Rein: Taming Tail Latency in Key-Value Stores via Multiget Scheduling

Waleed Reda[†◇] Marco Canini[⋆] Lalith Suresh[‡] Dejan Kostić[◇] Sean Braithwaite[§]

[†]Université catholique de Louvain [⋆]KAUST [‡]VMware Research
[◇]KTH Royal Institute of Technology [§]SoundCloud

Abstract

We tackle the problem of reducing tail latencies in distributed key-value stores, such as the popular Cassandra database. We focus on workloads of multiget requests, which batch together access to several data elements and parallelize read operations across the data store machines. We first analyze a production trace of a real system and quantify the skew due to multiget sizes, key popularity, and other factors. We then proceed to identify opportunities for reduction of tail latencies by recognizing the composition of aggregate requests and by carefully scheduling bottleneck operations that can otherwise create excessive queues. We design and implement a system called Rein, which reduces latency via inter-multiget scheduling using low overhead techniques. We extensively evaluate Rein via experiments in Amazon Web Services (AWS) and simulations. Our scheduling algorithms reduce the median, 95th, and 99th percentile latencies by factors of 1.5, 1.5, and 1.9, respectively.

CCS Concepts • **General and reference** → **Performance**; • **Computer systems organization** → **Distributed architectures**; • **Information systems** → *Distributed storage*

1. Introduction

Modern cloud services require increasingly stricter service level objectives for performance. These objectives are often defined in terms of a percentile of the latency distribution (such as the 99.9^{th}-ile [24]). Thanks to scale-out designs and simple APIs, key-value storage systems have emerged as a fundamental building block for creating services capable of delivering microseconds of latency and high throughput.

Moreover, for interactive Web services, requests with a high fan-out that parallelize read operations across many different machines are becoming a common pattern for reducing latency [10, 11, 37, 42]. These requests, which batch together access to several data elements, typically generate tens to thousands of operations performed at backend servers, each hosting a partition of a large dataset [46]. This makes reducing latency at the tail even more imperative because the larger a fan-out request is, the more likely it is to be affected by the inherent variability in the latency distribution, where the 99th-ile latency can be more than an order of magnitude higher than the median [23, 28].

To support this request pattern, many popular storage systems, including wide-column stores (e.g., Cassandra [2]) and document stores (e.g., Elasticsearch [3]), offer a `multiget` API. A multiget request allows multiple values at the specified keys to be retrieved in one request. Batching multiple read operations together is typically far more efficient than making several individual single-key requests.

Multiget requests provide opportunities to reduce latency via scheduling. These requests follow the "all-or-nothing" principle, where a multiget only finishes once all of its operations are completed and the last operation's response is received. In practice, multiget requests in real-world workloads present variability in attributes like size, processing time, and fan-out as well as load imbalances and accessed keys (§2). Such variability means that the response time of different requests is likely affected by different bottleneck operations across different servers. Our insight is that it is possible to improve latency by scheduling operations while considering what bottleneck affects the overall request completion time, and "slacking" non-bottleneck operations for as long as they cause no extra delay to the request. Slacking certain operations can allow other operations with tighter bottlenecks to be serviced earlier and thus decrease aggregate response times as well as latencies at the tail.

Scheduling operations of multiget requests across cluster nodes of a storage system is challenging. Even when a priori knowledge of all operations to be scheduled and their service times is assumed, the problem can be expressed as a concurrent open-shop scheduling problem, which is known to be NP-complete [40]. Moreover, no single scheduling algo-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

EuroSys '17, April 23 - 26, 2017, Belgrade, Serbia

© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ISBN 978-1-4503-4938-3/17/04...$15.00

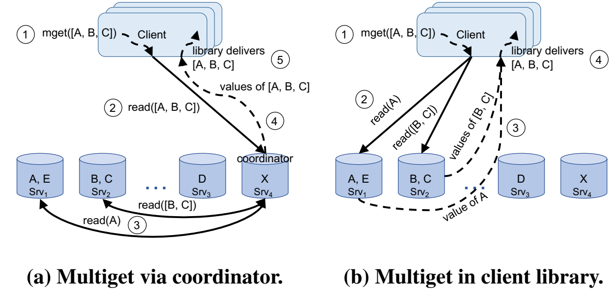
DOI: http://dx.doi.org/10.1145/3064176.3064209

rithm provides optimal ordering for reducing aggregate response times. For heavy-tailed distributions of request sizes, policies such as Shortest Job First (SJF) help to reduce average makespans, but for light-tailed distributions First Come First Served (FCFS) can become the optimal scheduling strategy [25]. It has also been proven that there is no non-learning policy that can optimize scheduling across a wide variety of workloads [50].

In practice, the problem is even more challenging because operation completion times are not known in advance due to several factors, including variable service rates, server load imbalances, skewed access patterns, etc. In addition, these operations are meant to have very low latency and are typically serviced in parallel across a number of different servers. As such, an efficient scheduling algorithm has to operate on multiple uncoordinated servers with minimal overhead. Thus, this problem is distinct from scheduling batch jobs in big data systems (e.g., [53]) in which a centralized scheduler can make coordinated decisions.

To address these challenges, we introduce Rein (§3), a multiget scheduler that leverages variability in the structure of multiget requests to reduce the median as well as tail latency. We first devise a method for predicting bottleneck operations within requests, and, based on these predictions, we employ a combination of two policies to schedule requests in an efficient manner (§3.2). The first policy – Shortest Bottleneck First (SBF) – prioritizes requests with smaller bottlenecks, which reduces head-of-line blocking and improves average makespans. The second policy – Slack-Driven Scheduling (SDS) – allows us to use resources efficiently by delaying non-bottleneck operations without impacting the overall request completion time. The priorities computed by the aforementioned policies are assigned at the client-side and then enforced at the backend nodes responsible for servicing the requests. We leverage both policies to schedule requests in a manner that is optimized for both median and higher percentiles of the latency distribution. We then provide a blueprint for how to enforce our scheduling order in highly concurrent systems by using a novel scheduling approach called multi-level queues (§3.3), which combines the advantages of classic scheduling algorithms while yielding an efficient implementation. Our technique achieves a low-overhead implementation by adopting non-blocking FIFO queues that significantly reduce contention among threads.

We demonstrate the feasibility and benefits of our approach by instantiating our prototype implementation (§4) within Cassandra, a popular distributed database. Through empirical evaluation (§5.1) and simulations (§5.3) using both production and synthetic workloads, we show that Rein can reduce the median latency and 95th percentile by 1.5 times and the 99th percentile latency by 1.9 times compared with multiget-agnostic, FIFO scheduling. We also compare our approach with other state-of-the-art techniques and find that



(a) Multiget via coordinator. **(b) Multiget in client library.**

Figure 1: Common styles of multiget requests in partitioned key-value stores.

Rein consistently outperforms the others under varying system conditions. We expect that our results will inspire the design of new multiget-aware scheduling algorithms achieving even greater improvements. Our Rein implementation is available as open source.[1]

2. Multiget Requests

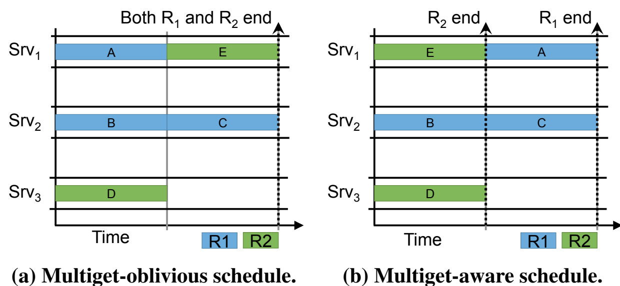
Multiget requests are a common idiom employed in modern cloud services to use storage systems efficiently. Many popular systems, including Redis [6], Memcached [4], Cassandra [2], MongoDB [5] and Elasticsearch [3], offer support for multiget requests. Their usage is simple. Developers write client-side code that batches access to multiple values as a single request. For example, `mget(A, B, C)` requests values of keys A, B, C. Their execution dependes on the system, as we illustrate in Figure 1. For systems like Cassandra (Figure 1a), the client-side library sends the multiget request to one node of the cluster (referred to as the coordinator, e.g., Srv_4), which is responsible for reading all values across cluster nodes and combining them into a single response to be sent back to the client. In other systems like Memcached (Figure 1b), the client-side library itself could obtain the response by parallelizing data fetches across cluster nodes, each of which receives only (batched) read operations for keys that fall in the data partition it serves.

Regardless of the implementation, as data is typically partitioned (and replicated) across cluster nodes, the likelihood that a multiget request would require fetching values distributed among nodes increases with cluster size and request size (i.e., number of requested keys). The resulting workloads consist of fan-out operations across nodes. Moreover, these workloads are likely to exhibit significant variations as key-value workloads for cloud applications are often skewed [13, 21], multiget requests vary in size and processing time [37], and individual servers are exposed to performance variability (e.g., due to resource contention) [23, 44].

2.1 Benefits of Multiget Scheduling

Variations in the structure of multiget requests give rise to possible performance improvements through inter-multiget

[1] Code at https://github.com/wreda/cassandra-rein.



(a) Multiget-oblivious schedule. **(b) Multiget-aware schedule.**

Figure 2: Performance of requests using a multiget-oblivious (a) and a multiget-aware (b) schedule. Multiget-aware scheduling reduces average response time.

scheduling. We illustrate these with a simple example. Consider again the system in Figure 1. Assume that two multiget requests, R_1 and R_2, arrive at the same time: one request is for three keys, A, B, C and the second is for two keys, D, E. Given how the data are partitioned, request R_1 is broken down into two sets of operations (called *opset*s) to read the values of A and B, C from servers Srv_1 and Srv_2, respectively.

What will the response times of these requests be? For presentation sake, assume that every server serves each operation with a service rate, μ, of one operation per unit of time (e.g., 1 op/ms). This means that serving the B, C opset will last two units of time (we omit network latencies for now). We call the opset with the longest response time the *bottleneck opset* (which is B, C in this example). A multiget response time depends on its bottleneck's completion time. Thus, assuming no queuing, R_1 will complete in two units of time whereas R_2 will complete in one unit of time thanks to parallelization across two servers. However, when R_1 and R_2 execute concurrently, operations might happen in the order A, E. With this schedule, both requests complete in two units of time as shown in Figure 2a. This is because servers are oblivious to the structure of multiget requests.

Would processing requests in a multiget-aware fashion lead to benefits? Because a multiget request is complete only once all its operations complete, there is some *slack* for accessing A — in particular, to cause no extra delay to R_1, the access to A can be postponed for as long as it completes with a delay of up to two units of time. Given this information about the global deadline of a multiget request, a server could serve operations to meet their deadlines while minimizing the delay of other operations. In our example, server Srv_1 can perform the read operation for E before the one for A. With this optimal schedule, the completion time of R_2 is just one unit of time and the average response time minimized to 1.5 as shown in Figure 2b.

Underpinning these performance improvements are the variations in multigets' attributes like size, processing time, and fan-out as well as the load imbalances across cluster nodes and other common skews of key-value store workloads, such as heavy-tailed key access frequency. In principle, the higher the variance of these factors, the greater the opportunity for slack-driven latency reductions, in particular at the tail of the latency distribution. Using a trace from a production cluster at SoundCloud, we next highlight the variations in request sizes and workloads and then quantify the performance improvements that these variations may yield in practice.

2.2 Analysis of Multiget in Production

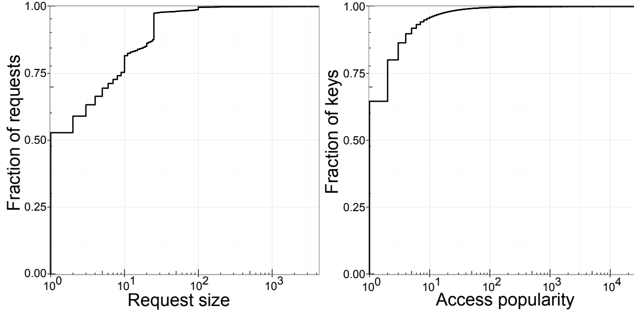
We focus on understanding the characteristics of multiget requests by analyzing a 30-minute trace from a production cluster at SoundCloud gathered during its operation. We highlight that the structure of multiget requests exhibits significant variations in size and key popularity. While this single workload does not generalize to all systems and environments, we note that similar properties were observed in other environments [10, 11, 31, 37, 42]. These insights motivate and inform Rein's design.

Figure 3a plots the distribution of multiget request sizes. The request sizes show a heavy-tailed distribution: ∼40% of requests involve more than a single key; the average size is 8.6 keys and the maximum is as high as ∼2000 keys.

Figure 3b plots the distribution of key access frequency. This distribution is also heavy-tailed. Most keys are accessed just once in the collected trace, whereas a few keys are accessed up to 1000 times.

The values being accessed in this trace are of fixed size. This allows us to validate that there exists a positive correlation between a multiget size and its response time. However, as we were limited in the intrusiveness of our instrumentation, we are not able to report on the variation in performance from the production cluster. To quantify the variation in performance across multiget requests and to provide figures in the context of the results of this paper, we use the trace to benchmark our Cassandra testbed composed of 16 AWS EC2 m3.xlarge nodes (our setup is detailed §5). We generate our workload at 75% utilization of the system capacity and measure the response time of multiget requests. Figure 4 shows the distribution of response times. We also analyze and plot the average multiget size of requests binned by their response times across 10 percentile intervals of the response time distribution. We observe that there exists a positive correlation between the response time of a multiget and its corresponding size (the Pearson's correlation coefficient is 0.403). In other words, the larger the multiget size, the more likely it will incur a higher response time.

The main takeaway from our analysis is that multiget requests vary widely in size, key access pattern, and response time. Combined these variations manifest as uneven load on the serving nodes. These variations bring several challenges to achieve performance predictability but at the same time they create opportunities for performance improvements through scheduling read operations, as we discuss next.



(a) Distribution of request sizes. **(b) Distribution of key popularity.**

Figure 3: Multiget requests from a production trace exhibit significant variations in size (a) and key popularity (b).

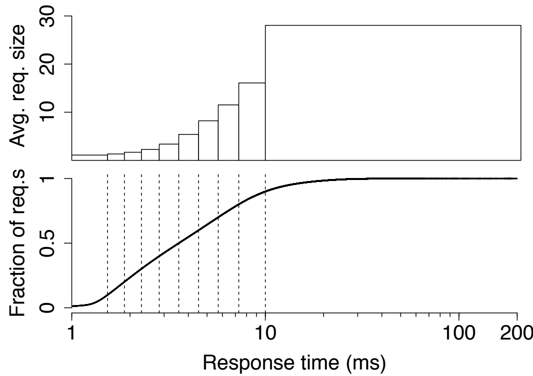


Figure 4: The distribution of multiget response times (bottom). The average multiget size of the requests binned by response time for each of the 10 percentile intervals of the response time distribution (top).

2.3 Quantifying Latency Reduction Opportunities

We illustrate the potential benefits of inter-multiget scheduling by analyzing our production traces to quantify latency reductions at different percentiles. This analysis focuses on upper and lower bounds estimated on a simplified model of the system (detailed in §5.3), though our evaluations are based on production and synthetic workloads on a real system prototype. We present simple estimates for this analysis because the problem of choosing the optimal request allocation can be shown to converge into a more complex version of the online knapsack problem, which is a difficult problem [16].

Recall that we denote by slack the possible delay of an operation without affecting the completion time of its containing request. As an upper bound, we assume that every operation that can be slacked could ideally produce a latency reduction equal to the slack for that operation. We compute it by subtracting the response time of an operation from the response time of its request. To estimate a lower bound, our model assumes that every server has a queue of operations where operations are enqueued if the server is not idle. We

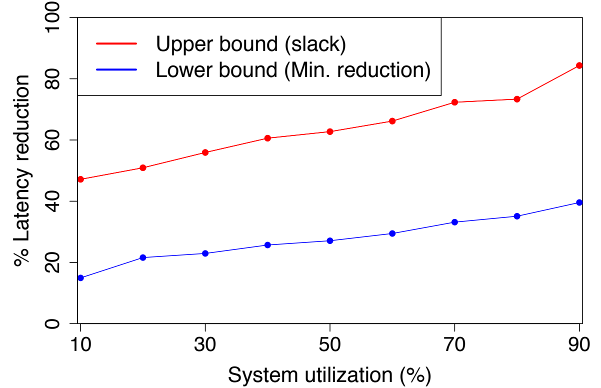


Figure 5: Upper and lower bounds for latency reductions at different system utilization levels.

then consider every operation, x, that can be slacked and check whether the server processing that operation could have processed the subsequent operation, y, in its operation queue such that the service time of y is no larger than the slack time of x. If that is the case, we measure the resulting latency reduction if it is greater than zero.

Figure 5 plots the range of average per-operation latency reduction (area between upper and lower bounds) for increasing system utilizations. It can be seen that latency reduction opportunities increase with utilization levels and that at 90% utilization is roughly double that at 10% utilization.

In summary, these results demonstrate that there exist potentially good opportunities to reduce request latencies via slacking operations.

3. Rein Scheduling

Rein aims to reduce latency of key-value stores via inter-multiget scheduling by exploiting variations in the attributes of multigets, such as size, processing time, and fan-out. Specifically, the scheduler's goals are as follows:

• Achieve best-effort minimization of median multiget response times.

• Provide a more predictable performance by reducing high-percentile latencies.

Our objective is not to determine an optimal schedule, given the hardness of the underlying scheduling problem. Rather, we seek to design heuristics that can improve response times under realistic settings. For this reason, we resolve to employ a novel heuristic solution that fits within the desired constraints that we fix in the design space as we elaborate below.

The design space of solutions for minimizing latency of storage systems is large, even if only scheduling-based solutions are considered. We make the following decisions to guide us towards a practical solution:

• The scheduler should operate online, with minimal overhead, and assuming no prior knowledge of requests.

• There should be no coordination between clients and servers, nor centralized components; rather, clients may only pass information to servers in the form of meta-data assigned to individual operations.

3.1 Solution Overview

We address the design goals via two main components: (i) server-side, multiget-aware scheduling based on (ii) client-side priority assignment. Figure 6 visualizes the architecture used in our approach.

As seen in the figure, the process begins when a multiget request is issued at the requester endpoint. According to what we illustrated in Figure 1, this endpoint is either the client that issues the request via the client-side library or the coordinator node that is tasked with processing the request submitted by a client of the cluster.

The multiget request is first subdivided into a collection of *opsets*, wherein each opset comprises of all operations for each distinct data partition. Thus, operations are split according to the same strategy used for selecting servers to serve operations (typically some form of hashing). Empty opsets are pruned out. Note that, in replicated storage systems, this split maps to the number of replica groups; that is, it maps to the set of servers responsible for a data partition.

Next, the requester endpoint assigns each opset with a certain priority using a priority assignment strategy (detailed later). This priority is then inserted as meta-data in every operation of a given opset; all operations of an opset share the same priority.

The scheduling of operations does not occur until they are processed on the server side. After priority assignment, each operation is sent to a server responsible for the correct data partition. At a high level, the server then serves requests according to their assigned priority, reaping the benefits of scheduling operations in a multiget-aware fashion. However, to perform this scheduling efficiently in practice, our design makes use of a multiple queues. As we discuss below, this design approximates the desired highest-priority-first order while preventing starvation and enabling an implementation based on non-blocking FIFO queues.

While this design appears conceptually simple, the devil is in the detail. The difficulty stems from the fact that, by design, each client (or coordinator) in our solution needs to make independent decisions based on only the structure of each mutiget request. Thus, there are two major challenges that we need to solve: (i) what strategy to use for priority assignment? and (ii) how do we efficiently perform the operation scheduling?

3.2 Design Details

We now discuss the components of our solution that address these challenges.

3.2.1 Multiget-Aware Scheduling

Multiget-aware scheduling exploits the fact that a multiget's response time depends on the slowest of its operations to complete (i.e., the bottleneck operation). Our first approach was to consider having a simple priority queue on the server side and assigning priorities on the client side. Priorities are assigned based on a notion of the cost of an operation. The cost aims to reflect the amount of work necessary to process an operation. The cost of an opset refers to the sum of the costs of all its operations. The cost is also used to estimate the bottleneck of a request, as we explain below.

Bottleneck estimation. Estimating the bottleneck of a request (that is, the response time of the last operation) is a difficult problem because this time can be influenced by many factors, including server performance fluctuations (e.g., garbage collection, performance interference, or other background activities), resource contention, skewed access patterns, and caching. Knowing the requested value size is therefore not necessarily a good predictor of the response time. In queuing theory, our system can be modeled by a fork-join queue, which factors in the splitting (or forking) of requests amongst K servers and and then being joined into a single response after all operations return. The problem with these models, however, is that they fail to provide exact solutions for more than 2 servers [26]. As such, due to the difficulties of exact-analyses in these models, most studies focus on approximation techniques [45]. To make matters worse, based on our consultations with queuing theory experts, it is challenging for this model to accurately account for the complexities of our system given the multi-threaded nature of server nodes and the fact that multigets do not have a constant K and their opset sizes vary.

One might consider tracking several run-time metrics, such as the distribution of response times, the likelihood of finding a key in cache, etc., and performing an estimation using these data. We found that making an accurate estimation based on the distribution of previous response times resulted in sub-optimal scheduling decisions due to staleness of the information. In addition, given the need for sub-millisecond decision making, performing these predictions in an online fashion can be challenging. Relying on a simple heuristic can therefore be advantageous.

We choose a simple method that we found to yield reasonably good results in practice. For a given request, we assume that the bottleneck opset is the opset with the highest number of operations in it. Multiple opsets can be treated as the bottleneck if they are the same size. In other words, the cost scales linearly with the opset size, which follows what has been observed for the sizes of I/O queries in distributed execution environments [8, 12, 53]. Implicitly, we assume that all operations experience the same response time, which is of course not true in practice. Based on our experiments, we find that there is a weak correlation between the size of the data being requested and the service time if all data is

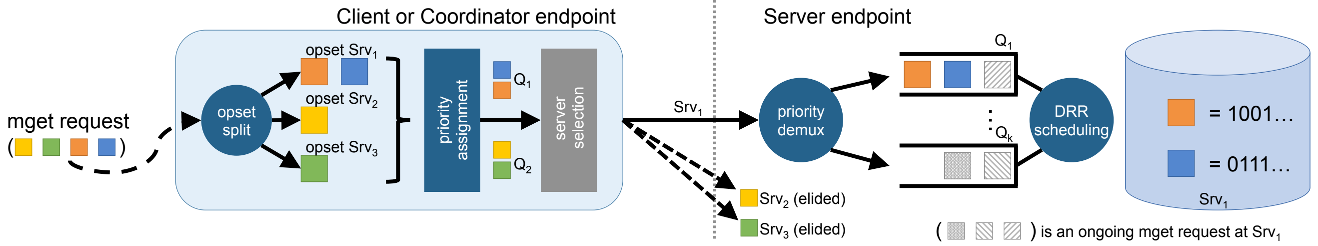


Figure 6: Overview of Rein scheduling for a multiget request arriving at the system. On the requester endpoint, the request ( ) is split into multiple opsets, each grouping the operations towards the same data partition. Priority assignment accounts for the cost of each opset and marks each operation with corresponding priority as meta-data. For example, operations are assigned the highest priority (level 1) as they are estimated to be the bottleneck and sent to Srv_1. On the server endpoint, each operation is enqueued into a queue based on its priority. Multiple queues are serviced using Deficit Round Robin (DRR), which efficiently approximates processing operations in highest-priority-first order.

stored in memory. However, this does not hold true if the data is only partially cached. Our evaluation for instance presents the impact that caching can have on our performance improvement. It is part of our on-going work to determine computationally cheap and effective bottleneck estimation techniques.

Initial scheduling algorithms. Based on this understanding of the problem, we started to investigate classic scheduling algorithms by considering the following insights:

- Prioritizing operations with shorter execution times can reduce head-of-line blocking and improve latencies.
- Deprioritizing operations that can be slacked can allow bottleneck operations to be processed earlier, which results in lower aggregate response times.

Using these two insights, we considered two natural scheduling schemes: *Shortest Bottleneck First* (SBF) and *Slack-Driven Scheduling* (SDS).

Shortest Bottleneck First assigns to every operation of a multiget request a priority that corresponds to the cost of the bottleneck opset of that multiget request. The intuition is that requests with shorter bottlenecks should be given precedence to minimize the average request makespan and reduce head-of-line blocking. This strategy is similar to Shortest Job First (SJF) scheduling as well as Shortest Remaining Processing Time (SRPT); however, in our case, given that request completion times are determined by the last operation to finish, we use the cost of the bottleneck opset instead of the cost of individual operations. SBF favors smaller requests by scheduling them ahead of larger multigets, penalizing them in the process. For workloads with heavy-tailed multiget sizes, which we use throughout the paper, we find that this strategy can be used to reduce *both* the median as well as higher percentile latencies (e.g., 95th and 99th). Like SJF, SBF and SRPT can be prone to starvation under certain workload distributions [14, 15]. To counter this, we also incorporate a technique to boost priority of an operation after a certain duration has passed since the operation entered the servicing server's queue.

Slack-Driven Scheduling assigns the priority for every operation x of a non-bottleneck opset, O, as the cost of x plus the slack of x divided by the number of operations in O, that is, $(cost(x) + slack(x))/size(O)$. A lower value corresponds to a higher priority. This deprioritizes operations based on how long they are allowed to be slacked without becoming the bottleneck themselves. In doing so, this policy aims to use server capacity more wisely by prioritizing servicing opsets in proportion to how they are bottlenecking their multiget request. As opposed to SBF, SDS does not penalize one request in favor of another. This is because the operations that are delayed are chosen such that they do not affect the overall response time of their parent multiget request.

While these approaches gave good results in simulation, we recognize that, in high-throughput systems (such as key-value stores), canonical priority queue implementations can act as significant performance bottlenecks due to lock contention in multi-producer/multi-consumer settings [39]. This is a well-known problem in the literature; several solutions have been proposed [9, 51]. We address this problem with a novel technique, which we term *multi-level queues*.

3.3 Multi-Level Queues to the Rescue

A way to avoid the inefficiencies of priority queues is to utilize multiple FIFO queues. This allows us to escape the lock contention problem since there exist implementations of lock-free FIFO queues that allow concurrent access without incurring in synchronization overheads.

We now introduce multi-level queues, a way to use K FIFO queues to approximate a single priority queue by assigning to each of the K-th FIFO queue a different dequeue rate, $w_{Q_i}, i \in [1, K]$. In this design, objects with higher priorities are allocated to queues with higher rates (and vice versa). This provides us with a way to approximate the behavior of a single priority queue in a contention-free manner. As a trade-off, we do not have complete control over the

scheduling order. The fact that our bottleneck estimation is just approximate makes this less of an issue for us.

3.3.1 High-Level Idea

Our aim is to use the multi-level queue data structure to realize our two scheduling policies — namely, SBF and SDS. To do so, we need to answer the following questions: (i) How can operations be assigned to queues in a way that realizes the aforementioned scheduling policies?
(ii) How do we configure the number of queues and their respective rates?

We describe some intuitive ways of answering (i) and then tackle (ii) in the sensitivity analysis later in this section. To realize SBF and SDS, we want to conceptually map these policies to multi-level queues while preserving the main insights that govern the policies.

For SBF, we find we can simply assign all the operations of a multiget request to a queue based on the request's cost. In other words, operations belonging to costlier requests get assigned to queues with lower rates, while operations for smaller multigets are assigned to the faster queues. In doing so, we allow smaller requests to bypass larger multiget requests and get serviced with higher priority. This allows us to approximate the SBF schedule.

To realize SDS, we can assign the bottleneck opset to the fastest queue. The non-bottleneck opsets are then "slacked" by assigning them to queues with lower rates based on the ratio of their own cost to that of the bottleneck. In other words, if an opset cost is half of its bottleneck, then, ideally, it should be assigned to a queue that is twice as slow. More generally, however, it should be assigned to the queue that minimizes the difference between the cost and rate ratios. We explain this in greater detail below.

Based on this high-level description, we see another opportunity for *combining both* approaches into one scheme. We achieve this by, first, assigning the bottleneck opset based on its cost (and not simply to the fastest queue as described for SDS). In doing so, we allow opsets with shorter bottlenecks to finish first, which satisfies the SBF policy. The non-bottleneck opsets are then assigned to queues with equal or lower rates according to the aforementioned ratios. This allows us to attain the benefits of SBF by deprioritizing requests with longer bottlenecks while at the same time reducing response time variability between opsets within a request, thereby realizing SDS and increasing the efficiency of resource utilization.

3.3.2 Multi-Level Queue Design

Our multi-level queue scheduler (depicted in Figure 7) consists of a set of K queues $\mathbb{Q} = \{Q_1, Q_2, \ldots, Q_K\}$. Each queue is assigned different dequeue rates, with the highest assigned w_{Q_1} and successive queues assigned progressively lower rates. The scheduler uses Deficit Round Robin (DRR) to dequeue operations based on the assigned dequeue rates.

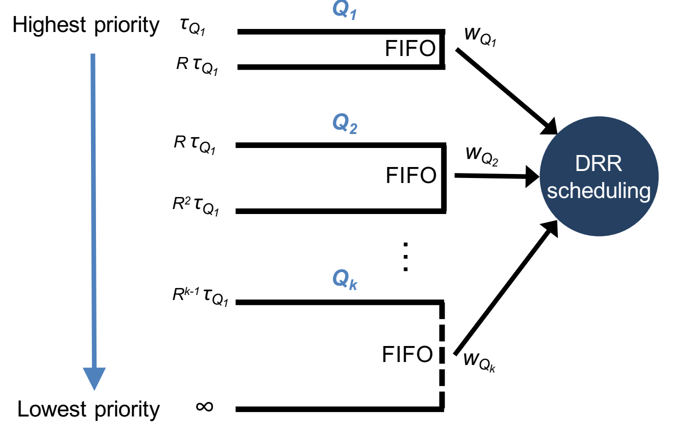


Figure 7: A multi-level queue scheme showing K FIFO queues. Consecutive queues have incrementally decreasing rates and exponentially larger bin sizes. Our scheduler uses DRR to dequeue operations from the queues.

Each queue is assigned an interval of opset costs so that each opset can be mapped to a queue based on its cost. For the queue Q_i, the interval is the pair of thresholds $\tau_{Q_i}, \tau_{Q_{i+1}}$. For $i = 1$, $\tau_{Q_i} = 1$, the minimum cost. For $i > 1$, successive thresholds are calculated as $\tau_{Q_{i+1}} = \tau_{Q_i} * R$ where R is the range factor. In other words, the queue intervals increase exponentially. We opt to use exponentially ranged thresholds since fine-grained prioritization can result in suboptimal makespans if the opset completion times are unknown [17]. Despite the fact that we know the size of each opset, our cost function is merely an estimate and the actual execution times can differ across servers due to variation of performance. Thus, we do not to differentiate between opsets in a fine-grained manner based on their sizes.

Opset queue assignments. We now detail how we assign priority to each opset (technically, the operations within it). Once a multiget request is split into its opsets, we calculate the cost of the bottleneck opset, B. Opset B is then assigned to Q_i such that $\tau_{Q_i} \leq cost(B) < \tau_{Q_{i+1}}$. In other words, bottlenecks with higher costs are assigned to lower priority queues, which preserves the SJF properties of SBF. At the same time, since $w_{Q_i} > 0, \forall i \in [1, K]$, our requests do not suffer from starvation as they are guaranteed a non-zero service rate.

We use a different method to assign non-bottlenick opsets to queues. Algorithm 1 gives the pseudo-code for the procedure. For a non-bottleneck opset, *op*, we calculate the ratio between its cost and that of the bottleneck opset. We then loop over all the queues and choose the queue that minimizes the absolute difference between the cost ratio and corresponding dequeue rate ratios. In other words, we try to find

$$Q_{min} = \underset{q \in \mathbb{Q}}{\text{argmin}} \left\| \frac{\text{cost}(op)}{\text{cost}(B)} - \frac{w_q}{w_B} \right\|$$

where w_B is the rate of the queue to which B is assigned.

Algorithm 1: Opset queue assignment algorithm.

Data: opsets, queues

1 R = range factor;

// bottleneck opset

2 $bn = \underset{op \in opsets}{\text{argmax}}\ cost(op)$;

3 $\tau_{next} = 1$;

4 $q_{bn} = queues[0]$;

5 **for** q *in queues* **do**

  // Find the queue for the bottleneck opset

6   **if** *cost(bn)* $\geq \tau_{next}$ *and cost(bn)* $< \tau_{next} * R$ **then**

7     $q_{bn} = q$;

8   break;

9   $\tau_{next} = \tau_{next} * R$;

// Assign bottleneck opset to queue

10 tagPriority(B, q_B);

// Find and assign queues to non-bottleneck opsets

11 **for** *op in opsets* **do**

12   **if** *op* == *bn* **then**

13   continue;

14   costRatio = cost(op)/cost(bn);

15   q_{op} = null;

16   minError = arbitrarily high value (e.g. 10^9);

17   **for** q *in queues* **do**

18   rateRatio = $w_q / w_{q_{bn}}$;

19   error = $abs(costRatio - rateRatio)$;

    // Find the queue with min. diff between cost and rate ratios

20   **if** $error < minError$ *and* $w_{q_{bn}} \geq w_q$ **then**

21   minError = error;

22       $q_{op} = q$;

23   tagPriority(op, q_{op});

The intuition behind this is that these opsets are likely to complete before the bottleneck. As such, we opt to assign them to a lower priority queue to slack them in a manner that is proportionate to the ratio between their cost and the bottleneck's cost. This assignment captures the properties of our second priority policy – SDS – which is mainly designed to synchronize the completion times of the different opsets (thereby reducing response time variability).

Sensitivity analysis. As with any system with tunable parameters, a primary concern is to determine the reduction in performance if these parameters are not configured optimally. To address this concern, we resort to trace-driven simulations to conduct a sensitivity analysis of our main parameters. We used the SoundCloud trace to generate our workload and the same settings as in Section 5.3. The primary goal of this analysis is to assess how changing the number of queues (K) and the range factor (R) – our two main configurable parameters – can affect the attained latency reductions. In both experiments, we set the initial range to one and the default values of K and R to four and three respectively. We observed minimal changes for the median and 95th percentiles, so we exclude their plots for brevity. In Figure 8, we plot the normalized performance degradation (%) experienced compared to the optimal settings for different values of K and R. We see that varying the number of queues can impact the 99th percentile latency by up to 25%, whereas varying the range factor changes it by up to 15%. The biggest performance drop occurs at the setting with the low number of queues and smallest range where this can be expected (the benefits of the multi-level queue are effectively being removed). More importantly, however, the performance drop in the 1-hop neighborhood of the optimal setting is a more manageable 8%. As such, even in unfavorable settings, Rein is able to realize reasonable performance gains with only minor hits at the higher percentiles.

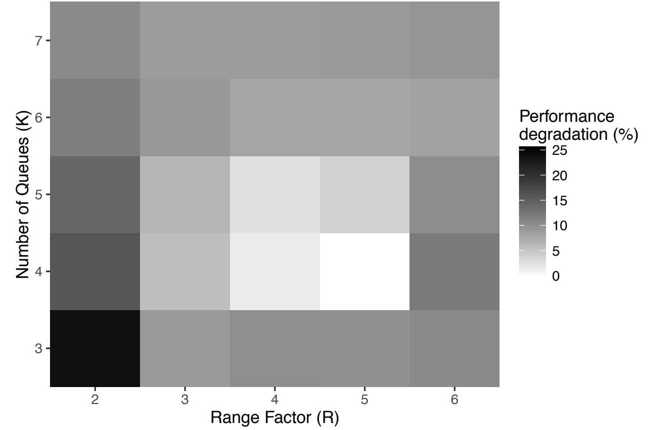


Figure 8: Sensitivity analysis of the range factor (R) and the number of queues (K), showing that the performance drop is about 8% in the 1-hop neighborhood of the optimal setting.

Based on these results, it is important to note that, while having more queues can provide increased control over scheduling, it can result in queue load imbalances if the average number of operations entering each queue is not carefully accounted for. We explored different combinations of these parameters through simulations and our best performing configuration was $K = 4$ with queue rates increasing in increments of 1. We also found the optimal value for both R and r_{Q_1} to be 1 and 5 respectively. However, we believe that these parameters are dependent on the workload and, as such, need to be tuned according to the target workload to reach optimal results. We leave developing adaptive algorithms for tuning these parameters for future work.

4. Implementation

Rein is implemented in Cassandra, a widely used distributed database offering a key-value interface and multiget operations. We used version 2.2.4 of Cassandra. For every request

sent to Cassandra, each node in the cluster can act as a coordinator, server or both based on the utilized internal routing mechanism. Cassandra itself is based on a Staged Event-Driven Architecture (SEDA) [49]. Each stage performs different functions and has its own thread pool as well as queue for all the outstanding operations. The stages communicate among each other via a messaging service.

We considered two different thread pools that are relevant to Rein's implementation; these are the native-transport and the read thread pools. The former is responsible for reading incoming requests from the TCP socket buffer. The latter handles the assignment of threads for coordinating the actual read operation on the target server. Rein's scheduling algorithms are implemented in the read thread pool queue. Note that Rein cannot be implemented in the native-transport queues as this is where requests are being parsed. As such, there is no notion of priorities at that stage and the requests can only be handled in FIFO order.

Rein's priority assignment takes place at the coordinator. Since these nodes are responsible for dispatching operations to the servers handling the operations' partitions, we can count the number of operations going to each partition and assign our priorities accordingly. As per Cassandra's code path that handles reads, these operations can either happen locally (if the data is present on the coordinator node itself) or are sent to other nodes. If data is replicated, Cassandra uses a snitch module to load balance the requests across the different replicas. However, since our priorities are assigned based on the number of operations headed to a certain partition (i.e., the opset size) and not a replica, our approach is logically separated from the load-balancing scheme employed by Cassandra.

5. Evaluation

We evaluate the effectiveness of our approach compared to a baseline and other latency reduction techniques. We also leverage simulations for running higher-scale experiments and to test our system for a wider variety of configurations and workloads. Our main results are as follows:

(i) We evaluate the performance of Rein in a real system under both realistic and synthetic workloads. We find that Rein substantially outperforms all other approaches that we ran against it; e.g., at the 99th-ile at 75% utilization, Rein reduces latency of multiget requests by about a factor of two. Moreover, our results show that request duplication (both straightforward and speculative) is ineffective in reducing the tail latency because it simply doubles the load offered to the system.

(ii) We assess the overheads of using single priority queues and establish the need to use multi-level queues.

(iii) We test Rein under higher-scales and under a wide variety of workload conditions and corner cases, observing that its performance is in several cases close to an idealized clairvoyant oracle strategy.

5.1 Effectiveness of Rein

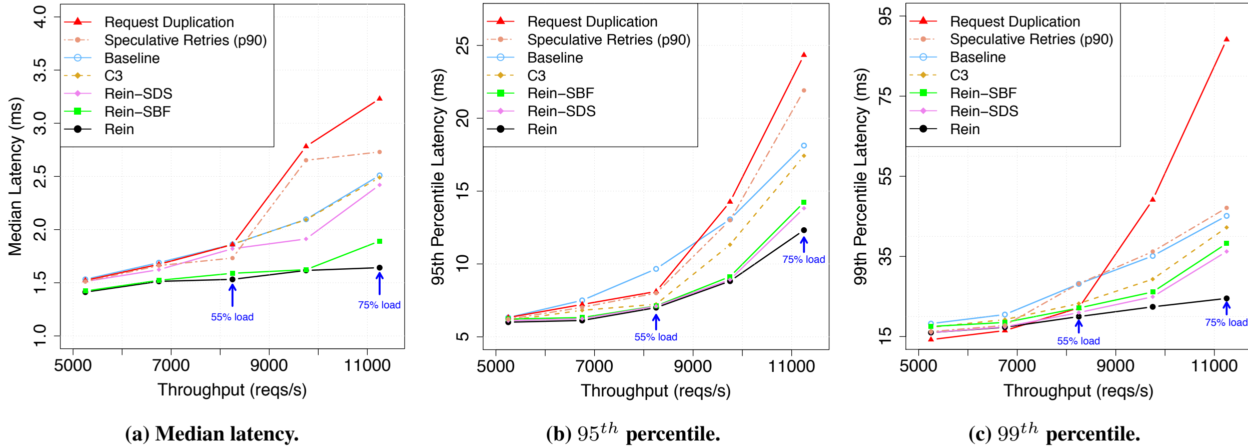
We first evaluate Rein in a realistic testbed using production workloads. For these experiments, we use 16 m3.xlarge AWS EC2 instances. To generate our workloads, we used a modified version of the Yahoo! Cloud Serving Benchmark (YCSB) [21] – a general-purpose cloud systems benchmarking tool – and configured it to run on a separate node. The instances each have a total of 15 GB of memory, 2x40 GB SSD, and 4 vCPUs. We insert data items (also called rows) into Cassandra with value sizes generated following the distribution of Facebook's Memcached deployment [13]. Our experiments focus on evaluating the effectiveness of Rein at different operational conditions. In all runs, we set the replication factor of our partitions to three and the concurrency level, or how many requests a can node serve simultaneously, to eight. The consistency level for all requests is set to one. We also disable the automatic paging feature in Cassandra to make sure our queries are not sent in a sequential manner.

Realistic workloads. We first evaluate how Rein performs under different system load levels. We use YCSB to generate multiget requests based on our trace (described in Section 2.2). For this scenario, we first insert a dataset composed of 100,000 rows, such that the entire dataset can fit into memory. As such, most of the reads are satisfied through the operating system's page cache (since Cassandra delegates memory management to the operating system). Our first set of experiments uses 500 closed-loop YCSB threads. We first test the cluster and find the maximum attainable throughput, which is $\sim$15,000 requests/sec. Note that these are multiget requests, which fetch 8.6 keys on average. As such, our cluster is actually serving around 129,000 read operations/s for this workload. We then cap YCSB's sending rate for different experiments to evaluate the performance under different system utilization levels.

We compare six different latency reduction techniques against the baseline, which uses Cassandra's default settings. Our results are presented in Figure 9.

We evaluate the effectiveness of request duplication, which involves pre-emptively sending an additional request for each query. This allows the client to receive the fastest returning response. Although such a technique has long been used to reduce latency at the tail, it is not universally applicable across a wide-variety of settings. In fact, in our experiments, once we go higher than 55% system utilization, the tail latencies are greatly inflated and become twice as high as the baseline at 75% utilization. This is because sending an extra request essentially doubles the demand on the servers, which degrades performance if the cluster is already saturated, confirming the results in [48].

We also evaluate speculative retries, which is another popular latency-reduction technique that is a more general



(a) Median latency. (b) 95th percentile. (c) 99th percentile.

Figure 9: Latency attained by the different variants of Rein compared to other latency reduction techniques. The x-axis represents the offered load. We see that Rein's approach achieves the highest gains in the median as well as high percentile latencies.

case of request duplication. In this case, the coordinator triggers another request once a response is delayed past a predefined timeout. We have tested many different timeout values and found that setting it to the 90th percentile of the response time provides the most favorable results. We can see that, given its less aggressive nature, speculative retries provides modest benefits at the tail (between 15% and 25% reduction) but suffers the same problems as request duplication once our saturation level exceeds the midpoint (albeit with lower performance degradation than straightforward duplication).

In addition to these techniques, we also tested a modified version of C3 [44] – an optimized dynamic load-balancer – to see how it fairs against Rein. C3 is composed of a replica ranking system (which chooses which server should answer a request) as well as a distributed rate-limiter. We removed the rate-limiter, as we found that its Akka-based [1] implementation was causing performance bottlenecks in our experimental settings. As seen in the results, C3 manages to provide tangible benefits at the tail (upwards of 25%) while not suffering any performance problems at the higher utilization levels. However, we can also observe that there are little to no improvements in the median latency, which we believe is a consequence of the fact that C3 was optimized primarily for the tail.

Lastly, we experiment with 3 different variations of Rein. Rein-SDS and Rein-SBF use the pure forms of Slack-Driven Scheduling and Shortest Bottleneck First, respectively. We first quantify the gains of each of the policies to discern whether a combination of both is indeed superior. As we can see, both approaches provide increasingly higher gains as we move towards higher utilization levels. This is because as the load increases, the system experiences higher burstiness, which temporarily causes requests to queue. In turn, this provides our scheduling heuristics with greater opportunities to re-order the execution of these operations and im-

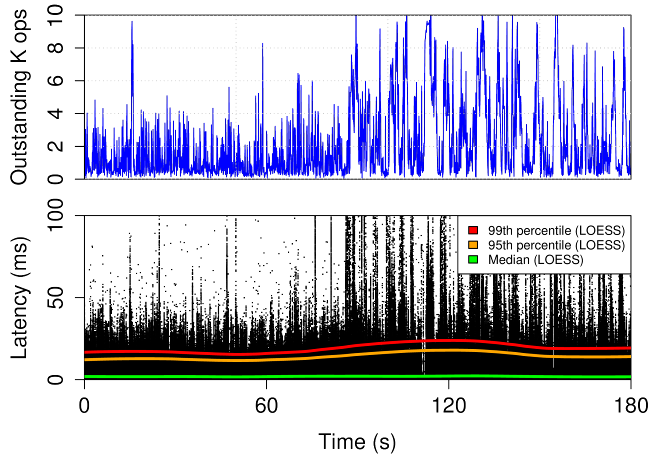
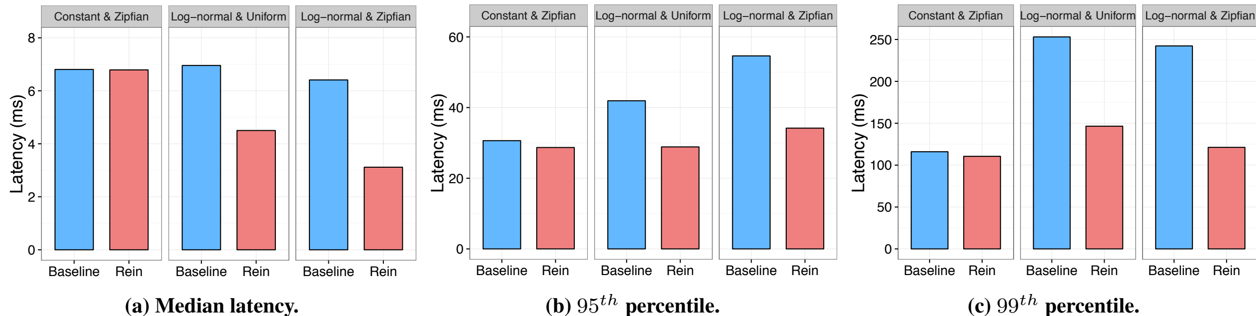


Figure 10: Time series showing the multiget latencies as well as the number of outstanding operations during a three-minute run. We also plot the moving median, 95th, and 99th percentiles, which we smoothen using LOESS regression. The median, 95th, and 99th percentiles are averaging at around 1.9, 14.3, and 19.2 ms respectively. Points exceeding 100 ms latency are not shown for readability reasons.

prove tail latency. We can see that both variations can provide upwards of 50% reduction at the 95th and 99th percentiles at 75% utilization. In addition, Rein-SBF, which is optimized for reducing average makespans, is able to cause a sizable reduction in the median latencies, reaching up to 30% at higher load levels. Despite the fact that SBF relies on prioritizing requests with shorter bottlenecks at the cost of delaying larger requests, it is still effective at reducing higher percentile latencies due to the heavy-tailed nature of our multiget sizes. In essence, delaying very large requests has no negative impact on the 99th percentile but can, counterintuitively, even improve it. In summary, both variations provide much higher gains at different configurations than



(a) Median latency. **(b) 95th percentile.** **(c) 99th percentile.**

Figure 11: Latency comparison of Rein versus the baseline using different synthetic workloads at 75% utilization.

the other techniques we compare against. We can also see that Rein is able to outperform the two policies, which shows that a combination of both policies is in fact better for both the median and tail latencies.

To investigate the level of burstiness of our workload generation and to develop a deeper understanding of response time characteristics, we plot in Figure 10 the time series of the number of outstanding operations (from the point of view of YCSB) sampled at 100 ms intervals as well as the multiget latencies during a three-minute interval of Rein at steady state. We also plot the moving median, 95th, and 99th percentiles for the multiget latencies, which are smoothened using local regression (LOESS) [20] with a span parameter of 0.3. We can see that the number of outstanding read operations can vary by as much as 10,000 operations during the experiment with a standard deviation of 2.38, which is mostly due to the heavy-tailed nature of multiget sizes in our workload. We also observe latency spikes, which increase in frequency and magnitude during periods of higher loads (e.g., after the 80 s mark). However, despite this, both the 95th and 99th percentiles remain relatively stable throughout the experiment.

Synthetic workloads. We also evaluate Rein against a wide variety of workloads to assess its generality and to ensure that our solution is not tailor-fitted to our trace. To formulate these workloads, we focus on two key features: multiget size distribution and the access patterns.

For multiget distributions, we want to observe the behavior of Rein against both short- and heavy-tailed workloads. For the former, we simply use a constant multiget size of 50, which corresponds to the fan-out factor of requests reported in Bing's cluster [10]. We use a constant multiget size to assess the performance of Rein in a challenging scenario in terms of latency reduction opportunities due to the uniformity of multiget size. Secondly, we generate a heavy-tailed multiget size distribution by fitting a log-normal curve to the distribution of multiget sizes reported inside Facebook's memcached production clusters [37]. The values of μ and σ of the resulting distribution are set at 2.5 and 1.25, respectively.

For the key popularity distribution, we configure YCSB to use Zipfian access patterns to emulate skewed reads (where certain keys are orders of magnitude popular than others). In addition, we also use uniform access patterns in which keys are equally likely to be requested in a multiget.

We then combine the above workload characteristics to generate three distinct workloads.[2] The three workloads have multiget size and access pattern distributions as follows: *Constant & Uniform*, *Log-normal & Uniform*, and *Log-normal & Zipfian*. We evaluate Rein performance with these workloads at a throughput of 900, 1,700, and 1,900 requests/s, respectively. This equates to around 75% system utilization for each of the specified workloads.

In Figure 11, we can see the performance of Rein compared to the baseline for varying multiget sizes and key access patterns. In the scenario with constant multiget size and Zipfian access pattern, head-of-line blocking is minimized due the uniformity of multiget sizes. As such, Rein achieves modest performance gains of 6% and 5% at the 95th and 99th percentile, respectively. In contrast, having heavy-tailed, log-normal distributed multiget sizes allows Rein to attain gains with the SBF aspect of our scheduling scheme. Since the SBF policy is designed to reduce head-of-line blocking, it exploits non-uniform request sizes to attain performance gains. When using a uniform access pattern, we observe a 35% reduction at the median and up to 30% and 42% at the 95th and 99th percentiles, respectively. Finally, when using a workload with log-normal distributed multiget sizes and Zipfian access pattern, Rein achieves higher performance gains with a 1.6x reduction at the 95th percentile and a roughly two-fold reduction at both the median and 99th percentile latencies.

SSD-heavy reads. In addition to memory-heavy settings, we evaluate Rein's performance with SSD-heavy reads in high-load settings. To do this, we inserted a dataset composed of 300 million rows, such that only approximately one-third of the data can fit into the operating system's

[2] We omit the workload with constant multiget size and uniform access patterns since it does not offer opportunities for scheduling-based optimizations.

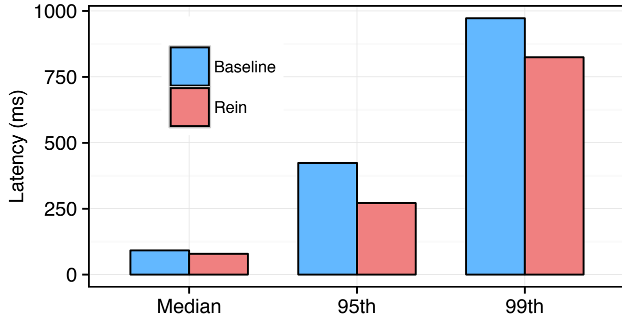


Figure 12: Latency comparison of Rein versus the baseline for SSD-heavy reads. The dataset size was adjusted such that the nodes can only cache one-third of the stored rows.

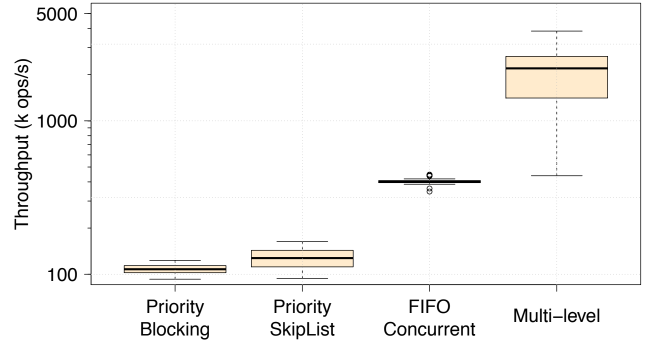


Figure 13: Throughput benchmark comparing different queue implementations.

page cache. In this scenario, most of the reads are going to be performed on the instance's SSD. As seen in Fig. 12, Rein improves the 95th and 99th percentiles by up to 35% and 15%, respectively, with only minor benefits at the median. These results, while still a significant improvement, are likely smaller than in the cache-heavy read scenarios, due to Rein's inability to predict whether a requested value exists in-memory or on SSD. Since one-third of the dataset can exist inside the cache, our ability to predict bottlenecks can be compromised since our scheduler determines bottleneck opsets solely based on the number of operations inside them. We leave how to improve our bottleneck prediction given the interactions between multiple caching layers for future work.

5.2 Performance of Single-Priority Queues

Next, we evaluate the performance of different implementations of priority queues by measuring their throughput under high-load settings. Namely, we use Java's Priority Blocking Queue as well as a concurrent Priority SkipList Queue that is based on Lotan and Shavit's design [41]. We compare them with our multi-level queue (configured with $N = 5$). In addition, we also include a comparison to a concurrent FIFO queue – namely, Java's Concurrent Linked Queue, which is based on Michael and Scott's algorithms [35] – and use it as a baseline.

We measure the throughput of these data structures (in a similar manner to [43]) by using 32 threads that either enqueue or dequeue elements from the target queue with a 50% probability. The priorities of inserted items are integers chosen uniformly at random between 1 and 5. Each thread generates a total of 100,000 operations.

We run these experiments on a single machine with 128 GB of RAM and an Intel Xeon E5-2640v3 with 16 physical cores at 2.60 GHz with hyper-threading enabled. We repeat each experiment 50 times with different random seeds.

Figure 13 shows the results. Priority Blocking Queue and Priority SkipList Queue obtain on average around 110,000 and 130,000 ops/s, respectively. On the other hand, Concurrent Linked Queue achieves much higher throughput as it does not incur the overheads of maintaining a priority order for its elements. In comparison, multi-level queue provides much higher throughput, at roughly 2,000,000 ops/s, which is one order of magnitude higher than the other priority queue implementations and a five-fold improvement over the non-blocking FIFO queue.

The reason multi-level queue outperforms traditional priority queue implementations is that it substantially reduces lock-contention, thanks to the fact that it does not need to maintain a fine-grained priority order. Multi-level queue also has a significant performance advantage over FIFO queue despite being based on the same data structures and incurring the overhead of performing scheduling rounds for Deficit Round Robin (DRR). The reason is that, while the Concurrent Linked Queue is indeed lock-free by relying on compare-and-swap (CAS), it does not eliminate the contention between multiple producers trying to enqueue elements concurrently. Multi-level queue reduces this type of contention due to the fact that the threads' activity is spread over multiple queues — five in this case. The relatively higher variance of multi-level queue is primarily due to the workload generated, which differs across experiments. The more balanced the workload on the target queues, the lower the contention. However, even at its minimum, the performance of multi-level queue is still higher than that of FIFO queue. This shows that in multi-consumer/multi-producer settings, multi-level queue is likely a better choice than lock-free priority queues and even concurrent FIFO queues. This finding supports our adoption of multi-level queue for realizing Rein in highly concurrent systems such as key-value stores.

5.3 Simulations

We now turn to simulations to assess the performance of our techniques at larger scales. We built our simulator in Python using a discrete-event simulation framework called SimPy. We evaluate the performance of Rein's different multi-level queue policies and compare them to an oracle (idealized)

approach as well as the baseline policy (i.e., FIFO). For the purposes of load-balancing requests across servers, we use a straw-man approach, where replicas are chosen for each operation using round-robin.

We simulate a system with 128 clients and 128 servers at a concurrency level of eight (i.e., they can service up to eight operations in parallel), each operating with a replication factor of three at an average service rate of 3,750 requests/s. We set our one-way network latency to 50 μs. Similarly to the real experiments, we drive our workload using the SoundCloud trace. We also generate the value sizes for the requests using the same distribution that we used in our previous experiments. We then generate request inter-arrival times using a Poisson process in which the mean rate is set to match 75% of system capacity. The experiments are then repeated six times with different random seeds.

In addition to using the different variations of Rein, we also use a clairvoyant oracle. For this method, the client has complete knowledge of the system's state. It knows the instantaneous load on all servers (i.e., queue sizes) as well as their service rates. In addition, it can also observe all the in-flight requests that are sent from each client in the system and their target servers. Using this information, the oracle can construct a more idealized cost estimation that not only considers the opset sizes for a request but also the present as well as the future load on the target servers; the latter of which is calculated by counting all the in-flight requests. This allows the client to have a more accurate prediction of the slack as well as the bottleneck, which results in better scheduling decisions. Using an oracle allows us to assess the performance gap that we experience by adopting a completely decentralized scheduling protocol in Rein.

Figure 14 shows the read latencies at the median, and 95th and 99th percentiles averaged across the six runs. The standard deviation is not shown as it is largely negligible. As shown, Rein outperforms the baseline scenario across all percentiles and improves the latencies by up to a factor of two at the median, and 95th and 99th percentiles. In addition, Rein is within 7% of the performance attained by the oracle, which has the advantage of having global state information that allows it to make better scheduling decisions. Despite Rein's lack of coordination, we are still within reach of the performance of a fully coordinated system.

6. Discussion

How generic is Rein? Rein uses a system model that is commonly employed in distributed key-value stores. Its scheduling policies do not leverage any system-specific parameters and are logically separated from the load-balancing algorithms. As such, the same scheduling heuristics can be applied to a number of key-value stores (e.g., Memcached, MongoDB, Redis) without any customization.

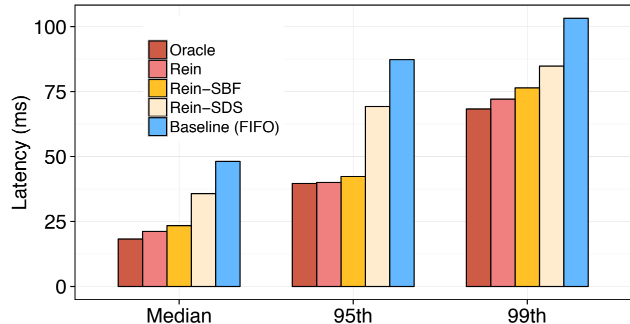


Figure 14: Latency comparison of Rein's multi-level queue policies versus the oracle and FIFO baseline.

What is the overhead of using Rein? Our scheduling heuristics are simple. At the client-side, for every request, we count the number of operations in each opset and calculate the maximum operation count (to determine the bottleneck). No running statistics or other types of complicated mathematical procedures are performed. To put this into context, Cassandra's dynamic snitch requires considerably more complex operations to be performed for ranking the replicas, as it maintain exponentially decaying reservoirs of latencies toward the different backends. As such, we consider our approach to be minimalistic in this regard.

Why is Rein effective despite being static? Even though Rein does not account for system state, its lack of dynamism does not detract from its viability. For larger multigets – which generate 100s to 1000s of operations and for which our scheduling decisions matter the most – their response times are mostly dominated by the service times of their operations. As such, despite the various sources of variability (e.g., variable waiting times, service rate variations, network latency spikes, etc.), our statically-derived cost estimate remains valid. However, as part of our ongoing work, we are modifying the SDS approach to incorporate feedback information from the servers pertaining to the sizes of the different priority queues (in the form of running averages). Preliminary results have shown reasonable gains from applying this approach.

Does Rein work at different consistency levels? In the evaluation, we maintained a consistency level of *one* for all our read requests (that is, each operation requires a response from only one replica, e.g., the server chosen by Cassandra's snitch). This setting is quite commonplace in large Web services today [29, 46]. It remains to be seen how Rein can perform at higher consistency levels; however, it is important to note that increasing the consistency level also increases the number of operations being sent to the replicas, which would provide more opportunities for our scheduling policies to exploit.

Can Rein be applied to low-latency key-value stores? Ultra-fast key-value stores [33, 7] have been gaining traction recently. They usually employ shared-nothing architectures,

where each core is assigned to a partition and there exists no single-point of contention in the system. Doing this effectively eliminates context-switching overhead and reduces operation latency to the order of microseconds. However, these ultra-low services times can create challenges since the kernel could become a bottleneck [30]. The implications for Rein is that most of the queuing in the system could end up happening at the kernel level. This would put our system at a disadvantage since Rein operates strictly on the application layer. However, kernel-bypass techniques – which are also increasing in popularity [30, 33] – can be leveraged to effectively merge kernel-level with application-level queues, allowing us to realize the full benefits of Rein.

7. Related Work

There is a large body of work in the literature on the problem of reducing tail latencies for distributed storage systems. Load-balancing techniques [36, 44] have been employed to make replica server selections in replicated settings. Other systems techniques such as speculative reissues and duplication of requests [28, 48, 52] have also been used to reduce latency at the tail. However, all these approaches offer optimizations that work on the granularity of requests (or operations). Other approaches that perform adaptations at longer time-scales have proposed selective replication of data [19, 47], tuning the placement of data according to keys accessed frequently [22, 38], configuring priorities and rate limits across multiple systems stages (e.g., network and storage) [54], and batching requests to adapt to variation in storage-layer performance [34]. None of these works addresses the added dimension of multiget workloads and how to leverage knowledge about their structure to optimize scheduling at the backends.

Adaptive parallelization techniques [27, 32] have also been used for latency reduction in adaptive server systems. Haque et al. [27] uses dynamic multi-threading to reduce tail latencies by keeping track of the progress of request execution times and increasing the level of parallelism the longer the request stays in the system. To determine the level of parallelism, they profiled the workload and hardware resources offline and computed a policy. The requests then decide online on their level of parallelism based on the computed policy, the system load level, and their own progress. Li et al. [32] generalized this approach and aimed to reduce the number of requests that miss a user-defined target latency. They serialized large requests in the system to reduce the impact of queuing delay on the smaller requests which is a form of work-stealing akin to our Shortest Bottleneck First (SBF) policy. Both the aforementioned works, however, focus on optimizing execution per server. They do not deal with the problem of scheduling requests across different servers. In addition, they do not explicitly target key-value stores, which offer their own set of challenges.

There is also work in the literature that has some commonalities with Rein but is otherwise applied to an entirely different domain — namely, network flow scheduling. Baraat [25] is a decentralized co-flow scheduler that primarily utilizes FIFO scheduling, but avoids head-of-line blocking by dynamically modifying the multiplexing level in the network. Varys [18] – a network scheduling system for data-intensive frameworks – utilizes a combination of Smallest-Effective-Bottleneck-First (SEBF) to minimize flow completion times and Minimum-Allocation-for-Desired-Duration (MADD) to decide the rate allocation for each flow in a way that slows down all flows to match the longest flow. Aalo [17], similarly to Varys, operates on the co-flow scheduling problem but does not assume a priori knowledge of co-flow sizes. To work around this lack of clairvoyance, its scheduling algorithm involves using multiple queues with different weights and assigns flows dynamically from higher priority to lower priority queues based on the amount of data that each flow has accrued. While these approaches have some commonalities with Rein, they tackle a different set of problems with their own list of challenges. To the best of our knowledge, no other work has considered the benefits of task-aware scheduling for multiget workloads within the context of distributed key-value stores.

8. Conclusion

In this paper, we propose scheduling techniques that take advantage of the structure of multiget requests in real-world data store workloads to reduce aggregate median and tail latencies. Under heavy loads, our scheduling algorithms reduced the median, 95th, and 99th percentile latencies by factors of 1.5, 1.5, and 1.9, respectively. Rein demonstrates that distributed scheduling can provide significant benefits in the context of key-value stores without requiring coordination. While our evaluation focused on Cassandra, because our solution is non-intrusive and relatively straightforward to implement, it should be easy to apply to other systems.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Kathryn McKinley for their feedback. We are grateful to Mohammad Alizadeh, Florin Ciucu, Mosharaf Chowdhury, Juan Perez, and Simon Peter for their valuable comments and suggestions on earlier drafts of this work. Waleed Reda was supported by a fellowship from the Erasmus Mundus Joint Doctorate in Distributed Computing (EMJD-DC) program funded by the European Commission (EACEA) (FPA 2012-0030). This project is in part financially supported by the Swedish Foundation for Strategic Research. In addition, this work was partially supported by the Wallenberg Autonomous Systems Program (WASP).

References

[1] Akka. http://akka.io/. (Cited on page 10.)

[2] Apache Cassandra. http://cassandra.apache.org/. (Cited on pages 1 and 2.)

[3] Elasticsearch. https://www.elastic.co/products/elasticsearch. (Cited on pages 1 and 2.)

[4] Memcached. https://memcached.org/. (Cited on page 2.)

[5] MongoDB. https://www.mongodb.com/. (Cited on page 2.)

[6] Redis. http://redis.io/. (Cited on page 2.)

[7] ScyllaDB. http://scylladb.com/. (Cited on page 13.)

[8] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. In *EuroSys*, 2013. (Cited on page 5.)

[9] D. Alistarh, J. Kopinsky, J. Li, and N. Shavit. The SprayList: A Scalable Relaxed Priority Queue. *ACM SIGPLAN Notices*, 50(8):11–20, 2015. (Cited on page 6.)

[10] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data Center TCP (DCTCP). In *SIGCOMM*, 2010. (Cited on pages 1, 3 and 11.)

[11] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. PACMan: Coordinated Memory Caching for Parallel Jobs. In *NSDI*, 2012. (Cited on pages 1 and 3.)

[12] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the Outliers in Map-reduce Clusters Using Mantri. In *OSDI*, 2010. (Cited on page 5.)

[13] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of a Large-scale Key-value Store. In *SIGMETRICS*, 2012. (Cited on pages 2 and 9.)

[14] N. Bansal and M. Harchol-Balter. Analysis of SRPT Scheduling: Investigating Unfairness. In *SIGMETRICS*, 2001. (Cited on page 6.)

[15] O. Boxma and B. Zwart. Tails in Scheduling. *ACM SIGMETRICS Performance Evaluation Review*, 34(4):13–20, 2007. (Cited on page 6.)

[16] D. Chakrabarty, Y. Zhou, and R. Lukose. Budget Constrained Bidding in Keyword Auctions and Online Knapsack Problems. In *WINE*, 2008. (Cited on page 4.)

[17] M. Chowdhury and I. Stoica. Efficient Coflow Scheduling Without Prior Knowledge. In *SIGCOMM*, 2015. (Cited on pages 7 and 14.)

[18] M. Chowdhury, Y. Zhong, and I. Stoica. Efficient Coflow Scheduling with Varys. In *SIGCOMM*, 2014. (Cited on page 14.)

[19] R. G. Christopher Stewart, Aniket Chakrabarti. Zoolander: Efficiently Meeting Very Strict, Low-Latency SLOs. In *ICAC*, 2013. (Cited on page 14.)

[20] R. A. Cohen. An Introduction to PROC LOESS for Local Regression. In *SUGI*, 1999. (Cited on page 11.)

[21] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *SoCC*, 2010. (Cited on pages 2 and 9.)

[22] S. Das, D. Agrawal, and A. El Abbadi. G-Store: A Scalable Data Store for Transactional Multi Key Access in the Cloud. In *SoCC*, 2010. (Cited on page 14.)

[23] J. Dean and L. A. Barroso. The Tail At Scale. *Communications of the ACM*, 56:74–80, 2013. (Cited on pages 1 and 2.)

[24] G. Decandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *SOSP*, 2007. (Cited on page 1.)

[25] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron. Decentralized Task-Aware Scheduling for Data Center Networks. In *SIGCOMM*, 2014. (Cited on pages 2 and 14.)

[26] L. Flatto and S. Hahn. Two Parallel Queues Created by Arrivals with Two Demands I. *SIAM Journal on Applied Mathematics*, 44(5):1041–1053, 1984. (Cited on page 5.)

[27] M. E. Haque, Y. hun Eom, Y. He, S. Elnikety, R. Bianchini, and K. S. McKinley. Few-to-Many: Incremental Parallelism for Reducing Tail Latency in Interactive Services. In *ASPLOS*, 2015. (Cited on page 14.)

[28] V. Jalaparti, P. Bodik, S. Kandula, I. Menache, M. Rybalkin, and C. Yan. Speeding up Distributed Request-Response Workflows. In *SIGCOMM*, 2013. (Cited on pages 1 and 14.)

[29] C. Kalantzis. Eventual Consistency != Hopeful Consistency, talk at Cassandra Summit, 2013. https://www.youtube.com/watch?v=A6qzx_HE3EU. (Cited on page 13.)

[30] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat. Chronos: Predictable Low Latency for Data Center Applications. In *SOCC*. ACM, 2012. (Cited on page 14.)

[31] G. Kumar, G. Ananthanarayanan, S. Ratnasamy, and I. Stoica. Hold 'em or Fold 'em? Aggregation Queries under Performance Variations. In *EuroSys*, 2016. (Cited on page 3.)

[32] J. Li, K. Agrawal, S. Elnikety, Y. He, I.-T. A. Lee, C. Lu, and K. S. McKinley. Work Stealing for Interactive Services to Meet Target Latency. In *PPoPP*, 2016. (Cited on page 14.)

[33] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *NSDI*, 2014. (Cited on pages 13 and 14.)

[34] J. C. McCullough, J. Dunagan, A. Wolman, and A. C. Snoeren. Stout: An Adaptive Interface to Scalable Cloud Storage. In *ATC*, 2010. (Cited on page 14.)

[35] M. M. Michael and M. L. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *PODC*, 1996. (Cited on page 12.)

[36] M. Mitzenmacher. The Power of Two Choices in Randomized Load Balancing. *IEEE Trans. Parallel Distrib. Syst.*, 12(10):1094–1104, Oct. 2001. (Cited on page 14.)

[37] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee,

H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *NSDI*, 2013. (Cited on pages 1, 2, 3 and 11.)

[38] J. Paiva, P. Ruivo, P. Romano, and L. Rodrigues. AUTOPLACER: Scalable Self-Tuning Data Placement in Distributed Key-value Stores. In *ICAC*, 2013. (Cited on page 14.)

[39] H. Rihani, P. Sanders, and R. Dementiev. MultiQueues: Simpler, Faster, and Better Relaxed Concurrent Priority Queues. *CoRR*, abs/1411.1209, 2014. (Cited on page 6.)

[40] T. A. Roemer. A Note on the Complexity of the Concurrent Open Shop Problem. *J. of Scheduling*, 9(4):389–396, Aug. 2006. (Cited on page 1.)

[41] N. Shavit and I. Lotan. Skiplist-Based Concurrent Priority Queues. In *IPDPS*, 2000. (Cited on page 12.)

[42] A. Shieh, S. Kandula, A. Greenberg, C. Kim, and B. Saha. Sharing the Data Center Network. In *NSDI*, 2011. (Cited on pages 1 and 3.)

[43] H. Sundell and P. Tsigas. Fast and Lock-Free Concurrent Priority Queues for Multi-Thread Systems. *Journal of Parallel and Distributed Computing*, 65(5):609–627, 2005. (Cited on page 12.)

[44] L. Suresh, M. Canini, S. Schmid, and A. Feldmann. C3: Cutting Tail Latency in Cloud Data Stores via Adaptive Replica Selection. In *NSDI*, 2015. (Cited on pages 2, 10 and 14.)

[45] E. Varki, A. Merchant, and H. Chen. The M/M/1 Fork-Join Queue with Variable Sub-Tasks. `http://www.cs.unh.edu/~varki/publication/2002-nov-open.pdf`, 2002. (Cited on page 5.)

[46] V. Venkataramani, Z. Amsden, N. Bronson, G. Cabrera III, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, J. Hoon, S. Kulkarni, N. Lawrence, M. Marchukov, D. Petrov, and L. Puzar. TAO: How Facebook Serves the Social Graph. In *SIGMOD*, 2012. (Cited on pages 1 and 13.)

[47] H. T. Vo, C. Chen, and B. C. Ooi. Towards Elastic Transactional Cloud Storage with Range Query Support. *Proc. VLDB Endow.*, 3(1-2), Sept. 2010. (Cited on page 14.)

[48] A. Vulimiri, P. B. Godfrey, R. Mittal, J. Sherry, S. Ratnasamy, and S. Shenker. Low Latency via Redundancy. In *CoNEXT*, 2013. (Cited on pages 9 and 14.)

[49] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *SOSP*, 2001. (Cited on page 9.)

[50] A. Wierman and B. Zwart. Is Tail-Optimal Scheduling Possible? *Operations Research*, 60(5):1249–1257, Sept. 2012. (Cited on page 2.)

[51] M. Wimmer, F. Versaci, J. L. Träff, D. Cederman, and P. Tsigas. Data Structures for Task-based Priority Scheduling. In *PPoPP*, 2014. (Cited on page 6.)

[52] Z. Wu, C. Yu, and H. V. Madhyastha. CosTLO: Cost-Effective Redundancy for Lower Latency Variance on Cloud Storage Services. In *NSDI*, 2015. (Cited on page 14.)

[53] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *OSDI*, 2008. (Cited on pages 2 and 5.)

[54] T. Zhu, A. Tumanov, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger. PriorityMeister: Tail Latency QoS for Shared Networked Storage. In *SoCC*, 2014. (Cited on page 14.)