhong Chen, Yinliang Yue, and Xiao Qin. 2024. gLSM: Using GPGPU to Accelerate Compactions in LSM-tree-based Key-value Stores. *ACM Transactions on Storage* 20, 1 (2024), 1–41.

[71] Xuan Sun, Jinghuan Yu, Zimeng Zhou, and Chun Jason Xue. 2020. FPGA-based compaction engine for accelerating LSM-tree key-value stores. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1261–1272.

[72] Jeff Swenson, Andy Kimball, Raphael'kena' Poss, Rebecca Taft, Jay Lim, Adam Storm, Sumeer Bhola, Paul Bulkley-Logston, Pj Tatlow, Rachael Harding, et al. 2025. CockroachDB Serverless: Sub-second Scaling from Zero with Multi-region Cluster Virtualization. In *Companion of the 2025 International Conference on Management of Data*. 648–661.

[73] Difan Tan, Jiawei Li, Hua Wang, Xiaoxiao Li, Wenbo Liu, Zijin Qin, Ke Zhou, Ming Xie, and Mengling Tao. 2025. Tela: A temporal load-aware cloud virtual disk placement scheme. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 1084–1100.

[74] Kapil Vaidya, Subarna Chatterjee, Eric Knorr, Michael Mitzenmacher, Stratos Idreos, and Tim Kraska. 2022. SNARF: a learning-enhanced range filter. *Proceedings of the VLDB Endowment* 15, 8 (2022).

[75] 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.

[76] Hengrui Wang, Te Guo, Junzhao Yang, and Huanchen Zhang. 2024. Grf: A global range filter for lsm-trees with shape encoding. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–27.

[77] Hengrui Wang, Jiaoyi Zhang, Jiansheng Qiu, Fangzhou Yuan, and Huanchen Zhang. 2026. Improving Range Scan Performance in LSM-trees with Group Caching. *Proceedings of the ACM on Management of Data* 4, 1 (SIGMOD (2026), 1–26.

[78] Xiaoliang Wang, Peiquan Jin, Yongping Luo, and Zhaole Chu. 2024. Range cache: An efficient cache component for accelerating range queries on lsm-based key-value stores. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 488–500.

[79] Yingjia Wang, Lok Yin Chow, Xirui Nie, Yuhong Liang, and Ming-Chang Yang. 2024. Znh2: Augmenting zns-based storage system with host-managed heterogeneous zones. In *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*. 1–8.

[80] Yingjia Wang, Caixin Gong, Guoyun Zhu, Sheng Wang, Zhengheng Wang, Huan Liu, Junzhi Shi, Wu Qin, Wei Zhang, Feifei Li, and Ming-Chang Yang. 2026. Making LSM-Tree-based Key-Value Store Practical and Efficient for Multi-Tenant Serverless Cloud Databases. *Proceedings of the ACM on Management of Data* (2026).

[81] 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.

[82] 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.

[83] Yingjia Wang and Ming-Chang Yang. 2026. Black-Box Performance Evaluation of Elastic Block Storage: Contract, Rate-Limiting Model, and Software Exploration. *arXiv preprint arXiv:2607.20319* (2026).

[84] Zhiqi Wang and Zili Shao. 2023. MirrorKV: An Efficient Key-Value Store on Hybrid Cloud Storage with Balanced Performance of Compaction and Querying. *Proceedings of the ACM on Management of Data* 1, 4 (2023), 1–27.

[85] Zhongyu Wang, Yaheng Song, Erci Xu, Haonan Wu, Guangxun Tong, Shizhuo Sun, Haoran Li, Jincheng Liu, Lijun Ding, Rong Liu, et al. 2024. Ransom Access Memories: Achieving Practical Ransomware Protection in Cloud with {DeftPunk}. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 687–702.

[86] Ziwei Wang, Zheng Zhong, Jiarui Guo, Yuhan Wu, Haoyu Li, Tong Yang, Yaofeng Tu, Huanchen Zhang, and Bin Cui. 2023. Rencoder: A space-time efficient range filter with local encoder. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 2036–2049.

[87] Fenggang Wu, Ming-Hong Yang, Baoquan Zhang, and David HC Du. 2020. {ACKey}: Adaptive caching for {LSM-based}{Key-Value} stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 603–615.

[88] Haonan Wu, Erci Xu, Ligang Wang, Yuandong Hong, Changsheng Niu, Bo Shi, Lingjun Zhu, Jinnian He, Dong Wu, Weidong Zhang, et al. 2025. Hey Hey, My My, Skewness is here to stay: Challenges and opportunities in cloud block store traffic. In *Proceedings of the Twentieth European Conference on Computer Systems*. 736–752.

[89] 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.

[90] Jiexiong Xu, Yiquan Chen, Yijing Wang, Wenhui Shi, Guoju Fang, Yi Chen, Huasheng Liao, Yang Wang, Hai Lin, Zhen Jin, et al. 2024. LightPool: A NVMe-of-based high-performance and lightweight storage pool architecture for cloud-native distributed database. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 983–995.

[91] Peng Xu, Nannan Zhao, Jiguang Wan, Wei Liu, Shuning Chen, Yuanhui Zhou, Hadeel Albahar, Hanyang Liu, Liu Tang, and Zhihu Tan. 2022. Building a fast and efficient LSM-tree store by integrating local storage with cloud storage. *ACM Transactions on Architecture and Code Optimization (TACO)* 19, 3 (2022),

1–26.

[92] Lei Yang, Hong Wu, Tieying Zhang, Xuntao Cheng, Feifei Li, Lei Zou, Yujie Wang, Rongyao Chen, Jianying Wang, and Gui Huang. 2020. Leaper: A Learned Prefetcher for Cache Invalidation in LSM-tree based Storage Engines. *Proc. VLDB Endow.* 13, 11 (2020), 1976–1989.

[93] Leping Yang, Yanbo Zhou, Gong Zeng, Li Zhang, Saisai Zhang, Ruilin Wu, Chaoyang Sun, Shiyi Luo, Wenrui Li, Keqiang Niu, et al. 2026. Here, There and Everywhere: The Past, the Present and the Future of Local Storage in Cloud. In *24th USENIX Conference on File and Storage Technologies (FAST 26)*. 1–17.

[94] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. 2020. {MatrixKV}: Reducing write stalls and write amplification in {LSM-tree} based {KV} stores with matrix container in {NVM}. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 17–31.

[95] Hobin Yoon, Juncheng Yang, Sveinn Fannar Kristjansson, Steinn E Sigurdarson, Ymir Vigfusson, and Ada Gavrilovska. 2018. Mutant: Balancing storage cost and latency in lsm-tree data stores. In *Proceedings of the ACM Symposium on Cloud Computing*. 162–173.

[96] Qiaolin Yu, Chang Guo, Jay Zhuang, Viraj Thakkar, Jianguo Wang, and Zhichao Cao. 2024. CaaS-LSM: compaction-as-a-service for LSM-based key-value stores in storage disaggregated infrastructure. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–28.

[97] Weiping Yu, Siqiang Luo, Zihao Yu, and Gao Cong. 2024. Camal: Optimizing lsm-trees via active learning. *Proceedings of the ACM on Management of Data* 2, 4 (2024), 1–26.

[98] 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.

[99] Jianshun Zhang, Fang Wang, Sheng Qiu, Yi Wang, Jiaxin Ou, Junxun Huang, Baoquan Li, Peng Fang, and Dan Feng. 2024. Scavenger: Better space-time trade-offs for key-value separated LSM-trees. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 4072–4085.

[100] Qiang Zhang, Yongkun Li, Patrick PC Lee, Yinlong Xu, and Si Wu. 2022. {DEPART}: replica decoupling for distributed {Key-Value} storage. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. 397–412.

[101] Teng Zhang, Jianying Wang, Xuntao Cheng, Hao Xu, Nanlong Yu, Gui Huang, Tieying Zhang, Dengcheng He, Feifei Li, Wei Cao, et al. 2020. {FPGA-Accelerated} compactions for {LSM-based}{Key-Value} store. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 225–237.

[102] Weidong Zhang, Erci Xu, Qiuping Wang, Xiaolu Zhang, Yuesheng Gu, Zhenwei Lu, Tao Ouyang, Guanqun Dai, Wenwen Peng, Zhe Xu, et al. 2024. What's the Story in {EBS} Glory: Evolutions and Lessons in Building Cloud Block Store. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*. 277–291.

[103] 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.

[104] Wenshao Zhong, Chen Chen, Xingbo Wu, and Song Jiang. 2021. {REMIX}: Efficient range query for {LSM-trees}. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 51–64.

[105] 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.