When to Hedge in Interactive Services

Mia Primorac, *Oracle Labs;* Katerina Argyraki and Edouard Bugnion, *EPFL*

https://www.usenix.org/conference/nsdi21/presentation/primorac

This paper is included in the
Proceedings of the 18th USENIX Symposium on
Networked Systems Design and Implementation.

April 12–14, 2021

978-1-939133-21-2

Open access to the Proceedings of the
18th USENIX Symposium on Networked
Systems Design and Implementation
is sponsored by

When to Hedge in Interactive Services

Mia Primorac
*Oracle Labs**

Katerina Argyraki
EPFL

Edouard Bugnion
EPFL

Abstract

In online data-intensive (OLDI) services, each client request typically executes on multiple servers in parallel; as a result, "system hiccups", although rare within a single server, can interfere with many client requests and cause violations of service-level objectives. Service providers have long been fighting this "tail at scale" problem through "hedging", *i.e.,* issuing redundant queries to mask system hiccups. This, however, can potentially cause congestion that is more detrimental to tail latency than the hiccups themselves.

This paper asks: when does it make sense to hedge in OLDI services, and how can we hedge enough to mask system hiccups but not as much as to cause congestion? First, we show that there are many realistic scenarios where hedging can have no benefit—where *any* hedging-based scheduling policy, including the state-of-the-art, yields no latency reduction compared to optimal load balancing without hedging. Second, we propose LÆDGE, a scheduling policy that combines optimal load balancing with work-conserving hedging, and evaluate it in an AWS cloud deployment. We show that LÆDGE strikes the right balance: first, unlike the state of the art, it never causes unnecessary congestion; second, it performs close to an ideal scheduling policy, improving the 99^{th} percentile latency by as much as 49%, measured on 60% system utilization—without any difficult parameter training as found in the state of the art.

1 Introduction

This work concerns Online Data-Intensive (OLDI) services like web search (searching through inverted document indices), content-based image similarity search, recommendation services, graph processing and social applications. Such services involve hundreds or thousands of "leaf" nodes, each holding a part ("shard") of the data needed to answer client requests; a tier of "root" nodes receives client requests, breaks each client request into distinct queries, forwards the queries to different leaves, and waits for the slowest query to finish in order to create the final client response.

OLDI services typically operate under hard-to-meet service-level objectives (SLOs) expressed in terms of tail latency [7, 8, 16, 18, 32, 36, 37, 40]. Each SLO captures a customer expectation and failing to meet it has concrete consequences, *e.g.,* a hit to the service provider's reputation, a loss of customers, and a drop in revenue [11, 15, 23, 73]. The nature of OLDI services makes meeting such SLOs challenging at large scale: because answering a client request involves many queries, a small fraction of slow queries can impact a significant fraction of client requests [16].

We focus on two main causes of latency variability: One is variable queuing delay, *e.g.,* due to load fluctuations [16, 20, 31, 62]. Another one is variable service time, which, in turn, comes from two distinct sources: (1) Different queries may take different amounts of time to execute on a given hardware and software stack, because of different complexities [33, 44, 53, 81]. (2) Different instances of the same query may take significantly different amounts of time to execute on a given system because of system events that are unrelated to the service itself: decisions made by an OS scheduler or power-management algorithm, interrupts, garbage collection, virtualization effects, interfering with other applications, *etc.,* [12, 16, 19, 26, 41–43, 46–48, 54, 57, 59, 69, 75, 80].

Even though a lot can and has been done to reduce latency variability, completely eliminating its causes has proved elusive. In a modern cloud environment, where different services share resources, there always exist unexpected performance "hiccups". Debugging these hiccups is notoriously hard. For instance, there is the case of an application suffering random 12*ms* scheduling delays, because a kernel feature caused the jitter of interrupt requests to be significantly higher than the timer interval [12]; or the case of non-work-conserving scheduling, in which the kernel was throttling programs that exceeded a misconfigured purchase quota [69]. But even when performance hiccups are easy to debug, fixing them is often beyond the control of the interested parties: most service providers do not own a datacenter and do not develop their

*The project was completed while the author was at EPFL.

own operating systems and entire software stacks—yet they still offer interactive services that need to meet strict SLOs.

This reality has given birth to *hedging*. In a modern large-scale service, each data shard is typically replicated in at least two nodes for fault tolerance [16, 55]. Hence, any query for a distinct shard can be replicated and sent to multiple nodes that serve that shard. This way, the system "hedges its bets", as it needs to wait only for the fastest response to each query. Hedging was initially proposed and adopted in combination with performance monitoring, to improve completion time of map-reduce-style [17] jobs that may take tens to hundreds of seconds [3–5]. More recently, hedging is proposed as a general solution for reducing tail latency in large-scale services, including OLDI services [16, 25, 32, 37, 39, 40, 64, 67, 72, 74, 78], where expected query completion time is orders of magnitude shorter. This change in time scale makes it significantly harder to determine whether hedging will improve tail latency or make it worse.

Hedging masks service-time variability at the cost of extra system load (caused by the replicated queries), hence extra queuing delay. So, if we take any standard hedging policy and any standard load-balancing policy (that tries to minimize latency without hedging), and we measure latency as a function of system load, we expect to see a tipping point: at first, hedging will outperform plain load balancing; however, for some offered system load, the cost of replicating queries will start to outweigh the benefit and the situation will be reversed. The challenge, then, is knowing *when* to hedge in order to operate before the tipping point.

In this paper, we look critically at hedging for OLDI services. We make two contributions:

- We study *when* hedging makes sense: when does it have the potential to improve tail latency relative to plain load balancing, and by how much? To answer this question, we define Idealized Hedge, a theoretical hedging policy that, by design, maximizes the hedging potential of any implementable hedging policy for a given setup. We experimentally compare Idealized Hedge to Per-Shard Queuing, which is, in our context, the best load-balancing policy that does not hedge [45, 76]. This allows us to identify regimes where hedging has the potential to improve tail latency, and to bound the potential improvement.
- We propose LÆDGE (short for "Load-Aware Hedge", pronounced like "*ledge*"), a combination of Per-Shard Queuing and hedging that hedges only if the current system load allows for latency improvement through hedging. The gist of LÆDGE is to only hedge when a replica is idle, through a work-conserving centralized scheduler.

Through a combination of simulations and experiments, we show that LÆDGE approximates Idealized Hedge and outperforms the state-of-the-art hedging policies. We implemented LÆDGE within an open-source OLDI framework [28] and evaluated it on a popular enterprise search engine deployed in the AWS cloud. Our experimental results closely match theoretical expectations and show significant gains: for jobs with mean service times as low as 800µs, and system utilization up to 60%, LÆDGE reduces the 99^{th} percentile latency of Per-Shard Queuing by, on average, 5.3*ms*—an improvement that corresponds to 6.4× the mean service time.

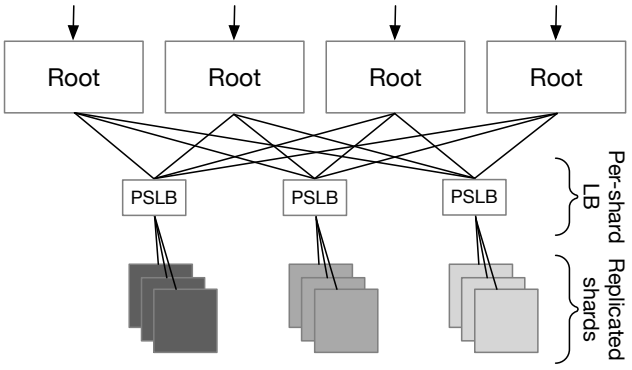


Figure 1: Architecture of an OLDI service with per-shard load balancers (PSLBs).

The rest of the paper is organized as follows: §2 presents our simulation setup and defines IQ-jitter—the metric we use to model and measure performance hiccups. §3 presents Idealized Hedge and uses it to identify when hedging has the potential to improve tail latency. §4 presents LÆDGE and evaluates it through simulation, while §5 evaluates it based on a system implementation using a real OLDI application deployed in the AWS cloud. §6 discusses open issues, §7 presents related work, and §8 concludes.

2 Background and Setup

In this section, we describe a standard OLDI setup (§2.1); define IQ-jitter, which helps us model performance hiccups (§2.2); summarize state-of-the-art hedging (§2.3) and load-balancing (§2.4) policies; and describe the simulation (§2.5) that drives the rest of the paper (up until §5).

2.1 OLDI Setup

We consider a cluster that serves OLDI applications, as illustrated in Fig. 1: a tier of root nodes serves client requests, while a tier of leaf nodes stores shards of application data sets; the root and leaf tiers are connected via a tier of per-shard load balancers (PSLBs). The load balancers are "per shard," in the sense that they maintain *a distinct queue per shard*. In practice, the same process implements multiple PSLBs; PSLB processes run either in servers that are dedicated to load balancing, or within the root nodes when that is permitted by the scheduling policy.

A client request requires accessing multiple data shards. When a root receives a client request, it issues one query per distinct shard, and sends each query to the corresponding

	Policy	Queuing model	Hedging probability	Load balancing
a)	*Naïve Hedge* [16, 72]	push	1	random
b)	*d-Hedge* [16, 40]	push	$Pr(RTT > d)$	random
c)	*p-Hedge* [40]	push	$q \cdot Pr(RTT > d)$	randomization + JSQ
d)	*Random load balancing*	push	0	none
e)	*Join-shortest-queue (JSQ)* [29]	push	0	JSQ
f)	*Join-bounded-shortest-queue (JBSQ)* [45]	push & pull	0	JBSQ
g)	*Per-Shard Queuing (PSQ)*	pull	0	centralized queue
h)	*Idealized Hedge (IH)* (§3)	pull	load-dependent	centralized queue
i)	*Load-aware Hedge* (LÆDGE) (§4)	pull	load-dependent	centralized queue

Table 1: Existing and proposed tail-mitigation strategies using hedging and load balancing.

PSLB, which schedules the query on one or more leaves. Each leaf is equipped with a queue and processes queries first-come, first-serve (FCFS) [40, 57], which is the best non-preemptive scheduling policy when tail latency is the most important metric [10, 40, 48, 57, 76]. To compute the final client response, the root needs one response per distinct shard.

We define the *end-to-end latency* of a client request as the time from the moment a root fans-out the original request into multiple queries until it has received at least one response per distinct shard. It is equal to the service time plus queuing delay experienced by the slowest query that needs to be answered in order to compute the final response.

2.2 IQ-jitter: Modelling Hiccups

Consider a set of leaves with identical hardware and software configuration, serving queries of a given application.

We define *IQ-jitter* (short for *intra-query jitter*), denoted by J, as the service-time variability that results from the fact that two leaves hosting the same data shard may take different amounts of time to serve the same query.

Similarly to Mirhosseini *et al.* [57], we express the query service time S as the sum of two random variables:

$$S = P + J,$$

where P is determined by query complexity and shard content/size, while J (IQ-jitter) is determined by system events that are independent of the application: OS-scheduler decisions, power-management algorithm decisions, interference by other applications—in general, the current software and hardware state of the leaf executing the query.

2.3 Hedging State of the Art

Hedging was invented to mitigate the effect of IQ-jitter on tail latency: As long as the system events causing performance hiccups are independent across leaves, two or more leaves hosting the same data shard are unlikely to all suffer a hiccup while serving the same query. Hence, by replicating a query across multiple leaves and collecting the response that arrives first, we reduce the probability of the query suffering a hiccup.

In this paper, we consider three representative hedging policies (top three rows in Table 1):

Naïve Hedge [72]: The PSLB *always and immediately* sends each query to any two leaves that host the corresponding shard. This is conceptually the simplest hedging policy as it does not require storing any state at the PSLB. It has been applied to many different contexts, including map-reduce jobs [3–5, 14, 82], DNS queries, database servers, and packet forwarding [72].

Delayed Hedge (d-Hedge) [16, 40]: For each query, the PSLB randomly picks a leaf that hosts the relevant shard and sends the query to it; if the reply does not arrive within a pre-configured, fixed delay d, the PSLB replicates the query on another leaf. When a query finishes, the PSLB cancels its replica if it exists. The value of d is tuned by the system operator to control the number of replicated queries in the system. This is the policy that was proposed by Dean *et al.* when they introduced the "tail at scale" problem [16].

Probabilistic Hedge (p-Hedge) [40]: This is similar to d-Hedge, but introduces an extra tuning knob: the probability q of replicating each delayed query; both the probability q and the delay d are trained based on the workload. This was proposed by Kaler *et al.* in their recent study of hedging policies for data centers [40] and is the most sophisticated hedging policy that we found in the literature.

We should clarify that all these policies—as any hedging policy we are aware of—limit the number of simultaneously run hedged requests to two. We follow the same strategy in all the policies that we define in this paper.

Ideally, a hedging policy walks the fine line between (a) replicating too many queries and adding too much system load (hence queuing delay), and (b) not replicating enough queries and failing to mitigate the effect of IQ-jitter on tail latency. Naïve Hedge errs toward the former (it always replicating as

much as possible); Vulimiri *et al.* [72] have showed that this reduces tail latency only when system load is below 30% [72]. d-Hedge and p-Hedge provide knobs for controlling the added load; however, as we will see, they still do not enable a good balance between (a) and (b), *i.e.,* they can still unnecessarily overload the system or fail to mitigate the effect of IQ-jitter, even if their knobs are carefully tuned (§3.3).

2.4 Load Balancing: State of the Art

The standard approach to managing latency is load balancing (LB). In this paper, we compare hedging with four standard LB policies ((d) to (g) in Table 1):

Random: For each query, the PSLB randomly picks a leaf that hosts the relevant shard and sends the query to it.

Join-shortest-queue (JSQ) [29]: For each query, the PSLB picks a leaf that hosts the relevant shard and sends the query to it; of all the candidate leaves, the PSLB picks the one with the smallest number of pending queries for that shard. JSQ outperforms Random but is far from optimal for FCFS servers with highly-variable job sizes [29,34,35]. We consider it because it is simple to implement and very popular in the industry, *e.g.,* it is widely deployed in reverse-proxies [29,58].

Per-Shard Queuing (PSQ): The PSLB stores each query until a leaf that can serve it becomes available. In other words, the PSLB dispatches a query to a leaf *only* if the leaf is idle. From a queuing-theory perspective, PSQ corresponds to a single-queue $M/G/k$ model, where k is the number of leaves to choose from. In theory, this outperforms any LB policy that uses multiple distinct queues (*e.g.,* JSQ) in the presence of non-deterministic service times [68]. In practice, PSQ exposes the round-trip latency between the PSLB and the leaf tiers, which may significantly impact throughput. It is, however, an excellent candidate for low-latency environments like modern datacenters [57,60] (*e.g.,* it takes $< 20\mu s$ to perform an RPC between two VMs in Microsoft Azure [22]).

Join-bounded-shortest-queue (JBSQ) [45]: This policy combines PSQ and JSQ by splitting the pending queries between the PSLB and leaf tiers. It takes a parameter n, which specifies the number of pending requests that each leaf can hold, *e.g.,* JBSQ(1) is equivalent to PSQ, while JBSQ(∞) is equivalent to JSQ. The value of n can be configured so as to hide the round-trip latency between the PSLB and leaf tiers and enable full throughput.

2.5 Simulation Setup

In the next two sections, we rely on discrete event simulation to compare hedging against standard LB policies. The goal is not to evaluate precisely how these policies perform in real systems (we use different experiments for that, later in the paper), but to understand some of their fundamental properties, *e.g.,* how does hedging compare to PSQ in an idealized setting (where PSQ is the optimal LB policy)?

Our simulation setup mimics an OLDI application deployed in a small- to medium-sized cluster: (a) The number of shards ranges from $N = 5$, which represents small-scale public-cloud deployments, and it is the default number of shards in Elasticsearch [21]; to $N = 500$ distinct shards, which represents larger public-cloud deployments [21, 70]. Each shard is replicated in $r = 2$, 3 or 6 leaves (depending on the experiment). The number of replicas in OLDI applications tends to be limited given DRAM costs [55]; 6 was the highest number of replicas per shard that we found in the literature [14, 40, 63, 70]. There are $N \times r$ leaves processing queries, all with the same hardware and software configuration. Leaves process queries FCFS [40]. All queues have infinite capacity. (b) Each PSLB fans out queries to leaves according to the simulated (hedging or LB) policy. Network latency between any two nodes is zero (revisited in § 5). (c) Client requests arrive at the root tier following an open-loop Poisson arrival process. (d) As stated earlier, guided by previous studies [57] and our own observations, we model query service time as the sum of two components, $S = P + J$. Unless otherwise noted, P follows an exponential distribution (the randomness comes from different queries of a workload), while J (IQ-jitter) follows a bimodal distribution. This means that, in any single experiment, an instance of a query executing at a leaf node experiences a hiccup with some probability (*e.g.,* 10^{-3}), while all hiccups have the same duration (*e.g.,* $15 \times \bar{P}$)—which approximates the results in [12] and reflects our observations from § 5.1. Across experiments, we vary hiccup probability and duration to model a range of real problematic system events.

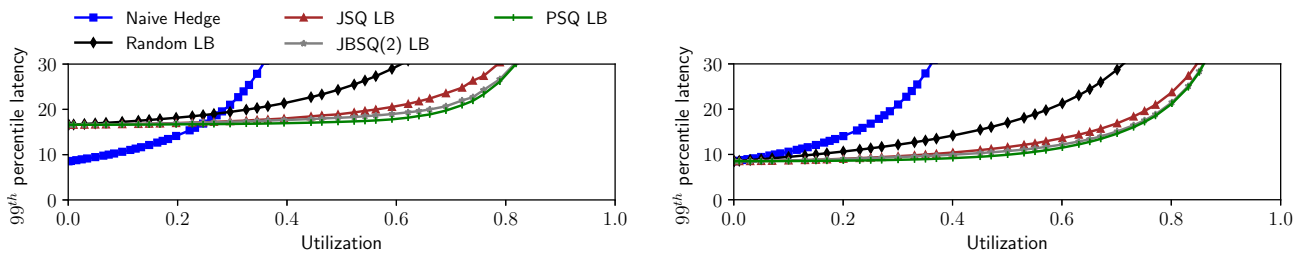
We report tail latency as a multiple of $\bar{P}$ (average application-dependent latency). For example, when we report that 99^{th} percentile latency is equal to 30, this means that the slowest 1% of client requests experience end-to-end latency that is higher than $30 \times \bar{P}$. We think that this provides more insight than an absolute number, especially since (in these particular experiments) we are measuring a simulated, idealized setup. In several plots, we show tail latency as a function of system utilization; a maximum utilization of 1.0 corresponds to $r/\bar{S}$ queries per time unit, where r is the replication factor.

3 Idealized Hedging

In this section, we present Idealized Hedge—an idealized hedging policy—and use it to gain insight into the applicability of hedging, as a general policy, to OLDI services.

3.1 Two Simple Observations

We start by comparing all the LB policies and Naïve Hedge: Fig. 2 shows 99^{th} percentile latency as a function of system utilization in a cluster with $N = 50$ shards, each replicated



(a) IQ-jitter with hiccup probability $\frac{1}{1000}$, hiccup duration $15 \times \bar{P}$.

(b) No IQ-jitter.

Figure 2: 99^{th} percentile latency as a function of utilization, in a cluster with 50×2 leaves, with and without IQ-jitter.

in $r = 2$ leaves, first with IQ-jitter of probability 10^{-3} and duration $15 \times \bar{P}$ (left), then without IQ-jitter (right).

We make two observations:

First, PSQ outperforms the other LB policies in all situations. This is expected from queuing theory. The performance difference is greater in the presence of IQ-jitter (Fig. 2a), which makes sense given that performance hiccups increase the dispersion of service-time distribution—and more service-time dispersion creates a bigger challenge for policies like JSQ and Random.

Second, in the presence of IQ-jitter (Fig. 2a), there is a clear "turning point": At low utilization, Naïve Hedge (despite its naïveté) delivers significantly lower tail latency than any LB policy. For instance, in an unloaded system, tail latency is $8 \times \bar{P}$ with Naïve Hedge and $17 \times \bar{P}$ with PSQ—close to a $2\times$ improvement. However, when utilization exceeds 25%, the LB policies deliver lower tail latency and higher throughput.

As a side note, in the absence of IQ-jitter (Fig. 2b), hedging unsurprisingly does not improve tail latency relative to LB, for any system utilization.

These observations suggest that a combined hedging/PSQ policy, which adapts the fraction of hedged queries to system utilization, might achieve lower tail latency than either hedging or LB alone.

3.2 The Design of Idealized Hedge

Idealized Hedge maximizes the potential of hedging to reduce latency in the following way:

1. A leaf is never idle when it can serve a pending query currently being served by at most one other leaf.
2. A leaf serves a hedged (replicated) query only when it cannot serve a non-hedged (yet-unserved) one.

The second property ensures that hedged queries never increase the queuing delay experienced by non-hedged queries. The two properties together ensure what we might call *work conservation in the presence of hedging*: no resources are idle when they could be doing useful work, and no resources are dedicated to hedging when they could be used for other work.

Idealized Hedge is not implementable because it requires perfect prediction of the completion times of currently executing queries: To ensure that the two properties stated above always hold, the system may need to cancel one copy of a hedged query currently executing on a leaf (so that the leaf can serve a new, non-hedged query that just arrived). To maximize the potential of hedging in our setup, Idealized Hedge must always cancel the copy that will take longer to complete, hence the need for perfect prediction.

We simulated Idealized Hedge as follows: Each PSLB maintains a queue with all the pending queries in order of arrival, and it knows the status of each leaf and which query it is processing (if busy). Moreover, if two copies of a hedged query are executing on different leaves and a new query arrives (that can be served by the same leaves), the PSLB's scheduler perfectly predicts which copy will finish executing first. With this knowledge, the PSLB performs the following operations: (a) Dispatches queries to the leaves in an FCFS manner using a pull-based discipline like PSQ. (b) Hedges a query as soon as a leaf that can serve it becomes idle. (c) Cancels any copy of a hedged query if another copy finishes first (we call this a *cleanup cancellation* or CC for brevity). (d) Cancels one copy of a hedged query upon arrival of a new query that can be served by the same leaf (we call this a *pre-emptive cancellation* or PC).

Fig. 3 shows the finite-state machine of Idealized Hedge for a given shard that is replicated in two leaves:

S_0 Both leaves are idle. There are no pending queries.

S_1 (Initial hedging) Both leaves are serving the same query Q_A, which they started to serve simultaneously; there are no other pending queries. This state occurs on a transition from S_0, following the arrival of query Q_A.

S_2 (No hedging) The two leaves are serving different queries, Q_A and Q_B, and there are no additional pending queries. This state occurs on a transition from S_1, following the arrival of query Q_B, at which point one copy of Q_A (the one that would have finished later) was pre-emptively cancelled.

S_3 (PSQ) The two leaves are serving different queries, Q_A and Q_B, and there are additional pending queries. In this

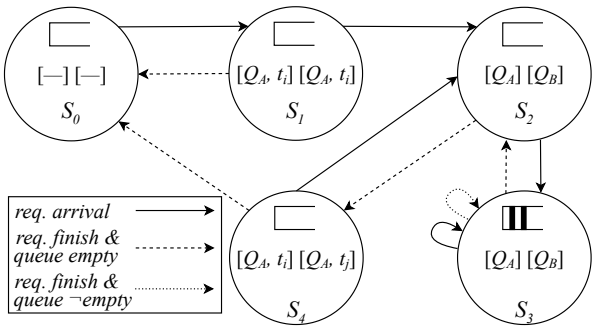


Figure 3: The finite-state machine of both Idealized Hedge and LÆDGE with CC+PC on one shard with two replicas. The states show: whether the shard queue is empty, whether replicas are running the same or different queries (Q), and whether the queries started at the same time (t).

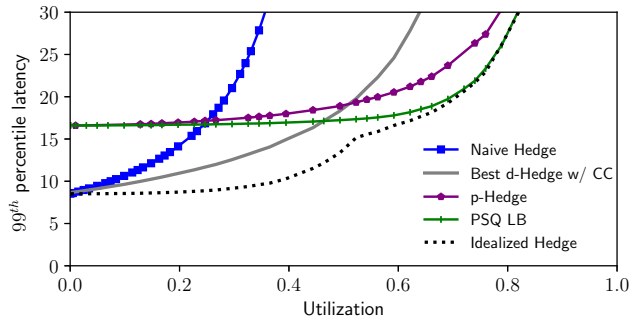


Figure 4: 99^{th} percentile latency as a function of utilization. Ideal Hedge vs. existing hedging policies and PSQ. Same setup as in Fig. 2a.

state, when a leaf finishes serving its query, it pulls the next one from the head of the queue (without hedging), as in the PSQ policy.

S_4 (Delayed hedging) Both leaves are serving the same query Q_A, which they started to serve at different times; there are no other pending queries. This state occurs on a transition from S_2, following the completion of one query, at which point the other query (Q_A) is reissued to the otherwise idle leaf.

3.3 Idealized Hedge versus State of the Art

Fig. 4 compares Idealized Hedge against the three hedging policies (Naïve Hedge, d-Hedge, p-Hedge), as well as PSQ (the best LB policy). The experimental setup matches that of Fig. 2a (50×2 leaves and IQ-jitter with probability 10^{-3} and duration $15 \times \bar{P}$).

Idealized Hedge outperforms (as expected) the real policies and exhibits the following behavior: at low utilization (until around 5%), when the leaves are mostly in states S_0 and S_1, it behaves like Naïve Hedge; at high utilization (from around 60%), when the leaves are mostly in state S_3, it converges to PSQ; in between, it clearly outperforms (by up to $8 \times \bar{P}$) all the real policies.

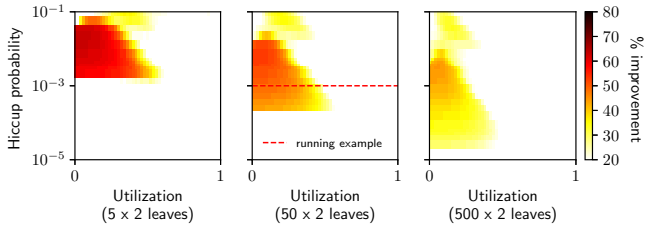
The most interesting comparison is between Idealized Hedge (black dotted line) and PSQ (green solid line with vertical lines), because it provides an upper bound on the tail-latency benefit that can be expected from *any* form of hedging. This comparison indicates two points: (1) There exists a significant utilization range (from ∼60% and up, in our setup) where no real hedging policy may bring any significant benefit relative to PSQ. (2) Outside this range, hedging *may* bring significant benefit, but the two state-of-the-art hedging policies cannot fulfill this potential. Only Naïve Hedge (blue solid line with squares) achieves all the benefit that hedging could achieve, but only at low utilization (until around 5%, in our setup). d-Hedge (gray solid line) outperforms the other real policies for a utilization range between ∼10 and ∼45%, but it remains far from Idealized Hedge. p-Hedge (purple solid line with pentagons) is outperformed by PSQ in all situations.

Of course, the behavior of d-Hedge and p-Hedge depends dramatically on how their configuration parameters are tuned; we followed all the instructions in the relevant literature, and we did our best to maximize their performance. For instance, in d-Hedge, we set the delay after which hedging occurs to $d = 5 \times \bar{P}$, because we experimentally found that higher values do not noticeably mitigate tail latency on low and medium loads; differently said, we allowed the least amount of hedging that has an impact on tail latency on low and medium loads comparable to that of Naïve Hedge (and yet the algorithm still led to congestion collapse at ∼60% utilization). In p-Hedge, we trained the parameters d and q, using the most successful of the methods explored in [40]: for each level of system utilization, we computed a "reissue budget" (the percentage of hedged queries in the system) using their iterative algorithm; then, for each level of system utilization, we trained d and q on sampled latency measurements using the proposed training algorithm that accounts for queuing delays for a fixed reissue budget. We tried sampling rates up to 80%; the results we show are for a sampling rate of 60%, because increasing it further did not significantly change the results: p-Hedge did not capture the rare hiccups through sampling, as they only occur once every 1000 queries. None of this proves that d-Hedge and p-Hedge could not perform any better, but it illustrates the difficulty of tuning them so as to achieve a desired balance between too little and too much hedging.

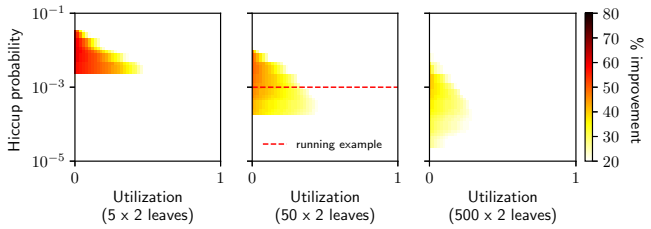
3.4 Beyond One Example

We now extend our observations beyond the specific setup of Fig. 4: how much potential does hedging have to improve tail latency as the cluster size and nature of IQ-jitter vary?

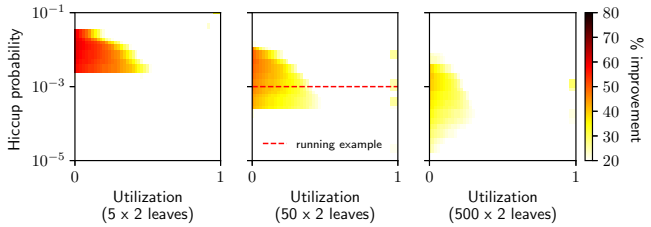
We consider the following scenarios: clusters of 5×2, 50×2, and 500×2 leaves, *i.e.,* small, medium, and large; hiccup probability ranging from 10^{-1} to 10^{-5}; hiccup duration $15 \times$



(a) Idealized Hedge

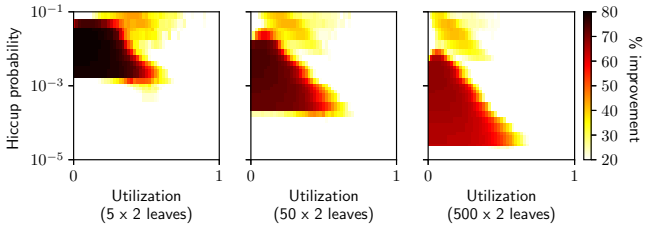


(b) LÆDGE

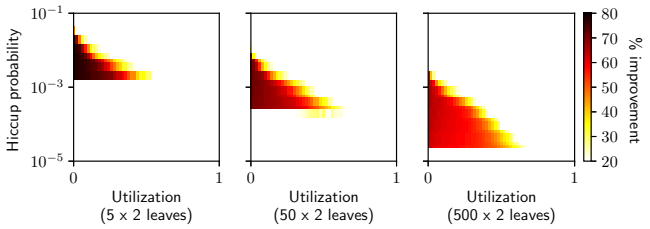


(c) LÆDGE with CC

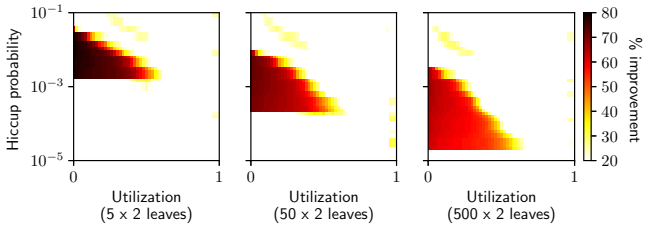
Figure 5: Heat maps showing how much various hedging policies improve the 99^{th} percentile latency relative to PSQ. Hiccup duration is $15 \times \bar{P}$.



(a) Idealized Hedge



(b) LÆDGE



(c) LÆDGE with CC

Figure 6: Heat maps showing how much various hedging policies improve the 99^{th} percentile latency relative to PSQ. Hiccup duration is $30 \times \bar{P}$.

$\bar{P}$ and $30 \times \bar{P}$. Regarding the latter, our choice of parameters is motivated by the literature: $15 \times \bar{P}$ has occurred as the result of badly configured timer intervals [12], while $30 \times \bar{P}$ as a result of non-conserving job-to-core allocation [69].

We summarize our results in two sets of heat maps, one for hiccup duration $15 \times \bar{P}$ (Fig. 5a), the other for $30 \times \bar{P}$ (Fig. 6a). Each heat map illustrates the relative improvement in 99^{th} percentile latency that Idealized Hedge brings relative to PSQ: the x-axis is system utilization, the y-axis is hiccup probability (on a logarithmic scale), and the intensity of each data point is the relative improvement in the 99^{th} percentile latency (so, a darker data point indicates higher potential for hedging to improve tail latency). We only show improvement greater than 20%, to focus on scenarios with significant improvement potential. Each column corresponds to a different cluster size: 5×2, 50×2, and 500×2 leaves, from left to right. The dashed horizontal line in Fig. 5a, middle heat map (so, 50×2 leaves) corresponds to the setup of Fig. 2a and 4.

First, we observe that **hedging cannot significantly improve tail latency when system utilization exceeds** $\sim$**60%** (all heat maps are empty beyond $\sim$60% utilization). Beyond this turning point, hedging improves latency at most by 20%, independently from cluster size and hiccup probability or duration. The intuition is simple: as system utilization increases and leaves become busier, opportunities for hedging disappear; as a result, Idealized Hedge eventually converges to PSQ. The turning point corresponds to medium-heavy utilization, where queues are starting to form, and below the point where the well-known heavy-traffic approximation determines behavior irrespective of service-time distribution [30]. Between $\sim$60% and $\sim$80% utilization, hedging provides at most 10% improvement (not visible in the heat maps); and beyond $\sim$80% utilization, no improvement at all.

Second, **hiccup duration does not affect the existence of potential improvement (the heat-map shape), only the amount of potential improvement (the heat-map intensity)**. Compare any two heat maps for the same cluster size in Figures 5a and 6a: they shade mostly the same (x, y) surface, but the heat map on the right (longer hiccup duration) is darker than the one on the left. The intuition is that, for any given cluster size, hedging can be useful only within a given hiccup probability range; outside this range, performance hiccups are either too rare or too frequent for hedging to make any difference, and this is independent of hiccup duration.

Third, **the larger the cluster size, the smaller the hiccup probability for which hedging may be useful**. For example, when the hiccup probability is between 10^{-4} and 10^{-5}, hedging may be useful only in the 1000-leaf cluster (and would

be useful in larger clusters as well); in the two smaller clusters, each request needs access to fewer distinct shards, and the probability of IQ-jitter slowing down the serving of one or more of these shards becomes insignificant. Conversely, the higher the hiccup probability, the smaller the cluster size for which hedging may be useful. For example, when hiccup probability exceeds 10^{-2}, hedging may be helpful in all three clusters (but less so in the 1000-leaf cluster, where it may improve tail latency by at most 35%).

4 LÆDGE: Approaching Idealized Hedge

In this section, we present Load-Aware Hedge (LÆDGE), an implementable policy that approximates Idealized Hedge by adapting hedging to system utilization.

Algorithm 1 shows LÆDGE in pseudo code: $SQ[i]$ is a shard queue that corresponds to shard i; q is a query that belongs to request r and concerns shard s, while q' is a replica of q. Each query that concerns a given shard may be: (a) hedged, i.e., sent to two nodes that serve the shard (line 7), (b) sent to a single node that serves the shard if that is the only available one (line 9), or (c) queued up at a shard queue if zero nodes that serve the shard are available (line 11). When a node becomes available, the oldest query in the shard's queue is sent to the node (line 18). If there are no queries in the shard's queue, the oldest running query that concerns the shard is hedged, i.e., a replica of the query is sent to the node (line 21).

LÆDGE has the following properties:

1. A leaf is never idle when it can serve a pending query currently being served by at most one other leaf.

2. A leaf *starts* to serve a hedged (replicated) query when it cannot serve a non-hedged (yet-unserved) one.

Unlike Idealized Hedge, LÆDGE does not guarantee that, at any point in time, any leaf serving a hedged query could not be serving a non-hedged one; it only guarantees that when a leaf *starts* to serve a hedged query it could not be serving a non-hedged one. Like PSQ, an efficient LÆDGE implementation requires a PSLB with µs-scale round-trip latency to the leaves, which is commonly found in datacenter environments [22,45].

We implemented three variants of LÆDGE, which differ only in the type of query cancellation that they support:

Plain LÆDGE: Similarly to PSQ, a PSLB dispatches queries to leaves from a centralized queue; similarly to Idealized Hedge, a leaf serves both non-hedged and hedged queries, prioritizing the former. There are no query cancellations.

LÆDGE with CC: This policy augments plain LÆDGE with cleanup cancellation (CC), *i.e.,* all copies of a hedged query are cancelled when another copy finishes executing first. An efficient implementation of this policy requires a low-overhead mechanism for interrupting a query and cleaning up its side effects. Whether such a mechanism exists or not depends on the application itself (for instance, Boucher *et al.* [13] enabled efficient µs-scale cancellations for microservices [1,27,56] written in Rust).

Algorithm 1: Generalized LÆDGE

```
1  // Initialize a shard queue (SQ) per shard
2  SQ[i] ← [ ], i ∈ [1,...,n_shards] ;
3  on request r arrival
4      for each shard s do
5          if available replicas of shard s ≥ 2 then
6              // Replication on arrival
7              send q and q' to 2 random replicas of shard s;
8          else if available replicas of shard s == 1 then
9              send q to the available replica ;  // No replication
10         else
11             enqueue q to SQ[s] ;
12         end
13     end
14 end;
15 on response p arrival from node n serving shard s
16     if size(SQ[s]) > 0 then
17         pop a pending query q from SQ[s] ;
18         send q to n ;                          // No replication
19     else
20         if ∃ a non-replicated unfinished query on shard s then
21             replicate the oldest query to node n ;
                //Delayed replication
22         end
23     end
24 end;
```

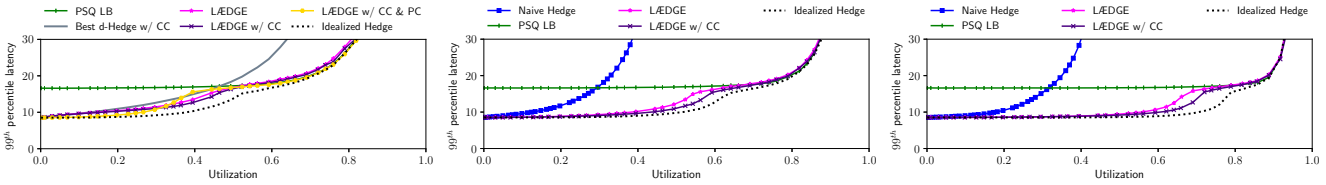
LÆDGE with CC+PC: This policy adds pre-emptive cancellation (PC), *i.e.,* one copy of a hedged query is cancelled when a new query arrives that can be served by the same leaves. The state machine corresponds to that of Idealized Hedge, shown in Fig. 3; the only difference is the lack of perfect completion-time prediction (while transitioning from S_1 to S_2, and from S_4 to S_2); instead, this policy cancels the copy that started executing most recently.

All cancellations are zero cost, in the sense that they introduce no extra processing delay and no extra communication between the PSLB and leaf tiers.

4.1 LÆDGE versus Idealized Hedge

Fig. 7a compares the three LÆDGE variants against Idealized Hedge, as well as d-Hedge and PSQ (the two best existing policies among the ones we simulated). The setup matches that of Figures 2a and 4.

We observe that plain LÆDGE reduces the gap to Idealized Hedge to at most $3.8 \times \bar{P}$ (while hiccup duration is $15 \times \bar{P}$, in this setup). At low utilization (up to ∼20%), it behaves like d-Hedge, which is the best existing policy at that utilization range. From some point on (∼50%), it converges to PSQ, which is the best existing policy at that utilization range. In between 20% and 50% utilization, it outperforms the existing policies, without using cancellations or parameter training, closing the average gap to Idealized Hedge to only $2.16 \times \bar{P}$.



(a) 50×2 shards (2 replicas) (b) 50×3 shards (3 replicas) (c) 50×6 shards (6 replicas)

Figure 7: 99^{th} percentile latency as a function of utilization. Comparing LÆDGE with different number of replicas per shard.

Second, we observe that cancellations not only do not significantly help LÆDGE but may actually hinder it. Adding CCs to LÆDGE marginally improves tail latency (given our assumption of zero-cost cancellations, it could not increase it). More interestingly, adding PCs to LÆDGE-with-CC *increases* tail latency at some utilization levels, starting from ∼30%. It turns out that cancelling the wrong copy of a hedged query (the one that would have finished first) is an expensive mistake; without any sophisticated completion-time predictors, one is better off not cancelling at all.

To better understand these results, we completed a careful analysis of PCs and their effect on tail latency. We define the "prediction accuracy" of a LÆDGE policy with cancellations as the proportion of PCs that correctly cancel the copy that would finish later. Overall, LÆDGE achieves $> 99\%$ prediction accuracy, which is unsurprising given the rarity of IQ-jitter events. However, once we consider only the PCs where at least one copy of the cancelled query experiences IQ-jitter, LÆDGE's prediction accuracy drops significantly. For example, at 40% system utilization, when transitioning from S_1 to S_2 in the presence of IQ-jitter, LÆDGE achieves prediction accuracy ∼50%; when transitioning from S_4 to S_2, prediction accuracy drops to ∼19%. The reason is that LÆDGE does not attempt to predict whether an IQ-jitter event is likely to have occurred—it simply picks the most-recently started copy. Cancellations appear to pay off only if we can assume a good predictor of performance hiccups due to system events; while this may be feasible according to Hao *et al.* [32], LÆDGE was designed in the absence of such an assumption.

So far, we considered only $r = 2$ replicas per shard. While this is typical in practice (the replication factor is often limited by high DRAM costs [55]), research proposals consider replication factors between $r = 2$ and 6 replicas per shard [40, 63]. The simulation behind the Figures 7b and 7c is set up the same way as Figure 7a, but with 3 and 6 replicas per shard, respectively. We see that, for the larger number of replicas, (1) the tail latency of Idealized Hedge and LÆDGE follows the minimal latency of Naïve Hedge up to 60% utilization (for 6 replicas), and (2) LÆDGE continues to achieve a significant part of the tail latency reduction of Idealized Hedge.

Finally, we should note that, out of curiosity, we experimented with two more types of application-independent noise (other than bimodal): exponential and bimodal+exponential. For the former, not even Idealized Hedge can improve tail latency, PSQ is the best policy, and LÆDGE performs almost the same as PSQ; this is not surprising, given that hedging was invented to deal with noise due to unpredictable system events, which is better modelled with a bimodal distribution. For the latter, the results were almost identical to the ones we got for bimodal noise.

4.2 Beyond One Example

We now extend our observations beyond the specific setup of Fig. 7: how well does LÆDGE fulfill the hedging potential as the cluster size and nature of IQ-jitter vary?

We consider the same scenarios as in §3.4 and once more summarize our results in heat maps in Figures 5 and 6. To assess how well LÆDGE approximates Idealized Hedge, we have to compare the heat maps. To simplify the comparison, we introduce Table 2, which summarizes Figures 5 and 6 as a percentage of the "surface" of each heat map that indicates improvement above a certain threshold (20%, 30% and 40%). For instance, consider the row that corresponds to hiccup duration $30 \times \bar{P}$ and cluster size 50×2 leaves, and the two columns that correspond to Idealized Hedge and LÆDGE, with $> 30\%$ latency improvement; the two cells where this row and columns intersect indicate that Idealized Hedge achieves such improvement for 29.7 % of the data points, while LÆDGE does for 12.6 % of data points.

To summarize, LÆDGE fulfills as much as half of the hedging potential, depending on the setup. Consider again the columns that correspond to Idealized Hedge and LÆDGE with $> 30\%$ latency improvement, and compare the values of these two columns that are in the same row; LÆDGE improves from 23% to 56% of the data points that are improved by Idealized Hedge (*i.e.,* that could possibly be improved through hedging). In general, LÆDGE is closer to Idealized Hedge for the medium and large clusters, and the longer hiccup duration.

On a side note, LÆDGE does not deteriorate lower latency percentiles compared to PSQ (*e.g.,* the median), but improves the tail. The rare hiccups that we analyzed, however, only influence the tail—not, for example, the 50^{th} percentile latency.

	% reduction wrt PSQ→	Idealized Hedge			LÆDGE			LÆDGE with CC			LÆDGE with PC		
		>20	>30	>40	>20	>30	>40	>20	>30	>40	>20	>30	>40
$15\times\bar{P}$	5×2 leaves	26.3	19.6	15.6	9.8	8.3	6.4	13.6	11.8	9.6	14.4	11.6	10.1
	50×2 leaves	29.9	21.5	15.4	13.4	9.8	3.8	16.8	12.2	5.2	15.9	12.3	9.2
	500×2 leaves	31.9	17.9	4.4	12.6	4.2	0.2	14.8	5.2	0.2	15.5	8.0	0.9
$30\times\bar{P}$	5×2 leaves	36.9	28.1	22.3	9.6	7.6	6.2	18.9	13.4	11.8	21.8	15.6	12.6
	50×2 leaves	40.5	29.7	22.5	15.2	12.6	11.1	22.6	17.4	15.5	23.3	18.2	14.5
	500×2 leaves	43.1	33.4	26.7	20.9	19.0	16.7	27.2	22.2	20.5	25.1	21.2	17.9

Table 2: Percentage of the total surface (in Figures 5 and 6) that has more than 20%, 30% and 40% reduction in 99^{th} percentile tail latency over Per-Shard Queuing for the workloads with the hiccup duration of $15\times\bar{P}$ and $30\times\bar{P}$.

4.3 Are Cancellations Worth the Effort?

First, in agreement with [4], our sensitivity analysis confirms that cleanup cancellation (CC) improves LÆDGE only marginally. Consider the columns of Table 2 that correspond to LÆDGE and LÆDGE with CC; on average, LÆDGE with CC offers the same improvement to 2.3% more data points than LÆDGE when hiccup duration is $15\times\bar{P}$, and to 5.6% more data points than LÆDGE when hiccup duration is $30\times\bar{P}$.

Second, while our sensitivity analysis in Table 2 shows that PCs have a marginally positive impact on LÆDGE, actually there exist a few areas of highly-negative impact. We have already discussed the intuition in §4.1; further analysis shows that the effect persists across the parameter space.

In a way, the non-effectiveness of cancellations is good news for application developers: Cancellations can be complicated and expensive to implement, they often require non-trivial application changes and language-specific mechanisms to avoid memory leaks and inconsistent application state [13], as well as additional interaction between the scheduler and the leaves. The fact that our simulated zero-cost cancellations bring no significant improvement to tail latency suggests that real-life cancellations are not worth the effort in our context.

5 System Evaluation with Lucene

We now evaluate LÆDGE on Lucene, a popular open-source enterprise search engine. It is representative of OLDI services because (1) it involves sharding, and (2) client requests are interactive, with expected latency in the millisecond scale.

Our workload is Lucene's standard nightly regression test, which consists of $\sim$10,000 search queries of four different types: phrase, term, multiterm and boolean [51]. The data consists of an inverted index of 18 million Wikipedia English web pages [77], split into 16 sub-indices (for parallel execution in 16 threads). Lucene comes in C++ and Java flavors; we used the former [52].

We implemented two LB policies (Random and PSQ) and two hedging policies (Naïve Hedge and LÆDGE) in OLDIsim, Google's open-source OLDI cloud benchmarking framework [28]. We changed its architecture to support shard replication, and we added PSLBs (as in Fig. 1). The I/O part of the framework stayed unchanged: it uses the event-based `libevent` API [49], on top of vanilla Linux and TCP. We extended the framework with request generation following an open-loop Poisson arrival process.

We ran our Lucene workload on AWS EC2 virtual machines (VMs) organized in a "cluster" placement group [2] in the same availability zone. We used two VM types: (1) compute-optimized instances with 16 vCPUs @3.0 *GHz* and 32 *GB* of memory (*c*5.4*xlarge*), and (2) general-purpose instances with 16 vCPUs @2.2 *GHz* and 64 *GB* of memory (*m*5*a*.4*xlarge*). All VMs were running the default Ubuntu 16.04.6 image, kernel version 4.4.0-1092-aws.

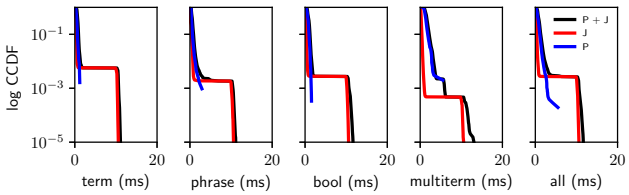
We deployed 5×2 leaf servers, *i.e.,* 5 distinct shards, each replicated in 2 leaves. When a leaf executes a query, it uses 16 parallel threads (one per vCPU). To avoid the introduction of data-driven bias in our results, we replicated the same index on all 5 shards (though, from the point of view of the application, they are still distinct shards served by different leaves). This decision simplifies the comparison with the simulation results in §4 without fundamentally changing the conclusions.

5.1 Empirical IQ-jitter Measurement

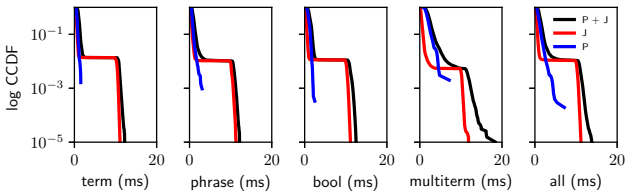
We started our experimental evaluation by measuring the real IQ-jitter experienced by our Lucene workload.

We executed the 10,000 queries of our workload 1000 times each, in a random order, always on the same server type. Consider a specific query Q. For each execution of this query, $Q_i, i = 1...1000$, we measured the service time S_i (which does not include any queuing or network delay). We approximated the application-dependent component of the service time experienced by Q as the minimum service time across executions: $P(Q) = \min_{i=1...1000} S_i$. Then we approximated the IQ-jitter experienced by each query execution Q_i as $J_i = S_i - P(Q)$. By putting together all the IQ-jitter values for a given query type, we obtained the IQ-jitter distribution for this query type.

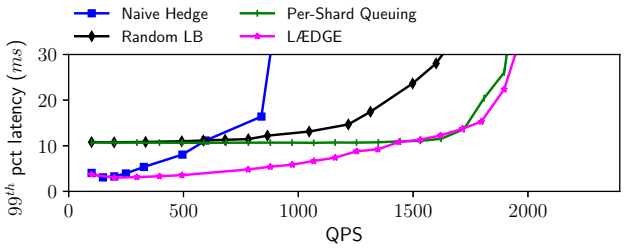
Fig. 8a and 8b show (in the form of CCDFs) the empirical distributions of P, J, and $P+J$, for the four query types of our workload, and for the two different server types. The curves differ in length as P's distribution size depends on the number of queries ($\sim$10,000), whereas the two other distribution sizes depend on the duration of the measurement experiment. The means of P and J in the two server types are 0.637 *ms* and



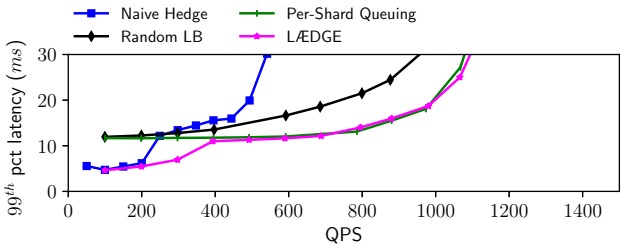
(a) Lucene service time in compute-optimized VM



(b) Lucene service time in general-purpose VM



(c) Latency vs throughput with compute-optimized VMs



(d) Latency vs throughput with general-purpose VMs

Figure 8: Mitigating the Lucene hiccups in a system implementation deployed on 2×5 leaves in EC2 VMs.

0.198 *ms* in Fig. 8a, and 0.926 *ms* and 0.444 *ms* in Fig. 8b, while the hiccup duration and probability are 10.162 *ms* and 0.0027 in Fig. 8a, and 10.249 *ms* and 0.0109 in Fig. 8b.

As a side note, we measured the latency between our VMs and found that the round-trip time is on average 91.2µs, with minimum and maximum latency 61.5µs and 150.6µs, respectively. The average network latency amounts to 10.94% and 6.7% of Lucene's mean service time ($P+J$) in the compute-optimized and general-purpose scenario, respectively.

We observe that IQ-jitter is substantial in both server types, and that the empirical distributions are consistent with our simulation setup: the application-dependent component (P) can be well approximated with an exponential distribution (a straight line on a log-based CCDF), while the IQ-jitter component (J) has a clear bimodal nature. This holds across the four query types that vary significantly in complexity (with "multiterm" queries being the most complex ones).

We investigated the reasons behind IQ-jitter and found that a significant part is due to involuntary rescheduling of Lucene threads, which occurs less frequently in the compute-optimized VMs. Of course, different Lucene deployments may experience less IQ-jitter: if the same workload runs on fully-controlled physical machines, IQ-jitter can be reduced by tweaking the OS or the application itself to mitigate the impact of involuntary thread rescheduling on tail latency.

5.2 Mitigating Tail Latency Through Hedging

Next, we measured the end-to-end latency experienced by our Lucene workload under varying system load. Figures 8c and 8d show the 99^{th} percentile latency as a function of queries per second, for the two server types, respectively.

LÆDGE behaves as expected: it matches Naïve Hedge at low utilization, converges to PSQ at some point, and outperforms the best alternative in between. The exact behavior depends on server type: On the compute-optimized VMs, LÆDGE converges to PSQ at about 60% utilization; before that, it improves tail latency by 49%, or 5.3 *ms*, on average, relative to PSQ. On the general-purpose VMs, convergence to PSQ happens quite earlier—at about 27% utilization—and the improvement of tail latency before that point is somewhat smaller (40%, or 4.7 *ms*, on average).

Our LÆDGE implementation (deployed in the cloud with real system noise and non-zero network latencies) behaved as our simulation predicted: With compute-optimized VMs, our experimental setup consists of 5×2 leaves with IQ-jitter of hiccup probability 0.0027 and hiccup duration $15.95 \times \bar{P}$. The closest simulated setup is a cluster of the same size with IQ-jitter of the same hiccup probability and hiccup duration $15 \times \bar{P}$. This corresponds to the leftmost heat map in Fig. 5b, y-axis value 0.0027 (which is close to the base of the triangular shape of the heat map). If we observe this heat map at the given y-axis value, we can see that our simulated LÆDGE policy significantly outperforms PSQ until utilization ∼40% and then converges to PSQ at utilization ∼60%—matching the behavior of our LÆDGE implementation.

6 Discussion

Scalability: PSQ-based scheduling scales naturally with the number of leaves, because the queues are centralized *only within* a shard. The degree of shard replication is typically small due to DRAM costs [55], and easily manageable by a single PSLB. As more shards are needed, this design scales horizontally by adding more machines for (co-located)

PSLBs, as well as adding more root nodes.

Higher network latency: LÆDGE exposes the round-trip latency between the PSLB and leaf tiers. In the Amazon environment that we used for our system evaluation (§5), this latency is an order of magnitude smaller than the service time. In an environment with higher PSLB-leaf latency, however, exposing this latency could significantly impact throughput. Adapting LÆDGE to such an environment would be straightforward: we would replace LÆDGE's Per-Shard Queuing component with JBSQ [45] (described in §2.4). More specifically, in Alg. 1, instead of always enqueuing the query at line 11, the PSLB would keep track of the queue sizes on the leaf nodes and send the query to the leaf with the shortest queue.

Hedging to more replicas: Kaler *et al.* [40] have shown that hedging the same query to more than two replicas does not provide additional latency benefits. This is consistent with our evaluation results, where hedging to two replicas was enough to mitigate *rare* hiccups.

Fault-tolerance: The architecture in Fig. 1 is resilient to the failure of all of its components: (1) the state in PSLBs is soft, (2) the shards are replicated across different leaf nodes, and (3) if a root fails, another one can take over.

Scheduling in the leaf nodes: We considered only FCFS scheduling in the leaf nodes (see § 2.1). It is possible that a change in scheduling discipline affects service time. However, as long as service time has a component J that depends on application-independent events and has a bimodal distribution, our insights still apply.

7 Related Work

Taming map-reduce latency: Early attempts of hedging studied long-running map-reduce jobs with execution times measured in seconds or minutes [17]. This timescale allows for "observe-then-predict" type of algorithms, with sophisticated execution profiling based on which a decision can be made about when it pays off to hedge [4, 5, 61, 82]. Systems such as LATE [82], Mantri [5] and Dolly [4] used hedging to mitigate the stragglers that would delay the entire phase. We focused on OLDI services operating at different timescales and do not lend themselves to heavy-weight profiling.

Hedging at low latencies: § 2.3 describes Naïve Hedge [72], d-Hedge [16] and p-Hedge [40] that we compared against. State-of-the-art reissue policies such as p-Hedge [40] address the throughput limitations of Naïve Hedge and d-Hedge. Recent work by Mirhosseini *et al.* [57], advocates PSQ as a plausible means to reduce tail latency. We show in §3.3 that PSQ-based hedging policies can outperform carefully-designed push-based ones. Hedging techniques that target network latency [24, 72, 79] are out of the scope of this work.

Early stop: In some cases, the user can receive a meaningful response without her request querying all the shards. In the literature, there are two main such scenarios: First, when accuracy of the results is sacrificed for lower latency [36, 66]. Second, when the final result can be decoded if only *a part* of queries finish [46, 65]. Such approaches are orthogonal ours and can be integrated to further reduce latency.

Advanced load balancing: Lu *et al.* [50] decouple discovery of lightly-loaded servers from job assignment. Since it offers no redundancy, this technique is prone to increased tail latency in the presence of IQ-jitter. "Snitching" is another interesting LB technique in which the root node monitors request latency and picks the fastest replica [6, 71]. This technique also offers no redundancy and is ineffective in case of bursty noise [32].

Adaptive parallelism: Many researchers have tried to predict the service time of interactive services and accordingly adjust the level of parallelism in the processing nodes, or prioritize short-running queries over long-running ones [33, 38, 44, 53, 70]. This approach trades-off service-time for throughput adaptively as a function of the load but does not attempt to reduce jitter due to underlying system events.

Cancellations: Prior work extensively studied cancellations [4, 9, 13, 16, 32]. Ananthanarayanan *et al.* [4] observed that cancellations do not improve tail latency of map-reduce workloads. Recent advances have shown that canceling microsecond-scale RPCs can be feasible, but memory leaks remain a problem. Bashir *et al.* [9] recently studied duplications and cancellations at multiple layers of a cloud system, while Hao *et al.* studied application-specific or OS-level instrumentation to increase the accuracy of cancellation decisions [32]. LÆDGE is simple and can be deployed everywhere, including in the cloud. It applies duplication only at the application level, and our results show comparable latency reductions with and without cancellations. We leave the study of combining cancellations with profiling to future work.

Infrastructure jitter: §1 includes numerous examples of system events that cause jitter. Hao *et al.* [32] observe and quantify noise in EC2 with a focus on disk read and write jitter. Our work focuses on in-memory, CPU-bound applications, which also observe a varying amount of jitter determined in part by the underlying cloud VM.

8 Conclusion

This paper demonstrates that hedging can be applied without significant throughput reduction by combining the best of hedging and load balancing, *i.e.,* by hedging only when the current system load allows it. We show the drawbacks of existing policies and, in turn, propose LÆDGE, an integration of hedging within a per-shard load-balancer, and quantify its benefits against an idealized hedging policy designed to outperform any realistic hedging policy. We show that LÆDGE can yield significant latency reductions over the state-of-the-art approaches. We also validate its benefits with a cloud-based deployment of an interactive web search application.

Acknowledgments

We would like to thank our shepherd Asaf Cidon and anonymous reviewers for their constructive feedback. This research is supported in part by a research grant from VMware.

References

[1] Amazon. AWS Lambda. https://aws.amazon.com/lambda/.

[2] Amazon. Placement groups. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/placement-groups.html.

[3] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Why Let Resources Idle? Aggressive Cloning of Jobs with Dolly. In *HOTCLOUD*, 2012.

[4] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective Straggler Mitigation: Attack of the Clones. In *NSDI*, pages 185–198, 2013.

[5] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the Outliers in Map-Reduce Clusters using Mantri. In *OSDI*, pages 265–278, 2010.

[6] Apache. Cassandra snitches. https://docs.datastax.com/en/archived/cassandra/3.0/cassandra/architecture/archSnitchesAbout.html.

[7] L. A. Barroso, J. Clidaras, and U. Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2013.

[8] L. A. Barroso, J. Dean, and U. Hölzle. Web Search for a Planet: The Google Cluster Architecture. *IEEE Micro*, 23(2):22–28, 2003.

[9] H. M. Bashir, A. B. Faisal, M. A. Jamshed, P. Vondras, A. M. Iftikhar, I. A. Qazi, and F. R. Dogar. Reducing tail latency using duplication: a multi-layered approach. In *CONEXT*, pages 246–259, 2019.

[10] A. Belay, G. Prekas, M. Primorac, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. The IX Operating System: Combining Low Latency, High Throughput, and Efficiency in a Protected Dataplane. *ACM Trans. Comput. Syst.*, 34(4):11:1–11:39, 2017.

[11] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy. *Site Reliability Engineering: How Google Runs Production Systems*. " O'Reilly Media, Inc.", 2016.

[12] M. Bligh, M. Desnoyers, and R. Schultz. Linux kernel debugging on google-sized clusters. In *Proceedings of the Linux Symposium*, pages 29–40, 2007.

[13] S. Boucher, A. Kalia, D. G. Andersen, and M. Kaminsky. Putting the "Micro" Back in Microservice. In *USENIX ATC*, pages 645–650, 2018.

[14] H. Casanova. Benefits and Drawbacks of Redundant Batch Requests. *J. Grid Comput.*, 5(2):235–250, 2007.

[15] Chris Jones and John Wilkes and Niall Murphy and Cody Smith. Service Level Objectives. https://landing.google.com/sre/sre-book/chapters/service-level-objectives/.

[16] J. Dean and L. A. Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, 2013.

[17] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[18] 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*, pages 205–220, 2007.

[19] C. Delimitrou and C. Kozyrakis. iBench: Quantifying interference for datacenter applications. In *IISWC*, pages 23–33, 2013.

[20] C. Delimitrou and C. Kozyrakis. Amdahl's law for tail latency. *Commun. ACM*, 61(8):65–72, 2018.

[21] Elasticsearch. https://www.elastic.co/what-is/elasticsearch.

[22] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. M. Caulfield, E. S. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. G. Greenberg. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *NSDI*, pages 51–66, 2018.

[23] B. Forrest. Bing and Google Agree: Slow Pages Lose Users. http://radar.oreilly.com/2009/06/bing-and-google-agree-slow-pag.html.

[24] A. Frömmgen, T. Erbshauser, A. P. Buchmann, T. Zimmermann, and K. Wehrle. ReMP TCP: Low latency multipath TCP. In *ICC*, pages 1–7, 2016.

[25] K. Gardner, S. Zbarsky, S. Doroudi, M. Harchol-Balter, and E. Hyytiä. Reducing Latency via Redundant Requests: Exact Analysis. In *SIGMETRICS*, pages 347–360, 2015.

[26] P. Garraghan, X. Ouyang, R. Yang, D. McKee, and J. Xu. Straggler Root-Cause and Impact Analysis for Massive-scale Virtualized Cloud Datacenters. *IEEE Trans. Serv. Comput.*, 12(1):91–104, 2019.

[27] Google. Cloud functions. https://cloud.google.com/functions/.

[28] Google Cloud Platform Blog. Benchmarking web search latencies. http://cloudplatform.googleblog.com/2015/03/benchmarking-web-search-latencies.html.

[29] V. Gupta, M. Harchol-Balter, K. Sigman, and W. Whitt. Analysis of join-the-shortest-queue routing for web server farms. *Perform. Evaluation*, 64(9-12):1062–1081, 2007.

[30] S. Halfin and W. Whitt. Heavy-Traffic Limits for Queues with Many Exponential Servers. *Oper. Res.*, 29(3):567–588, 1981.

[31] M. Handley, C. Raiciu, A. Agache, A. Voinescu, A. W. Moore, G. Antichi, and M. Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *SIGCOMM*, pages 29–42, 2017.

[32] M. Hao, H. Li, M. H. Tong, C. Pakha, R. O. Suminto, C. A. Stuardo, A. A. Chien, and H. S. Gunawi. MittOS: Supporting Millisecond Tail Tolerance with Fast Rejecting SLO-Aware OS Interface. In *SOSP*, pages 168–183, 2017.

[33] M. E. Haque, Y. H. 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-XX*, pages 161–175, 2015.

[34] M. Harchol-Balter. Task assignment with unknown duration. *J. ACM*, 49(2):260–288, 2002.

[35] M. Harchol-Balter, M. Crovella, and C. D. Murta. On Choosing a Task Assignment Policy for a Distributed Server System. *J. Parallel Distributed Comput.*, 59(2):204–228, 1999.

[36] Y. He, S. Elnikety, J. R. Larus, and C. Yan. Zeta: scheduling interactive services with partial execution. In *SOCC*, page 12, 2012.

[37] V. Jalaparti, P. Bodík, S. Kandula, I. Menache, M. Rybalkin, and C. Yan. Speeding up distributed request-response workflows. In *SIGCOMM*, pages 219–230, 2013.

[38] M. Jeon, S. Kim, S. won Hwang, Y. He, S. Elnikety, A. L. Cox, and S. Rixner. Predictive parallelization: taming tail latencies in web search. In *SIGIR*, pages 253–262, 2014.

[39] G. Joshi, E. Soljanin, and G. W. Wornell. Efficient Redundancy Techniques for Latency Reduction in Cloud Systems. *TOMPECS*, 2(2):12:1–12:30, 2017.

[40] T. Kaler, Y. He, and S. Elnikety. Optimal Reissue Policies for Reducing Tail Latency. In *SPAA*, pages 195–206, 2017.

[41] M. Kambadur, T. Moseley, R. Hank, and M. A. Kim. Measuring interference between live datacenter applications. In *SC*, page 51, 2012.

[42] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat. Chronos: predictable low latency for data center applications. In *SOCC*, page 9, 2012.

[43] H. Kasture and D. Sánchez. Ubik: efficient cache sharing with strict qos for latency-critical workloads. In *ASPLOS-XIX*, pages 729–742, 2014.

[44] S. Kim, Y. He, S. won Hwang, S. Elnikety, and S. Choi. Delayed-Dynamic-Selective (DDS) Prediction for Reducing Extreme Tail Latency in Web Search. In *WSDM*, pages 7–16, 2015.

[45] M. Kogias, G. Prekas, A. Ghosn, J. Fietz, and E. Bugnion. R2P2: Making RPCs first-class datacenter citizens. In *USENIX ATC*, pages 863–880, 2019.

[46] J. Kosaian, K. V. Rashmi, and S. Venkataraman. Parity models: erasure-coded resilience for prediction serving systems. In *SOSP*, pages 30–46, 2019.

[47] J. Leverich and C. Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *EUROSYS*, pages 4:1–4:14, 2014.

[48] J. Li, N. K. Sharma, D. R. K. Ports, and S. D. Gribble. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. In *SOCC*, pages 9:1–9:14, 2014.

[49] libevent – an event notification library. https://libevent.org/.

[50] Y. Lu, Q. Xie, G. Kliot, A. Geller, J. R. Larus, and A. G. Greenberg. Join-Idle-Queue: A novel load balancing algorithm for dynamically scalable web services. *Perform. Evaluation*, 68(11):1056–1071, 2011.

[51] Lucene nightly benchmarks. https://home.apache.org/~mikemccand/lucenebench/.

[52] Lucene++. https://github.com/luceneplusplus/LucenePlusPlus.

[53] C. Macdonald, N. Tonellotto, and I. Ounis. Learning to predict response times for online query scheduling. In *SIGIR*, pages 621–630, 2012.

[54] A. Maricq, D. Duplyakin, I. Jimenez, C. Maltzahn, R. Stutsman, R. Ricci, and A. Klimovic. Taming Performance Variability. In *OSDI*, pages 409–425, 2018.

[55] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch. Power management of online data-intensive services. In *ISCA*, pages 319–330, 2011.

[56] Microsoft. Azure functions. https://azure.microsoft.com/services/functions/.

[57] A. Mirhosseini and T. F. Wenisch. The Queuing-First Approach for Tail Management of Interactive Services. *IEEE Micro*, 39(4):55–64, 2019.

[58] NGINX. https://www.nginx.com/.

[59] D. M. Novakovic, N. Vasic, S. Novakovic, D. Kostic, and R. Bianchini. DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments. In *USENIX ATC*, pages 219–230, 2013.

[60] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: distributed, low latency scheduling. In *SOSP*, pages 69–84, 2013.

[61] X. Ouyang, P. Garraghan, B. Primas, D. McKee, P. Townend, and J. Xu. Adaptive Speculation for Efficient Internetware Application Execution in Clouds. *ACM Trans. Internet Techn.*, 18(2):15:1–15:22, 2018.

[62] G. Prekas, M. Kogias, and E. Bugnion. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *SOSP*, pages 325–341, 2017.

[63] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. R. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *ISCA*, pages 13–24, 2014.

[64] Z. Qiu and J. F. Pérez. Evaluating the Effectiveness of Replication for Tail-Tolerance. In *CCGRID*, pages 443–452, 2015.

[65] K. V. Rashmi, M. Chowdhury, J. Kosaian, I. Stoica, and K. Ramchandran. EC-Cache: Load-Balanced, Low-Latency Cluster Caching with Online Erasure Coding. In *OSDI*, pages 401–417, 2016.

[66] L. Ravindranath, J. Padhye, R. Mahajan, and H. Balakrishnan. Timecard: controlling user-perceived delays in server-based mobile applications. In *SOSP*, pages 85–100, 2013.

[67] N. B. Shah, K. Lee, and K. Ramchandran. When Do Redundant Requests Reduce Latency? *IEEE Trans. Communications*, 64(2):715–722, 2016.

[68] J. F. Shortle, J. M. Thompson, D. Gross, and C. M. Harris. *Fundamentals of queueing theory*, volume 399. John Wiley & Sons, 2018.

[69] R. Sites. Data Center Computers: Modern Challenges in CPU Design. https://youtu.be/QBu2Ae8-8LM?t=1205.

[70] A. Sriraman and T. F. Wenisch. μTune: Auto-Tuned Threading for OLDI Microservices. In *OSDI*, pages 177–194, 2018.

[71] P. L. Suresh, M. Canini, S. Schmid, and A. Feldmann. C3: Cutting Tail Latency in Cloud Data Stores via Adaptive Replica Selection. In *NSDI*, pages 513–527, 2015.

[72] A. Vulimiri, P. B. Godfrey, R. Mittal, J. Sherry, S. Ratnasamy, and S. Shenker. Low latency via redundancy. In *CONEXT*, pages 283–294, 2013.

[73] J. Wagner. Why Performance Matters. https://developers.google.com/web/fundamentals/performance/why-performance-matters/.

[74] D. Wang, G. Joshi, and G. W. Wornell. Efficient task replication for fast response times in parallel computation. In *SIGMETRICS*, pages 599–600, 2014.

[75] Q. Wang, Y. Kanemasa, J. Li, D. Jayasinghe, T. Shimizu, M. Matsubara, M. Kawaba, and C. Pu. Detecting Transient Bottlenecks in n-Tier Applications through Fine-Grained Analysis. In *ICDCS*, pages 31–40, 2013.

[76] A. Wierman and B. Zwart. Is Tail-Optimal Scheduling Possible? *Oper. Res.*, 60(5):1249–1257, 2012.

[77] Wikipedia: Database download. https://en.wikipedia.org/wiki/Wikipedia:Database_download#english.

[78] Z. Wu, C. Yu, and H. V. Madhyastha. CosTLO: Cost-Effective Redundancy for Lower Latency Variance on Cloud Storage Services. In *NSDI*, pages 543–557, 2015.

[79] H. Xu and B. Li. RepFlow: Minimizing flow completion times with replicated flows in data centers. In *INFOCOM*, pages 1581–1589, 2014.

[80] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey. Bobtail: Avoiding Long Tails in the Cloud. In *NSDI*, pages 329–341, 2013.

[81] X. Yang, S. M. Blackburn, and K. S. McKinley. Elfen Scheduling: Fine-Grain Principled Borrowing from Latency-Critical Workloads Using Simultaneous Multithreading. In *USENIX ATC*, pages 309–322, 2016.

[82] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *OSDI*, pages 29–42, 2008.