Calcspar: A Contract-Aware LSM Store for Cloud Storage with Low Latency Spikes

Abstract

Cloud storage is gaining popularity because of features such as pay-as-you-go that significantly reduces storage costs. However, the community has not sufficiently explored its contract model and latency characteristics. As LSM-Tree-based key-value stores (LSM stores) become the building block for numerous cloud applications, how cloud storage would impact the performance of key-value accesses is vital. This study reveals the significant latency variances of Amazon Elastic Block Store (EBS) under various I/O pressures, which challenges LSM store read performance on cloud storage. To reduce the corresponding tail latency, we propose Calcspar, a contract-aware LSM store for cloud storage, which efficiently addresses the challenges by regulating the rate of I/O requests to cloud storage and absorbing surplus I/O requests with the data cache. We specifically developed a fluctuation-aware caching to lower the high latency brought on by workload fluctuations. Additionally, we build a congestion-aware IOPS allocator to reduce the impact of LSM-Tree internal operations on read latency. We evaluated Calcspar on EBS with different real-world workloads and compared it to the cutting-edge LSM stores. The results show that Calcspar can significantly reduce tail latency while maintaining regular read and write performance, keeping the 99^{th} percentile latency under 550μs and reducing average latency by 66%.

1 Introduction

In recent years, the trend that many businesses and organizations shift their data to the cloud has fueled the growth of cloud storage [19, 31]. This is due to the advanced features and cost-effectiveness of cloud storage. For example, Amazon Web Services (AWS), the world's most broadly adopted cloud platform, provides various storage services with high scalability and reliability on a pay-as-you-go basis [7] (e.g., Elastic Block Store, EBS), making them more appealing. Another important trend is that LSM-Tree-based key-value stores (LSM stores), such as RocksDB [5], LevelDB [1], Bigtable [13], Dynamo [16] and TiDB [17], are becoming the building block for many cloud applications. However, none of the existing LSM stores is optimized for cloud storage to eliminate long-tail latency. Notably, it is challenging to balance the estimated peak performance with the budget for cloud storage performance (e.g., IOPS). Although many cloud storage providers advertise elastic storage volumes that can accommodate changing performance needs, these volumes' scaling capabilities fail to adapt to an inevitable traffic fluctuation. For instance, AWS EBS only supports increasing the purchased IOPS, which would take hours or even days to complete [2]. Hence, it is impractical to rely solely on elastic volumes for timely adjustments in the face of short-term workload changes.

To understand how cloud storage would respond to traffic fluctuation, we have explored the latency characteristics of AWS EBS volumes. Results show that EBS guarantees a service agreement called Service Level Agreement (SLA) in which the processing latency of each request falls within an appropriate threshold *if* the accesses do not exceed the paid IOPS. We observe that the processing latency of each consecutive request dramatically increases when the demands in a time window exceed the IOPS agreement. Besides, the cloud storage's contract model shows that the higher the paid IOPS, the lower the latency. However, such an agreement is constrained by IOPS budgets, and naturally, performance in terms of latency will suffer if the IOPS is overdrawn.

The latency spike caused by limited IOPS in cloud storage severely impacts the performance of latency-sensitive applications on top of LSM stores. We take one of the most widely deployed LSM store implementations, RocksDB, as an example. The RocksDB first writes the in-memory table (memtable) to respond quickly with reasonably low latency. Until the in-memory table is full, RocksDB then persists the table to the storage volume in large chunk writes (e.g., SSTables), thus aggregating the random writes into sequential ones. Such a write scheme reduces the number of write requests and achieves high write throughput. It then employs internal compaction mechanisms to merge and resort the incoming data with multiple levels of on-disk tables. Although the internal compaction operation ensures the orderliness of data in each

level to improve the lookup performance, a read operation still needs to traverse multiple levels, resulting in read amplification. As the IOPS on a cloud storage volume is limited, the read performance of RocksDB is significantly throttled.

There are several challenges to avoid read latency spikes in LSM stores. First, the read request performance fluctuates significantly because the cloud storage volume isn't flexible enough to timely keep up with the changing workload. The fluctuating workload causes the number of read I/Os of an LSM store to access cloud storage volumes to vary significantly. The request latency increases when the I/O number exceeds the paid IOPS of the cloud storage volume. Second, the read amplification in an LSM store further strengthens the workloads fluctuations. Multi-level data layouts inevitably cause read amplification problems, such as those found at the LSM-Tree L0 level requires traversing multiple tables, so reading a single key-value pair may generate multiple I/Os. Third, the speed limit mechanism of cloud storage volume conflicts with LSM stores' internal multi-thread concurrency mechanism, and requests among multiple threads congestion on the cloud storage volume leads to an increase in latency multiples. Fourth, LSM stores' internally inherent mechanisms amplify the damage on the read latency of cloud storage volumes. Irregular flush operations or indeterminate size compaction operations cause a sudden increase in the number of I/Os accessing the cloud storage volume, resulting in high tail latency. Finally, the tradeoff between cost and performance increases the cost exponentially to get better tail latency, resulting in significant resource waste and limited throughput improvement.

One natural solution to the above challenges is contracting a higher IOPS budget with cloud storage volumes, ensuring that the LSM store's I/O number do not exceed the paid IOPS to maintain optimal latency. However, this raises the costs. Also, the peak IOPS demand in real production environments is difficult to predict. Instead, we aim to explore the best performance of an LSM store under a specific IOPS budget.

This paper presents Calcspar, a cloud storage volume contract-aware LSM store based on Amazon's EBS with reduced latency spikes, and it tolerates both external workload fluctuations and internal operation contentions. Calcspar first employs fluctuation-aware caching, which combines hotspot-aware proactive prefetching and shift-aware passive caching, to adapt to changing workloads. The prefetching strategy identifies hotspots for high load periods and proactively fetches them during low load periods, thus smoothing out the external load changes. Then, during the high load periods, the passive caching leverages the temporal locality to extrude the stale prefetched data and adapt to hotspot shifts without issuing extra requests. Calcspar then leverages a congestion-aware IOPS allocator to assign priority for different internal requests and avoid elevated latency due to limited IOPS budgets. The allocator employs a multi-queues throttling structure to prevent thread congestion. The opportunistic compaction then assigns write requests in different LSM levels into different priority queues, thus balancing the read amplification and write throttling. The contributions of this paper include:

1) We conducted an in-depth analysis of the performance of cloud storage volumes, which first illustrates the unwritten contract between latency and load pressures.

2) We propose a rate-limiting performance model for cloud storage volumes based on the observations, experimentally validate the model and reveal opportunities to obtain optimal latency.

3) Our proposed Calcspar is better suitable for AWS cloud storage volumes where IOPS budgets are vital to the performance and significantly reduced the tail latency of LSM-Tree.

The rest of this paper is organized as follows. Section 2 takes Amazon's EBS as the example to models the performance characteristics of cloud storage volumes. The challenges of reducing the latency for an LSM store on cloud storage are discussed in section 3. Calcspar designs are then introduced in section 4 to address these challenges and they are evaluated in section 5. Finally, the related work and conclusions are presented in sections 6 and 7, respectively.

2 Modeling Cloud Storage Performance

2.1 Contract Model of Cloud Storage

The cloud storage providers, such as AWS, offer a variety of cost-efficient storage volumes for users to meet their distinct needs and adapt to the changing market. Table 1 shows the contract model of the cloud storage, which illustrates the price and performance relationship of the corresponding volume type. The pricing is based on block storage in the AWS ap-northeast-2 region in July 2022 [3, 4]. The contract model indicates that as the price of IOPS increases, the lower latency of the corresponding type. Thus, it entails users choosing the appropriate storage volume and IOPS budget based on their needs. However, the paid IOPS only guarantees the number of returned I/Os rather than the optimal latency. Also, EBS performance scaling supports only increasing paid IOPS and takes hours to days to take effect [2]. There's no agreement on how the request would be responded to when the loads exceed the IOPS. Hence the corresponding latency characteristics are widely ignored by existing LSM stores.

2.2 Unwritten Latency Performance

To unwrap the hidden latency characteristics and understand how the above contract model would affect the performance of an LSM store, we first perform a series of experiments on cloud storage volumes, then proposing a performance model.

Table 1: EBS IOPS prices and latencies.

Type	Init IOPS	IOPS price ($)	Latency (μs)
gp2	3×GB	0.038	∼200
gp3	3000	0.0058	∼300
io1	100	0.0666	∼100
io2	100	0.067	∼10

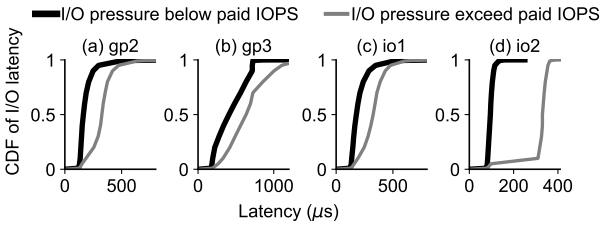

Figure 1: Latency CDF of different cloud storage.

Experiment #1: *Cumulative Distribution Function (CDF) of latency under vary I/O pressures.* We measure the latency of EBS volumes gp2, gp3, io1, and io2 with paid 3000 IOPS for each by sending 4KB random read requests with varying pressures. We employ fio [20] to tune the I/O pressure by controlling the size of **Submit IOPS**, which is the number of I/Os submitted to EBS per second. Yet, the cloud storage volume *won't* handle more than the paid IOPS. Figure 1 shows the latency CDF results that support the following two findings.

Finding 1: When the I/O pressure exceeds the paid IOPS, the latency increases deterministically and significantly. On the contrary, their average latency performance is much better when the Submit IOPS is under paid IOPS. For example, the average latency of io2 even reached 100μs.

Finding 2: The IOPS budget is proportional to the cost when considering Table 1. The slightly higher-cost io2 has the best and most stable latency. The latency performance of the cheapest gp3, which initially provides 3000 IOPS, is far lower than the other three EBS types.

Experiment #2: *Limited IOPS budgets*. In this experiment, we explore the latency CDF under different IOPS budgets. We evaluate one io2 under two different pressures. Request latency distributions are given in Figure 2. Results indicate that regardless of the paid IOPS, the access latency when the Submit IOPS exceeds the paid IOPS is more than 5× worse than the access latency when not exceeded. The 99th percentile latency are below 270μs when the Submit IOPS

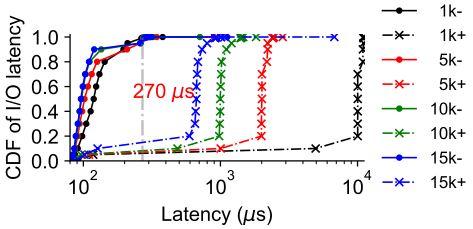

Figure 2: Latency CDF under different paid IOPS. "1k" means io2's paid IOPS. "+" indicates that the Submit IOPS exceeds the paid IOPS; "-" means not exceed.

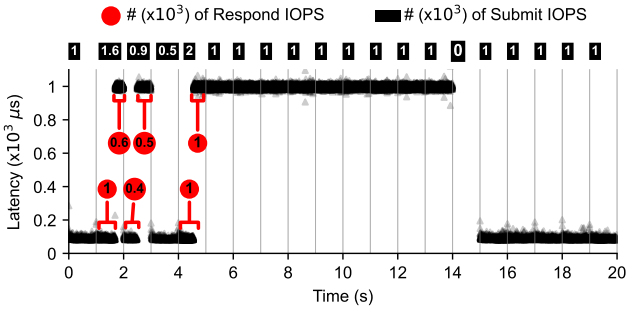

Figure 3: Latencies under rigorously controlled pressure. Respond IOPS is the number of requests returned.

does not exceed the corresponding paid IOPS. However, when the Submit IOPS exceeds the paid IOPS, the access latency increases to 1000∼11000μs. These long-tail latencies degrade the user experience.

Experiment #3: *Elevated latency spike*. To explore reasons behind the latency spike under high I/O pressures, we rigorously control the I/O send rate in a single thread for an 1000-IOPS io2 volumes. The latency of each request is shown in Figure 3. In the 1st second, when the Submit IOPS does not exceed the paid IOPS, the latency is lower than 200μs. In 2nd second, the Submit IOPS is 1600, the latencies of the first 1000 requests are identical to that of the first second. However, the latencies of the rest 600 requests increase significantly to about 1000μs, which renders $1/IOPS$ second. The Submit IOPS in the 5th second is twice the paid IOPS, the first 1000 requests can get low latency while the latencies of the last 1000 requests equal $1/IOPS$ second again. Although the Submit IOPS drops to 1000, the latencies of subsequent requests remain high. Until we pause the workload at the 14th second and resume it at the 15th second, the latencies recuperate.

Speculative Reason #1: *We speculate the reason behind the observation is due to the speed-limiting mechanism inside EBS, which handles the current excess I/O by overdrawing the next 1 second of IOPS, and at the same time, the "punitive" improvement latency is 1/IOPS to prevent the requests beyond the payment from continuing to be responded.*

For example, the last 600 I/O requests in the 2nd-second overdraw 600 IOPS from the 3rd second. Hence, only the remaining 400 (=1000 − 600) can be served quickly in the 3rd second. The overdraft is paid off when no I/O request is sent in the 14th second. Therefore, the latency returns to a lower level in the following 15∼19 seconds

Since the resources of cloud services are on a pay-per-use basis, cloud storage providers use this mechanism to maintain SLAs to prevent users from constantly acquiring benefits beyond what they paid. Meanwhile, by increasing the delay, the operation continues from the user's perspective; thus, there is no opportunity for recalling the service.

Experiment #4: *Thread congestion.* The effect of the number of threads on the latency is evaluated in this experiment. Two different I/O pressures are sent to each io2 volume with

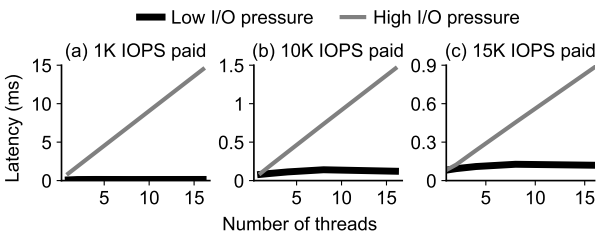

Figure 4: Average latency under different # of threads.

different number of threads. Request average latencies are shown in Figure 4. When the Submit IOPS does not exceed the paid IOPS, the average latency holds around 120μs and does not increase with more threads. However, when the Submit IOPS is higher than the IOPS paid, the access latency first grows to the level of $1/IOPS$ seconds while growing linearly with the number of threads. This phenomenon is consistent with queuing theory [27], so we conjecture the following based on the knowledge gained from queuing theory:

Speculative Reason #2: *Requests are queued when they enter the EBS, and the queue size equals the paid IOPS. The latency in the EBS will follow the following two rules: 1) When the Submit IOPS is lower than the paid IOPS, the latency of each request will be around the threshold, depending on the EBS type. 2) When the Submit IOPS exceeds the paid IOPS, the latency (respone time W) conforms to the queueing equation $W = L/A$, where A is the paid IOPS and L is the number of threads.*

2.3 Latency Model of EBS

Based on the above analysis, we construct an EBS latency model to reveal the internal latency mechanism of cloud storage, as shown in Figure 5.

Suppose an EBS volume a buffer queue (called I/O domain) to maintain requests, where the queue length is equal to the paid IOPS. When a request arrives, the I/O domain allocates a free slot in the queue. EBS then pulls requests from the I/O domain to promptly executes them. When the number of the responded requests exceeds paid IOPS, the EBS stops fetching new requests. Consequently, the pending requests are blocked until EBS can resume the services. Congestion can occur under multi-threaded workloads, although the total number of requests submitted by all threads is estimated to be no higher than the budget in one second. This is because thread scheduling uncertainties would fire some working threads more often, thus accidentally overdrawing the budget. Such an overdrawing leads the consequent requests to be blocked, resulting in significantly higher latency per request.

Overdraft Rule. EBS controls the responding speed of the request to manage the *rotation* of the I/O domain. Specifically, we use the “tokens" to describe the speed control mechanism of EBS. Each cloud storage volume retains a token bucket and a borrowing pool, which contains the same number of tokens equal to the paid IOPS. The user request first gets the regular token from the token bucket. If no token is in

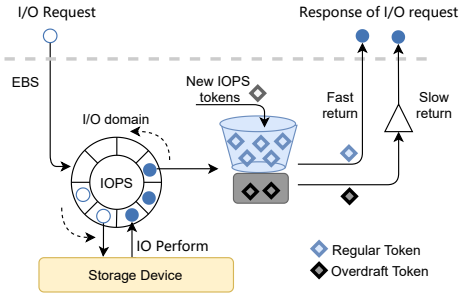

Figure 5: Estimated EBS IOPS throttling mechanism.

the bucket, the overdraft token must be obtained from the borrowing pool. EBS ensures that requests carrying regular tokens return quickly. In contrast, requests with overdraft tokens are processed slowly to ensure that the user does not use too many IOPS to maintain SLAs. The EBS is replenished with IOPS tokens per second, which are prioritized in the borrowing pool.

The above EBS latency model explains the aforementioned findings (#1 and #2) and speculations (reasons #1 and #2) about cloud storage latency: 1) When the Submit IOPS does not exceed the paid IOPS, requests get regular tokens and return quickly, and requests between threads do not block, so the optimal latency is obtained. 2) When the Submit IOPS exceeds the paid IOPS, some requests obtain overdraft tokens. 3) When the return speed is lower than the request arrival speed, the unprocessed requests will fill the I/O domain slots, and further tokens are replenished by replenishing the borrowing pool first so that the high latency state will last for a long time. 4) The inter-thread blocking in I/O domain leads to increasing user-perceived latency.

3 Modeling RocksDB Performance

3.1 RocksDB under IOPS Limitation

As suggested by the above performance model, cloud storage emphasizes more limitations on latency and IOPS rather than bandwidth. The evidence is that the allowed request size is relatively large, and increasing the IOPS budget is expensive. Such a contract and performance model fits bandwidth-sensitive workloads; however, it is unfavored by many latency-sensitive and request-heavy workloads. For example, our analysis indicates that the data write and compaction in RocksDB works well on cloud storage as its’ request sizes are more significant and the number of requests is limited. However, the read performance of RocksDB is seriously affected by the limited IOPS budget. Many applications that employ RocksDB to serve metadata indexing expect excellent read throughput as well as low and stable read latency [25]. Unfortunately, they will fail to achieve their purpose if the underline device is cloud storage.

RocksDB does not work well on cloud storage because its LSM-Tree is generally a write-optimized indexing struc-

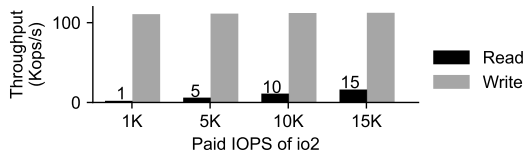

Figure 6: RocksDB performance on EBS-io2.

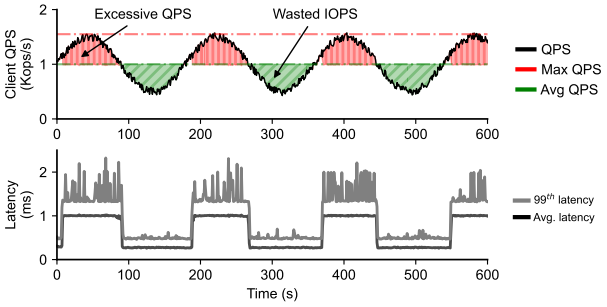

Figure 7: Client random read QPS and request latency with single thread.

ture instead of tuned for reducing the number of read I/Os. Moreover, the hierarchical structure of the LSM-Tree results in many read requests needing to access multiple levels of the indexing table to locate the corresponding key. On the contrary, write operations can be cached and aggregated into large chunks before they finally hit the cloud storage in one backend request. As shown in Figure 6, we perform read and write stress tests on io2 storage volumes with different paid IOPS. As the IOPS increases, the read throughput increases, and the write throughput remains the same. Therefore, when deploying RocksDB on the cloud, the maximum 1000MB/s bandwidth of the cloud storage volume is usually sufficient. However, the RocksDB read I/Os will easily and repeatedly hit the IOPS limitations, which causes elevated tail latency.

3.2 Challenges in Avoiding Latency Spikes

This subsection elaborates on several challenges encountered when optimizing the read performance of RocksDB in cloud storage. To analyze the performance, we employ Facebook's most recent benchmark Mixgraph [11], which synthetically generates key-value requests that accurately represent the real-platform load fluctuation.

Challenge #1: *The read latency fluctuates significantly because cloud storage isn't flexible enough to meet the changing demand.* In this experiment, we send fluctuating read requests

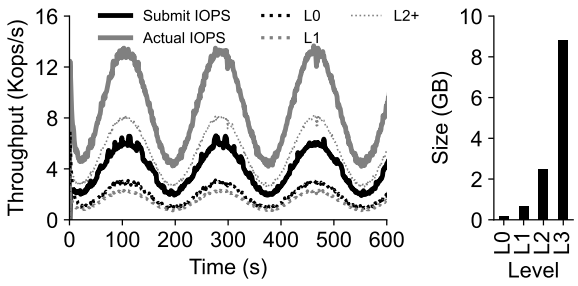

Figure 8: Read amplification of RocksDB.

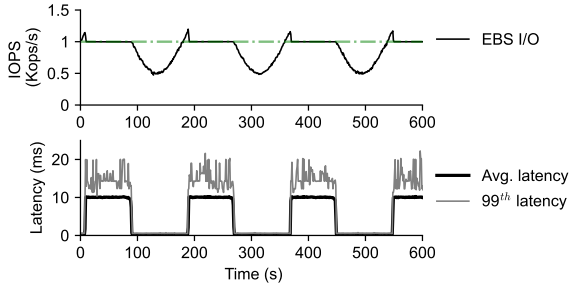

Figure 9: Io2 Respond IOPS and latency with 10 threads.

to an io2 volume in a single thread. To reduce expense, the paid IOPS of io2 storage volumes is the average of fluctuating requests. The experimental results in Figure 7 show that RocksDB is significantly affected by the latency characteristics of cloud storage volumes. When the client queries per second (QPS) exceeds the paid IOPS, the QPS being executed (the white part under the black line) can exceed the paid IOPS in a brief period, which is due to the overdraft rule. Then executed QPS will fall to the paid IOPS and the user-perceived latency is significantly higher than during low load. This also results in the user's excessive requests not being executed (the red part of the top in Figure 7), and they will be blocked. As the number of client requests drops below the paid IOPS, the latency drops to the bottom level, which causes another problem where the paid IOPS are wasted during low-load periods, as shown in green in Figure 7.

Challenge #2: *The read amplification in LSM-Tree further strengthens the workloads fluctuation.* RocksDB's write aggregation and hierarchical data layouts result in significant read I/O amplification. As shown in Figure 8, the actual IOPS is about 3× of the submit IOPS, amplifying the volatility of the load, when the access I/O exceeds the paid IOPS, the request latency will be high. In addition, the L0 and L1 level, which occupy a very small amount of data, but taking up close to 1/3 of the I/O accesses.

Challenge #3: *Thread I/O competition.* Requests among multiple threads are congested in the I/O domain resulting in a multiplication of tail latency. We send the same QPS as in Figure 7 with ten threads. The results in Figure 9 show that the latency increases 10× at high loads compared to single threads. This is because with more requests being sent to the io2 per second, the requests are returned slower than the requests that enter the I/O domain. When the I/O domain is full, requests are queued on each thread, so the latency perception is a multiple of the thread.

Challenge #4: *Bulk write blocking.* We sent some write requests while keeping the read QPS constant to measure the impact of write operations on user read requests, and the results are shown in Figure 10. At the 70th second, RocksDB launched compaction operations, which took up more IOPS, causing some of the user's requests to wait in a queue. Hence the 99th percentile latency bursts to 40ms. Secondly, at the 300th second, even in low workload period, RocksDB inter-

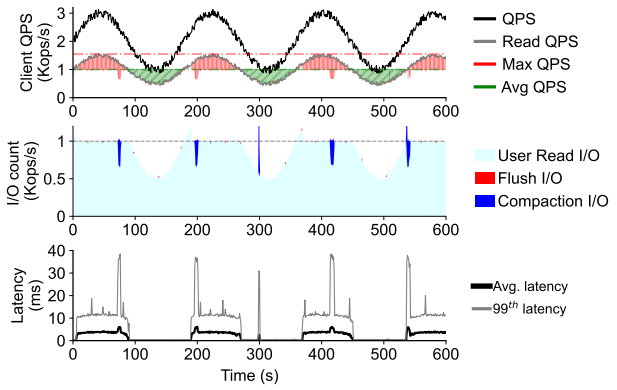

Figure 10: RocksDB I/Os and latency for a mixed read and write load with 10 threads.

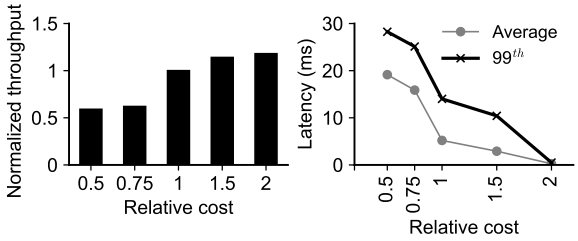

Figure 11: RocksDB performance at different storage costs.

nally initiates some compression operations, which blocks user requests, resulting in high latency.

Challenge #5: *Performance vs. cost.* We choose the average of the highest QPS and the lowest QPS of the load as the benchmark cost to measure the performance of RocksDB when choosing storage volumes with different costs under the same load. Results in Figure 11 show that, as the cost increases, although the latency appears to decrease, the throughput does not improve due to serious resource waste.

4 Calcspar Design

To build an LSM store that fully exploit the optimal latency of cloud storage volumes, the high latency caused by the observed overdraft rules and thread I/O congestion must be addressed. Rather than simply increasing expenses to improve the paid IOPS of storage volumes to avoid overdraft and congestion, we propose Calcspar to investigate the optimal latency of LSM stores in a cost-effective manner. With a limited number of IOPS available per second, Calcspar's four designs in Figure 12 holistically answer questions on getting the optimal latency: (1) How to smooth out I/O plateaus that are higher than the paid IOPS (§4.1 and §4.3). (2) How to take the most of available I/Os per second for user and LSM internal I/O requests (§4.2 and §4.4).

4.1 IOPS Stabilizer for EBS

Calcspar first aims to prevent latency fluctuation from the EBS. As discussed in §2.2, the overdraft rules do not result in throughput improvements but higher processing latency for EBS under high request pressures. Once the EBS enters

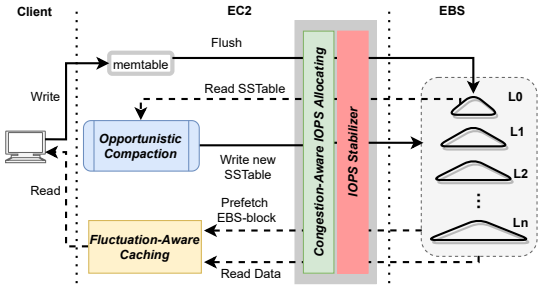

Figure 12: Architecture of Calcspar.

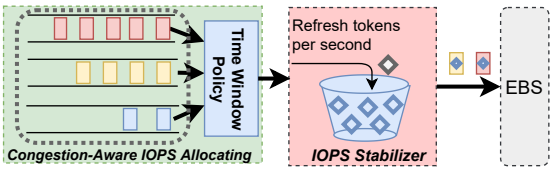

Figure 13: Congestion-Aware IOPS Allocator and Stabilizer

the overdraft status, the application is not able to withdraw any pending requests, thus missing opportunities for further optimization but waiting. Rather than passively detecting unexpected latency spikes, Calcspar proactively controls the number of I/Os during high-load periods by only submitting requests with the highest priority. To achieve this, Calcspar employs the observed EBS latency model (§2.3).

Figure 13 details the IOPS Stabilizer, which throttles the request rate to match the EBS I/O budget, thus eliminating overdraft latency spikes. The essence is to mimic the token speed limit mechanism, which insides EBS and is widely ignored, to the upper-layer applications. We demand each request must obtain a token before accessing the EBS. The number of tokens is refreshed every second decided by the paid IOPS. By controlling the number of tokens, Calcspar guarantees requests sent to EBS do not exceed the paid IOPS, thus EBS processing latencies can be secured in the tens of microseconds. Consequently, applications can expect more stable latencies once a request successfully obtains a token.

4.2 Congestion-Aware IOPS Allocating

The second goal of Calcspar is to eliminate the latency spike caused by request congestion among threads. To minimize cloud storage costs, we assume the paid IOPS is only guaranteed to meet I/Os for the average usage. Hence, the muti-threads design adopted by modern LSM stores will inevitably congest due to limited tokens provided by the IOPS Stabilizer. Many works prioritize the execution of latency-sensitive I/Os by adjusting the I/O stack or leveraging multi-device parallelism [10, 18, 24]. However, they can not prevent less critical requests from occupying unnecessary tokens. Calcspar uses multi-priority queues to ensure that critical requests are not occasionally blocked and are served in a best-effort manner.

Calcspar first categorizes different requests into multiple queues, then conditionally allocates IOPS budges for them. Calcspar treats I/Os that straightly affecting the LSM store

as *user-aware requests*, and classifies I/Os that do not instantly hinder user requests as *non-user-aware requests*. User-aware requests are mainly for responding foreground user requests, so Calcspar places them into the highest priority queue. Non-user-aware requests are mainly from LSM background tasks (e.g., compactions and prefetching). Since different background tasks impact the read/write amplifications differently, Calcspar assigns them to medium-priority or low-priority queues. For example, prefetching requests go into the lowest priority queue. Note that compaction requests are further differentiated later (§4.4).

Calcspar uses a dynamic time window policy to make the most of available paid I/Os per second. A time window for a queue grants a period of time for its requests to obtain tokens, so the size of a window matches the priority of the queue. Time windows of queues are tail-aligned, so that Calcspar starts handling I/Os from the higher priority queue earlier than those in the lower priority queues. The highest priority queue has the largest time window, ensuring its requests are processed first to get tokens and avoid being blocked by other requests. Calcspar then dynamically adjusts the sizes of window for other queues based on the number of tokens remaining. If requests from the highest priority queue consume too many tokens, the time window sizes for mid and low priority queues are reduced proportionally.

4.3 Fluctuation-Aware Caching

Considering the read I/O amplification problem of an LSM store will cause significant latency spikes on the EBS, an EBS latency-aware cache plays an essential role on flattening I/O request plateaus to the EBS when the workload is heavy as well as improving the paid IOPS utilization when the workload is light. We find few existing cache schemes are designed on this purpose, and their design metrics do not take into account the available paid IOPS of the underlying EBS. This will significantly affect choices of the optimal cache policy when workload fluctuates.

Hotspot-Aware Proactive Prefetching. When the workload is light, Calcspar consumes spare paid IOPS to trade for a better cache hit ratio by prefetching SSTable. Calcspar manage data in the unit of EBS-block (e.g., 256KB, which is the maximum size allowed by the EBS for one I/O request), this ensures that each prefetching I/O reads as many data as possible. Calcspar then maintains a global table to track the hotness of EBS-blocks using the exponential smoothing algorithm based on their access history. Furthermore, Calcspar periodically and proactively rewarms the frequently accessed LSM top layer (e.g., L0 and L1) data, because LSM stores retrieves key-value pairs in a top-to-bottom layer fashion.

Shift-Aware Passive Caching. When the workload is heavy, an EBS latency-aware cache should minimizes its I/Os to the EBS while improving space efficiency. Calcspar manages the cache space passively in this case. Calcspar refines cache space management in the unit of 4KB and uses the LRU policy for better space efficiency because evicting any data can be punished by competing one I/O with user requests to access the EBS.

Cache Integration. Calcspar integrates the two cache policies introduced above and switches between them based on workload. These two policies manages the same cache space, but at any time only one of them is active and actually evicts data. When the highest priority queue requests consume more than 95% of the tokens, Calcspar considers the workload to be heavy and activates the Shift-Aware Passive Caching policy. Otherwise, Calcspar harvests the available paid IOPS using the Hotspot-Aware Proactive Prefetching policy. It is worth noting that the global track table has a negligible memory overhead, as 1GB of data requires about 64KB of memory.

4.4 Opportunistic Compaction

The last goal of Calcspar is to remedy LSM compaction I/Os. After launching a compaction, its bulk read operations on at least two SSTables and write operations on at least one SSTable will compete with user I/O requests. An LSM store, on the other hand, retrieves a key-value pair level by level and merges SSTables in a copy-on-write manner, providing Calcspar with opportunities for differentiating I/Os for compaction jobs on different levels, thereby mitigating the competition on paid IOPS from LSM compaction operations. For L0 SSTables, which significantly affects read I/O amplifications, Calcspar prioritizes compaction on them. For L1 and L2 SSTables, Calcspar puts their compaction I/Os into the medium priority queue, where they are opportunistically processed. As for SSTables in levels below L2, Calcspar assigns these compaction I/Os to the lowest priority queue, since short-term deferral has no noticeably affect on performance.

5 Evaluation

We implement Calcspar based on RocksDB and evaluate it to demonstrate its advantages. Specifically, we perform an extensive time delay to answer the following questions. (1) How does Calcspar perform compared to the state-of-the-art approach? (§5.2) (2) The impact of several techniques of Calcspar on performance. (§5.3, §5.4, §5.5) (3) The sensitivity analysis of Calcspar (§5.6).

5.1 Experimental Setup

Test platform. We employ the most widely deployed AWS as our test platform. The EC2 instance is m5d.2xlarge, configured with 8 vCPUs and 32 GB Memory. A representative io2 storage volume with 100 GB capacity and 1000 IOPS is used by default for performance evaluation.

Comparisons. We compare Calcspar with RocksDB and the other three state-of-the-art key-value stores. They are: 1) Autotuned RocksDB [6]: isolating the I/O bandwidth between user requests and internal flush/compaction operations to improve tail latency. 2) SILK [9]: Opportunistically allocates

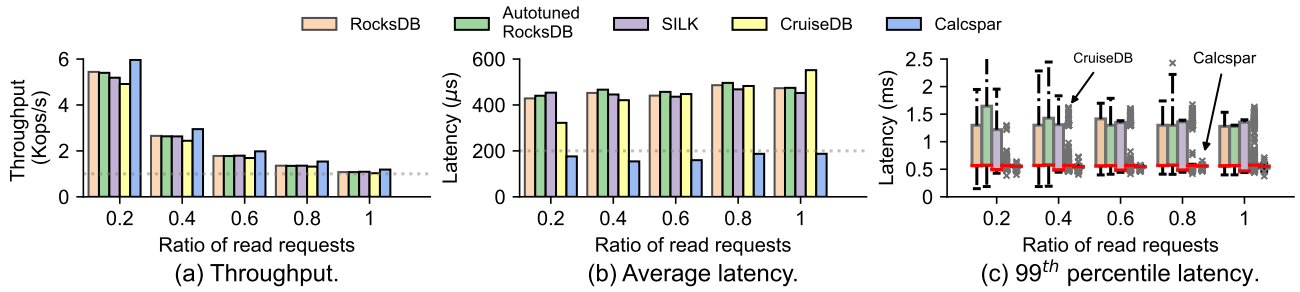

Figure 14: Evaluation results with different read request ratios under Mixgraph workload.

bandwidth to different internal I/O operations and allows low-level Compaction preemption. 3) CruiseDB [23]: Maintains SLAs employing an adaptive access mechanism based on memory usage, removes L0, and optimizes memory buffer. To make a fair comparison, all the databases take 4 threads for compaction and 4 for flushing. The default key, value, and SSTable sizes are set to 16B, 256B, and 8MB. The size of Memtables is set to the default value for RocksDB. A cache space of 500MB is opened for each database.

Table 2: YCSB workload characteristics.

Workload	Description
A	write-intensive:50% Update, 50% Read, Zipfian
B	read-intensive: 5% Update, 95% Read, Zipfian
C	read-only: 100% Read, Zipfian
D	read-latest: 95% Read, 5% Insert, Latest
E	scan-intensive: 5% Update, 95% Scan, Zipfian
F	write-intensive:50%Read,50%read-modify-write,Zipfian

Benchmarks. Two benchmarks, Mixgraph [11] and YCSB [14] are used to evaluate performance. YCSB is a widely used benchmark for evaluating the key-value store systems, providing six workloads configurations and key-value pair access distribution models listed in Table 2. YCSB can also provide uniform distribution workloads. Mixgraph is the latest benchmark test developed by Facebook. The workload is more spatially localized to simulate Facebook production workloads better and generate more accurate key-value queries. Benchmarks run in 10 threads in all the key-value stores by default, except for SILK, as multi-threading is not supported. In all experiments, 100 million Key-value pairs are first inserted into the key-value store system, and the key-value store has about 25 GB of data in its initial state.

5.2 Overall Performance

We first use the latest Mixgraph with fluctuating load characteristics to evaluate the overall performance of the five key-value stores on the cloud. Then we evaluate the performance using the YCSB benchmark and explore the effect under uniform load using the YCSB benchmark. To guarantee the fairness of the evaluation, the hardware resource allocation is the same for each key-value store. All evaluations start by randomly writing 100 million key-value pairs and executing one million requests.

The Mixgraph benchmarks. Figure 14 shows the performance for different read/write ratio configurations under Mixgraph. Since read requests affect IOPS more, the ratio of read requests is increased by varying the number of write requests. Based on the comparison of test results, the following conclusions can be drawn: 1) The throughput of Calcspar is better than other systems under all read ratios. In Figure 14(a), Calcspar throughput exceeds paid IOPS the most because of the high spatial locality of the Mixgraph load that is fully exploited. 2) Calcspar significantly reduces the average latency. As seen in Figure 14(b), the average latency of Calcspar does not exceed 200μs, which is 45$\sim$66% lower than the average latency of other key-value store systems. 3) Calcspar achieves a lower and more stable tail latency. Figure 14(c) shows the statistical plot of 99th percentile latency, and it can be seen that Calcspar has the smallest box plot volume with almost no outliers of extra-long delays. The 99th percentile latency can be stabilized at around 0.55ms with minimal fluctuations. CruiseDB also reduces tail latency by limiting request access, but it is unstable and sacrifices throughput.

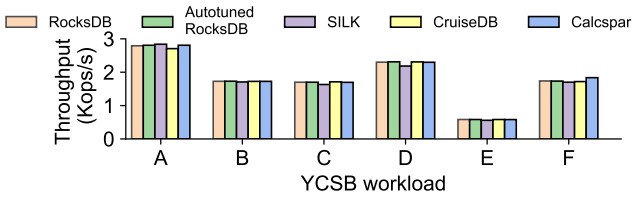

Figure 15: Throughput under YCSB workload.

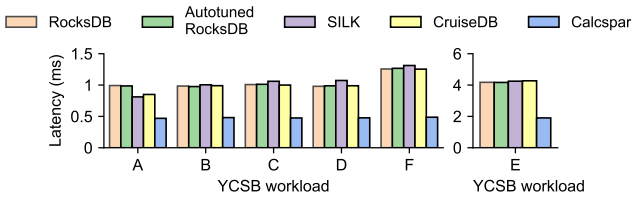

Figure 16: The 99th percentile latency under YCSB workload.

The YCSB benchmarks. We use YCSB workloads with stronger time locality for performance evaluation. Figure 15 and Figure 16 show the throughput and the 99th percentile latency for each key-value stores system under the six workloads of YCSB. Calcspar can guarantee that the throughput is not lower than RocksDB in all cases, and the throughput can be improved under workload F. Throughput is not further improved due to the cloud storage IOPS budget constraints.

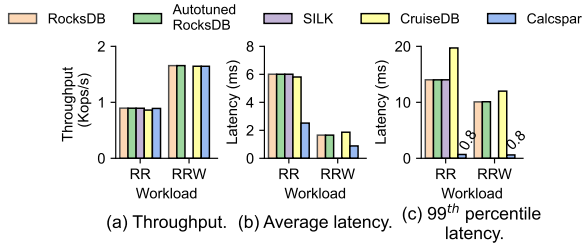

Figure 17: Evaluation results under Uniform workload. "RR" means random read, "RRW" means 50% random read and 50% random write.

Furthermore, the hot key of Zipfian distribution is scattered throughout the whole key space, resulting in underutilization of aggressive prefetching cache performance. Calcspar's main goal is optimizing latency, and as shown in Figure 16, the 99th percentile latency of Calcspar is reduced by 50% compared to other schemes. The best access latency is obtained because Calcspar is throttling the number of I/Os per second to access the cloud storage volume within a range of no more than paid IOPS. And cache redirection solves the queuing latency problem, so the latency per request is low. In other systems, requests beyond the paid IOPS will only to debit without strict limits, and the request latency will be significantly higher.

For uniform workloads. We further use YCSB to evaluate the performance of calcspar under a uniform workload. Figure 17 shows that although the fluctuation-aware caching is less efficient under Uniform load, Calcspar exhibits shorter latency due to its flexible I/O throttling. CruiseDB's adaptive access mechanism can also reduce the average latency, but its tail latency increases to 20ms.

5.3 Congestion Mitigation Effectiveness

Here, we investigate the effect of Calcspar on solving thread congestion. We first test the latency performance under different threads and then explore the effect of the time window allocation IOPS strategy.

Avoid multi-thread congestion. Figure 18 shows the average latency and 99th percentile tail latency of the experiment running the default Mixgraph load on io2 with 1k paid IOPS using different user threads. Calcspar can keep the average latency at 175μs, 99th percentile latency always around 500μs, and the other schemes keep increasing both the average latency and 99th percentile latency as the number of threads increases. On cloud drives, the 99th percentile latency growth reaches 60$\times$ under 20 threads. Because other Key-value stores limit bandwidth without restricting the granularity of I/O. Therefore, as the number of threads increases, congestion becomes increasingly severe. Calcspar utilizes the EBS rate-limiting model, and the Congestion-Aware IOPS allocating avoids requests queuing in the I/O domain queue.

IOPS allocation strategy evaluation. We compared Calcspar's time window policy allocate IOPS (TWA) with three other IOPS allocation schemes: contention IOPS without allocation (NA), static allocation of IOPS among three queues in the ratio of 6:3:1 (SA), and dynamic allocation of IOPS based on the usage of the highest priority queue (DA). Using mixgraph load, where 5% are read requests and 95% are write operations, guarantees enough flush and compaction operations with 10 user threads running. The evaluation results in Figure 19 show that the time window strategy has a good throughput and 99th percentile latency is reduced by 50% compared to the NA. The SA has the worst performance because of resource wastage. The DA can fully utilize IOPS and the average latency is the lowest, but the 99th percentile latency is higher than TWA because the requests in other queues will block user requests.

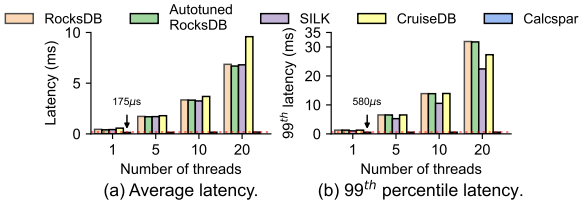

Figure 18: The latency with different user threads.

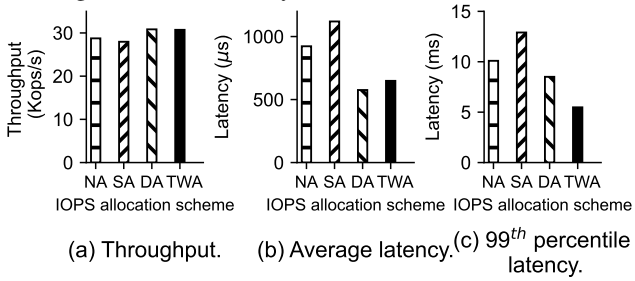

Figure 19: Performance of four IOPS allocation schemes.

5.4 Cache Effectiveness

Then, we evaluate the effect of Fluctuation-Aware Caching regarding the cache hit ratio, the impact of cache size and the corresponding read amplification.

Cache Hit Ratio. We compared Calcspar's fluctuation-aware cache (FA-Cache) with only passive cache (P-Cache) and only proactive prefetching cache (PP-Cache) under YCSB and Mixgraph workloads. Figure 20 shows that FA-cache has the highest hit ratio under both workloads. For YCSB load, the hotkeys are randomly distributed in the key space, so it is more suitable for P-cache with small prefetching. However,

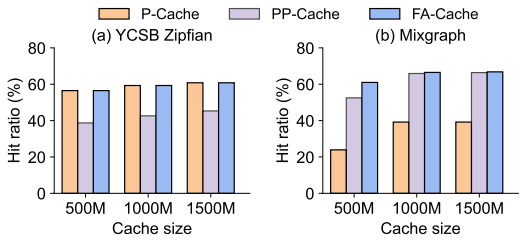

Figure 20: Hit ratios of different caching schemes under different workloads.

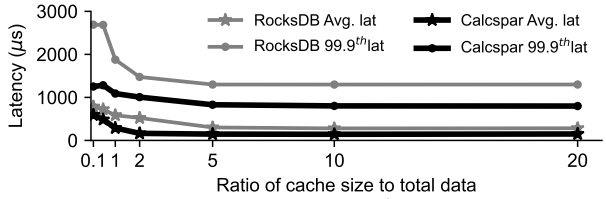

Figure 21: Average latency and 99.9th percentile latency of RocksDB and Calcspar with different cache sizes under Mixgraph load.

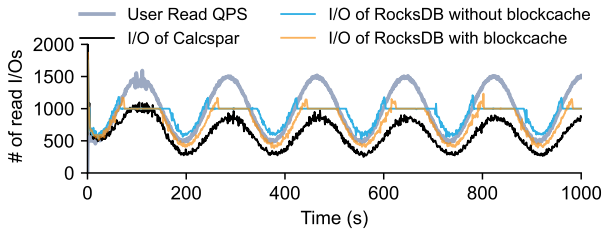

Figure 22: Read QPS and number of I/Os send to EBS.

under Mixgraph load, the hotkeys are relatively concentrated and more suitable for PP-cache. Calcspar's FA-cache combines these two advantages. Also, we can find less than 5% of data space can achieve up to 60% hit rate, which can increase the overall capacity of the system at a lower cost.

Impact of Cache Sizes. We increased the cache size from 0.1% to 20% of the total number to explore its impact on performance. Figure 21 shows that Calcspar outperforms RocksDB regarding latency at any cache size. With 1% cache size, Calcspar can reduce the average latency to below 200μs. RocksDB's block cache requires the cache size at least 5% of the total data to get an average latency close to 200μs, but the 99.9th percentile latency is still as high as 1200μs.

Read amplification. We count the number of read requests sent from the client side and the I/O accesses to the EBS to explore the effect of Calcspar in mitigating read amplification. We use Mixgraph default configuration for evaluation and compare it with RocksDB without cache and RocksDB with the same size (500MB) blockcache turned on. In Figure 22, the results show that Calcspar sends the least number of I/O requests to EBS with the same user requests, even during peak periods, because the fluctuation-aware caching can cache L0 and L1 in advance during low-load periods. RocksDB with blockcache enabled is limited by the IOPS budget during high load, and RocksDB without blockcache enabled sends more I/O to EBS during low load because of read amplification.

5.5 Impact of Opportunistic Compaction

Write performance. We compare the performance of Calcspar with the rest of the schemes under full write load. Figure 23(a) shows the throughput of 10 threads writing 100 million key-value pairs randomly (except for SILK single threads). Compared to RocksDB, Calcspar's write performance is only 1.2Both Autotuned RocksDB and CruiseDB allocate bandwidth to prioritize upper-level write operations, which improves performance. SILK can only write in a single thread,

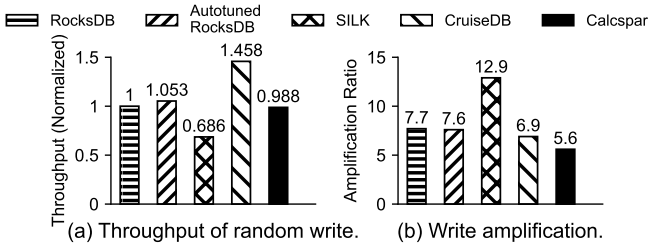

Figure 23: Write performance.

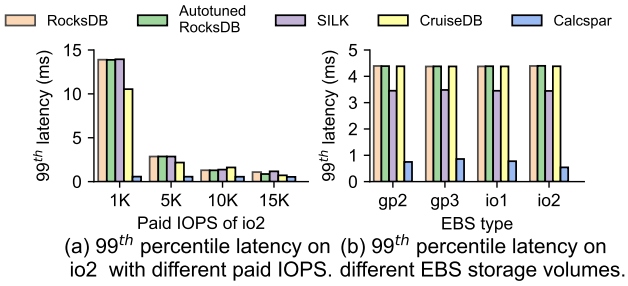

Figure 24: 99th percentile latency under Mixgraph.

so performance is poor.

Write amplification. Figure 23(b) shows that Calcspar reduces write amplification the most because Calcspar blocks L0 level to L1 compaction slightly. However, the write performance can not be improved because of IOPS allocation. CruiseDB removes the L0 level to reduce the write amplification. SILK prioritizes the execution of flush and L0 level compaction, resulting in frequent reads and writes of L1 level data, which in turn causes higher write amplification.

5.6 Sensitivity Analysis

Paid IOPS of EBS. We first evaluate five KV stores on io2 with different paid IOPS using Mixgraph to explore the impact of paid IOPS on performance, where Mixgraph uses the default read/write configuration and ensures that the average value of load fluctuations is equal to paid IOPS. Figure 24(a) shows that regardless of the paid IOPS of io2, Calcspar ensures the 99th percentile latency stays low. As the paid IOPS increases, the request latency under high pressure becomes progressively smaller, but at 15k paid IOPS, Calcspar's 99th percentile latency is still 24% ~50% less than other solutions.

Type of EBS. We use four types of EBS volumes with 3000 IOPS to explore the applicability of Calcspar on AWS EBS. Figure 24(b) shows that Calcspar exhibits the lowest 99th percentile latency on all four EBS compared to the remaining four options. As the performance of the storage volume gets better, the 99th percentile latency of Calcspar gets lower, e.g., io2 has a lower 99th percentile latency than gp3 by 200µs. In contrast, the other schemes have a higher latency or no change. The results fully illustrate that our scheme is suitable for various types of cloud block storage devices in AWS.

Workload pressure. We evaluate the pressure resistance by varying the workload intensity, which is the ratio of the average read requests of Mixgraph fluctuating load to paid

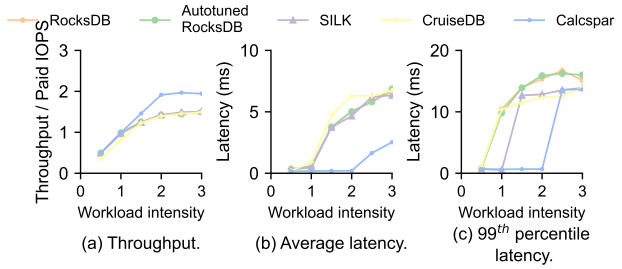

Figure 25: Performance at different workload intensities.

IOPS of io2. Figure 25(a) shows that Calcspar can handle up to twice the paid IOPS for read requests. Figure 25(b) shows that Calcspar can guarantee an average latency of 200μs even at twice the workload, while the rest of the solutions have higher latency when the read workload exceeds paid IOPS. The 99th percentile latency of Calcspar is still the smallest at high workloads in Figure 25(c). Overall, Calcspar has good pressure resistance and adaptability.

6 Related Work

Latency-aware Storage Stack. Many works focus on providing fast I/O services by optimizing the I/O stack [10] to exploit the μs-scale latency of storage devices. Blk-switch [18] brings network switch techniques into the block storage stack and proposes an I/O scheduler, thus solving the head-of-line blocking problem and achieving low tail latency. FastResponse [24] targets on *ultra-low latency* SSDs, and it coordinates the scheduling of different I/O levels to mitigate I/O interferences. PAIO [26] proposes an I/O optimization framework, enabling flexible I/O scheduling policies through I/O information propagation. However, these efforts focus on either how to fully explore the potential of multi-core/hardware resources or how to alleviate contentions between latency-sensitive and throughput-demanding applications. Calcspar targets LSM-Tree key-value stores over the cloud storage, and Calcspar addresses the I/O contentions between user read I/Os and LSM internal I/Os, achieving low latency.

LSM Store Compaction. Compaction I/O operations within LSM-Tree will compete with user read operations, resulting in long-tail latency. There are approaches to reduce data writes by delaying or merging some compaction actions [28, 30, 38]. Some studies reduce contention for storage devices by tuning and scheduling internal tasks [8, 9, 12]. Some adaptive compaction schemes have also been proposed for performance optimizing [15, 32]. However, these are all compaction optimizations targeting bandwidth-constrained SSDs. Optimizing compaction operation alone is not enough, as user-read requests already dominate the paid IOPS.

Software and Hardware Co-design for Latency. Hardware and software coordination can better reduce long-tail latency. For example, RStore [22] fully utilizes the advantages of multiple cores to reduce the tail latency of in-memory key-value stores. BCW [34] achieves low write latency on HDDs by reshaping patterns that utilize the HDD internal buffer. Vigil-KV [21] demonstrates the latency state of NVMe SSDs using a predictable latency mode interface and ensures controllable tail latency by scheduling compaction/flush operations and client requests.

Data Cache. Prefetching or caching frequently accessed data to a high-performance cache device can greatly improve read performance by reducing the number of slow I/O operations. Leaper [37] leverages machine learning methods to predict hot data and proactively prefetch them to the cache. AC-Key [35] aims at LSM cache mechanisms in the memory, hybrids different kinds of cache objects, and dynamically adjusts their sizes to improve cache efficiency. To reduce cache invalidation due to hotspot shifting and internal compaction, A parallel cache prefetching method [40] is proposed to prefetch the most valuable blocks into the cache by hotspot key-value pair tracking. Thus, read operations are not affected by the compression. LSM-tree [33] uses a compaction buffer to minimize these cache pollutions.

Reduce Storage Cost. Cloud storage users are often sensitive to storage costs, and they usually hybrids cloud storage volumes of different prices to cut the overall storage cost. Mutant [39] controls the overall storage cost by adaptively tuning the size of expensive high-performance storage volumes. PrismDB [29] pines hot data in the upper LSM levels to reduce storage costs. RocksMash [36] stores all metadata and frequently accessed data in local storage, while putting the rest in the cloud for cost efficiency. SA-LSM [41] uses survival analysis algorithms to predict hot and cold data at records granularity, and schedules compaction with external services to reduce costs by storing hot and cold data separately. Calcspar trades higher IOPS capabilities with the smaller memory or higher performance storage devices as cache, rather than simply purchasing more IOPS for cloud volumes.

7 Conclusions

This paper profoundly explores and models the latency mechanism of AWS's EBS cloud storage. Experimental analyses show that limited IOPS budgets in EBS contracts cause high latency spikes to endpoint users when requests exceed a threshold. We specifically investigate the LSM-tree based key-value store, which both amplifies external workload fluctuations and develops internal request congestion. We propose Calcspar, a contract-aware LSM store for cloud storage with reduced latency spikes. The fluctuation-aware caching strategy in Calcspar reduces 99th percentile latency by more than 66% under varying workload pressures without incurring noticeable caching costs. The congestion-aware IOPS allocation further avoids up to 60$\times$ tail latency spikes by assigning different priorities for internal operations and preventing thread I/O contentions. Our sensitivity study demonstrates that Calcspar is generalizable for different cloud storage volumes. Accordingly, Calcspar can be offered as a companion service to cloud storage providers, significantly balancing the performance and cost for endpoint users.

References

[1] LevelDB. https://github.com/google/leveldb.

[2] Amazon EBS Volume Modify Limitations. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/modify-volume-requirements.html, 2022.

[3] Amazon EBS Volume Pricing. https://aws.amazon.com/cn/ebs/pricing, 2022.

[4] Amazon EBS Volume Types. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ebs-volume-types.html, 2022.

[5] RocksDB. https://github.com/facebook/rocksdb, 2022.

[6] RocksDB Autotuned Rate Limiter. https://rocksdb.org/blog/2017/12/18/17-auto-tuned-rate-limiter, vistied Jan 2019.

[7] Amazon.2022. Cloud Storage. https://aws.amazon.com/what-is-cloud-storage/, 2022.

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

[9] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 753–766, 2019.

[10] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. Linux block io: Introducing multi-queue ssd access on multi-core systems. In *Proceedings of the 6th international systems and storage conference*, pages 1–10, 2013.

[11] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, 2020.

[12] Yunpeng Chai, Yanfeng Chai, Xin Wang, Haocheng Wei, and Yangyang Wang. Adaptive lower-level driven compaction to optimize lsm-tree key-value stores. *IEEE Transactions on Knowledge and Data Engineering*, 34(6):2595–2609, 2022.

[13] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26, 2008.

[14] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC 10)*, pages 143–154, 2010.

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

[16] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS operating systems review*, 41(6):205–220, 2007.

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

[18] Jaehyun Hwang, Midhul Vuppalapati, Simon Peter, and Rachit Agarwal. Rearchitecting linux storage stack for µs latency and high throughput. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 113–128. USENIX Association, July 2021.

[19] IDC, The digitization of the world from edge to core. https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf, 2022.

[20] Jens Axboe. Flexible I/O tester. https://github.com/axboe/fio, 2022.

[21] Miryeong Kwon, Seungjun Lee, Hyunkyu Choi, Jooyoung Hwang, and Myoungsoo Jung. Vigil-KV: Hardware-Software Co-Design to integrate strong latency determinism into Log-Structured merge Key-Value stores. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 755–772, Carlsbad, CA, July 2022. USENIX Association.

[22] Lucas Lersch, Ivan Schreter, Ismail Oukid, and Wolfgang Lehner. Enabling low tail latency on multicore key-value stores. *Proc. VLDB Endow.*, 13(7):1091–1104, mar 2020.

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

[24] Mingzhe Liu, Haikun Liu, Chencheng Ye, Xiaofei Liao, Hai Jin, Yu Zhang, Ran Zheng, and Liting Hu. Towards Low-Latency I/O Services for Mixed Workloads Using Ultra-Low Latency SSDs. In *Proceedings of the 36th ACM International Conference on Supercomputing (ICS 22)*, New York, NY, USA, 2022. Association for Computing Machinery.

[25] Kai Lu, Nannan Zhao, Jiguang Wan, Changhong Fei, Wei Zhao, and Tongliang Deng. Tridentkv: A read-optimized lsm-tree based KV store via adaptive indexing and space-efficient partitioning. *IEEE Trans. Parallel Distributed Syst.*, 33(8):1953–1966, 2022.

[26] Ricardo Macedo, Yusuke Tanimura, Jason Haga, Vijay Chidambaram, José Pereira, and João Paulo. PAIO: General, portable I/O optimizations with minor application modifications. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 413–428, Santa Clara, CA, February 2022. USENIX Association.

[27] Harchol-Balter Mor. *Performance modeling and design of computer systems*, volume 576. Cambridge University Press, 2013.

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

[29] Ashwini Raina, Asaf Cidon, Kyle Jamieson, and Michael J Freedman. Prismdb: Read-aware log-structured merge trees for heterogeneous storage. *arXiv e-prints*, pages arXiv–2008, 2020.

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

[31] Reseach and Markets. Cloud Storage Market. https://www.researchandmarkets.com/reports/4306260/cloud-storagemarket-forecasts-from-2017-to-2022, 2022.

[32] Nicholas Joseph Ruta. *CuttleTree: Adaptive tuning for optimized log-structured merge trees*. PhD thesis, 2017.

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

[34] Shucheng Wang, Ziyi Lu, Qiang Cao, Hong Jiang, Jie Yao, Yuanyuan Dong, and Puyuan Yang. BCW: Buffer-Controlled writes to HDDs for SSD-HDD hybrid storage server. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 253–266, Santa Clara, CA, February 2020. USENIX Association.

[35] Fenggang Wu, Ming-Hong Yang, Baoquan Zhang, and David H.C. Du. AC-Key: Adaptive caching for LSM-based Key-Value stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 603–615. USENIX Association, July 2020.

[36] Peng Xu, Nannan Zhao, Jiguang Wan, Wei Liu, Shuning Chen, Yuanhui Zhou, Hadeel Albahar, Hanyang Liu, Liu Tang, and Zhihu Tan. 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):1–26, 2022.

[37] Lei Yang, Hong Wu, Tieying Zhang, Xuntao Cheng, Feifei Li, Lei Zou, Yujie Wang, Rongyao Chen, Jianying Wang, and Gui Huang. Leaper: A learned prefetcher for cache invalidation in lsm-tree based storage engines. *Proc. VLDB Endow.*, 13(12):1976–1989, jul 2020.

[38] Ting Yao, Jiguang Wan, Ping Huang, Xubin He, Qingxin Gui, Fei Wu, and Changsheng Xie. A light-weight compaction tree to reduce i/o amplification toward efficient key-value stores. In *Proc. 33rd Int. Conf. Massive Storage Syst. Technol.(MSST)*, pages 1–13, 2017.

[39] Hobin Yoon, Juncheng Yang, Sveinn Fannar Kristjansson, Steinn E Sigurdarson, Ymir Vigfusson, and Ada Gavrilovska. Mutant: Balancing storage cost and latency in lsm-tree data stores. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC 18)*, pages 162–173, 2018.

[40] Shuo Zhang, Guangping Xu, YuLei Jia, Yanbing Xue, and Wenguang Zheng. Parallel cache prefetching for lsm-tree based store: From algorithm to evaluation. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 222–236. Springer, 2021.

[41] Teng Zhang, Jian Tan, Xin Cai, Jianying Wang, Feifei Li, and Jianling Sun. Sa-lsm: Optimize data layout for lsm-tree based storage using survival analysis. *Proc. VLDB Endow.*, 15(10):2161–2174, jun 2022.