Taming Tail Latency in Key-Value Stores: a Scheduling Perspective

Sonia Ben Mokhtar[1], Louis-Claude Canon[2], Anthony Dugois[3], Loris Marchal[3], and Etienne Rivière[4]

[1] LIRIS, Lyon, France
sonia.benmokhtar@insa-lyon.fr
[2] FEMTO-ST Institute, Besançon, France
louis-claude.canon@femto-st.fr
[3] LIP, Lyon, France
{anthony.dugois,loris.marchal}@ens-lyon.fr
[4] ICTEAM, UCLouvain, Louvain-la-Neuve, Belgium
etienne.riviere@uclouvain.be

Abstract. Distributed key-value stores employ replication for high availability. Yet, they do not always efficiently take advantage of the availability of multiple replicas for each value, and read operations often exhibit high tail latencies. Various replica selection strategies have been proposed to address this problem, together with local request scheduling policies. It is difficult, however, to determine what is the absolute performance gain each of these strategies can achieve. We present a formal framework allowing the systematic study of request scheduling strategies in key-value stores. We contribute a definition of the optimization problem related to reducing tail latency in a replicated key-value store as a minimization problem with respect to the maximum weighted flow criterion. By using scheduling theory, we show the difficulty of this problem, and therefore the need to develop performance guarantees. We also study the behavior of heuristic methods using simulations, which highlight which properties are useful for limiting tail latency: for instance, the EFT strategy—which uses the earliest available time of servers—exhibits a tail latency that is less than half that of state-of-the-art strategies, often matching the lower bound. Our study also emphasizes the importance of metrics such as the stretch to properly evaluate replica selection and local execution policies.

Keywords: Online Scheduling· Key-Value Store· Replica Selection· Tail Latency· Lower Bound

1 Introduction

Online services are used by a large number of users accessing ever-increasing amounts of data. One major constraint is the high expectation of these users in terms of service responsiveness. Studies have shown that an increase in the average latency has direct effects on the use frequency of an online service, e.g.,

experiments at Google have shown that an additional latency of 400 ms per request for 6 weeks reduced the number of daily searches by 0.6 % [6].

In modern cloud applications, data storage systems are important actors in the evolution of overall user-perceived latency. Considerable attention has been given, therefore, to the performance predictability of such systems. Serving a single user request usually requires fetching multiple data items from the storage system. The overall latency is often that of the slowest request. As a result, a very small fraction of slow requests may result in overall service latency degradation for many users. This problem is known as the *tail latency problem*. In large-scale deployments of cloud applications, it has been observed that the 95^{th} and 99^{th} percentiles in the query distribution show latency values that can be several orders of magnitude higher than the median [1, 8].

In this study, we focus on the popular class of storage systems that are key-value stores, where each value is simply bound to a specific key [9, 16]. These systems scale horizontally by distributing responsibility for fractions of the key space across a large number of storage servers. They ensure disjoint-access parallelism, high availability and durability by relying on data *replication* over several servers. As such, read requests may be served by any of these replica.

Replica selection strategies [13, 15, 23] dynamically schedule requests to different replicas in order to reduce tail latency. When the request reaches the selected replica, it is inserted into a queue and a local queue scheduling strategy may decide to prioritize certain requests over others. These combinations of global and local strategies are well adapted to the distributed nature of key-value stores, as they assume no omniscient or real-time knowledge of the status of each replica, or of concurrently-scheduled requests. It remains difficult, however, to systematically assess their potential. On the one hand, there is no clear upper bound on the performance that a global, omniscient strategy could theoretically achieve. On the other hand, it is difficult to determine what is the impact of using only local or partial information on achievable performance. Our goal in this paper is to bridge this gap, and equip designers of replica selection and local scheduling strategies with tools enabling their formal evaluation. By modeling a corresponding scheduling problem, we develop a number of guarantees that apply to a variety of designs.

Outline. We make the following contributions:

- a formal model to describe replicated key-value stores and the scheduling problem associated to the minimization of tail latency (Section 3);
- a polynomial-time offline algorithm, a $(2-\frac{1}{m})$-approximation guarantee and a NP-completeness result for related scheduling problems (Section 4);
- online heuristics to solve the online optimization of maximum weighted flow, representing compromises in locally available information at the different servers of the key-value store (Section 5);
- the comparison of these heuristics in extensive simulations (Section 6).

The algorithms, the related code, data and analysis are available online[5].

[5] https://doi.org/10.6084/m9.figshare.14755359

2 Related Work

We provide here a short review of related studies. An extended survey can be found in our companion research report [3].

Key-value stores. Key-value stores implement data partitioning for horizontal scalability, typically using consistent hashing. This consists of treating the output of a hash function as a ring; each server is then assigned a position on this circular space and becomes responsible of all data between it and its predecessor (the position of a data item is decided by hashing the corresponding key) [9,16]. Replication is implemented on top of data partitioning, by duplicating each data item on the successors of its assigned server.

Replica selection strategies [13,15,23] generally target the reduction of tail latency. They seek to avoid that a request be sent to a busy server when a more available one would have answered faster. The server receiving a request (the *coordinator*) is generally not the one in charge of the corresponding key. All servers know, however, the partitioning and replication plans. Coordinators can, therefore, associate a key with a list of replicas and select the most appropriate server to query. Cassandra uses Dynamic Snitching [16], which selects the replica with the lowest average load over a time window. This strategy is prone to instabilities, as lowly-loaded servers tend to receive swarm of requests. C3 [23] uses an adaptive replica selection strategy that can quickly react to heterogeneous service times and mediate this instability. Dynamic snitching and C3 both assume that values are served with the same latency. Héron [13] addresses the problem of head-of-line blocking arising when values have heterogeneous sizes: requests for small values may be scheduled behind requests for large values, increasing tail latency. It propagates across the cluster the identity of values whose size is over a threshold, together with load information and pending requests to such large values. Size-aware sharding [10] avoids head-of-line blocking on a specific server, by specializing some of its cores to serve only large values. Other systems, such as REIN [22] or TailX [14], focus on the specific case of multi-get operations, whereby multiple keys are read in a single operation. We intend to consider multi-get queries in our future work, as an extension of our formal models.

All the solutions mentioned above empirically improve tail latency under the considered test workloads. There is, however, no strong evidence that no better solution exists as the proposed heuristics are not compared to any formal ground. In contrast, and similarly to our objective, Li *et al.* [21] propose a *single-node* model of a complete hardware, application and operating system stack using queuing theory. This allows determining expected tail latencies in the modeled system. The comparison of the model and an actual hardware and software stack shows important discrepancies. The authors were able to identify performance-impacting factors (e.g. request re-ordering, limited concurrency, non-uniform memory accesses, etc.) and address them, matching close to optimal performance under the knowledge of predictions from the model. Our goal is to be an enabler for such informed optimization and development for the case of distributed (*multi-node*) storage services.

Flow minimization in scheduling. Minimization of latency—the time a request spends in the system—is commonly approached as the optimization of flow time in theoretical works, and a great diversity of scheduling problems deal with this criterion. The functions that usually constitute the objective to minimize are the max-flow ($F_{\max}$) and the sum-flow ($\sum F_i$). It is well-known that the max-flow criterion is minimized by the FIFO (First In First Out) strategy on a single-machine, and it has also been proven $(3 - \frac{2}{m})$-competitive on m machines [4].

Sometimes the maximum weighted flow $\max w_i F_i$ is considered in order to give more importance to some requests. For example Bender *et al.* introduced the stretch, where the weight is the inverse of the request serving time ($w_i = 1/p_i$), to express and study the notion of fairness for scheduling HTTP requests in web servers [4]. Bender *et al.* derive an $O(\sqrt{\Delta})$-competitive algorithm from the EDF (Earliest Deadline First) strategy, with Δ being the ratio between the largest processing time to the smallest one ($\Delta = \frac{\max p_i}{\min p_i}$). Later, Legrand *et al.* presented a polynomial-time algorithm to solve the offline minimization of max weighted flow time on unrelated machines when preemption is allowed [18].

Optimizing the average performance is obtained through minimizing the sum flow time criterion. However, this optimization objective may lead to starvation: some requests may be infinitely delayed in an optimal solution [4].

Replication in scheduling. An important consequence of replication is that a given request cannot be executed by any server; it must be processed by a server in the subset of replicas able to handle it. This constraint is commonly known as "multipurpose machines". Brucker *et al.* proposed a formalization and analyzed the complexity of some of these problems [5]. They show for example that minimizing the sum flow on identical multipurpose machines can be solved in polynomial time. To the best of our knowledge, there exists no work considering replication for the minimization of the maximum (weighted) flow.

3 Formal Model

We propose a formal model of a distributed and replicated key-value store. This section describes the theoretical framework and states the optimization problem related to the minimization of tail latency.

Application and platform models. We start by defining a key-value map (K, V) as a set of associations between keys and values. We associate c keys $K = \{K_1, \ldots, K_c\}$ to c values $V = \{V_1, \ldots, V_c\}$: each unique key K_l refers to a unique value V_l whose size is $z_l > 0$.

The considered problem is to schedule a set of n requests $T = \{T_1, \ldots, T_n\}$ on m parallel servers $M = \{M_1, \ldots, M_m\}$ in a replicated key-value store. The set K is spread over these servers. For one server M_j, the function Ψ gives the subset of keys $\Psi(M_j) \subseteq K$ that is owned by M_j. Each request T_i carries a key that will be used to retrieve a specific value in the store. For one request T_i, the function φ gives this key $\varphi(T_i) = K_l$. The same key can be carried by different

requests. Fig. 1 shows the relationship between requests, keys and values. A server M_j may execute a request T_i if and only if $\varphi(T_i) \in \Psi(M_j)$, i.e., M_j holds the value for the carried key of T_i.

All requests are independent: no request has to wait for the completion of another request, and no communication occurs between requests. We limit ourselves to the non-preemptive problem, as real implementations of key-value stores generally do not interrupt requests.

In addition, each request T_i has a processing time $p_i = \alpha z_l + \beta$, where $\alpha, \beta > 0$ (with $\varphi(T_i) = K_l$). Processing time is equal to the average network latency β plus data sending time, which is proportional to the size of the value this request is looking for (factor α represents the inverse of the bandwidth). A request is also unavailable before time $r_i \geq 0$ and its properties are unknown as well.

As a server M_j may execute a request T_i only if it holds the key $\varphi(T_i)$, we treat the multipurpose machines scheduling problem where the set $\mathcal{M}_i \subseteq M$ represents the set of machines able to execute the request T_i, i.e., $\mathcal{M}_i = \{M_j \mid \varphi(T_i) \in \Psi(M_j)\}$. In the Graham $\alpha|\beta|\gamma$ notation of scheduling problems, this constraint is commonly denoted by $\mathcal{M}_i$ in the β-part. This aspect of the problem models data replication on the cluster. Key-value stores tend to express the replication factor, i.e., the number of times the same data is duplicated, as a parameter k of the system. Therefore, we have $|\mathcal{M}_i| = k$.

Problem statement. There is no objective criterion that can straightforwardly represent the formal optimization of tail latency, as there is no formal definition of this system concept. Different works consider the 95^{th} percentile, the 99^{th} percentile, or the maximum, and it should be highlighted that we do not want to degrade average performance too much. We propose to approach the tail latency optimization by minimizing a well-known criterion in online scheduling theory: the maximum time spent by requests in the system, also known as the maximum flow time $\max F_i$, where $F_i = C_i - r_i$ expresses the difference between the completion time C_i and the release time r_i of a request T_i.

However, it seems unfair to wait longer for a request for a small value to complete than for a large one: for example, we know that a user's tolerance for

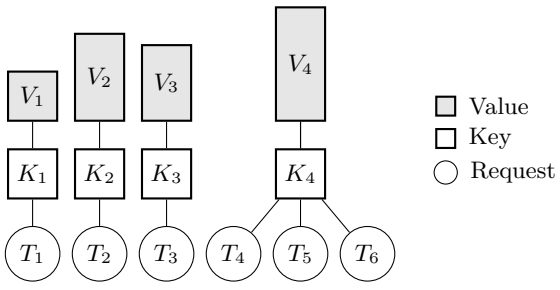


Fig. 1. A bipartite graph showing relations between requests, keys and values. Different requests may hold the same key (e.g. K_4).

the response time of a real system is greater when a process considered to be heavy is in progress. Hence, the latency should be weighted to emphasize the relative importance of a given request; we are looking for a "fairness" property. To formalize this idea, we associate a weight w_i to each T_i. The definition of this weight is flexible, in order to allow the key-value store system designer to consider different kinds of metrics. We focus on two weighting strategies in our simulations. First, the flow time ($w_i = 1$) gives an importance to each request that is proportional to its cost, which favors requests for large values. Second, the stretch ($w_i = \frac{1}{p_i}$) gives the same importance to each request, but this favors requests for small values because they are more sensitive to scheduling decisions.

In summary, our optimization problem consists in finding a schedule minimizing the maximum weighted flow time $\max w_i F_i$ under the following constraints:

- P: there are m parallel identical servers.
- $\mathcal{M}_i$: each request T_i is executable by a subset of servers.
- online-r_i: each request T_i has a release time $r_i \geq 0$ and request characteristics (r_i, p_i and w_i) are not known before r_i.

A solution to this problem $(P|\mathcal{M}_i, \text{online-}r_i|\max w_i F_i)$ is to find a schedule Π, which, for a request T_i, gives its executing server and starting time. Then we define a pair $\Pi(T_i) = (M_j, \sigma_i)$, where the server M_j executes T_i at time $\sigma_i \geq r_i$. Server M_j must hold the required value ($M_j \in \mathcal{M}_i$), and there are no simultaneous executions: two different requests cannot be executed at the same time on the same server.

As mentioned earlier, we are also interested in minimizing the average latency ($\sum w_i F_i$) as a secondary objective; even if the main goal is to reduce tail latency, it would not be reasonable to degrade average performance doing so. This bi-objective problem could be approached by the optimization of the more general ℓ_p-norm function of flow times [2], but it is left for future work.

4 Maximum Weighted Flow Optimization

In order to evaluate the performance of replica selection heuristics, it would be interesting to derive optimal or guaranteed algorithms for the offline version of our problem, namely the minimization of the maximum weighted flow time of requests, or even for restricted variants. We show here that we can derive optimal or approximation algorithms when tasks are all released at time 0, but as soon as we introduce release dates, the problem gets harder to tackle. Nevertheless, a lower bound can be computed.

Without release times. We first focus on the non-preemptive problem of minimizing the maximum weighted flow time on a single server when there is no release times, i.e., all requests are available at time 0. Remark that in this case, the more common completion times equivalently replace flow times (i.e., $C_i = F_i$). We consider a simple algorithm named Single-Simple which schedules requests by non-increasing order of weights w_i, to solve this scheduling problem.

Theorem 1. SINGLE-SIMPLE *solves* $1||\max w_iC_i$ *in polynomial time.*

Proof. Let OPT be an optimal schedule. If all requests are ordered by non-increasing weight, then SINGLE-SIMPLE is optimal. If they are not, we can find two consecutive requests T_j and T_k in OPT such that $w_j \leq w_k$. Then, their contribution to the objective is $\mathcal{C} = \max(w_jC_j, w_k(C_j + p_k)) = w_k(C_j + p_k)$ because $w_jC_j \leq w_j(C_j + p_k) \leq w_k(C_j + p_k)$. If we swap the requests, then the contribution becomes $\mathcal{C}' = \max(w_kC'_k, w_j(C'_k + p_j))$ where C'_k is the completion time of request T_k in this new schedule. By construction, $C_j + p_k = C'_k + p_j$. We have $w_kC'_k \leq w_k(C'_k + p_j)$, $w_j(C'_k + p_j) \leq w_k(C'_k + p_j)$ and $w_k(C'_k + p_j) = w_k(C_j + p_k) = \mathcal{C}$. Therefore, $\max(w_kC'_k, w_j(C'_k + p_j)) = \mathcal{C}' \leq \mathcal{C}$.

It follows that if two consecutive requests are not ordered by non-increasing weight in OPT, we can switch them without increasing the objective. By repeating the operation request by request, we transform OPT in another optimal schedule where requests are sorted by non-increasing w_i. Hence, SINGLE-SIMPLE is optimal. □

SINGLE-SIMPLE does not extend to m parallel machines in the general case. However, it solves the case where all requests have homogeneous size p. The proof of this result (available in the companion report [3]) follows a similar argument as the previous proof.

Theorem 2. SINGLE-SIMPLE *solves* $P|p_i = p|\max w_iC_i$ *in polynomial time.*

For m machines when processing times are not identical, the problem is trivially NP-hard even with unit weights because $P||C_{\max}$ is NP-hard [19]. We prove that SINGLE-SIMPLE is an approximation algorithm.

Theorem 3. SINGLE-SIMPLE *computes a* $(2 - \frac{1}{m})$*-approximation for the problem* $P||\max w_iC_i$*, and this ratio is tight.*

Proof. Let us consider a schedule S built by SINGLE-SIMPLE and an optimal schedule OPT. We denote by T_j the request for which $w_jC_j = \max w_iC_i^S$, i.e., the request that reaches the objective in S. Then we remove from S and OPT all T_i such that $w_i < w_j$ (it does not change the objective $\max w_iC_i^S$ in S and can only decrease $\max w_iC_i^{OPT}$ in OPT). Let $C^*_{\max}$ denote the optimal makespan when scheduling only the remaining requests. As S is a list-scheduling (in the sense of Graham), we have $C^S_{\max} \leq \left(2 - \frac{1}{m}\right) \cdot C^*_{\max}$ [12], where $C^S_{\max}$ is the completion time of the last request in S (i.e., $C_j = C^S_{\max}$). Let T_k be the last completed request in OPT, such that $C_k = C^{OPT}_{\max}$. This makespan is bounded by the optimal one (i.e., $C^*_{\max} \leq C_k^{OPT}$). Therefore,

$$\begin{aligned}\max w_iC_i^S = w_jC_j^S = w_jC^S_{\max} &\leq \left(2 - \frac{1}{m}\right) \cdot w_jC^*_{\max} \leq \left(2 - \frac{1}{m}\right) \cdot w_jC_k^{OPT} \\ &\leq \left(2 - \frac{1}{m}\right) \cdot \frac{w_j}{w_k} \cdot w_kC_k^{OPT} \leq \left(2 - \frac{1}{m}\right) \cdot \frac{w_j}{w_k} \cdot \max w_iC_i^{OPT}.\end{aligned}$$

As we removed all requests weighted by a smaller value than w_j, we have $\frac{w_j}{w_k} \leq 1$ and it follows that $\max w_i C_i^S \leq \left(2 - \frac{1}{m}\right) \cdot \max w_i C_i^{OPT}$. We now prove that this bound is asymptotically tight, by considering the instance with m machines and $n = m(m-1)+1$ requests $T_{1 \leq i \leq n}$ with the following weights and processing times:

- $w_i = W + 1$, $p_i = 1$ for all $1 \leq i < n$;
- $w_n = W$, $p_n = m$.

Request T_n will be scheduled last in S, which gives an objective of $\max w_i C_i^S = (2m-1)W$, whereas an optimal schedule starts this request at time 0 and has an objective of $\max w_i C_i^{OPT} = m(W+1)$. On this instance, the approximation ratio $\left(2 - \frac{1}{m}\right) \cdot \frac{W}{W+1}$ tends to $2 - \frac{1}{m}$ as $W \to \infty$. □

Offline problem with release times. Legrand *et al.* solved the scheduling problem $R|r_i, pmtn| \max w_i F_i$ in polynomial time using a linear formulation of the model [18]. This offline problem is very similar to the one we are interested in, as the platform relies on unrelated machines, which generalizes our parallel multipurpose machines environment ($P|\mathcal{M}_i| \max w_i F_i$ is a special case of $R|| \max w_i F_i$ [20]). In fact, it only differs on one specific aspect: it allows preempting and migrating jobs, which we do not permit in our model.

We establish below the complexity of the problem $P|r_i, pmtn^*| \max w_i F_i$, where non-migratory[6] preemption is allowed. Interestingly, preventing migration makes the problem NP-complete. The proof of this result (available in the report [3]) consists in a reduction from the NP-complete problem $P||C_{\max}$ [19].

Definition 1 (NonMigratory-Dec(T, M, B)). *Given a set of requests T, a set of machines M and a bound B, if we define the deadline $d_i = r_i + \frac{B}{w_i}$ for all T_i, is it possible to build a non-migratory preemptive schedule where each request meets its deadline?*

Theorem 4. *The problem* NonMigratory-Dec(T, M, B) *is NP-complete.*

Online problem. We now study problems in an online context, where properties of requests are not known before their respective release time. We prove that there exists no optimal online algorithm for the minimization of maximum weighted flow even on the very simple case of a single machine and unit request sizes, as outlined in the following theorem. The corresponding proof (available in the companion report [3]) consists in a well-chosen example for which no algorithm can make an optimal choice without knowing the tasks that will be submitted in the future.

Theorem 5. *No online algorithm can be optimal for the scheduling problem* $1|\text{online-}r_i, p_i = 1| \max w_i F_i$.

We now present an adaptation of Single-Simple to the online case, restricted to unit tasks, Single-Unit: at each time step, we consider all submitted

[6] We express non-migratory preemption as $pmtn^*$ in the β-part, not to be confused with the classic *pmtn* constraint.

requests at this time and schedule the one whose flow (if processed now) is the largest. This gives priority to the currently most impacting requests. Unfortunately, even on unit size tasks, this strategy does not lead to an approximation algorithm, as outlined by the following theorem (see proof in the report [3]).

Theorem 6. *The competitive ratio of* Single-Unit *is arbitrarily large for the scheduling problem* $1|\text{online-}r_i, p_i = 1|\max w_i F_i$.

Lower bound. We have seen that our initial scheduling problem, with heterogeneous processing times, no preemption and in an online setting is far from being solvable or even approximable. This motivates the search of lower bounds to constitute a formal baseline and derive performance guarantees. The solution to $R|r_i, pmtn|\max w_i F_i$ provides such a lower bound, which is found by performing a binary search on a Linear Program [18], followed by the reconstruction scheme from Lawler *et al.* [17]. The whole process is detailed in the companion report [3]. This bound is used to assess the performance of practical heuristics in Section 6.

5 Online Heuristics

We recall that a solution to our problem consists, for each request, in choosing a server among the ones holding a replica of the requested data as well as a starting time for each request. These two decisions appear at different places in a real key-value store: the selection strategy R used by the coordinator gives a replica $R(T_i) = M_r$, whose execution policy E_r defines the request starting time $E_r(T_i) = \sigma_i$. This section describes several online replica selection heuristics and execution policies that we then compare by simulation.

5.1 Replica Selection Heuristics

We consider several replica selection heuristics with different levels of knowledge about the cluster state. Some of these levels are hard to achieve in a real system; for instance, the information about the load of a given server will often be slightly out of date. Similarly, the information about the processing time can only be partial, as the size of the requested value cannot be known by the coordinator for large scale data sets, and practical systems generally employ an approximation of this metric, e.g., by keeping track of size *categories* of values using Bloom filters [13]. However, we exploit this exact knowledge in our simulations to estimate the maximal performance gain that a given type of information allows. We now describe selection heuristics, whose properties are shown in Table 1.

Random. The replica is chosen uniformly at random among compatible servers: $M_r = \text{rand}\,\mathcal{M}_i$. This strategy has no particular knowledge.

LeastOutstandingRequests *(*LOR*).* Let $\mathcal{R}(M_j)$ be the number of outstanding requests sent to M_j, i.e., the number of sent requests that received no response yet. This strategy choses the replica that minimizes $\mathcal{R}(M_j)$. It is easy to implement, as it only requires local information; in fact, it is one of the most commonly used in load-balancing applications [23].

HÉRON. We also consider an omniscient version of the replica selection heuristic used by Héron [13]. It identifies requests for values with size larger than a threshold, and avoids scheduling other requests behind such a request for a large value by marking the chosen replica as *busy*. When the request for a large value completes, the replica is marked *available* again. The replica is chosen among compatible servers that are *available* according to the scoring method of C3 [23].

EARLIESTFINISHTIME (EFT). Let FINISHTIME(M_j) denote the earliest time when the server M_j becomes available, i.e., the time at which it will have emptied its execution queue. The chosen replica is the one with minimum FINISHTIME(M_j) among compatible servers. Knowing FINISHTIME is hard in practice, because it assumes the existence of a mechanism to obtain the exact current load of a server. A real system would use a degraded version of this heuristic.

EARLIESTFINISHTIME-SHARDED (EFT-S). For this heuristic, servers are specialized: we define small servers, which execute only requests for small values, and large servers, which execute all requests for large values and some requests for small values when possible (similarly to size-aware sharding [10]). Once the server is chosen, each request is scheduled using the EFT strategy.

For the following experiments, we define large servers as the set of servers $\{M_b\}_{1 \leq b \leq m}$ such that $b \bmod k = 0$ (recall k is the replication factor). This makes sure that one server in each interval $\mathcal{M}_i$ is capable of treating requests for large values. We define a threshold parameter ω to distinguish between requests for small and large values: requests with duration larger than ω are treated by large servers only, while others can be processed by all available servers. We derive the threshold ω from the size distribution, by choosing the parameter ω so that the total work is k times larger than the work on large servers on average (see details in the companion research report [3]).

5.2 Local Queue Scheduling Policies

We now present scheduling policies locally enforced by replicas. Each replica handles an execution queue $\mathcal{Q}$ in which coordinators send requests, and then decides of the order of executions. In a real key-value store, these policies should be able to extract exact information on the local values, and in particular their sizes, as a single server is responsible for a limited number of keys. We consider the following local policies, summarized in Table 1.

FIRSTINFIRSTOUT (FIFO). This is a classic strategy, which is commonly used as a local scheduling policy in key-value stores (e.g., Cassandra [16]). The requests in $\mathcal{Q}$ are ordered by non-increasing insertion time, i.e., the first request that entered the queue (the one with the minimum r_i) is the first to be executed.

MAXSTRETCH. We propose another strategy, which locally reorders requests. When a server becomes available at time t, the next request T_i to be executed is the one in $\mathcal{Q}$ whose stretch $(t + p_i - r_i)/p_i$ is the highest. This favors requests for

Table 1. Properties of replica selection and local queue scheduling heuristics. ACK-Done denotes the need to acknowledge the completion of sent requests. FinishTime is the knowledge of available times of each server. p_i denotes the processing times of local requests and r_i their release times. N is the number of local requests in $\mathcal{Q}$ and m is the total number of servers.

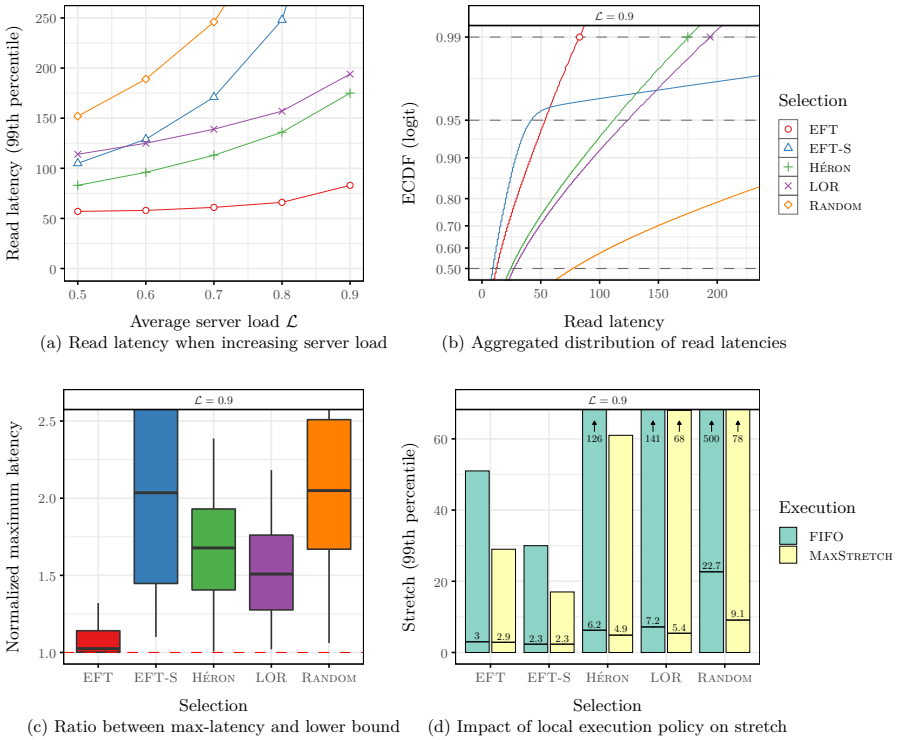
Replica Selection		Local Policy	
Heuristic Knowledge	Complexity	Heuristic Knowledge	Complexity
Random None	$O(1)$	FIFO None	$O(1)$
LOR ACK-Done	$O(m)$	MaxStretch p_i, r_i	$O(N)$
Héron ACK-Done, $p_i \geq \omega$	$O(m)$		
EFT FinishTime	$O(m)$		
EFT-S FinishTime, $p_i \geq \omega$	$O(m)$		

small values in front of requests for large ones, and thus is a way to mitigate the problem of head-of-line blocking. Note that in any case, starvation is not a concern: focusing on the maximum stretch ensures that all requests will eventually be processed [4].

6 Simulations

We analyse the behavior of previously described strategies and compare them with each other in simulations. We built a discrete-event simulator based on `salabim` 21.0.1 for this purpose, which mimics a real key-value store: coordinators receive user requests and send them to replicas in the cluster, which execute these requests. Each request is first headed to the queue of a server holding a replica of the requested data by the selection heuristic. Then, the queue is reordered by the local execution policy and requests are processed in this order.

Workload and settings. We designed a synthetic heterogeneous workload to evaluate our strategies: value sizes follow a Weibull distribution with scale $\eta = 32\,000$ and shape $\theta = 0.5$; these parameters yield a long-tailed distribution that is consistent with existing file sizes characterizations [11]. User requests arrive at coordinators according to a Poisson process with arrival rate $\lambda = m\mathcal{L}/\bar{p}$, where m is the number of servers, $\mathcal{L}$ is the wanted average server load (defined as the average fraction of time spent by servers on serving requests), and $\bar{p}$ is the mean processing time of requests. Each key has the same probability of being requested, i.e., we do not model skewed popularity. The cluster consists in $m = 12$ servers and we set the replication factor to $k = 3$, which is a common configuration in real implementations [9,16]. The network bandwidth is set to $1/\alpha = 100$ Mbps and the average latency is set to $\beta = 1$ ms. The threshold between small and large values is set to $\omega = 26$ ms, resulting in a proportion of 5 % of requests for large values in the workload. Each experiment is repeated on 15 different scenarios; a given scenario defines the processing times p_i, the release times r_i, and the replication groups $\mathcal{M}_i$ according to described settings.



(a) Read latency when increasing server load
(b) Aggregated distribution of read latencies
(c) Ratio between max-latency and lower bound
(d) Impact of local execution policy on stretch

Fig. 2. Distributions of the objective functions for 5 replica selection heuristics. For Fig. (a), (b) and (d), simulations run over 120 s. For Fig. (c), simulations are launched over 1000 requests.

Results. Fig. 2(a) shows the 99$^{\text{th}}$ quantile of read latency (in milliseconds) as a function of average server load for each selection heuristic and for load values ranging from 0.5 to 0.9, with FIFO as local policy. In this context, the maximum of the distribution is impacted by rare events of varying amplitude, which makes this criterion unstable. The stability of the 99$^{\text{th}}$ quantile allows comparing more confidently the performance between scenarios with identical settings. For an average load $\mathcal{L} = 0.9$, Fig. 2(b) shows Empirical Cumulative Distribution Functions (ECDF) of read latencies for each selection heuristics. The dashed horizontal lines respectively represent median, 95$^{\text{th}}$ and 99$^{\text{th}}$ percentile. The comparison of selection heuristics with the lower bound introduced in Section 4 is shown in Fig. 2(c). We normalize the max-latency obtained by a selection heuristic with this lower bound. Each boxplot represents the distribution of these normalized maximums among the 15 different scenarios. Horizontal red line locates the lower bound. Fig. 2(d) illustrates the effect of different lo-

cal execution policies on the 99$^{\text{th}}$ quantile of stretch. Horizontal lines represent median values.

We see in Fig. 2(a) that the choice on selection heuristic is critical for read latency, as the 99$^{\text{th}}$ quantile can often be improved by a factor 2 compared to state-of-the-art strategies LOR and HÉRON, without increasing median performance as confirmed in Fig. 2(b). This highlights the fact that knowing the current load of a server, and thus its earliest available time, as used in the EFT strategy, allows to get very close to the lower bound. Fig. 2(c) also shows that EFT yields the most stable maximums between scenarios, as more than 50 % of normalized values range from 1.0 to 1.15. This improves the confidence that this strategy will perform well in a majority of cases. For the stretch metric, where latencies are weighted by processing times, EFT-S performs even better than EFT (Fig. 2(d)), yielding a 99$^{\text{th}}$ quantile of 30 (resp. 18) when coupled with FIFO (resp. MAXSTRETCH). This is due to the nature of EFT-S that favors requests for small values, which are in majority in the workload. However, EFT-S does not perform well for the last quantiles in the latency distribution; this corresponds to the 5 % of requests for large values that are delayed in order to avoid head-of-line blocking situations. Fig. 2(d) also illustrates the significant impact of local execution policies on the stretch metric: local reordering according to MAXSTRETCH favors requests for small values, which results in an improvement for all selection strategies, even on the median values. Note that this does not necessarily improve latency, as FIFO is well-known to be the optimal strategy for max-flow on a single machine [4].

7 Conclusion

This study defines a formal model of a key-value store in order to derive maximal performance achievable by a real online system, and states the associated optimization problem. We also provide theoretical results on various problems related to our main scheduling problem. After showing the difficulty of this problem, we describe some investigations on a lower bound. We develop online heuristics and compare them with state-of-the-art strategies such as LOR, HÉRON [13] or size-aware sharding [10] using simulations. This allows understanding more finely the impact of replica selection and local execution on performance metrics. We hope that our work will help practitioners draw new scheduling strategies. We plan to continue to improve on a lower bound, for example by using resource augmentation models,and we propose to formally analyze EFT with various techniques such as competitive analysis. We wish to study the effect of various assumptions on scheduling, e.g., the impact of skewed key popularity, and to extend the model with multi-get operations [14, 22].

Acknowledgements and Data Availability Statement

The datasets and code generated during and/or analyzed during the current study are available in the Figshare repository: https://doi.org/10.6084/m9.figshare.14755359 [7].

References

1. Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., Paleczny, M.: Workload analysis of a large-scale key-value store. In: ACM SIGMETRICS Performance Evaluation Review. vol. 40, pp. 53–64. ACM (2012)
2. Bansal, N., Pruhs, K.: Server scheduling in the l_p norm: a rising tide lifts all boat. In: ACM STOCS (2003)
3. Ben Mokhtar, S., Canon, L.C., Dugois, A., Marchal, L., Rivière, E.: Taming Tail Latency in Key-Value Stores: a Scheduling Perspective (extended version). Tech. rep. (2021), available online at https://hal.inria.fr/hal-03144818
4. Bender, M.A., Chakrabarti, S., Muthukrishnan, S.: Flow and stretch metrics for scheduling continuous job streams. In: ACM-SIAM Symp. on Disc. Algo. (1998)
5. Brucker, P., Jurisch, B., Krämer, A.: Complexity of scheduling problems with multi-purpose machines. Annals of Operations Research **70**, 57–73 (1997)
6. Brutlag, J.: Speed matters for google web search (2009)
7. Canon, L.C., Dugois, A., Marchal, L.: Artifact and instructions to generate experimental results for the euro-par 2021 paper: "taming tail latency in key-value stores: a scheduling perspective". https://doi.org/10.6084/m9.figshare.14755359
8. Dean, J., Barroso, L.A.: The tail at scale. Communications of the ACM **56**(2), 74–80 (2013)
9. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: amazon's highly available key-value store. In: ACM SIGOPS Oper. Sys. Rev. vol. 41, pp. 205–220 (2007)
10. Didona, D., Zwaenepoel, W.: Size-aware sharding for improving tail latencies in in-memory key-value stores. In: NSDI. pp. 79–94 (2019)
11. Feitelson, D.G.: Workload modeling for computer systems performance evaluation. Cambridge University Press (2015)
12. Graham, R.L.: Bounds for certain multiprocessing anomalies. Bell System Technical Journal **45**(9), 1563–1581 (1966)
13. Jaiman, V., Ben Mokhtar, S., Quéma, V., Chen, L.Y., Rivière, E.: Héron: Taming tail latencies in key-value stores under heterogeneous workloads. SRDS, IEEE (2018)
14. Jaiman, V., Mokhtar, S.B., Rivière, E.: TailX: Scheduling heterogeneous multiget queries to improve tail latencies in key-value stores. DAIS (2020)
15. Jiang, W., Xie, H., Zhou, X., Fang, L., Wang, J.: Haste makes waste: The on–off algorithm for replica selection in key–value stores. JPDC **130**, 80–90 (2019)
16. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review **44**(2), 35–40 (2010)
17. Lawler, E.L., Labetoulle, J.: On preemptive scheduling of unrelated parallel processors by linear programming. Journal of the ACM (JACM) **25**(4), 612–619 (1978)
18. Legrand, A., Su, A., Vivien, F.: Minimizing the stretch when scheduling flows of divisible requests. Journal of Scheduling **11**(5), 381–404 (2008)

19. Lenstra, J.K., Kan, A.R., Brucker, P.: Complexity of machine scheduling problems. Studies in integer programming **1**, 343–362 (1977)
20. Leung, J.Y.T., Li, C.L.: Scheduling with processing set restrictions: A survey. International Journal of Production Economics **116**(2), 251–262 (2008)
21. Li, J., Sharma, N.K., Ports, D.R., Gribble, S.D.: Tales of the tail: Hardware, OS, and application-level sources of tail latency. In: ACM Symp. Cloud Comp. (2014)
22. Reda, W., Canini, M., Suresh, L., Kostić, D., Braithwaite, S.: Rein: Taming tail latency in key-value stores via multiget scheduling. EuroSys (2017)
23. Suresh, L., Canini, M., Schmid, S., Feldmann, A.: C3: Cutting tail latency in cloud data stores via adaptive replica selection. NSDI (2015)