Extending Delayed Fair Sharing: A Generalizable Framework for Multi-Resource Performance Isolation

James DeLoye
Tyler Griggs
Dev Bali
Wenjie Ma
Audrey Cheng
Jae Hong
Soujanya Ponnapalli
Natacha Crooks
Scott Shenker
Ion Stoica
Matei Zaharia
Joseph Gonzalez, Ed.

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2025-97
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2025/EECS-2025-97.html

May 16, 2025

Acknowledgement

Thank you to Professor Gonzalez for your recommendation and support, and to Professor Zaharia for inspiring this project. Thank you Tyler, for your mentorship and camaraderie both in the lab and in class. Thanks to Soujanya for your advice which helped immensely. To my family—Mom, Dad, and Tommy—your love and support made this possible. To my friends, thank you for keeping me grounded and motivated. And to everyone else I crossed paths with this year—thank you for being part of the journey.

Extending Delayed Fair Sharing: A Generalizable Framework for Multi-Resource Performance Isolation

by James DeLoye

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of Master of Science, Plan II.

Approval for the Report and Comprehensive Examination:

Professor Joseph Gonzalez
Research Advisor

May 15, 2025
(Date)

* * * * * * *

Professor Matei Zaharia
Second Reader

May 15, 2025
(Date)

Acknowledgements

To Professor Gonzalez, thank you for recommending me to this program and for always being responsive whenever I needed assistance. To Professor Zaharia, thank you for inspiring the subject of this project and suggesting I contribute to FairDB in the first place.

To Tyler, thank you for your constant mentorship, technical guidance, and camaraderie both in the lab and in class. To Soujanya, thank you for always offering a balanced and unbiased perspective when I sought your advice.

To my family—Mom, Dad, and Tommy—without your unwavering support, none of this would have been possible. Your encouragement means the world to me, and I love you more than anything.

To my friends—Alan, Alsyl, Cy, D'Angelo, Hailey, Hari, Imran, John, Karim, Noor, Parwiz, Rohit, Sabreen, and Tianchen—thank you all for significantly improving my mental health throughout this journey and providing the motivation to persevere. Your friendship means so much to me, and I deeply appreciate and love you all.

To anyone else with whom I interacted this past year but didn't explicitly mention—thank you for being a part of my journey.

Extending Delayed Fair Sharing: A Generalizable Framework for Multi-Resource Performance Isolation

James DeLoye,[1] Tyler Griggs,[1] Dev Bali,[1] Wenjie Ma,[1] Audrey Cheng,[1]
Jaewan Hong,[1] Soujanya Ponnapalli,[1] Natacha Crooks,[1] Scott Shenker,[1,2] Ion Stoica,[1] Matei Zaharia[1]

[1]UC Berkeley [2]ICSI

Abstract

Modern storage systems, often deployed to support multiple tenants, must provide performance isolation. Traditional approaches, like fair sharing resources, work poorly for these systems as storage resources exhibit *high preemption delays*. These delays force clients to wait to receive their share of resources, and lead to unacceptable spikes in tail latency.

Delayed Fair Sharing is a new fairness definition that explicitly bounds the delay a client incurs to receive its fair share of resources. DFS originally proposed policies that bound the preemption delays of commonly contended storage resources, such as write buffers, and read caches and implemented them in FairDB, an extension of RocksDB. In this work we attempt to extend Delayed Fair Sharing to other candidate resources, specifically the RocksDB write-ahead log (WAL) and flush threadpool, and implement these extensions in FairDB and evaluate their effectiveness.

FairDB ensures that clients receive their resource shares within a configurable time window (δ) and bounds the resource acquisition delays end-to-end. Here we show that this logic can be extend to more resources than those in the original work and effectively isolate clients from bursty or aggressive clients in the same database.

1 Introduction

Cloud computing and Software-as-a-Service (SaaS) environments frequently deploy storage systems (e.g., RocksDB [18], Cassandra [31], MySQL [36], PostgreSQL [2], along with commercial solutions [3, 13, 24, 39]) to handle multi-tenant workloads. Developers commonly consolidate data from multiple tenants into shared storage instances to achieve cost-effectiveness and higher resource utilization. This consolidation inevitably leads to tenants with diverse and fluctuating workloads competing for storage resources [11, 41, 52].

A critical challenge in these multi-tenant scenarios is ensuring *performance isolation*, meaning one tenant's workload should not degrade the performance experienced by others, irrespective of workload bursts or access pattern variations [11, 41, 52].

Common industry practices for achieving isolation involve imposing per-client rate limits or quotas, as seen in Amazon RDS [7] and Google Bigtable [23]. However, these methods face significant drawbacks. First, rate limits alone cannot fully guarantee isolation, especially when individual requests significantly vary in size or trigger resource-intensive system activities like data compaction (detailed in § 2). Second, these limits often require manual and careful tuning: overly conservative settings reduce resource utilization, whereas overly generous limits may lead to significant interference among tenants.

Fair sharing techniques, commonly utilized in domains such as networking [16], cluster scheduling [21, 46], and CPU scheduling [1, 48, 49], represent an alternative strategy. These methods guarantee performance isolation by allocating resources proportionally according to fairness metrics while redistributing idle resources to demanding clients, thus improving utilization.

Nevertheless, existing fair-sharing approaches are generally unsuitable for storage resources due to the assumption of negligible *preemption delay*—the speed at which resources can be redistributed among tenants [16, 20, 21, 46, 48, 49]. For instance, redistributing network bandwidth occurs swiftly (microseconds after packet transmission), but storage resources, like write buffers or read caches, involve significant preemption delays due to slower disk operations.

To address this challenge, *Delay Fair Sharing (DFS)* [26] introduces a fairness model explicitly designed for resources exhibiting high preemption delays. DFS ensures configurable bounds on the delays tenants experience when reclaiming their fair share of resources. Administrators can define a client's delay tolerance (δ), and DFS guarantees clients receive their fair share within this specified tolerance interval, irrespective of other clients' workloads.

Achieving strict DFS guarantees for latency-sensitive tenants (with near-zero delay tolerance) necessitates resource reservation. Intuitively, if the preemption delay of a resource surpasses a client's tolerance, a certain fraction of the resource must remain reserved and unallocated, even during periods of low demand. This reservation mitigates delays when demand suddenly increases.

Critically, DFS fairness can also *compose* across multiple resources, limiting a client's maximum aggregate delay to the sum of individual resource tolerances.

DFS identifies and defines essential metrics, such as the *peak burst threshold*—the maximum simultaneous demand

increase among tenants—and the resource *refill rate*, or how rapidly resources become available under worst-case scenarios. These metrics guide appropriate resource reservations, balancing isolation with resource utilization. Empirical evidence suggests most tenant demand spikes are limited in scale [47, 52], allowing minimal reservations and high overall utilization.

Building on Delayed Fair Sharing for storage resources, we investigate the application of DFS to two other common database resources — write-ahead logs and flush threadpools. Both resources are typically shared across all clients with little isolation. Flush threads in particular exhibit very high preemption delays – the smallest granularity that they can take action is that of a single entire flush, which may take hundreds to thousands of milliseconds. Fair sharing algorithms that assume negligible preemption delay will often fail to make guarantees about time to reach fair share due to the high preemption delay (finishing a flush) before moving threads to a new request.

We integrate our enhanced DFS policies into FairDB, our prototype built atop RocksDB [18] first built in the original DFS paper [26]. Traditional fair-sharing approaches within RocksDB demonstrate significant vulnerabilities, with single aggressive tenants causing latency spikes exceeding **10** for others. DFS demonstrates that FairDB, equipped with DFS, maintains isolation effectively within tenants' configured delay tolerances while sustaining high resource utilization (over **90%**). On workloads derived from production traces [47], FairDB achieves up to **9** lower tail latency with a modest trade-off in utilization compared to traditional fair sharing, and it significantly outperforms static resource partitioning methods in throughput (up to **38%**). Additionally, our extension of DFS to WALs and thread pools reduces latency spikes caused by WAL contention by over **5** and almost completely eliminates flush-thread-related delays for latency-sensitive tenants.

In summary, our key contributions are:

- We review Delayed Fair Sharing for storage resources with high preemption delays, which bounds client stalls to fair share and helps meet their explicitly-configured delay tolerance.
- We investigate policies to achieve δ-fairness on other system resources— write-ahead logs and non-preemptible threadpools—by considering the specific mechanics of each resource and the measurable properties of a workload to maximize resource utilization and meet clients' delay tolerance.
- We extend FairDB's δ-fair policies on top of RocksDB and show that it provides strong, configurable performance isolation with high resource utilization.

2 Background and Motivation

Multi-tenant storage systems aggregate physical resources to serve multiple clients concurrently, improving efficiency and utilization in modern cloud workloads [4–6]. However, one of the core challenges in these environments is maintaining strong performance isolation: the workload of one tenant must not degrade the experience of others, even when workloads are bursty or highly variable [33].

A commonly explored strategy for addressing this challenge is fair sharing. In this approach, each tenant is allocated a proportional share of resources, with excess capacity opportunistically redistributed to tenants with higher demand. This method offers a balance between isolation and utilization. Yet, as discussed in prior work [26], storage system resources such as write buffers and read caches exhibit high preemption delays—delays that arise because freeing and reallocating these resources often requires slow disk operations. These delays violate the assumptions of traditional fair sharing models, which rely on the ability to rapidly reassign resources among tenants.

Moreover, this issue is not limited to caches and buffers. Other storage-related resources—such as non-preemptible thread pools—can also incur high preemption delays. For instance, when threads are bound to long-running, uninterruptible tasks (like large flush operations), switching to serve another tenant introduces significant latency overheads. Our work builds upon the insights from DFS [26] to explore how fairness can be maintained across such high-delay resources.

2.1 Overview of Traditional Fair Sharing

Fair sharing is a standard approach for achieving performance isolation and high resource utilization. It provides two key properties:

Share guarantee: Each of n clients receive at least $1/n^{th}$ of the shared resource, regardless of other clients' demands.

Pareto efficiency: The resource share of one client cannot be increased without decreasing the share of another client.

Fair sharing schedulers aim to dynamically maintain both isolation and utilization by reallocating resources in response to changes in client demand. When a client that is currently receiving less than its fair share increases its demand, the scheduler shifts resources away from over-allocated clients. Similarly, when a client reduces its usage, those resources are redistributed to others who need them. This reactive, demand-driven mechanism helps balance competing workloads efficiently.

Such algorithms have been widely adopted in systems where resources can be reallocated quickly, such as CPU scheduling [20, 21], job scheduling in data centers [21, 46], and network bandwidth allocation [16, 28, 37]. In these domains, the overhead of shifting resources between clients

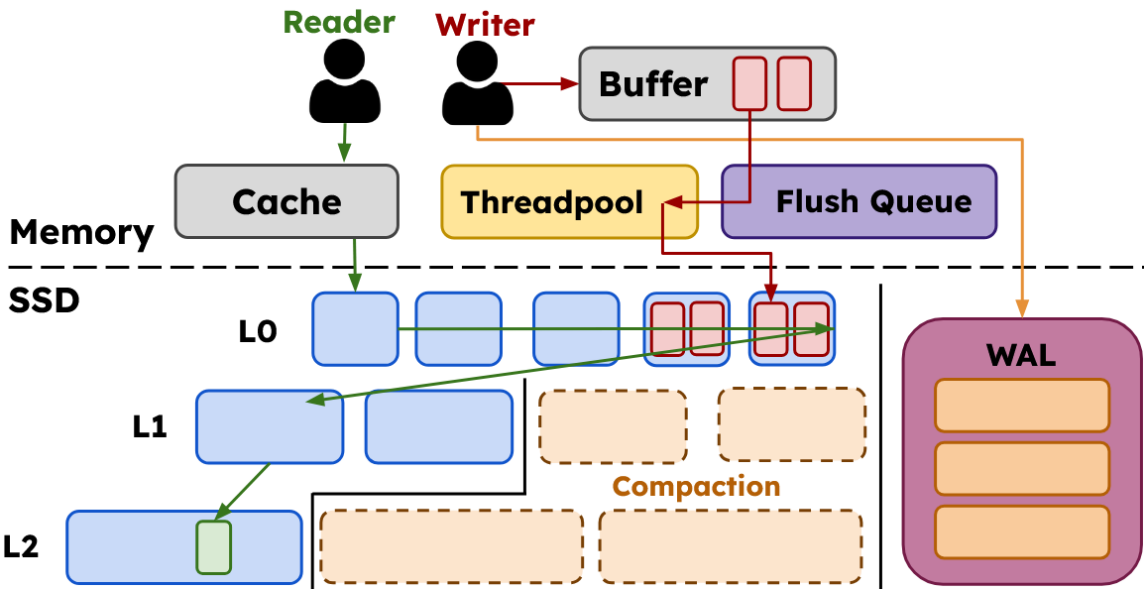


Figure 1. **RocksDB Architecture Overview.**

is minimal—resources can be redistributed in microseconds or milliseconds—so clients rarely experience delays in accessing their fair share. This rapid adaptability is a key reason why traditional fair sharing techniques deliver strong performance isolation in those settings.

2.2 High Preemption Delays in Resources

The original DFS paper [26] notes that system resources—such as write buffers and read caches—exhibit high preemption delays *i.e.,* the time required to redistribute resources among clients. Consider write buffers that hold recent writes in memory: reclaiming a client's share requires flushing dirty pages to disk, which is approximately 200× slower than an in-memory write. This results in significant delays before the client receives its fair share. Similarly, redistributing read cache shares requires fetching previously evicted cache pages from disk, introducing similarly high preemption delays. Thus, storage system resources are in contrast to other resources can be reallocated with negligible delay.

Traditional fair sharing mechanisms applied to storage resources typically provide only eventual fairness: over time, allocations converge to fair shares, but only after clients endure potentially long preemption delays. During this convergence, the intended benefits of fair sharing—namely performance isolation and efficient utilization—are postponed. In practice, these delays can cause significant degradation in client experience. For example, write operations may stall while waiting for buffer space to be flushed to disk, and read operations may block until evicted cache pages are reloaded—both resulting in elevated tail latencies.

High preemption delays therefore compromise performance isolation by allowing one client's resource spike to directly increase another client's response time. This is particularly concerning in storage systems, where tail latency is often a critical metric for user-facing applications. Despite this, the implications of high preemption delays on performance isolation in multi-tenant storage contexts remain insufficiently explored in prior work.

A similar problem arises in thread scheduling. Classic schedulers are designed for settings with highly preemptible jobs and fine-grained time slices (on the order of microseconds), allowing fast redistribution of compute resources. However, threadpools that handle long-running, non-preemptible tasks—such as block flushes or compactions—violate this assumption. In these environments, the smallest unit of work may still take tens or hundreds of milliseconds, introducing significant delays before resources can be reassigned. As with storage buffers and caches, traditional fair sharing will eventually rebalance the system, but only after introducing substantial latency overheads, again undermining isolation guarantees for latency-sensitive clients.

2.3 Empirical Study on Preemption Delays

The original DFS paper [26] evaluates preemption delays in RocksDB, a widely used write-optimized key-value store deployed in SaaS backends and cloud-native databases [18].

Experiment setup: 16 clients run issuing steady read and write operations. To study write buffer behavior, a bursty client is introduced that periodically floods the system with writes, saturating the buffer. For read cache behavior, one client goes idle and later resumes activity, attempting to access data that was cached before its idle period.

RocksDB	Latency (p99 in ms)	Throughput (GB)
Rate limits	977	3.12
Resource quotas	**<1**	3.10
Fair Sharing	943	**3.92**

***Table 1.* Performance Interference in RocksDB.** The DFS authors compare three standard approaches for performance isolation with clients running a write-only, YCSB Load-A workload.

RocksDB Configuration. Multiple tenants share the RocksDB store by storing data in logically separate LSM trees, termed column families. The Log-Structured Merge (LSM) tree data structure in RocksDB is illustrated in Figure 1. Writes are first written sequentially to an on-disk write-ahead-log (WAL) which ensures that system state can be restored in the event of a crash. In addition, writes are initially stored in an in-memory write buffer, termed memtable. These memtables are then queued for flushing and popped off by a flush threadpool. They are then sorted by keys and flushed to disk as an immutable log file, called an SSTable. Over time, SSTables are compacted by another compaction threadpool into larger SSTables and moved to higher logical levels (L0→L1→L2, etc.). RocksDB also maintains an in-memory cache to accelerate read operations, and cache misses are served from the first on-disk tree level that holds the desired record. By default, RocksDB provides each client logically separate write buffers with configurable per-client and global limits on available buffer space. The read cache can be optionally shared or separated per-client, and likewise supports per-client and global capacity limits.

The DFS authors compare three standard practices: (1) Rate limits: which throttle request rates per client; (2) Resource quotas: which partition resources equally across clients; and (3) Fair sharing: which dynamically redistributes resources based on clients demand to preserve fair sharing (§2.1). These represent common strategies for balancing isolation and utilization in multi-tenant storage.

Results. Results are summarized in Table 1. Rate limits control write buffer usage by reducing demand spikes for each client, providing good isolation. However, when clients operate below their limits, unused buffer capacity reduces overall utilization. Per-client resource quotas offer strong isolation by partitioning resources across clients. But when clients underutilize their share, reserved resources remain idle, leading to poor utilization. Traditional fair sharing allocates each client a fair share of the resource, and redistributes unused capacity to clients with higher demand. We can observe that the high preemption delays undermine performance isolation. A bursty client that fills its entire write buffer, forces others to wait for flushes before receiving their fair share. These delays result in significant performance degradation, with tail latencies increasing by orders of magnitude.

Summary and takeaways. Conventional strategies for resource management—such as rate limiting, fixed quotas, and traditional fair sharing—fall short in meeting the dual goals of strong performance isolation and high resource utilization in multi-tenant storage systems. In systems like RocksDB, write buffers and read caches are common points of contention, and their behavior has a direct impact on the tail latencies experienced by latency-sensitive clients.

Rate limiting offers a blunt control mechanism, which often fails during traffic bursts when interference is most harmful. Resource quotas can isolate tenants but require static partitioning, which frequently leads to underutilization. Fair sharing addresses utilization more effectively but fails to account for the high preemption delays of storage resources. As a result, clients may suffer significant latency spikes when trying to reclaim their fair share under contention.

These shortcomings highlight a broader issue: existing approaches largely ignore the unique temporal dynamics of storage resource preemption. Under bursty workloads, these delays can be exacerbated, and without a model that adapts to them, performance isolation remains elusive. This motivates the need for a new model—one that explicitly incorporates resource-specific preemption delays while still preserving fairness and efficiency.

3 Delayed Fair Sharing

From the original paper, Delayed Fair Sharing is defined as a new fairness definition to effectively share resources with high preemption delays among multiple clients. Specifically, Delayed Fair Sharing provides a configurable bound on the delay to acquire fair shares of a resource. Specifically, the authors define:

δ-fairness: When a client demands resource shares at time t, it must receive these shares, up to its fair share, by $t+\delta$.

The concept of δ-fairness provides a per-client guarantee on delay: specifically, it bounds the time a client may wait to receive its fair share of a contended resource to at most δ, regardless of other clients' demands. This delay-bound fairness model allows system administrators to explicitly configure δ on a per-client basis, aligning resource allocation with application-specific latency requirements. Clients with tight latency constraints can be assigned a small δ (approaching zero), effectively ensuring near-immediate access to their fair share. In contrast, clients with more relaxed requirements can tolerate larger delays and thus benefit from higher resource flexibility or efficiency.

This delay is not necessarily interpreted as instantaneous acquisition of all resources at a single point in time; rather, it bounds the time until the system makes sufficient progress to provide the client's fair share—whether delivered incrementally or in bursts. The scheduling policy does not

assume atomic or synchronous allocation, allowing it to adapt to the dynamics of each resource's behavior.

Importantly, the δ parameter is distinct from weights used in weighted fair sharing. While weights define the proportion of the total resource each client is entitled to over time, δ defines the maximum delay a client is willing to tolerate to obtain that entitlement.

Crucially, δ-fairness also composes naturally in multi-resource settings. If a client simultaneously requests multiple resources, each with its own δ-bound policy, the total delay the client experiences is bounded by the sum of these individual δ values—though in many cases, parallel acquisition can result in a lower effective delay. This composability enables predictable end-to-end performance even in complex, multi-resource systems.

3.1 Reduced Delays vs Resource Utilization

To provide δ-fairness for latency-sensitive clients sharing resources with high preemption delays, systems must rely on resource reservations—introducing an inherent trade-off between minimizing client delays and maximizing overall resource utilization. When the time required to reassign a resource exceeds a client's delay tolerance (δ), the only way to meet that guarantee is to proactively reserve a portion of the resource exclusively for that client.

In practical terms, this means setting aside a baseline allocation of the resource that remains unavailable to other clients, even when unused. These reserved units ensure that latency-sensitive clients can access at least part of their fair share without waiting for others to release or flush the resource. This reservation reduces the burden of reclaiming resources under pressure and helps bound the client's worst-case delay.

At one extreme, static partitioning—where each client has a fixed resource quota—achieves δ-fairness with δ set to zero. In this case, clients always retain access to their full share and experience no delay when demand increases. However, this comes at the cost of poor utilization: unused allocations remain idle even when other clients could benefit from them.

As δ increases, the system can relax its reservation requirements. Clients that tolerate longer delays require smaller reserved shares, allowing more of the resource pool to be dynamically reallocated based on demand. This captures the core tension in managing resources with high preemption delays: strict performance isolation requires holding back capacity, while high utilization favors sharing. The δ parameter allows system designers to navigate this trade-off in a principled and configurable way.

3.2 Relationship to fairness properties

Two key properties desirable in resource fair sharing are *share guarantee* and *Pareto efficiency*. The share guarantee ensures each of n clients receives at least $\frac{1}{n}$ fraction of a resource, or $\frac{w_i}{\sum_{j \in W} w_j}$ of a resource in weighted fair sharing. δ-fairness extends this guarantee by explicitly limiting the delay to receive the shares within a configured bound.

For resources with high preemption delays, the goal of δ-fairness fundamentally conflicts with the principle of Pareto efficiency. Pareto efficiency demands that any available resources be fully redistributed to maximize overall utility—ensuring that no client's allocation can be improved without reducing another's. However, δ-fairness often requires holding back a portion of the resource in the form of reservations to meet strict delay guarantees. These reserved units may remain unused temporarily, even when other clients could benefit from them, in order to protect latency-sensitive clients from future contention.

This tension highlights a key trade-off: optimizing for maximum resource efficiency may compromise performance isolation, and vice versa. In systems where tail latency and client-specific delay bounds are critical, δ-fairness deliberately prioritizes predictable access over immediate throughput gains. As a result, some inefficiency is accepted to ensure stronger, more configurable isolation guarantees.

4 δ-Fair Policies for the WAL and Flush Thread Pool

In addition to the δ-fair policies for RocksDB's write buffers and caches detailed in the orginal DFS paper, we seek to extend RAD fairness to two further shared resources: the write-ahead log (WAL) and the thread pool that flushes memtables to on-disk SSTables. Like buffers and caches, WALs are ubiquitous in storage systems because they provide consistency and durability; unlike buffers, they seldom block writes directly, but are constrained by practical size limits. When the WAL reaches its configured maximum, even a single slow client can prevent old WAL files from being reclaimed, delaying all writers. Thread pools are another common source of interference, particularly when their tasks are non-preemptible. We first explain how these resources differ from buffers and caches, why they cause interference, and how δ-fair policies must adapt. We then propose a δ-fair policy for the flush thread pool and argue that no such policy is needed for the WAL. Finally, we discuss how these resources compose with others in the overall latency budget.

4.1 δ-Fairness in the WAL

4.1.1 WAL-Induced Interference Every write in RocksDB is appended to the shared on-disk WAL before it is inserted into the in-memory memtable. The WAL guarantees that committed transactions survive crashes until the corresponding memtable is flushed. A new WAL file is created whenever a memtable is flushed. All subsequent writes are logged to this new file, but the old WAL file remains on disk until *every* client that wrote to it has had its

data flushed. Moreover, all newer WAL files must also stay intact if any *older* file still contains unflushed data, because crash recovery replays the log sequentially.

Consequently, if all clients write quickly enough to fill and flush their memtables regularly, WAL files are deleted promptly and the log stays below its size cap. If, however, one or more clients write so slowly that their memtable remains unflushed for an extended period, the WAL can grow to its maximum size. When that happens, all clients with unflushed data in the oldest WAL file are forced to flush, producing a burst of flushes equal to the number of affected clients.

This forced flush does not itself stall writers—they may continue writing into free memtables—but a writer that fills *all* of its memtables while the flush backlog is draining will be blocked until at least one flush completes.

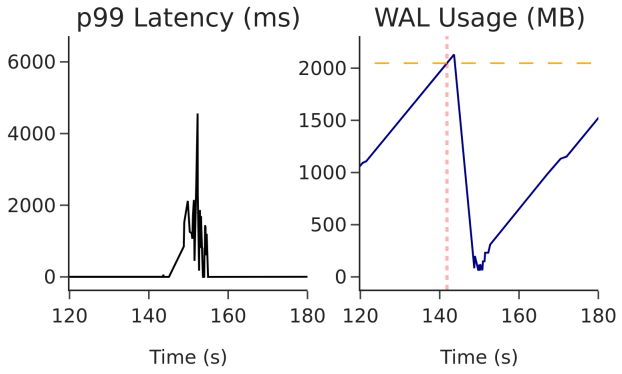


Figure 2. Latency spike caused by a WAL-triggered flush burst. Vertical red lines represent WAL flush triggers, horizontal line is WAL size limit.

4.2 Why Reservations Are Unnecessary in the WAL

Because the WAL never blocks writes as long as a client still has a free memtable, embedding a fairness mechanism in the WAL yields little benefit. The real bottleneck is the flush threadpool, which determines how quickly both memtables and WAL files are reclaimed. Fairness should therefore be enforced at the thread-pool level, where *all* flushes can be regulated, not only those triggered by a full WAL.

Other mitigations, such as predictive or piecewise flushing of data that is anchoring the WAL, may be effective at limiting WAL-mediated latency. However, in the worst case, this methodology still does not give guarantees about expected latencies that other clients may experience as a result of these WAL flushes. As a result, the only way to provide guarantees towards WAL-mediated latency would be to insert blocking behavior and fairly share the WAL or implement fair sharing the threadpool actually managing the WAL-triggered flushes.

4.3 δ-Fairness in the Flush Thread Pool

4.3.1 Flush Thread Pool Interference As illustrated in Figure 1, RocksDB flushes memtables to disk using a dedicated flush thread pool.[1] Threads dequeue memtables in FIFO order and execute each flush atomically: once a thread starts flushing a memtable, it runs to completion.

Thread pools are normally shareable with classical fair schedulers [1] because threads can be pre-empted cheaply. This is *not* true for flush tasks: a thread must run for the entire flush duration T_{flush} before it can be reassigned. If all M threads are busy when a client requests a flush, the client waits at least T_{flush}; if $n-1$ other flushes are queued ahead of it, the wait can grow to $\left\lfloor \frac{n-1}{M} \right\rfloor T_{\text{flush}}$.

Figure 2 shows an example of this in action. There are 16 clients in the database. 15 of them write only occasionally (every 50 seconds or so), while one writes constantly at a high rate. This could very feasibly happen in a real-world scenario: imagine 15 human users occasionally using a database while an automated data ingest continuously writes. If the occasional users don't write enough data to overflow a write buffer, then the data that they wrote in effect "anchors" the WAL, resulting in its size to continually grow as long as that data goes unflushed.

In our example, eventually a full WAL induces a cascade of 15 flushes, and the client waits up to $\left\lfloor \frac{15}{M} \right\rfloor T_{\text{flush}}$ before its turn in the FIFO flush queue. Adding more threads rarely helps under full utilization: the OS scheduler may still execute long flushes in FIFO order, preserving the worst-case total delay. While the queuing delay will be low, the time spent will be transferred over to the delay of the actual operation as the OS thread scheduler intersperses operations between scheduled threads.

Conventional fair schedulers for the thread pool behave similarly. When the WAL forces several clients to flush, those clients have accumulated little recent service and thus receive high priority, again delaying the steady writer.

By contrast, if the steady writer is given priority (Figure 3), the spike disappears. Indeed, we see a reduction of peak latency by around 5×. This observation motivates a δ-fair scheduler that bounds per-client flush delay by making a reservation. Also note that the WAL continues growing while the other flushes are happening – because the WAL-triggered flushes have not yet completed, the heavy writer can continue writing to the database. This has the beneficial side effect of increasing total throughput. In this process, because the WAL is still above the maximum size, it may call to trigger more flushes, though since the other clients are already in the process of having their data flushed, this call is largely idempotent.

[1] If the compaction pool is idle, its threads may flush as well, but in long-lived workloads the compaction backlog generally keeps the two pools separate.

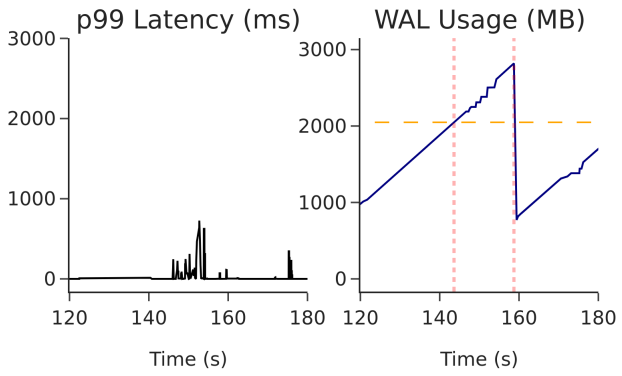


Figure 3. A priority scheduler eliminates WAL-induced latency spikes. Vertical red lines represent WAL flush triggers, horizontal line is WAL size limit.

4.3.2 δ-Fair Thread-Pool Scheduling We derive the minimal reservations needed to guarantee δ delay (§4.3.2.1) and then translate these reservations into an allocation policy (§4.3.2.2).

4.3.2.1 Minimal Thread-Width Reservations Let the flush thread pool contain M threads, yielding M thread-seconds/s of *thread-width*. Client i's fair share is $f_i = Mw_i/\sum_{j\in W} w_j$, where w_i is its weight. Flush tasks refill thread-width at M thread-seconds/s^2 (i.e. M thread-seconds/s of thread-width replenish every second). As a result, we can find that the delay in the time it takes for a client to receive its fair share of thread-width becomes:

$$T_{\text{queue}} = \frac{\lceil f_i \rceil}{M}. \tag{1}$$

Because flushes are single-threaded, the smallest reservable unit is one thread. Reserving $\lceil f_i \rceil$ threads for client i guarantees that the deviation between its received service and its entitlement stays within δ:

$$\frac{\lceil f_i \rceil - \lceil \rho_{\text{threadpool}}, i \rceil}{M} \le \delta. \tag{2}$$

When k clients simultaneously demand their fair share, the potential delay of the operation becomes:

$$T_{\text{multi-queue}} = \frac{\lceil kf \rceil}{M}, \tag{3}$$

This can be on the order of hundreds to thousands of milliseconds! As a result we choose the minimal $\rho_{\text{threadpool}}$ satisfying:

$$\frac{\lceil kf \rceil - \lceil \rho_{\text{threadpool}} \rceil}{M} \le \delta. \tag{4}$$

4.3.2.2 Thread-Width Allocation Policy Define U_i as client i's current thread usage. The global pool size is

$$G = M - \rho_{\text{threadpool}} - \sum_{i=1}^{n} U_i. \tag{5}$$

When a steady client queues a flush:

- If $U_i < f_i$, it first draws from the reserved thread(s).
- Otherwise it draws from the global pool.

If neither pool has threads available, the flush remains queued. As threads complete, they replenish the reserved pool first, then the global pool. Queued clients are admitted in increasing order of U_i/f_i; ties break by least-recent service. This is visualized in Figure 4.

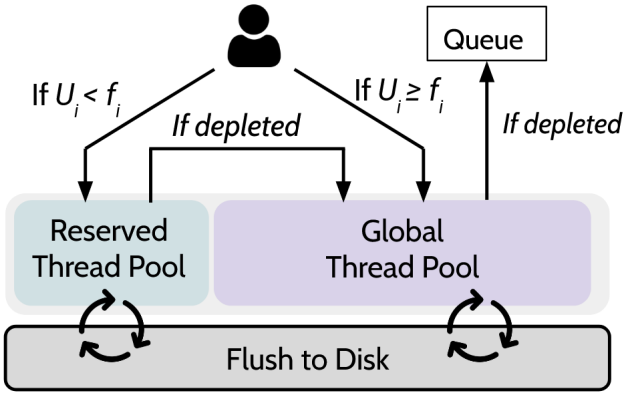


Figure 4. Design of the δ-fair flush thread-pool manager.

4.3.3 Flush Thread Pool Summary The proposed δ-fair scheduler lets each client reclaim its entitled share of the flush thread pool within δ. Although the pool is a CPU resource rather than pure storage, reserving thread-width yields the same bounded-delay guarantee. In evaluation (§6) we show that this policy can in some cases completely eliminate queuing related latency spikes with little penalty to aggregate throughput.

4.3.4 Cross-Resource δ-Fairness With no thread-pool contention, write latency is bounded by δ_{buffer}. Under contention, an additional δ_{queue} may be incurred, when the flush is forced to wait in the threadpool queue, so

$$\delta_{\text{write}} = \delta_{\text{buffer}} + \delta_{\text{queue}}.$$

For a worst-case serial read-then-write,

$$\delta_{\text{total}} = \delta_{\text{cache}} + \delta_{\text{buffer}} + \delta_{\text{queue}},$$

while parallel acquisition yields

$$\delta_{\text{total}} = \max\left(\delta_{\text{cache}}, \delta_{\text{buffer}} + \delta_{\text{threadpool}}\right).$$

Thus δ-fair guarantees compose naturally across resources: by configuring each resource's delay bound, we can bound the latency of an entire operation.

5 FairDB Implementation

FairDB is implemented as an extension of RocksDB. The modifications within RocksDB fall into two categories: (1) implementing policies that adhere to δ-fairness (§??), and (2) enabling per-client resource accounting. Prior DFS modifications are primarily in the write buffer management, block cache, I/O rate limiter, and flush queue components.

The original authors modify RocksDB's existing managers to implement the per-resource δ-adhering scheduling policies rather than implementing a single, centralized resource scheduler.

To implement δ-fair policies and enforce per-client usage, resource usage is traced back to clients. The authors assign each client an ID and propagate the client ID through RocksDB's request handling path, to ensure that background tasks (compaction and flush) are also accounted with the correct client.

RocksDB's resource managers for write buffers, cache, and I/O use this ID to maintain per-client usage statistics.

FairDB extends the built-in I/O rate limiter that manages the read and write I/O rates separately. This is built on to implement a token-based fair I/O scheduler inspired by Gimbal [34].

Each client is guaranteed a base allocation of write bandwidth (its fair share) and unused capacity is redistributed to clients with higher demand, ensuring high utilization.

The Write Buffer Manager (WBM) is also modified, which intercepts requests prior to buffering them, to track the aggregate memory usage across all column families. The δ-fair write buffer policy is then implemented and enforced by preventing eviction for clients under their reserved share.

The WBM object tracks statistics: (a) per-client write buffer usage, (b) global pool availability, (c) reserved pool availability, and (d) currently queued write requests, and handles a write request (as detailed in §??).

FairDB builds on the sharded LRU Cache Manager (CM) to manage the read cache. It extends the CM to track cache usage per client and maintain per-client reservation sizes. In FairDB, clients share a single logical cache, and pool unused capacity for high utilization. The CM dynamically updates reserved shares. A read request that is a cache hit only updates LRU statistics and does not pass through CM. Instead, the CM intercepts reads when they bring pages in from the disk. When a page is being added to the cache, the original authors identify the shard it belongs to and navigate the LRU list to find the first block belonging to a client that is above its reservation size, as detailed in (§??).

To manage thread scheduling, we modify the machinery that manages flushing to have per-client accounting of the thread-width being used. To make scheduling decisions, we replace the FIFO flush queue with two custom priority queues. If a flush request meets the requirements for the reserved pool, then it is pushed to the reserved priority queue. If not, then it goes to the global priority queue. Likewise, we also implement two separate threadpools, one for the reserved priority queue and one for the global priority queue. The reserved thread pool exclusively services the reserved priority queue while the global thread pool will serve both queues in accordance with the logic in §4.3.2.2. Each entry into the flushing queues associates the request with its client ID, fair-share, and current threadpool usage. As a result, all δ bounds will be honored.

Compactions consume significant I/O resources for reading, merging, and then writing large files to disk. They interfere with the I/O bandwidth otherwise used to reclaim read cache or write buffer resources promptly. To prevent compactions from disrupting client performance, the original authors follow the approach adopted by production LSM-based KV stores [13] and set aside approximately 30% of the total I/O bandwidth available for compactions. This dedicated allocation ensures that compactions proceed at a steady, controlled pace without monopolizing I/O resources. Within the dedicated compaction bandwidth, the same fair I/O scheduling policies are applied.

The RocksDB global stall triggers are also modified to apply on a per-tenant basis. The orignal authors associate stall trigger thresholds with the client that generates them. When a stall threshold is exceeded, only that client's requests are throttled, leaving others unaffected. In a FairDB, a single bursty client cannot degrade system-wide performance fairness.

6 Evaluation

We evaluate the benefits of extending δ-fairness in FairDB.

Experimental Setup. All experiments are run on a Debian `n2-standard-96` VM (96 vCPUs, 48 physical cores, 384 GB RAM) on GCP, with 16×375 GB NVMe local SSDs (6,240 MB/s read, 3,120 MB/s write). The storage system and the workload generation are pinned to separate NUMA nodes.

Baselines. FairDB is originally compared against two baselines. The primary baseline, RocksDB-FS with instantaneous fair sharing, represents the latest research on fair resource allocation. To reflect common industry practices, we implement another baseline, RocksDB-Qts with static per-client resource quotas. These two baselines capture common approaches and represent a broad range in the trade off of isolation and utilization.

RocksDB Configuration. We follow the official RocksDB tuning guide for configuration [19]. To limit performance interference, we modify global write stalls to affect only the column family that triggers them across all baselines.

Client workload configurations and are described within each experiment.

Workloads. We use the Yahoo Cloud Serving Benchmark (YCSB) suite to generate client workloads [15]. Clients run in an open-loop setting, issuing requests into per-client queues which are then serviced by RocksDB/FairDB worker threads. Request rates, access patterns, and working set sizes are detailed in each experiment. The authors create multi-tenant workloads for the experiments after analyzing caching traces from Twitter and Snowflake; we describe their generation later (§6.1.4) and plan to open-source these workloads.

First, we review the results from the original DFS. Per-resource δ-fair policies from are evaluated and we (§6.1.1) and discuss their composability (§6.1.2). The adaptivity of δ-fairness to applications with varying latency sensitivity and workloads is also shown (§6.1.3). Lastly the end-to-end benefits of FairDB with other RocksDB baselines is evaluated (§6.1.4). We then evaluate the extension of DFS to the RocksDB flush threadpool through our implementation of δ-fair sharing in FairDB.

6.1 Delayed Fair Sharing Results

6.1.1 Microbenchmarks Microbenchmarks isolate and evaluate the effectiveness of FairDB's δ-fair policies for individual resources: I/O bandwidth, write buffers, and read caches.

!

6.1.1.1 I/O Bandwidth. Fair sharing of I/O bandwidth is crucial as contention significantly exacerbates latency for resource-sensitive clients. Aggressive clients, in particular, can monopolize resources, causing substantial spikes in latency for others. The token-based fair-sharing mechanism effectively eliminates significant tail latency spikes due to contention, owing to negligible I/O preemption overhead (<1ms). In all subsequent experiments, fair I/O sharing is consistently applied across all clients and baselines, ensuring this resource is not a limiting factor in evaluating buffer or cache policies.

6.1.1.2 Write Buffer The client reclaiming its fair share experience about **30× higher tail latency** despite fair sharing resources in RocksDB. Writers are assumed to instantaneously release their buffer pages for such clients. FairDB improves this and achieves **1.8× lower tail latency spikes**, as seen in Figure 5. This improvement is configurable: here, FairDB is configured to satisfy clients' delay tolerance (δ) of 350ms; FairDB meets this deadline effectively and achieves higher buffer utilization relative to RocksDB-Qts, as shown in Figure 5. FairDB reserves resources in the system based on clients' configured latency sensitivity. It effectively supports δ =0ms for strict latency sensitivity, δ =inf for high delay tolerance, and δ =350ms for moderate tolerance, as shown in Figure 5. Other values for δ are explored later (§6.1.3).

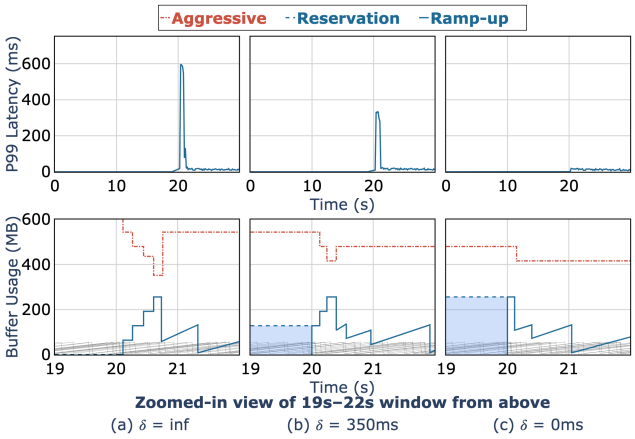


Figure 5. Write buffer microbenchmarks. The authors show the p99 latency and buffer usage of the ramp up client (blue), aggressive (red), and steady clients (grey). FairDB has minimal reservations to effectively meet ramp-up clients' delay tolerance (here, δ = 350ms)

The original authors analyze write buffer management under intense competition. Experiments use 16 writing clients: 12 steady (50 MB/s each), 2 aggressive (192 MB/s each), and 2 ramp-up clients initially writing large batches (128 MB each). Total write demand matches the available bandwidth (980 MB/s), fully saturating system resources.

Figure 5 from the original Delayed Fair Sharing illustrates the dramatic reduction of tail latency spikes under FairDB 's approach compared to RocksDB-FS. With RocksDB-FS, ramp-up clients encounter approximately 30× higher tail latency due to aggressive clients' delayed buffer releases. In contrast, FairDB configured with moderate delay tolerance (δ =350ms) significantly mitigates these spikes, achieving a 1.8× reduction in latency compared to RocksDB-FS.

FairDB achieves this reduction by reserving buffer space according to clients' specified latency sensitivity. Under moderate delay tolerance, the buffer reservation corresponds to approximately 6% of total capacity (128 MB of 2 GB). Stricter delay tolerances (δ =0ms) require larger reservations (up to 12.5%), ensuring near-instantaneous access, whereas looser tolerances dramatically reduce reservation size, highlighting the flexibility and configurability of FairDB's reservation policy.

6.1.1.3 Read Cache For cache evaluations, the authors conducted tests with 32 readers, 30 steady clients (50 MB/s each), and 2 aggressive readers (1400 MB/s each). Total requested read bandwidth (4300 MB/s) significantly surpasses the system's I/O capability (1280 MB/s read bandwidth), making cache effectiveness crucial.

Figure 6 illustrates the impact on tail latency during cache reclamation. Ramp-up clients reclaiming cache resources experience substantial tail latency spikes in RocksDB-FS, approximately 30× higher compared to a system with static

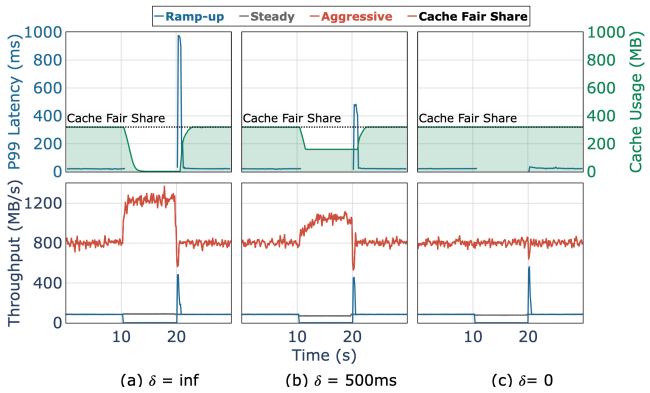


Figure 6. Cache microbenchmarks. The original authors show the p99 latency and throughput of ramp-up (blue), aggressive (red), and steady (grey) clients. FairDB has minimal reservations to effectively meet ramp-up clients' delay tolerance (here, δ = 500ms)

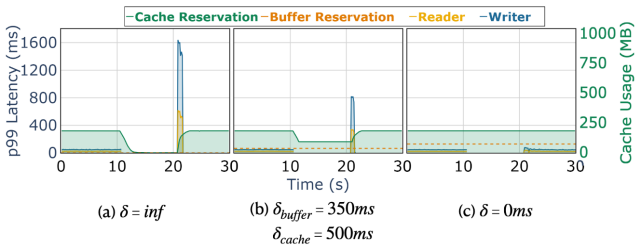
quotas (RocksDB-Qts). FairDB reduces these spikes by a factor of two when configured with moderate latency tolerance (δ =500ms). The reservation policy ensures these clients rapidly regain their fair shares while substantially improving resource utilization. FairDB 's reservations (160 MB per client for moderate tolerance) optimize this balance between latency sensitivity and efficiency, yielding up to 30% better cache utilization compared to static quotas.

6.1.2 Delays Compose across Multiple Resources Realistic workloads frequently contend for multiple resources simultaneously. Therefore, evaluating delay composability is essential to understanding FairDB 's practical benefits.

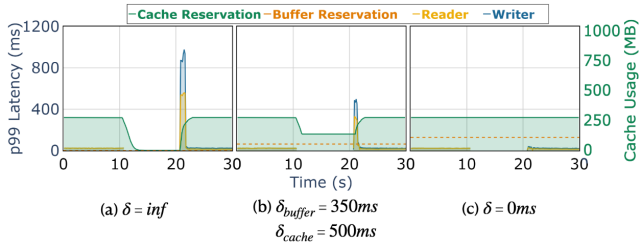
The authors examine two scenarios:

- Sequential acquisition (Read-Modify-Write workload): This scenario demonstrates additive delay composition. A client sequentially reads from cache, modifies data, then writes to buffers. FairDB bounds end-to-end latency to within the sum of configured individual tolerances (δ_{buffer}=350ms, δ_{cache}=500ms). Experimental results (originally shown in Figure 7b) confirm that FairDB 's composability guarantees the aggregate delay does not exceed specified bounds. RocksDB-FS exhibits significantly higher unbounded latencies (up to 817ms), demonstrating the clear benefit of FairDB 's reservation approach.
- Parallel acquisition: The authors simulate simultaneous read and write operations from clients, capturing the best-case scenario for composability. Here, FairDB restricts total latency to the worst single resource delay. Results, as depicted in Figure 7, validate that parallel resource access in FairDB substantially outperforms RocksDB-FS, affirming efficient resource handling and latency containment.

Note that while these evaluations were done before implementing δ-fairness for the flush threadpool, high delay from the flush queue is not a major concern as the write latency was still under the configured δ_{buffer}, suggesting that there



(a) The worst-case, additive composition of delays in FairDB when resources are acquired sequentially. Clients have read-modify-write workloads, YCSB Run-F.



(b) The best-case latency spike in FairDB *i.e.,* the maximum delay across resources when they are acquired in parallel. Clients have independent read and write requests, YCSB Run-A.

Figure 7. Multiple resources. The delay bounds compose across multiple resources for the end-to-end latency spikes of ramp-up clients; FairDB meets the configured delay tolerances for each resource.

was either enough headroom from the buffer reservation to sufficiently subsume the queue delay or there was simply no competition for the flush threadpool in this instance.

6.1.3 Sensitivity and Configurability (δ, k) The effectiveness and adaptability of FairDB policies are further investigated by varying latency sensitivity (δ) and workload burstiness (k):

6.1.3.1 Latency Sensitivity Systematic exploration across various δ values demonstrates FairDB 's flexibility. As latency tolerance relaxes (higher δ values), required reservations decrease significantly, directly benefiting system throughput. Tables 2 and 4 from the original clearly illustrate the fine-grained configurability offered by FairDB, consistently achieving latencies within specified δ constraints, thus enabling administrators to balance isolation guarantees with throughput objectives.

6.1.3.2 Varying Workload Parameterization (k) The authors assess how FairDB handles simultaneous resource reclamation (k clients ramping concurrently). Higher burst thresholds demand larger resource reservations, notably impacting system performance (Table 3). Even as reservation needs grow with increasing burst thresholds, FairDB consistently maintains performance within configured latency bounds, showcasing the system's robustness under varying workload intensities.

Read Cache (δ)	0 ms	250 ms	500 ms	750 ms	inf
Additional Delay (ms)	<1 ms	236	484	729	977
System Throughput (GB/s)	3.10 (+0 %)	3.36 (+8.4 %)	3.64 (+17.4 %)	3.82 (+23.2 %)	3.92 (+26.5 %)
Write Buffer (δ)	**0 ms**	**200 ms**	**350 ms**	**500 ms**	**inf**
Additional Delay (ms)	<1 ms	161	332	494	596
Reservation Size (%)	12.5 %	9.4 %	6.3 %	3.1 %	0 %

Table 2. Sensitivity to δ bounds: FairDB effectively maintains minimal reservations for each resource read cache and write buffer for varying δ *i.e.,* clients with different latency sensitivity.

δ (ms)	k	Res (%)	P99 Latency (ms)	Sys Tput (GB/s)
750	1	25%	729	3.82
750	2	62.5%	739	3.53
750	3	75%	741	3.41
750	4	81.25%	726	3.29
750	5	85%	734	3.18
750	6	87.5%	732	3.10

Table 3. Sensitivity to the system model. Increasing peak burst threshold (k) allows more clients to reclaim their fair share of resources simultaneously; FairDB effectively computes minimal reservations to support an administrator-configured k.

δ_{buffer} (ms)	δ_{cache} (ms)	P99 Latency (ms)
200	250	425
200	500	694
200	750	925
350	250	582
350	500	831
350	750	1040
500	250	752
500	500	983
500	750	1154

Table 4. Sensitivity to δ bounds: FairDB effectively maintains minimal reservations across multiple resources, with varying δ for clients with read–modify–write workloads.

FairDB	Read (GB/s)	Write (GB/s)	Overall (GB/s)
δ=0	5.4	7.5	6.2 (**1**×)
δ = inf	7.4	0.91	8.3 (**1.34**×)
δ_{buffer} =350ms, δ_{cache} =250ms	7.1	0.88	8.0 (**1.29**×)

Table 5. FairDB overall combined throughput: FairDB with δ-fair policies has comparable throughput relative to RocksDB with static quotas (δ = 0), or fair sharing (δ = inf). FairDB achieves **4–9.3**× better tail latency compared to fair sharing.

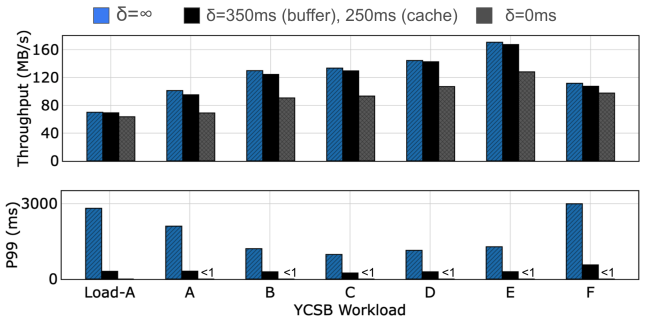


Figure 8. FairDB on YCSB. Index (from left to right): δ = {inf, 350ms buffer, 250ms cache, 0}. The authors show the throughput and p99 latency of clients, per YCSB workload. The δ =inf (blue) depicts RocksDB with fair sharing and δ =0 represents static quotas (gray). δ-fair policies achieves up to 9× lower p99 latency compared to fair sharing with comparable overall throughput.

6.1.4 YCSB MacroBenchmark Finally, the authors evaluate FairDB using representative workloads generated from Snowflake production traces, with 64 diverse clients (reads, writes, and mixed operations):

- **Performance Isolation:** FairDB significantly improves isolation, reducing tail latency by 4–9.3× compared to RocksDB-FS, ensuring latency-sensitive clients remain unaffected by aggressive behaviors from others.
- **Throughput Efficiency:** FairDB achieves throughput within 4% of RocksDB-FS and up to 35% higher than static quota-based allocation, reflecting the effective balance between isolation and utilization.

Detailed experimental results (as summarized in Table 5 and Figure 8 from the original) confirm the efficacy and practicality of the proposed framework, particularly in scenarios with mixed read-modify-write demands. The macrobenchmark clearly validates that FairDB reliably meets latency guarantees under realistic multi-resource contention scenarios.

6.2 Flush Threadpool

6.2.1 Flush Threadpool Microbenchmark A client reclaiming its fair share experiences nearly an 800× **higher tail latency**, despite fairly sharing the flush threadpool in RocksDB. In this scenario, threads are assumed to be

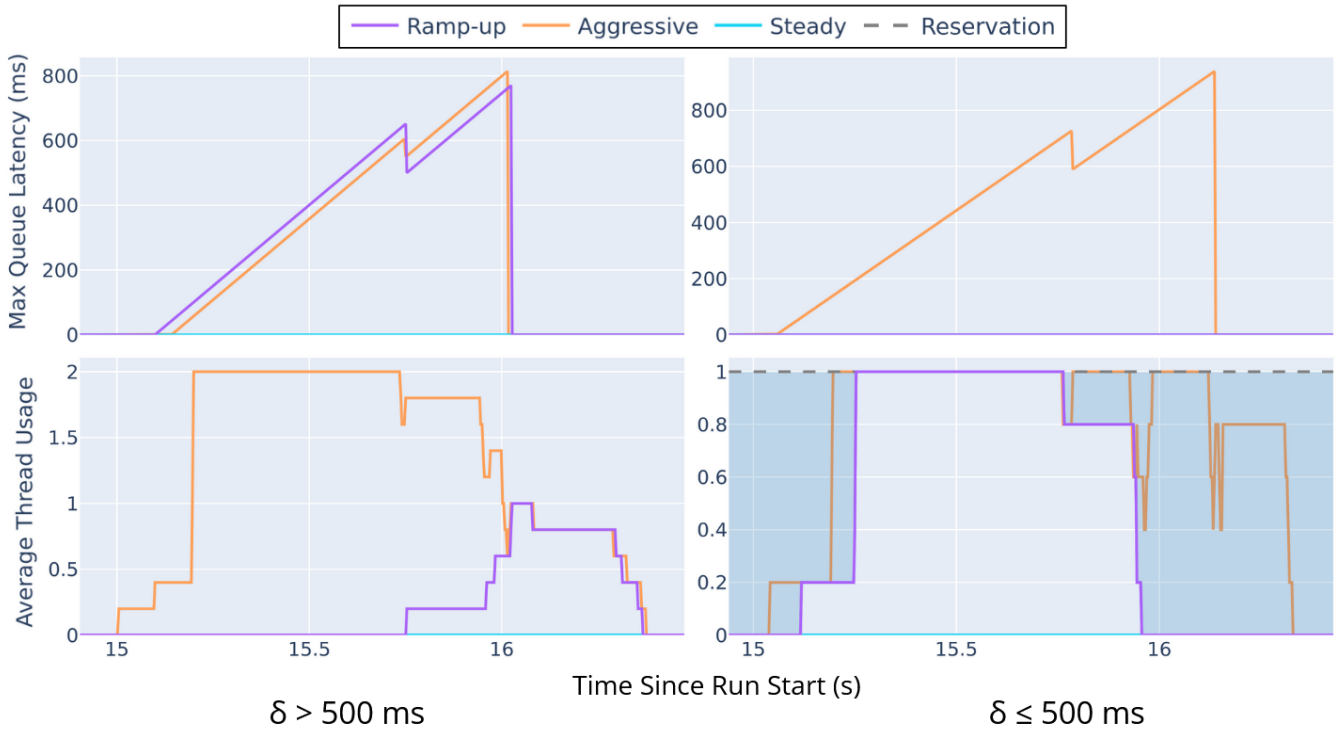


Figure 9. Benchmark showing δ-fairness behavior in the flush threadpool for differing δ values.

instantaneously released for such clients. FairDB improves this behavior, completely mitigating tail latency spikes in our microbenchmark, as illustrated in Figure 9. However, configurability remains limited due to the small number of flush threads relative to the number of clients, constraining the possible number of reservations.

In this experiment, we utilize 16 writers: 12 operate at 2 MB/s each, while 3 aggressively write at 4.2 MB/s. At 15 seconds, an additional client initiates a ramp-up by issuing a large 64 MB batch write, sufficient to trigger a flush, and subsequently continues at a steady 2 MB/s. Concurrently, the three aggressive clients also initiate flushes, saturating the available thread-width capacity of 2 thread-seconds/s. Clients execute the YCSB Load-A workload (write-only) with 8 kB requests. To precisely evaluate the effectiveness of the δ-fair policy for threadpool resources, we minimize interference from other resources and ensure fair sharing of write I/O among clients. The values in the chart are taken from 5 independent runs of the benchmark, with the top values being the max latency seen at each timepoint across all 5 benchmarks and the lower graph showing average thread usage by the ramp-up and aggressive threads at a given time during the 5 runs.

The flush for the ramp-up client cannot complete until the aggressive clients release the required thread-width (1 thread-second/s). Because each aggressive client occupies 1 thread-second/s, passing this thread-width on average takes approximately 500 ms. Consequently, ramp-up clients experience significantly increased tail latency under FairDB if $\delta > 500$ ms and no reservation is made for the ramp-up client. Conversely, FairDB can constrain this delay to nearly zero when $\delta \leq 500$ ms. Notably, we do not observe a substantial throughput reduction when reservations are made. The duration of high latency experienced during flush operations is similar with or without a reserved thread for the ramp-up client, and flush operations generally conclude by 16 seconds. Therefore, writes do not become blocked, as the second memtable continues filling for each client.

However, the limited number of flush threads inherently restricts how fine-grained this approach can be. As demonstrated in this example with only two flushing threads, reservations consume a substantial portion (50%) of available resources to guarantee δ fairness for the ramp-up client. Additionally, the observed maximum queuing latency exceeds 500 ms, contradicting our theoretical predictions. Although our theoretical framework accurately describes average behaviors over longer periods, it fails during short-term bursts because individual flush job durations can exceed 500 ms, resulting in inaccurate short-term predictions.

In larger databases with significantly higher write rates and more flush threads, resource reservations can become more fine-grained, as clients utilize multiple threads simultaneously to handle the greater number of memtables being flushed. This is because the fair share of a single client will

be on the order of a multiple threads, meaning that resource reservations will have much less overhead on the system as a whole since any "over-reservation" past the exact fair share will be smaller relative to the total system size. In addition, with a single client occupying multiple flush threads at the same time by having a high write rate, decisions on whether or not threads can access the reserve pool will be less all-or-nothing. Conversely, in smaller databases, it may be advantageous to maintain fully pooled thread resources to ensure higher throughput if threads risk becoming bottlenecked in the flush threadpool. Alternatively, dynamic reservations could offer advantages when anticipating large write bursts, with resources subsequently returned to the global pool.

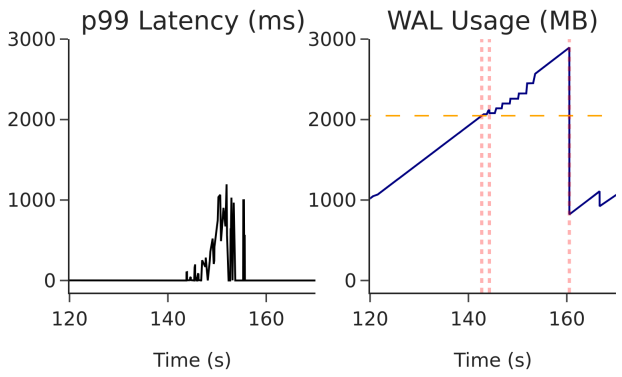


Figure 10. δ-fair Scheduling in the Flush Threadpool Mitigates WAL-Mediated Latency Spikes

6.2.2 WAL-Mediated Delay Mitigation via δ-fair Flush Threadpool Sharing Here we demonstrate how threadpool reservations mitigate WAL-mediated delay spikes. We adopt the same experimental setup described earlier, featuring 15 intermittent writers and a single steady, intensive writer. When the WAL fills, the 15 intermittent writers must flush data to free up space. Due to the reserved thread allocated for the steady writer, end-client latency spikes associated with these flush cascades decrease nearly five-fold compared to naive fair-sharing approaches, as depicted in Figure 10. Moreover, we observe that writes can continue beyond the WAL limit as other clients flush their data, eventually allowing the WAL to shrink.

This finding highlights the efficacy of fairly shared flush threads within a broader system context. Particularly in scenarios involving bursty writers without stringent latency constraints, DFS enables clients to reserve buffer portions, providing stronger guarantees regarding resource allocation latency.

7 Related Work

Fair scheduling. This work builds a long line of work on fair scheduling, including start-time fair queuing (SFQ) [25], weighted fair queuing (WFQ) [17, 38], and many other algorithms [10, 22]. For multi-resource fairness, DRF [21] is widely applied and provides both the share guarantee and Pareto efficiency. Other algorithms have been proposed, such as Competitive Equilibrium from Equal Incomes (CEEI) [45], but they are not strategy-proof. Dominant Resource Fair Queuing (DRFQ) [20] shares several features with the DFS algorithm since it ensures fair allocation over multiple (exclusive) resources for network packets. However, DRFQ does not address resources with high preemption delay, nor does it focus on latency-sensitive settings.

DB resource allocation. A range of work [8, 27, 29, 32, 35, 40, 42, 44, 50] addresses multi-resource allocation for DBMSes. Retro [32] implements various scheduling policies, including DRF, to achieve performance isolation of physical resources. pBox [29] handles intra-application interference of virtual resources (e.g., shared buffers, queues) but does not address cross-application interference. Pisces [42] applies DRF to a cloud key-value storage service while SQLVM [35] provides fixed resource reservations on SQL stores. However, these past works do not account resources with high preemption delay and therefore they do not introduce this class of performance interference. Other related work focuses on providing performance isolation via latency SLAs [12]/SLOs [9]. Predictable performance [30, 43], often via admission control techniques [14, 51], is another area complementary to latency sensitive resource sharing.

8 Conclusion

In this work, we explored the conceptual extension of Delayed Fair Sharing beyond the two resources in the original paper, identifying two other important resources that can be contended for. Our analysis revealed that applying delta-fairness to the Write-Ahead Log (WAL) was ineffective due to the nature of the latency source, which we found to be primarily driven by delays originating from the flush threadpool rather than the WAL itself. Consequently, we proposed and implemented a δ-fair flush thread scheduler within FairDB, a modified version of RocksDB, to directly target and mitigate these delay spikes. Our empirical results demonstrated significant reductions in both queuing and end-to-end latencies, underscoring the efficacy of managing fairness at the threadpool level rather than at the WAL. This approach builds upon prior DFS work, furthering our understanding of how targeted minimal reservations can effectively reduce tail latencies in system resource management.

References

[1] Linux 2.6.23: Completely Fair Scheduler. https://docs.kernel.org/scheduler/sched-design-CFS.html. Accessed: 2024-12-05.

[2] PostgreSQL: The World's Most Advanced Open Source Database. Accessed: 2024-12-10. URL: https://www.postgresql.org/.

[3] Amazon Web Services. Amazon Aurora. https://aws.amazon.com/rds/aurora/. Accessed: 2024-12-09.

[4] Amazon Web Services. Multi-Tenant Architectures on AWS. https://aws.amazon.com/solutions/guidance/multi-tenant-architectures-on-aws/. Accessed: 2024-12-10.

[5] Amazon Web Services. Multitenancy on Amazon Redshift. https://docs.aws.amazon.com/whitepapers/latest/multi-tenant-saas-storage-strategies/multitenancy-on-amazon-redshift.html. Accessed: 2024-12-09.

[6] Amazon Web Services. Multitenancy on DynamoDB. https://docs.aws.amazon.com/whitepapers/latest/multi-tenant-saas-storage-strategies/multitenancy-on-dynamodb.html. Accessed: 2024-12-09.

[7] Amazon Web Services. Quotas and Constraints for Amazon RDS. https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/CHAP_Limits.html, July 2024. Accessed: 2024-12-10.

[8] Sebastian Angel, Hitesh Ballani, Thomas Karagiannis, Greg O'Shea, and Eno Thereska. End-to-end performance isolation through virtual datacenters. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, page 233–248, USA, 2014. USENIX Association.

[9] Michael Armbrust, Kristal Curtis, Tim Kraska, Armando Fox, Michael J. Franklin, and David A. Patterson. Piql: success-tolerant query processing in the cloud. *Proc. VLDB Endow.*, 5(3):181–192, nov 2011.

[10] Jon C. R. Bennett and Hui Zhang. Wf2q: worst-case fair weighted fair queueing. In *Proceedings of the Fifteenth Annual Joint Conference of the IEEE Computer and Communications Societies Conference on The Conference on Computer Communications - Volume 1*, INFOCOM'96, page 120–128, USA, 1996. IEEE Computer Society.

[11] Yue Cheng, Ali Anwar, and Xuejing Duan. Analyzing alibaba's co-located datacenter workloads. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 292–297. IEEE, 2018.

[12] Yun Chi, Hyun Jin Moon, and Hakan Hacigümüş. icbs: incremental cost-based scheduling under piecewise linear slas. *Proc. VLDB Endow.*, 4(9):563–574, jun 2011.

[13] Cockroach Labs. CockroachDB Cloud. https://www.cockroachlabs.com/product/cloud/, 2024. Accessed: 2024-12-09.

[14] CockroachDB. Admission control in cockroachdb: How it protects against unexpected overload, 2024. URL: https://www.cockroachlabs.com/blog/admission-control-unexpected-overload/.

[15] 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, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[16] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and simulation of a fair queueing algorithm. *ACM SIGCOMM Computer Communication Review*, 19(4):1–12, 1989.

[17] Danny Dolev, Dror G. Feitelson, Joseph Y. Halpern, Raz Kupferman, and Nathan Linial. No justified complaints: on fair sharing of multiple resources. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, page 68–75, New York, NY, USA, 2012. Association for Computing Machinery.

[18] Facebook. Rocksdb: A persistent key-value store for fast storage environments. https://rocksdb.org, 2022. Accessed: 2024-011.

[19] Facebook, Inc. RocksDB Tuning Guide. https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide. Accessed: 2024-12-10.

[20] Ali Ghodsi, Vyas Sekar, Matei Zaharia, and Ion Stoica. Multi-resource fair queueing for packet processing. SIGCOMM '12, page 1–12, New York, NY, USA, 2012. Association for Computing Machinery.

[21] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: fair allocation of multiple resource types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, page 323–336, USA, 2011. USENIX Association.

[22] S Jamaloddin Golestani. A self-clocked fair queueing scheme for broadband applications. In *Proceedings of INFOCOM'94 Conference on Computer Communications*, pages 636–646. IEEE, 1994.

[23] Google Cloud. Bigtable Quotas and Limits, 2024. Accessed: 2024-12-10. URL: https://cloud.google.com/bigtable/quotas.

[24] Google Cloud. Cloud Spanner. https://cloud.google.com/spanner?hl=en, 2024. Accessed: 2024-12-09.

[25] Pawan Goyal, Harrick M. Vin, and Haichen Chen. Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks. *SIGCOMM Comput. Commun. Rev.*, 26(4):157–168, aug 1996.

[26] Tyler Griggs, Dev Bali, Wenjie Ma, James DeLoye, Audrey Cheng, Jaewan Hong, Soujanya Ponnapalli, Natacha Crooks, Scott Shenker, Ion Stoica, and Matei Zaharia. Delayed fair sharing: Performance isolation for multi-tenant storage systems. Manuscript submitted for publication, 2025.

[27] Ajay Gulati, Arif Merchant, and Peter J. Varman. mclock: handling throughput variability for hypervisor io scheduling. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, page 437–450, USA, 2010. USENIX Association.

[28] Avital Gutman and Noam Nisan. Fair allocation without trade. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems - Volume 2*, AAMAS '12, page 719–728, Richland, SC, 2012. International Foundation for Autonomous Agents and Multiagent Systems.

[29] Yigong Hu, Gongqi Huang, and Peng Huang. Pushing performance isolation boundaries into application with pbox. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 247–263, New York, NY, USA, 2023. Association for Computing Machinery.

[30] Virajith Jalaparti, Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Antony Rowstron. Bazaar: Enabling predictable performance in datacenters. *Microsoft Res., Cambridge, UK, Tech. Rep. MSR-TR-2012-38*, 2012.

[31] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS operating systems review*, 44(2):35–40, 2010.

[32] Jonathan Mace, Peter Bodik, Rodrigo Fonseca, and Madanlal Musuvathi. Retro: Targeted resource management in multi-tenant distributed systems. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, page 589–603, USA, 2015. USENIX Association.

[33] Matei Zaharia. Lessons from Large-Scale Cloud Software at Databricks, 2019. Accessed: 2024-12-9. URL: https://acmsocc.org/2019/slides/socc19-slides-keynote-zaharia.pdf.

[34] Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. Gimbal: enabling multi-tenant storage disaggregation on smartnic jbofs. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 106–122, 2021.

[35] Vivek Narasayya, Sudipto Das, Manoj Syamala, Badrish Chandramouli, and Surajit Chaudhuri. Sqlvm: Performance isolation in multi-tenant relational database-as-a-service. In *CIDR 2013*. 6th Biennial Conference on Innovative Data Systems Research, January 2013.

[36] Oracle Corporation. MySQL: The World's Most Popular Open Source Database. Accessed: 2024-12-10. URL: https://www.mysql.com/.

[37] Abhay K. Parekh and Robert G. Gallager. A generalized processor sharing approach to flow control in integrated services networks—the single node case. In *Proceedings of the Eleventh Annual Joint Conference*

of the IEEE Computer and Communications Societies on One World through Communications (Vol. 2), IEEE INFOCOM '92, page 915–924, Washington, DC, USA, 1992. IEEE Computer Society Press.

[38] Abhay K. Parekh and Robert G. Gallager. A generalized processor sharing approach to flow control in integrated services networks—the single node case. In *Proceedings of the Eleventh Annual Joint Conference of the IEEE Computer and Communications Societies on One World through Communications (Vol. 2)*, IEEE INFOCOM '92, page 915–924, Washington, DC, USA, 1992. IEEE Computer Society Press.

[39] PingCAP. TiDB: Distributed SQL Database, 2024. Accessed: 2024-12-9. URL: https://www.pingcap.com/.

[40] Qifan Pu, Haoyuan Li, Matei Zaharia, Ali Ghodsi, and Ion Stoica. Fairride: near-optimal, fair cache sharing. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, page 393–406, USA, 2016. USENIX Association.

[41] Charles Reiss, Alexey Tumanov, Gregory R Ganger, Randy H Katz, and Michael A Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the third ACM symposium on cloud computing*, pages 1–13, 2012.

[42] David Shue, Michael J. Freedman, and Anees Shaikh. Performance isolation and fairness for Multi-Tenant cloud storage. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 349–362, Hollywood, CA, October 2012. USENIX Association.

[43] Zilong Tan and Shivnath Babu. Tempo: robust and self-tuning resource management in multi-tenant parallel databases. *Proc. VLDB Endow.*, 9(10):720–731, jun 2016.

[44] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. Ioflow: a software-defined storage architecture. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 182–196, New York, NY, USA, 2013. Association for Computing Machinery.

[45] Hal R Varian. Equity, envy, and efficiency. 1973.

[46] Midhul Vuppalapati, Giannis Fikioris, Rachit Agarwal, Asaf Cidon, Anurag Khandelwal, and Eva Tardos. Karma: Resource allocation for dynamic demands. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 645–662, 2023.

[47] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. Building an elastic query engine on disaggregated storage. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 449–462, Santa Clara, CA, February 2020. USENIX Association.

[48] Carl A Waldspurger and William E Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, pages 1–es, 1994.

[49] Carl A Waldspurger and William E Weihl. Stride scheduling: deterministic proportional-share resource management. 1995.

[50] Andrew Wang, Shivaram Venkataraman, Sara Alspaugh, Randy Katz, and Ion Stoica. Cake: enabling high-level slos on shared storage systems. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, New York, NY, USA, 2012. Association for Computing Machinery.

[51] Pengcheng Xiong, Yun Chi, Shenghuo Zhu, Junichi Tatemura, Calton Pu, and Hakan HacigümüŞ. Activesla: a profit-oriented admission control framework for database-as-a-service providers. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, New York, NY, USA, 2011. Association for Computing Machinery.

[52] Juncheng Yang, Yao Yue, and KV Rashmi. A large-scale analysis of hundreds of in-memory key-value cache clusters at twitter. *ACM Transactions on Storage (TOS)*, 17(3):1–35, 2021.