A Scheduling Framework for Distributed Key-Value Stores and its Application to Tail Latency Minimization

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

[1]LIRIS, Lyon, France.
[2]FEMTO-ST Institute, Université de Franche-Comté, Besançon, France.
[3]LIP, École Normale Supérieure de Lyon, CNRS & Inria, Lyon, France.
[4]ICTEAM, UCLouvain, Louvain-la-Neuve, Belgium.
This is an extended version of Ben Mokhtar, Canon, Dugois, Marchal, and Rivière (2021).

*Corresponding author(s). E-mail(s): anthony.dugois@ens-lyon.fr;
Contributing authors: sonia.benmokhtar@insa-lyon.fr; louis-claude.canon@univ-fcomte.fr;
loris.marchal@ens-lyon.fr; etienne.riviere@uclouvain.be;
[†]These authors contributed equally to this work.

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 that highlight which properties enable limiting tail latency: for instance, the EARLIESTFINISHTIME strategy—which uses the earliest next 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% (Brutlag, 2009).

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 degrade the overall service latency 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 95th and 99th percentiles in the distribution of query latencies show values that can be several orders of magnitude higher than the median (Atikoglu, Xu, Frachtenberg, Jiang, & Paleczny, 2012; Dean & Barroso, 2013).

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 (DeCandia et al., 2007; Lakshman & Malik, 2010). 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 dynamically schedule requests to different replicas in order to reduce tail latency (Jaiman, Ben Mokhtar, Quéma, Chen, & Rivière, 2018; Jiang, Xie, Zhou, Fang, & Wang, 2019; Suresh, Canini, Schmid, & Feldmann, 2015). When the request reaches the selected replica, it is inserted into a queue and a local 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);
- an optimal polynomial-time offline algorithm, an NP-completeness result and a lower bound on the competitive ratio of any online algorithm 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, as well as the related code, data and analysis, are available online[1]. A shorter version of this work has also been published at EuroPar 2021 (Ben Mokhtar et al., 2021).

2 Related Work

We review related work on key-value stores and systems contributions for reducing latency, and on latency minimization in scheduling theory.

2.1 Tail Latency in Key-Value Stores

The principles of key-value stores were first documented by Amazon with Dynamo (DeCandia et al., 2007). Cassandra (Lakshman & Malik, 2010) is a widely-used open-source key-value store, following principles similar to that of Dynamo; other popular key-value stores include Redis (Carlson, 2013), memcached (Jose et al., 2011), and document stores such as MongoDB (Chodorow, 2013).

Key-value stores implement data partitioning for horizontal scalability. Typically, data is spread over a cluster of servers using consistent hashing. This consists in treating the output of a hashing function as a ring; each server is then assigned a position on this circular space and becomes responsible for all data between it and its predecessor. The position of a data item is decided by

---

[1]https://doi.org/10.6084/m9.figshare.21750605.v1

hashing the corresponding key (DeCandia et al., 2007; Lakshman & Malik, 2010). Replication is implemented on top of data partitioning, by duplicating each data item on the successors of its assigned server.

Various designs have been proposed to improve response time and in particular reduce tail latency in key-value stores. They include using redundant requests (Vulimiri et al., 2013; Wu, Yu, & Madhyastha, 2015), performing smart resource allocation (Didona & Zwaenepoel, 2019), or employing hybrid scheduling (Delgado, Didona, Dinu, & Zwaenepoel, 2016; Delgado, Dinu, Kermarrec, & Zwaenepoel, 2015). We are interested in this paper in replica selection strategies (Jaiman et al., 2018; Jiang et al., 2019; Suresh et al., 2015). 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 (Lakshman & Malik, 2010), 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 (Suresh et al., 2015) 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. This does not reflect the reality of storage workloads, where values may be highly heterogeneous in size (Atikoglu et al., 2012). Requests for small values may be scheduled behind requests for large values, creating a head-of-line blocking problem and increasing tail latency. Coordinator servers only know the requested key and they do not know the size of data items they do not hold. This makes inferring the service time of a request difficult, as the workload is not known in advance. Héron (Jaiman et al., 2018) proposes to address this problem by propagating across the cluster the identity of values whose size is over a threshold, together with load information and pending requests to such large values. This information is stored at each coordinator using a Bloom filter, and Héron avoids scheduling requests for small values behind pending requests for large values. Size-aware sharding (Didona & Zwaenepoel, 2019) avoids head-of-line blocking on a specific server, by specializing some of its cores to serve only requests for large values. Local scheduling can take into account the specificities of the data structure used for storing updates to the local values under write-heavy workloads, such as with Log-Structured Merge Key-Value Stores (Balmau et al., 2020). Other systems, such as REIN (Reda, Canini, Suresh, Kostić, & Braithwaite, 2017) or TailX (Jaiman, Mokhtar, & Rivière, 2020), 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, Sharma, Ports, and Gribble (2014) propose a *single-node* model of a complete hardware, application and operating system stack using queueing 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 (i.e., *multi-node*) storage services.

2.2 Latency in Scheduling Theory

Minimization of latency, i.e., 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 maximum flow (also known as *max-flow*, or $F_{\max}$) and the average flow (also known as *sum-flow*, or $\sum F_i$). We focus on the former, as explained in Section 3.

Minimizing maximum flow. It is well-known that the maximum flow criterion is minimized by the First In First Out (FIFO) strategy on a single-machine (Bender, Chakrabarti, & Muthukrishnan, 1998). This scheme is also $(3 - 2/m)$-competitive on m machines with and without preemption (Bender et al., 1998; Mastrolilli, 2004). Ambühl and Mastrolilli (2005) gave a $(2 - 1/m)$-competitive algorithm for the preemptive case, which is the best possible competitive ratio for this problem. The maximum weighted flow $(\max w_i F_i)$ is sometimes considered in order to give more importance to some requests. For example, Bender et al. (1998) introduced the stretch S_i, where the weight w_i is the inverse of the request service time p_i $(S_i = F_i/p_i)$, to express and study the notion of fairness for scheduling HTTP requests in web servers. For single-machine problems, they proved that no polynomial-time algorithm can approximate the offline non-preemptive problem of optimizing the max-stretch criterion (i.e., $S_{\max} = \max S_i$) within a factor $\Omega(n^{1-\varepsilon})$ for any $\varepsilon > 0$, unless $P = NP$. They also exhibit a Fully Polynomial-Time Approximation Scheme for the preemptive case, and they derive an $O(\sqrt{\Delta})$-competitive algorithm from the Earliest Deadline First strategy, with Δ being the ratio between the largest processing time to the smallest one $(\Delta = \max p_i/\min p_i)$. Later, Legrand, Su, and Vivien (2008) presented a polynomial-time algorithm to solve the offline minimization of maximum weighted flow time on unrelated machines when preemption is allowed. The FIFO strategy is also shown to be Δ-competitive for the online problem of minimizing the max-stretch on one machine, and a lower bound of $\frac{1}{2}\Delta^{\sqrt{2}-1}$ is derived. Saule, Bozdağ, and Çatalyürek (2012) improved this result by showing a lower bound of $\frac{1}{2}(\Delta + 1)$ on a single server and $\frac{1}{2}(\frac{\Delta}{m+1} + 1)$ on m servers. Finally, Dutot, Saule, Srivastav, and Trystram (2016) closed the online problem of minimizing the stretch on a single machine by proving a lower bound of $\frac{1}{2}\Delta(\sqrt{5} - 1) + 1$, which is tight. Some additional works deal with the minimization of maximum weighted flow time under different assumptions and models (Anand, Bringmann, Friedrich, Garg, & Kumar, 2017; Bansal & Cloostermans, 2016; Lucarelli, Moseley, Thang, Srivastav, & Trystram, 2019). Tables 1 and 2 provide a summary of results on max-flow minimization.

Data replication in scheduling. An important consequence of data replication is that a given request i cannot be executed by any server; it must be processed by a server in the subset of replicas $\mathcal{M}_i$ able to handle it. In scheduling literature, this constraint is known as *restricted assignment*, *multipurpose machines*, *processing set restrictions* or even *eligibility constraints*. The great majority of problems involving such constraints focus on makespan minimization in various settings (Lee, Leung, & Pinedo, 2013; Leung & Li, 2008, 2016). Anand et al. (2017) give a strong lower bound of $\Omega(m)$ on the competitive ratio of any online algorithm trying to minimize the maximum flow time when unstructured processing set restrictions are considered. Brucker, Jurisch, and Krämer (1997) used a routine based on the minimum cost matching problem to solve problems $Q|\mathcal{M}_i, p_i = 1|\sum w_i U_i$ and $P|\mathcal{M}_i, r_i, p_i = 1|\sum w_i U_i$ in polynomial time. Another notable result is that $P|\mathcal{M}_i, pmtn|\sum C_i$ can be solved by transforming any preemptive schedule in a non-preemptive one without worsening the objective.

3 Formal Model

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

3.1 Application and Platform

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. Each request contains a *key* that is used to designate a specific *value* (or data item) in the store. Answering (or processing) the request consists in sending the corresponding value to the initiator of the request. As key-value stores replicate each data item on k servers (k is called the *replication factor*), we get a multipurpose machines problem, where a given server may execute a request only if it holds the corresponding value. We define $\mathcal{M}_i \subseteq M$ as the set of machines able to process the request T_i. In Graham's $\alpha|\beta|\gamma$ notation of scheduling problems, this constraint is commonly denoted by $\mathcal{M}_i$ in the β-part. Note that in our case, $|\mathcal{M}_i| = k$ for all T_i.

Table 1: Complexity of max-flow minimization problems. P, Q and R respectively denote parallel machines, related machines, and unrelated machines. $\max w_i F_i$, $F_{\max}$ and $S_{\max}$ respectively denote maximum weighted flow, max-flow and max-stretch criteria. Arrows are reduction relationships ($A \to B$ means that A is a special case of B). A sign **+** (resp. **-**) means that the problem is NP-hard (resp. polynomially solvable) via the reduction relationship. Incompatible problem designations are noted $\emptyset$.

Objective	Constraints	1 $\longrightarrow$	P $\longrightarrow$	Q $\longrightarrow$	R
$\to \max w_i F_i$	$\to \quad r_i$	+	+	+	+
	$r_i, pmtn$	-	-	-	p. solvable (2008)
	$r_i, pmtn^*$	-	NP-hard [Th. 3]	+	+
	$\Rightarrow \quad \circ$	p. solvable (1993)	+	+	+
	$p_i = p$	-	-	p. solvable [Th. 1]	$\emptyset$
$F_{\max}$	r_i	p. solvable (1998)	NP-hard [Sec. 4.1]	+	+
$S_{\max}$	$\to \quad r_i$	NP-hard (1998)	+	+	+
	$\circ$	-	s. NP-hard (2021)	+	+

Table 2: Results on maximum (weighted) flow minimization. $P|\mathcal{M}_i$ denotes parallel machines with processing set restrictions. We have $P \to Q \to R$ and $P \to P|\mathcal{M}_i \to R$. (F)PTAS stands for (Fully) Polynomial-Time Approximation Scheme. *any* indicates that the approximation ratio applies to any algorithm of the corresponding type. The approximation ratio of an online algorithm should be seen as a competitive ratio.

Objective	Env.	Preemption	Algorithm	Type	Approx./Competitive ratio	Ref.
$\max w_i F_i$	1	Non-preemptive	*any*	Online	$\geq \Delta + 1$	Th. 4
	P	Non-preemptive	*any*	Online	$\geq \Omega(w_{\max}/w_{\min})$	(2016)
	R	Preemptive	Legrand et al. (2008)	Offline	Optimal	(1978) (1984) (2008)
$F_{\max}$	P	Non-preemptive	FIFO	Online	$3 - 2/m$	(1998)
			any	Online	$\geq 2 - 1/m$	(2005)
		Preemptive	FIFO	Online	$3 - 2/m$	(2004)
			Ambühl and Mastrolilli (2005)	Online	$2 - 1/m$	(2005)
			any	Online	$\geq 2 - 1/m$	(2005)
	$P|\mathcal{M}_i$	Non-preemptive	*any*	Online	$\geq \Omega(m)$	(2017)
	Q	Non-preemptive	DOUBLE-FIT	Online	13.5	(2016)
			SLOW-FIT	Online	$\geq \Omega(m)$	(2016)
			GREEDY	Online	$\geq \Omega(\log m)$	(2016)
	R	Non-preemptive	Bansal and Kulkarni (2015)	Offline	$O(\log n)$	(2015)
			Bansal and Kulkarni (2015)	PTAS	$1 + \varepsilon$ in $n^{O(m/\varepsilon)}$	(2005)
			Mastrolilli (2004)	FPTAS	$1 + \varepsilon$ in $O(nm(n^2/\varepsilon)^m)$	(2004)
$S_{\max}$	1	Non-preemptive	FIFO	Online	Δ	(2008)
			WDA	Online	$\Delta(\sqrt{5} - 1)/2 + 1$	(2016)
			any	Online	$\geq \Delta(\sqrt{5} - 1)/2 + 1$	(2016)
			any	Offline	$\geq \Omega(n^{1-\varepsilon})$	(1998)
		Preemptive	STRETCH-SO-FAR EDF	Online	$O(\sqrt{\Delta})$	(1998)
			Bender et al. (1998)	PTAS	$1 + \varepsilon$	(1998)
	P	Non-preemptive	FIFO	Online	$2\Delta + 1$	(2012)
			any	Online	$\geq \Omega(\Delta)$	(2016)

There is no precedence relations 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 that is equal to the average network latency L plus data sending time $z_i\tilde{B}$, which is proportional to the size z_i of the value this request is looking for (factor $\tilde{B}$ represents the inverse of the bandwidth), i.e., $p_i = z_i\tilde{B} + L$. We consider the online model, that is to say, a request is unavailable before its release time r_i ($r_i \geq 0$) and its properties are unknown as well. Unless stated otherwise, the model is clairvoyant, i.e., the exact processing time p_i of a request T_i becomes known at release time r_i.

3.2 Problem Statement

There is no objective criterion that can straightforwardly represent the optimization of tail latency, as there is no unique formal definition of this system concept. Different works consider the 95th percentile, the 99th percentile, or the maximum of the latency distribution, 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 the response time of a real system is higher 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 request 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 three 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 ($w_i = 1/p_i$) gives the same importance to each request, but this favors requests for small values because they are more sensitive to scheduling decisions. Third, the inverse of the square root of the processing time ($w_i = 1/\sqrt{p_i}$) constitutes a compromise between the two previous cases. This last weighting strategy is denoted as the *weak stretch* in this paper.

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:

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

We note this problem $P|\mathcal{M}_i, \text{online} - r_i| \max w_i F_i$ in Graham's notation. A solution is to find a schedule that provides each request T_i with an executing server M_j and a starting time $\sigma_i \geq r_i$. The 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.

4 Max-Flow Minimization

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 requests are all released at the same time, but as soon as we introduce different release dates or processing set restrictions, the problem gets harder to tackle. Nevertheless, a lower bound can be computed.

4.1 Zero Release Times

We first focus on the non-preemptive problem of minimizing the maximum weighted flow time when all requests are available at time 0. Remark that in this case, minimizing the maximum weighted flow time is strictly equivalent to minimizing the weighted makespan $\max w_i C_i$.

Algorithm 1 MAX-WEIGHT

Schedule requests in non-increasing order of w_i and put each request on the machine that finishes it the earliest

---

Let us consider the MAX-WEIGHT scheduling algorithm (Algorithm 1), which schedules requests by non-increasing order of weights w_i. It is known that MAX-WEIGHT is optimal on a single server (Ben Mokhtar et al., 2021). It does not extend to m parallel identical machines in the general case, but it solves the related machines case when all requests have homogeneous size p.

Theorem 1. MAX-WEIGHT *(Algorithm 1) solves* $Q|p_i = p|\max w_i C_i$ *in polynomial time.*

Proof Let OPT be an optimal schedule with two requests T_j and T_k such that $w_j < w_k$ and where T_k completes at time $C_j + c$ with $c > 0$ (on any server).

Their contribution is $\mathcal{C} = \max(w_j C_j, w_k(C_j + c)) = w_k(C_j + c)$ because $w_j C_j < w_k C_j < w_k(C_j + c)$. If we switch T_j and T_k, then the contribution is $\mathcal{C}' = \max(w_k C'_k, w_j(C'_k + c))$ because $p_j = p_k = p$. By construction, $C'_k + c = C_j + c$, i.e., $C'_k = C_j$. We have $w_k C'_k = w_k C_j < w_k(C_j + c)$ and $w_j(C'_k + c) = w_j(C_j + c) < w_k(C_j + c)$. Hence, $\mathcal{C}' < \mathcal{C}$.

It follows that we can transform OPT in another optimal schedule by switching repeatedly non-sorted requests. MAX-WEIGHT schedules requests one by one in non-increasing order of w_i and put each request on the machine that completes it the earliest. Then, MAX-WEIGHT is optimal because it ensures that if T_j and T_k are two requests such that $w_j \geq w_k$, then T_k completes after T_j (i.e., $C_k = C_j + c$ with $c > 0$). $\square$

On m identical machines, the problem with arbitrary processing times is trivially NP-hard even with unit weights because $P||C_{\max}$ is NP-hard (Lenstra, Kan, & Brucker, 1977). Moreover, minimizing the max-stretch ($P||S_{\max}$) is also NP-Hard (Benoit et al., 2021). We prove that MAX-WEIGHT is an approximation algorithm.

Theorem 2. MAX-WEIGHT *(Algorithm 1) computes a $(2 - 1/m)$-approximation for the problem* $P||\max w_i C_i$*, and this ratio is tight.*

Proof The bound has been established by Hall (1993) and Ben Mokhtar et al. (2021). We prove that it is asymptotically tight by considering the instance with m machines and $n = m(m-1)+1$ requests $\{T_i\}_{1 \leq i \leq n}$ with the following weights and processing times:

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

The request T_n will be scheduled last by the MAX-WEIGHT algorithm, which gives an objective of $\max w_i C_i = (2m-1)W$, whereas an optimal schedule OPT 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 $(2 - 1/m) \cdot W/(W + 1)$ tends to $2 - 1/m$ as $W \to \infty$. $\square$

The two proofs of this section may be easily adapted to a slightly more general problem where the objective is to minimize $\max w_i F_i$ when all release times are identical, but not necessarily equal to 0, i.e., $r_i = r \geq 0$ for all i. However, in this case there is no equivalence anymore between $\max w_i C_i$ and $\max w_i F_i$, because of the weighted objective: $\max w_i F_i = \max w_i(C_i - r)$, which does not simplify.

4.2 Non-Zero Release Times

Legrand et al. (2008) solved the scheduling problem $R|r_i, pmtn|\max w_i F_i$ in polynomial time using a linear formulation of the model. 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 $R||\max w_i F_i$ (Leung & Li, 2008)). In fact, it only differs on one specific aspect: it allows preempting and migrating jobs between machines, that is to say, resuming the execution of an interrupted job on a different machine, 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[2] preemption is allowed. Interestingly, preventing migration makes the problem NP-complete. The proof of this result consists in a reduction from the NP-complete decision problem associated to $P||C_{\max}$ (Lenstra et al., 1977).

Definition 1 (NONMIGRATORY-DEC(T, M, B))**.** *Given a set of requests T, a set of machines M*

---

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

and a bound B, is there a valid non-migratory pre-emptive schedule where each request T_i completes before time $r_i + B/w_i$?

Theorem 3. NONMIGRATORY-DEC(T, M, B) *is NP-complete.*

Proof We prove the NP-completeness of this problem by reduction from $P||C_{\max}$, which is well-known to be NP-complete (Lenstra et al., 1977). To prove that NONMIGRATORY-DEC(T, M, B) belongs to NP, we consider solutions where the number of preemptions of each job is bounded by n, as it is useful to preempt a job only when a new job is released. The validity of such a solution can then be verified in polynomial time.

Building instance. We consider an instance $\mathcal{I}_1$ of the decision version of $P||C_{\max}$: given a set of requests T', a set of servers M' and a bound B', is there a valid non-preemptive schedule where each request completes before time B'? We construct the following instance $\mathcal{I}_2$ of NONMIGRATORY-DEC(T, M, B) from $\mathcal{I}_1$. We first set $M = M'$ and $B = B'$. For each request T'_i we define a request T_i with processing time p_i, a release time $r_i = 0$ and a weight $w_i = 1$. $\mathcal{I}_2$ can clearly be constructed in polynomial time in the size of $\mathcal{I}_1$.

Equivalence of problems. A solution to $\mathcal{I}_1$ trivially constitutes a non-preemptive (thus non-migratory) solution to $\mathcal{I}_2$.

Assume now that $\mathcal{I}_2$ has a solution Π. It means that for each server M_j, we know a set $A_j \subseteq T$ of requests that are preemptively scheduled on M_j exclusively (recall migration is not allowed), and $\max_{T_i \in A_j} C_i$ is the makespan of M_j in Π. As Π is a solution to $\mathcal{I}_2$, $C_i \leq r_i + B/w_i = B$ for all i (by definition), and thus, for all j, $\max_{T_i \in A_j} C_i \leq B$.

For each request T_i, we define the associated set of n_i processing intervals as $\Lambda_i = \{(\sigma_{i,k}, \delta_{i,k})$ s.t. $1 \leq k \leq n_i\}$, where $\sigma_{i,k}$ and $\delta_{i,k}$ respectively denote the start time and the duration of the k-th processing interval of T_i. Note that for all i, k, $\sigma_{i,k} + \delta_{i,k} \leq \sigma_{i,k+1}$.

We can build a solution Π' to $\mathcal{I}_1$ by removing preemptions from Π, i.e., for each request T_i, we rearrange its n_i intervals (without migrating them) such that for all k, $\sigma'_{i,k} + \delta_{i,k} = \sigma'_{i,k+1}$. This is clearly feasible in polynomial time, and as we only permute processing intervals, it does not change the server makespan values. Therefore,

$$\max_{T_i \in A_j} C'_i = \max_{T_i \in A_j} C_i \leq B = B'.$$

$\square$

4.3 Online Problems

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 a lower bound of $\Delta + 1$ (where Δ is the ratio between maximum and minimum processing times) on the competitive ratio of any online algorithm trying to minimize the maximum weighted flow time on a single server, as outlined in the following theorem.

Theorem 4. *The competitive ratio of any online algorithm is at least $\Delta + 1$ for the problem* $1|\text{online} - r_i| \max w_i F_i$.

Proof Let a, b be values such that $a \geq b > 0$. By contradiction, suppose there exists a ρ-competitive online algorithm $\mathcal{A}$ for the problem $1|\text{online} - r_i| \max w_i F_i$ such that $\rho < a/b + 1$. We now build an adversary request submission strategy that will lead to exceeding this ratio when $\Delta = a/b$. The adversary sends two requests T_1 and T_2 with the following characteristics:

- $r_1 = 0$, $p_1 = a$, $w_1 = 1$;
- $r_2 = \sigma_1 + \varepsilon$, $p_2 = b$, $w_2 = W$, where σ_1 is the start time of T_1 when scheduled by $\mathcal{A}$, ε is an arbitrary value such that $0 < \varepsilon < b(a/b + 1 - \rho)$, and $W = 2a/b + 1$.

When scheduled by $\mathcal{A}$, T_1 completes at time $\sigma_1 + a$ and T_2 completes at time $\sigma_1 + a + b$ in the best case: as the adversary sends T_2 at time $\sigma_1 + \varepsilon$, T_1 has already started and we must wait for its completion. Thus, in this schedule, $w_1 F_1 = \sigma_1 + a$ and $w_2 F_2 \geq W(\sigma_1 + a + b - (\sigma_1 + \varepsilon)) = W(a + b - \varepsilon)$. Therefore,

$$\max w_i F_i \geq \max(\sigma_1 + a, W(a + b - \varepsilon))$$
$$\geq W(a + b - \varepsilon).$$

We now study the performance of an offline schedule OFF on this instance, which executes T_2 first if and only if $\sigma_1 < a - \varepsilon$. We will see that OFF is indeed optimal, as it always reaches an objective of Wb, which is a lower bound on the weighted flow for request T_2. We consider two cases in the analysis, depending on whether T_2 is scheduled first or not.

Case 1: The algorithm $\mathcal{A}$ decides to execute T_1 before time $a - \varepsilon$, i.e., $\sigma_1 < a - \varepsilon$. In the offline schedule, T_2 is executed first at time $r_2 = \sigma_1 + \varepsilon$, which gives $w_2 F_2^{OFF} = Wb$, and then T_1 at time $\sigma_1 + \varepsilon + b$, which gives

$$w_1 F_1^{OFF} = \sigma_1 + \varepsilon + b + a$$
$$< a - \varepsilon + \varepsilon + b + a = 2a + b.$$

Algorithm 2 MAX-FLOW

1: **when** the machine is idle at time t **do**
2:     Execute the pending request i whose current weighted flow $w_i(t + 1 - r_i)$ is the highest

---

As we have chosen W such that $W = 2a/b + 1$, we have $Wb = 2a + b$. Hence, $w_1 F_1^{OFF} < Wb$, and then $\max w_i F_i^{OFF} = w_2 F_2^{OFF} = Wb$.

Case 2: The algorithm $\mathcal{A}$ decides to execute T_1 after time $a - \varepsilon$, i.e., $\sigma_1 \geq a - \varepsilon$. In the offline schedule OFF, T_1 is executed first at time $r_1 = 0$, which gives $w_1 F_1^{OFF} = a$, and then T_2 at time $r_2 = \sigma_1 + \varepsilon \geq a$, which gives $w_2 F_2^{OFF} = Wb$. We have $a < Wb$, hence, $\max w_i F_i^{OFF} = w_2 F_2^{OFF} = Wb$.

In both cases, the objective value of the offline schedule OFF is Wb (and hence OFF is optimal). Thus,

$$\frac{\max w_i F_i}{\max w_i F_i^{OFF}} \geq \frac{W(a + b - \varepsilon)}{Wb} = \frac{a}{b} + 1 - \frac{\varepsilon}{b}.$$

As $\varepsilon < b(a/b + 1 - \rho)$, we have

$$\frac{\max w_i F_i}{\max w_i F_i^{OFF}} > \frac{a}{b} + 1 - \frac{b(a/b + 1 - \rho)}{b} = \rho.$$

This contradicts the ρ-competitiveness of $\mathcal{A}$, thus the competitive ratio is at least $a/b + 1$. Note that in this instance, $\max p_i = p_1 = a$ and $\min p_i = p_2 = b$, i.e., $a/b = \max p_i / \min p_i = \Delta$. We conclude that the competitive ratio of any online algorithm is at least $\Delta + 1$ for the problem $1|\text{online}-r_i| \max w_i F_i$. $\qquad\square$

We now present an adaptation of the MAX-WEIGHT algorithm to the online case and restricted to unit requests, called MAX-FLOW (Algorithm 2): at each time step t, we consider all submitted 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 requests, this strategy does not lead to an approximation algorithm, as outlined by the following theorem.

Theorem 5. *The competitive ratio of* MAX-FLOW *(Algorithm 2) is arbitrarily large for the problem* $1|\text{online}-r_i, p_i = 1| \max w_i F_i$.

The proof (detailed in Appendix A) consists in an adversary that first submits a set of k tasks of weight k at time 0. Then, at each time step, a new task is submitted with a weight slightly larger than the previous one (the weight is increased by a factor $1 + 1/k$ at each time step). An optimal strategy consists in scheduling tasks with larger weight

as soon as they arrive, to prevent them from long delays. MAX-FLOW schedules tasks in the order of their submission, as it only computes the potential flow of scheduling each task in the next slot and is not aware that some tasks may be largely delayed. This results in a maximum weighted flow that is larger than k times the optimal one.

4.4 Lower Bound

Our initial problem $P|\mathcal{M}_i, \text{online}-r_i| \max w_i F_i$ (i.e., with heterogeneous processing times, processing set restrictions, arbitrary release times and no preemption allowed) is far from being solvable in polynomial time. It also seems highly difficult to get a reasonable approximation. Still, we would like to have an idea of the best achievable performance on a given instance, in order to fairly compare various online heuristics. This motivates the search of lower bounds to constitute a formal baseline. In fact, the solution to the preemptive problem $R|r_i, pmtn| \max w_i F_i$ provides such a lower bound (even if not tight for our non-migratory problem). It is found by performing a binary search on a Linear Program (Legrand et al., 2008), followed by the reconstruction scheme provided by Lawler and Labetoulle (1978). This bound is used to assess the performance of practical heuristics in Section 6.

5 Online Heuristics

We recall that a solution to the general problem $P|\mathcal{M}_i, \text{online}-r_i| \max w_i F_i$ consists, for each request, in choosing a server among the ones holding a replica of the requested data item 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_u used by the coordinator M_u gives a replica $R_u(T_i) = M_r$, 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.

From now on, we consider the sub-problem where each replication group $\mathcal{M}_i$ follows the standard configuration of key-value store implementations, that is, the cluster is a circular, ordered set of servers $M_1, M_2, \ldots, M_m$, and each $\mathcal{M}_i$ is an interval of servers of size k. In other words, $\mathcal{M}_i = \{M_j, M_{j+1}, \ldots, M_{j+k-1}\}$ for all T_i, where

M_j is the first server storing the requested data item of T_i. We show an example of this structure in Fig. 1.
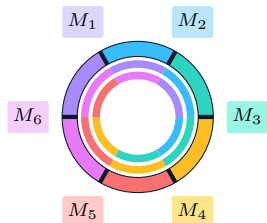
Figure 1: A cluster where replication groups follow a fixed-size interval structure. The ring represents the dataset, splitted over the servers of the cluster. Each colored segment represents a partition of data that is stored on a particular server and replicated on the two consecutive neighbors; for example, the red partition is stored on server M_5 and replicated on servers M_6 and M_1.

5.1 Replica Selection

We consider several replica selection heuristics that leverage different types of information on the cluster state. Note that the perfect knowledge of the cluster state is not achievable 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 (Jaiman et al., 2018). 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.

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 us define $\text{OUT}_u(M_j)$ to be the number of outstanding requests sent from the coordinator M_u to server M_j, i.e., the number of sent requests that received no response yet. The chosen replica minimizes $\text{OUT}_u(M_j)$: $M_r = \text{argmin}_{M_j \in \mathcal{M}_i} \text{OUT}_u(M_j)$. 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 (Suresh et al., 2015).

HÉRON. We also consider an omniscient version of the replica selection heuristic used by Héron (Jaiman et al., 2018). It identifies requests for values whose size is 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 (Suresh et al., 2015). The threshold is chosen according to the wanted proportion of large requests in the workload.

EARLIESTFINISHTIME (EFT). Let $\text{AVAIL}(M_j)$ 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 $\text{AVAIL}(M_j)$ among compatible servers: $M_r = \text{argmin}_{M_j \in \mathcal{M}_i} \text{AVAIL}(M_j)$. Knowing AVAIL 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.

EFT-SHARDED (EFT-S). In this heuristic, we specialize servers; there are 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 technique (Didona & Zwaenepoel, 2019)). Each request for a large value is scheduled on large servers using the EFT strategy, while each request for a small value is scheduled on any server (small or large), also using EFT.

For the following experiments, we define large servers as the set of servers $\{M_b\}_{1 \le b \le m}$ such that $b \mod k = 0$ (recall k is the replication factor). This makes sure that one server in each replication group $\mathcal{M}_i$ is capable of treating requests for large values, as each $\mathcal{M}_i$ is an interval of size k. 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 of stored values. In the best case, when all servers in each replication group are perfectly balanced, requests for small values are scheduled on small servers only and requests for large values on large servers only. It means that the total work is k times larger than the work on large servers on average. Let X be the random variable that models the size distribution of stored values, and f_X denote its probability density function. We denote by $p(X) = \tilde{B}X + L$ the duration of the corresponding request (where $\tilde{B}$ is the inverse of the bandwidth and L the latency), and by $p_\omega(X)$ the duration if it is a large value (and zero otherwise), that is:

$$p_\omega(X) = \begin{cases} p(X) & \text{if } p(X) \geq \omega, \\ 0 & \text{otherwise.} \end{cases}$$

Then, the expected work on large servers when any request is submitted is

$$E\left[p_\omega(X)\right] = \int_{x=0}^{\frac{\omega-L}{\tilde{B}}} 0 f_X(x)dx \\ + \int_{x=\frac{\omega-L}{\tilde{B}}}^{\infty} (\tilde{B}x + L)f_X(x)dx.$$

It should be equal to the expected work when any request is submitted, $E[p(X)]$, divided by k. This leads to finding ω such that

$$E\left[p_\omega(X)\right] = \frac{1}{k}E[p(X)]. \qquad (1)$$

This heuristic has to be able to distinguish requests for small and large values with respect to ω; it could be achieved in practice with combined Bloom filters, in a similar fashion than Héron (Jaiman et al., 2018).

STATICWINDOW (SW). Requests are no longer scheduled on reception, but every q time units, where q is a parameter of the heuristic. The set Q denotes the requests received during this window of q time units. Let t be the time at which requests from Q must be scheduled (requests with $t < r_i \leq t + q$ form the next batch and must be scheduled at time $t+q$). We assume here a *centralized* system, where a unique scheduler receives and schedules all requests. The underlying idea is to be able to make choices based on more exhaustive

---

Algorithm 3 SUFFERAGE-SW

1:	**repeat** every q time units
2:	**for all** $T_i \in Q$ **do**
3:	$\rho(T_i) \leftarrow \text{argmin}_{M_j \in \mathcal{M}_i} \mathcal{F}(T_i, M_j)$
4:	$\rho'(T_i) \leftarrow \text{argmin}_{M_j \in \mathcal{M}_i \setminus \rho(T_i)} \mathcal{F}(T_i, M_j)$
5:	$\text{SUF}(T_i) \leftarrow \mathcal{F}(T_i, \rho'(T_i)) - \mathcal{F}(T_i, \rho(T_i))$
6:	**end for**
7:	**while** Q is not empty **do**
8:	$T_s \leftarrow \text{argmax}_{T_i \in Q} \text{SUF}(T_i)$
9:	Schedule T_s on $\rho(T_s)$
10:	$Q \leftarrow Q \setminus \{T_s\}$
11:	Update ρ, ρ' and SUF
12:	**end while**

---

workload information than previous greedy heuristics. This heuristic must therefore also decide the order in which the requests of Q are scheduled. We derive two versions.

SUFFERAGE-SW (SSW). The Sufferage heuristic (Maheswaran, Ali, Siegel, Hensgen, & Freund, 1999) inspired this strategy. Let $\mathcal{F}$ be the function giving the estimated weighted flow $\mathcal{F}(T_i, M_j) = w_i(\max(r_i, \text{AVAIL}(M_j)) + p_i - r_i)$ of T_i when scheduled on M_j as soon as possible. Let $\rho(T_i) = \text{argmin}_{M_j \in \mathcal{M}_i} \mathcal{F}(T_i, M_j)$ be the *best* server for T_i, i.e., the one minimizing its weighted flow, and $\rho'(T_i) = \text{argmin}_{M_j \in \mathcal{M}_i \setminus \rho(T_i)} \mathcal{F}(T_i, M_j)$ be the *second best* server for T_i. Then, we define the sufferage value $\text{SUF}(T_i) = \mathcal{F}(T_i, \rho'(T_i)) - \mathcal{F}(T_i, \rho(T_i)) > 0$ as the difference of weighted flow values on $\rho'(T_i)$ and $\rho(T_i)$. The request we choose to schedule is the one which suffers the most if we schedule it on its second best server: $T_s = \text{argmax}_{T_i \in Q} \text{SUF}(T_i)$. The chosen replica is $\rho(T_s)$: $M_r = \rho(T_s) = \text{argmin}_{M_j \in \mathcal{M}_i} \mathcal{F}(T_s, M_j)$.

Request T_s is then removed from Q, and we update sufferage values of remaining requests. Algorithm 3 describes this procedure. This strategy runs in time $O(n^2m)$ and uses a space $O(n)$ per time window.

MAXMIN-SW (MSW). This strategy is inspired from the Max-Min heuristic (Maheswaran et al., 1999). We build a matrix MAT whose rows are requests of set Q and columns are servers, where

$$\text{MAT}[T_i, M_j] = \begin{cases} \mathcal{F}(T_i, M_j) & \text{if } M_j \in \mathcal{M}_i, \\ +\infty & \text{otherwise.} \end{cases}$$

Algorithm 4 MAXMIN-SW

1:	**repeat** every q time units
2:	**for all** $T_i \in Q$ **do**
3:	**for all** $M_j \in M$ **do**
4:	**if** $M_j \in \mathcal{M}_i$ **then**
5:	MAT$[T_i, M_j] \leftarrow \mathcal{F}(T_i, M_j)$
6:	**else**
7:	MAT$[T_i, M_j] \leftarrow +\infty$
8:	**end if**
9:	**end for**
10:	**end for**
11:	**while** Q is not empty **do**
12:	$T_s \leftarrow \text{argmax}_{T_i \in Q} \mathcal{F}_{\text{best}}(T_i)$
13:	$M_r \leftarrow \text{argmin}_{M_j \in M} \text{MAT}[T_s, M_j]$
14:	Schedule T_s on M_r
15:	$Q \leftarrow Q \setminus \{T_s\}$
16:	Remove row T_s from MAT
17:	Update column M_r in MAT
18:	**end while**

The best weighted flow of request T_i is $\mathcal{F}_{\text{best}}(T_i) = \mathcal{F}(T_i, \rho(T_i)) = \min_{M_j \in M} \text{MAT}[T_i, M_j]$. Then, we schedule the request T_s whose *best* objective value is the highest: $T_s = \text{argmax}_{T_i \in Q} \mathcal{F}_{\text{best}}(T_i)$. The chosen replica minimizes the objective value of T_s: $M_r = \text{argmin}_{M_j \in M} \text{MAT}[T_s, M_j]$.

The request T_s is then removed from the set Q, as well as the related row in the matrix MAT, and the column M_r is updated with new values. These operations are repeated until Q is empty (see Algorithm 4). This strategy runs in time $O(n^2 m)$ and uses a space $O(nm)$ per time window.

Table 3 summarizes the properties of our selection heuristics.

Table 3: Properties of replica selection heuristics. ACK denotes the need to acknowledge the completion of sent requests. AVAIL 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 requests in Q and m is the total number of servers.

Heuristic	Knowledge	Type	Complexity
RANDOM	None	Distributed	$O(1)$
LOR	ACK	Distributed	$O(m)$
HÉRON	ACK, $p_i \geq \omega$	Distributed	$O(m)$
EFT	AVAIL	Distributed	$O(m)$
EFT-S	AVAIL, $p_i \geq \omega$	Distributed	$O(m)$
SSW	AVAIL, p_i, r_i	Centralized	$O(n^2 m)$
MSW	AVAIL, p_i, r_i	Centralized	$O(n^2 m)$

Table 4: Properties of local scheduling heuristics. p_i denotes the processing times of local requests and r_i their release times. N is the number of local requests in $\mathcal{Q}$.

Heuristic	Knowledge	Complexity
FIFO	None	$O(1)$
MWF	p_i, r_i	$O(N)$

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.

FIRSTINFIRSTOUT (FIFO). This strategy is commonly used as a local scheduling policy in key-value stores (e.g., Cassandra (Lakshman & Malik, 2010)). 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.

MAXWEIGHTEDFLOW (MWF). We propose another strategy, which reorders requests. When the server becomes available at time t, the next request T_s to be executed is the one whose weighted flow is the highest: $T_s = \text{argmax}_{T_i \in \mathcal{Q}} w_i(t + p_i - r_i)$. We consider that p_i is always known, as the request T_i necessarily looks for a value that is hosted on the local server. Consequently, we know the size of the value, and the request processing time can be estimated accordingly. MWF is a general execution policy that considers the request weights as defined by the system designer. In any case, starvation is not a concern: focusing on the maximum weighted flow ensures that all requests will eventually be processed. Note that when coupled with the stretch metric ($w_i = 1/p_i$), MWF is equivalent to the strategy that selects the request with maximal stretch. This favors requests for small values in front of requests for large ones, and thus is a way to mitigate the problem of head-of-line blocking. Table 4 summarizes the properties of our local heuristics.

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 Python 3.8 and the salabim package[3] (v21.0.1) 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.

6.1 Workload and Settings

We designed a synthetic heterogeneous workload to evaluate our strategies: the sizes of data items follow a Weibull distribution with scale $\eta = 32\,000$ and shape $\theta = 0.5$, which gives an average value size of 64 kilobytes (standard deviation: 143 kB; median: 15 kB). These parameters yield a long-tailed distribution that is consistent with existing file sizes characterizations (Feitelson, 2015). User requests arrive at coordinators according to a Poisson process with arrival rate $\lambda = m\mathcal{L}/\overline{p}$, 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 $\overline{p}$ is the average processing time of requests. Hence, release times are chosen such that the time between two consecutive arrivals follows an exponential distribution with parameter λ. Each key has the same probability of being requested, i.e., we do not model skewed popularity. In other words, replication groups $\mathcal{M}_i$ are chosen with uniform probability. The cluster consists in $m = 15$ servers and we set the replication factor to $k = 3$, which is a common configuration in real implementations (DeCandia et al., 2007; Lakshman & Malik, 2010). The network bandwidth is set to $1/\tilde{B} = 100$ Mbps and the average latency is set to $L = 1$ ms. Note that the number of requests directly depends on the arrival rate λ and the duration of the simulation; for example, a simulation running over 120 seconds on 15 servers with a 90% average load and an average service time of 6.12 ms yields about 250 000 requests in total.

For the threshold between requests for small and large values, we plug the density function of our Weibull distribution in Equation (1) and solve it numerically for ω:

$$E\left[p_\omega(X)\right] = \frac{1}{k}\int_{x=0}^{\infty}(\tilde{B}x + L)\frac{\theta}{\eta}\left(\frac{x}{\eta}\right)^{\theta-1}e^{-\left(\frac{x}{\eta}\right)^{\theta}}dx.$$

This yields a threshold $\omega \approx 26.4$ ms (for a value size of 318 kB), resulting in a proportion of 5% of requests for large values in the workload. Each experiment is repeated on 10 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.

6.2 Weight Values

We recall that each request in our model is associated to a weight value w_i. Thus far, we considered these weights to be completely arbitrary. We now describe and explain the values we used in our simulations:

- $w_i = 1$ for all $T_i \in T$. This is the classic flow time (or latency) metric.
- $w_i = 1/p_i$. Latency tends to favor large requests over the small ones. One way to work around this behavior is to consider the stretch (weighting the latency with the processing time): it measures the slowdown of a request, i.e., the cost for sharing resources with other requests.
- $w_i = 1/\sqrt{p_i}$. Although the stretch metric is more fair than latency, we noted in some experiments that it tends to be inappropriate under heterogeneous workloads where the majority of requests are small. Small requests are too favored. For instance, if a small request of 1 ms and a large request of 100 ms have a stretch value of 2, then the large request can tolerate a 100 ms delay ($F_i = 200$), whereas the small one can only tolerate a 1 ms delay ($F_i = 2$). Yet it seems reasonable to delay small requests a little more to avoid impacting the large ones too much. This weighting seems to be a trade-off between latency and stretch metrics, and we denote it as the *weak stretch*.

---

[3]https://www.salabim.org.

6.3 Results

Fig. 2 shows Empirical Cumulative Distribution Functions[4] (ECDF) of the flow, the stretch and the weak stretch, for each combination of *distributed* selection heuristic and execution strategy. The dashed horizontal lines respectively represent median, 95th and 99th percentile. Data items are requested with a load $\mathcal{L} = 0.9$, and the simulations run for 120 seconds.

We show in Fig. 3 the ECDF of window-based strategies when servers are subject to a burst, i.e., the arrival rate is very high and the average load is greater than 1. We measure the metrics with average load values of 1 and 3, combined to a FIFO execution. For SSW and MSW, we consider the stretch weighting ($w_i = 1/p_i$), to favor small requests that are in majority in the workload. We recall that these heuristics are centralized, i.e., all requests are scheduled by one coordinator, and the time window is set to $q = 100$ ms. The simulations run over 3 seconds in order to simulate a short burst of requests.

Fig. 4a shows the 99th quantiles of each metric as a function of average server load for each combination of selection and execution heuristics and for load values ranging from 0.5 to 0.9. 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 99th quantile allows comparing more confidently the performance between scenarios with identical settings. For the local execution policy MWF, we discard the case $w_i = 1/\sqrt{p_i}$, as it exhibits performances always worst than the case $w_i = 1/p_i$. The simulations run for 120 seconds.

The comparison of online heuristics with the lower bound introduced in Section 4.4 is shown in Fig. 4b. We normalize the maximum objective $\max w_i F_i$ generated by a given heuristic with the lower bound. Each boxplot[5] represents the distribution of these normalized maximums among 10 different scenarios, for each combination of strategies. Horizontal red bars help to locate the lower bound. Data items are requested with a load

$\mathcal{L} = 0.9$, and the 10 scenarios are solved over 1200 requests.

The first thing to note in Fig. 2-4b is that the choice on replica selection heuristic is indeed critical for read latency, as the 99th 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. This highlights the fact that some properties of the cluster and the workload are more suitable to taming tail latency; in particular, knowing the current load of a server, and thus its earliest available time, allows implementing the EFT strategy and getting very close to the lower bound (Fig. 4b).

Fig. 4b also shows that EFT yields the most stable maximums between scenarios, as more than 50% of normalized max-flow range from 1.0 to 1.15, in particular when coupled with FIFO. This improves the confidence that this strategy will perform close to optimal in a majority of cases, and cannot be significantly improved. On the opposite, when considering the stretch, the gap between the best achieved performance and the lower bound increases significantly. It is yet unclear whether this is because the lower bound is far from the optimal as it exploits migration, or whether the proposed heuristics are not the best suited to the stretch metric, even if EFT-S shows the best results. On a side note, the effect of switching from FIFO to MWF and the relative performance between the heuristics are consistent with Fig. 4a.

For the stretch metric, where latencies are weighted by processing times, EFT-S performs even better than EFT (Fig. 2, 4a), yielding a 99th quantile of 30 (resp. 18) when coupled with FIFO (resp. MWF ($w_i = 1/p_i$)). 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, 4a and 4b also illustrate the significant impact of local execution policies on the stretch metric: local reordering according to MWF ($w_i = 1/p_i$) 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

---

[4]An Empirical Cumulative Distribution Function is the distribution function obtained from the empirical measure of a sample. With enough realizations, it converges to the actual, underlying cumulative distribution function.

[5]A boxplot consists of a bold line for the median, a box for the quartiles and whiskers that extend at most to 1.5 times the interquartile range from the box.
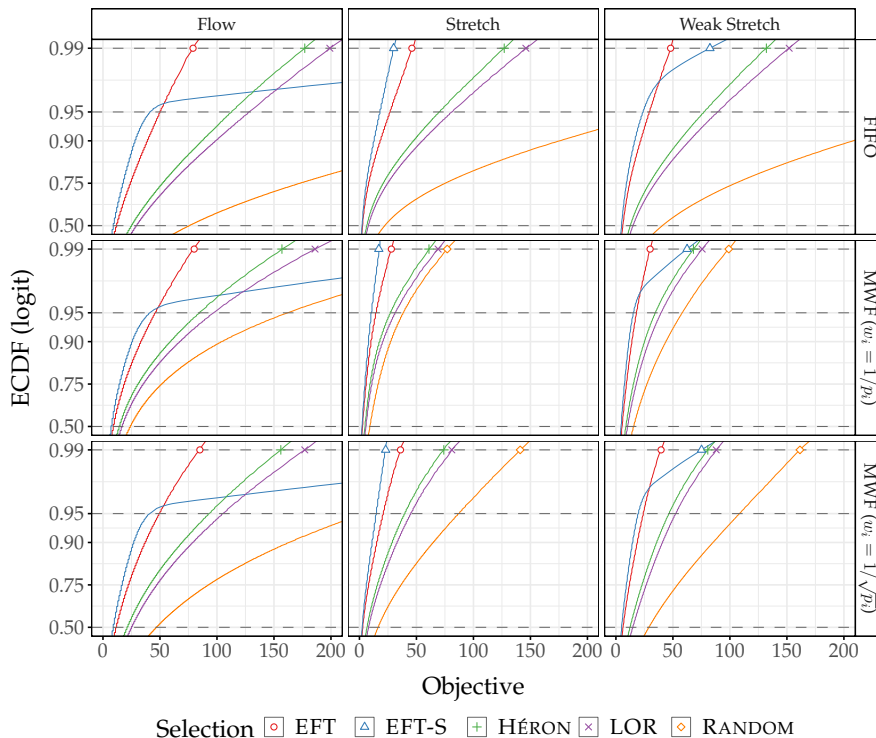
Figure 2: ECDF of flow, stretch and weak stretch metrics given by each combination of distributed selection and execution heuristics in steady-state over 120 seconds, under average load of 90%.

strategy for max-flow on a single machine (Bender et al., 1998). It is confirmed by our observations, as MWF worsen the tail latency.

When a burst occurs, Fig. 3 shows the value of our window-based heuristics. Interestingly, these replica selection strategies do not benefit a lot from centralized and global information about the workload, and are not even effective for realistic load values. When the average load exceeds 300% ($\mathcal{L} \geq 3$) we see that ECDF of EFT and SSW or MSW are similar, but the window-based heuristics never outperform EFT. This seems to confirm that EFT is a close-to-optimal strategy in average, as additional information do not allow to increase performance.

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 (Jaiman et al., 2018) or size-aware sharding (Didona & Zwaenepoel, 2019) 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 (Choudhury, Das, Garg, & Kumar, 2018; Kalyanasundaram & Pruhs, 2000), 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 (Jaiman et al., 2020; Reda et al., 2017).
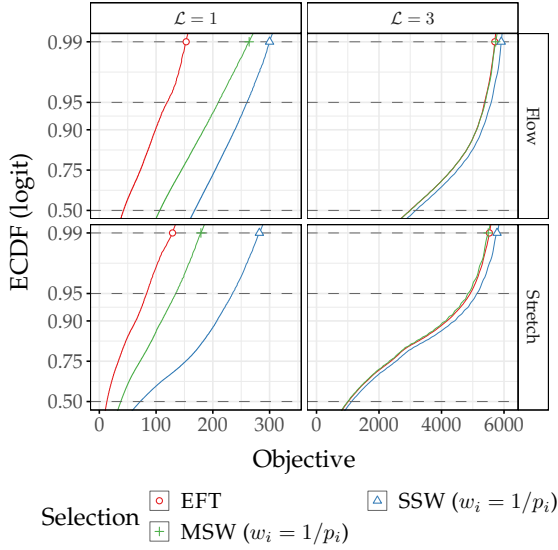
Figure 3: ECDF of flow and stretch metrics given by centralized heuristics SSW and MSW combined to a local FIFO execution in a burst over 3 seconds, under average loads of 100% and 300%.

Appendix A   Proof of Theorem 5

Proof First, we build an instance designed to reach an arbitrarily large ratio. Then, we determine a lower bound on the objective achieved with MAX-FLOW, and finally, an upper bound on the optimal one.

Instance characteristics. For an arbitrary competitive ratio $k \geq 1$, we build the following instance with n requests. The first k requests have a weight $w_i = k$ and release time $r_i = 0$. Then, a new request arrives at each new time step with a weight that is the highest integer lower than or equal to $1 + 1/k$ times the weight of the previous request (i.e., $w_i = \lfloor (1+1/k)w_{i-1} \rfloor$ and $r_i = i - k$ for $k < i \leq n$). In total, $n = k^2 + 11$ requests are submitted.

Lower bound. At time $t = 0$, MAX-FLOW starts one of the first k requests because they are the only ones that are ready. We now prove that at any time t such that $1 \leq t < k$, MAX-FLOW starts one of the remaining first k requests, which delays all arriving requests (any request T_i such that $k < i < 2k$).

On the one hand, $w_i(t + 1 - r_i) = k(t + 1)$ for any of the first k requests ($1 \leq i \leq k$). On the other hand, for $k < i \leq n$, $w_i \leq (1 + 1/k)w_{i-1}$, and thus, $w_i \leq (1 + 1/k)^{i-k}k$. Therefore, $w_i(t + 1 - r_i) \leq (1 + 1/k)^{i-k}k(t + 1 - i + k)$.

Let us show that at any time t such that $1 \leq t < k$, any of the first k requests has the highest value, that is $k(t+1) \geq (1+1/k)^{i-k}k(t+1-i+k)$ for all $k < i \leq t+k$. By changing variables ($j = i - k$ and $t' = t + 1$), this corresponds to proving $(1 + 1/k)^j(t' - j) \leq t'$ for all $1 \leq j < t' \leq k$. We show by induction that $(1 + 1/k)^j(t' - j) \leq t'$ for all $0 \leq j$ and for a given t' ($2 \leq t' \leq k$). The induction basis with $j = 0$ is direct. The induction step assumes $(1 + 1/k)^j(t' - j) \leq t'$ to be true for a given $j \geq 0$. We have

$$(1 + 1/k)\frac{t' - j - 1}{t' - j} = (1 + 1/k)(1 - \frac{1}{t' - j})$$

$$= 1 + 1/k - \frac{1}{t' - j} - \frac{1}{k(t' - j)} \leq 1.$$

The last line is obtained by remarking that $t' \leq k$ and $j \geq 0$ (thus, $1/k \leq \frac{1}{t'-j}$). Therefore,

$$(1 + 1/k)^{j+1}(t' - (j + 1))$$
$$= (1 + 1/k)^j(1 + 1/k)(t' - j)\frac{t' - j - 1}{t' - j}$$
$$\leq (1 + 1/k)^j(t' - j) \leq t',$$

which concludes the induction proof.

At time $t = k$, all of the first k requests have been completed. We now prove that at any time t such that $k \leq t < n$, MAX-FLOW starts request T_{t+1}. This would mean that at time t, only requests T_i such that $t < i \leq t + k$ are ready and not completed. We prove by induction that at time $k \leq t < n$, all requests T_i with $i \leq t$ are completed. The induction basis with $t = k$ is already proven above. Assume the hypothesis is true for a given $k \leq t < n$. It remains to prove that at time $t' = t + 1$, T_{t+2} is started among requests T_i such that $t' < i \leq t' + k$.

On the one hand, $w_i(t' + 1 - r_i) = w_{t+2}k$ for request T_{t+2}. On the other hand, for $t' + 1 < i \leq t' + k$, $w_i(t' + 1 - r_i) \leq (1 + 1/k)^{i-t'-1}w_{t+2}(t' + 1 - i + k)$. Let us show that $(1+1/k)^{i-t'-1}w_{t+2}(t'+1-i+k) < w_{t+2}k$ for $t' + 1 < i \leq t' + k$ and for a given $k \leq t < n$. By changing variables ($j = i - t' - 1$), this corresponds to proving that $(1 + 1/k)^j(k - j) < k$ for all $0 < j < k$. We show this again by induction on j for a given $k \geq 1$. For the induction basis, $(1+1/k)(k-1) = k+1-1-1/k < k$. For the induction step, we can show that $(1+1/k)\frac{k-j+1}{k-j} \leq 1$ by remarking that $k > k - j$, which concludes the induction proof.

To conclude on the performance of MAX-FLOW, request T_i is started at time $i - 1$ and therefore, the objective value is at least $w_nF_n = w_n(n - (n - k)) = kw_n$.

Upper bound. A better objective value can be obtained by starting all requests as soon as they arrive except for the first k ones: request T_1 is started at time $t = 0$; then, request T_i is started at time $t = i - k$ for $k < i \leq n$; finally, the remaining requests among
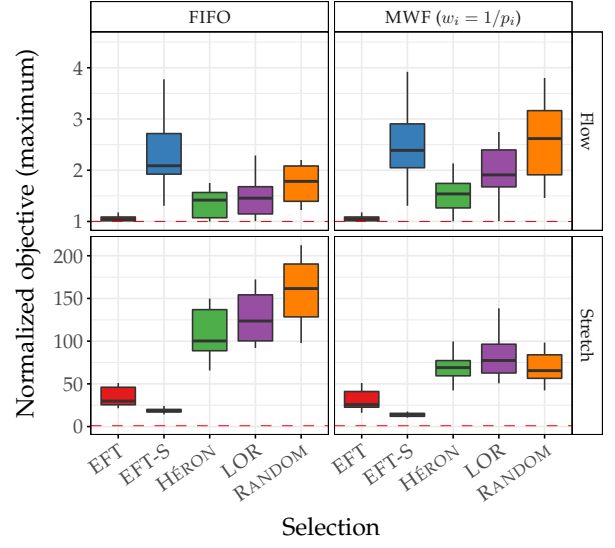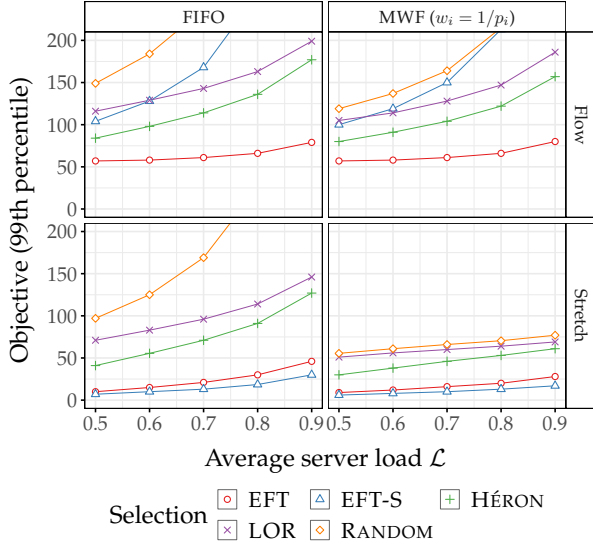
(a) 99$^{\text{th}}$ quantile of flow and stretch metrics for each combination of selection/execution heuristics in steady-state over 120 seconds, under average loads ranging from 50% to 90%.

(b) Distributions of normalized flow maximums and stretch maximums for each combination of selection/execution heuristics. Items are requested with a load of 90%, and the 10 different scenarios consist of 1200 requests.

Figure 4: Comparisons of absolute and relative performance for each combination of selection/execution heuristics.

the first k ones are started (T_i is started at time $t = n - k + i - 1$ for $1 < i \leq k$). We analyse the objective value for request T_k because it is the last one to be executed among the first k requests, and T_n because it is the one with the highest weight among the last $n - k$ requests. For T_k, $w_k F_k = k(C_k - r_k) = kn$. For T_n, $w_n F_n = w_n$.

We prove that $w_n \geq kn$ by deriving a lower bound on w_n. The weights increase in multiple stages. At first, each increment is unitary: $w_{i+1} = w_i + 1$ for $k \leq i < 2k$. Then, the increment increases at the second stage and $w_{i+1} = w_i + 2$ for $2k \leq i < 2k + \lceil k/2 \rceil$. At the k-th stage, $w_{i+1} = w_i + k$ for a single request. At a given stage j, the increment of the weight is j for at most $\lceil k/j \rceil$ requests. Let $n_1 = \sum_{j=1}^{k} \lceil k/j \rceil$ be the number of such requests (assuming $n - k \geq n_1$). Finally, the remaining $n_2 = n - k - n_1$ requests are incremented by a value that increases by at least 1 for each new request: $w_{i+1} \geq w_i + (k + i - n + n_2)$ for $n - n_2 < i \leq n$.

The last weight w_n is at least the sum of the increments of all these stages:

$$w_n \geq k + \sum_{j=1}^{k} j \lceil k/j \rceil + \sum_{j=1}^{n_2} (k + j).$$

Thus, $w_n \geq k(k+1) + kn_2 + n_2^2/2$. Our hypothesis is that $w_n \geq kn$, which would be verified if

$$k(k+1) + kn_2 + n_2^2/2 \geq kn.$$

By replacing n_2 and simplifying, the condition becomes

$$n \geq k + n_1 + \sqrt{2k(n_1 - 1)}. \tag{A1}$$

We bound n_1 using the asymptotic expansion of the harmonic number H_k:

$$
\begin{aligned}
n_1 = \sum_{j=1}^{k} \lceil k/j \rceil &< k \sum_{j=1}^{k} \frac{1}{j} + k \\
&< k(H_k + 1) \\
&< k(\log(k) + \gamma + \frac{1}{2k} + 1),
\end{aligned}
$$

where $\gamma \approx 0.577$ is the Euler-Mascheroni constant.

Thus, the optimal objective is at most w_n and the one achieved with MAX-FLOW is at least kw_n, which concludes the proof. □

Appendix B   Notations

Table B1 summarizes the notations used in this paper.

Table B1: List of the most used notations.

Symbol	Definition
i	index of requests
j	index of machines
n	number of requests
m	number of machines
k	replication factor
$T_i \in T$	request i
$M_j \in M$	machine j
$\mathcal{M}_i \subseteq M$	machines able to process T_i
w_i	weight of T_i
r_i	release time of T_i
z_i	size of the stored value for T_i
p_i	processing time of T_i
σ_i	start time of T_i
C_i	completion time of T_i
C_i^S	completion time of T_i in a schedule S
F_i	flow time of T_i
S_i	stretch of T_i
$\tilde{B}$	inverse of network bandwidth
L	network latency
λ	request submission rate
η	Weibull scale parameter
θ	Weibull shape parameter
$\mathcal{L}$	average load

Appendix C   Approximation

We provide a comprehensive summary of results related to approximation and competitive analysis of scheduling problems that address the minimization of flow time. Table C2 presents competitive analysis results that are related to average weighted flow.

Appendix D   Complexity

To complete our survey on scheduling problems related to our study, we present a summary on complexity results related to sum-flow in Table D3.

References

Ambühl, C., & Mastrolilli, M. (2005). On-line scheduling to minimize max flow time: an optimal preemptive algorithm. *Operations Research Letters*, *33*(6), 597–602.

Anand, S., Bringmann, K., Friedrich, T., Garg, N., Kumar, A. (2017). Minimizing maximum (weighted) flow-time on related and unrelated machines. *Algorithmica*, *77*(2), 515–536.

Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., Paleczny, M. (2012). Workload analysis of a large-scale key-value store. *ACM SIGMETRICS Performance Evaluation Review*, *40*(1), 53–64.

Awerbuch, B., Azar, Y., Leonardi, S., Regev, O. (2002). Minimizing the flow time without migration. *SIAM Journal on Computing*,

Table C2: Existing results on average (weighted) flow minimization. *any deterministic* (resp. *any randomized*) indicates that the approximation ratio applies to any deterministic (resp. randomized) algorithm of the corresponding type. The approximation ratio of an online algorithm should be seen as a competitive ratio.

Objective	Env.	Preemption	Algorithm	Type	Approximation/Competitive Ratio	Ref.
$\sum w_i F_i$	1	Preemptive	Chekuri, Khanna, and Zhu (2001)	Online	$O(\log^2 \Delta)$	(2001)
			any deterministic	Online	≥ 1.618	(2001)
			any randomized	Online	$\geq 4/3$	(2001)
			Bansal and Dhamdhere (2007)	Online	$O(\log(w_{\max}/w_{\min}))$	(2007)
			Bansal and Dhamdhere (2007)	Offline	$O(\log n + \log \Delta)$	(2007)
	P	Preemptive	*any randomized*	Online	$\geq \Omega(\min(\Delta^{1/2}, (w_{\max}/w_{\min})^{1/2}, (n/m)^{1/4}))$	(2001)
$\sum F_i$	1	Non-preemptive	FIFO	Online	Δ	(2008)
			any	Offline	$\geq \Omega(n^{1/2-\varepsilon})$	(1999)
			Kellerer et al. (1999)	Offline	$O(n^{1/2})$	(1999)
		Preemptive	SRPT	Online	Optimal	(1974)
			RMLF	Online	$O(\log n)$	(2004)
	P	Non-preemptive	*any*	Offline	$\geq \Omega(n^{1/3-\varepsilon})$	(2007)
			Leonardi and Raz (2007)	Offline	$O((n/m)^{1/2}\log(n/m))$	(2007)
		Non-migratory	Awerbuch, Azar, Leonardi, and Regev (2002)	Online	$O(\min(\log \Delta, \log n))$	(2002)
		Preemptive	SRPT	Online	$O(\min(\log \Delta, \log(n/m)))$	(2007)
			any randomized	Online	$\geq \Omega(\log(n/m + \Delta))$	(2007)
			RMLF	Online	$O(\log n \min(\log \Delta, \log(n/m)))$	(2004)
	$P\vert M_i$	Non-migratory	Garg and Kumar (2007)	Offline	$O(\log \Delta)$	(2007)
			any	Offline	∞ (no bounded ratio)	(2007)
$\sum S_i$	1	Non-preemptive	FIFO	Online	Δ^2	(2008)
		Preemptive	SRPT	Online	2	(1999)
	P	Non-migratory	Chekuri et al. (2001)	Online	17.32	(2001)
		Preemptive	SRPT	Online	14	(1999)
			Chekuri et al. (2001)	Online	9.82	(2001)
$\sum F_i^p$	1	Preemptive	*any*	Online	$\geq n^{o(1)}$	(2003)
$\sum S_i^p$	1	Preemptive	*any*	Online	$\geq n^{o(1)}$	(2003)

Table D3: Complexity of sum-flow minimization problems. P, Q and R denote parallel machines, related machines, and unrelated machines, respectively. Arrows are reduction relationships ($A \to B$ means that A is a special case of B). Unknown problem complexity is denoted by **?**. A sign **+** (resp. **-**) means that the problem is NP-hard (resp. polynomially solvable) via the reduction relationship. Incompatible problem designations are noted $\emptyset$.

Objective	Constraints	1	$\longrightarrow$ P	$\longrightarrow$ Q	$\longrightarrow$ R
$\sum w_i F_i$	$\mathcal{M}_i$	?	+	+	+
	$\circ$	?	s. NP-hard (1974)	+	+
	$r_i, \mathcal{M}_i, p_i = 1$	-	p. solvable (1997)	?	?
	$r_i, p_i = p$	-	p. solvable (2008)	?	$\emptyset$
	$r_i, pmtn$	s. NP-hard (1984)	+	+	+
$\sum F_i$	r_i	s. NP-hard (1977)	+	+	+
	$\mathcal{M}_i$	-	-	-	p. solvable (1974)
	$\circ$	-	-	-	-
	$r_i, \mathcal{M}_i, p_i = 1$	-	-	?	?
	$r_i, p_i = p$	-	p. solvable (1983)	?	$\emptyset$
	$r_i, pmtn$	p. solvable (1974)	s. NP-hard (2007)	+	+
	$\mathcal{M}_i, pmtn$	-	p. solvable (1997)	?	s. NP-hard (2001)
	$r_i, p_i = p, pmtn$	-	-	p. solvable (2009)	?
$\sum S_i$	r_i	NP-hard (2008)	+	+	+
$\sum F_i^p$	$r_i, pmtn$	s. NP-hard (2013)	+	+	+
$\sum S_i^p$	$r_i, pmtn$	s. NP-hard (2013)	+	+	+

$31(5)$, 1370–1382.

Baker, K.R. (1974). *Introduction to sequencing and scheduling.* John Wiley & Sons.

Balmau, O., Dinu, F., Zwaenepoel, W., Gupta, K., Chandhiramoorthi, R., Didona, D. (2020). Silk+ preventing latency spikes in log-structured merge key-value stores running heterogeneous workloads. *ACM Transactions on Computer Systems*, $36(4)$, 1–27.

Bansal, N. (2005). Minimizing flow time on a constant number of machines with preemption. *Operations Research Letters*, $33(3)$, 267–273.

Bansal, N., & Cloostermans, B. (2016). Minimizing maximum flow-time on related machines. *Theory of Computing*, $12(1)$, 1–14.

Bansal, N., & Dhamdhere, K. (2007). Minimizing weighted flow time. *ACM Transactions on*

Algorithms, $3(4)$, 39.

Bansal, N., & Kulkarni, J. (2015). Minimizing flow-time on unrelated machines. *Proceedings of the forty-seventh annual acm symposium on theory of computing* (pp. 851–860).

Bansal, N., & Pruhs, K. (2003). Server scheduling in the l_p norm: a rising tide lifts all boat. *Proceedings of the thirty-fifth annual acm symposium on theory of computing* (pp. 242–250).

Baptiste, P., Brucker, P., Chrobak, M., Dürr, C., Kravchenko, S.A., Sourd, F. (2007). The complexity of mean flow time scheduling problems with release times. *Journal of Scheduling*, $10(2)$, 139–146.

Becchetti, L., & Leonardi, S. (2004). Non-clairvoyant scheduling to minimize the total flow time on single and parallel machines. *Journal of the ACM*, $51(4)$, 517–539.

Bender, M.A., Chakrabarti, S., Muthukrishnan, S. (1998). Flow and stretch metrics for scheduling continuous job streams. *ACM-SIAM symposium on discrete algorithms* (pp. 270–279).

Ben Mokhtar, S., Canon, L.-C., Dugois, A., Marchal, L., Rivière, E. (2021). Taming tail latency in key-value stores: a scheduling perspective. *European conference on parallel processing* (pp. 136–150).

Benoit, A., Elghazi, R., Robert, Y. (2021). Max-stretch minimization on an edge-cloud platform. *2021 ieee international parallel and distributed processing symposium* (pp. 766–775).

Brucker, P., Jurisch, B., Krämer, A. (1997). Complexity of scheduling problems with multi-purpose machines. *Annals of Operations Research*, *70*, 57–73.

Brucker, P., & Kravchenko, S.A. (2008). Scheduling jobs with equal processing times and time windows on identical parallel machines. *Journal of Scheduling*, *11*(4), 229–237.

Bruno, J., Coffman Jr, E.G., Sethi, R. (1974). Scheduling independent tasks to reduce mean finishing time. *Communications of the ACM*, *17*(7), 382–387.

Brutlag, J. (2009). *Speed matters for google web search.*

Carlson, J.L. (2013). *Redis in action.* Manning Publications Co.

Chekuri, C., Khanna, S., Zhu, A. (2001). Algorithms for minimizing weighted flow time. *Proceedings of the thirty-third annual acm symposium on theory of computing* (pp. 84–93).

Chodorow, K. (2013). *Mongodb - the definitive guide: Powerful and scalable data storage.* O'Reilly.

Choudhury, A.R., Das, S., Garg, N., Kumar, A. (2018). Rejecting jobs to minimize load and maximum flow-time. *Journal of Computer and System Sciences*, *91*, 42–68.

Dean, J., & Barroso, L.A. (2013). The tail at scale. *Communications of the ACM*, *56*(2), 74–80.

DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., ... Vogels, W. (2007). Dynamo: amazon's highly available key-value store. *ACM SIGOPS operating systems review*, *41*(6), 205–220.

Delgado, P., Didona, D., Dinu, F., Zwaenepoel, W. (2016). Job-aware scheduling in eagle: Divide and stick to your probes. *Proceedings of the seventh acm symposium on cloud computing* (pp. 497–509).

Delgado, P., Dinu, F., Kermarrec, A.-M., Zwaenepoel, W. (2015). Hawk: Hybrid datacenter scheduling. *2015 USENIX annual technical conference* (pp. 499–510).

Didona, D., & Zwaenepoel, W. (2019). Size-aware sharding for improving tail latencies in in-memory key-value stores. *16th USENIX symposium on networked systems design and implementation* (pp. 79–94).

Dutot, P.-F., Saule, E., Srivastav, A., Trystram, D. (2016). Online non-preemptive scheduling to optimize max stretch on a single machine. *Computing and combinatorics - 22nd international conference* (Vol. 9797, pp. 483–495).

Feitelson, D.G. (2015). *Workload modeling for computer systems performance evaluation.* Cambridge University Press.

Garg, N., & Kumar, A. (2007). Minimizing average flow-time: Upper and lower bounds. *48th annual ieee symposium on foundations of computer science (focs'07)* (pp. 603–613).

Hall, L.A. (1993). A note on generalizing the maximum lateness criterion for scheduling. *Discrete applied mathematics*, *47*(2), 129–137.

Jaiman, V., Ben Mokhtar, S., Quéma, V., Chen, L.Y., Rivière, E. (2018). Héron: Taming tail latencies in key-value stores under heterogeneous workloads. *37th symposium on reliable distributed systems* (pp. 191–200).

Jaiman, V., Mokhtar, S.B., Rivière, E. (2020). TailX: Scheduling heterogeneous multiget queries to improve tail latencies in key-value stores. *Ifip international conference on distributed applications and interoperable systems* (pp. 73–92).

Jiang, W., Xie, H., Zhou, X., Fang, L., Wang, J. (2019). Haste makes waste: The on–off algorithm for replica selection in key–value stores. *Journal of Parallel and Distributed Computing*, *130*, 80–90.

Jose, J., Subramoni, H., Luo, M., Zhang, M., Huang, J., Wasi-ur Rahman, M., . . . others (2011). Memcached design on high performance rdma capable interconnects. *International conference on parallel processing* (pp. 743–752).

Kalyanasundaram, B., & Pruhs, K. (2000). Speed is as powerful as clairvoyance. *Journal of the ACM*, *47*(4), 617–643.

Kellerer, H., Tautenhahn, T., Woeginger, G. (1999). Approximability and nonapproximability results for minimizing total flow time on a single machine. *SIAM Journal on Computing*, *28*(4), 1155–1166.

Kravchenko, S.A., & Werner, F. (2009). Preemptive scheduling on uniform machines to minimize mean flow time. *Computers & Operations Research*, *36*(10), 2816–2821.

Labetoulle, J., Lawler, E.L., Lenstra, J.K., Kan, A.R. (1984). Preemptive scheduling of uniform machines subject to release dates. *Progress in combinatorial optimization* (pp. 245–261).

Lakshman, A., & Malik, P. (2010). Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, *44*(2), 35–40.

Lawler, E.L., & Labetoulle, J. (1978). On preemptive scheduling of unrelated parallel processors by linear programming. *Journal of the ACM*, *25*(4), 612–619.

Lee, K., Leung, J.Y., Pinedo, M.L. (2013). Makespan minimization in online scheduling with machine eligibility. *Annals of Operations Research*, *204*(1), 189–222.

Legrand, A., Su, A., Vivien, F. (2008). Minimizing the stretch when scheduling flows of divisible requests. *Journal of Scheduling*, *11*(5), 381–404.

Lenstra, J.K., Kan, A.R., Brucker, P. (1977). Complexity of machine scheduling problems. *Studies in integer programming*, *1*, 343–362.

Leonardi, S., & Raz, D. (2007). Approximating total flow time on parallel machines. *Journal of Computer and System Sciences*, *73*(6), 875–891.

Leung, J.Y.-T., & Li, C.-L. (2008). Scheduling with processing set restrictions: A survey. *International Journal of Production Economics*, *116*(2), 251–262.

Leung, J.Y.-T., & Li, C.-L. (2016). Scheduling with processing set restrictions: A literature update. *International Journal of Production Economics*, *175*, 1–11.

Li, J., Sharma, N.K., Ports, D.R., Gribble, S.D. (2014). Tales of the tail: Hardware, OS, and application-level sources of tail latency. *Acm symposium on cloud computing* (pp. 1–14).

Lucarelli, G., Moseley, B., Thang, N.K., Srivastav, A., Trystram, D. (2019). Online non-preemptive scheduling to minimize maximum weighted flow-time on related machines. *39th IARCS annual conference on foundations of software technology and theoretical computer science* (Vol. 150, pp. 24:1–24:12).

Maheswaran, M., Ali, S., Siegel, H.J., Hensgen, D., Freund, R.F. (1999). Dynamic mapping of a class of independent tasks onto heterogeneous computing systems. *Journal of parallel and distributed computing*, *59*(2), 107–131.

Mastrolilli, M. (2004). Scheduling to minimize max flow time: Off-line and on-line algorithms. *International Journal of Foundations of Computer Science*, *15*(02), 385–401.

Moseley, B., Pruhs, K., Stein, C. (2013). The complexity of scheduling for p-norms of flow and stretch. *International conference on integer programming and combinatorial optimization* (pp. 278–289).

Muthukrishnan, S., Rajaraman, R., Shaheen, A., Gehrke, J.E. (1999). Online scheduling to minimize average stretch. *40th annual symposium on foundations of computer science* (pp. 433–443).

Reda, W., Canini, M., Suresh, L., Kostić, D., Braithwaite, S. (2017). Rein: Taming tail latency in key-value stores via multiget scheduling. *12th european conference on computer systems* (pp. 95–110).

Saule, E., Bozdağ, D., Çatalyürek, Ü.V. (2012). Optimizing the stretch of independent tasks on a cluster: From sequential tasks to moldable tasks. *Journal of Parallel and Distributed Computing*, *72*(4), 489–503.

Simons, B. (1983). Multiprocessor scheduling of unit-time jobs with arbitrary release times and deadlines. *SIAM Journal on Computing*, *12*(2), 294–299.

Sitters, R. (2001). Two np-hardness results for preemptive minsum scheduling of unrelated parallel machines. *International conference on integer programming and combinatorial optimization* (pp. 396–405).

Suresh, L., Canini, M., Schmid, S., Feldmann, A. (2015). C3: Cutting tail latency in cloud data stores via adaptive replica selection. *12th USENIX symposium on networked systems design and implementation* (pp. 513–527).

Vulimiri, A., Godfrey, P.B., Mittal, R., Sherry, J., Ratnasamy, S., Shenker, S. (2013). Low latency via redundancy. *9th acm conference on emerging networking experiments and technologies* (pp. 283–294).

Wu, Z., Yu, C., Madhyastha, H.V. (2015). Costlo: Cost-effective redundancy for lower latency variance on cloud storage services. *12th USENIX symposium on networked systems design and implementation* (pp. 543–557).