Open access • Proceedings Article • DOI:10.1109/SRDS.2018.00030

# Héron: Taming Tail Latencies in Key-Value Stores Under Heterogeneous Workloads



Vikas Jaiman, Vikas Jaiman, Sonia Ben Mokhtar, Vivien Quéma ...+2 more authors

**Institutions:** Université catholique de Louvain, University of Grenoble, Grenoble Institute of Technology, IBM

**Published on:** 02 Oct 2018 - Symposium on Reliable Distributed Systems

# Héron: Taming Tail Latencies in Key-Value Stores under Heterogeneous Workloads

Vikas Jaiman, Sonia Ben Mokhtar, Vivien Quéma, Lydia Chen, Etienne Rivière

# Héron: Taming Tail Latencies in Key-Value Stores under Heterogeneous Workloads

Vikas Jaiman*¶, Sonia Ben Mokhtar†, Vivien Quéma*‡, Lydia Y. Chen§ and Etienne Rivière¶
*Université Grenoble Alpes, LIG (CNRS UMR 5217), France
†INSA Lyon, LIRIS, CNRS, France ‡Grenoble INP, France
§IBM Research – Zurich, Switzerland ¶Université catholique de Louvain, Belgium
*{firstname.lastname}@univ-grenoble-alpes.fr, †{firstname.lastname}@insa-lyon.fr, ‡{firstname.lastname}@grenoble-inp.fr,
§{yic}@zurich.ibm.com, ¶{firstname.lastname}@uclouvain.be

***Abstract*—Avoiding latency variability in distributed storage systems is challenging. Even in well-provisioned systems, factors such as the contention on shared resources or the unbalanced load between servers affect the latencies of requests and in particular the tail (95th and 99th percentile) of their distribution. One effective counter measure for reducing tail latency in key-value stores is to provide efficient replica selection algorithms. However, existing solutions are based on the assumption that all requests have almost the same execution time. This is not true for real workloads. This mismatch leads to increased latencies for requests with short execution time that get scheduled behind requests with large execution times. We propose Héron, a replica selection algorithm that supports workloads with heterogeneous request execution times. We evaluate Héron in a cluster of machines using a synthetic dataset inspired from the Facebook dataset as well as two real datasets from Flickr and WikiMedia. Our results show that Héron outperforms state-of-the-art algorithms by reducing both median and tail latency by up to 41%.**

***Index Terms*—Distributed Storage, Performance, Scheduling**

## I. Introduction

Users of online services have increasingly high expectations on the performance and responsiveness of these services [1], [2]. A slow service directly impacts the revenue of its provider, even if the performance drop is only for a short period of time [3]. Enforcing *predictable* performance is a challenging task even for well-provisioned systems. End user requests go through hundreds of servers. Performance hiccups at any of these servers may dramatically inflate the observed latency for some of these requests. For instance, measurements from a real Google service [3] running in a cluster of 2,000 servers show that if one in 100 user requests gets slow (e.g., has a 1 second latency) while handled by one server that is collecting responses from 100 other servers in parallel, then 63% of user requests will take more than one second to execute. This problem is commonly known as the *tail latency* problem. Several approaches have been proposed to reduce tail latency for the different components of large scale distributed systems [4]–[11]. They include techniques such as reissuing requests, using preferential resource allocation, leveraging parallelism for individual requests or sending redundant requests.

Of the services used for building cloud application, storage plays a fundamental role for overall services tail latencies. Key-Value Stores are the dominant class of storage solutions in this context and are the focus of this paper. In a key-value store, each piece of data or value is replicated on multiple servers for fault-tolerance. Replica selection strategies can help reducing tail latency when the performance of these servers differ. Specifically, a request attempting to access the value for a given key can be directed to the presumably best replica, i.e. the one that is expected to serve the request with the smallest latency.

State-of-the-art replica selection algorithms that are Cassandra dynamic snitching [12] and C3 [13] have been designed for workloads where requests access values of the same size. An analysis of real life key-value stores' workloads (e.g., of Facebook's Memcached deployment [14]) shows that this assumption does not hold, as user requests typically access values of sizes ranging from 1 KB to few MBs. We show in this paper that under such workloads, the tail latency of a key-value store using these algorithms increases dramatically compared to a scenario where all values have the same size. The increase in tail latency ranges from $\times 10$ when clients access $1\%$ of large values to $\times 126$ when this proportion reaches $20\%$.

The reason why existing algorithms do not perform well under heterogeneous workloads is that fast requests accessing small values can get stuck behind slow ones accessing large values. This can dramatically increase the latency of fast requests, a phenomenon known as *head-of-line-blocking* [3]. Selecting the best replica for a request, while preventing fast requests from being stuck behind slow ones, requires addressing two challenges. First, when a request for a key arrives at the entry-point of a key-value store, the size of the corresponding value is not known. A first challenge is thus to be able to predict this size based on the key with minimal operational overhead. Assuming that the size of values can be correctly estimated at request time, a second challenge is to be able to choose a replica that can prevents the head-of-line-blocking scenario.

**Contributions.** We present Héron, a replica selection algorithm that reduces tail latencies under heterogeneous workloads. Héron predicts which requests will require significant time by keeping track of the keys corresponding to large values. It does so using a Bloom filter at each server. Once a request has been identified as accessing a small (respectively, a large) value, it applies an appropriate replica selection algorithm, which avoids head-of-line-blocking.

| | |
|---|---|
| Cassandra [12] | Choose replica based on history of read latencies |
| MongoDB [19] | Select nearest node using network latency |
| OpenStack Swift [20], Apache Accumulo [21] | Read from a single node, choose a new replica in case of a node failure |
| Riak [22] | Recommend to use an external load balancer |

TABLE I: Replica selection strategies (adapted from [13]).

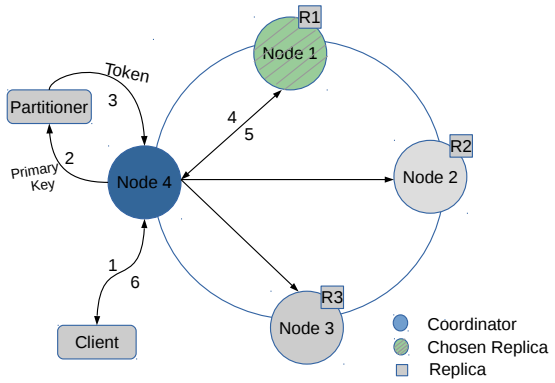


Fig. 1: Replica selection in Cassandra.

Héron can be applied to any low latency key-value store where value replication is enabled. Among such stores, Cassandra [12] implements the most efficient of existing replica selection algorithms (see Table I). It is therefore the best reference point for our study. We implemented Héron in Cassandra and compared its performance to state-of-the-art algorithms (e.g., dynamic snitching [12] – DS for short – and C3 [13]) in a public cluster of 15 machines on Grid5000 [15].

We evaluate Héron with four datasets, two synthetic datasets including one based on access statistics from Facebook [14] and two real datasets from Flickr [16] and WikiMedia [17]. We use YCSB [18] to generate different requests workloads, allowing to evaluate the system under different read/write ratios. Our results show that Héron improves tail latency over state-of-the-art algorithms without compromising median latency.

The remaining of this paper is structured as follows. We first present the background on replica selection in key-value stores (§II) and explain the limitations of the C3 and DS replica selection strategies in presence of heterogeneous workloads. Next, we further detail the challenges addressed in this paper (§III). Afterwards, we present a detailed description of Héron (§IV) and present its implementation and performance evaluation (§V). Finally we discuss related work (§VI) and conclude the paper (§VII).

## II. Replica selection in key-value stores

A key-value store maintains a number of tables, each mapping a collection of keys to values of arbitrary size. A partitioning scheme distributes keys between servers. To enforce fault tolerance, key-value stores use data replication. To retrieve a value for a given key, a client sends a request to one of the nodes in the system, which then redirects this request to one of the servers holding a replica. We take the example of Cassandra [12] in Figure 1. Each value is by default replicated to three replica servers (in the example, let us consider a value $v$ replicated on R1, R2 and R3). The request initially arrives at a node (step 1) that will act as its *coordinator*. Depending on the implementation of the coherence mechanisms, the coordinator has to fetch the value from one or multiple replicas. We consider the most-common configuration where the coordinator waits only for the response from a *single* replica. The coordinator asks the partitioner to hash the key (step 2) and gets a token in return (step 3). The coordinator can then identify the first node holding the key/value pair (i.e., R1). The coordinator identifies all the nodes holding the key/value pair based on the replication strategy. Here, replicas of a given pair are placed in successive nodes clockwise. The coordinator uses a *replica selection* algorithm to select the best replica for executing the query (step 5), and replies to the client (step 6).

### A. Dynamic snitching

By default, in Cassandra, replica selection is done using *dynamic snitching* (DS for short) [12]. The performance of read requests from various replicas is monitored over time and the best replica is selected based on its recent performance history. DS maintains a score for each replica that is updated every 100 ms. All replica scores are reset every 600 s, to allow replicas to possibly recover. One of the limitations of this approach is that it is solely based on replicas' past read performance, without considering the forthcoming load at each replica (i.e., its queue size). This may lead to overloading recently-fast replicas and to the appearance of bottlenecks in the system, ultimately impacting tail latency.

### B. The C3 algorithm

C3 [13] is a replica selection algorithm that improves over DS by handling service time variations among replicas. C3 computes replica scores based on both service time and queue size. This score is then used by a request coordinator for choosing the replica that is expected to better help reducing the request waiting time. Additionally, to avoid overloading a given replica queue (e.g., because the replica is fast and thus simultaneously selected by several coordinators), C3 uses a rate control mechanism at each replica to limit the arrival of requests. Results show that C3 significantly improves tail latencies compared to DS [13].

### C. Performance of DS and C3 under heterogeneous workloads

Several studies, including the analysis of a Facebook Memcached deployment [23], show that workloads are heterogeneous: key-value stores contain values of sizes ranging from a few bytes (e.g., text messages) to MB (e.g., photos or videos). Our analysis of the Flickr [16] and WikiMedia [17] datasets confirms this trend. Figure 2 shows the CDF of value sizes for the two datasets, ranging from a few bytes to MBs.

We illustrate the limitations of C3 and DS in presence of heterogeneous workloads with an example in Figure 4. In the left figure, three replica servers A, B and C currently have a service time of 2 ms, 3 ms and 1 ms, respectively, for requests

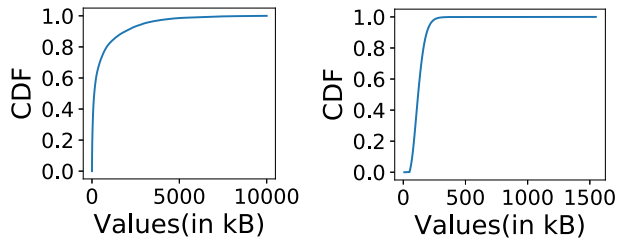


Fig. 2: CDF of value sizes for WikiMedia (left) and Flickr (right) datasets.

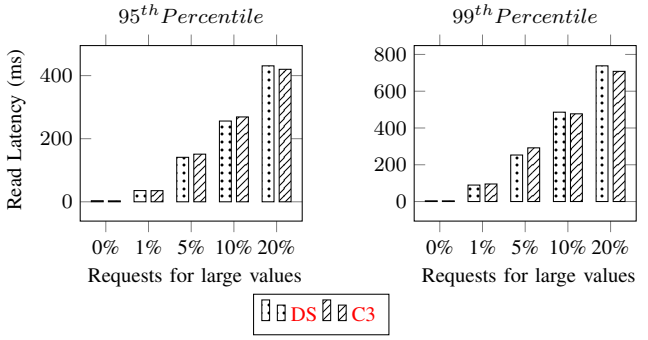


Fig. 3: Tail latency when varying the proportion of requests for large values with DS and C3.

of values size 1 KB. For the sake of simplicity, let us assume that the large request depicted with a large square and a dark color has a size of 512 KB and that small requests depicted with small rectangles and a fair color have a size of 1 KB. C3 will select replica server A for subsequent requests since it only considers the queue size and service time for estimating the fastest replica (queue size $\times$ service time of replica A is smaller than queue size $\times$ service time for the other replicas). DS only considers the past read performance (measured every 100 ms). Let us assume that in the recent past replica A only processed small requests. In that case, DS will select this replica to process the subsequent requests. As a result, under both algorithms, the latency of the next small request will be of (X+2) ms where X$\gg$2: X ms waiting for the large request to be processed, and 2 ms for processing the small request (central figure).

In an ideal scenario, the next small request should complete in a much shorter time: it should not be stuck behind the large request at replica server A. This means that, while processing a large request, a server should no longer be selected to process small requests. These small requests should be scheduled instead on the following best replica based on its score (service time $\times$ queue size). In the example of Figure 4, the next small requests should be sent to replica server C, which would yield a latency of 5 ms (right figure).

More practically, to study the impact of heterogeneous workloads on DS and C3, we use a synthetic workload containing small (1 KB) and large (512 KB) values. We use YCSB [18] to generate a read-heavy workload. We vary the proportion of large values from 0% to 20%. Our experimental settings are detailed in Section V. Figure 3 presents the $95^{th}$ and $99^{th}$ percentile of the read latency distribution. We observe a significant increase in tail latency when increasing the proportion of large values for both DS and C3. For the $95^{th}$ percentile, the increase ranges from $\times 10$ when there are 1% of requests for large values, to $\times 126$ when there are 20% of requests to large values.

## III. Challenges

This section details the challenges associated with reducing tail latencies under heterogeneous workloads, and the design space for building a key/value store with this goal.

Dealing with heterogeneous requests first requires being able to distinguish requests accessing small values from requests accessing large values. This must be done by the coordinator node, but this node only knows the key that is requested. It does not hold the value and must therefore use a specific mechanism to estimate the category of size the value belongs to. Under the high performance constraints of key-value-stores, this mechanism must be cost-effective.

Once the coordinator is able to distinguish between requests for small and large values, it must make appropriate scheduling decisions, i.e. decide which of the replicas to user for handling the request. Both static and dynamic scheduling strategies can be considered.

*Static* scheduling strategies permanently assign one or multiple servers to handle requests accessing small (or large) values. While this has been shown to be effective in other contexts (e.g., scheduling in high load situations where large request overwhelm a server [11], [24]), it may lead to the under utilization of the dedicated servers if the workloads include a minority of large requests.

With a *dynamic* scheduling strategy, a particular replica is reserved at runtime and only for a small amount of time to handle requests for large values, leaving the others available for handling requests to small values as shown in figure 5. Compared to a static assignment policy, this approach has the benefit of blocking replicas only temporarily. With dynamic scheduling, a request can be handled by any replica of the value. Two main approaches are available for selecting this replica, priority-based and FIFO algorithms. Priority-based algorithms, such as shortest job first [25] give the priority to fastest jobs over slower ones. In our context, this means giving priority to requests accessing small values over requests accessing large values. This approach is not ideal for a key-value store. It creates imbalance between the requests of different clients, and clients accessing larger values can potentially observe a significant increase in latency. FIFO algorithms process requests in their order of arrival, regardless of the size of the access values. They do not incur fairness or consistency problems and are therefore more appropriate for key/value stores. The challenge however is to offer a FIFO scheduling algorithm that avoids the head-of-line-blocking problem.

## IV. Héron design and implementation

This section presents Héron, a replica selection algorithm that aims at reducing tail latency under heterogeneous workloads.

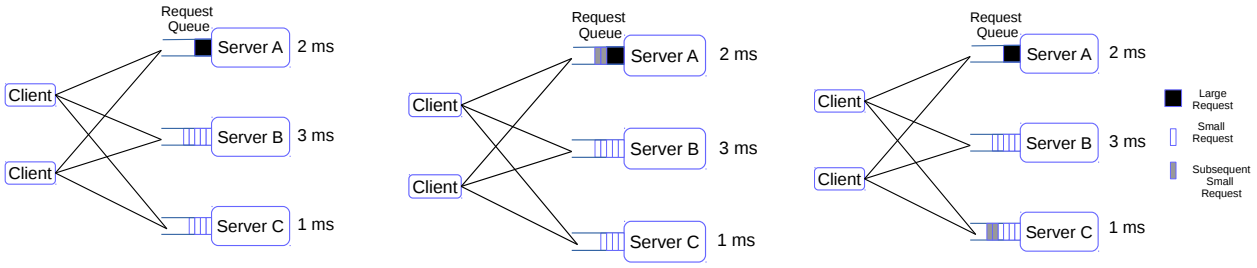


Fig. 4: *Left:* An example scenario where a large request has been scheduled on replica server A. The question is: where should subsequent small requests be scheduled? *Middle:* Shows how C3 and DS schedule the subsequent small requests to replica server A by considering the (queue size $\times$ service time) and past read performance respectively. For C3 and DS we observe that small requests get stuck behind the large one on replica server A. *Right:* Shows how Héron dynamically blocks the replica server A for small period of time until the latter finishes the processing of the large request. Meanwhile, subsequent small requests are scheduled on replica server C.

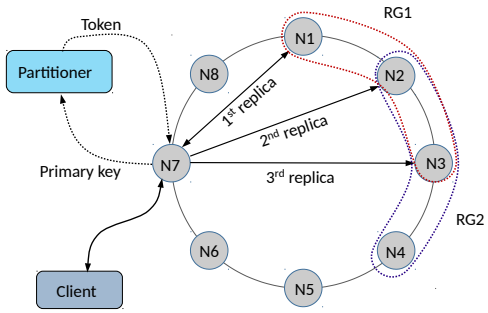


Fig. 5: Cassandra placement strategy.

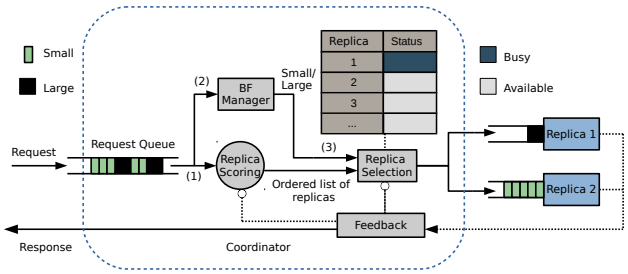


Fig. 6: Operating principle of Héron.

Figure 6 presents the architecture of Héron and how it handles client requests. When a request from a client reaches a coordinator, it first goes through a replica scoring module (step 1). The coordinator retrieves the set of replicas storing the requested value by querying the partitioner. The replica scoring module ranks these replicas based on periodic feedback on their latest service time and queue size. In parallel, the request is sent to the Bloom filter manager (step 2). This module estimates whether a given request will access a small or a large value. It uses a Bloom filter that keeps track of large requests. The replica selection module (step 3) uses the input from both modules to select the replica that is expected to serve the request faster.

We present first the size estimation using Bloom filters (§IV-A), followed by mechanisms for replica scoring (§IV-B) and replica selection (§IV-C).

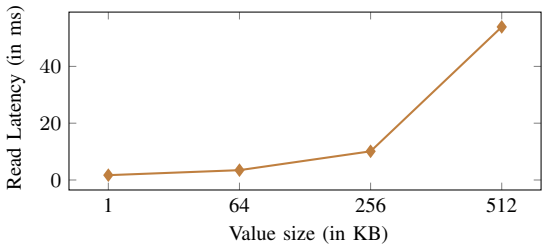


Fig. 7: Example of requests read latencies.

### A. *Value size estimation*

The objective of Héron is to process fast and slow requests in a way that avoids head-of-line-blocking. To reach this objective, Héron needs to predict whether a request will access a large (slow request) or a small value (fast request). The size threshold between these two types of requests is application-specific. We therefore assume that an application and database administrator will be able to set a threshold value $THR_L$ according to the data distribution over her database. For instance, for a database containing values ranging from 1 KB to 512 KB, the system designer may rely on an evaluation of the average read time latencies of these values as shown by Figure 7. This figure shows that a request accessing a 512 KB value is 32 times slower than a request accessing a 1 KB value, and that there is a significant gap in latency between values of sizes 256 KB and 512 KB. In this example, the system designer may set the threshold parameter $THR_L$ to 256 KB.

Héron uses Bloom filters to keep track of keys corresponding to large values. Bloom filters [26] are space-time efficient probabilistic data structures that allow performing set-membership queries (i.e., testing whether a given item belongs to a set). A Bloom filter is a vector of $m$ bits initially set to 0, with an associated set of $k$ hash functions (with $k \ll m$). Inserting an element in a Bloom filter is done by hashing the element (in our context the key contained in the request) using the $k$ hash functions and setting the corresponding bit positions to 1. Testing the presence of an element in a Bloom filter is done by hashing the element using the $k$ hash functions and testing whether *all* corresponding bit positions are set to 1. Querying

a Bloom filter may lead to false positives but will never lead to false negatives. The false positive rate depends on the size of the vector, the number of hash functions and the maximum number of elements to be inserted in the set. Héron sets these values so as to maintain the false positive rate below $0.1\%$.

*a) Constructing the Bloom filter:* We update the Bloom filter when new large values get inserted in the database. The filter is built at each of the replicas. For each write request, if the value size exceeds $THR_L$, the key is inserted into the Bloom filter. The Bloom filter is also updated when an existing small value is replaced by a value whose size exceeds $THR_L$.

We observe that in workloads such as the one from Facebook [14], different tables have different value distribution patterns. Some may contain only small values and be relatively homogeneous, while others are highly heterogeneous. To account for this fact, Héron allows administrators to set policies that disable the use of the Bloom filters for tables that have less than a configurable proportion of large values, as observed from the collected statistics, and avoid paying the overhead of querying the Bloom filter when it is not necessary.

*b) Synchronizing the Bloom filter between nodes:* The addition of information to the filter is performed at all of the replicas for a given key. The construction of a common global filter across all coordinators requires aggregating the filters from all replicas, i.e. keeping the result of the logical `or` operation between all filters. Due to the append-only nature of this construction, and to the idempotency of the aggregation, there is no need to complex synchronization involving a consensus protocol. Héron disseminates updates in an asynchronous, gossip-like, way. When the local filter is modified by a given node, upon the addition of a new large value or the replacement of a small value by a large one, it is piggybacked on the write acknowledgment sent to the coordinator. Coordinators gradually construct the global filter by interacting with storage servers, and merging newly-set bits to their local filter.

*c) Handling deletions and growth:* Deletion of large values (or their replacement by small values) would require removing their keys from the filter aggregated by coordinators. Bloom filters do not allow this operation, as un-setting the bits for the corresponding key comes with the risk of un-setting bits set for keys of other large values still present in the store. Handling deletions with a compact membership representation is actually only possible using more complex data structures, such as counting Bloom filters [27] or counting quotient filters [28]. These data structures have higher costs in memory and computation. More importantly, they are less amenable to the simple, asynchronous synchronization that Héron uses: their aggregation would require more costly consistency maintenance for coordinators.

Another linked issue is that of filters that happen to be insufficiently large after the growth of the dataset. In this case, the rate of false positives increases and the system is at risk of incorrectly considering too many keys as being associated with large values. Again, dynamically-resizable compact membership representations such as incremental Bloom filters [29] or counting quotient filters [28] can address this issue, but they also come at the cost of additional complexity in particular for aggregation and querying. As a result, Héron does not implement such features but relies on periodic system updates as detailed next.

*d) Periodic system updates:* Our system includes a number of updates that take place periodically, and using a low-priority background task. These tasks include the *regeneration* of Bloom filters and their synchronization, the periodic updates of table statistics and possibly the updates on the threshold for large values. Specifically, the datastore is periodically analyzed and the distribution of value sizes is updated. Using the gathered data, each storage node computes a new filter for the values it owns and whose size exceeds the threshold. The parameters of this filter (size, number of hash functions) may change according to some administrator-defined policy, e.g. if the size of the store exceeds the value initially estimated. Coordinators aggregate filters for several generations and start using the latest generation as soon as they have received an update for it from all storage servers. Similarly, the list of tables that contain a given proportion of large requests is updated. Finally, if the distribution of values changes dramatically, the administrator gets informed and may possibly decide to adjust the threshold for large values accordingly.

### B. Replica scoring

Héron includes a replica scoring mechanism similar to the one used by C3 [13]. Coordinators periodically collect as scoring metric for each server the product of its average service time and its queue size. Replicas with a lower score are better candidates for serving incoming requests, if all requests are for value of the same size. The difference with C3 is that Héron does not rely on a control flow mechanism to balance the load between servers. Instead, Héron uses differentiated scheduling in which a dynamic assignment policy is used to schedule requests as described in the following.

### C. Replica selection

The replica selection selects the replica that is expected to serve an incoming request faster than the other replicas. It uses three types of information: (i) whether the request is expected to access a small or a large value, as provided by the Bloom filter manager; (ii) the relative score of the servers holding replicas for that key, as provided by the replica scoring module; (iii) whether these replicas are currently handling a request for a large value of not. The last information is maintained over time by the replica selection module. Specifically, the initial status of a replica having no request to process is set to *available*. As long as this replica processes requests accessing small values, its status remains *available*. Instead, when a request accessing a large value is scheduled on a given replica, its status becomes *busy*. The latter comes back to the *available* status when the processing of the large request is over.

*a) Scheduling of requests tagged as large:* If the request is tagged as large by the Bloom filter manager and if there exists a replica $R$ whose status is *available*, then the request is sent to $R$ and $R$'s status becomes *busy*. If there is more

than one replica whose status is *available*, the replica selection module uses the scores provided by the replica scoring module to chose the one that is expected to be the fastest. After finishing processing the request, the replica selection module updates the status of $R$ to *available*. If there is no *available* replica, then the request is blocked until at least one of the replicas becomes *available*.

*b) Scheduling of requests tagged as small:* If the request to schedule is tagged as small by the Bloom filter manager, the replica manager uses the first replica server whose status is *available* from the ordered list provided by the replica scoring module. In this situation, the replica's status remains unchanged since short jobs are not expected to affect tail latency by head-of-line blocking.

*c) Synchronizing information about replicas' status:* A coordinator relies on its local knowledge of the replica status for making scheduling decisions. This information is synchronized in a best-effort manner between replicas. Specifically, the propagation of this information leverages existing communication between servers and coordinators for requesting queues sizes and service times for replica scoring. Storage servers include their current status with all exchanged messages and coordinators update their replica status table accordingly when receiving these messages. This asynchronous and best-effort propagation may lead to inconsistent views between coordinators about the status of a given storage server. For instance, a given coordinator may schedule a request to a given replica and locally update the latter's status to *busy*. At the same time, another coordinator can schedule small requests into this same replica because it did not yet receive the feedback that this node was busy. In this situation, the head-of-line-blocking problem may occur but only for a limited period of time. We consider this to be an acceptable compromise as a fully consistent exchange of information about replica status would greatly impair the horizontal scalability of the key/value store and its overall performance. Our evaluation show that Héron still drastically reduces tail latencies despite the potential inconsistencies in replica state tables.

## V. Evaluation

We implement Héron as an extension of the Cassandra [12] key/value store. We evaluate its effectiveness in reducing tail latency using both synthetic datasets generated using the Yahoo Cloud Serving Benchmark (YCSB) [18], and real datasets from WikiMedia [17] and Flickr [16]. We compare the reduction in tail latency with DS [12] and C3 [13]. We conduct the experiments on a public high-performance cluster representative of current cloud data centers hardware [15]. We analyze the impact of three types of heterogeneous workloads, i.e., read only (100% read), read heavy (95% read-5% write) and update heavy (50% read-50% write), with varying ratios of requests for large values and varying large value sizes.

Our evaluation aims at answering the following questions:

1) How does Héron compare to C3 and DS when the dataset contains a mix of small and large values? (§V-B1)
2) How is the performance of Héron impacted by the proportion of large values in the key-value store? (§V-B2)
3) How does Héron perform when the proportion of read vs write requests varies? (§V-B3)
4) Is the impact of Héron confirmed with real datasets? (§V-C)

We start this section by presenting our evaluation setup (§V-A) before presenting our results (§V-B and V-C).

### A. Experimental setup

**Experimental platform.** We evaluate Héron on Grid5000 [15]. We use 15 servers equipped with 2 Intel Xeon E5-2630 v3 CPUs (8 cores per CPU), 128 GB of RAM, and 2 558 GB HDDs. Servers are interconnected by a 10Gbps Ethernet network and run the Debian 8 GNU/Linux OS.

**Cassandra configuration.** We use a replication factor of 3, which means that each value is available on three replica servers. We consider a write-all read-from-one coherency mechanism in which consistency is achieved by reading from a single replica and not from a quorum. Each experiment involves 2 million requests accessing small and large values. We systematically check that Cassandra achieves its maximum throughput (i.e. that we use enough clients to saturate the system). The measured peak throughput depends on the proportion of requests for large values in the system. Figure 8 shows the maximal achieved throughput across all experiments and value sizes in this section. When there are no requests to large values, the throughput peaks at $\approx$ 72000 requests/sec. It reduces to $\approx$ 2200 requests/sec when 20% of the requests are for large values.

**Synthetic datasets.** We use the industry standard Yahoo Cloud Serving Benchmark (YCSB) [18] to generate synthetic workloads. YCSB originally only generates workloads with values of a single size. We modified its source code to generate a configurable proportion of large and small values. The size of small and large values is also configurable. The distribution of access frequency for all stored values (small and large) follows a Zipfian distribution with a pareto index of $\alpha = 0.99$. This means informally that the most popular value is almost twice as popular as the second-most-popular value, and so on with decreasing popularities. With a replication factor of 3 and 15 servers, each storage node holds about 170 GB of data.

**Real datasets.** We evaluate Héron with the publicly available Flickr [16] and WikiMedia [17] datasets, which contain 1 million images and 225 thousand images, respectively. The CDF of image sizes on these datasets is shown by Figure 2. We use YCSB [18] to generate workloads based on these datasets and considered the 10% largest values of each dataset as large in all scenarios. The access frequency distribution for stored values (small and large) follows the same Zipfian distribution as for the synthetic workloads.

### B. Héron on variable configurations of the synthetic dataset

We evaluate Héron along three dimensions of heterogeneous workloads, i.e., the varying size of large values (§V-B1), the ratio of requests for large values (§V-B2), and types of workloads (§V-B3). We compile the absolute latency values

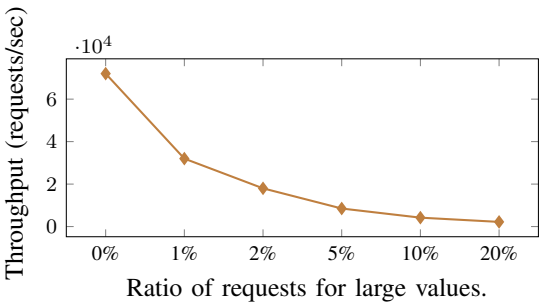


Fig. 8: Maximum throughput attained across all scenarios.

| Dataset | Update Heavy | | Read Heavy | | Read Only | |
|---|---|---|---|---|---|---|
| | $95^{th}$%ile | $99^{th}$%ile | $95^{th}$%ile | $99^{th}$%ile | $95^{th}$%ile | $99^{th}$%ile |
| 64 KB | 12.5 ms | 27.8 ms | 21.2 ms | 33.7 ms | 24 ms | 41.8 ms |
| 128 KB | 25.2 ms | 41.3 ms | 51 ms | 101 ms | 47 ms | 86 ms |
| 256 KB | 50 ms | 81 ms | 95 ms | 160 ms | 105 ms | 209 ms |
| 512 KB | 130 ms | 218 ms | 196 ms | 340 ms | 199 ms | 336 ms |
| 1% | 19.1 ms | 57.2 ms | 35.2 ms | 89.6 ms | 35.3 ms | 96.2 ms |
| 2% | 38.2 ms | 92.7 ms | 58.8 ms | 114.6 ms | 65 ms | 134.2 ms |
| 5% | 77.6 ms | 140 ms | 121 ms | 216 ms | 139 ms | 272 ms |
| 10% | 130 ms | 218 ms | 196 ms | 340 ms | 199 ms | 336 ms |
| 20% | 183 ms | 310 ms | 336 ms | 549 ms | 349 ms | 574 ms |

TABLE II: Héron absolute performance measurements.

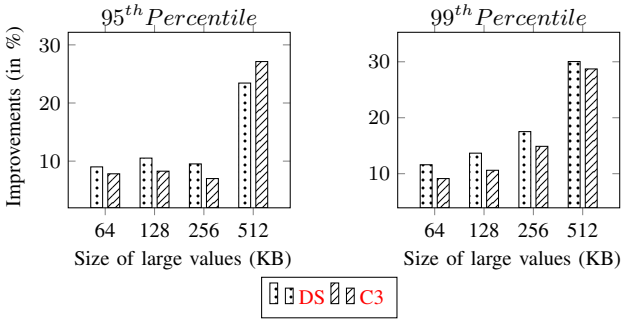


Fig. 9: Improvement of tail latency with Héron for different sizes of large values.

measurements of Héron in Table II. We will refer to this table on the three subsequent sections. We skip the presentation of absolute values for DS and C3 in the interest of space, but present the relative improvements between the different systems in Figures 9, 10 and 11.

*1) Varying the size of large values:* We start by studying the impact of the size of large values. We fix the proportion of requests to large values to 10%. We vary the size of large values between 64 KB, 128 KB, 256 KB and 512 KB. By looking at the table of absolute values (Table II), we first note that, when the size of values increases in the database (e.g., 64 KB values versus 512 KB values), tail latencies are higher because the system takes more time to execute the overall workloads. Moreover, in the case of values of size 512 KB, the probability of running into head of line blocking in Héron is higher than in the case of 64 KB. This is not surprising as requests for large values occupy servers for a longer period of time.

We present the percentage of improvement of Héron over DS and C3 for read heavy workloads in Figure 9. For instance,

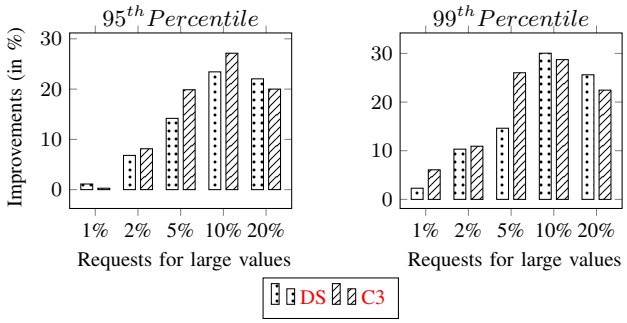


Fig. 10: Improvement of Héron for different proportion of request for large values.

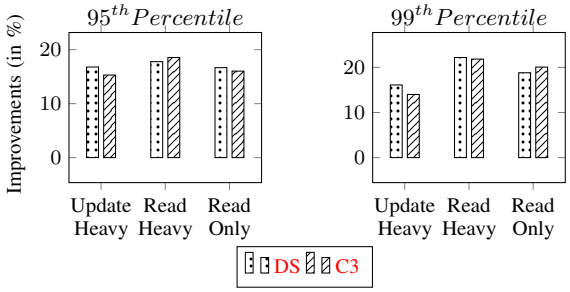


Fig. 11: Improvement of Héron over the average of all heterogeneous workloads.

the improvement of Héron over DS for large values of size 64 KB is 9% for the $95^{th}$ percentile, meaning that the tail latency at this percentile of Héron is 91% that of DS under the same conditions. The read latency with Héron is lower than the read latency of both C3 and DS in all tested configurations. More importantly, the improvement increases with the value sizes, i.e., Héron achieves higher latency reduction than DS and C3. For large value sizes of 64 KB, 128 KB, and 256 KB, the improvements over C3 and DS are relatively modest, ranging between 8%-10% and 12%-17% for $95^{th}$ and $99^{th}$ percentiles tail latency, respectively. For large values sizes of 512 KB, Héron improves the tail latency by up to 30% over DS and 28% over C3. This is explained by the fact that Héron is able to mitigate the waiting time of request for small values by avoiding scheduling them behind requests for large values. As C3 and DS are oblivious to the size of the requested value, not only requests for small but also request for large values can be scheduled behind requests for large values, significantly degrading tail latency. Even though not represented in the interest of space, our measurements further show that the improvement brought by Héron for median latency ranges from 1.5 ms to 13 ms for large value sizes ranging from 64 KB to 512 KB.

*2) Varying the proportion of requests for large values:* We study the impact of the proportion of requests for large value on system performance. We fix the size of large values to 512 KB and we vary their proportion from 1% to 20% for all three types of workloads. From the absolute values reported in Table II, for both the $95^{th}$ and $99^{th}$ percentiles and for all

three workload types we observe that the latency increases with the percentage of requests for large values, which is expected as these requests take longer to execute.

We present the improvement of Héron over DS and C3 for read heavy workloads in Figure 10. We can observe that Héron outperforms DS and C3 in all configurations. Further, we observe that the effectiveness of Héron increases with the percentage of requests for large values. With 10% of requests for 512 KB values, Héron achieves an improvement of around 25%; however the improvement slightly drops in the case of 20% of such requests. This can be explained by the fact that the probability that all servers are blocked by a request for a large value is higher in the case of 20% of such requests than the same probability in the other configurations. In this situation, Héron looses part of its ability to dispatch the requests agilely among servers. In the following we zoom into each specific percentage of requests for large values.

**95-5.** In this experiment, 5% of the values in the database are large ones. Héron shows similar improvement over DS, i.e., roughly 15% for the $95^{th}$ and $99^{th}$ percentile of tail latency. Compared to C3, Héron achieves better gain, i.e., roughly 26% improvement for the $99^{th}$ percentile.

**90-10.** In this experiment, 10% of the values of the database are large ones. Héron shows similar improvement over C3, i.e., around 27% for the $95^{th}$ percentile. Against DS, Héron achieves a slightly better gain for the $99^{th}$ percentile than for the $95^{th}$ percentile, i.e., roughly 23% v.s. 28%. In terms of absolute latency, say for the $99^{th}$ percentile, it is 340 ms for Héron but roughly 486 and 477 ms for DS and CS, respectively. Héron achieves the best performance for this workload, i.e., 10% of requests for 512 KB large values, where there is a sufficient number of available servers for Héron to schedule requests for large values without blocking incoming requests.

**80-20.** In this experiment, 20% of the overall values are large ones. Héron shows consistent improvement over C3 and DS for both the $95^{th}$ and $99^{th}$ percentiles, i.e., roughly 23% and 20%. This increases up to 41% in case of the $99.9^{th}$ percentile tail latency. In terms of absolute latency, for instance for the $95^{th}$ percentile, it is 336 ms for Héron but roughly 431 and 420 ms for DS and CS, respectively. The gap is even more significant for the $99^{th}$ percentile, where the observed latencies for Héron, DS, C3 are 549 ms, 738 ms, and 708 ms, respectively.

In addition to tail latency, we also report the absolute values of the median latency under the three workloads (Figure 12). We observe that Héron has a comparable median latency to the one of DS and C3 when the proportion of requests for large values is small $(1 - 5\%)$. When the proportion of requests for large values exceeds $5\%$ we can observe that Héron improves over DS and C3 by up to 35%.

In summary, Héron is particularly more effective in reducing tail latency when the percentage of requests for large values is higher. Compared to C3 and DS, Héron reduces tail latencies by up to 41% without compromising median latency in all the considered synthetic workloads.

*3) Consolidated performance improvement:* We analyze the average improvement over 8 heterogeneous read only, read heavy, and update heavy workloads. Figure 11 shows the average improvement over all configurations (proportion of requests for large values, and different large value sizes, as detailed in Table II). Héron achieves the highest improvement for read heavy workloads $\approx 23\%$ and the smallest improvement for update heavy workloads $\approx 14\%$. Héron takes scheduling decisions for read requests only, as write requests have to reach all replicas in all cases. It can better cut the tail latency for workloads that contain higher percentages of read requests, such as read only and read heavy workload. On the other hand, reading requests take longer time than updating requests in our particular Cassandra setting (as also illustrated in Table II), as writes can be buffered to memory while reads most often have to reach the servers' disks. Hence, a higher percentage of read requests can further stress the system. The average response time of the read only workload is therefore higher than the one of the read heavy workload.

Finally, we show the consolidated observation of the average number of requests for small values scheduled behind requests for large values over all experiments in Figure 13. This confirms our initial intuition that C3 avoids overwhelming

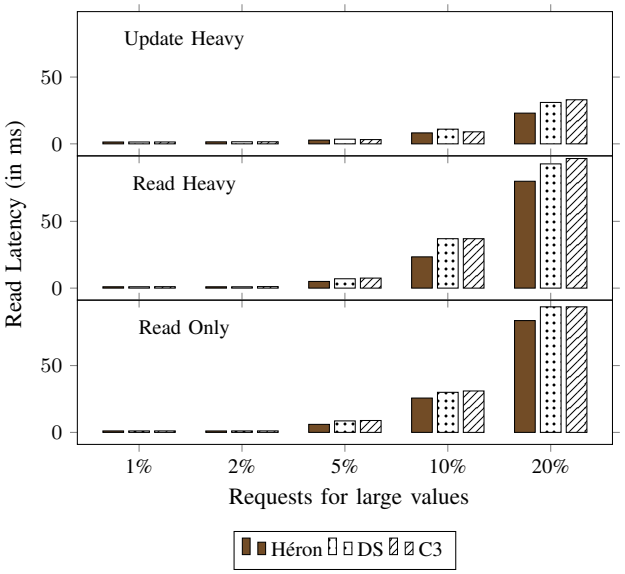


Fig. 12: Median latency over different proportions of request for large values.

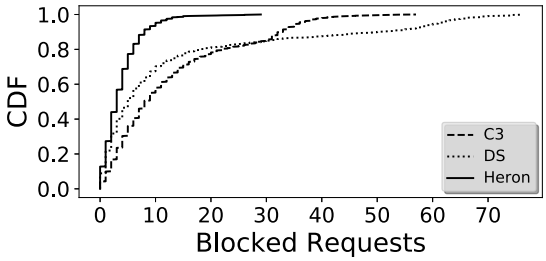


Fig. 13: Impact of head-of-line-blocking when small requests are queued behind large requests.

well-performing replicas with requests compared to DS, but will let requests accumulate behind requests for large values, leading to a large number of blocked requests in most cases. On the other hand, the adaptive strategy used by Héron allows limiting the number of blocked requests in the queues of all replicas, limiting the impact of head-of-line blocking.

### C. Real workloads

We now proceed to evaluating the performance of Héron in reducing tail latency under value sizes distributions obtained from two datasets from WikiMedia and Flickr. The distribution of these sizes is shown by Figure 2. We generate three access workloads, update heavy, read heavy and read only, using the same proportions of accesses as for the previous experiments. For each dataset, we compute the median, $95^{th}$, $99^{th}$, and $99.9^{th}$ percentile latencies for Héron, DS and C3.

Figure 14 present the results for the WikiMedia trace. Héron yields the lowest latency for almost all combinations of latency metrics and workload types, whereas the performance of DS and C3 vary across different combinations. Similar to the synthetic case, absolute latency improvements of Héron are more significant for the higher tail latency, e.g., $99^{th}$ percentile. Moreover, Héron achieves the best gain for the read heavy workloads, up to 70%. The only case where Héron has inferior performance compared to C3 and DS is for the median of the read only workload. The latency of Héron remains around 1 ms. Actually, Héron achieves rather minor performance improvements in both median and tail latency against DS and CS for read only workload in this dataset.

We present the results for the Flickr dataset in Figure 15. We first note that these results show different latency characteristics from WikiMedia, i.e., lower tail latency, though their median latency is in a similar range. This can be explained by the fact that the Flickr dataset value size distribution has a shorter tail, that is, the maximal value for sizes is smaller than for the WikiMedia dataset. Here, Héron achieves the lowest latency for almost all combinations of latency metrics and workload types, except for the 99.9 percentile of read heavy workloads. The overall improvement compared to C3 and DS is also less significant than with the WikiMedia dataset. Héron is particularly designed to handle the heterogeneous workload that have a high variance across requests' sizes. When the sizes of requested values have lower variability, the impact of size-aware scheduling becomes less visible, even if they are still present. Different from the synthetic data set and WikiMedia, Héron achieves the best median and tail latency improvement for update heavy workloads, compared against read only and read heavy workloads.

## VI. Related work

C3 [13] is a replica selection algorithm as discussed earlier in this paper, since the incoming request size was assumed to be the same, C3 does not perform well with heterogeneous workloads. Héron complements this feedback on service time and queue size with a differentiated scheduling of requests according to their estimated value size.

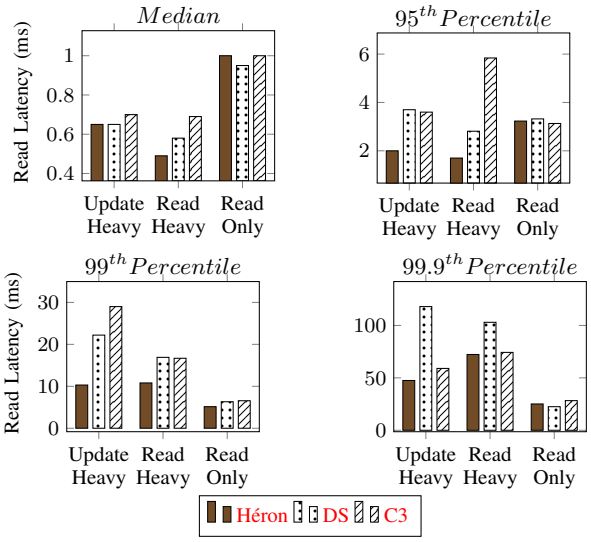


Fig. 14: WikiMedia Dataset

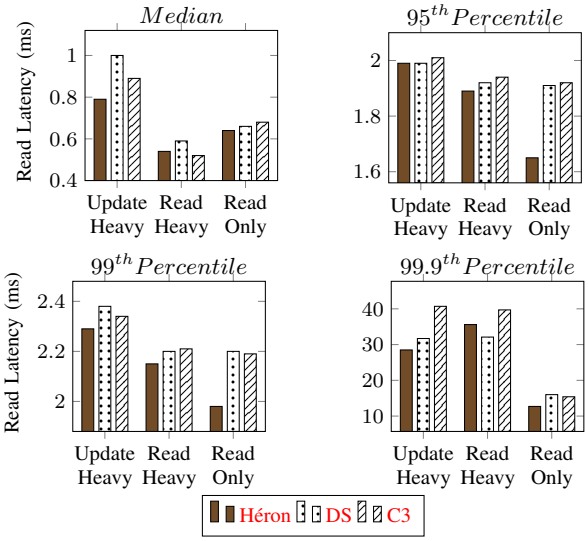


Fig. 15: Flickr Dataset

Besides replica section, other mechanisms have been proposed to reduce tail latency. Dean and Barroso [3] analyze the reasons for latency variability and describe a set of tail-latency tolerance techniques implemented in Google's large scale systems. Reda et al. propose Rein [30] for reducing tail latency of Multiget requests, where multiple keys are requested in a single query. CosTLO [31] reduces high latency variance by issuing requests redundantly. D-SPTF [32] adapts caching mechanisms to reduce tail latency. Li et al. [33] explore the hardware, OS and application-level causes behind tail latencies and propose mechanisms to overcome these problems. All these mechanisms can be used complementary to replica selection.

Task scheduling for large computation jobs is another area of related research. In task aware scheduling, a task decomposes into tens to hundreds sub-tasks. A slow sub-task can slowdown the overall response time. Related work shows the impact of large sub-tasks over small sub-tasks for overall tail latencies.

Baraat [34] dynamically changes the level of multiplexing in the network to avoid blocking. It schedules the tasks in FIFO order such that small tasks are not starved behind large tasks. Hawk [11] and Eagle [24] are two systems proposing a hybrid scheduler that schedules jobs according to their sizes. Long jobs are scheduled using a centralized scheduler while small jobs are scheduled in a fully distributed way. Since long jobs are in fewer number than small jobs, their centralized scheduling allows a good placement of jobs without introducing coordination bottlenecks. Kwiken [8] optimizes the end-to-end latency using a DAG of interdependent jobs. It further uses latency reduction techniques such as request reissues to improve the latency of request-response workflows. Haque et al. [9] propose solutions for decreasing tail latencies by dynamically increasing the parallelism of individual requests. They parallelize only long requests that are the ones contributing the most to the tail latency. However the algorithm parallelizes long requests on all servers thus also including slow servers. Héron takes advantage of the fact that in a key/value store data is replicated on multiple servers and can perform replica selection.

Recent efforts [9], [10], [35] show that it is challenging to schedule tasks during the arrival of variable size jobs. These works try to predict the long running queries and parallelize them selectively. Instead of targeting the more general problem of predicting job sizes, which in some cases involves costly computations, Héron keeps track of value sizes by relying on Bloom filters for fast and efficient estimation.

## VII. Conclusion

In this paper, we addressed the problem of tail latency for heterogeneous workloads in key value stores through replica selection. We proposed Héron, a replica selection algorithm that deals with requests accessing large values by avoiding the head-of-line-blocking of requests accessing small requests behind these requests. The result is an improved overall performance of the key-value-store for a wide variety of heterogeneous workloads. Our experiments with heterogeneous YCSB workloads in a Cassandra based implementation showed that Héron outperforms state-of-the-art algorithms (C3 and DS), reducing tail latencies by up to 41% and reducing the median latency by up to 31%.

## Acknowledgments

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations. This work is supported by the French PIA OCCIware project, by CHIST-ERA under project DIONASYS, and by the Swiss National Science Foundation (SNSF) under grant 155249 and NRP75 project 407540_167266.

## References

[1] E. Schurman and J. Brutlag, "Performance related changes and their user impact," in *Velocity: web performance and operations conference*, 2009.
[2] J. Brutlag, "Speed matters for google web search," *Google. June*, 2009.
[3] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, 2013.
[4] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: Distributed, low latency scheduling," in *SOSP*, 2013.
[5] 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.
[6] C. Stewart, A. Chakrabarti, and R. Griffith, "Zoolander: Efficiently meeting very strict, low-latency SLOs," in *ICAC*, 2013.
[7] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca, "Jockey: Guaranteed job latency in data parallel clusters," in *EuroSys*, 2012.
[8] V. Jalaparti, P. Bodik, S. Kandula, I. Menache, M. Rybalkin, and C. Yan, "Speeding up distributed request-response workflows," in *SIGCOMM*, 2013.
[9] 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*, 2015.
[10] M. Jeon, S. Kim, S.-w. Hwang, Y. He, S. Elnikety, A. L. Cox, and S. Rixner, "Predictive parallelization: Taming tail latencies in web search," in *SIGIR*, 2014.
[11] P. Delgado, F. Dinu, A.-M. Kermarrec, and W. Zwaenepoel, "Hawk: Hybrid datacenter scheduling," in *USENIX ATC*, 2015.
[12] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, 2010.
[13] L. Suresh, M. Canini, S. Schmid, and A. Feldmann, "C3: Cutting tail latency in cloud data stores via adaptive replica selection," in *NSDI*, 2015.
[14] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *SIGMETRICS*, 2012.
[15] D. Balouek, A. Carpen Amarie, G. Charrier, F. Desprez, E. Jeannot, E. Jeanvoine, A. Lèbre, D. Margery, N. Niclausse, L. Nussbaum, O. Richard, C. Pérez, F. Quesnel, C. Rohr, and L. Sarzyniec, "Adding virtualization capabilities to the Grid'5000 testbed," in *Cloud Computing and Services Science*, 2013.
[16] M. J. Huiskes and M. S. Lew, "The MIR Flickr retrieval evaluation," in *MIR*, 2008.
[17] "Wikimedia downloads," http://download.wikimedia.org/.
[18] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *SoCC*, 2010.
[19] "Mongodb," https://www.mongodb.com/.
[20] "Openstack swift," https://docs.openstack.org/swift/latest/.
[21] "Apache accumulo," https://accumulo.apache.org/.
[22] "Riak Load Balancing and Proxy Configuration," http://docs.basho.com/riak/1.4.0/cookbooks/Load-Balancing-and-Proxy-Configuration/.
[23] 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.
[24] P. Delgado, D. Didona, F. Dinu, and W. Zwaenepoel, "Job-aware scheduling in Eagle: Divide and stick to your probes," in *SoCC*, 2016.
[25] J. Lenstra, A. R. Kan, and P. Brucker, "Complexity of machine scheduling problems," in *Studies in Integer Programming*, 1977.
[26] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, 1970.
[27] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, "An improved construction for counting bloom filters," in *European Symposium on Algorithms*, 2009.
[28] P. Pandey, M. A. Bender, R. Johnson, and R. Patro, "A general-purpose counting filter: Making every bit count," in *SIGMOD*, 2017.
[29] F. Hao, M. Kodialam, and T. V. Lakshman, "Incremental bloom filters," in *INFOCOM*, 2008.
[30] W. Reda, M. Canini, L. Suresh, D. Kostić, and S. Braithwaite, "Rein: Taming tail latency in key-value stores via multiget scheduling," in *EuroSys*, 2017.
[31] Z. Wu, C. Yu, and H. V. Madhyastha, "Costlo: Cost-effective redundancy for lower latency variance on cloud storage services," in *NSDI*, 2015.
[32] C. R. Lumb, R. Golding, and G. R. Ganger, "D-SPTF: Decentralized request distribution in brick-based storage systems," in *ASPLOS*, 2004.
[33] 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*, 2014.
[34] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron, "Decentralized task-aware scheduling for data center networks," in *SIGCOMM*, 2014.
[35] M. Jeon, Y. He, H. Kim, S. Elnikety, S. Rixner, and A. L. Cox, "TPC: Target-driven parallelism combining prediction and correction to reduce tail latency in interactive services," in *ASPLOS*, 2016.