TailX: Scheduling Heterogeneous Multiget Queries to Improve Tail Latencies in Key-Value Stores

Vikas Jaiman, Sonia Ben Mokhtar, Etienne Rivière

TailX: Scheduling Heterogeneous Multiget Queries to Improve Tail Latencies in Key-Value Stores

Vikas Jaiman[1,2], Sonia Ben Mokhtar[3], and Etienne Rivière[2]

[1] Université Grenoble Alpes, LIG (CNRS UMR 5217), France
er.vikasjaiman@gmail.com
[2] ICTEAM, UCLouvain, Belgium
etienne.riviere@uclouvain.be
[3] INSA Lyon, LIRIS, CNRS, France
sonia.benmokhtar@insa-lyon.fr

Abstract. Users of interactive services such as e-commerce platforms have high expectations for the performance and responsiveness of these services. Tail latency, denoting the worst service times, contributes greatly to user dissatisfaction and should be minimized. Maintaining low tail latency for interactive services is challenging because a request is not complete until all its operations are completed. The challenge is to identify bottleneck operations and schedule them on uncoordinated backend servers with minimal overhead, when the duration of these operations are heterogeneous and unpredictable. In this paper, we focus on improving the latency of multiget operations in cloud data stores. We present TailX, a task-aware multiget scheduling algorithm that improves tail latencies under heterogeneous workloads. TailX schedules operations according to an estimation of the size of the corresponding data, and allows itself to procrastinate some operations to give way to higher priority ones. We implement TailX in Cassandra, a widely used key-value store. The result is an improved overall performance of the cloud data stores for a wide variety of heterogeneous workloads. Specifically, our experiments under heterogeneous YCSB workloads show that TailX outperforms state-of-the-art solutions and reduces tail latencies by up to 70% and median latencies by up to 75%.

Keywords: Distributed storage · Performance · Scheduling.

1 Introduction

Serving users requests in interactive applications or websites generally involves handling a number of operations to backend services and databases. For instance, the display of a social network page may involve fetching and aggregating a number of images, posts, ads, etc. NoSQL cloud databases increasingly offer *multi-get* operations in their APIs, enabling to fetch values associated with a collection of keys with a single call [3, 17, 27, 32]. In practice, multiget requests vary in the number of accessed keys and value size. A workload analysis at Facebook [32] shows that a request contains an average of 24 keys while 5% of the requests contain more than 95 keys. Another analysis from a SoundCloud trace presented by the authors of Rein [35] shows a heavy-tailed distribution of the number of keys: 40% of the requests involve multiple keys with an average size of 8.6 keys and the maximum number of keys reaches up to ∼2,000 keys. Similarly, another analysis of key-value stores production workloads at Facebook [4] shows that value size typically ranges from a few Bytes to several MBs: Value sizes are highly skewed towards smaller sizes but very few large value sizes consume a large share of computational resources [11].

A multiget request finishes when all of its operations complete. The response time of a request depends on the response time of the slowest operation in that multiget request and, as a result, multiget operations are affected more often by high *tail latencies* [11, 15, 17, 28]. Reducing tail latency is of uttermost importance in online services, as high service delays may have serious consequences on user quality-of-experience and satisfaction.

Several past works have considered the problem of reducing tail latency by scheduling single-key requests in key-value stores [20, 24, 37]. These approaches offer solutions to the *head-of-line-blocking* problem that results from the heterogeneity in the value sizes stored in the database: single-key requests for small values may get scheduled after a request for a large value (incurring, therefore, a long processing time). Requests for small values may be delayed after requests for large values, increasing average and tail latencies. In contrast, other works have considered the scheduling of multiget requests in key-value stores [14, 35], but under the assumption of homogeneous service times for operations, i.e., of requests for fixed-size values. Scheduling multiget requests is more involved than scheduling single-key requests but also offers more opportunities when it is performed in a task-aware manner, i.e., when taking into account the entirety of the request for scheduling its constituents rather than considering these constituents independently. In particular, as the completion time of a multiget requests is, *in fine*, that of its longest operation, a task-aware scheduling algorithm may decide to delay the processing of non-critical operations of a multiget request in favor of more critical operations of another multiget request. The occurrence of long operations is intrinsically linked with the number but also with the size of the values fetched by these requests and, thus, by the heterogeneity in the size of queried data.

Contributions. We present TailX, a task-aware multiget scheduling algorithm that reduces tail latencies under heterogeneous workloads i.e. (i) when multiget requests are formed of operations for values of different sizes and (ii) when the number of operations for different multiget requests vary. TailX addresses two key challenges associated with the scheduling of multiget requests in a distributed, horizontally-scalable key-value store:

- First, a multiget request arrives at an entry point server in the key-value store, called the *coordinator*, which must split it into multiple sub-requests called *opset*, fetch values from different replicas, and send an aggregated response to the client. Selecting the appropriate replica for each opset must be performed in an online fashion, and service time cannot be known *a priori* and based solely on the keys. In other words, requests are processed in a non-clairvoyant fashion [31]. This is a result of two factors: (i) the load at the different replicas (amount of pending requests) is unknown by the coordinator and (ii) the size of the values corresponding to the keys is known by the replicas who hold them, but unknown to the coordinator that performs request splitting and replica selection.
- Second, once an opset reaches the selected replica, it must be scheduled for execution at that replica based on the overall execution time for the corresponding multiget request. Ideally, opset that are more critical for the overall execution time of a multiget query should be executed with higher priority than opset that are not as critical. The notion of "criticality" of a specific opset is, however, unknown to the replica, as the knowledge of the overall multiget requests is at the coordinator. As a result, a replica may take non-optimal decisions in processing opset, such as answering opset that could have been postponed without impacting the latency of the corresponding multiget requests, and conversely postponing critical opset.

TailX implements the sharing of information between coordinators and replicas and associated algorithms for end-to-end, task-aware scheduling of multiget requests:

- For coordinators, it enables awareness of the load of the different replicas and awareness of the size of values associated with given keys. The necessary information is exchanged between all nodes (coordinators and replicas–in many designs, nodes assume both roles) using an efficient and fast gossip protocol. The load of replicas, as indicated by the length of their queues of pending requests, enables avoiding overloads and reduces the impact of head-of-line-blocking. As sharing globally a map between all keys and the size of corresponding values would be impractical in terms of costs and scalability, and as request splitting and scheduling happen in the critical path of the request/response loop, TailX favors the pragmatic and efficient use of a compact data structure–a Bloom

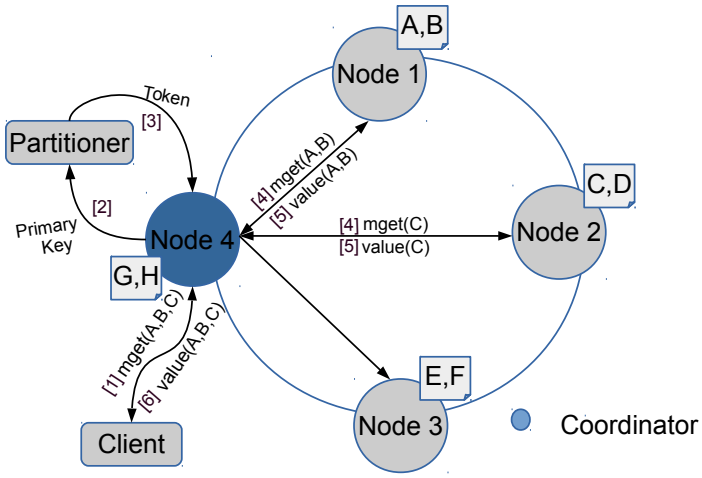


Fig. 1: Handling of a multiget request in Cassandra.

filter [6]–that probabilistically indicates keys that are associated to large values (i.e. above a threshold size).

– For replicas, TailX scheduling takes into account the possible influence of opset on tail latency and supports *procrastinating* non-critical opset in favor of the execution of more critical opset. These decisions are based on information embedded by the coordinator in an opset, indicating how much this opset is estimated to be allowed to *wait* before it can influence negatively the latency of its enclosing multiget request.

We implement TailX in the industry-grade key/value store Cassandra [25]. We compare TailX with Rein [35], a state-of-the-art algorithm for multiget requests scheduling, using a deployment on a cluster of 16 servers on the Grid'5000 testbed [5]. We use YCSB [10] to generate workloads that contain various proportions of accessed keys and value size, based on the description of production traces by Facebook [4]. Compared to Rein, TailX improves median latency by 75% as well as tail latency by up to 70%.

The remaining of this paper is structured as follows. We first present background on multiget scheduling in key-value stores (§2) and explain state-of-the-art algorithms. Next, we further detail the design of TailX (§3) and present its implementation and performance evaluation (§4). Finally, we discuss related work (§5) and conclude the paper (§6).

2 Multiget requests in key-value stores

We detail the execution of multiget queries in key-value stores with the example of Cassandra [25]. We note that the operation of other horizontally scalable, hash-partitioned key-value stores [3, 17, 27, 32] supporting multiget queries are very similar. In the example of Figure 1, nodes 1, 2 and 3 are replicas for the values associated with keys (A,B), (C,D) and (E,F) respectively. The example uses a single replica per key, but replication is used in practice to guarantee data availability. A client sending a multiget request mget(A, B, C) connects to any of the nodes that will act as coordinator (step 1). The coordinator uses a partitioner that returns tokens, as hash values for these keys (steps 2 and 3). These tokens together with the knowledge of the replication policy allow identifying the replicas holding copies of the values associated with the keys. The coordinator is in charge of (1) splitting

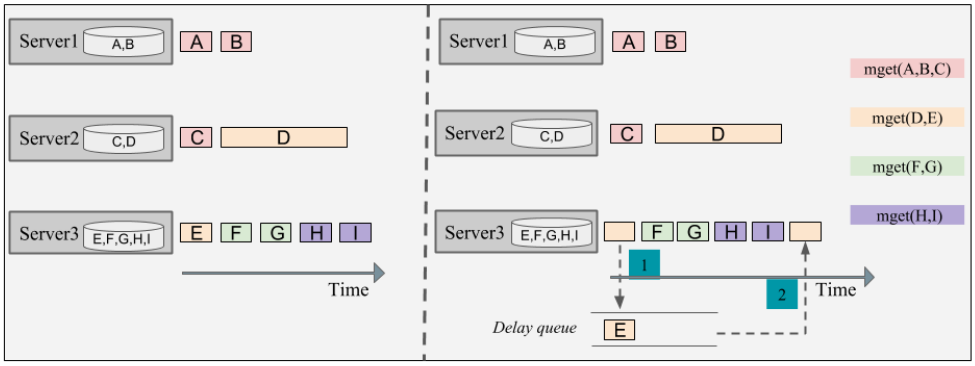


Fig. 2: An example scenario. *Left:* Requests assigned to server facing delayed response time. *Right:* Procrastinate opsets into delay queue to take benefits of delay allowance

the multiget request into a set of requests for one or more keys and (2) fetching the values from the corresponding replicas (steps 4 and 5). When all opsets have been answered, the coordinator may serialize the result and send it back to the client (step 6).

We illustrate the difficulty in scheduling multi-get requests efficiently to obtain low overall latencies with an example in Figure 2 where the same request mget(A, B, C) is processed in a system where other single-key and multiget requests are ongoing. On the left of Figure 2 servers 1, 2, 3 hold values for keys (A, B), (C, D) and (E, F, G, H, I) respectively. A small box represents a request to a small value and a large rectangle box (in this example for key D) represents a request to a large value. For the sake of simplicity, we assume that all replicas have a service time of 1 operation per unit time for serving a small value and of 5 unit time for serving a large value. For the request mget(A,B,C), (A,B) and (C) are the two opsets. With a FIFO scheduling as shown on the left of the figure, mget(A, B, C), mget(D, E), mget(F, G) and mget(H, I) will complete in 2, 6, 3 and 5 time units respectively, yielding an average response time of 4 time units.

We note that task awareness in scheduling individual opset at the replicas can allow reducing the average response time. A key observation is that each opset can be associated with a *delay allowance* that the replica can use to schedule other operations from its queue with higher priority (and, therefore, not necessarily in FIFO order). The delay allowance can be calculated as the difference in time between an approximated execution time for the largest or costliest opset. In the multiget request mget(D, E), the collection of D takes 6 time units whereas the collection of E will take 1 time unit. It is, therefore, possible to postpone (or *procrastinate*) the request for key E by at most 5 time units, leaving way for other requests. In this scenario, on the right side of the figure, mget(A, B, C), mget(D, E), mget(F, G) and mget(H, I) will complete in 2, 6, 2 and 4 time units respectively yielding an average response time of 3.5 time units.

Multiget scheduling state-of-the-art. The state-of-the-art in multiget requests scheduling is represented by Rein [35]. It uses two policies which include the Shortest Bottleneck First (SBF) and Slack-Driven Scheduling (SDS). In SBF, every operation of a multiget request has a priority which corresponds to the cost of the bottleneck opset while in SDS, it deprioritizes the operations based on how long they can afford to be slacked. The goal of Rein is to improve tail latency. To this end, Rein predicts which of the operations will likely be a bottleneck, i.e. create a head-of-line-blocking situation. This detection is based on the number of keys in the opset. The opset(s) with the highest number of keys is or are simply considered as the bottleneck opset. Based on this information, Rein uses a client-side

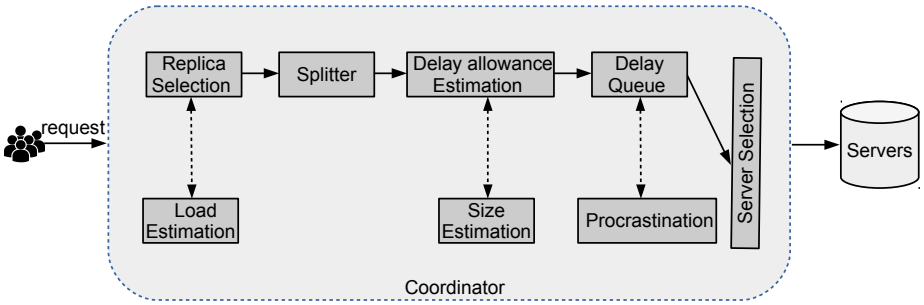


Fig. 3: Overview of TailX.

priority assignment that prioritizes multiget requests with a smaller number of keys in their bottleneck opset. The determination of the bottleneck requests in Rein, however, does only take into account the number of keys in the opset but never the size of the corresponding values. The result is that the detected bottleneck opset may execute much faster than another opset from the same multiget query that is not detected as such.

3 TailX design and implementation

An overview of the architecture of TailX is given by Figure 3. When a request is issued, the coordinator node selects the best replica out of total target replicas based on the past read performance of replica servers. An appropriate replica selection mechanism (dynamic snitching [39]) is applied to select the best replica.

Afterwards, the request goes to a *splitter* where it is split into opsets by a partitioner (Murmur3 [34]). The number of operations and value sizes associated with keys varies in these opsets. Among these, some opsets that contain smaller operations with shorter execution time will finish earlier whereas opsets that contain operations with larger execution time finish later. To execute a multiget request with minimum latency, all the opsets should finish at the approximately same time. Therefore, to correctly estimate the total execution time of each opset, TailX identifies the operations that take more time. For this, it passes through *size estimation* module. The objective of this module is to estimate whether a given operation will access a *small* or a *large* value. It keeps track of keys that associated with large values and store the keys of those operations.

Once the value size of an operation is identified, *delay allowance estimation* module estimates the cost of each opset i.e. approximate total execution time and calculates the approximate delay allowance that occurred by each opset. This delay allowance is inserted as metadata in each operation of an opset. After delay allowance assignment, opsets go through the *delay queue*. The objective of this step is to procrastinate each opset which has delay allowance and let other requests execute at that time. If an operation has delay allowance then it inserted in a delay queue with given procrastinating time. The operations reside in the delay queue until the given procrastinate time expires.

Finally, operations go to the required server that is holding the data. Once the operations finish, they return the data to the coordinator. We present in the following sections the details of all proposed modules. First, we present the replica selection mechanism based on the load estimated among servers (§3.1). Next, we describe the request splitting based on the data storage (§3.2). Finally, we explain the delay allowance policies including delay estimation of operations and scheduling mechanism (§3.3).

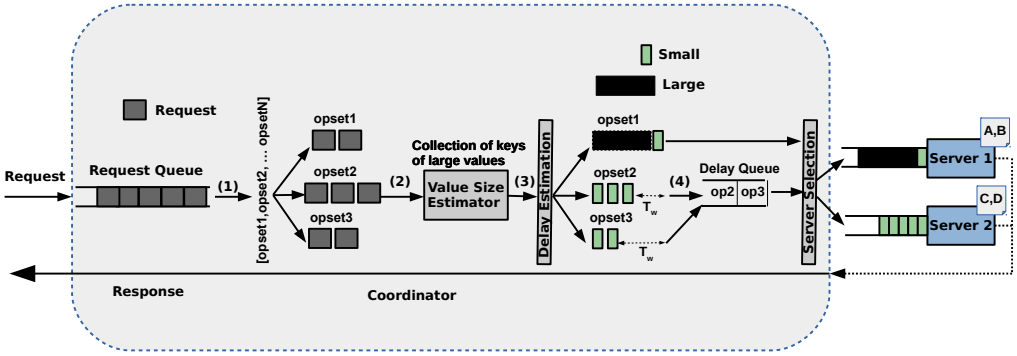


Fig. 4: Operating principle of TailX scheduling.

3.1 Load estimation and replica selection

The operations of a multiget request select the target replicas according to the hash-based mechanism followed by the replica server. The number of replicas depends on the replication factor followed by the storage systems. Afterwards, a replica selection algorithm (dynamic snitching [25] which considers past read performance of the replicas) is applied for scoring the replicas and a faster replica is chosen to complete the operation. The role of this component is to select the replica that is expected to serve a given request faster than other replicas.

3.2 Request splitting

In a key-value store, all storage nodes are divided into hash-based token ranges. After selecting the intended replica, request splits into opset according to the partitioner (e.g. Murmur3 [34]). Each opset goes to a different replica server and contains a varied number of operations with different value size. Our goal is to schedule the operations in a way that can complete each opset at the approximately same time. This gives better flexibility to other requests to execute at that time.

3.3 Delay allowance policies

The algorithms for delay allowance policies are described in Algorithm 1 and Algorithm 2. The role of these algorithms is to procrastinate the opsets which are finishing earlier than the other opsets.
Every opset has a different completion time due to the variations in value size and the number of operations in it. Therefore, some operations of an opset have to wait for bottleneck operations. This results in increasing the latency of the overall request.
To overcome this situation, the delay allowance module calculates the cost of each opset (*opcost*) i.e. opset execution time on the server. Calculation of the opset cost is based on the value size estimation since we need to know the number of operations for large values (N_L) in each opset. The operations of large values are the sole reason for inflating the operation cost. Therefore, we match the keys of large value to the keys stored in Bloom filter [6] (step 4 of Algorithm 1). Next, it calculates the opset cost of each opset based on the request service time for small value (T_S) and request service time for large value (T_L) (step

5 of Algorithm 1). Afterwards, it calculates delay allowance T_w ((step 8 of Algorithm 1)) and tags the allowance to each opset. Finally, it procrastinates operations that have delay allowance (step 11 of Algorithm 1) otherwise send the opset to the corresponding replica server. In Algorithm 2, if the delay allowance time has finished then the request is dequeued and sent to the corresponding replica server.

Delay allowance estimation The role of delay estimation is to estimate the approximate execution cost of each opset and calculate the approximate delay allowance which can be occurred at each opset. The calculation of delay allowance is based on the value size estimation of each operation.

Value size estimation. An important question that TailX addresses is to determine whether an operation will access a large or a small value. In this context, we set a threshold (say THR_L) where values above this threshold are considered as large by TailX. We assume that this choice is application dependent and that it is up to the database administrator to set the value of THR_L according to the data distribution over her database.

TailX uses Bloom filters to keep track of keys corresponding to large values. A Bloom filter [6] is a space-time efficient probabilistic data structure that allows performing to test whether a given item belongs to a predefined set. It is a vector of m bits initially set to 0, with an associated set of k hash functions (generally $k \ll m$). Inserting an element in the Bloom filter is done by hashing the element using the k hash functions and setting the corresponding bit positions to 1. Testing the presence of an element in the Bloom filter is done similarly by hashing the element using the k hash functions and testing whether all the corresponding bit positions are set to 1. Querying a Bloom filter may lead to false-positive but will never lead to false-negative.
After identifying keys that correspond to the large value, it calculates the opset cost i.e. how much time the opset will take to execute. To estimate the opset cost ($opcost$), it calculates the service time of operations for large values (T_L) and small values (T_S). Afterwards, it multiplies them by their respective number of operations to get the overall cost of the opset. Further, it calculates the delay allowance (T_w) for each opset. Delay allowance is calculated based on the cost difference of maximum opset cost ($opcost_{max}$) and cost of opset for which it is calculating the delay allowance. It means every opset has the allowance time in which it can wait and let other operations to complete.

Delay scheduling The role of a delay queue is to procrastinate the opset which has some delay allowance. This gives better flexibility for other queries to execute in the delay allowance time.

Delay queue design. Delay queue (Q_d) is an unbounded blocking queue implemented in Java for opsets which have delayed allowance. The idea of the delay queue is to procrastinate some operations. An element can be taken out once the delay has expired. The element which is at the head of the queue has the expired delay furthest in the past.

Scheduling of requests which has delay allowance. If the request is tagged by delay allowance ($T_w > 0$) during delay estimation then the request will be sent to delay queue. The scheduler adds the system current time in the delay allowance i.e. procrastinate time (T_d), which helps to correctly estimate the procrastinated opset.

Scheduling of requests with zero delay allowance. If the request is tagged by delay allowance ($T_w == 0$) during delay estimation then the request will be sent directly to the server without delay. Since these are the requests which take time to execute and don't offer any allowance for slacking that opset.
Finally, operations are sent to the intended server directly or after completion of the procrastination time.

Algorithm 1: Opset delay allowance algorithm

Data: $ksName$ = keyspace name, K = set of keys, CF = tablename, op = opset, $opcost_{max}$= max opset cost, req = multiget request, $opsets$ = set of opsets, N_L = set of keys correspond to large values in an opset, Q_d = delay queue, BF = bloom filter;
Input: req $(ksName, K, CF)$;
Output: Procrastinated opsets.

begin
  $opcost_{max} = 0$;
  for $op \in opsets$ **do**
    /* Calculate number of keys correspond to large values in an opset */
    $N_L := \{opr \in op \mid match(BF, opr.key) = 1\}$;
    /* Calculate opset cost */
    // T_L= request service time (in nanosec) for large value
    // T_S= request service time (in nanosec) for small value
    // opsize = number of keys in an opset
    $opcost = T_L * |N_L| + T_S * (opsize - |N_L|)$;
    /* Calculate max opset cost */
    $opcost_{max} = max(opcost, opcost_{max})$;
  for $op \in opsets$ **do**
    /* Calculate delay allowance */
    $T_w = opcost_{max} - op.opcost$;
    /* Tag T_w to each opset */
    $tag(T_w, op)$;
    /* Calculate procrastinating time */
    // $T_{current}$ = current system time
    $T_d \longleftarrow T_{current} + T_w$;
    if $op.T_w > 0$ **then**
      /* insert opset in delay queue */
      $Q_d.enqueue(op, T_d)$;
    else
      send op to corresponding replica;

Algorithm 2: Opset dequeue algorithm

begin
  while $Q_d \neq \emptyset$ && $T_{current} - T_d \geq 0$ **do**
    deque from Q_d;
    send op to corresponding replica;

4 Evaluation

We implement TailX as an extension of Cassandra [25], a very popular key-value store. We evaluate its effectiveness in reducing tail latency using synthetic dataset generated using the Yahoo! Cloud Serving Benchmark (YCSB) [10]. We compare different latency percentiles, particularly the tail, under TailX, against state-of-the-art algorithm i.e. Rein. We conduct extensive experiments on Grid'5000 [5], exploring the impact of varying ratios of multiget request sizes and their value sizes. Overall, our evaluation answers the following questions:

1. How does TailX performance effects by the multiget request sizes in the key-value stores? (§4.2)
2. How does TailX performance effects by the proportion of large values in the key-value stores? (§4.2)

We start this section by presenting our evaluation setup (§4.1) before presenting our results (§4.2).

4.1 Experimental setup

Experimental Platform. We evaluate TailX on Grid'5000 [5]. We use a 16 node cluster in which each machine is equipped with 2 Intel Xeon X5570 CPUs (4 cores per CPU), 24GB of RAM and a 465GB HDD. The machines are running the Debian 8 GNU/Linux operating system.

Configuration. We evaluate TailX in Cassandra. We used the industry-standard Yahoo! Cloud Serving Benchmark (YCSB) [10] to generate datasets and run our workloads. As YCSB only generates a single value size datasets for each given client, we modified its source code to allow generation of mixed size datasets. Specifically, for mixed size workloads, we kept the proportion of large values compared to small values the same. For generating client workloads, we configured YCSB on a separate node.
Moreover, in all the generated workloads, the access pattern of stored values (whether small or large) follows a Zipfian distribution (with a Zipfian parameter ρ=0.99). To have an idea of the size a given synthetic dataset, we insert 20 million of small records (1KB size) and 100K of large records (2 MB size). This approximately represents $\tilde{4}$1GB of data per node. We kept the replication factor as 3 which means each piece of value is available on 3 servers. Each measurement involves 1 million or 10 million requests and is repeated 5 times. Each multiget request access various operations with different value sizes. We test the cluster of its maximum attainable throughput and kept the 75% system load for all our experiments.

4.2 TailX on variable configurations of the synthetic dataset

We evaluate in this section the effectiveness of TailX along different dimensions of heterogeneous workloads, i.e., the impact of *long* operations on multiget requests and the impact of operations correspond to large values.

Impact of multiget requests containing large number of operations To study the impact of the proportion of long multiget requests (i.e. multiget request size is large) on the system performance, we fix the size of multiget request as 100 and short multiget request to 5. We keep the ratio of long multiget request to 20% i.e. for each 100 multiget requests, 80 multiget are of size 5 and 20 multiget are of size 100. Through this, we can see the impact of long multiget over short multiget requests. We present the improvement of TailX over Rein for 1 million operations and 10 million operations in Figure 5 and 6 respectively. Figure 7 shows the different latency percentiles to give a closer look in system. In this experiment,

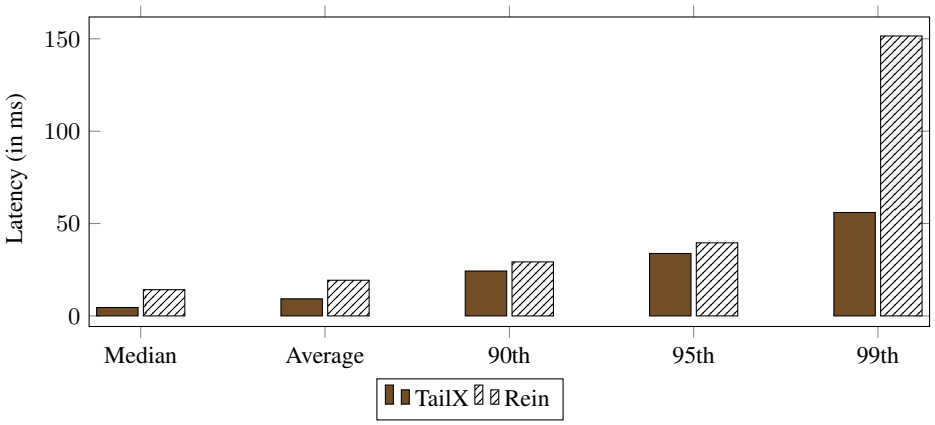


Fig. 5: Improvement of TailX over latency with different multiget request sizes (80% multiget of size 5 and 20% multiget of size 100) for 1 million operations

we start by generating datasets in which each multiget request contains 1KB values. Results show that TailX reduces the tail latencies over Rein by up to 63% while reducing the median latency by up to 71%. TailX achieves a better gain for median latency compare to tail latency. In terms of absolute latency (for 1 million operations), say for 99th percentile, it is 56 ms for TailX but roughly 152 ms for Rein respectively. For median latency, absolute value is 4.57 ms for TailX whereas it is around 14.3 ms for Rein.

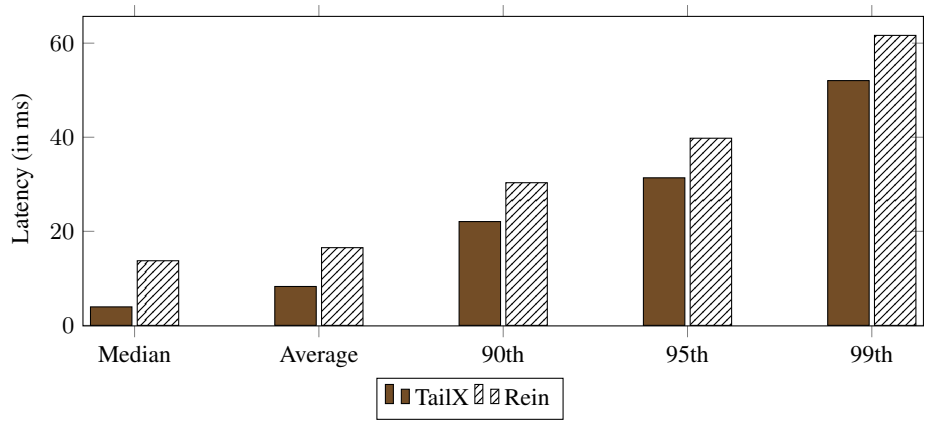


Fig. 6: Improvement of TailX over latency with different multiget request sizes (80% multiget of size 5 and 20% multiget of size 100) for 10 million operations

Impact of multiget requests having keys of large value sizes To study the impact of the proportion of large requests (request having large value i.e. 2 MB) on the system performance, we fix the size of multiget request as 20. We keep the percentage of large multiget requests as 20% and vary the proportion of large values.

Varying proportion of large value sizes. We vary the proportion of large value from 10% to 50% in a multiget request. As specified before, these variations are only for 20% of multiget requests. We present the latency reduction of TailX over Rein for 1 million operations. In the following, we zoom into the specific percentage of large value sizes.

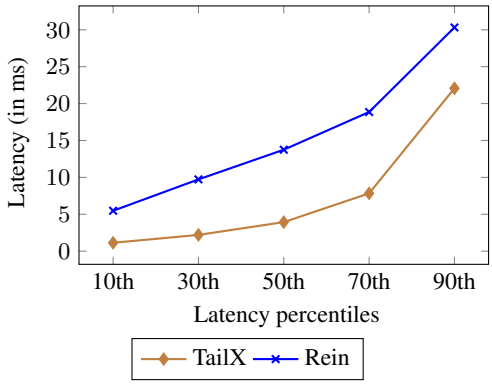


Fig. 7: Analysis of different latency percentiles for different multiget request sizes (80% multiget of size 5 and 20% multiget of size 100) for 10 million operations

Multiget of 10% large values. In this experiment, 20% of each multiget contains 10% of large values. Figure 8 and 9 show the improvement of TailX over Rein i.e., 30% latency reduction in 95th and 99th percentiles. TailX achieves a better gain for median latency compare to tail latency, i.e., roughly 75% v.s. 30%. In terms of absolute latency, say for 99th percentile, it is 97 ms for TailX but roughly 135 ms for Rein respectively. For median latency, absolute value is 11 ms for TailX whereas it is around 43 ms for Rein.

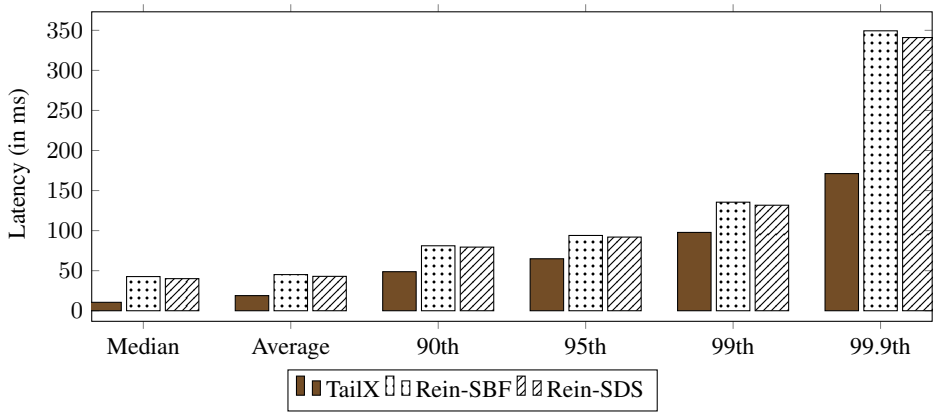


Fig. 8: Improvement of TailX over latency with different multiget request value sizes (80% multiget requests have small values (1 KB) and rest 20% multiget requests have 10% of large values) for 1 million operations

Multiget of 20% large values. Figure 10 and 12 show the improvement of TailX over Rein i.e., 40% and 45% latency reduction in 95th and 99th percentiles respectively. TailX achieves a better gain for median latency compare to tail latency, i.e., roughly 56% v.s. 45%. In terms of absolute latency, say for 99th percentile, it is 112 ms for TailX but roughly 203 ms for Rein respectively. For median latency, absolute value is 8 ms for TailX whereas it is around 18 ms for Rein.

Multiget of 50% large values. Figure 11 and 12 show the improvement of TailX over Rein i.e., 18% and 27% latency reduction in 95th and 99th percentiles respectively. TailX achieves a little less gain for median latency compare to tail latency, i.e., roughly 13%. In

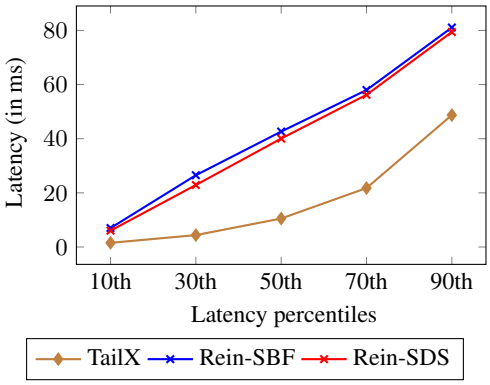


Fig. 9: Analysis of different latency percentiles for different multiget request value sizes (80% multiget requests have small values (1 KB) and rest 20% multiget requests have 10% of large values) for 1 million operations

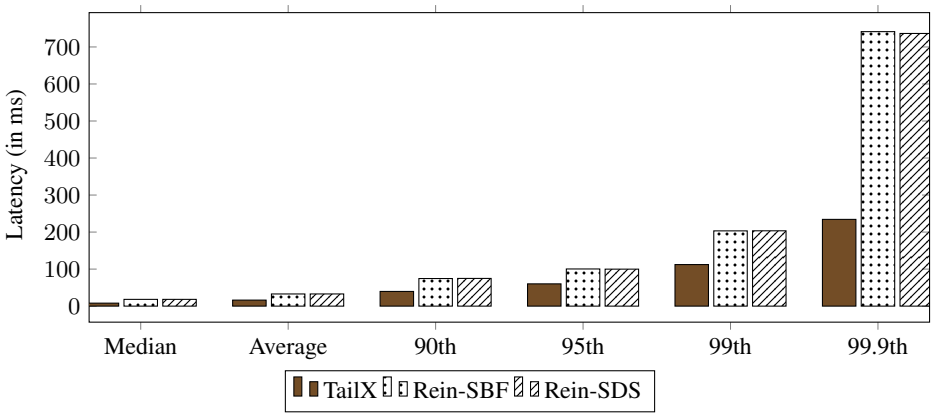


Fig. 10: Improvement of TailX over latency with different multiget request value sizes (80% multiget requests have small values (1 KB) and rest 20% multiget requests have 20% of large values) for 1 million operations

terms of absolute latency, say for 99th percentile, it is 109 ms for TailX but roughly 150 ms for Rein respectively. For median latency, absolute value is 5.9 ms for TailX whereas it is around 6.76 ms for Rein.

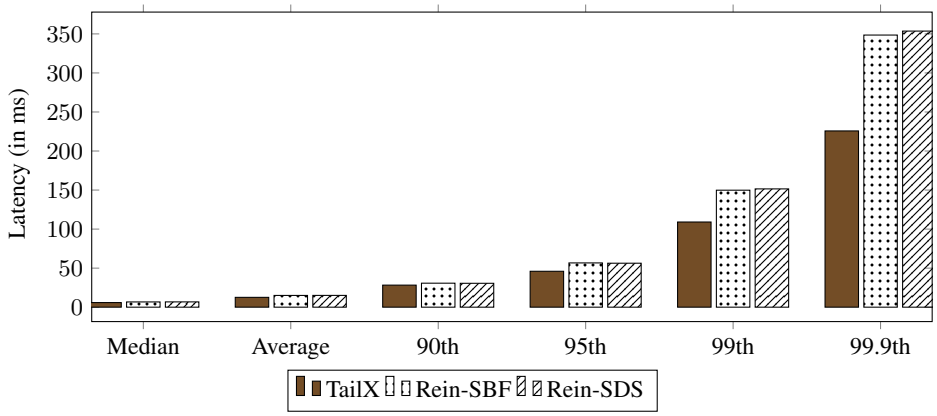
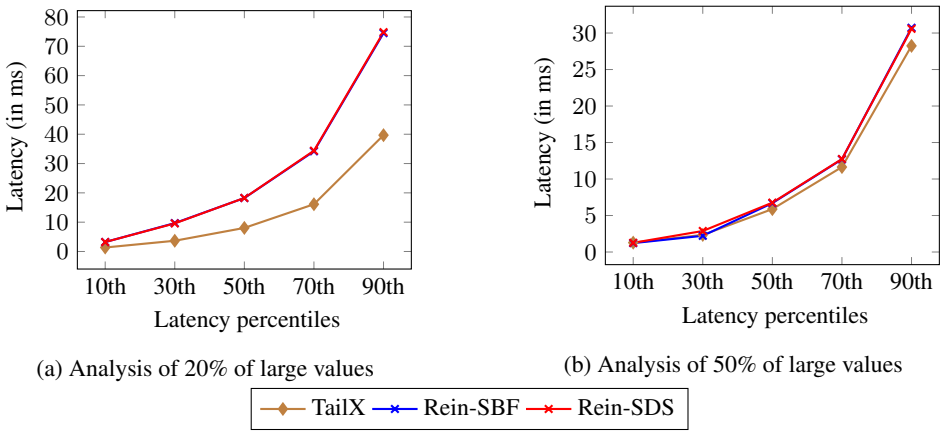


Fig. 11: Improvement of TailX over latency with different multiget request value sizes (80% multiget requests have small values (1 KB) and rest 20% multiget requests have 50% of large values) for 1 million operations



(a) Analysis of 20% of large values

(b) Analysis of 50% of large values

Fig. 12: Analysis of different latency percentiles for different multiget request value sizes (80% multiget requests have small values (1 KB) and rest 20% multiget requests have a) 20% of large values b) 50% of large values) for 1 million operations.

Summarizing, TailX outperforms Rein in most of the configurations. TailX is effective when there are some long requests in the systems. Also, the effectiveness of TailX can be seen when some multiget requests have some percentage of large values. We note that TailX is designed to handle heterogeneous workloads that have high variance across requests sizes w.r.t number of operations and their value sizes. When the proportion of value sizes of requests increases, the impact of TailX is visible more. TailX improves the tail latency till 20% whereas while workload having 50% of large value, the improvement decreases compare to the 20%. Since TailX filter the requests with large values with bloom filter and therefore if there are bulk of requests which have large value it increases the overhead.

Therefore, less impact is seen in this case. Overall in these configurations, TailX reduces the median latency up to 75% and tail latency by up to 70%.

5 Related Work

Several works addressed the problem of tail latency in distributed storage systems. Some have addressed the impact of incoming workloads that are coming on the system. We present our related work as follows:

Web workloads. Atikoglu et al. [4] described the workload analysis of a Memcached [32] traffic at Facebook. It studies 284 billion requests over 58 days for five different Memcached use cases. It presents the CDFs of value size in different Memcached pools. ETC pool is the largest and most heterogeneous value size pool where value sizes vary from few bytes to MBs.

Network-specific. Orchestra [8] uses weighted fair sharing where each transfer is assigned a weight and each link in the network is shared proportionally to the weight of the transfer. Baraat [15] is a decentralized task-aware scheduling system that dynamically changes the level of multiplexing in the network to avoid head-of-line-blocking. It uses task arrival time to assign a globally unique identifier and put a priority for each task. All flows of a task use this priority irrespective of the network they traverse. Varys [9] is another coflow scheduling system that decreases communication time for data-intensive jobs and provides predictable communication time. It assumes complete prior knowledge of coflow characteristics such as the number of flows, their sizes, etc. Aalo [7] is another scheduling policy that improves performance in data-parallel clusters without prior knowledge. To improve the performance in datacenters, pFabric [2] decouples flow scheduling from rate control mechanisms.

Redundancy-specific. Redundancy is a powerful technique in which clients initiate an operation multiple times on multiple servers. The operation which completes first is considered and the rest of them is discarded. Vulimiri et al. [38] characterize the scenarios where redundancy improves latency even under exceptional conditions. It introduces a queuing model that gives an analysis of system utilization and server service time distribution. Sparrow [33], a stateless distributed scheduler that adapts the power of two choices technique [29] by selecting two random servers. It put the tasks on the server which has fewer queued tasks. Sparrow [33] uses batch sampling where instead of sampling each task it places m tasks of a job on least loaded randomly selected servers. This approach performs better for parallel jobs since they are sensitive to tail task wait time.

Task-aware schedulers. Hawk [13] and Eagle [12] are two systems proposing a hybrid scheduler that schedules jobs according to their sizes. In Hawk [13], long jobs are scheduled using a centralized scheduler while small jobs are scheduled in a fully distributed way. Omega [36] is a shared-state scheduler in which a separate centralized resource manager maintains a shared scheduling state.

Request reissues and parallelism. Kwiken [21] 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. [19] propose solutions for decreasing tail latencies by dynamically increasing the parallelism of individual requests in interactive services. *Few-to-Many* (FM) selectively parallelizes the long running requests since that are the ones contributing the most to the tail latency. Recent efforts [22,23] 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.

Jeon et al. [23] focus on the parallelizing long running queries which are few compared to the short ones. It aims to achieve consistent low response time for web search engines.

Multiget scheduling. In key-value stores, multiget scheduling is a common pattern for scheduling requests efficiently. Systems like Cassandra [25], MongoDB [30] offer such algorithms in these systems. Rein [35] uses a multiget scheduling algorithm to schedule the multiget request in a fashion that can reduce the median as well as tail latency.

Tail latency specific. Minos [14] is another in-memory key-value store that uses size aware sharding to send small and large requests to different cores. It ensures that small requests never wait due to the large request which improves tail latencies. Metis [26] is an auto-tuning service to improve tail latency by using customized Bayesian optimization. SyncGC [18] tries to reduce the tail latency in Cassandra by proposing a synchronized garbage collection technique that schedules multiple GC instances to sync with each other. Sphinx [16] uses a thread-per-core approach to reduce tail latency in a key-value store by using application-level partitioning and inter-thread messaging. Some authors [1] provide bounds on tail latency for distributed storage systems by using erasure coding. It helps to provide optimization to minimize weighted tail latency probabilities.

6 Conclusion

In this paper, we addressed the problem of tail latencies in key-value stores under heterogeneous workloads for multiget requests. For multiget scheduling, an in-depth study of state-of-the-art has highlighted the fact that it doesn't perform well under heterogeneous workloads. We proposed TailX, a task-aware multiget scheduling algorithm that effectively deals with heterogeneous multiget requests. It identifies the bottleneck operations apriori and procrastinates them to avoid head-of-line-blocking. The result is an improved overall performance of the key-value store for a wide variety of heterogeneous workloads. Specifically, our experiments under heterogeneous YCSB workloads in a Cassandra based implementation shows that TailX outperforms state-of-the-art algorithm and reduces the tail latencies by up to 70% while reducing the median latency by up to 75%.

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 was partially supported by the *European Union's Horizon 2020 research and innovation programme* under grant agreement No 692178 (EBSIS project), by CHIST-ERA under project DIONASYS, and by the Swiss National Science Foundation (SNSF) under grant 155249.

References

1. Al-Abbasi, A.O., Aggarwal, V., Lan, T.: Ttloc: Taming tail latency for erasure-coded cloud storage systems. IEEE Transactions on Network and Service Management (2019)
2. Alizadeh, M., Yang, S., Sharif, M., Katti, S., McKeown, N., Prabhakar, B., Shenker, S.: pfabric: Minimal near-optimal datacenter transport. In: SIGCOMM (2013)
3. Ananthanarayanan, G., Ghodsi, A., Warfield, A., Borthakur, D., Kandula, S., Shenker, S., Stoica, I.: Pacman: Coordinated memory caching for parallel jobs. In: NSDI (2012)
4. Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., Paleczny, M.: Workload analysis of a large-scale key-value store. In: SIGMETRICS (2012)
5. Balouek, D., Carpen Amarie, A., Charrier, G., Desprez, F., Jeannot, E., Jeanvoine, E., Lèbre, A., Margery, D., Niclausse, N., Nussbaum, L., Richard, O., Pérez, C., Quesnel, F., Rohr, C., Sarzyniec, L.: Adding virtualization capabilities to the Grid'5000 testbed (2013)

6. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Communications of the ACM (1970)
7. Chowdhury, M., Stoica, I.: Efficient coflow scheduling without prior knowledge. In: SIGCOMM (2015)
8. Chowdhury, M., Zaharia, M., Ma, J., Jordan, M.I., Stoica, I.: Managing data transfers in computer clusters with orchestra. In: SIGCOMM (2011)
9. Chowdhury, M., Zhong, Y., Stoica, I.: Efficient coflow scheduling with varys. In: SIGCOMM (2014)
10. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: SoCC (2010)
11. Dean, J., Barroso, L.A.: The tail at scale. Communications of the ACM (2013)
12. Delgado, P., Didona, D., Dinu, F., Zwaenepoel, W.: Job-aware scheduling in Eagle: Divide and stick to your probes. In: SoCC (2016)
13. Delgado, P., Dinu, F., Kermarrec, A.M., Zwaenepoel, W.: Hawk: Hybrid datacenter scheduling. In: USENIX ATC (2015)
14. Didona, D., Zwaenepoel, W.: Size-aware sharding for improving tail latencies in in-memory key-value stores. In: NSDI (2019)
15. Dogar, F.R., Karagiannis, T., Ballani, H., Rowstron, A.: Decentralized task-aware scheduling for data center networks. In: SIGCOMM (2014)
16. Enberg, P., Rao, A., Tarkoma, S.: The impact of thread-per-core architecture on application tail latency. In: ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS) (2019)
17. Fan, B., Andersen, D.G., Kaminsky, M.: Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In: NSDI (2013)
18. Han, S., Lee, S., Hahn, S.S., Kim, J.: Syncgc: A synchronized garbage collection technique for reducing tail latency in cassandra. In: Proceedings of the 9th Asia-Pacific Workshop on Systems. APSys (2018)
19. Haque, M.E., Eom, Y.h., He, Y., Elnikety, S., Bianchini, R., McKinley, K.S.: Few-to-many: Incremental parallelism for reducing tail latency in interactive services. In: ASPLOS (2015)
20. Jaiman, V., Mokhtar, S.B., Quéma, V., Chen, L.Y., Rivière, E.: Héron: Taming tail latencies in key-value stores under heterogeneous workloads. In: SRDS (2018)
21. Jalaparti, V., Bodik, P., Kandula, S., Menache, I., Rybalkin, M., Yan, C.: Speeding up distributed request-response workflows. In: SIGCOMM (2013)
22. Jeon, M., He, Y., Kim, H., Elnikety, S., Rixner, S., Cox, A.L.: TPC: Target-driven parallelism combining prediction and correction to reduce tail latency in interactive services. In: ASPLOS (2016)
23. Jeon, M., Kim, S., Hwang, S.w., He, Y., Elnikety, S., Cox, A.L., Rixner, S.: Predictive parallelization: Taming tail latencies in web search. In: SIGIR (2014)
24. Jiang, W., Xie, H., Zhou, X., Fang, L., Wang, J.: Understanding and improvement of the selection of replica servers in key–value stores. Information Systems (2019)
25. Lakshman, A., Malik, P.: Cassandra: A decentralized structured storage system. SIGOPS Oper. Syst. Rev. (2010)
26. Li, Z.L., Liang, C.J.M., He, W., Zhu, L., Dai, W., Jiang, J., Sun, G.: Metis: Robustly tuning tail latencies of cloud systems. In: USENIX ATC (2018)
27. Lim, H., Han, D., Andersen, D.G., Kaminsky, M.: Mica: A holistic approach to fast in-memory key-value storage. In: NSDI (2014)
28. Misra, P.A., Borge, M.F., Goiri, I.n., Lebeck, A.R., Zwaenepoel, W., Bianchini, R.: Managing tail latency in datacenter-scale file systems under production constraints. In: EuroSys (2019)
29. Mitzenmacher, M.: The power of two choices in randomized load balancing. IEEE Transactions on Parallel and Distributed Systems (2001)
30. MongoDB: https://www.mongodb.com/
31. Motwani, R., Phillips, S., Torng, E.: Non-clairvoyant scheduling. In: Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms. SODA (1993)
32. Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H.C., McElroy, R., Paleczny, M., Peek, D., Saab, P., Stafford, D., Tung, T., Venkataramani, V.: Scaling Memcache at Facebook. In: NSDI (2013)
33. Ousterhout, K., Wendell, P., Zaharia, M., Stoica, I.: Sparrow: Distributed, low latency scheduling. In: SOSP (2013)

34. Partitioners: https://docs.datastax.com/en/cassandra/3.0/cassandra/architecture/archPartitionerAbout.html
35. Reda, W., Canini, M., Suresh, L., Kostić, D., Braithwaite, S.: Rein: Taming tail latency in key-value stores via multiget scheduling. In: EuroSys (2017)
36. Schwarzkopf, M., Konwinski, A., Abd-El-Malek, M., Wilkes, J.: Omega: Flexible, scalable schedulers for large compute clusters. In: EuroSys (2013)
37. Suresh, L., Canini, M., Schmid, S., Feldmann, A.: C3: Cutting tail latency in cloud data stores via adaptive replica selection. In: NSDI (2015)
38. Vulimiri, A., Godfrey, P.B., Mittal, R., Sherry, J., Ratnasamy, S., Shenker, S.: Low latency via redundancy. In: CoNEXT (2013)
39. Williams, B.: Dynamic snitching in Cassandra: past, present, and future. http://www.datastax.com/dev/blog/dynamic-snitching-in-cassandra-past-present-and-future (2012)