Taming Tail Latency in Key-Value Stores: a Scheduling Perspective (extended version)

Sonia Ben Mokhtar, Louis-Claude Canon, Anthony Dugois, Loris Marchal, Etienne Rivière

▶ **To cite this version:**

Sonia Ben Mokhtar, Louis-Claude Canon, Anthony Dugois, Loris Marchal, Etienne Rivière. Taming Tail Latency in Key-Value Stores: a Scheduling Perspective (extended version). 2021. hal-03144818v2

HAL Id: hal-03144818
https://inria.hal.science/hal-03144818v2

Preprint submitted on 12 Mar 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Taming Tail Latency in Key-Value Stores: a Scheduling Perspective (extended version)

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 % [19].

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 [3, 25].

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 [26, 42]. 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 [34, 36, 58] 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 6);
- the comparison of these heuristics in extensive simulations (Section 7).

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

2 Related Work

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

2.1 Dealing with Tail-Latency in Key-Value Stores

The principles of key-value stores were first documented by Amazon with Dynamo [26]. Cassandra [42] is a widely-used open-source key-value store, following principles similar to that of Dynamo; other popular key-value stores include Redis [20], memcached [37], and document stores such as MongoDB [23].

Key-value stores implement data partitioning for horizontal scalability. Typically, data is spread over a cluster of servers using consistent hashing. This consists of treating the output of a hashing 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) [26, 42]. Replication is implemented on top of data partitioning, by duplicating each data item on the successors of its assigned server. Fig. 1 shows an example of a cluster with a replication factor of 3: value whose keys' hash values fall in the range of M_2 can also be requested from servers M_3 and M_4.

Design and implementation improvements have been proposed to improve response time and in particular reduce tail-latency in key-value stores. They include the use of redundant requests [59, 60], performing smart resource allocation [29], or employing hybrid scheduling [27, 28]. We are interested in this paper in replica selection strategies [34, 36, 58]. 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.

[5] https://doi.org/10.6084/m9.figshare.13114196

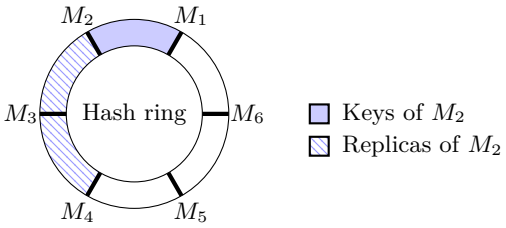

Fig. 1. Data partitioning and replication on a cluster of 6 servers.

Coordinators can, therefore, associate a key with a list of replicas and select the most appropriate server to query. Cassandra uses Dynamic Snitching [42], 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 [58] 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 [3]. 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 do not have, however, directly access to the size of data items, and only know the requested key. Héron [34] addresses 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 [29] avoids head-of-line blocking on a specific server, by specializing some of its cores to serve only 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 [6]. Other systems, such as REIN [53] or TailX [35], 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.* [48] 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.

2.2 Latency Minimization in Scheduling Theory

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 ($\sum F_i$).

Minimizing maximum (weighted) flow. It is well-known that the maximum flow criterion is minimized by the FIFO (First In First Out) strategy on a single-machine [14]. This scheme is also $(3 - \frac{2}{m})$-competitive on m machines when preemption is allowed or not [14, 51], and Ambühl *et al.* gave a $(2 - \frac{1}{m})$-competitive algorithm for the preemptive case, which is optimal [1]. 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 [14]. For single-machine problems, they proved that no polynomial-time algorithm can approximate the offline non-preemptive problem of optimizing the max-stretch criterion within a factor $\Omega(n^{1-\varepsilon})$, for any $\varepsilon > 0$, unless $P = NP$. They also exhibit an FPTAS for the preemptive case, and they 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 [44]. 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 *et al.* improved this result by showing a lower bound of $\frac{1}{2}(1+\Delta)$ on a single server and $\frac{1}{2}(1+\frac{\Delta}{m+1})$ on m servers [54]. Finally, Dutot *et al.* closed the online problem of minimizing the stretch on a single machine by proving a lower bound of $\frac{1}{2}\Delta(\sqrt{5}-1)$, which is tight [30]. Some additional works deal with the minimization of maximum weighted flow time under different assumptions and models [2, 8, 49].

Table 4 provides a more exhaustive summary of results on max-flow minimization.

Minimizing sum (weighted) flow. Optimizing the average performance is obtained through minimizing the sum-flow criterion. It is well known that this problem is hard without preemption, while the preemptive version can be polynomially solved by the SRPT (Shortest Remaining Processing Time) strategy [5]. Kellerer *et al.* gave an $O(\sqrt{n})$-approximation algorithm for the non-preemptive case, and proved that no polynomial-time algorithm can approximate the offline non-preemptive problem of minimizing the sum-flow within a factor $\Omega(n^{\frac{1}{2}-\varepsilon})$, for any $\varepsilon > 0$, unless $P = NP$ [39]. On m servers, Leonardi *et al.* analyzed SRPT and showed a competitive ratio of $O(\min(\log \Delta, \log \frac{n}{m}))$ [46]. They provide a lower bound of $\Omega(n^{\frac{1}{3}-\varepsilon})$ on the approximability of this problem, unless $P = NP$. For the weighted sum-flow, SRPT has been proven 2-competitive on a single server, and 14-competitive on parallel machines [52]. Chekuri *et al.* improved this last competitiveness result by providing a 9.82-competitive algorithm on m machines, and a 17.32-competitive algorithm for non-migratory version of the problem [22]. They also present a $(2+\varepsilon)$-approximation for the offline optimization of weighted sum-flow on one machine. Note that one must be careful when using the (weighted) sum-flow criterion, as it may lead to starvation: some requests may be infinitely delayed in an optimal solution [14]. This is one reason why we discard this criterion in our study.

For a more detailed survey about sum-flow optimization, see Table 5.

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. In scheduling literature, this constraint is known as "multipurpose machines", "processing set restrictions" or even "eligibility constraints". Brucker *et al.* proposed a formalization and analyzed the complexity of some of these problems [16], by introducing the $\mathcal{M}_i$ term in the common $\alpha|\beta|\gamma$ notation. They also used a routine based on solving the minimum cost matching problem to polynomially solve $Q|\mathcal{M}_i, p_i = 1|\sum w_i U_i$ and $P|\mathcal{M}_i, r_i, p_i = 1|\sum w_i U_i$. Another notable result is that $P|\mathcal{M}_i, pmtn|\sum C_i$ can be solved in polynomial time by transforming any preemptive schedule in a non-preemptive one without worsening the objective. A more recent survey is also proposed by Leung *et al.* [47], and Shabtay *et al.* treated a similar problem where there are two job types that can be processed by a specific subset of machines [55].

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.

3.1 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. 2 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 $|\mathcal{M}_i| = k$.

3.2 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 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

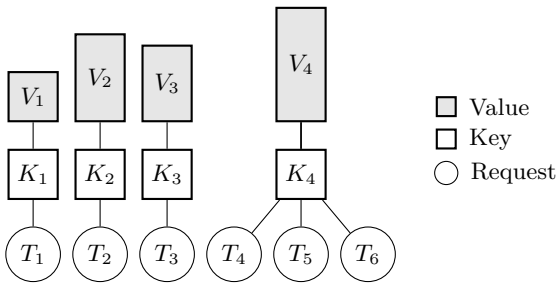

Fig. 2. A bipartite graph showing relations between requests, keys and values. Different requests may hold the same key (e.g. K_4).

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 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 ($w_i = \frac{1}{p_i}$) gives the same importance to each request, but this favors requests for small values because they are more sensitive to scheduling decisions. Third, the inverse of the square root of the processing time ($w_i = \frac{1}{\sqrt{p_i}}$) 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:

- 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 [11], 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.

4.1 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 (Algorithm 1), which schedules requests by non-increasing order of weights w_i, to solve this scheduling problem.

Theorem 1. Single-Simple *(Algorithm 1) solves* $1||\max w_i C_i$ *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 $w_j \leq w_k$.

Algorithm 1 Single-Simple

Input: w_i
Output: a schedule
1: schedule requests by non-increasing order of w_i
2: **return** schedule

Then, their contribution to the objective is $\mathcal{C} = \max(w_j C_j, w_k(C_j + p_k)) = w_k(C_j + p_k)$ because $w_j C_j \leq w_j(C_j + p_k) \leq w_k(C_j + p_k)$. If we swap the requests, then the contribution becomes $\mathcal{C}' = \max(w_k C'_k, w_j(C'_k + p_j))$ 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 $w_k C'_k \leq w_k(C'_k + p_j)$, $w_j(C'_k + p_j) \leq w_k(C'_k + p_j)$ and $w_k(C'_k + p_j) = w_k(C_j + p_k) = \mathcal{C}$. Therefore, $\max(w_k C'_k, w_j(C'_k + p_j)) = \mathcal{C}' \leq \mathcal{C}$.

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.

Theorem 2. Single-Simple *(Algorithm 1) solves* $P|p_i = p| \max w_i C_i$ *in polynomial time.*

Proof. We use the same kind of argument than for the single-machine case. Let OPT be an optimal schedule. If all requests are ordered by non-increasing weight, then Single-Simple is optimal. If they are not, let T_j and T_k be two requests in OPT such that $w_j \leq w_k$. By definition, $p_j = p_k = p$. Without loss of generality, we consider that there is no idle time intervals in OPT, and T_j starts at time t, whereas T_k starts at time $t + cp$ with $c \geq 0$ (on any server).

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

It follows that we can transform OPT in another optimal schedule by switching repeatedly non-sorted requests. Then, Single-Simple is optimal. □

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 [45]. We prove that Single-Simple is an approximation algorithm.

Theorem 3. Single-Simple *(Algorithm 1) computes a* $(2-\frac{1}{m})$*-approximation for the problem* $P|| \max w_i C_i$*, 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 $w_j C_j = \max w_i C_i^S$, 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 $\max w_i C_i^S$ in S and can only decrease $\max w_i C_i^{OPT}$ 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 $C^S_{\max} \leq \left(2 - \frac{1}{m}\right) \cdot C^*_{\max}$ [33], 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,

$$\begin{aligned}\max w_i C_i^S = w_j C_j^S = w_j C^S_{\max} \leq \left(2 - \frac{1}{m}\right) \cdot w_j C^*_{\max} &\leq \left(2 - \frac{1}{m}\right) \cdot w_j C_k^{OPT} \\ &\leq \left(2 - \frac{1}{m}\right) \cdot \frac{w_j}{w_k} \cdot w_k C_k^{OPT} \leq \left(2 - \frac{1}{m}\right) \cdot \frac{w_j}{w_k} \cdot \max w_i C_i^{OPT}.\end{aligned}$$

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 $W \to \infty$. □

4.2 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 [44]. 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 $R|| \max w_i F_i$ [47]). 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 consists in a reduction from the NP-complete problem $P||C_{\max}$ [45].

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.*

Proof. We prove the NP-completeness of this problem by reduction from $P||C_{\max}$, which is NP-complete [45]. Obviously, NonMigratory-Dec(T, M, B) belongs to NP.

Building instance. We consider an instance I_1 of the decision problem associated to $P||C_{\max}$: given a set of requests T', a set of servers M' and a bound B', is it possible to schedule each request T'_i non-preemptively on M' such that $C_{\max} = B'$? We construct the following instance I_2 of NonMigratory-Dec(T, M, B) from I_1. We first set $M = M'$ and $B = B'$. For each request T'_i we define a request T_i to which we associate a release time $r_i = 0$ and a weight $w_i = 1$; hence, $d_i = B$. I_2 can clearly be constructed in polynomial time in the size of I_1.

Equivalence of problems. A solution to I_1 trivially constitutes a non-preemptive (thus non-migratory) solution to I_2.

Assume now that I_2 has a solution S. 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 $\max_{T_i \in A_j} C_i^S$ is the makespan of M_j in S. For each request T_i, we denote the associated set of n_i processing intervals by $\Lambda_i = \{(\sigma_{i,k}, \delta_{i,k}) \mid 1 \leq k \leq n_i\}$, where $\sigma_{i,k}$ is the start time of the k-th processing interval of T_i and $\delta_{i,k}$ is its duration. For all i, k, $\sigma_{i,k} + \delta_{i,k} \leq \sigma_{i,k+1}$, and for all j,

$$\begin{aligned}\max_{T_i \in A_j} C_i^S &= \sum_{T_i \in A_j} \sum_{k=1}^{n_i} \delta_{i,k} \\ &\leq C_{\max}^S = B.\end{aligned}$$

We can build a solution S' to I_1 by removing preemptions from S, i.e., for each request T_i, we rearrange its n_i intervals (without migrating them) such that for all k, $\sigma_{i,k} + \delta_{i,k} = \sigma_{i,k+1}$. As we only switch processing intervals of requests, it does not change the makespan values of servers, therefore,

$$\begin{aligned}\max_{T_i \in A_j} C_i^{S'} &= \sum_{T_i \in A_j} \sum_{k=1}^{n_i} \delta_{i,k} \\ &= \max_{T_i \in A_j} C_i^S \\ &\leq C_{\max}^S = B = B'.\end{aligned}$$

□

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

4.3 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.

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

Proof. Let us prove this by building an example instance. We consider 10 requests $T_{1 \leq i \leq 10}$ with the following weights and release times:

- $w_i = 3$, $r_i = i - 1$ for all $1 \leq i \leq 8$;
- $w_9 = 1$, $r_9 = 0$;
- $w_{10} = 1$, $r_{10} = 0$.

We can construct an optimal schedule OPT by starting T_9 and T_{10} before T_8, which gives an objective of $\max w_i F_i^{OPT} = 9$. Now let T_{11} be a request released at $r_{11} = 9$ with a weight $w_{11} = 11$. The new optimal schedule OPT' has to schedule T_8 before T_{11}, i.e., T_9 or T_{10} will start last, to obtain an objective of $\max w_i F_i^{OPT'} = 11$. OPT' cannot be built from OPT; adding a new request may lead to reconsider our previous scheduling choices. Hence, the competitive ratio is always strictly greater than 1. □

We now present an adaptation of Single-Simple to the online case, restricted to unit tasks, Single-Unit (Algorithm 2): at each time step, 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.

Algorithm 2 Single-Unit

Input: w_i, r_i
Output: a schedule
1: **for** each time t with at least one available request **do**
2: execute request with highest $w_i(t + 1 - r_i)$
3: **end for**
4: **return** schedule

Unfortunately, even on unit size tasks, this strategy does not lead to an approximation algorithm, as outlined by the following theorem.

Theorem 6. *The competitive ratio of* Single-Unit *(Algorithm 2) is arbitrarily large for the scheduling problem* $1|\text{online-}r_i, p_i = 1|\max w_i F_i$.

Proof. First, we build an instance designed to reach an arbitrary large ratio. Then, we determine a lower bound on the objective achieved with Single-Unit 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 processing cost of $w_i = k$ and release time $r_i = 0$. Then, a new request arrives at each new time step with a processing cost that is the highest integer lower than or equal to $1 + \frac{1}{k}$ times the previous submitted request (i.e., $w_i = \lfloor (1 + \frac{1}{k}) w_{i-1} \rfloor$ and $r_i = i - k$ for $k < i \leq n$). In total, $n = k^2 + 11$ requests are submitted.

SINGLE-UNIT *lower bound.* At time $t = 0$, SINGLE-UNIT 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 $1 \leq t < k$, SINGLE-UNIT starts one of the remaining first k requests, which delays all arriving requests (any request T_i such that $k < i < 2k$). On 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 $k < i \leq n$, $w_i \leq (1+\frac{1}{k})w_{i-1}$ and thus, $w_i \leq (1+\frac{1}{k})^{i-k}k$. Therefore, $w_i(t+1-r_i) \leq (1+\frac{1}{k})^{i-k}k(t+1-i+k)$. It remains to prove that at any time t such that $1 \leq t < k$, any of the first requests has the highest value, that is $k(t + 1) \geq (1 + \frac{1}{k})^{i-k}k(t + 1 - i + k)$ for all $k < i \leq t + k$. By changing variables ($j = i - k$ and $t' = t + 1$), this corresponds to proving $(1 + \frac{1}{k})^j(t' - j) \leq t'$ for all $1 \leq j < t' \leq k$. We show by induction that $(1 + \frac{1}{k})^j(t' - j) \leq t'$ for all $0 \leq j$ and for a given $t' \geq 2$ and $k \geq t'$. The induction basis with $j = 0$ is direct ($t' \leq t'$). The induction step assumes $(1 + \frac{1}{k})^j(t' - j) \leq t'$ to be true for a given $j \geq 0$.

$$\begin{aligned}(1 + \frac{1}{k})\frac{t' - j - 1}{t' - j} &= (1 + \frac{1}{k})(1 - \frac{1}{t' - j}) \\ &= 1 + \frac{1}{k} - \frac{1}{t' - j} - \frac{1}{k(t' - j)} \leq 1\end{aligned}$$

The last line is obtained by remarking that $t' \leq k$ and $j \geq 0$. Thus, $\frac{1}{k} \leq \frac{1}{t'-j}$. Therefore, $(1 + \frac{1}{k})^{j+1}(t' - (j + 1)) = (1 + \frac{1}{k})^j(t' - j)(1 + \frac{1}{k})^{\frac{t'-j-1}{t'-j}} \leq t'$, 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 $k \leq t < n$, SINGLE-UNIT 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 proved 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 one hand, $w_i(t' + 1 - r_i) = w_{t+2}k$ for request T_{t+2}. On the other hand, for $t' + 1 < i \leq t' + k$, $w_i(t' + 1 - r_i) \leq (1 + \frac{1}{k})^{i-t'-1}w_{t+2}(t' + 1 - i + k)$. It remains to prove that $(1 + \frac{1}{k})^{i-t'-1}w_{t+2}(t' + 1 - i + k) < w_{t+2}k$ 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 $(1 + \frac{1}{k})^j(k - j) < k$ for all $0 < j < k$. We show this again by induction on j for a given $k \geq 1$. For the induction basis, $(1 + \frac{1}{k})(k - 1) = k + 1 - 1 - \frac{1}{k} < k$. For the induction step, we can show that $(1 + \frac{1}{k})\frac{k-j+1}{k-j} \leq 1$ by remarking that $k > k - j$, which concludes the induction proof.

To conclude on the performance of SINGLE-UNIT, request T_i is started at time $t = i - 1$ and therefore, the objective value is at least $w_nF_n = w_n(C_i - r_i) = kw_n$.

Optimal 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 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 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, $w_kF_k = k(C_k - r_k) = kn$. For T_n, $w_nF_n = w_n$.

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 kth 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

$$\begin{aligned} k(k+1) + kn_2 + n_2^2/2 &\geq kn \\ k(k+1) + k(n-k-n_1) + (n-k-n_1)^2/2 &\geq kn \\ 2k(1-n_1) + (n-k-n_1)^2 &\geq 0 \\ n-k-n_1 &\geq \sqrt{2k(n_1-1)} \\ n &\geq k + n_1 + \sqrt{2k(n_1-1)}. \end{aligned}$$

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

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

where $\gamma \approx 0.577$ is the Euler-Mascheroni constant. What remains to be proved is

$$n \geq k(\log(k) + \gamma + \frac{1}{2k} + 2 + \sqrt{2(\log(k) + \gamma + \frac{1}{2k} + 1)}.$$

Numerical analysis shows this occurs when $n = k^2 + 11$, which proves that $w_n \geq kn$.

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

5 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. One way to do this is to compute an optimal solution to the offline problem (or a relaxed version) and compare it to a solution generated by an online heuristic.

5.1 Maximum Weighted Flow with Machines Restrictions

We propose an ILP (Integer Linear Program) to solve the scheduling problem $P|\mathcal{M}_i, r_i, p_i = 1|\max w_i F_i$, which we adapt to provide a lower bound to the case with arbitrary processing times p_i. We denote allocations of requests to servers by $x_{i,j,t}$: $x_{i,j,t} = 1$ if request T_i is allocated to server M_j at time t, and 0 otherwise. $x_{i,j,t}$ is defined for each request $T_i \in T$, for each server $M_j \in \mathcal{M}_i$, for all times $r_i \leq t \leq \tau$ where $\tau = \max_i r_i + n$, which yields at most $nk\tau$ variables. Then solving the Mixed Integer Linear Program (1) provides a solution to the scheduling problem:

- we want to minimize the maximum weighted flow time (Equation (1a));
- each request must be scheduled (Equation (1b), n constraints);
- each request starts after its release time (Equation (1c), n constraints);
- there is no simultaneous execution on the same machine (Equation (1d), $k\tau$ constraints).

$$\textbf{minimize} \quad \max_i w_i \left[\left(\sum_{j,t} t \cdot x_{i,j,t} \right) + p_i - r_i \right] \tag{1a}$$

$$\textbf{subject to} \quad \forall i, \sum_{j,t} x_{i,j,t} = 1 \tag{1b}$$

$$\forall i, \sum_{j,t} t \cdot x_{i,j,t} \geq r_i \tag{1c}$$

$$\forall j, \forall t, \sum_{i} x_{i,j,t} \leq 1 \tag{1d}$$

$$x_{i,j,t} \in \{0,1\} \tag{1e}$$

In order to constitute a lower bound for the problem with arbitrary processing times, we transform each non-unitary request and we adapt the objective. Each request T_i arriving at time r_i with a processing time p_i is replaced with p_i requests arriving at times $r_i, r_i + 1, \ldots, r_i + p_i - 1$. The new objective function to minimize is $\max w_{i'}(C_i - r_{i'})$ where i' is the index of the original non-unitary request.

Even if this ILP theoretically constitutes a lower bound to the considered problem, its resolution is unfortunately too difficult in practice for non-trivial instances that would be representative ($n > 1000$ requests). A more practical solution would require to reduce the number of variables and constraints, or relax the model to a pure Linear Program. Alternatively, we tackle these performance issues by focusing on another related problem that is polynomially solvable and provides a lower bound to our scheduling problem.

5.2 Maximum Weighted Flow with Preemption

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 [44], followed by the reconstruction scheme from Lawler *et al.* [43]. In order to facilitate the work for the reader, we explain the complete procedure here (refer to the mentioned articles for details about the correctness).

The algorithm of Legrand *et al.* relies on the fact that this problem is equivalent to the deadline scheduling problem: we want to find the minimum objective f such that, when we fix a deadline $d_i(f) = r_i + \frac{f}{w_i}$ to each request T_i, it is possible to find a feasible schedule where all requests are executed during their feasible interval $[r_i, d_i(f)]$.

We define the ordered set of epochal times $E^f = \{r_1, \ldots, r_n, d_1(f), \ldots, d_n(f)\}$, for any value f. Each epochal time E^f_t has a position t in the set, and let $\mathcal{T}_r(i)$ (resp. $\mathcal{T}_d(i)$) give the position of the value r_i (resp. $d_i(f)$) in E^f. Adjacent epochal times constitute time intervals $[E^f_t, E^f_{t+1}]$ (of course, for $t = 2n$, the considered interval is $[E^f_t, +\infty[$).

It is critical to observe that the relative ordering of epochal times only changes for specific values f, i.e., there is an ordered set $\{\mathcal{F}_k\} \in 2^{\mathbb{Q}}$ such that, for all k and for any $f, g \in]\mathcal{F}_k, \mathcal{F}_{k+1}[$, $f < g$, the relative ordering of E^f is the same as the relative ordering of E^g. Each $\mathcal{F}_k$ is called a milestone, and corresponds to a value f for which one deadline is equal to the release time or the deadline of another request.

Computing milestones. We first need to get the set of milestones, i.e., all values f for which the relative ordering of epochal times E^f changes. This happens when the deadline of a request T_i coincides with the release time or the deadline of a request T_j, with $i \neq j$. There are two cases to consider:

- $d_i(f) = r_j \implies r_i + \frac{f}{w_i} = r_j \implies f = w_i(r_j - r_i)$;
- $d_i(f) = d_j(f) \implies r_i + \frac{f}{w_i} = r_j + \frac{f}{w_j} \implies f = \frac{w_i w_j}{w_j - w_i}(r_j - r_i)$, where $w_i \neq w_j$[7].

[7] With different release times, two deadlines will never coincide if $w_i = w_j$.

Solving in a milestone interval. Let $\mathcal{F}_1$ and $\mathcal{F}_2$ be two consecutive milestones. We want to know if there exists $f \in [\mathcal{F}_1, \mathcal{F}_2]$ such that a preemptive deadline scheduling is feasible, and if so, get the minimal one. Let $x_{i,j,t}$ be the fraction of T_i processed by M_j during the time interval $[E_t^f, E_{t+1}^f]$. Then the Linear System (2) provides a solution:

- we want to minimize the objective value (Equation (2a));
- each request must be completely processed (Equation (2b));
- the total processing time during a time interval cannot exceed its capacity (Equation (2c));
- the processing time of a request during a time interval cannot exceed its capacity, i.e., a given request cannot be simultaneously executed by several servers (Equation (2d));
- a request cannot be executed before its release time (Equation (2e));
- a request cannot be executed after its deadline (Equation (2f)).

$$\textbf{minimize} \quad f \tag{2a}$$

$$\textbf{subject to} \quad \forall i, \sum_{j,t} x_{i,j,t} = 1 \tag{2b}$$

$$\forall j, \forall t, \sum_{i} x_{i,j,t} \cdot p_{i,j} \leq E_{t+1}^f - E_t^f \tag{2c}$$

$$\forall i, \forall t, \sum_{j} x_{i,j,t} \cdot p_{i,j} \leq E_{t+1}^f - E_t^f \tag{2d}$$

$$\forall i, \forall j, \forall t, \mathcal{T}_r(i) \geq t+1 \implies x_{i,j,t} = 0 \tag{2e}$$

$$\forall i, \forall j, \forall t, \mathcal{T}_d(i) \leq t \implies x_{i,j,t} = 0 \tag{2f}$$

$$\mathcal{F}_1 \leq f \leq \mathcal{F}_2 \tag{2g}$$

Optimal solution. We have the set of milestones and a way to obtain the optimal solution in a milestone interval if there is one, hence we are able to find the globally optimal objective value by performing a binary search on the set of milestones. Let us now build the schedule from the provided solution f.

Schedule reconstruction. Let us assume that we are considering the t-th time interval $[E_t^f, E_{t+1}^f]$ (we will repeat the same procedure for all intervals, and concatenate the partial schedules). First, we build the $m \times n$ cost matrix A such that $A_{j,i} = x_{i,j,t} \cdot p_{i,j}$, which represents the fraction of request that should be executed during the current time interval. The procedure is to build iteratively the schedule by choosing a subset $\mathcal{D}$ of elements in A, called the decrementing set (at most one element per row and per column), and a time length δ at each step, until all elements of A are equal to zero. Let us construct the $(m+n) \times (m+n)$ bistochastic matrix

$$B = \left(\begin{array}{c|c} A & D_m \\ \hline D_n & A^T \end{array}\right),$$

where A^T is the transpose of A, and D_m (resp. D_n) is an $m \times m$ (resp. $n \times n$) diagonal matrix whose elements are such that each row sum and column sum of B is equal to $E_{t+1}^f - E_t^f$. As stated by the Birkhoff-von Neumann theorem, each bistochastic matrix is a convex combination of permutation matrices, i.e., $B = \sum_k c_k \mathcal{P}_k$, where each c_k is a coefficient and $\mathcal{P}_k$ is a $(m+n) \times (m+n)$ permutation matrix. The top-left $m \times n$ block of any $\mathcal{P}_k$ gives a decrementing set $\mathcal{D}$ to schedule in the current iteration: if $\mathcal{P}_{k_{j,i}} = 1$, then $A_{j,i} \in \mathcal{D}$, which means the request T_i may be executed by the server M_j.

We now compute the maximum processing time δ allowed during the current iteration. We denote a row j in A as tight if its sum is equal to $E_{t+1}^f - E_t^f$, and slack otherwise. The same terminology is used for a column i. δ is chosen to be maximum subject to the following constraints:

- if $A_{j,i} \in \mathcal{D}$ and is in a tight row or column, then $\delta \leq A_{j,i}$;
- if $A_{j,i} \in \mathcal{D}$ and is in a slack row, then $\delta \leq A_{j,i} + (E_{t+1}^f - E_t^f) - \sum_k A_{j,k}$;

– if $A_{j,i} \in \mathcal{D}$ and is in a slack column, then $\delta \leq A_{j,i} + (E^f_{t+1} - E^f_t) - \sum_k A_{k,i}$;
– if row j contains no element of $\mathcal{D}$, then $\delta \leq (E^f_{t+1} - E^f_t) - \sum_k A_{j,k}$;
– if column i contains no element of $\mathcal{D}$, then $\delta \leq (E^f_{t+1} - E^f_t) - \sum_k A_{k,i}$.

Finally, for each $A_{j,i} \in \mathcal{D}$, T_i is scheduled on M_j as soon as possible for $\min(\delta, p_{i,j})$ time units, and $A_{j,i}$ is replaced by $\max(0, A_{j,i} - \delta)$ in A.

6 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.

6.1 Replica Selection

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 [34]. 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: $M_r = \operatorname{rand} \mathcal{M}_i$. 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. The chosen replica minimizes $\mathcal{R}(M_j)$:

$$M_r = \operatorname*{argmin}_{M_j \in \mathcal{M}_i} \mathcal{R}(M_j).$$

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 [58].

HÉRON. We also consider an omniscient version of the replica selection heuristic used by Héron [34]. 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 [58]. The threshold is chosen according to the wanted proportion of large requests in the workload.

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:

$$M_r = \operatorname*{argmin}_{M_j \in \mathcal{M}_i} \text{FINISHTIME}(M_j).$$

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 [29]). 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 $\{M_b\}_{1 \leq b \leq m}$ 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. In the best case, when all servers in each interval of replicas 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, and f_X denote its probability density function. We denote by $p(X) = \alpha X + \beta$ the duration of the corresponding request (where α is the inverse of the bandwidth and β 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

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

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:

$$E\left[p_\omega(X)\right] = \frac{1}{k} \cdot E[p(X)]. \tag{3}$$

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 [34].

StaticWindow (SW). The 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 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). This strategy is inspired from the Sufferage heuristic [50]. Let $\mathcal{F}$ be the function giving the estimated weighted flow $\mathcal{F}(T_i, M_j) = w_i(\max(r_i, \text{FinishTime}(M_j)) + p_i - r_i)$ of T_i when scheduled on M_j as soon as possible. Let $\rho(T_i) = \operatorname{argmin}_{M_j \in \mathcal{M}_i} \mathcal{F}(T_i, M_j)$ be the "best" server for T_i, i.e., the one minimizing its weighted flow, and $\rho'(T_i) = \operatorname{argmin}_{M_j \in \mathcal{M}_i \setminus \rho(T_i)} \mathcal{F}(T_i, M_j)$ be the "second best" server for T_i. Then, we define the sufferage value $\text{Suff}(T_i) = \mathcal{F}(T_i, \rho'(T_i)) - \mathcal{F}(T_i, \rho(T_i)) > 0$ 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:

$$T_s = \operatorname*{argmax}_{T_i \in Q} \text{Suff}(T_i).$$

The chosen replica is $\rho(T_s)$:

$$M_r = \rho(T_s) = \operatorname*{argmin}_{M_j \in \mathcal{M}_i} \mathcal{F}(T_s, M_j).$$

Algorithm 3 SUFFERAGE-SW

loop {every q time units}
  for all $T_i \in Q$ **do**
    $\rho(T_i) \leftarrow \text{argmin}_{M_j \in \mathcal{M}_i} \mathcal{F}(T_i, M_j)$
    $\rho'(T_i) \leftarrow \text{argmin}_{M_j \in \mathcal{M}_i \setminus \rho(T_i)} \mathcal{F}(T_i, M_j)$
    $\textsc{Suff}(T_i) \leftarrow \mathcal{F}(T_i, \rho'(T_i)) - \mathcal{F}(T_i, \rho(T_i))$
  end for
  while Q is not empty **do**
    $T_s \leftarrow \text{argmax}_{T_i \in Q} \textsc{Suff}(T_i)$
    schedule T_s on $\rho(T_s)$
    $Q \leftarrow Q \setminus \{T_s\}$
    update ρ, ρ' and SUFF
  end while
end loop

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 $O(n^2 \cdot m)$ and uses a space $O(n)$ per time window.

MAXMIN-SW (MSW). This strategy is inspired from the Max-Min heuristic [50]. We build a matrix MAT whose rows are requests of set Q and columns are servers, where

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

The best weighted flow of request T_i is $\mathcal{F}_{\text{best}}(T_i) = \mathcal{F}(T_i, \rho(T_i)) = \min_{M_j \in M} \textsc{Mat}[T_i, M_j]$. Then, we schedule the request T_s whose "best" objective value is the highest:

$$T_s = \underset{T_i \in Q}{\text{argmax}}\, \mathcal{F}_{\text{best}}(T_i).$$

The chosen replica minimizes the objective value of T_s:

$$M_r = \underset{M_j \in M}{\text{argmin}}\, \textsc{Mat}[T_s, M_j].$$

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 $O(n^2 \cdot m)$ and uses a space $O(n \cdot m)$ per time window.

Table 1 summarizes the properties of our heuristics.

Table 1. Properties of replica selection 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 requests in Q and m is the total number of servers.

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

Algorithm 4 MaxMin-SW

```
1: loop {every q time units}
2:   for all T_i ∈ Q do
3:     for all M_j ∈ M do
4:       if M_j ∈ M_i then
5:         Mat[T_i, M_j] ← F(T_i, M_j)
6:       else
7:         Mat[T_i, M_j] ← +∞
8:       end if
9:     end for
10:  end for
11:  while Q is not empty do
12:    T_s ← argmax_{T_i∈Q} F_best(T_i)
13:    M_r ← argmin_{M_j∈M} Mat[T_s, M_j]
14:    schedule T_s on M_r
15:    Q ← Q \ {T_s}
16:    remove row T_s from Mat
17:    update column M_r in Mat
18:  end while
19: end loop
```

6.2 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 is a classic strategy, which is commonly used as a local scheduling policy in key-value stores (e.g., Cassandra [42]). 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 locally 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:

$$T_s = \operatorname*{argmax}_{T_i \in \mathcal{Q}} w_i(t + p_i - r_i).$$

This 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 ($w_i = \frac{1}{p_i}$), MWF is equivalent to the strategy that selects the request with maximal stretch (MaxStretch). 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 2 summarizes the properties of our heuristics.

Table 2. Properties of local queue 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 $\mathcal{Q}$ and m is the total number of servers.

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

7 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 `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.

7.1 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 $\theta = 0.5$, which gives a mean size equal to $\eta \cdot \Gamma\left(1 + \frac{1}{\theta}\right) = 64$ kB, where Γ is the gamma function; these parameters yield a long-tailed distribution that is consistent with existing file sizes characterizations [31]. 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 $\overline{p} = \alpha \cdot \eta \cdot \Gamma\left(1 + \frac{1}{\theta}\right) + \beta$ 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 $m = 15$ servers and we set the replication factor to $k = 3$, which is a common configuration in real implementations [26, 42]. The network bandwidth is set to $1/\alpha = 100$ Mbps and the average latency is set to $\beta = 1$ ms. For the threshold between requests for small and large values, we plug the density function of our Weibull distribution in Equation (3) and solve it numerically for ω:

$$\int_{x=\frac{\omega-\beta}{\alpha}}^{\infty} (\alpha x + \beta) \cdot \frac{\theta}{\eta} \cdot \left(\frac{x}{\eta}\right)^{\theta-1} \cdot e^{-\left(\frac{x}{\eta}\right)^{\theta}} dx = \frac{1}{k} \cdot \int_{x=0}^{\infty} (\alpha x + \beta) \cdot \frac{\theta}{\eta} \cdot \left(\frac{x}{\eta}\right)^{\theta-1} \cdot e^{-\left(\frac{x}{\eta}\right)^{\theta}}$$

This yields a threshold $\omega \approx 26.4$ ms (for a 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.

7.2 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.
- $w_i = \frac{1}{p_i}$. Pure 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.
- $w_i = \frac{1}{\sqrt{p_i}}$. Although the stretch metric is more fair than pure 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 tradeoff between latency and stretch metrics, and we denote it as the "weak stretch".

7.3 Results

Fig. 3 shows Empirical Cumulative Distribution Functions (ECDF) of the flow, the stretch and the weak stretch, for each combination of *distributed* selection heuristic and execution strategy. The dashed horizontal

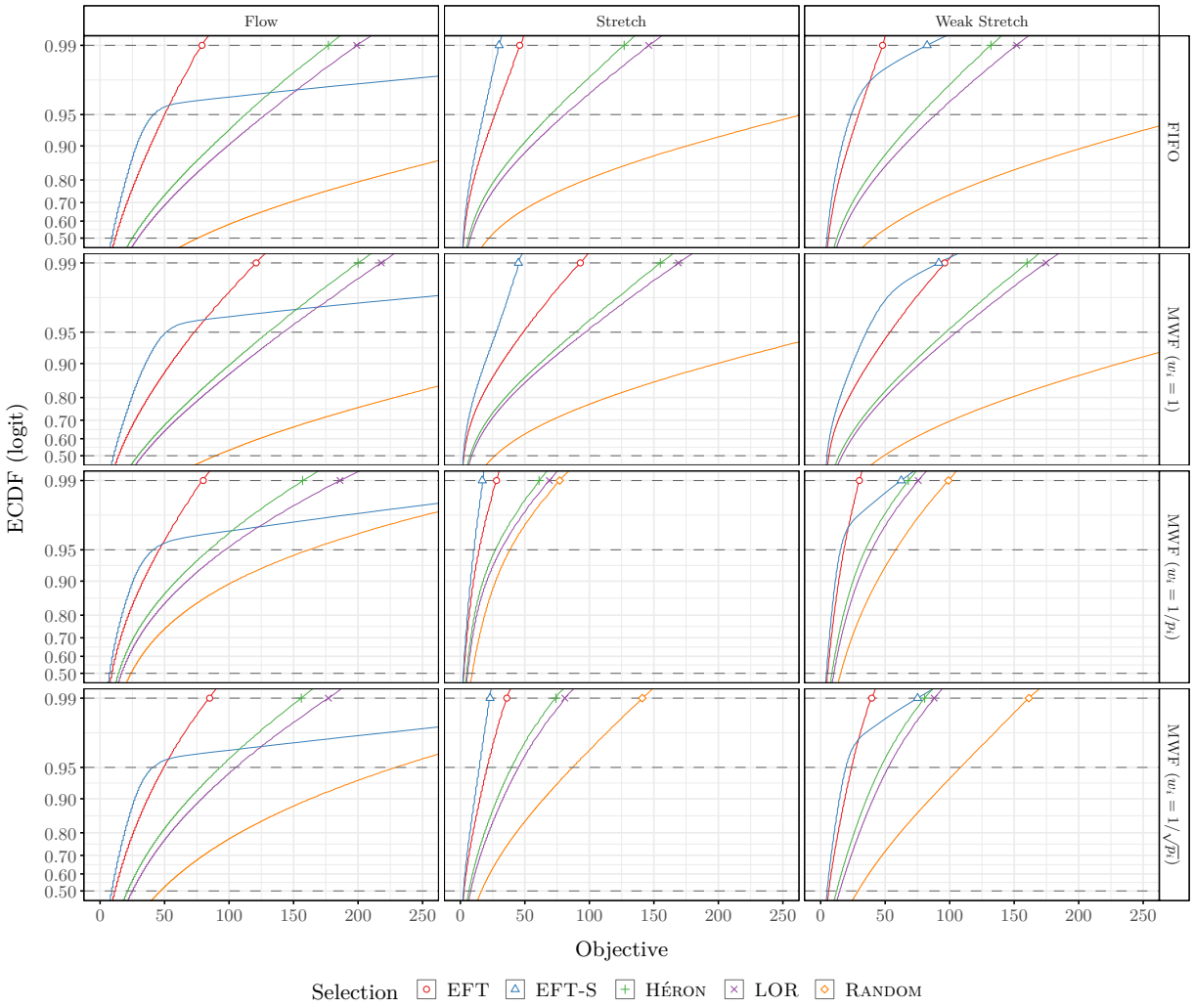

Fig. 3. ECDF of each metric generated by each combination of selection and execution heuristics in steady-state over 120 s, for an average load $\mathcal{L} = 0.9$.

lines respectively represent median, 95$^{\text{th}}$ and 99$^{\text{th}}$ percentile. Files are requested with a load $\mathcal{L} = 0.9$, and the simulations run for 120 s.

We show in Fig. 4 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 ranging from 1.0 to 9.0, combined to a FIFO execution. For SSW and MSW, we consider the stretch weighting ($w_i = \frac{1}{p_i}$), 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 s in order to simulate a short burst of requests.

Fig. 5 shows the 99$^{\text{th}}$ 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 99$^{\text{th}}$ 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 s.

The comparison of online heuristics with the lower bound introduced in Section 5 is shown in Fig. 6. We normalize the maximum objective $\max w_i F_i$ generated by a given heuristic with the lower bound. Each boxplot[8] 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. Files 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. 3-6 is that the choice on replica selection heuristic is indeed 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. 3. 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. 6).

Fig. 6 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. 5.

For the stretch metric, where latencies are weighted by processing times, EFT-S performs even better than EFT (Fig. 3, 5), yielding a 99$^{\text{th}}$ 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. 3, 5-6 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 strategy for max-flow on a single machine [14]. It is confirmed by our observations, as MWF worsen the tail-latency.

When a burst occurs, Fig. 4 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 outperforms EFT. This seems to confirm that EFT is a close-to-optimal strategy in average, as additional information do not allow to increase performance.

[8] Each boxplot consists of a bold line for the median, a box for the quartiles, whiskers that extend at most to 1.5 times the interquartile range from the box and additional points for outliers.

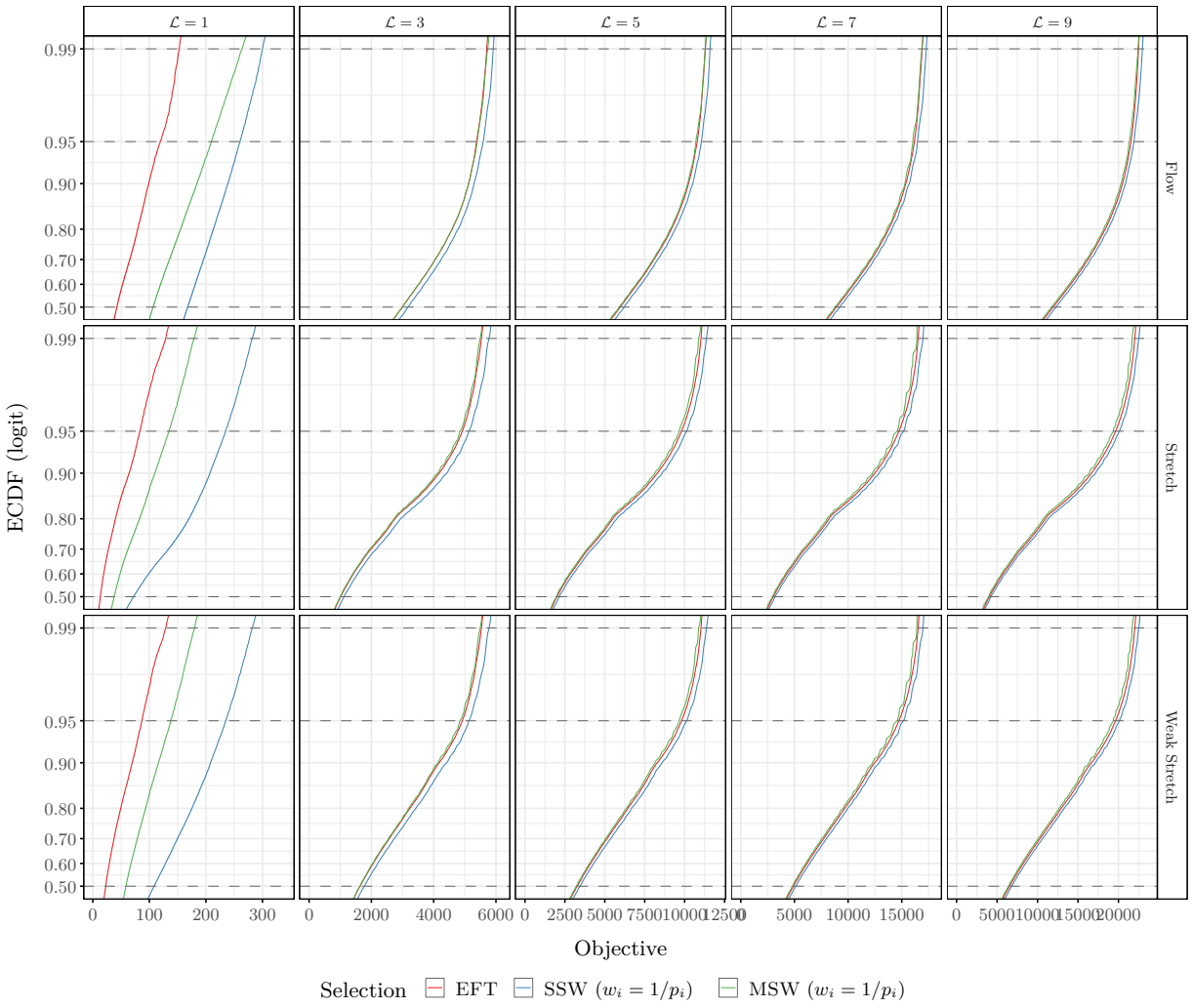

Fig. 4. ECDF of each metric generated by SSW and MSW in steady-state over 3 s, for load values ranging from 1.0 to 9.0.

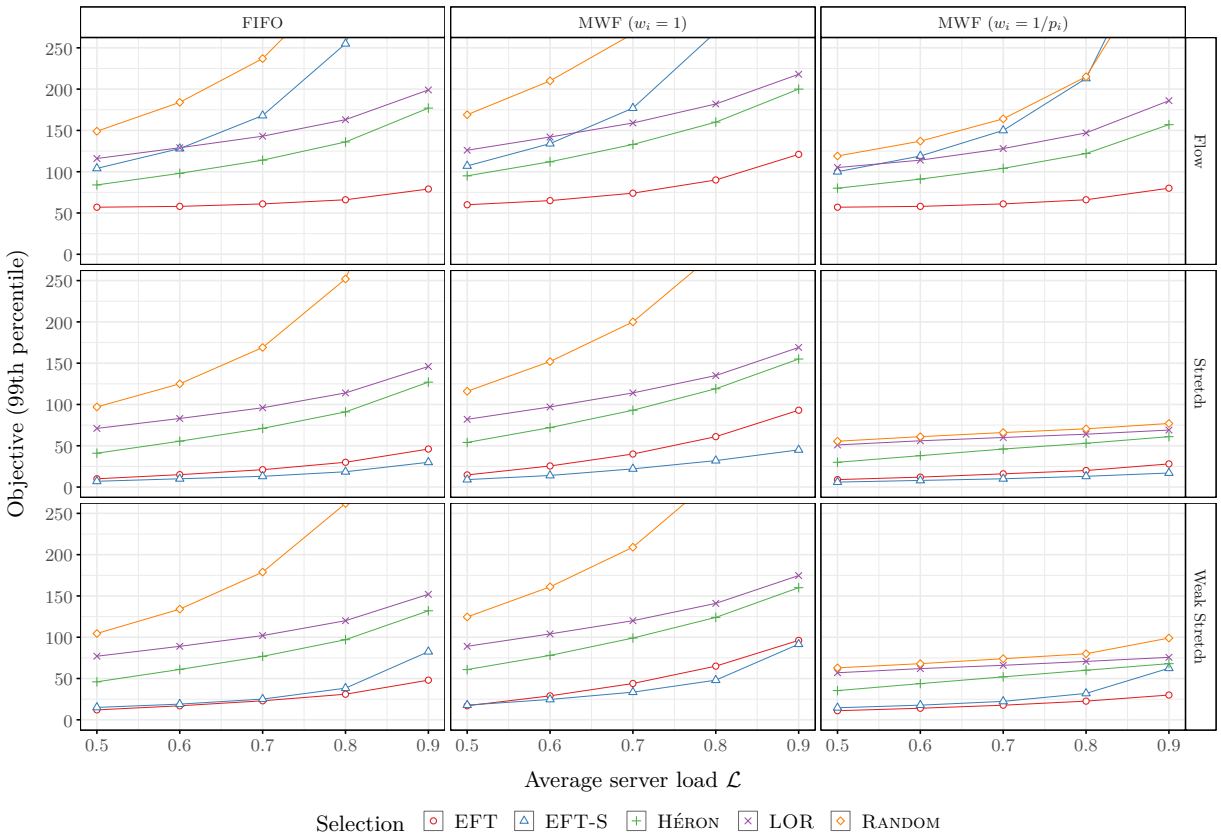

Fig. 5. 99^{th} quantiles of each metric as a function of average server load for each combination of selection and execution heuristics in steady-state over 120 s and for load values ranging from 0.5 to 0.9.

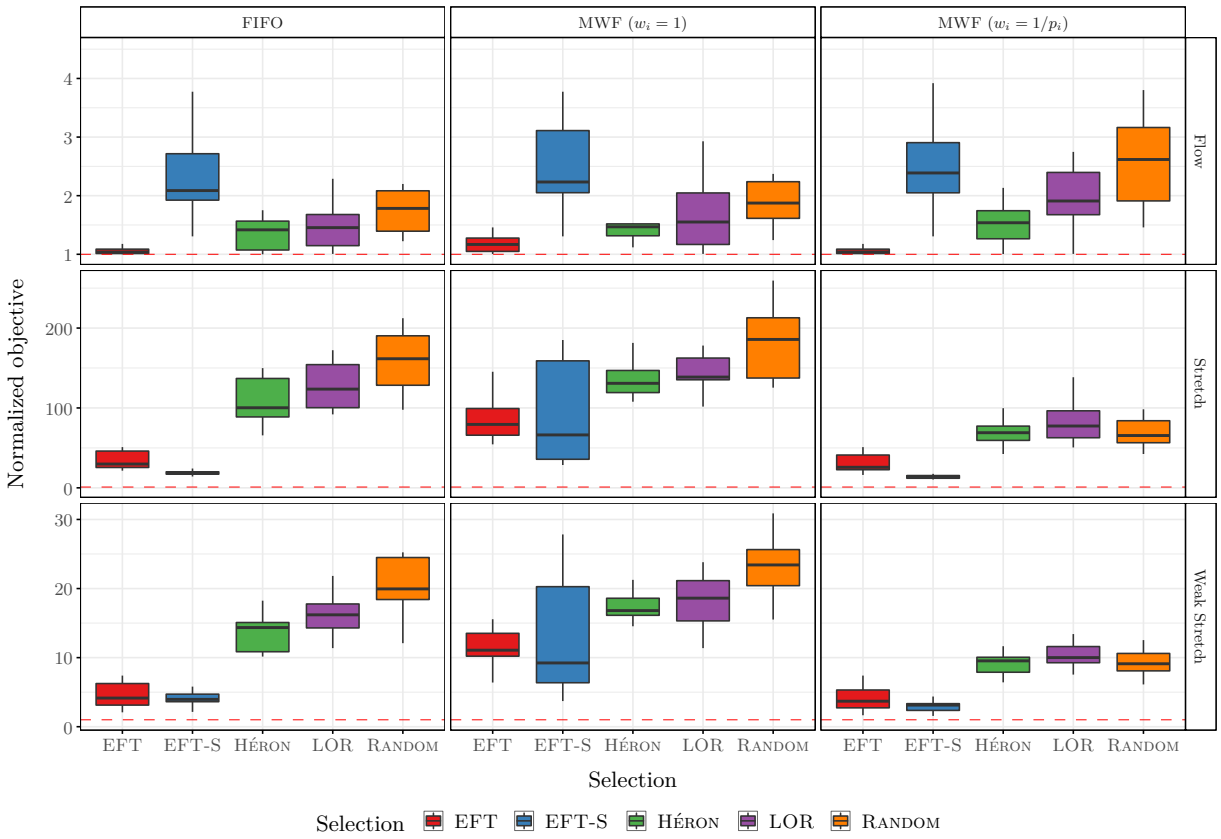

Fig. 6. Distributions of normalized maximums for each metric and for each combination of selection/execution heuristics. Files are requested with a load $\mathcal{L} = 0.9$, and the 10 scenarios consists of 1200 requests.

8 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 [34] or size-aware sharding [29] 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 [24,38], 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 [35,53].

References

1. Ambühl, C., Mastrolilli, M.: On-line scheduling to minimize max flow time: an optimal preemptive algorithm. Operations Research Letters **33**(6), 597–602 (2005)
2. Anand, S., Bringmann, K., Friedrich, T., Garg, N., Kumar, A.: Minimizing maximum (weighted) flow-time on related and unrelated machines. Algorithmica **77**(2), 515–536 (2017)
3. 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)
4. Awerbuch, B., Azar, Y., Leonardi, S., Regev, O.: Minimizing the flow time without migration. SIAM Journal on Computing **31**(5), 1370–1382 (2002)
5. Baker, K.R.: Introduction to sequencing and scheduling. John Wiley & Sons (1974)
6. Balmau, O., Dinu, F., Zwaenepoel, W., Gupta, K., Chandhiramoorthi, R., Didona, D.: Silk+ preventing latency spikes in log-structured merge key-value stores running heterogeneous workloads. ACM TOCS **36**(4), 1–27 (2020)
7. Bansal, N., Chan, H.L., Khandekar, R., Pruhs, K., Stein, C., Schieber, B.: Non-preemptive min-sum scheduling with resource augmentation. In: 48th Annual IEEE Symposium on Foundations of Computer Science (FOCS'07). pp. 614–624. IEEE (2007)
8. Bansal, N., Cloostermans, B.: Minimizing maximum flow-time on related machines. Theory of Computing **12**(1), 1–14 (2016)
9. Bansal, N., Dhamdhere, K.: Minimizing weighted flow time. ACM Transactions on Algorithms (TALG) **3**(4), 39–es (2007)
10. Bansal, N., Dhamdhere, K., Könemann, J., Sinha, A.: Non-clairvoyant scheduling for minimizing mean slowdown. In: Annual Symposium on Theoretical Aspects of Computer Science. pp. 260–270. Springer (2003)
11. Bansal, N., Pruhs, K.: Server scheduling in the l_p norm: a rising tide lifts all boat. In: ACM STOCS (2003)
12. Baptiste, P., Brucker, P., Chrobak, M., Dürr, C., Kravchenko, S.A., Sourd, F.: The complexity of mean flow time scheduling problems with release times. Journal of Scheduling **10**(2), 139–146 (2007)
13. Becchetti, L., Leonardi, S.: Nonclairvoyant scheduling to minimize the total flow time on single and parallel machines. Journal of the ACM (JACM) **51**(4), 517–539 (2004)
14. Bender, M.A., Chakrabarti, S., Muthukrishnan, S.: Flow and stretch metrics for scheduling continuous job streams. In: ACM-SIAM Symp. on Disc. Algo. (1998)
15. Benoit, A., Elghazi, R., Robert, Y.: Max-stretch minimization on an edge-cloud platform. In: IPDPS (2021)
16. Brucker, P., Jurisch, B., Krämer, A.: Complexity of scheduling problems with multi-purpose machines. Annals of Operations Research **70**, 57–73 (1997)
17. Brucker, P., Kravchenko, S.A.: Scheduling jobs with equal processing times and time windows on identical parallel machines. Journal of Scheduling **11**(4), 229–237 (2008)
18. Bruno, J., Coffman Jr, E.G., Sethi, R.: Scheduling independent tasks to reduce mean finishing time. Communications of the ACM **17**(7), 382–387 (1974)
19. Brutlag, J.: Speed matters for google web search (2009)
20. Carlson, J.L.: Redis in action. Manning Publications Co. (2013)
21. Chadha, J.S., Garg, N., Kumar, A., Muralidhara, V.: A competitive algorithm for minimizing weighted flow time on unrelatedmachines with speed augmentation. In: Proceedings of the forty-first annual ACM symposium on Theory of computing. pp. 679–684 (2009)

22. Chekuri, C., Khanna, S., Zhu, A.: Algorithms for minimizing weighted flow time. In: Proceedings of the thirty-third annual ACM symposium on Theory of computing. pp. 84–93 (2001)
23. Chodorow, K.: MongoDB: the definitive guide: powerful and scalable data storage. " O'Reilly Media, Inc." (2013)
24. Choudhury, A.R., Das, S., Garg, N., Kumar, A.: Rejecting jobs to minimize load and maximum flow-time. In: ACM-SIAM Symp. Disc. Algo. pp. 1114–1133 (2014)
25. Dean, J., Barroso, L.A.: The tail at scale. Communications of the ACM **56**(2), 74–80 (2013)
26. 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)
27. Delgado, P., Didona, D., Dinu, F., Zwaenepoel, W.: Job-aware scheduling in eagle: Divide and stick to your probes. In: Proceedings of the Seventh ACM Symposium on Cloud Computing. pp. 497–509. ACM (2016)
28. Delgado, P., Dinu, F., Kermarrec, A.M., Zwaenepoel, W.: Hawk: Hybrid datacenter scheduling. In: 2015 USENIX Annual Technical Conference (USENIX ATC 15). pp. 499–510 (2015)
29. Didona, D., Zwaenepoel, W.: Size-aware sharding for improving tail latencies in in-memory key-value stores. In: NSDI. pp. 79–94 (2019)
30. Dutot, P.F., Saule, E., Srivastav, A., Trystram, D.: Online non-preemptive scheduling to optimize max stretch on a single machine. In: International Computing and Combinatorics Conference. pp. 483–495. Springer (2016)
31. Feitelson, D.G.: Workload modeling for computer systems performance evaluation. Cambridge University Press (2015)
32. Garg, N., Kumar, A.: Minimizing average flow-time: Upper and lower bounds. In: 48th Annual IEEE Symposium on Foundations of Computer Science (FOCS'07). pp. 603–613. IEEE (2007)
33. Graham, R.L.: Bounds for certain multiprocessing anomalies. Bell System Technical Journal **45**(9), 1563–1581 (1966)
34. 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)
35. Jaiman, V., Mokhtar, S.B., Rivière, E.: TailX: Scheduling heterogeneous multiget queries to improve tail latencies in key-value stores. DAIS (2020)
36. 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)
37. Jose, J., Subramoni, H., Luo, M., Zhang, M., Huang, J., Wasi-ur Rahman, M., Islam, N.S., Ouyang, X., Wang, H., Sur, S., et al.: Memcached design on high performance rdma capable interconnects. In: 2011 International Conference on Parallel Processing. pp. 743–752. IEEE (2011)
38. Kalyanasundaram, B., Pruhs, K.: Speed is as powerful as clairvoyance. Journal of the ACM (JACM) **47**(4), 617–643 (2000)
39. Kellerer, H., Tautenhahn, T., Woeginger, G.: Approximability and nonapproximability results for minimizing total flow time on a single machine. SIAM Journal on Computing **28**(4), 1155–1166 (1999)
40. Kravchenko, S.A., Werner, F.: Preemptive scheduling on uniform machines to minimize mean flow time. Computers & Operations Research **36**(10), 2816–2821 (2009)
41. Labetoulle, J., Lawler, E.L., Lenstra, J.K., Kan, A.R.: Preemptive scheduling of uniform machines subject to release dates. In: Progress in combinatorial optimization, pp. 245–261. Elsevier (1984)
42. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review **44**(2), 35–40 (2010)
43. 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)
44. Legrand, A., Su, A., Vivien, F.: Minimizing the stretch when scheduling flows of divisible requests. Journal of Scheduling **11**(5), 381–404 (2008)
45. Lenstra, J.K., Kan, A.R., Brucker, P.: Complexity of machine scheduling problems. Studies in integer programming **1**, 343–362 (1977)
46. Leonardi, S., Raz, D.: Approximating total flow time on parallel machines. Journal of Computer and System Sciences **73**(6), 875–891 (2007)
47. Leung, J.Y.T., Li, C.L.: Scheduling with processing set restrictions: A survey. International Journal of Production Economics **116**(2), 251–262 (2008)
48. 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)
49. Lucarelli, G., Moseley, B.: Online non-preemptive scheduling to minimize maximum weighted flow-time on related machines. In: FSTTCS (2019)

50. Maheswaran, M., Ali, S., Siegel, H.J., Hensgen, D., Freund, R.F.: Dynamic mapping of a class of independent tasks onto heterogeneous computing systems. Journal of parallel and distributed computing **59**(2), 107–131 (1999)
51. Mastrolilli, M.: Scheduling to minimize max flow time: Off-line and on-line algorithms. International Journal of Foundations of Computer Science **15**(02), 385–401 (2004)
52. Muthukrishnan, S., Rajaraman, R., Shaheen, A., Gehrke, J.E.: Online scheduling to minimize average stretch. In: 40th Annual Symposium on Foundations of Computer Science (Cat. No. 99CB37039). pp. 433–443. IEEE (1999)
53. Reda, W., Canini, M., Suresh, L., Kostić, D., Braithwaite, S.: Rein: Taming tail latency in key-value stores via multiget scheduling. EuroSys (2017)
54. Saule, E., Bozdağ, D., Çatalyürek, Ü.V.: 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 (2012)
55. Shabtay, D., Karhi, S.: Online scheduling of two job types on a set of multipurpose machines with unit processing times. Computers & Operations Research **39**(2), 405–412 (2012)
56. Simons, B.: Multiprocessor scheduling of unit-time jobs with arbitrary release times and deadlines. SIAM Journal on Computing **12**(2), 294–299 (1983)
57. Sitters, R.: Two np-hardness results for preemptive minsum scheduling of unrelated parallel machines. In: International Conference on Integer Programming and Combinatorial Optimization. pp. 396–405. Springer (2001)
58. Suresh, L., Canini, M., Schmid, S., Feldmann, A.: C3: Cutting tail latency in cloud data stores via adaptive replica selection. NSDI (2015)
59. Vulimiri, A., Godfrey, P.B., Mittal, R., Sherry, J., Ratnasamy, S., Shenker, S.: Low latency via redundancy. In: Proceedings of the ninth ACM conference on Emerging networking experiments and technologies. pp. 283–294. ACM (2013)
60. Wu, Z., Yu, C., Madhyastha, H.V.: Costlo: Cost-effective redundancy for lower latency variance on cloud storage services. In: 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15). pp. 543–557 (2015)

A Notations

Table 3 provides a list of the most used notations in this document.

Table 3. List of notations.

Symbol	Definition
i	index of requests
j	index of machines
l	index of keys/values
n	number of requests
m	number of machines
c	number of keys/values
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
$K_l \in K$	key l
$V_l \in V$	value l
w_i	weight of T_i
r_i	release time of 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
z_l	size of V_l
$\varphi : T \to K$	gives the key carried by a request
$\Psi : M \to 2^K$	gives the key subset owned by a machine
$\Pi : T \to M \times \mathbb{N}$	gives the machine and start time of a request
α	bandwidth inverse
β	latency
λ	submission rate
η	Weibull scale parameter
θ	Weibull shape parameter
$\mathcal{L}$	load

B Competitive Analysis

We provide a comprehensive summary of results related to competitive analysis of online scheduling problems which address the minimization of flow time. Tables are organized in the following way: the results are classified into two general columns, where the first one deals with offline optimization and the second one deals with online aspects, and more particularly with competitiveness.

We specify the modalities for each competitive analysis, where the symbol C indicates a clairvoyant problem (i.e., all request properties are known) and S stands for a speed augmentation characteristic (i.e., the online solution is allowed to use an increased fraction of resources compared to offline). Table 4 (resp. Table 5) presents competitive analysis results which are related to maximum (resp. sum) weighted flow. The more general objective ℓ_p-norm is described in Table 6.

Table 4. Competitive analysis results related to maximum (weighted) flow minimization.

	Offline		Online		
Problem	Opt	Approx	C	S	Competitive Ratio
$1\|r_i\|F_{\max}$	FIFO [14]		no	no	– FIFO is 1-competitive
$P\|r_i\|F_{\max}$			no	no	– LB of $2-\frac{1}{m}$
					– FIFO is $(3-\frac{2}{m})$-competitive [14]
$P2\|r_i\|F_{\max}$			no	no	– LB of 2 [51]
					– FIFO is 2-competitive [1, 51]
$P\|r_i, pmtn\|F_{\max}$			no	no	– LB of $2-\frac{1}{m}$
					– FIFO is $(3-\frac{2}{m})$-competitive [51]
					– $(2-\frac{1}{m})$-competitive algorithm [1]
$Q\|r_i, pmtn^*\|F_{\max}$			yes	no	– Slow-Fit is $\Omega(m)$-competitive
					– Greedy is $\Omega(\log m)$-competitive
					– $O(1)$-competitive algorithm [8]
$1\|r_i, p_i=1\|\max w_iF_i$					– no 1-competitive algorithm [Th. 5]
					– Single-Unit has no competitive ratio [Th. 6]
$P\|\|\max w_iF_i$		– $2-\frac{1}{m}$ [Th. 3]			
$P\|r_i\|\max w_iF_i$			yes	no	– LB: $\Omega(W)$ [8]
$R\|r_i, pmtn\|\max w_iF_i$	LP [44]				
$1\|r_i\|S_{\max}$		– LB: $\Omega(n^{1-\varepsilon})$ [14]	yes	no	– FIFO is P-comp [44]
					– LB: $1+\frac{\sqrt{5}-1}{2}P$
					– $(1+\frac{\sqrt{5}-1}{2}P)$-comp algo [30]
$1\|r_i, pmtn\|S_{\max}$		– $1+\varepsilon$ [14]	yes	no	– $\max(\alpha, 1+\frac{P}{\alpha})$-comp algo [14]

Table 5. Competitive analysis results related to sum (weighted) flow minimization.

	Offline		Online		
Problem	Opt	Approx	C	S	Competitive Ratio
$1\|r_i\|\sum F_i$		– $O(\sqrt{n})$ – LB: $\Omega(n^{\frac{1}{2}-\varepsilon})$ [39]	no	no	– FIFO is P-comp [44]
$1\|r_i, pmtn\|\sum F_i$	SRPT [5]		yes	no	– SRPT is 1-comp
$1\|r_i, pmtn\|\sum F_i$	SRPT		no	$1+\varepsilon$	– SETF is $(1+\frac{1}{\varepsilon})$-comp [38]
$1\|r_i, pmtn\|\sum F_i$	SRPT		no	no	– RMLF is $O(\log n)$-comp [13]
$P\|r_i\|\sum F_i$		– $O(\sqrt{\frac{n}{m}}\log\frac{n}{m})$ – LB: $\Omega(n^{\frac{1}{3}-\varepsilon})$ [46]	no	no	– Random is $O(\varepsilon^{-3}\log\frac{1}{\varepsilon})$-comp [22]
$P\|r_i, pmtn^*\|\sum F_i$		– $O(\log P)$	no	no	– $O(\min(\log P, \log n))$-comp algo [4] – LB: $\Omega(\log P)$
$P\|r_i, pmtn\|\sum F_i$			yes	no	– SRPT is $O(\min(\log P, \log\frac{n}{m}))$-comp – LB: $\Omega(\log(\frac{n}{m}+P))$ (randomized) [46]
$P\|r_i, pmtn\|\sum F_i$			no	no	– RMLF is $O(\log n\cdot\min(\log P, \log\frac{n}{m}))$-comp [13]
$P\|\mathcal{M}_i, r_i, pmtn^*\|\sum F_i$		– $O(\log P)$ [32]	no	no	– no LB [32]
$1\|r_i\|\sum w_iF_i$		– $O(1)$ [7]		$O(1)$	
$1\|r_i, pmtn\|\sum w_iF_i$		– $2+\varepsilon$ [22] – $O(\log n+\log P)$ [9]	yes	no	– $O(\log^2 P)$-comp algo – LB: 1.618 – LB: $\frac{4}{3}$ (randomized) [22] – $O(\log W)$-comp algo [9]
$1\|r_i, pmtn\|\sum w_iF_i$			no	$1+\varepsilon$	– $(1+\frac{1}{\varepsilon})$-comp algo [9]
$P\|r_i, pmtn\|\sum w_iF_i$			no	no	– LB: $\Omega(\min(\sqrt{P}, \sqrt{W}, (\frac{n}{m})^{\frac{1}{4}}))$ (randomized) [22]
$R\|r_i, pmtn^*\|\sum w_iF_i$			no	$1+\varepsilon$	– $O((1+\frac{1}{\varepsilon})^2)$-comp algo [21]
$1\|r_i\|\sum S_i$			no	no	– FIFO is P^2-comp [44]
$1\|r_i, pmtn\|\sum S_i$			yes	no	– SRPT is 2-comp [52]
$1\|r_i, pmtn\|\sum S_i$			no	$1+\varepsilon$	– MLF is $O((\frac{1}{\varepsilon})^5\log^2 P)$-comp [10]
$P\|r_i, pmtn\|\sum S_i$			yes	no	– SRPT is 14-comp [52] – 9.82-comp algo [22]
$P\|r_i, pmtn^*\|\sum S_i$			yes	no	– 17.32-comp algo [22]

Table 6. Competitive analysis results related to ℓ_p-norm minimization.

	Offline		Online		
Problem	Opt	Approx	C	S	Competitive Ratio
$1\|r_i, pmtn\|\|F_i\|_p$			yes	no	– no $n^{o(1)}$-comp algo [11]
$1\|r_i, pmtn\|\|F_i\|_p$			yes	$1+\varepsilon$	– SJF is $O(\frac{1}{\varepsilon})$-comp – SRPT is $O(\frac{1}{\varepsilon})$-comp [11]
$1\|r_i, pmtn\|\|F_i\|_p$			no	$1+\varepsilon$	– SETF is $O(\frac{1}{\varepsilon^{2+2/p}})$-comp [11]
$1\|r_i, pmtn\|\|S_i\|_p$			yes	no	– no $n^{o(1)}$-comp algo [11]
$1\|r_i, pmtn\|\|S_i\|_p$			yes	$1+\varepsilon$	– SJF is $O(\frac{1}{\varepsilon})$-comp – SRPT is $O(\frac{1}{\varepsilon})$-comp [11]
$1\|r_i, pmtn\|\|S_i\|_p$			no	$1+\varepsilon$	– LB: $\Omega(\min(n, \log P))$ – SETF is $O(\frac{1}{\varepsilon^{3+1/p}}\log^{1+1/p} P)$-comp [11]

C Problems Complexity

To complete our survey on scheduling problems related to our study, we present a summary on complexity results in Table 7.

Table 7. Classification of scheduling problems. Arrows are reduction relationships, e.g., $A \rightarrow B$ means B is more difficult than A. Entries marked with a symbol **?** mean that the corresponding problem complexity is still unknown. A **+** means the problem is NP-hard (via the reduction relationship), while a **-** indicates a polynomially solvable problem. Incompatible problem designations are noted $\emptyset$.

$\sum w_i C_i$	1	$\rightarrow$ P	$\rightarrow$ Q	$\rightarrow$ R
$\mathcal{M}_i$	?	s. NP-hard [18]	+	+
	?	s. NP-hard [18]	+	+
$r_i, \mathcal{M}_i, p_i = 1$	-	p. solvable [16]	?	?
$r_i, p_i = p$	-	p. solvable [17]	?	$\emptyset$
$r_i, pmtn$	s. NP-hard [41]	+	+	+
$\sum C_i$	1	$\rightarrow$ P	$\rightarrow$ Q	$\rightarrow$ R
r_i	s. NP-hard [45]	+	+	+
$\mathcal{M}_i$	-	-	-	p. solvable [18]
	-	-	-	p. solvable [18]
$r_i, \mathcal{M}_i, p_i = 1$	-	-	?	?
$r_i, p_i = p$	-	p. solvable [56]	?	$\emptyset$
$r_i, pmtn$	p. solvable [5]	s. NP-hard [12]	+	+
$\mathcal{M}_i, pmtn$	-	p. solvable [16]	?	s. NP-hard [57]
$r_i, pmtn, p_i = p$	-	-	p. solvable [40]	?
$\sum S_i$	1	$\rightarrow$ P	$\rightarrow$ Q	$\rightarrow$ R
r_i	NP-hard [44]	+	+	+
$\max w_i F_i$	1	$\rightarrow$ P	$\rightarrow$ Q	$\rightarrow$ R
r_i	NP-hard [14]	+	+	+
$r_i, pmtn$	-	-	-	p. solvable [44]
$r_i, pmtn^*$	-	NP-hard [Th. 4]	+	+
	p. solvable [Th. 1]	s. NP-hard [Sec. 4.1]	+	+
$p_i = p$	-	p. solvable [Th. 2]	?	$\emptyset$
$F_{\max}$	1	$\rightarrow$ P	$\rightarrow$ Q	$\rightarrow$ R
r_i	p. solvable [14]	NP-hard [Sec. 4.1]	+	+
$S_{\max}$	1	$\rightarrow$ P	$\rightarrow$ Q	$\rightarrow$ R
r_i	NP-hard [14]	+	+	+
	?	s. NP-hard [15]	+	+