Numéro National de Thèse : 1234

THÈSE DE DOCTORAT DE L'UNIVERSITÉ DE LYON

opérée par

l'École Normale Supérieure de Lyon

École Doctorale n° 512

École Doctorale en Informatique et Mathématiques de Lyon

Spécialité : Informatique

présentée et soutenue publiquement le 1/1/2000, par :

Anthony DUGOIS

Scheduling in Key-Value Stores

Ordonnancement dans les bases de données clé-valeur

Devant le jury composé de :

Contents

Introduction

Outline of the Thesis

Chapter 1: Preliminaries & Related Work

In this first chapter, we present the context of our work and we introduce some basic notions. In particular, we describe the general architecture of key-value stores and we explain the main challenges of these systems. We also quickly recall the basis of scheduling theory (i.e., definitions, Graham's notation, and classical results). Then, we review the existing literature on scheduling problems that are related to our work.

Chapter 2: Scheduling Requests in Distributed Key-Value Stores [33]

After having introduced key-value store systems and scheduling theory, we build the theoretical framework that we will use in the rest of this manuscript. We formulate the tail latency problem as a scheduling problem, that is to say, we want to execute requests on servers so as to minimize the maximum flow time $F_{\max}$ of the requests, which is not a well-understood objective function. However, one may easily see through simple reductions to classical results (e.g., the well-known makespan problem $P \,||\, C_{\max}$) that this general scheduling problem is already NP-hard. Thus, we begin to explore simpler variants in order to discover where stands the line between tractable and intractable problems in the considered settings. By considering the variant where preemption is allowed, we leverage an existing result to show how to find a lower bound for the maximum weighted flow time in any instance of the problem. This enables us to compare and evaluate scheduling heuristics according to an identical baseline through extensive simulations.

Chapter 3: Bounds and Inapproximability under Replicated Datasets [7, 8]

In typical systems, each partition of the dataset is replicated on a small number of servers, which means that a given request cannot be processed on any server. We continue to explore the theoretical framework described in the previous chapter by introducing such eligibility constraints in the scheduling problems. The goal is to model the limited replication of data items in key-value stores and to understand how different replication schemes may impact the performance of the system. We consider two aspects. First, we consider the online model in which requests arrive over time (i.e., we cannot predict the future), and we derive several lower bounds on the achievable competitive ratio of any online algorithm. AD *be more specific on contributions* Second, we design a general method to compute the theoretical maximum attainable throughput according to a given replication strategy and under a given distribution of key popularity. AD *be more specific on contributions*

Chapter 4: Partitioning Multi-Get Requests

We now focus on one particular scheme, in which the replication subsets form contiguous intervals of machines. This next chapter is devoted to the study of a specific type of request, namely *multi-get* requests,

which are requests able to retrieve multiple data items at once in the key-value store. When entering the system, a multi-get request must be split into a set of subrequests, each of which must be processed by a specific server of the cluster. This partitioning process must be done in such a way that the overall response time of the request is minimized, i.e., we do not want a subrequest to be too long compared to the others. This can be seen as the Restricted Assignment problem on Intervals (sometimes abbreviated RAI), which is a famous scheduling problem. We extend existing work to design efficient and guaranteed algorithms for some variants of the RAI problem, and we generalize the framework by introducing the notion of *circular* intervals that can be used to model the actual replication strategy of key-value stores. In this setting, we propose a general method to compute the optimal makespan of any instance where intervals are circular, at the condition that jobs may be categorized in (at most) K classes.

AD *TODO: add references to publications*

Chapter 5: Implementing and Evaluating Scheduling Strategies

In this final chapter, we take a more experimental approach and study a real key-value store system. We begin with a case study of Apache Cassandra, which is an industry-standard key-value store that is widely used in production. We detail how requests are actually scheduled in such a system, and we identify several related challenges. Then, we present Hector, a modular framework built on top of Apache Cassandra, whose goal is to facilitate the design and evaluation of scheduling algorithms for key-value stores, as well as to improve the evaluation step by providing a common baseline for comparing different solutions. We describe the various components of Hector and we illustrate their usage through several implementations. Then, we conduct a series of experiments to assess that Hector itself does not bring any significant overhead to the system, and we show how to evaluate various performance aspects in the key-value store. For instance, we show that, under certain conditions on the workload, leveraging the cache of the operating system may significantly improve the throughput of the system. Another example is that reordering requests locally on each server may improve the overall latency when the dataset is heterogeneous.

AD *TODO: add references to publications*

Résumé français

1

Preliminaries & Related Work

1.1 Introduction to Key-Value Stores

1.1.1 In-Memory vs. Persistent Key-Value Stores

1.2 Related Work on Key-Value Stores

1.3 Introduction to Scheduling Theory

Since several decades, scheduling theory has been a very active field of research in computer science. A huge amount of work has been done on the subject, and many different problems have been studied. In this section, we introduce the basic concepts of scheduling theory and we present some classical results. Then we review more particularly the literature that is related to this thesis.

1.3.1 Basic Concepts

Scheduling refers to a set of computational problems that consist in assigning *activities* (for instance, processes, tasks, lectures) to renewable *resources* (for instance, processors, workstations, rooms) in an optimal manner, for a given definition of optimality that depends on the considered problem (for instance, minimizing the total time to compute something, or maximizing the usage of rooms in a given time span).

Among these problems, we are interested in *machine scheduling* (also called *processor scheduling*). Simple problems of this class are the *single-machine* scheduling problems. Consider a set of n computational jobs, such that each job j has a *processing time* p_j. We want to schedule these jobs on a single machine such that a given objective function is minimized. For example, say we want to minimize the average completion time $\frac{1}{n}\sum_{j=1}^{n} C_j$ of the jobs, where C_j denotes the completion time of job j in a given schedule. Minimizing this value is strictly equivalent to minimizing the total completion time $\sum_{j=1}^{n} C_j$, as the number of jobs n is fixed among all possible schedules.

Example 1.1. *Consider an instance with $n = 3$ jobs and the following processing times: $p_1 = 5$, $p_2 = 1$, $p_3 = 1$. Scheduling these jobs in-order yields $C_1 = 5$, $C_2 = 6$ and $C_3 = 7$, which gives an average completion time of $\frac{1}{3}(5 + 6 + 7) = 6$. We clearly see that this is not the optimal solution, as one may schedule jobs 2 and 3 before job 1, which yields $C_1 = 7$, $C_2 = 1$ and $C_3 = 2$, for an average completion time of $\frac{10}{3} \approx 3.34$.*

AD *Give a Gantt chart here, as this is a useful tool that will be used throughout the manuscript*

Of course, this example of a single-machine problem is very simple. One may consider additional constraints that make the problem more difficult to solve to optimality. For example, the following (non-exhaustive) constraints are often considered and have been well-studied:

- **Precedence relations**: a precedence relation between two jobs j and j' means that the job j' cannot start before the completion of job j.
- **Release times**: each job j is released at a given time r_j and cannot start before this time.
- **Due dates**: each job j has a due date d_j and must be completed, if possible, before this specific time. The due dates are sometimes hard deadlines, meaning that a job j that is not completed before d_j is rejected.

These constraints are not exclusive and may be combined into a single problem (often making the model more accurate according to the context, at the cost of a harder problem to solve). A natural generalization of single-machine scheduling is *parallel* scheduling. In parallel scheduling problems, the jobs are scheduled on m identical machines. By considering *identical* machines, one means that the processing time p_j of a job j does not depend on the machine on which it is scheduled. A well-known example of parallel scheduling problem is the MAKESPAN problem, where the objective is to schedule n jobs on m identical machines, in such a way that the maximum completion time $C_{\max} = \max_{1 \leq j \leq n} \{C_j\}$, called the *makespan*, is minimized.

Example 1.2. *Consider an instance of the* MAKESPAN *problem with $n = 3$ jobs and $m = 2$ machines, and the following processing times: $p_1 = 1$, $p_2 = 2$ and $p_3 = 5$. Scheduling these jobs in-order as soon as a machine becomes idle (i.e., job 1 is assigned on machine 1, job 2 is assigned on machine 2, and job 3 is assigned on machine 1) yields a makespan of 6. An optimal solution could consist in scheduling jobs 1 and 2 on machine 1, and job 3 on machine 2, for a makespan of 5.*

A further generalization are *unrelated* scheduling problems. In this case, the machines are not necessarily identical, which means that a same job j could have different processing times according to the machine on which it is scheduled. In other words, the processing times can be seen as a matrix, where rows represent machines and columns represent jobs. Thus, a job j has a processing time p_{ij} when scheduled on machine i. The corresponding generalization of the MAKESPAN problem is the UNRELATED problem.

Example 1.3. *Consider an instance of the* UNRELATED *problem with $n = 3$ jobs and $m = 2$ machines, and the following processing times:*

$$\begin{pmatrix} 1 & 1 & 2 \\ 2 & 5 & 1 \end{pmatrix}.$$

For instance, job 1 has processing time 1 on machine 1 and processing time 2 on machine 3. In this case, an optimal solution could consist in scheduling job 1 on machine 1, job 2 on machine 2 and job 3 on machine 3.

An intermediary class between parallel and unrelated scheduling is *uniform* scheduling, where each machine i has a speed s_i, and each job j has a processing time $p_{ij} = \frac{p_j}{s_i}$ when scheduled on i.

1.3.2 Graham's Classification

Given the diversity of scheduling problems, a specific classification has been introduced several years ago by Graham et al. [17]. A scheduling problem consists of a triple $\alpha \,|\, \beta \,|\, \gamma$, where α is the *processor environment*, β is the list of *job characteristics*, and γ is the *objective function* (also called the *optimality criterion*).

First, the processor environment α defines the number and the nature of machines in the scheduling problem. 1 stands for single-machine scheduling, P for parallel scheduling, Q for uniform scheduling, and R for unrelated scheduling. More types exist, such that O, F, J for open-shop, flow-shop and job-shop problems, respectively, but we only consider the four types above in this thesis.

Second, the job characteristics β define properties of the jobs in the scheduling problem. There exists a large number of such properties in the literature. We describe only a few of them, and refer the reader to classical scheduling books for more details [13, 35]. $p_j = p$ denotes that all jobs have the same processing time p, and $p_j = 1$ is the special case where they all have unitary processing times. $pmtn$ indicates that job preemption is allowed, i.e., jobs can be interrupted and resumed later. r_j stands for release dates, and d_j for due dates (or deadlines). $\mathcal{M}_j$ denotes processing set restrictions, i.e., a job j can only be processed on a subset $\mathcal{M}_j \subseteq \{1, 2, \cdots, m\}$ of machines.

Finally, the objective function γ defines the criterion to optimize in the scheduling problem. Again, a large number of different criteria exist in the literature. We describe in Table 1.1 the various functions that are used throughout this thesis. Note that some problems are *multi-objective*, meaning that they seek to optimize several criteria simultaneously.

Notation	Name	Formal Definition
$C_{\max}$	makespan	$\max_{1\leq j\leq n}\{C_j\}$
$\max w_j C_j$	weighted makespan	$\max_{1\leq j\leq n}\{w_j C_j\}$
$\sum C_j$	average completion time	$\sum_{j=1}^{n} C_j$
$\sum w_j C_j$	average weighted completion time	$\sum_{j=1}^{n} w_j C_j$
$F_{\max}$	maximum flow time	$\max_{1\leq j\leq n}\{C_j - r_j\}$
$\max w_j F_j$	maximum weighted flow time	$\max_{1\leq j\leq n}\{w_j(C_j - r_j)\}$
$\sum F_j$	average flow time	$\sum_{j=1}^{n} C_j - r_j$
$\sum w_j F_j$	average weighted flow time	$\sum_{j=1}^{n} w_j(C_j - r_j)$
$S_{\max}$	maximum stretch	$\max_{1\leq j\leq n}\left\{\frac{C_j - r_j}{p_j}\right\}$
$\sum S_j$	average stretch	$\sum_{j=1}^{n} \frac{C_j - r_j}{p_j}$
$L_{\max}$	maximum lateness	$\max_{1\leq j\leq n}\{C_j - d_j\}$

Table 1.1 – Objective functions appearing in this thesis.

AD *give some examples of problems for illustration*

1.3.3 Complexity and Approximation

AD *Give classical results (e.g., $P\,||\,C_{\max}$ is NP-hard)*

AD *Explain reducibility among problems*

AD *Introduce approximation and related results (e.g., Graham's bound $2 - 1/m$)*

Definition 1.1 (Approximation algorithm)**.** *For a given minimization problem with objective function f, a ρ-approximation algorithm $\mathcal{A}$ is a polynomial-time algorithm that, for any instance $\mathcal{I}$ of the problem, returns a solution whose objective value is at most ρ times the optimal objective value, i.e., there exists a constant b such that*

$$f(\mathcal{A}(\mathcal{I})) \leq \rho \cdot f(\mathrm{OPT}(\mathcal{I})) + b,$$

where $\rho \geq 1$, b does not depend on the size of $\mathcal{I}$, and OPT *is an optimal (possibly unknown) algorithm. Then we say that ρ is the* approximation ratio *of $\mathcal{A}$.*

Definition 1.2 (Tightness)**.** *Suppose $\mathcal{A}$ is a ρ-approximation. Then $\mathcal{A}$ is said to be* tight *if there exists an instance $\mathcal{I}$ such that*

$$\frac{f(\mathcal{A}(\mathcal{I}))}{f(\mathrm{OPT}(\mathcal{I}))} = \rho.$$

$\mathcal{A}$ is said to be asymptotically tight *if there exists an instance $\mathcal{I}_n$, parameterized by n, such that*

$$\lim_{n\to\infty} \frac{f(\mathcal{A}(\mathcal{I}_n))}{f(\mathrm{OPT}(\mathcal{I}_n))} = \rho.$$

1.3.4 Online Scheduling

AD *Introduce online algorithms*

Definition 1.3 (Competitive algorithm)**.** *For a given minimization problem with objective function f, a ρ-competitive algorithm $\mathcal{A}$ is a polynomial-time* online *algorithm that, for any instance $\mathcal{I}$ of the problem, returns a solution whose objective value is at most ρ times the optimal objective value, i.e., there exists a constant b such that*

$$f(\mathcal{A}(\mathcal{I})) \leq \rho \cdot f(\mathrm{OPT}(\mathcal{I})) + b,$$

where $\rho \geq 1$, b does not depend on the size of $\mathcal{I}$, and OPT *is an optimal (possibly unknown)* offline *algorithm. Then we say that ρ is the* competitive ratio *of $\mathcal{A}$.*

Definition 1.4 (Lower bound)**.** *Let $\mathcal{P}$ be a minimization problem with objective function f. Then the online version of $\mathcal{P}$ is said to be* bounded *by B if and only if there exists an instance $\mathcal{I}$ such that, for all online algorithms $\mathcal{A}$, we have*

$$\frac{f(\mathcal{A}(\mathcal{I}))}{f(\mathrm{OPT}(\mathcal{I}))} \geq B,$$

where $B \geq 1$ and OPT *is an optimal offline algorithm. Then we say that B is a* lower bound *on the competitive ratio for $\mathcal{P}$.*

AD *Introduce clairvoyance model*
AD *Introduce adversary model*

1.4 Related Work on Scheduling

2

Scheduling Requests in Distributed Key-Value Stores

2.1 Introduction

Modern storage systems are subject to a wide range of challenges, going from easy scalability to efficient and reliable fault-tolerance. Among these challenges, keeping the system responsive under various types of workloads and load imbalance is critical to meet the user demand. As the architecture of distributed key-value stores makes them particularly sensitive to request scheduling, the understanding of how this affects the overall performance is a key factor to improve these systems and keep control over them in highly dynamic and unpredictable environments.

This chapter formally introduces scheduling in distributed key-value stores. We begin to explore request scheduling as a latency-minimization problem. The optimization of the average response time is well-understood, in both stochastic and deterministic settings [35], and key-value stores show excellent empirical results for this specific metric [22]. However, key-value stores are well-known to be subject to the tail latency problem, where a small fraction of slow requests affect the majority of the workload [10]. We choose to model the problem as an optimization problem where the objective is to minimize the maximum response time among requests. In other words, we want to build a schedule where each request is answered within a minimal time span. The first objective is to understand the intrinsic difficulty of the problem. As it is trivially NP-hard, we explore simpler variants in order to understand which constraints make the problem tractable or intractable. By doing so, we identify a tractable variant whose solution gives a lower bound on the optimal objective of the original problem in any case. Finally, we explore request scheduling from a more empirical approach, by simulating a distributed key-value store and comparing several scheduling heuristics on the basis of this common lower bound.

This chapter presents the following contributions:

- We introduce a formal scheduling framework for distributed key-value stores, and we discuss several performance metrics in Section 2.2. This general model is common to all chapters, with some slight variations.
- In Section 2.3, we derive simple optimality and approximation results on relaxed subproblems, such as $P \,||\, \max w_j C_j$, $Q \,|\, p_j = p \,|\, \max w_j C_j$ or $P \,|\, r_j, p_j = p \,|\, F_{\max}$.
- On the basis of an existing algorithm for the preemptive version of the original problem, we propose a computationally tractable lower bound on the optimal objective for any possible instance, and we show that removing the possibility to migrate jobs between machines makes the problem NP-hard in Section 2.4.
- We present a discrete-event simulator for distributed key-value stores, and we compare several scheduling heuristics on the basis of the previously-defined lower bound in Section 2.5. Our results highlight the importance of knowing some workload properties with enough precision to build efficient scheduling heuristics in practice.

2.2 Modeling Key-Value Stores

We propose a formal scheduling framework for key-value stores. Section 2.2.1 describes the application and platform models (i.e., properties of the considered workload and cluster), and we discuss objective criteria in Section 2.2.2.

2.2.1 Application and Platform

Performing the analysis of complex systems is a difficult task, and we need to make simplified assumptions to derive theoretical results. In this section, we describe and explain the choices we make on modeling the cluster and the workload of distributed, replicated and persistent key-value stores as a scheduling problem. Unless stated otherwise, these choices apply throughout the entire thesis.

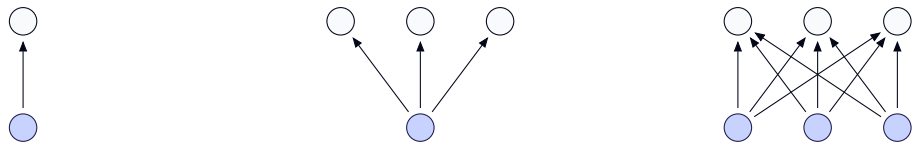
Scheduler model. A *scheduler* is a component that assigns jobs to machines. In *distributed* key-value stores, the cluster is composed of machines that receive and treat client requests. These requests constitute the jobs of the system and are assigned to machines by schedulers. Thus, if we denote the number of schedulers by c and the number of machines by m, we may use three possible analytic frameworks, as illustrated in Figure 2.1:

(i) Single-scheduler, single-machine ($c = 1, m = 1$): this is the simplest model, where the scheduler is a single entity that assigns jobs to a single machine.

(ii) Single-scheduler, parallel machines ($c = 1, m \geq 1$): this is the most common model, where the scheduler is a single entity that assigns jobs to a set of parallel machines.

(iii) Multiple schedulers, parallel machines ($c \geq 1, m \geq 1$): this is the most general model, where several schedulers assign jobs to a set of parallel machines.

The last model corresponds to the most realistic case, as persistent key-value stores are generally leaderless distributed systems where each machine is able to receive and schedule requests in the cluster. However, it is also the most difficult to analyze: as each scheduler only knows a subset of the jobs, we cannot consider the whole set of possible scheduling algorithms without taking into account the additional cost of communication and synchronization between schedulers. Given the difficulty of the last



(a) $c = 1, m = 1$. (b) $c = 1, m \geq 1$. (c) $c \geq 1, m \geq 1$.

Figure 2.1 – The three possible scheduler models. Blue circles represent schedulers, and gray circles represent machines. In the two first models, the single scheduler knows all the jobs of the instance, while in the last model, each scheduler knows only a subset of the jobs. Remark that the first model is a subcase of the second model, which is itself a subcase of the third model.

model, and considering the natural complexity of key-value stores, we choose to focus on the second model, which is by far the most common in the literature on scheduling. Furthermore, this choice is motivated by the fact that the second model is already complex enough to capture the intrinsic difficulty of scheduling in key-value stores, and that we can expect to build upon existing results more easily than with the last model. Nevertheless, we will indeed sometimes consider the last model in this thesis, for instance when approaching the problem with empirical simulations in Section 2.5.

Request model. Each client request contains a *key* that is used to designate a specific *value* (or *data item*) in the store. Thus, processing a *read* request (e.g., `SELECT` operation) consists in sending the corresponding value to the initiator of the request, whereas processing a *write* request (e.g., `INSERT` or `DELETE` operations) consists in updating the stored value corresponding to the key. Note that persistent key-value stores are usually extremely efficient at processing such requests, as the underlying data structure that handles the stored values is optimized for write operations. However, this is not the case for read requests, which take time to transfer data over the network, and occasionally perform costly disk-read operations. In a *replicated* key-value store, each value is duplicated on several machines to ensure fault-tolerance. This has strong implications on the way read requests are processed in the system: as a single read request can be processed by any machine holding a replica of the corresponding value, the scheduler must choose where the request should be handled AD *figure here*. This enables the system to balance the load between machines. On the other hand, write requests must be processed by all machines holding a replica of the value anyway to ensure the eventual consistency of the dataset, which makes scheduling less imperative for them. We focus mainly on scheduling *read* requests in this thesis.

To summarize, a read request—also simply called a *job* throughout this thesis, in order to match with the common terminology of scheduling theory—is a non-preemptive operation (i.e., we cannot interrupt the reading process and resume it later on a potentially different machine) that may be performed by any machine holding a replica of the value corresponding to the specified key. Moreover, there is no notion of precedence relation. More formally, for each job j, we denote its processing time by p_j and we note $\mathcal{M}_j$ the subset of machines that store a replica of the requested value. $\mathcal{M}_j$ is called the *processing set* of j, and we specify the corresponding constraint in the β-field of scheduling problem notation accordingly.

Online model. Key-value stores are real-time, online systems that handle a huge amount of client requests as a non-interruptible stream. In other words, the scheduler does not know the whole set of requests at the beginning of the run, and it discovers the instance as it goes. This is what is called an *online-over-time* model in the literature.

A job j is considered unavailable before its *release date* r_j ($r_j \geq 0$), that is to say, it cannot be scheduled on machines, and its properties are not known in advance by the scheduler. Unless stated otherwise, the model is clairvoyant, i.e., the exact processing time p_j of j becomes known at time r_j. This is a simplified assumption, as the processing time of a request is generally difficult to know with

precision in real systems, although several techniques have been proposed to estimate it (e.g., Bloom filters [20]).

2.2.2 Objective Functions

The usual performance metrics to optimize in a key-value store are:

- The average throughput, i.e., the number of completed requests per unit of time, which should be maximized. This can be expressed as the inverse of the average response time of a request, which means that maximizing the average throughput is equivalent to minimizing the average response time.
- Minimizing the tail latency, which corresponds to the last percentiles in the distribution of response times of the requests. Different percentiles are considered among authors, the most common being the 90th, 95th, 99th or even the 99.9th percentiles.

Minimizing the average response time has been the subject of a huge amount of work, both in the context of distributed systems such as key-value stores and in the context of scheduling theory. Several techniques have been proposed to analyze this metric, from the complexity analysis of related scheduling problems [35, 5, 6, 23] to the design of approximations and competitive online algorithms [4, 34, 21, 26, 28, 9, 15].

On the contrary, few papers have focused on the tail latency minimization from a theoretical point of view in the context of key-value stores. Indeed, expressing the minimization of tail latency is not straightforward in deterministic scheduling, and we propose the following approach. If we take a step back, the final objective is clearly to bound the time that is spent by each request in the system, in order to obtain a quantitative guarantee on the overall response time. A possible approach consists in minimizing the maximum response time among requests, which is a well-known criterion in scheduling theory. The *maximum flow time* $F_{\max} = \max_j \{F_j\}$ constitutes the corresponding objective function, the flow time F_j of a job j being the difference between its completion time C_j and its release time r_j (i.e., $F_j = C_j - r_j$).

Furthermore, as we know that a user's tolerance for the response time of a real system is higher when a process considered to be heavy is in progress, we also propose to weight the flow time to emphasize the relative importance of a given request. Indeed, it seems fair to wait a bit longer for a request for a large value to complete than for a small one, especially if requests for large values are less frequent. To formalize this idea, we associate a weight w_j to each job j, which can be used for adapting the considered performance metric. For instance, setting $w_j = 1$ for all j corresponds to the maximum flow time objective (i.e., $\max w_j F_j = F_{\max}$). This gives an importance to each job that is proportional to its cost, which favors requests for large values. Another possibility would be to set $w_j = 1/p_j$, which represents the maximum *stretch* (or *slowdown*) among requests ($\max w_j F_j = \max F_j/p_j = S_{\max}$). This gives the same importance to each request, but this favors requests for small values because they are more sensitive to scheduling decisions.

2.3 Scheduling Problem and Relaxed Variants

Scheduling problem. In summary, we want to schedule n jobs $J = \{1, 2, \cdots, n\}$ on m parallel, identical machines $M = \{1, 2, \cdots, m\}$ in order to minimize the maximum weighted flow time $\max w_j F_j$ under the following constraints:

Problem	Class	Approximation	Ref.
$1 \mid\mid \max w_j C_j$	P		Theorem 2.1
$P \mid\mid \max w_j C_j$	NP-hard	$2 - 1/m$	Theorem 2.3
$Q \mid p_j = p \mid \max w_j C_j$	P		Theorem 2.2
$1 \mid r_j \mid F_{\max}$	P		[4]
$P \mid r_j, p_j = p \mid F_{\max}$	P		Theorem 2.4
$P \mid r_j \mid F_{\max}$	NP-hard	$3 - 2/m$	[4]

Table 2.1 – Computational complexity of relaxed variants.

- Jobs have heterogeneous processing times, i.e., each job j has an arbitrary processing time p_j.
- Jobs cannot be assigned on all machines, i.e., each job j can be processed by a subset of machines $\mathcal{M}_j$ only.
- Jobs arrive as an unpredictable stream, i.e., each job j has a release time $r_j \geq 0$ and its properties ($\mathcal{M}_j$, r_j, p_j and w_j) are not known before time r_j.
- Jobs cannot be preempted.
- There are no simultaneous executions, i.e., two different jobs cannot be executed at the same time on the same machine.

We note the problem $P \mid \mathcal{M}_j, \textit{online-}r_j \mid \max w_j F_j$ in Graham's notation (with the corresponding offline version denoted by $P \mid \mathcal{M}_j, r_j \mid \max w_j F_j$). A solution to an instance of this problem consists in finding a schedule that assigns a tuple (μ_j, σ_j) to each job j, where μ_j is the executing machine (with $\mu_j \in \mathcal{M}_j$) and σ_j is the starting time of j (with $\sigma_j \geq r_j$). A schedule is a *valid* solution if and only if all jobs are scheduled and all constraints are satisfied.

Variants. The offline problem is trivially NP-hard by reduction to the MAKESPAN problem (solving the problem implies that any instance with $w_j = 1$, $r_j = 0$ and $\mathcal{M}_j = M$ for all j, which corresponds to $P \mid\mid C_{\max}$, can be solved). Thus, finding an optimal solution for any instance is unlikely to be possible in polynomial time. However, we want to go further and study *how* the hardness of the problem evolves when we relax some constraints. The objective is twofold. First, this enables us to identify more precisely the origin of the difficulty of scheduling requests in key-value stores, in order, for instance, to guide the future design of practical efficient heuristics. Second, this gives an intuition on the possibility to derive approximations or computationally tractable lower bounds on the optimal solution. In the following, we consider variants of the problem by relaxing a subset of the following constraints:

- the heterogeneous processing times p_j,
- the processing sets $\mathcal{M}_j$,
- the release dates r_j, and
- the weights w_j.

We summarize existing and new complexity results on relaxed variants in Table 2.1.

We first focus on the problem of minimizing the maximum weighted flow time when all jobs are available at time 0. Remark that in this case, minimizing the maximum weighted flow time is strictly equivalent to minimizing the weighted makespan $\max w_j C_j$. Let us consider the MAXWEIGHT scheduling algorithm (Algorithm 1), which schedules requests by non-increasing order of weights w_j. By a swapping argument, it is easy to show that MAXWEIGHT is optimal on a single server.

Algorithm 1 MaxWeight

Put each job in non-increasing order of w_j on the machine that finishes it the earliest

Theorem 2.1. MaxWeight *(Algorithm 1) solves* $1 \,||\, \max w_j C_j$ *in polynomial time.*

Proof: Let π^{OPT} be an optimal schedule. If all jobs are ordered by non-increasing weight in π^{OPT}, then MaxWeight is optimal. Otherwise, we can find two consecutive jobs j and k in π^{OPT} such that $w_j \leq w_k$. 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 jobs, the contribution becomes $\mathcal{C}' = \max(w_k C'_k, w_j(C'_k + p_j))$ where C'_k is the completion time of job k in this new schedule. 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 by construction, $C'_k + p_j = C_j + p_k$, thus $w_k(C'_k + p_j) = w_k(C_j + p_k) = \mathcal{C}$. Therefore, $\max(w_k C'_k, w_j(C'_k + p_j)) \leq \mathcal{C}$, i.e., $\mathcal{C}' \leq \mathcal{C}$.

It follows that if two consecutive jobs are not ordered by non-increasing weight in π^{OPT}, we can switch them without increasing the objective. By repeating the operation job by job, we transform π^{OPT} in another optimal schedule where jobs are sorted by non-increasing w_j. The conclusion follows. ■

By extending the previous proof, we show that MaxWeight also solves the related case when all jobs have the same homogeneous size p.

Theorem 2.2. MaxWeight *(Algorithm 1) solves* $Q \,|\, p_j = p \,|\, \max w_j C_j$ *in polynomial time.*

Proof: Let π^{OPT} be an optimal schedule with two jobs j and k such that $w_j < w_k$ and where k completes at time $C_j + c$ with $c > 0$ (on any machine). Their contribution is $\mathcal{C} = \max(w_j C_j, w_k(C_j + c)) = w_k(C_j + c)$ because $w_j C_j < w_k C_j < w_k(C_j + c)$.

If we swap j and k, the contribution becomes $\mathcal{C}' = \max(w_k C'_k, w_j(C'_k + c))$ because $p_j = p_k = p$. By construction, $C'_k + c = C_j + c$, i.e., $C'_k = C_j$. We have $w_k C'_k = w_k C_j < w_k(C_j + c)$ and $w_j(C'_k + c) = w_j(C_j + c) < w_k(C_j + c)$. Hence, $\mathcal{C}' < \mathcal{C}$.

It follows that we can transform π^{OPT} in another optimal schedule by swapping repeatedly non-sorted jobs. Then, MaxWeight is optimal because it ensures that if j and k are two jobs such that $w_j \geq w_k$, then k completes after j (i.e., $C_k = C_j + c$ with $c > 0$). ■

However, the result does not extend to the parallel case with arbitrary processing times, as shown by the following example.

Example 2.1. *Consider the following instance with* $m = 2$ *machines and* $n = 3$ *jobs:* $p_1 = 1$, $p_2 = 1$, $p_3 = 2$ *and* $w_1 = 6$, $w_2 = 4$, $w_3 = 3$. MaxWeight *schedules the jobs 1 and 2 on different machines, and job 3 on any machine, for an objective of 9. It is possible to schedule the jobs 1 and 2 on the same machine, and job 3 on the other machine, for an objective of 8.*

Nevertheless, MaxWeight gives a guaranteed approximation of an optimal solution for the parallel case.

Theorem 2.3. MaxWeight *(Algorithm 1) computes a tight* $(2-1/m)$*-approximation for* $P \,||\, \max w_j C_j$.

Proof: The bound has been established by [18] and [**mokhtar2021taming**]. Let us consider a MaxWeight schedule π and an optimal schedule π^{OPT}. Let u be the job for which $w_u C_u = \max w_j C_j$, i.e., the job that reaches the objective in π. Then we remove from π and π^{OPT} all jobs j such that $w_j < w_u$ (it does not change the objective $\max w_j C_j$ in π and can only decrease $\max w_j C_j^{\mathrm{OPT}}$ in π^{OPT}). Let $C^*_{\max}$ denote the optimal makespan when scheduling only the remaining jobs. As π is a list-scheduling (in the

sense of Graham), we have $C_{\max} \leq \left(2 - \frac{1}{m}\right) C^*_{\max}$ [16], where $C_{\max}$ is the completion time of the last job in π (i.e., $C_u = C_{\max}$). Let k be the last completed job in π^{OPT}, such that $C_k^{\mathrm{OPT}} = C_{\max}^{\mathrm{OPT}}$. This makespan is bounded by the optimal one (i.e., $C^*_{\max} \leq C_k^{\mathrm{OPT}}$). Therefore,

$$\begin{aligned}
\max w_j C_j = w_u C_u = w_u C_{\max} &\leq \left(2 - \frac{1}{m}\right) w_u C^*_{\max} \\
&\leq \left(2 - \frac{1}{m}\right) w_u C_k^{\mathrm{OPT}} \\
&\leq \left(2 - \frac{1}{m}\right) \frac{w_u}{w_k} w_k C_k^{\mathrm{OPT}} \\
&\leq \left(2 - \frac{1}{m}\right) \frac{w_u}{w_k} \max w_j C_j^{\mathrm{OPT}}.
\end{aligned}$$

As we removed all jobs weighted by a smaller value than w_u, we have $\frac{w_u}{w_k} \leq 1$ and it follows that $\max w_j C_j \leq (2 - 1/m) \max w_j C_j^{\mathrm{OPT}}$.

We prove that it is asymptotically tight by considering the instance with m machines and $n = m(m-1)+1$ jobs with the following weights and processing times:

- $w_j = W + 1$, $p_j = 1$ for all $1 \leq j < n$,
- $w_n = W$, $p_n = m$.

The job n will be scheduled last by the MaxWeight algorithm, which gives an objective of $\max w_j C_j = (2m-1)W$, whereas an optimal schedule starts this job at time 0 and has an objective of $\max w_j C_j^{\mathrm{OPT}} = m(W+1)$. On this instance, the approximation ratio $(2-1/m)W/(W+1)$ tends to $2-1/m$ as $W \to \infty$, and the conclusion follows. ■

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

It is also known that $P \,|\, r_j, p_j = p \,|\, F_{\max}$ can be solved in polynomial time [37]. As it solves a more general problem, the proposed algorithm is quite complex. We show that FIFO is sufficient to solve $P \,|\, r_j, p_j = p \,|\, F_{\max}$.

Theorem 2.4. FIFO *solves the problem* $P \,|\, r_j, p_j = p \,|\, F_{\max}$ *to optimality.*

Proof: Let OPT be an optimal offline strategy and π^{OPT} an optimal schedule. If jobs are processed by non-decreasing release time on each machine, then OPT corresponds to an execution of the FIFO algorithm: two jobs starting simultaneously on two machines may be allocated on different machines in OPT and FIFO, but it does modify neither their completion time nor the completion times of other jobs. Otherwise, let j and k be two jobs in π^{OPT} such that $r_j \leq r_k$, and where j starts after k ($\sigma_j \geq \sigma_k$). j can be on any machine, as well as k. Thus $\sigma_j + p \geq \sigma_k + p$, and then $C_j \geq C_k$ ($p_j = p_k = p$).

Their contribution to the objective is $\mathcal{F} = \max(C_j - r_j, C_k - r_k) = C_j - r_j$ because $r_j \leq r_k$ and $C_j \geq C_k$. Consider what happens if we swap j and k. Note that swapping is possible as k was originally started first although j is released before k. Their contribution to the maximum flow becomes $\mathcal{F}' = \max(C'_j - r_j, C'_k - r_k)$. By construction, $C'_j = C_k$ and $C'_k = C_j$. We have $C'_j - r_j = C_k - r_j \leq C_j - r_j$ (because $C_j \geq C_k$), and $C'_k - r_k = C_j - r_k \leq C_j - r_j$ (because $r_j \leq r_k$). Hence, $\mathcal{F}' \leq \mathcal{F}$.

It follows that we can transform π^{OPT} in another optimal schedule by swapping repeatedly non-sorted tasks. The conclusion follows. ■

2.4 Bounding the Objective of Each Instance

We have seen that our scheduling problem, with heterogeneous processing times, processing sets, release dates and no preemption ($P \,|\, \mathcal{M}_j, r_j \,|\, \max w_j F_j$) is far from being solvable in reasonable time. However, simplified variants may sometimes become tractable, as shown in the previous section. Even if the computation of an optimal solution in all cases is out of reach, we would still benefit from a lower bound on the optimal objective value of each instance of the general problem, for example to assess the quality of the solutions we obtain by heuristic methods.

For this, a practical approach is to consider a tractable, relaxed subproblem that captures the whole set of solutions of the offline problem. More formally, let $\mathcal{P}$ be the difficult problem $P \,|\, \mathcal{M}_j, r_j \,|\, \max w_j F_j$, and let $\Pi(\mathcal{I})$ be the set of valid solutions for a given instance $\mathcal{I}$. We want to find a problem $\mathcal{P}'$ with the following properties:

(i) the problem $\mathcal{P}'$ is solvable in polynomial time,

(ii) there exists a polynomial-time procedure $\phi_{\mathcal{P}\to\mathcal{P}'}$ that transforms any instance of the problem $\mathcal{P}$ into an instance of the relaxed problem $\mathcal{P}'$, and

(iii) $\Pi(\mathcal{I}) \subseteq \Pi(\phi_{\mathcal{P}\to\mathcal{P}'}(\mathcal{I}))$ for any instance $\mathcal{I}$ of the problem $\mathcal{P}$.

Then, if we note $f(\pi)$ the objective value of a solution π, we necessarily have

$$\min_{\pi\in\Pi(\phi_{\mathcal{P}\to\mathcal{P}'}(\mathcal{I}))} \{f(\pi)\} \leq \min_{\pi\in\Pi(\mathcal{I})} \{f(\pi)\}$$

for any instance $\mathcal{I}$ of the problem $\mathcal{P}$, and we note $\mathcal{L}(\mathcal{I}) = \min_{\pi\in\Pi(\phi_{\mathcal{P}\to\mathcal{P}'}(\mathcal{I}))} \{f(\pi)\}$, which is a computationally tractable lower bound on the optimal objective value of the initial instance $\mathcal{I}$. Then, the relative ratio between the objective value of a solution given by a heuristic $\mathcal{H}$ for the problem $\mathcal{P}$ and $\mathcal{L}(\mathcal{I})$ can be seen as a measure of the quality of $\mathcal{H}$ on $\mathcal{I}$. Ideally, we want this ratio to be close to 1. Of course, the problem $\mathcal{P}'$ must also be chosen carefully, so that the lower bound is as close to the optimal objective value of the initial problem as possible. In other words, we want to choose $\mathcal{P}'$ so that the difference $\min_{\pi\in\Pi(\mathcal{I})} \{f(\pi)\} - \mathcal{L}(\mathcal{I})$ is not too large.

Legrand et al. solved the scheduling problem $R \,|\, r_j, pmtn \,|\, \max w_j F_j$ in polynomial time by expressing the model as a linear program [26]. This problem is very similar to the one we are interested in, as the platform relies on unrelated machines, which generalizes our multipurpose machines environment: by definition, $P \,|\, \mathcal{M}_j \,|\, \max w_j F_j$ is a special case of $R \,||\, \max w_j F_j$ [30]. It only differs on one specific aspect, as it allows preempting and migrating jobs between machines, which we do not permit in our model. In fact, $P \,|\, \mathcal{M}_j, r_j, pmtn \,|\, \max w_j F_j$ is a relaxed version of $P \,|\, \mathcal{M}_j, r_j \,|\, \max w_j F_j$ that meets all the requirements we have stated above: (i) it is solvable in polynomial time through the procedure given by Legrand et al. [26], (ii) the identity function is a trivial transformation procedure between instances of the two problems, and (iii) the set of valid solutions of the preemptive problem necessarily contains all solutions of the non-preemptive problem.

2.4.1 Complexity of the Non-Migratory Variant of the Preemptive Problem

The drawback of using the preemptive version of the problem is that an optimal solution may include job migrations between machines. This seems to be a necessary compromise to make the problem tractable. Indeed, an idea could be to allow local preemption but forbid migration between machines, which would be closer to our initial problem. We define non-migratory preemption as follows.

Definition 2.1. *In a* preemptive non-migratory *schedule, each interrupted job is eventually resumed on the same machine it was initially assigned on.*

However, we show that the non-migratory variant of the preemptive problem is NP-complete. The proof of this result consists in a reduction from the MAKESPAN problem, which is NP-complete [27].

Definition 2.2 (NONMIGRATORY)**.** *Given a set of jobs J, a set of machines M and a bound B, is there a valid preemptive non-migratory schedule where each job j completes before time $r_j + B/w_j$?*

Theorem 2.5. NONMIGRATORY *is NP-complete.*

Proof: We prove the NP-completeness of this problem by reduction from $P\,||\,C_{\max}$, which is well-known to be NP-complete [27]. Obviously, NONMIGRATORY belongs to NP.

Building instance. We consider an instance $\mathcal{I}_1$ of the MAKESPAN problem: given a set of jobs J', a set of machines M' and a bound B', is there a valid non-preemptive schedule where each job completes before time B'? We construct the following instance $\mathcal{I}_2$ of NONMIGRATORY from $\mathcal{I}_1$. We first set $M = M'$ and $B = B'$. For each job $j' \in J'$, we define a job $j \in J$ with processing time $p_j = p_{j'}$, release time $r_j = 0$ and weight $w_j = 1$. $\mathcal{I}_2$ can clearly be constructed in a time that is polynomial in the size of $\mathcal{I}_1$.

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

Assume now that $\mathcal{I}_2$ has a solution π. It means that for each machine i, we know a set $\mathcal{J}_i \subseteq J$ of jobs that are preemptively scheduled on i exclusively (because migration is not allowed in NONMIGRATORY), and $\max_{j\in\mathcal{J}_i}\{C_j\}$ is the makespan of machine i in π. As π is a solution to $\mathcal{I}_2$, $C_j \leq r_j + B/w_j = B$ for all job j (by definition, $r_j = 0$ and $w_j = 1$), and thus, for all machine i, $\max_{j\in\mathcal{J}_i}\{C_j\} \leq B$.

For each job j, we define the associated set of n_j processing intervals as

$$\Lambda_j = \{(\sigma_{j,k}, \delta_{j,k})\}_{1\leq k\leq n_j},$$

where $\sigma_{j,k}$ and $\delta_{j,k}$ respectively denote the starting time and the duration of the k-th processing interval of j. Note that for all j, $\sigma_{j,k} + \delta_{j,k} \leq \sigma_{j,k+1}$ for all $1 \leq k < n_j$. We can build a solution π' to $\mathcal{I}_1$ by removing preemptions from π, i.e., for each machine i, we rearrange the intervals $\bigcup_{j\in\mathcal{J}_i}\Lambda_j$ (without migrating them) such that for all jobs j processed on machine i, $\sigma'_{j,k} + \delta_{j,k} = \sigma'_{j,k+1}$ for all $1 \leq k < n_j$. This is clearly feasible in polynomial time, and as we only permute processing intervals, it does not change the makespan of machines. Therefore, for all machines i,

$$\max_{j\in\mathcal{J}_i}\{C'_j\} = \max_{j\in\mathcal{J}_i}\{C_j\} \leq B = B'. \qquad \blacksquare$$

2.4.2 Optimal Procedure for the Preemptive Problem

The solution to $R\,|\,r_j, pmtn\,|\,\max w_jF_j$ consists in performing a binary search on a linear program [26], followed by the schedule reconstruction scheme from Lawler et al. [25]. For completeness, we explain the full procedure here (the reader may refer to the mentioned references for details about the correctness of this procedure).

The algorithm of Legrand et al. is based on the fact that this problem can be expressed as a deadline scheduling problem. We want to find the minimum objective value f such that, when we fix a deadline $d_j(f) = r_j + f/w_j$ to each job j, we can find a feasible schedule where j is executed during the time interval spanning from r_j to $d_j(f)$.

For any f, we define the ordered set of epochal times $E(f) = \{r_1, \ldots, r_n, d_1(f), \ldots, d_n(f)\}$. Each epochal time $e_t(f)$ has position t ($1 \leq t \leq 2n$) in the ordered set $E(f)$, and let $t_{r_j}(f)$ (resp. $t_{d_j}(f)$) give

the position of the value r_j (resp. $d_j(f)$) in $E(f)$. Adjacent epochal times $e_t(f)$ and $e_{t+1}(f)$ constitute a time interval (of course, for $t = 2n$, the considered interval spans from $e_t(f)$ to $+\infty$).

Observe that the relative ordering of epochal times only changes for specific values of f, i.e., there is an ordered set $\{\lambda_k\} \in 2^{\mathbb{Q}}$ such that, for all k and for any f, g such that $\lambda_k < f < g < \lambda_{k+1}$, the relative ordering of $E(f)$ is the same as the relative ordering of $E(g)$. Each λ_k is called a *milestone* and corresponds to a value f for which one deadline of a given job becomes equal to the release time or the deadline of another job AD *add figure here*.

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 job j coincides with the release time or the deadline of a different job j'. Thus there are two cases to consider:

(i) $d_j(f) = r_{j'}$, i.e., $r_j + f/w_j = r_{j'}$, which implies $f = w_j(r_{j'} - r_j)$, or

(ii) $d_j(f) = d_{j'}(f)$, i.e., $r_j + f/w_j = r_{j'} + f/w_{j'}$, which implies $f = \frac{w_j w_{j'}}{w_{j'} - w_j}(r_{j'} - r_j)$, where $w_j \neq w_{j'}$ (as two deadlines will never coincide if $r_j \neq r_{j'}$ and $w_j = w_{j'}$).

Solving in a milestone interval. Let λ_1 and λ_2 be two consecutive milestones. We want to know if there exists f such that $\lambda_1 \leq f \leq \lambda_2$ and such that there exists a feasible preemptive schedule. In this case, we want to get the minimal one. Let x_{ijt} be the fraction of job j processed by machine i during the time interval spanning from $e_t(f)$ to $e_{t+1}(f)$. Then the Linear Program 2.1 provides a solution:

- we want to minimize the objective value f (Equation (2.1a)),
- each job must be completely processed (Equation (2.1b)),
- the total processing time on a given machine during a time interval cannot exceed its capacity (Equation (2.1c)),
- the processing time of a given job during a time interval cannot exceed its capacity, i.e., a given job cannot be simultaneously executed by several machines (Equation (2.1d)),
- a job cannot be executed before its release time (Equation (2.1e)), and
- a job cannot be executed after its deadline (Equation (2.1f)).

$$\textbf{minimize} \quad f \tag{2.1a}$$

$$\text{subject to} \quad \forall j, \sum_{it} x_{ijt} = 1, \tag{2.1b}$$

$$\forall i, t, \sum_{j} x_{ijt} p_{ij} \leq e_{t+1}(f) - e_t(f), \tag{2.1c}$$

$$\forall j, t, \sum_{i} x_{ijt} p_{ij} \leq e_{t+1}(f) - e_t(f), \tag{2.1d}$$

$$\forall i, j, t, x_{ijt} = 0 \text{ if } t_{r_j}(f) > t, \tag{2.1e}$$

$$\forall i, j, t, x_{ijt} = 0 \text{ if } t_{d_j}(f) \leq t, \tag{2.1f}$$

$$\lambda_1 \leq f \leq \lambda_2 \tag{2.1g}$$

Schedule reconstruction. 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 optimal solution f.

Let us assume that we are considering the t-th time interval (spanning from $e_t(f)$ to $e_{t+1}(f)$). We will repeat the same procedure for all intervals, and simply concatenate the partial schedules. First, we build the $m \times n$ cost matrix A such that $A_{ij} = x_{ijt}p_{ij}$ for each row i and column j, which represents the fraction of job j that should be executed on machine i during the current time interval t. The procedure is to build iteratively the schedule by choosing a set $\mathcal{D}$ 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., $B = \sum_k c_k P_k$, where each c_k is a coefficient and P_k is a $(m+n) \times (m+n)$ permutation matrix. The top-left $m \times n$ block of any P_k gives a decrementing set $\mathcal{D}$ to schedule in the current iteration: if P_k has a 1 on row i and column j, then $(i,j) \in \mathcal{D}$, which means that the job j may be executed by the machine i in time interval t.

We now compute the maximum processing time δ allowed during the current iteration. We denote a row i 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 j. δ is chosen to be maximum subject to the following constraints:

- for each element $(i,j) \in \mathcal{D}$ such that row i or column j is tight, $\delta \leq A_{ij}$;
- for each element $(i,j) \in \mathcal{D}$ such that row i is slack, $\delta \leq A_{ij} + (e_{t+1}(f) - e_t(f)) - \sum_k A_{ik}$;
- for each element $(i,j) \in \mathcal{D}$ such that column j is slack, $\delta \leq A_{ij} + (e_{t+1}(f) - e_t(f)) - \sum_k A_{kj}$;
- for each row i that contains no element of $\mathcal{D}$, $\delta \leq (e_{t+1}(f) - e_t(f)) - \sum_k A_{ik}$;
- for each column j that contains no element of $\mathcal{D}$, $\delta \leq (e_{t+1}(f) - e_t(f)) - \sum_k A_{kj}$.

When δ is found, for each element $(i,j) \in \mathcal{D}$, j is scheduled on i as soon as possible for $\min(\delta, p_{ij})$ time units, and A_{ij} is replaced by $\max(0, A_{ij} - \delta)$ in A. Then we proceed to the next iteration, and we repeat the procedure until all elements of A are equal to 0. When all iterations are done, we have a schedule for the time interval t, and we proceed to the next interval. Finally, we concatenate all partial schedules to obtain the complete schedule.

2.5 Simulating a Key-Value Store

We mimic a distributed, replicated and persistent key-value store through a discrete-event simulator, whose architecture is described in Section 2.5.1. Then, we design and analyse the behavior of scheduling heuristics (Section 2.5.2), and we compare them with each other in simulations (Section 2.5.3) on the basis of the previously-developed lower bound.

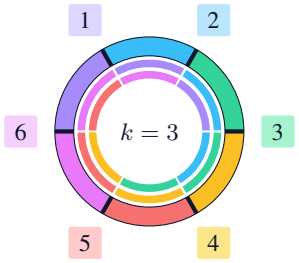


Figure 2.2 – A cluster where processing sets follow a fixed-size interval structure. The ring represents the dataset, splitted over the machines of the cluster. Each colored segment represents a partition of data that is stored on a particular machine and replicated on the two consecutive neighbors. For instance, the red partition is stored on machine 5 and replicated on machines 6 and 1.

2.5.1 Architecture of the Discrete-Event Simulator

The discrete-event simulator is built on Python 3.8 and the salabim package[1] (v21.0.1), which provides advanced features to model and simulate dynamic systems.

The essence of a key-value store consists in the following components:

- *clients*, which send requests to the key-value store,
- *coordinators*, which receive client requests and send them to replicas in the cluster,
- *replicas*, which execute client requests after receiving them from coordinators.

This constitutes the basis of the simulator. In a typical scenario, a client generates a request, which is randomly sent to a coordinator. Then, the coordinator selects a replica holding the requested value, and pushes the request to the local operation queue of this replica (the *replica selection* step). Finally, the replica dequeues local operations and eventually executes the request (the *local execution* step).

Moreover, we consider from now that each processing set $\mathcal{M}_j$ follows a typical replication strategy of key-value store implementations, that is, the cluster is a circular, ordered set of machines, and each set $\mathcal{M}_j$ can be seen as an interval of machines of size k. In other words, $\mathcal{M}_j = \{(i + x) \bmod m\}_{0 \leq x < k}$ for all jobs j, where i is the first machine storing the value that is requested by j. We show an example of this structure in Figure 2.2.

As requests mostly consist in reading and sending bytes over the network, the processing time of jobs is modeled as a linear function of the size of the requested value, i.e., $p_j = \tilde{B} z_j + L$ for all jobs j, where $\tilde{B}$ is the inverse of the network bandwidth, L is the average network latency, and z_j is the number of bytes dedicated to the storage representation of the requested value.

2.5.2 Scheduling Heuristics

We consider several scheduling heuristics with different levels of knowledge about the cluster state. Some of these levels are hard to achieve in a real system. For instance, information about the load of a given machine will often be slightly out of date. Similarly, the processing times of jobs are not exact, as the size of the requested value is cannot be always known by the coordinator for large datasets. Practical systems generally employ an approximation, e.g., by keeping track of size *categories* of values using Bloom filters [20]. 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 replica selection heuristics.

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

Random. The replica is chosen uniformly at random among compatible machines: $r = \operatorname{rand} \mathcal{M}_j$. This strategy does not need particular information.

LeastOutstandingRequests (LOR). Let us define $\textsc{Out}_u(i)$ to be the number of outstanding requests sent from the coordinator u to machine i, i.e., the number of sent requests that received no response yet. The chosen replica minimizes $\textsc{Out}_u(i)$: $r = \arg\min_{i \in \mathcal{M}_j} \textsc{Out}_u(i)$. 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 [39].

Héron. We also consider an omniscient version of the heuristic used by Héron [20]. 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 machines that are *available* according to the scoring method of C3 [39]. The threshold is chosen according to the wanted proportion of large requests in the workload.

EarliestFinishTime (EFT). Let $\textsc{Avail}(i)$ denote the earliest time when the machine i becomes available, i.e., the time at which it will have emptied its execution queue. The chosen replica is the one with minimum $\textsc{Avail}(i)$ among compatible machines: $r = \arg\min_{i \in \mathcal{M}_j} \textsc{Avail}(i)$. Knowing Avail is hard in practice, because it assumes the existence of a mechanism to obtain the exact current load of a machine. A real system would use a degraded version of this heuristic.

EFT-Sharded (EFT-S). In this heuristic, we specialize machines. Small machines execute only requests for small values, and large machines execute all requests for large values and some requests for small values when possible (similarly to size-aware sharding technique [12]). Each request for a large value is scheduled on large machines using the EFT strategy, while each request for a small value is scheduled on any machine (small or large), also using EFT.

For the following experiments, we define large machines as the set of machines i such that $i \bmod k = 0$ (recall k is the replication factor). This makes sure that one machine in each processing set $\mathcal{M}_j$ is capable of treating requests for large values, as each $\mathcal{M}_j$ is an interval of size k. We define a threshold parameter ω to distinguish between requests for small and large values: requests with duration larger than ω are treated by large machines only, while others can be processed by all available machines.

We derive the threshold ω from the size distribution. In the best case, when all machines in each processing set are perfectly balanced, requests for small values are scheduled on small machines only and requests for large values on large machines only. It means that the total work is k times larger than the work on large machines on average. Let X be the random variable that models the size distribution, and f_X denote its probability density function. We denote by $p(X) = \tilde{B}X + L$ the duration of the corresponding request (where $\tilde{B}$ is the inverse of the network bandwidth and L the average network 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 machines when a request is submitted is

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

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

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

This heuristic has to be able to distinguish requests for small and large values with respect to ω. This could be achieved in practice with combined Bloom filters, in a similar fashion than Héron [20].

StaticWindow (SW). Requests are no longer scheduled on reception, but every q time units, where q is a parameter of the heuristic. The set Q denotes the requests received during this window of q time units. Let t be the time at which requests from Q must be scheduled (requests with $t < r_j \leq t + q$ form the next batch and must be scheduled at time $t + q$). We assume here a *centralized* system, where a unique coordinator receives and schedules all requests. The underlying idea is to be able to make choices based on more information on the workload than previous greedy heuristics. This strategy must therefore also decide the order in which the requests of Q are scheduled. We derive two versions.

Sufferage-SW (SSW). The Sufferage heuristic [31] inspired this strategy. Let $\mathcal{F}$ be the function giving the estimated weighted flow $\mathcal{F}(i,j) = w_j(\max(r_j, \text{Avail}(i)) + p_j - r_j)$ of job j when scheduled on machine i as soon as possible. Let $\rho(j) = \arg\min_{i \in \mathcal{M}_j} \mathcal{F}(i,j)$ be the *best* machine for j, i.e., the one minimizing its weighted flow, and $\rho'(j) = \arg\min_{i \in \mathcal{M}_j \setminus \rho(j)} \mathcal{F}(i,j)$ be the *second best* machine for j. Then, we define the sufferage value

$$\text{Suf}(j) = \mathcal{F}(\rho'(j), j) - \mathcal{F}(\rho(j), j) > 0$$

as the difference of weighted flow values on $\rho'(j)$ and $\rho(j)$. The request we choose to schedule is the one which suffers the most if we schedule it on its second best machine: $s = \arg\max_{j \in Q} \text{Suf}(j)$. The chosen replica is $\rho(s)$: $r = \rho(s) = \arg\min_{i \in \mathcal{M}_j} \mathcal{F}(i,s)$.

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

Algorithm 2 Sufferage-SW

```
1:  repeat every q time units
2:      for all j ∈ Q do
3:          ρ(j) ← arg min_{i∈M_j} F(i,j)
4:          ρ'(j) ← arg min_{i∈M_j \ ρ(j)} F(i,j)
5:          Suf(j) ← F(ρ'(j), j) − F(ρ(j), j)
6:      while Q is not empty do
7:          s ← arg max_{j∈Q} Suf(j)
8:          Schedule s on ρ(s)
9:          Q ← Q \ {s}
10:         Update ρ, ρ' and Suf
```

MaxMin-SW (MSW). This strategy is inspired from the Max-Min heuristic [31]. We build a matrix Mat whose rows are machines and columns are requests from Q, where

$$\text{Mat}[i,j] = \begin{cases} \mathcal{F}(i,j) & \text{if } i \in \mathcal{M}_j, \\ +\infty & \text{otherwise.} \end{cases}$$

The best weighted flow of request j is $\mathcal{F}_{best}(j) = \mathcal{F}(\rho(j), j) = \min_{i \in M} \text{Mat}[i,j]$. Then, we schedule the request s whose *best* objective value is the highest: $s = \arg\max_{j \in Q} \mathcal{F}_{best}(j)$. The chosen replica minimizes the objective value of s: $r = \arg\min_{i \in M} \text{Mat}[i,s]$.

The request s is then removed from the set Q, as well as the related column in the matrix Mat, and the row r is updated with new values. These operations are repeated until Q is empty (see Algorithm 3). This strategy runs in time $O(n^2 m)$ and uses a space $O(nm)$ per time window.

Algorithm 3 MaxMin-SW

```
1: repeat every q time units
2:     for all i ∈ M do
3:         for all j ∈ Q do
4:             if i ∈ M_j then
5:                 Mat[i, j] ← F(i, j)
6:             else
7:                 Mat[i, j] ← +∞
8:     while Q is not empty do
9:         s ← arg max_{j∈Q} F_best(j)
10:        r ← arg min_{i∈M} Mat[i, s]
11:        Schedule s on r
12:        Q ← Q \ {s}
13:        Remove column s from Mat
14:        Update row r in Mat
```

Heuristic	Knowledge	Type	Complexity
Random	None	Distributed	$O(1)$
LOR	Ack	Distributed	$O(m)$
Héron	Ack, $p_j \geq \omega$	Distributed	$O(m)$
EFT	Avail	Distributed	$O(m)$
EFT-S	Avail, $p_j \geq \omega$	Distributed	$O(m)$
SSW	Avail, p_j, r_j	Centralized	$O(n^2m)$
MSW	Avail, p_j, r_j	Centralized	$O(n^2m)$

Table 2.2 – Properties of replica selection heuristics. Ack denotes the need to acknowledge the completion of sent requests. Avail is the knowledge of available times of each server. p_j denotes the processing times of local requests and r_j their release times. n is the number of requests in Q and m is the total number of servers.

Heuristic	Knowledge	Complexity
FIFO	None	$O(1)$
MWF	p_j, r_j	$O(N)$

Table 2.3 – Properties of local scheduling heuristics. p_j denotes the processing times of local requests and r_j their release times. N is the number of local requests in $\mathcal{Q}$.

Table 2.2 summarizes the properties of our selection heuristics. 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 machine is responsible for a limited number of keys. We consider the following local policies.

FirstInFirstOut (FIFO). This strategy is commonly used as a local scheduling policy in key-value stores (e.g., Apache Cassandra [24]). 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_j) is the first to be executed.

MaxWeightedFlow (MWF). We propose another strategy, which reorders requests. When the machine becomes available at time t, the next request s to be executed is the one whose weighted flow is the highest: $s = \arg\max_{j \in \mathcal{Q}} w_j(t + p_j - r_j)$. We consider that p_j is always known, as the request j necessarily looks for a value that is hosted on the local machine. Consequently, we know the size of the value, and the request processing time can be estimated accordingly. MWF is a general execution policy that considers the weights w_j as defined by the heuristic 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 ($w_j = 1/p_j$), MWF is equivalent to the strategy that selects the request with maximal stretch. This favors requests for small values in front of requests for large ones, and thus may be a way to mitigate the problem of head-of-line blocking. Table 2.3 summarizes the properties of our local heuristics.

2.5.3 Empirical Results

Settings. We design a synthetic heterogeneous workload to evaluate our heuristics. The sizes of data items follow a Weibull distribution with scale $\eta = 32\,000$ and shape $\theta = 0.5$, which gives an average value size of 64 kilobytes (with standard deviation of 143 kB and median of 15 kB). These parameters yield a long-tailed distribution that is consistent with existing file sizes characterizations [14]. Client requests arrive at coordinators according to a Poisson process with arrival rate $\lambda = m\mathcal{L}/\overline{p}$, where m is the number of machines, $\mathcal{L}$ is the wanted average load (defined as the average fraction of time spent by machines on serving requests), and $\overline{p}$ is the average processing time of requests. Hence, release times are chosen such that the time between two consecutive arrivals follows an exponential distribution with parameter λ. Each key has the same probability of being requested, i.e., we do not model skewed popularity. In other words, processing sets $\mathcal{M}_j$ are chosen with uniform probability. The cluster consists of $m = 15$ machines and we set the replication factor to $k = 3$, which is a common configuration in real implementations [24, 11]. The network bandwidth is set to 100 Mbps ($1/\tilde{B} = 12.5 \cdot 10^6$) and the average latency is set to 1 ms ($L = 10^{-3}$). Note that the number of requests directly depends on the arrival rate λ and the duration of the simulation. For instance, a simulation running over 120 seconds on 15 machines with a 90% average load and an average service time of 6.12 ms yields about 250 000 requests in total.

For the threshold between requests for small and large values, we plug the density function of our

Weibull distribution in Equation (2.2) and solve it numerically for ω:

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

This yields a threshold of 26.4 ms (for a value size of 318 kB), resulting in a proportion of 5% of requests for large values in the workload. Each experiment is repeated on 10 different scenarios. A given scenario defines the processing times p_j, the release times r_j, and the processing sets $\mathcal{M}_j$ according to described settings.

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

- $w_j = 1$ for all jobs j. This is the classic flow time (or latency) metric.
- $w_j = 1/p_j$. 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.
- $w_j = 1/\sqrt{p_j}$. Although the stretch metric is more fair than latency, we noted in some experiments that it tends to be inappropriate under heterogeneous workloads where the majority of requests are small. Small requests are too favored. For instance, if a small request of 1 ms and a large request of 100 ms have a stretch value of 2, then the large request can tolerate a 100 ms delay ($F_j = 200$), whereas the small one can only tolerate a 1 ms delay ($F_j = 2$). 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*.

Results. Figure 2.3 shows Empirical Cumulative Distribution Functions (ECDF) of the flow, the stretch and the weak stretch, for each combination of *distributed* selection heuristic and local execution strategy. The dashed horizontal lines respectively represent median, 95th and 99th percentile. Data items are requested with a load $\mathcal{L} = 0.9$, and the simulations run for 120 seconds.

We show in Figure 2.4 the ECDF of window-based strategies when machines are subject to a burst, i.e., the arrival rate is very high and the average load is greater than 1. We measure the metrics with average load values of 1 and 3, combined to a FIFO execution. For SSW and MSW, we consider the stretch weighting ($w_j = 1/p_j$), 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 100 ms. The simulations run over 3 seconds in order to simulate a short burst of requests.

Figure 2.5 shows the 99th percentile of each metric as a function of average server load for each combination of selection and execution heuristics and for load values ranging from 0.5 to 0.9. In this context, the maximum of the distribution is impacted by rare events of varying amplitude, which makes this criterion unstable. The stability of the 99th percentile allows comparing more confidently the performance between scenarios with identical settings. For the local execution policy MWF, we discard the case $w_j = 1/\sqrt{p_j}$, as it exhibits performance always worst than the case $w_j = 1/p_j$. The simulations run for 120 seconds.

The comparison of online heuristics with the lower bound introduced in Section 2.4 is shown in Figure 2.6. We normalize the maximum objective $\max w_j F_j$ generated by a given heuristic with the lower bound. Each boxplot[2] represents the distribution of these normalized maximums among 10 different

[2] A boxplot consists of a bold line for the median, a box for the quartiles and whiskers that extend at most to 1.5 times the interquartile range from the box.

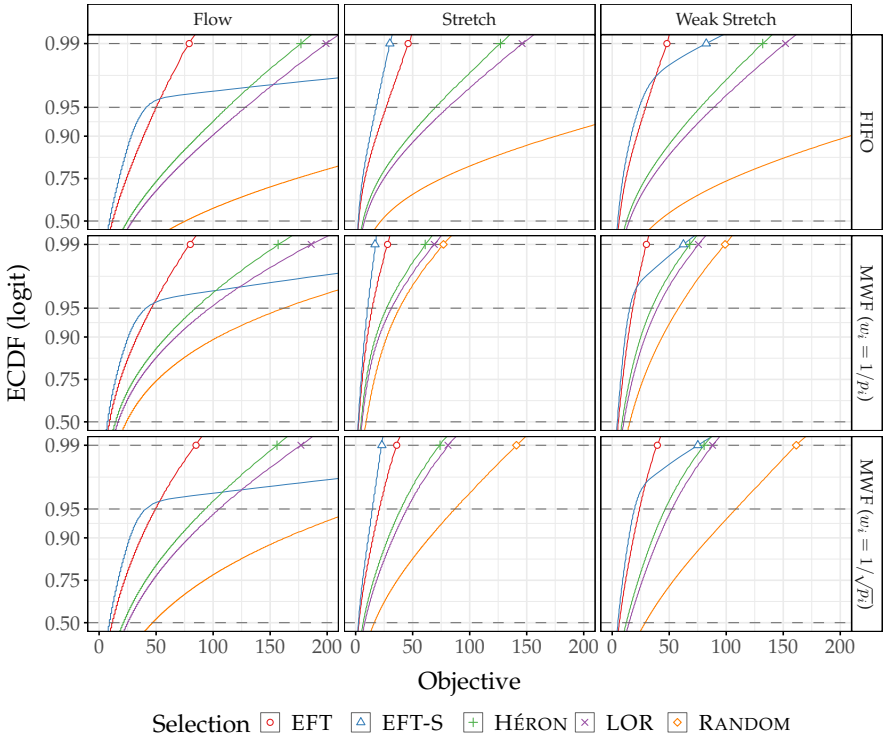


Figure 2.3 – ECDF of flow, stretch and weak stretch metrics given by each combination of distributed selection and execution heuristics in steady-state over 120 seconds, under average load of 90%.

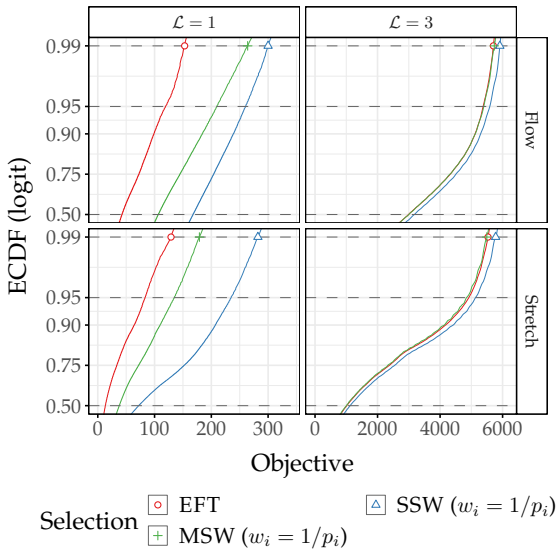


Figure 2.4 – ECDF of flow and stretch metrics given by centralized heuristics SSW and MSW combined to a local FIFO execution in a burst over 3 seconds, under average loads of 100% and 300%.

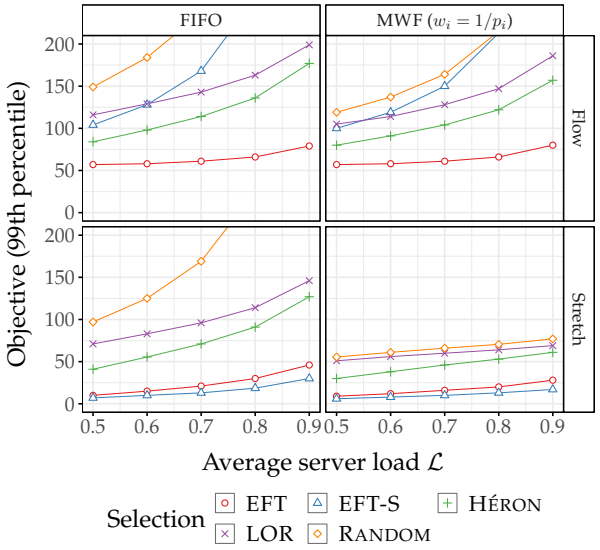


Figure 2.5 – 99th quantile of flow and stretch metrics for each combination of selection/execution heuristics in steady-state over 120 seconds, under average loads ranging from 50% to 90%.

scenarios, for each combination of strategies. Horizontal red bars help to locate the lower bound. Data items are requested with a load $\mathcal{L} = 0.9$, and the 10 scenarios are solved over 1200 requests.

The first thing to note in Figures 2.3 to 2.6 is that the choice on replica selection heuristic is indeed critical for read latency, as the 99th quantile can often be improved by a factor 2 compared to state-of-the-art strategies LOR and HÉRON, without increasing median performance as confirmed in Figure 2.3. 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 (Figure 2.6).

Figure 2.6 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 Figure 2.5.

For the stretch metric, where latencies are weighted by processing times, EFT-S performs even better than EFT (Figures 2.3 and 2.5), yielding a 99th quantile of 30 (resp. 18) when coupled with FIFO (resp. MWF ($w_j = 1/p_j$)). 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. Figures 2.3, 2.5 and 2.6 also illustrate the significant impact of local execution policies on the stretch metric: local reordering according to MWF ($w_j = 1/p_j$) favors requests for small values, which results in an improvement for all selection strategies, even on the median values. Note that this does not necessarily improve latency, as FIFO is well-known to be the optimal strategy for max-flow on a single machine [4]. It is confirmed by our observations, as MWF worsen the tail latency.

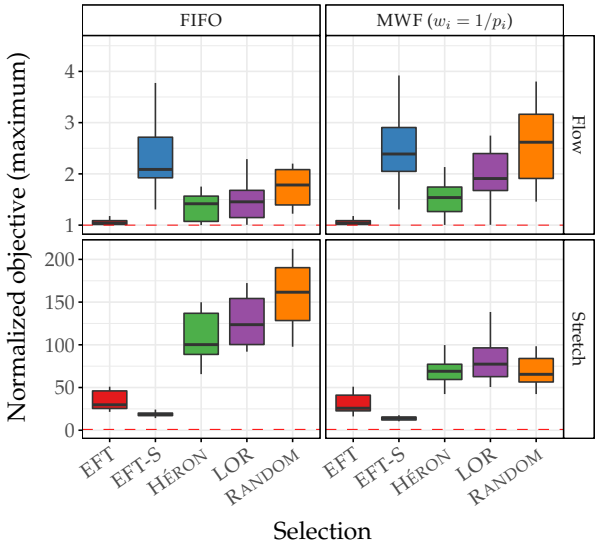


Figure 2.6 – Distributions of normalized flow maximums and stretch maximums for each combination of selection/execution heuristics. Data items are requested with a load of 90%, and the 10 different scenarios consist of 1200 requests.

When a burst occurs, Figure 2.4 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 outperform EFT. This seems to confirm that EFT is a close-to-optimal strategy in average, as additional information do not allow to increase performance.

2.6 Conclusion

In this chapter, we have introduced the intrinsic scheduling problem of distributed, replicated and persistent key-value stores. We modeled the problem as a latency-minimization problem, the final goal being to bound the time that is spent by each request in the system. The general problem is NP-hard, which means that we cannot expect to find an optimal solution in a reasonable time for all possible instances. Thus, we studied simplified variants of the offline problem, in order to better understand the role of each constraint in its overall difficulty. This enabled us to identify a relaxed, computationally tractable version that constitute a lower bound on the optimal objective value for any instance. Finally, we proposed a set of scheduling heuristics that are evaluated through discrete-event simulations and properly compared on the basis of the theoretical lower bound.

From a theoretical point of view, future work includes devising stronger guarantees on the lower bound. In our work, we only tested its quality in an empirical manner, but it is still unknown if the difference between its value and the optimal solution is small enough for any instance. On the practical side, our key-value store simulator turned out to be a useful tool for evaluating scheduling heuristics without the need to implement them in a real system. Moreover, this enables to keep control on a lot of parameters that are very difficult to handle in key-value stores (e.g., perfect accuracy on various information). This simulator could be extended to support skewed key access patterns, dynamic replication, multi-get

requests, dynamic cluster scalability, random faults, etc.

3

Bounds and Inapproximability under Replicated Datasets

3.1 Introduction

In replicated key-value stores, each data item is duplicated on several machines to ensure accessibility of the data in case of machine failures. This unlocks the possibility to balance read operations among the machines that hold a copy of the requested keys. In this chapter, the main question is how the replication strategy (i.e., the way data items are replicated on machines) impacts the potential guarantees we may obtain on system performance.

We begin by recalling the scheduling problem we are interested in, with a focus on the specific constraint that models the replication strategy, that is to say, the processing sets $\mathcal{M}_j$ defined for each job j. We introduce a hierarchy of processing set structures (which correspond to different replication strategies), from the most general one, which exhibits no particular property, to more restrictive structures such as fixed-size intervals or disjoint sets (Section 3.2). Then, we derive results on the online response time (expressed as the maximum flow time objective) through competitive analysis, for three particular classes of algorithms: (i) general online algorithms, which discover the instance as they run, (ii) immediate dispatch algorithms, which are a subset of online algorithms that schedule jobs as soon as they are released, and (iii) EarliestFinishTime algorithms, which are a subset of immediate dispatch algorithms that systematically schedule jobs on the machine that finishes the earliest (with specific tie-break strategies). We show that structured processing sets affect the attainable competitive ratio for these three categories (Section 3.3). Finally, we focus on the influence of different replication strategies on the average throughput that is achievable by the system, under a given distribution on the access frequencies of the data items. We develop a general method that computes the theoretical maximum achievable throughput of a system, for any given replication strategy and key access frequencies. By applying this method to some examples,

Figure 3.1

we show that the replication strategy has also a significant impact on the achievable throughput, and we validate our results through simulations (Section 3.4).

3.2 Scheduling Problem

We recall that our scheduling problem, formally defined in Chapter 2, consists in scheduling n jobs $J = \{j\}_{1 \leq j \leq n}$ on m identical machines $M = \{i\}_{1 \leq i \leq m}$ in order to minimize the maximum weighted flow time under the following set of constraints:

- jobs have heterogeneous processing times,
- jobs have processing set restrictions,
- jobs have release times,
- jobs arrive as an online stream and their properties are not known in advance,
- jobs cannot be preempted,
- there are no simultaneous executions on a given machine.

This problem is expressed as $P \,|\, \mathcal{M}_j, \mathit{online}\text{-}r_j \,|\, \max w_j F_j$. Up to now, we considered simplified offline variants only, that is to say, some constraints were relaxed in order to make the problem easier to solve or analyze. In particular, we discarded the constraint that prevents jobs to be processed on any machine, and the constraint that jobs arrive as an unpredictable stream. These two constraints add a lot of difficulty to the problem, and we focus on them in this chapter.

3.2.1 Hierarchy of Processing Set Structures

In key-value stores, data items are replicated on several machines according to a *replication strategy*. This means that a given request can be processed by a subset of the machines only, which correspond to the subset of machines storing a replica of the requested data item. More formally, we denote this subset by $\mathcal{M}_j$ (with $\mathcal{M}_j \subseteq M$) for each job j, and we call $\mathcal{M}_j$ the *processing set* of j.

In the general case, each processing set is arbitrarily defined. In other words, there is no *structure* in the construction of the subsets $\mathcal{M}_j$. This makes the problem particularly difficult, as illustrated by the results of Anand et al. [1], which prove that there is a lower bound of $\Omega(m)$ on the competitive ratio of any online algorithm for $P \,|\, \mathcal{M}_j, \mathit{online}\text{-}r_j \,|\, F_{\max}$. However, several authors introduced variants where the subsets $\mathcal{M}_j$ are not arbitrarily defined anymore [30, 29], as realistic applications often exhibit specific structures in the processing sets. For instance, a common replication strategy in existing key-value stores consists in placing machines on a clockwise virtual ring and replicating the dataset of a given machine on its two direct neighbors [11, 24]. In this case, each processing set can be seen as an interval of three consecutive machines, as illustrated by Figure 3.1.

In the following, we give the structures that are commonly used throughout the literature and for which we derive results in the rest of this chapter.

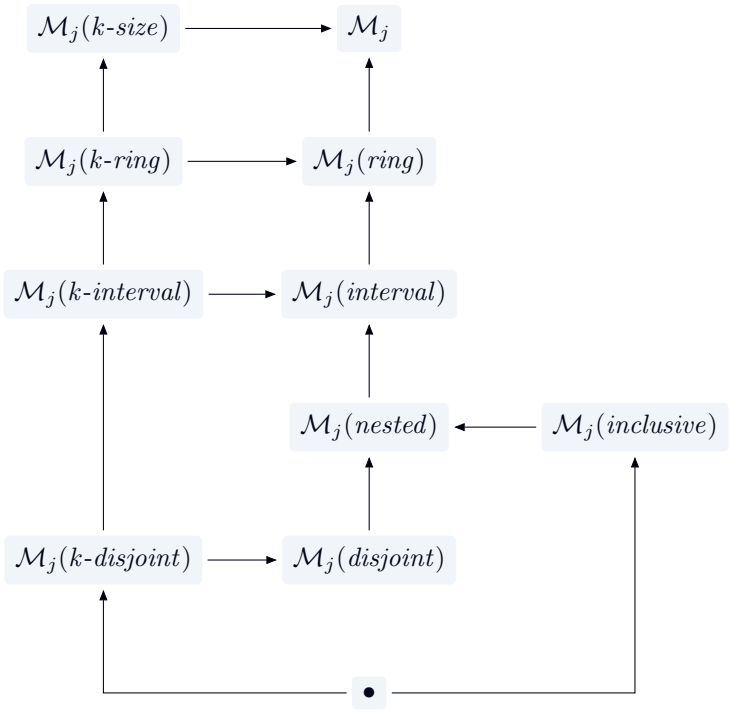


Figure 3.2 – Reduction graph of processing set restrictions defined in Definition 3.1, where $A \rightarrow B$ means that A is a special case of B. The constraint denoted by $\bullet$ corresponds to no processing set restriction, i.e., each job can be processed by any machine.

Definition 3.1. *For each processing set structure $\langle struct \rangle$, we denote the corresponding processing set restriction by $\mathcal{M}_j(\langle struct \rangle)$ in Graham's classification.*

$\mathcal{M}_j$(k-size). *All processing sets consist of k arbitrary machines, i.e., for all jobs j, $|\mathcal{M}_j| = k$.*

$\mathcal{M}_j$(nested). *For all jobs j, j' such that $j \neq j'$, either $\mathcal{M}_j \subseteq \mathcal{M}_{j'}$, $\mathcal{M}_{j'} \subseteq \mathcal{M}_j$, or $\mathcal{M}_j \cap \mathcal{M}_{j'} = \emptyset$.*

$\mathcal{M}_j$(inclusive). *For all jobs j, j' such that $j \neq j'$, either $\mathcal{M}_j \subseteq \mathcal{M}_{j'}$ or $\mathcal{M}_{j'} \subseteq \mathcal{M}_j$.*

$\mathcal{M}_j$(disjoint). *For all jobs j, j' such that $j \neq j'$, either $\mathcal{M}_j = \mathcal{M}_{j'}$, or $\mathcal{M}_j \cap \mathcal{M}_{j'} = \emptyset$.*

$\mathcal{M}_j$(k-disjoint). *Identical to $\mathcal{M}_j(disjoint)$, with the constraint that $|\mathcal{M}_j| = k$ for all jobs j.*

$\mathcal{M}_j$(ring). *All processing sets consist of an interval (possibly circular) of machines, i.e., for all jobs j, $\mathcal{M}_j = \{i \in M \text{ s.t. } a_j \leq i \text{ and } i \leq b_j\}$, for two arbitrary machines a_j, b_j.*

$\mathcal{M}_j$(k-ring). *Identical to $\mathcal{M}_j(ring)$, with the constraint that $|\mathcal{M}_j| = k$ for all jobs j.*

$\mathcal{M}_j$(interval). *Identical to $\mathcal{M}_j(ring)$, with the constraint that $a_j \leq b_j$ for all jobs j.*

$\mathcal{M}_j$(k-interval). *Identical to $\mathcal{M}_j(interval)$, with the constraint that $|\mathcal{M}_j| = k$ for all jobs j.*

Some structures are more general than others, and there exist reduction relationships between them. For instance, $\mathcal{M}_j(inclusive)$ is clearly a particular case of $\mathcal{M}_j(nested)$, which is itself a particular case of $\mathcal{M}_j(interval)$, because it is always possible to reorder the machines in each nested subset $\mathcal{M}_j$ to obtain contiguous intervals of machines. Figure 3.2 gives the reduction graph of all previously-defined structures.

3.2.2 Online Model

In this chapter, we focus on an online-over-time model, that is to say, jobs arrive as an unpredictable stream and their properties are unknown until they are released. Without loss of generality, we assume that jobs are numbered such that $j < j'$ implies $r_j \leq r_{j'}$ for any two jobs j, j'.

We recall that for a given instance of the problem, a schedule $\pi^{\mathcal{A}}$ built from a scheduling algorithm $\mathcal{A}$ assigns a machine $\mu_j^{\mathcal{A}}$ and a starting time $\sigma_j^{\mathcal{A}}$ to each job j of the instance. In addition, we denote the *scheduling time* of a job j under $\mathcal{A}$ by $\varrho_j^{\mathcal{A}}$, that is to say, the exact time at which $\mathcal{A}$ chooses a machine and a starting time for the job j. This new property makes sense for online algorithms, as they build the schedule over time as jobs arrive in the system. This also enables to define a specific class among online algorithms, called *immediate dispatch* algorithms, which are of particular importance in real-time distributed systems such as key-value stores.

Definition 3.2 (Immediate dispatch algorithm)**.** *An online algorithm $\mathcal{A}$ is said to be an* immediate dispatch *algorithm if and only if it schedules jobs as soon as they are released, i.e., $r_j \leq \varrho_j^{\mathcal{A}} \leq r_j + \varepsilon$ for all jobs j, where $0 \leq \varepsilon \ll 1$.*

These immediate dispatch algorithms provide several benefits. First, they increase the scalability of the application, as they avoid the need to handle waiting queues, which can become very large in high-throughput systems. Second, they are easier to implement in distributed systems, as they often avoid synchronization and communication between multiple schedulers.

3.3 Theoretical Bounds on the Response Time

We give lower and upper bounds on the competitive ratio of three classes of algorithms under specific processing set structures. Section 3.3.1 introduces lower bounds for general online algorithms. Then, Section 3.3.2 presents lower bounds for immediate dispatch algorithms. Finally, Sections 3.3.3 and 3.3.4 focus on EarliestFinishTime-like algorithms. Table 3.1 summarizes the results of this section.

3.3.1 Lower Bounds for Online Algorithms

We begin with general lower bounds for online algorithms. Our first result shows that no online algorithm can be optimal when minimizing the maximum weighted flow time on a single machine even when jobs are unitary, that is to say, the competitive ratio of any online algorithm is at least $1 + \varepsilon$ with $\varepsilon > 0$. The proof is based on an adversary that builds a problematic instance.

Theorem 3.1. *No online algorithm can be optimal for* $1 \mid \textit{online-}r_j, p_j = 1 \mid \max w_j F_j$.

Proof: First, let us consider 10 jobs with the following weights and release times:

- $w_j = 3$, $r_j = j - 1$ for all j such that $1 \leq j \leq 8$,
- $w_9 = 1$, $r_9 = 0$, and
- $w_{10} = 1$, $r_{10} = 0$.

An optimal schedule π^{OPT} consists in processing jobs 9 and 10 before job 8, which gives an objective of $\max w_j F_j^{\mathrm{OPT}} = 9$. If an online algorithm $\mathcal{A}$ processes job 8 before jobs 9 and 10, then the adversary stops there, and $\mathcal{A}$ is clearly not optimal. Otherwise, the adversary releases an 11th job at time $r_{11} = 9$ with weight $w_{11} = 11$. The optimal schedule now consists in processing job 8 before job 11, and job 9

Problem	Algorithm	Competitive Ratio	Ref.
$1 \mid \textit{online-}r_j, p_j = 1 \mid \max w_j F_j$	*Any online*	$\geq 1 + \varepsilon$	Theorem 3.1
$1 \mid \textit{online-}r_j, p_j = 1 \mid \max w_j F_j$	MaxFlow	$\geq \infty$	Theorem 3.2
$1 \mid \textit{online-}r_j \mid \max w_j F_j$	*Any online*	$\geq \Delta + 1$	Theorem 3.3
$P \mid \mathcal{M}_j, \textit{online-}r_j, p_j = 1 \mid F_{\max}$	*Any online*	$\geq \Omega(m)$	Anand et al. [1]
$P \mid \mathcal{M}_j(\textit{nested}), \textit{online-}r_j, p_j = 1 \mid F_{\max}$	*Any online*	$\geq \frac{1}{3}\lfloor \log_2(m) + 2 \rfloor$	Theorem 3.4
$P \mid \mathcal{M}_j(\textit{k-interval}), \textit{online-}r_j, p_j = p \mid F_{\max}$	*Any online*	≥ 2	Theorem 3.5
$P \mid \mathcal{M}_j(\textit{inclusive}), \textit{online-}r_j, p_j = p \mid F_{\max}$	*Any i.d.*	$\geq \lfloor \log_2(m) + 1 \rfloor$	Theorem 3.6
$P \mid \mathcal{M}_j(\textit{k-size}), \textit{online-}r_j, p_j = p \mid F_{\max}$	*Any i.d.*	$\geq \lfloor \log_k(m) \rfloor$	Theorem 3.7
$P \mid \textit{online-}r_j \mid F_{\max}$	EFT	$\leq 3 - 2/m$	Theorem 3.9
$P \mid \mathcal{M}_j(\textit{disjoint}), \textit{online-}r_j \mid F_{\max}$	EFT	$\leq 3 - 2/\max_j \{\lvert\mathcal{M}_j\rvert\}$	Theorem 3.12
$P \mid \mathcal{M}_j(\textit{k-disjoint}), \textit{online-}r_j \mid F_{\max}$	EFT	$\leq 3 - 2/k$	Theorem 3.12
$P \mid \mathcal{M}_j(\textit{k-interval}), \textit{online-}r_j, p_j = 1 \mid F_{\max}$	EFT-Min	$\geq m - k + 1$	Theorem 3.13
$P \mid \mathcal{M}_j(\textit{k-interval}), \textit{online-}r_j, p_j = 1 \mid F_{\max}$	EFT-Rand	$\geq m - k + 1$	Theorem 3.17
$P \mid \mathcal{M}_j(\textit{k-interval}), \textit{online-}r_j \mid F_{\max}$	EFT	$\geq m - k + 1$	Theorem 3.20

Table 3.1 – Competitive ratio guarantees of different classes of scheduling algorithms, with various processing set restrictions. *Any online* means that the result applies to any online algorithm. *Any i.d.* means that the result applies to any immediate dispatch algorithm. EFT means that the result applies to any EFT-like algorithm (i.e., any tie-break function). A sign $\geq$ denotes a lower bound, whereas $\leq$ denotes an upper bound.

(or job 10) should start last, to obtain an objective of $\max w_j F_j^{\mathrm{OPT}} = 11$. In this case, $\mathcal{A}$ has already processed jobs 9 and 10 when job 11 is released, thus it cannot be optimal. Hence, the competitive ratio of $\mathcal{A}$ is always strictly greater than 1. ■

Algorithm 4 MaxFlow

1: **when** the single machine is idle at time t **do**
2: Execute the job j whose current weighted flow $w_j(t + 1 - r_j)$ is the highest

Theorem 3.2. *The competitive ratio of* MaxFlow *(Algorithm 4) is arbitrarily large for* $1 \mid \textit{online-}r_j, p_j = 1 \mid \max w_j F_j$.

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

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

Lower bound. At time $t = 0$, MaxFlow starts one of the first ρ jobs because they are the only ones that are ready. We now prove that at any time t such that $1 \leq t < \rho$, MaxFlow starts one of the remaining first ρ jobs, which delays all arriving jobs (any job j such that $\rho < j < 2\rho$).

On the one hand, $w_j(t + 1 - r_j) = \rho(t + 1)$ for any of the first ρ jobs ($1 \leq j \leq \rho$). On the other hand, for $\rho < j \leq n$, $w_j \leq (1 + 1/\rho) w_{j-1}$ (by definition), and thus, $w_j \leq (1 + 1/\rho)^{j-\rho} \rho$. Therefore, $w_j(t+1-r_j) \leq (1+1/\rho)^{j-\rho} \rho(t+1-j+\rho)$. Let us show that at any time t such that $1 \leq t < \rho$, any of the first ρ jobs has the highest value, that is $\rho(t+1) \geq (1+1/\rho)^{j-\rho} \rho(t+1-j+\rho)$ for all $\rho < j \leq t+\rho$.

By changing variables ($h = j - \rho$ and $\tau = t + 1$), this corresponds to proving $(1 + 1/\rho)^h(\tau - h) \leq \tau$ for all $1 \leq h < \tau \leq \rho$.

We show by induction that $(1 + 1/\rho)^h(\tau - h) \leq \tau$ for all $0 \leq h$ and for a given τ ($2 \leq \tau \leq \rho$). The induction basis with $h = 0$ is direct. The induction step assumes $(1 + 1/\rho)^h(\tau - h) \leq \tau$ to be true for a given $h \geq 0$. We have

$$\begin{aligned}(1 + 1/\rho)\frac{\tau - h - 1}{\tau - h} &= (1 + 1/\rho)(1 - \frac{1}{\tau - h}) \\ &= 1 + 1/\rho - \frac{1}{\tau - h} - \frac{1}{\rho(\tau - h)} \leq 1.\end{aligned}$$

The last line is obtained by remarking that $\tau \leq \rho$ and $h \geq 0$ (thus, $1/\rho \leq \frac{1}{\tau - h}$). Therefore,

$$\begin{aligned}(1 + 1/\rho)^{h+1}(\tau - (h + 1)) &= (1 + 1/\rho)^h(1 + 1/\rho)(\tau - h)\frac{\tau - h - 1}{\tau - h} \\ &\leq (1 + 1/\rho)^h(\tau - h) \\ &\leq \tau,\end{aligned}$$

which concludes the induction proof.

At time $t = \rho$, all of the first ρ jobs have been completed. We now prove that at any time t such that $\rho \leq t < n$, MaxFlow starts job $t+1$. This would mean that at time t, only jobs j such that $t < j \leq t+\rho$ are ready and not completed. We prove by induction that at time $\rho \leq t < n$, all jobs j with $j \leq t$ are completed. The induction basis with $t = \rho$ is already proven above. Assume the hypothesis is true for a given $\rho \leq t < n$. It remains to prove that at time $\tau = t + 1$, job $t + 2$ is started among jobs j such that $\tau < j \leq \tau + \rho$.

On the one hand, $w_j(\tau + 1 - r_j) = w_{t+2}\rho$ for job $t + 2$. On the other hand, for $\tau + 1 < j \leq \tau + \rho$, $w_j(\tau+1-r_j) \leq (1+1/\rho)^{j-\tau-1}w_{t+2}(\tau+1-j+\rho)$. Let us show that $(1+1/\rho)^{j-\tau-1}w_{t+2}(\tau+1-j+\rho) < w_{t+2}\rho$ for $\tau + 1 < j \leq \tau + \rho$ and for a given $\rho \leq t < n$. By changing variables ($h = j - \tau - 1$), this corresponds to proving that $(1 + 1/\rho)^h(\rho - h) < \rho$ for all $0 < h < \rho$. We show this again by induction on h for a given $\rho \geq 1$. For the induction basis, $(1 + 1/\rho)(\rho - 1) = \rho + 1 - 1 - 1/\rho < \rho$. For the induction step, we can show that $(1 + 1/\rho)\frac{\rho-h+1}{\rho-h} \leq 1$ by remarking that $\rho > \rho - h$, which concludes the induction proof.

To conclude on the performance of MaxFlow, job j is started at time $j-1$ and therefore, the objective value is at least $w_nF_n = w_n(n - (n - \rho)) = \rho w_n$.

Upper bound. A better objective value can be obtained by starting all jobs as soon as they arrive except for the first ρ ones. Job 1 is started at time $t = 0$. Then, job j is started at time $t = j - \rho$ for $\rho < j \leq n$. Finally, the remaining jobs among the first ρ ones are started (j is started at time $t = n - \rho + j - 1$ for $1 < j \leq \rho$). We analyse the objective values for jobs ρ (because it is the last one to be executed among the first ρ jobs) and n (because it is the one with the highest weight among the last $n - \rho$ jobs). For job ρ, $w_\rho F_\rho = \rho(C_\rho - r_\rho) = \rho n$. For job n, $w_nF_n = w_n$.

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

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

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

Thus, $w_n \geq \rho(\rho+1) + \rho n_2 + n_2^2/2$, and our hypothesis $w_n \geq \rho n$ would be verified if

$$\rho(\rho+1) + \rho n_2 + n_2^2/2 \geq \rho n.$$

By replacing n_2 and simplifying, the previous condition becomes

$$n \geq \rho + n_1 + \sqrt{2\rho(n_1 - 1)}. \tag{3.1}$$

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

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

where $\gamma \approx 0.577$ is the Euler-Mascheroni constant. Let N denote this last bound, i.e., $N = \rho(\log(\rho) + \gamma + \frac{1}{2\rho} + 1)$. Overall, this means that if

$$n \geq \rho + N + \sqrt{2\rho(N-1)}, \tag{3.2}$$

then Condition 3.1 is verified in any case, as $n_1 < N$. Numerical analysis shows that Condition 3.2 (and thus Condition 3.1) holds when $n = \rho^2 + 11$, which proves that $w_n \geq \rho n$. Hence, the optimal objective is at most w_n and the one achieved with MaxFlow is at least ρw_n. The conclusion follows. ■

Theorem 3.3. *The competitive ratio of any online algorithm is at least* $\Delta+1$*, where* $\Delta = \max p_j / \min p_j$*, for* $1 \mid \textit{online-}r_j \mid \max w_j F_j$.

Proof: Let a, b be arbitrary values such that $a \geq b > 0$. By contradiction, suppose there exists a ρ-competitive online algorithm $\mathcal{A}$ for the problem $1 \mid \textit{online-}r_j \mid \max w_j F_j$ such that $\rho < a/b + 1$. We now build an adversary job submission strategy that will lead to exceeding this ratio when $\Delta = a/b$. The adversary sends two jobs with the following characteristics:

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

When scheduled by $\mathcal{A}$, job 1 completes at time $\sigma_1 + a$ and job 2 completes at time $\sigma_1 + a + b$ in the best case: as the adversary sends job 2 at time $\sigma_1 + \varepsilon$, job 1 has already started and we must wait for its completion. Thus, in this schedule, $w_1 F_1 = \sigma_1 + a$ and $w_2 F_2 \geq W(\sigma_1 + a + b - (\sigma_1 + \varepsilon)) = W(a + b - \varepsilon)$. Therefore,

$$\begin{aligned} \max w_j F_j &\geq \max(\sigma_1 + a, W(a + b - \varepsilon)) \\ &\geq W(a + b - \varepsilon). \end{aligned}$$

We now study the performance of an offline schedule π^{OFF} on this instance, which executes job 2 first if and only if $\sigma_1 < a-\varepsilon$. We will see that π^{OFF} is indeed optimal, as it always reaches an objective of Wb, which is a lower bound on the weighted flow for job 2. We consider two cases in the analysis, depending on whether job 2 is scheduled first or not.

Case 1. The algorithm $\mathcal{A}$ decides to execute job 1 before time $a-\varepsilon$, i.e., $\sigma_1 < a-\varepsilon$. In the offline schedule, job 2 is executed first at time $r_2 = \sigma_1 + \varepsilon$, which gives $w_2 F_2^{\mathrm{OFF}} = Wb$, and then job 1 at time $\sigma_1 + \varepsilon + b$, which gives

$$\begin{aligned} w_1 F_1^{\mathrm{OFF}} &= \sigma_1 + \varepsilon + b + a \\ &< a - \varepsilon + \varepsilon + b + a = 2a + b. \end{aligned}$$

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

Case 2. The algorithm $\mathcal{A}$ decides to execute job 1 after time $a-\varepsilon$, i.e., $\sigma_1 \geq a-\varepsilon$. In the offline schedule, job 1 is executed first at time $r_1 = 0$, which gives $w_1 F_1^{\mathrm{OFF}} = a$, and then job 2 at time $r_2 = \sigma_1 + \varepsilon \geq a$, which gives $w_2 F_2^{\mathrm{OFF}} = Wb$. We have $a < Wb$, hence, $\max w_j F_j^{\mathrm{OFF}} = w_2 F_2^{\mathrm{OFF}} = Wb$.

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

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

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

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

This contradicts the ρ-competitiveness of $\mathcal{A}$, thus the competitive ratio is at least $a/b + 1$. Note that in this instance, $\max p_j = p_1 = a$ and $\min p_j = p_2 = b$, i.e., $a/b = \max p_j / \min p_j = \Delta$. The conclusion follows. ■

Theorem 3.4. *The competitive ratio of any online algorithm is at least* $\frac{1}{3}\lfloor \log_2(m) + 2 \rfloor$*, where* m *is the number of machines, for* $P \,|\, \mathcal{M}_j(nested), online\text{-}r_j, p_j = 1 \,|\, F_{\max}$.

Proof: Let us assume that we work on a number of machines m that is a power of 2, i.e., $m = 2^{\lfloor \log_2(m') \rfloor}$, where m' is the actual number of machines. Let $\mathcal{A}$ be an arbitrary online scheduling algorithm. Let F be an arbitrary number such that $F \geq \log_2(m) + 2$. We construct the following instance. At time $t_0 = 0$, we consider the interval of machines of size s_0 and starting from machine u_0 (that is, the set of machines $\{u_0, u_0+1, \ldots, u_0+s_0-1\}$), which we denote by $I(u_0, s_0)$, where $u_0 = 1$ and $s_0 = m$. We submit s_0 unitary jobs at time t_0, with the processing set $\mathcal{M}_j = I(u_0, s_0)$. Let $\mathcal{G}_{1,0}$ denote this set of jobs. For each machine $i \in I(u_0, s_0)$, we release one unitary job at each time $t_0, t_0+1, \ldots, t_0+F-1$ and feasible only on the machine i. Let $\mathcal{G}_{2,0}$ denote this set. Note that at time t_0+F-1, algorithm $\mathcal{A}$ should have completed the jobs of $\mathcal{G}_{1,0}$, otherwise the maximum flow time would be greater than $\log_2(m) + 2$.

Now, for all $k > 0$, we set $t_k = t_{k-1} + F$ and $s_k = \frac{1}{2} s_{k-1}$. We choose u_k such that $u_{k-1} \leq u_k \leq u_{k-1} + s_{k-1} - s_k = u_{k-1} + s_k$ (in other words, $I(u_k, s_k)$ is a subinterval of $I(u_{k-1}, s_{k-1})$), and such that $|\mathcal{G}_{0,k}|$ is maximized, where $\mathcal{G}_{0,k} \subset \mathcal{G}_{2,k-1}$ is the set of jobs that are submitted before t_k but not completed at this time, and that can be executed on one machine only in the interval $I(u_k, s_k)$. Then we submit job sets $\mathcal{G}_{1,k}$ and $\mathcal{G}_{2,k}$ as previously: $\mathcal{G}_{1,k}$ is made of s_k jobs with processing set $I(u_k, s_k)$ released at time t_k, and $\mathcal{G}_{2,k}$ contains F jobs for each machine $i \in I(u_k, s_k)$ submitted at times $t_k, t_k+1, \ldots, t_k+F-1$ and that must be processed on i.

We prove the following statements by induction. For all $k \geq 0$,

(i) $s_k = m/2^k$, and

(ii) there are at least ks_k uncompleted jobs on $I(u_k, s_k)$ at time t_k before sending $\mathcal{G}_{1,k}$ and $\mathcal{G}_{2,k}$, i.e., $|\mathcal{G}_{0,k}| \geq ks_k$.

For the base case ($k = 0$), we have $s_0 = m/2^0 = m$, and $\mathcal{G}_{0,k} = \emptyset$, so there is no completed job on $I(1, m)$ at time 0 before sending $\mathcal{G}_{1,0}$ and $\mathcal{G}_{2,0}$.

Now assume that $s_k = m/2^k$ is true at a certain step k. At step $k + 1$, we have $s_{k+1} = \frac{1}{2}s_k$ by definition, so $s_{k+1} = \frac{1}{2}(m/2^k) = m/2^{k+1}$, which proves Item (i).

Suppose that there are at least ks_k uncompleted jobs on $I(u_k, s_k)$ at time t_k, i.e., $|\mathcal{G}_{0,k}| \geq ks_k$. Then we send $\mathcal{G}_{1,k}$ and $\mathcal{G}_{2,k}$, which means that there are at least

$$ks_k + s_k + Fs_k - Fs_k = (k + 1)s_k$$

uncompleted jobs on $I(u_k, s_k)$ at time $t_{k+1} = t_k + F$.

Now we choose the subinterval $I(u_{k+1}, s_{k+1}) \subset I(u_k, s_k)$ maximizing $|\mathcal{G}_{0,k+1}|$ at time t_{k+1}. Let us divide $I(u_k, s_k)$ into 2 disjoint subintervals of size $\frac{1}{2}s_k$ and by contradiction, assume that no such subinterval contains $(k + 1)\frac{1}{2}s_k$ uncompleted jobs, i.e., there are at most $(k + 1)\frac{1}{2}s_k - 1$ uncompleted jobs on each of these subintervals. Thus, there are at most $2((k+1)\frac{1}{2}s_k - 1) = (k+1)s_k - 2$ uncompleted jobs on $I(u_k, s_k)$, which contradicts the fact that $I(u_k, s_k)$ holds at least $(k + 1)s_k$ uncompleted jobs. Then, the chosen subinterval $I(u_{k+1}, s_{k+1})$ contains at least $(k+1)\frac{1}{2}s_k = (k+1)s_{k+1}$ uncompleted jobs at time t_{k+1} before sending $\mathcal{G}_{1,k+1}$ and $\mathcal{G}_{2,k+1}$ (that is, $|\mathcal{G}_{0,k+1}| \geq (k + 1)s_{k+1}$), which proves Item (ii).

We stop when we reach the step k such that $s_k = 1$. This means that $m/2^k = 1$, i.e., $k = \log_2(m)$. Therefore, there remains at least $ks_k = \log_2(m)$ uncompleted jobs on an interval of size 1 at time t_k, plus 1 job of $\mathcal{G}_{1,k}$ and 1 job of $\mathcal{G}_{2,k}$, which gives a maximum flow time of at least $\log_2(m) + 2$. Thus, on all m' machines, we have a maximum flow of

$$\begin{aligned}\log_2(m) + 2 &= \log_2(2^{\lfloor \log_2(m') \rfloor}) + 2 \\ &= \lfloor \log_2(m') \rfloor + 2 = \lfloor \log_2(m') + 2 \rfloor .\end{aligned}$$

The optimal strategy consists, at each step $0 \leq k < \log_2(m)$, in executing all jobs of $\mathcal{G}_{1,k}$ on the subinterval $I(u_k, s_k) \setminus I(u_{k+1}, s_{k+1})$, for a max-flow of 3: jobs of $\mathcal{G}_{1,k}$ are scheduled first (with flow 2), followed by jobs of $\mathcal{G}_{2,k}$, which have a flow at most 3. The conclusion follows. ■

Theorem 3.5. *The competitive ratio of any online algorithm is at least 2 for* $P \,|\, \mathcal{M}_j(k\text{-}interval), online\text{-}r_j, p_j = p \,|\, F_{\max}$.

Proof: Let $\mathcal{A}$ be an arbitrary online algorithm. At time 0, the adversary sends one job with processing time p and with $\mathcal{M}_1 = \{2, 3\}$. Now there are two cases: $\mathcal{A}$ executes this job on machine 2 or on machine 3, and we denote its starting time by σ_1. Note that if $\sigma_1 \geq p$, the flow time for this job is at least $2p$, while an optimal algorithm could schedule this job at time 0 with a flow time of 1, leading to a ratio larger than, or equal to 2. We thus assume that $\sigma_1 < p$.

Let us assume $\mathcal{A}$ executes job 1 on machine 2. Then the adversary sends two jobs at time $\sigma_1 + 1$ with processing time p and with $\mathcal{M}_2 = \mathcal{M}_3 = \{1, 2\}$. $\mathcal{A}$ will schedule at least one job at time $\sigma_1 + p$ at the earliest, and this job will complete at time $\sigma_1 + 2p$ at the earliest, for a max-flow of at least $2p - 1$. The optimal schedule consists in executing job 1 on machine 3 at time 0, to let the next two jobs execute on machines 1 and 2 at time 1, for a max-flow of p. As $p \to \infty$, the competitive ratio is 2. The other case is proved analogously by sending two jobs on machines $\{3, 4\}$. ■

3.3.2 Lower Bounds for Immediate Dispatch Algorithms

We first study the *inclusive* structure of processing sets. We show in Theorem 3.6 that restricting to this structure reduces the lower bound on the competitive ratios to $\lfloor \log_2(m) + 1 \rfloor$ for immediate dispatch algorithms. This is also true for the *nested* and *interval* structures, as they generalize the *inclusive* structure.

Theorem 3.6. *The competitive ratio of any immediate dispatch algorithm is at least* $\lfloor \log_2(m) + 1 \rfloor$, *where* m *is the number of machines, for* $P \,|\, \mathcal{M}_j(\mathit{inclusive}), \mathit{online}\text{-}r_j, p_j = p \,|\, F_{\max}$.

Proof: Let us assume that we work on a number of machines m that is a power of 2, i.e., $m = 2^{\lfloor \log_2(m') \rfloor}$, where m' is the actual number of machines. Let $\mathcal{A}$ be an arbitrary online immediate dispatch algorithm. We build the following adversary. For each ℓ such that $1 \leq \ell \leq \log_2(m)$, let $\mathcal{J}_\ell$ denote the set of $\frac{m}{2^\ell}$ jobs with $p_j = p > \log_2(m)$ and $r_j = \ell - 1$ for all $j \in \mathcal{J}_\ell$. A final job is released at time $r_j = \log_2(m)$.

Then we define $\mathcal{M}^{(1)} = \{1, \cdots, m\}$ and for all $\ell > 1$, $\mathcal{M}^{(\ell)}$ denotes the subset of machines of $\mathcal{M}^{(\ell-1)}$ of size $\frac{m}{2^{\ell-1}}$ with at least $(\ell-1)\frac{m}{2^{\ell-1}}$ allocated jobs in total after step $\ell - 1$ (we prove below that such a set exists). Finally, for each ℓ and for all $j \in \mathcal{J}_\ell$, we set $\mathcal{M}_j = \mathcal{M}^{(\ell)}$.

Let us prove by induction that the construction of $\mathcal{M}^{(\ell)}$ is valid, i.e., that such a subset exists for all $\ell > 0$. Note that as $\mathcal{A}$ is an immediate dispatch algorithm, all jobs of $\mathcal{J}_\ell$ are irremediably scheduled at time $\ell - 1$ on some machines of $\mathcal{M}^{(\ell)}$. For the construction of $\mathcal{M}^{(2)}$, we start from $\mathcal{M}^{(1)} = \{1, \cdots, m\}$ where $\frac{m}{2}$ jobs have been allocated on the first step. We select for $\mathcal{M}^{(2)}$ the subset of machines where these jobs have been allocated, possibly with additional machines to reach the proper size $\frac{m}{2}$.

We now assume that $\mathcal{M}^{(\ell)}$ has been constructed and prove that we can build $\mathcal{M}^{(\ell+1)}$. By induction, $\mathcal{M}^{(\ell)}$ has been allocated $(\ell-1)\frac{m}{2^{\ell-1}}$ jobs up to step $\ell - 1$, and $\frac{m}{2^\ell}$ new jobs on step ℓ. This makes a total of $(2\ell-1)\frac{m}{2^\ell}$ jobs. We select for $\mathcal{M}^{(\ell+1)}$ the $\frac{m}{2^\ell}$ machines that are the most loaded in $\mathcal{M}^{(\ell)}$. We consider two cases:

(i) Each of the selected machines has at least ℓ jobs. Then in total, we have at least $\ell\frac{m}{2^\ell}$ jobs, as requested.

(ii) There exists a selected machine with at most $\ell - 1$ jobs. This means that all non-selected machines have at most $\ell - 1$ jobs (otherwise, we would have selected one of them instead), for a total work (on the $\frac{m}{2^\ell}$ non-selected machines) of at most $(\ell-1)\frac{m}{2^\ell}$ jobs. Thus, on selected machines, the number of jobs is at least

$$(2\ell-1)\frac{m}{2^\ell} - (\ell-1)\frac{m}{2^\ell} = \ell\frac{m}{2^\ell}.$$

At step $\log_2(m)$, $\mathcal{M}^{(\log_2(m))}$ is reduced to two machines, with at least $2(\log_2(m)-1)$ allocated jobs, where a single job is scheduled at time $\log_2(m)-1$. This leaves one machine with at least $\log_2(m)$ jobs, where we finally allocate the last job at time $\log_2(m)$, leading to a maximum flow of $(\log_2(m)+1)p - \log_2(m)$. Note that

$$\begin{aligned}\log_2(m) + 1 &= \log_2(2^{\lfloor \log_2(m') \rfloor}) + 1 \\ &= \left\lfloor \log_2(m') \right\rfloor + 1 = \left\lfloor \log_2(m') + 1 \right\rfloor.\end{aligned}$$

The optimal strategy consists in scheduling each set $\mathcal{J}_\ell$ on machines $\mathcal{M}^{(\ell)} \setminus \mathcal{M}^{(\ell+1)}$, for a max-flow of p. Thus, as $p \to \infty$, we have a competitive ratio of $\lfloor \log_2(m') + 1 \rfloor$. ■

The previous result may be adapted for processing sets that do not present any particular structure, but have all the same size k.

Theorem 3.7. *The competitive ratio of any immediate dispatch algorithm is at least* $\lfloor \log_k(m) \rfloor$*, where* m *is the number of machines, for* $P \,|\, \mathcal{M}_j(k\text{-}size), online\text{-}r_j, p_j = p \,|\, F_{\max}$.

Proof: Let us assume that we work on a number of machines m that is a power of k, i.e., $m = k^{\lfloor \log_k(m') \rfloor}$, where m' is the actual number of machines. Let $\mathcal{A}$ be an arbitrary immediate dispatch algorithm. We proceed by building the following adversary. For each ℓ such that $1 \leq \ell \leq \log_k(m)$, let $\mathcal{J}_\ell$ denote the set of $\frac{m}{k^\ell}$ jobs with $p_j = p > \log_k(m)$ and $r_j = \ell - 1$ for all $j \in \mathcal{J}_\ell$.

Note that as $\mathcal{A}$ is an immediate dispatch algorithm, all jobs of $\mathcal{J}_\ell$ are irremediably scheduled at time $\ell - 1$. Then we define $\mathcal{M}^{(\ell)}$ as the set of machines on which the jobs of $\mathcal{J}_\ell$ are scheduled at this specific time, with the particular case $\mathcal{M}^{(0)} = M$. Finally, for each ℓ and for all $j \in \mathcal{J}_\ell$, we set $\mathcal{M}_j \subseteq \mathcal{M}^{(\ell-1)}$, with $|\mathcal{M}_j| = k$. Moreover, all processing sets of jobs that belong to the same set $\mathcal{J}_\ell$ are mutually disjoint, i.e., $\mathcal{M}_j \cap \mathcal{M}_{j'} = \emptyset$ for all $j, j' \in \mathcal{J}_\ell$ such that $j \neq j'$.

$\mathcal{A}$ will be forced to schedule each set $\mathcal{J}_\ell$ on the exact same machines that are already busy with the jobs of the previous set $\mathcal{J}_{\ell-1}$. As all processing sets are mutually disjoint, we know that the jobs $\mathcal{J}_\ell$ are scheduled on $|\mathcal{J}_\ell| = \frac{m}{k^\ell}$ machines exactly. Moreover, there are exactly $\ell\frac{m}{k^\ell}$ waiting jobs on these machines at step ℓ. Thus, at the last step $\ell = \log_k(m)$, the completion time is $\log_k(m)p$. Therefore, the maximum flow time is $\log_k(m)p - (\log_k(m) - 1)$. Note that

$$\begin{aligned} \log_k(m) &= \log_k(k^{\lfloor \log_k(m') \rfloor}) \\ &= \left\lfloor \log_k(m') \right\rfloor . \end{aligned}$$

The optimal strategy consists in scheduling each set $\mathcal{J}_\ell$ on machines $\mathcal{M}^{(\ell-1)} \setminus \mathcal{M}^{(\ell)}$, for a max-flow of p. Thus, as $p \to \infty$, we have a competitive ratio of $\lfloor \log_k(m') \rfloor$. ■

3.3.3 Upper Bounds for Earliest Finish Time

FIFO scheduling has been extensively studied in previous work. It consists of a single queue of jobs that is pulled whenever some machine is available (see Algorithm 5). It is known to be $(3 - 2/m)$-competitive when minimizing maximum flow time on parallel machines [3, 4, 32], which makes it optimal on a single machine. In the present paper, we move our focus to the EFT scheduler (see Algorithm 6), which pushes each released job on the machine that finishes the earliest. We show here that both schedulers are equivalent on any instance of the scheduling problem $P \,|\, online\text{-}r_j \,|\, F_{\max}$. However, EFT has two main advantages over FIFO, which motivates our choice:

1. FIFO needs a centralized queue, whereas EFT allocates jobs to machines as soon as they arrive (it is an immediate dispatch algorithm). Hence, it does not require a centralized scheduler with a potentially large queue of jobs, which is impractical, for instance, in most existing online systems with critical scalability needs. This makes EFT more adapted to distributed systems, whereas FIFO is more adapted to shared-memory parallelism.

2. EFT can easily be extended to scenarios with processing set restrictions, whereas transforming FIFO to allow such constraints is cumbersome.

For each machine $i \in M$ and for any job $j \in J$, let $H_{i,j}$ denote jobs $\{1, \cdots, j\}$ being assigned to i in a schedule π:

$$H_{i,j} = \left\{ j' \in J \text{ s.t. } 1 \leq j' \leq j \text{ and } \mu_{j'} = i \right\} .$$

Then we define $\mathcal{C}_{i,j}$ as the time at which machine i completes its assigned jobs among the first j jobs in π, i.e.,

$$\mathcal{C}_{i,j} = \max_{j' \in H_{i,j}} \left\{ C_{j'} \right\} ,$$

where $C_{j'} = \sigma_{j'} + p_{j'}$ is the completion time of j' in π, with the convention $\mathcal{C}_{i,0} = 0$. Finally, we define U_j as the set of machines that may start the job j at the earliest possible time $t_{\min,j} = \max\left(r_j, \min_{i \in M}\left\{\mathcal{C}_{i,j-1}\right\}\right)$, i.e., U_j is the set of machines that are in a tie for j:

$$U_j = \left\{i \in M \text{ s.t. } \mathcal{C}_{i,j-1} \leq t_{\min,j}\right\}. \tag{3.3}$$

Note that EFT needs to know the set U_j for each released job, which implies that one must know the processing time of arriving jobs with precision, in order to compute the completion times of machines at each step (we are in a clairvoyant setting). In this way, EFT can be readily modified to account for processing set restrictions by changing Equation (3.3) to

$$U'_j = \left\{i \in \mathcal{M}_j \text{ s.t. } \mathcal{C}_{i,j-1} \leq t'_{\min,j}\right\}, \tag{3.4}$$

where $t'_{\min,j} = \max(r_j, \min_{i \in \mathcal{M}_j}\left\{\mathcal{C}_{i,j-1}\right\})$.

For both EFT and FIFO strategies, a tie-break policy decides which machine will process each job. We consider that ties are broken according to the same policy BreakTie in FIFO and EFT (in FIFO, ties are broken when at least 2 machines are idle at the same time and we assume that the selected machine runs first).

Algorithm 5 FIFO

Require: Global first-in first-out queue Q
Input: Incoming jobs j
Output: Allocated machines μ_j and starting times σ_j

when a new job j is released **do**
    $enqueue(j, Q)$

In parallel, do:

when a set of machines U become idle at time t **do**
    $j \leftarrow dequeue(Q)$
    if $j \neq nil$ **then**
        $u \leftarrow \textsc{BreakTie}(U)$
        $\mu_j \leftarrow u$
        $\sigma_j \leftarrow t$

Algorithm 6 EFT

Input: Incoming jobs j
Output: Allocated machines μ_j and starting times σ_j

when a new job j is released **do**
    Get U_j according to completion times of machines M (Equation (3.3))
    $u \leftarrow \textsc{BreakTie}(U_j)$
    $\mu_j \leftarrow u$
    $\sigma_j \leftarrow \max(r_j, \mathcal{C}_{u,j-1})$
    Update the completion time of machine u

AD *fig fifo/eft?*

Now we show that EFT is equivalent to FIFO for the problem $P \mid online\text{-}r_j \mid F_{\max}$. Let π^{FIFO} (resp. π^{EFT}) denote the schedule obtained when applying FIFO (resp. EFT) on a given instance.

Proposition 3.8. *For any instance of $P \mid online\text{-}r_j \mid F_{\max}$, we have $\pi^{\text{FIFO}} = \pi^{\text{EFT}}$, i.e., $\mu_j^{\text{FIFO}} = \mu_j^{\text{EFT}}$ and $\sigma_j^{\text{FIFO}} = \sigma_j^{\text{EFT}}$ for all jobs j.*

Proof: We prove the following statement by induction: for any h such that $1 \leq h \leq n$, $\mu_j^{\text{FIFO}} = \mu_j^{\text{EFT}}$ and $\sigma_j^{\text{FIFO}} = \sigma_j^{\text{EFT}}$ for all jobs j such that $1 \leq j \leq h$.

Base case ($h = 1$). All machines are idle (thus all machines are in a tie, i.e., $U_1^{\text{FIFO}} = U_1^{\text{EFT}} = M$). As FIFO and EFT have the same tie-break policy and it is called on the same machine subset, they will choose the same machine and execute the first job as soon as it is released.

Induction step. Suppose that for a given $h < n$, $\mu_j^{\text{FIFO}} = \mu_j^{\text{EFT}}$ and $\sigma_j^{\text{FIFO}} = \sigma_j^{\text{EFT}}$ for all $1 \leq j \leq h$. We show that $\mu_{h+1}^{\text{FIFO}} = \mu_{h+1}^{\text{EFT}}$ and $\sigma_{h+1}^{\text{FIFO}} = \sigma_{h+1}^{\text{EFT}}$.

On the one hand, at time r_{h+1}, EFT will schedule the job $h+1$ on one machine u in the subset U_{h+1}^{EFT} according to the tie-break policy. Thus, we have $\mu_{h+1}^{\text{EFT}} = u$ and $\sigma_{h+1}^{\text{EFT}} = \max(r_{h+1}, \mathcal{C}_{u,h}^{\text{EFT}})$.

On the other hand, at time $\max(r_{h+1}, \min_i \left\{\mathcal{C}_{i,h}^{\text{FIFO}}\right\})$, one of the machine in the subset U_{h+1}^{FIFO} will wake up first according to the tie-break policy. Let u' denote this machine. The machine u' will pull the next job to process from the shared queue Q, which is necessarily the job $h+1$. Therefore, $\mu_{h+1}^{\text{FIFO}} = u'$ and $\sigma_{h+1}^{\text{FIFO}} = \max(r_{h+1}, \mathcal{C}_{u',h}^{\text{FIFO}})$.

As $\mu_j^{\text{FIFO}} = \mu_j^{\text{EFT}}$ and $\sigma_j^{\text{FIFO}} = \sigma_j^{\text{EFT}}$ for all jobs j such that $1 \leq j \leq h$, we deduce that all machines complete at the same time in π^{FIFO} and π^{EFT} when the first h tasks are considered, i.e., for all machines i, $\mathcal{C}_{i,h}^{\text{FIFO}} = \mathcal{C}_{i,h}^{\text{EFT}}$. This implies that $U_{h+1}^{\text{FIFO}} = U_{h+1}^{\text{EFT}}$. As FIFO and EFT break ties the same way, we have $u = u'$, and then $\mathcal{C}_{u,h}^{\text{FIFO}} = \mathcal{C}_{u',h}^{\text{EFT}}$. Therefore, $\mu_{h+1}^{\text{FIFO}} = \mu_{h+1}^{\text{EFT}}$ and $\sigma_{h+1}^{\text{FIFO}} = \sigma_{h+1}^{\text{EFT}}$, and the conclusion follows. ■

Theorem 3.9 (Bender et al. [4])**.** FIFO *is $(3 - 2/m)$-competitive for $P \mid online\text{-}r_j \mid F_{\max}$, where m is the number of machines.*

Let $\mathcal{J}_t$ denote the set of jobs released before time t and not yet started in a given schedule, i.e.,

$$\mathcal{J}_t = \{j \in J \text{ s.t. } r_j \leq t \text{ and } \sigma_j > t\},$$

and let $\delta_{t,i}$ be the remaining processing time of the job being executed by machine i at time t, i.e., $\delta_{t,i} = C_j - t$, where $\mu_j = i$ and $\sigma_j \leq t \leq C_j$. Obviously, if no task is being processed on machine i at time t, $\delta_{t,i}$ is set to 0. Then, the total work waiting to be processed at time t is defined as

$$W_t = \sum_{i \in M} \delta_{t,i} + \sum_{j \in \mathcal{J}_t} p_j.$$

We also define the maximum processing time among the first j jobs as $p_{\max,j}$, and the maximum flow time among the first j jobs as $F_{\max,j}$.

Lemma 3.10. *We have $W_{r_j}^{\text{FIFO}} \leq W_{r_j}^{\text{OPT}} + (m-1)p_{\max,j}$ for all jobs j, where* OPT *is an optimal offline strategy.*

Proof: Let us proceed by induction.

Base case ($j = 1$). At time r_1, all machines are idle and we have $W_{r_1}^{\text{FIFO}} = W_{r_1}^{\text{OPT}} = p_1$.

Induction step. Suppose that $W_{r_j}^{\text{FIFO}} \leq W_{r_j}^{\text{OPT}} + (m-1)p_{\max,j}$ for a given job j. We consider two cases:

(i) All machines are busy between r_j and r_{j+1} in π^{FIFO}. We have

$$W^{\text{FIFO}}_{r_{j+1}} = W^{\text{FIFO}}_{r_j} - m(r_{j+1} - r_j) + p_{j+1}.$$

Moreover, $W^{\text{OPT}}_{r_{j+1}} \geq W^{\text{OPT}}_{r_j} - m(r_{j+1} - r_j) + p_{j+1}$ (there may be idle times between r_j and r_{j+1} in π^{OPT}). Then, $W^{\text{OPT}}_{r_{j+1}} - W^{\text{OPT}}_{r_j} \geq W^{\text{FIFO}}_{r_{j+1}} - W^{\text{FIFO}}_{r_j}$. Hence,

$$\begin{aligned} W^{\text{FIFO}}_{r_{j+1}} &\leq W^{\text{OPT}}_{r_{j+1}} + W^{\text{FIFO}}_{r_j} - W^{\text{OPT}}_{r_j} \\ &\leq W^{\text{OPT}}_{r_{j+1}} + (m-1)p_{\max,j} \\ &\leq W^{\text{OPT}}_{r_{j+1}} + (m-1)p_{\max,j+1}. \end{aligned}$$

(ii) There is at least one idle machine between r_j and r_{j+1} in π^{FIFO}. At time r_{j+1}, there is thus no waiting jobs except job $j+1$ (otherwise, it would have already started on an idle machine). In the worst case, $m-1$ machines start to process some jobs just before time r_{j+1} for $p_{\max,j}$ time units. Then we have $W^{\text{FIFO}}_{r_{j+1}} \leq p_{j+1} + (m-1)p_{\max,j}$.

Furthermore, in the best case, the job $j+1$ is the only job in the system at time r_{j+1} in π^{OPT}, thus $W^{\text{OPT}}_{r_{j+1}} \geq p_{j+1}$. Therefore,

$$\begin{aligned} W^{\text{FIFO}}_{r_{j+1}} &\leq p_{j+1} + (m-1)p_{\max,j} \\ &\leq W^{\text{OPT}}_{r_{j+1}} + (m-1)p_{\max,j} \\ &\leq W^{\text{OPT}}_{r_{j+1}} + (m-1)p_{\max,j+1}. \end{aligned}$$

■

Proof of Theorem 3.9: We consider an online schedule π^{FIFO} and an optimal offline schedule π^{OPT}. We start by describing two lower bounds for $F^{\text{OPT}}_{\max,j}$:

$$F^{\text{OPT}}_{\max,j} \geq p_{\max,j}, \tag{3.5}$$

$$F^{\text{OPT}}_{\max,j} \geq W^{\text{OPT}}_{r_j}/m. \tag{3.6}$$

Lower Bound 3.5 is immediate. Lower Bound 3.6 follows from the fact that there is always a non-finished job j' such that $r_{j'} \leq r_j$ and that will necessarily complete after time $r_j + W^{\text{OPT}}_{r_j}/m$.

Now let j be a job in π^{FIFO}. Then—as it is scheduled by FIFO—it is the last job in $\mathcal{J}^{\text{FIFO}}_{r_j}$, and it will not be able to start before time $r_j + (W^{\text{FIFO}}_{r_j} - p_j)/m$ in the worst case. Hence,

$$F^{\text{FIFO}}_j \leq W^{\text{FIFO}}_{r_j}/m + p_j - \frac{p_j}{m} \leq W^{\text{FIFO}}_{r_j}/m + \left(1 - \frac{1}{m}\right) p_{\max,j}$$

is an upper bound for FIFO. By Theorem 3.10, we know that $W^{\text{FIFO}}_{r_j} \leq W^{\text{OPT}}_{r_j} + (m-1)p_{\max,j}$ for each job j. Then,

$$\begin{aligned} F^{\text{FIFO}}_j &\leq W^{\text{FIFO}}_{r_j}/m + \left(1 - \frac{1}{m}\right) p_{\max,j} \\ &\leq W^{\text{OPT}}_{r_j}/m + 2\left(1 - \frac{1}{m}\right) p_{\max,j} \\ &\leq \left(3 - \frac{2}{m}\right) F^{\text{OPT}}_{\max,j} \text{ (by Lower Bounds 3.5 and 3.6).} \end{aligned}$$

■

As a corollary, the problem is polynomial on a single-machine, and FIFO is optimal in this case.

The case of disjoint processing sets is particular. We may apply a competitive algorithm independently on each set, which leads to an algorithm with adapted competitive ratio.

Theorem 3.11. *From any $f(m)$-competitive algorithm for $P \mid online\text{-}r_j \mid F_{\max}$, we can design an algorithm with a competitive ratio of $\max_j \{f(|\mathcal{M}_j|)\}$ for $P \mid \mathcal{M}_j(disjoint), online\text{-}r_j \mid F_{\max}$.*

Proof: Let $\mathcal{I}$ be an arbitrary instance of $P \mid \mathcal{M}_j(disjoint), online\text{-}r_j \mid F_{\max}$, and let $\mathcal{A}$ be an $f(m)$-competitive algorithm for $P \mid online\text{-}r_j \mid F_{\max}$. By definition of the disjoint processing set restriction, we have $\mathcal{M}_j \cap \mathcal{M}_{j'} = \emptyset$ or $\mathcal{M}_j = \mathcal{M}_{j'}$ for all jobs j, j' (with $j \neq j'$) of the instance $\mathcal{I}$. Let $\mathcal{M}$ denote the set of all subsets $\mathcal{M}_j$.

Then, for all $\mathcal{M}_u \in \mathcal{M}$, we construct the set of jobs $\mathcal{J}_u = \{j \in J \text{ s.t. } \mathcal{M}_j = \mathcal{M}_u\}$. As $\mathcal{M}_u \cap \mathcal{M}_v = \emptyset$ for all $\mathcal{M}_u, \mathcal{M}_v \in \mathcal{M}$ such that $u \neq v$, we clearly have $\mathcal{J}_u \cap \mathcal{J}_v = \emptyset$. Moreover,

$$\bigcup_{\mathcal{M}_u \in \mathcal{M}} \mathcal{J}_u = J.$$

Hence, for all $\mathcal{M}_u \in \mathcal{M}$, $\mathcal{J}_u$ and $\mathcal{M}_u$ can clearly constitute an instance $\mathcal{I}_u$ of $P \mid online\text{-}r_j \mid F_{\max}$. We design an online algorithm $\mathcal{A}'$ for the original problem by applying $\mathcal{A}$ in parallel to each instance $\mathcal{I}_u$. By definition of the competitive ratio of $\mathcal{A}$, we have $F^{\mathcal{A}}_{\max}(\mathcal{I}_u) \leq f(|\mathcal{M}_u|) F^{\mathrm{OPT}}_{\max}(\mathcal{I}_u)$, where OPT is an optimal offline strategy. As $\mathcal{I}_u$ is a subinstance of $\mathcal{I}$, we also have

$$F^{\mathrm{OPT}}_{\max}(\mathcal{I}_u) \leq F^{\mathrm{OPT}'}_{\max}(\mathcal{I})$$

for all $\mathcal{I}_u$, where OPT' is an optimal offline strategy built by applying OPT in parallel on each instance $\mathcal{I}_u$. Then, $F^{\mathcal{A}}_{\max}(\mathcal{I}_u) \leq f(|\mathcal{M}_u|) F^{\mathrm{OPT}'}_{\max}(\mathcal{I})$, and

$$\begin{aligned} F^{\mathcal{A}'}_{\max}(\mathcal{I}) &= \max_u \left\{ F^{\mathcal{A}}_{\max}(\mathcal{I}_u) \right\} \\ &\leq \max_u \left\{ f(|\mathcal{M}_u|) \right\} F^{\mathrm{OPT}'}_{\max}(\mathcal{I}). \end{aligned}$$

■

This result has an important corollary for EFT on disjoint processing sets.

Corollary 3.12. EFT *is $(3 - 2/\max_j \{|\mathcal{M}_j|\})$-competitive for $P \mid \mathcal{M}_j(disjoint), online\text{-}r_j \mid F_{\max}$ and $(3 - 2/k)$-competitive for $P \mid \mathcal{M}_j(k\text{-}disjoint), online\text{-}r_j \mid F_{\max}$.*

Proof: By Theorems 3.8 and 3.9, EFT is $(3 - 2/m)$-competitive for $P \mid online\text{-}r_j \mid F_{\max}$. The conclusion follows by applying Theorem 3.11. ■

3.3.4 Lower Bounds for Earliest Finish Time

To exhibit this result, we need to focus on a specific tie-break function. We start by studying the Min tie-break function: in the set U_j of candidate machines that may finish job j at the earliest, we choose the machine with smallest index. The obtained algorithm is called EFT-Min (Algorithm 7) and its competitive ratio is bounded in Theorem 3.13.

Theorem 3.13. *The competitive ratio of* EFT-Min *is at least $m-k+1$ for $P \mid \mathcal{M}_j(k\text{-}interval), online\text{-}r_j, p_j = 1 \mid F_{\max}$, where $1 < k < m$.*

We show that the competitive ratio of EFT-Min is at least $m - k + 1$ for the problem of minimizing max-flow when the processing set is an interval of size k, with $1 < k < m$, even when jobs are unitary. For ease of reading, we say that a given job j is of type λ if its processing interval starts on machine λ, i.e., $\mathcal{M}_j = \{\lambda, \cdots, \lambda + k - 1\}$, and we say that it is of type λ'_{low} (resp. λ'_{up}) if $\lambda \geq \lambda'$ (resp. $\lambda \leq \lambda'$).

Let us build the following adversary (we illustrate an EFT-Min schedule of this adversary in Figure 3.3). At each time t, we send m jobs such that:

Algorithm 7 EFT-MIN

when a new job j is released **do**
  Get U'_j according to completion times of machines $\mathcal{M}_j$ (Equation (3.4))
  $u \leftarrow \textsc{Min}(U'_j)$
  $\mu_j \leftarrow u$
  $\sigma_j \leftarrow \max(r_j, \mathcal{C}_{u,j-1})$
  Update the completion time of machine u

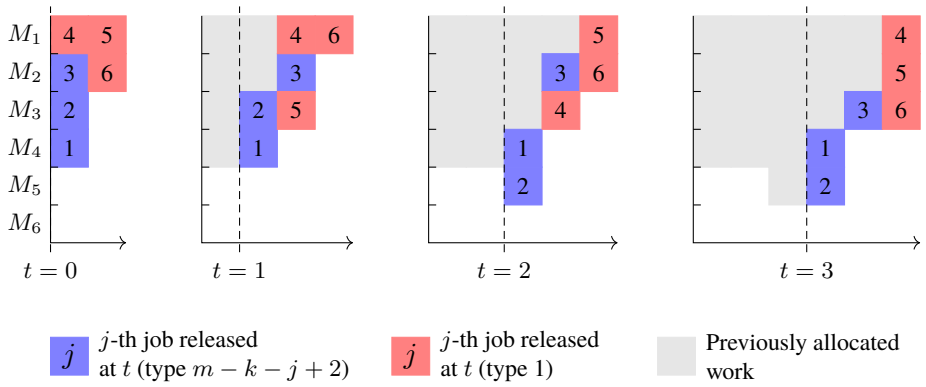


Figure 3.3 – An EFT-MIN schedule of the adversary from time $t = 0$ to $t = 3$, for $m = 6$ and $k = 3$. Colored jobs are released in-order at each time t.

(i) for $1 \leq j \leq m - k$, the job j is of type $m - k - j + 2$ (blue job in Figure 3.3), and

(ii) for $m - k < j \leq m$, the job j is of type 1 (red job in Figure 3.3).

This adversary relies on the key observation that EFT-MIN is naive: when several machines present the minimum load value among all machines, it will choose the first machine that satisfies its load-minimality criterion, i.e., the machine whose index is the lowest.

Note that at time t, just before sending the m next jobs, mt jobs have already been scheduled in π^{EFT}, and each machine i completes at time $\mathcal{C}_{i,mt}$. Let $w_t(i) = \max(0, \mathcal{C}_{i,mt} - t)$ be the work allocated on machine i and waiting to be processed, just before the adversary releases the m jobs. We call w_t the *schedule profile* of EFT at time t.

The proof consists in showing that EFT-MIN converges to a stable schedule profile w_τ such that for all machines i, we have

$$w_\tau(i) = \min(m - i, m - k).$$

Figure 3.4 shows an example of a schedule profile w_t, which is behind the stable profile w_τ.

Definition 3.3. *For any $t \neq t'$, we say that*

(i) $w_t = w_{t'}$ *if* $w_t(i) = w_{t'}(i)$ *for all* i,

(ii) $w_t \leq w_{t'}$ *if* $w_t(i) \leq w_{t'}(i)$ *for all* i (w_t *is* behind $w_{t'}$), *and*

(iii) $w_t < w_{t'}$ *if* $w_t \leq w_{t'}$ *and there is at least one machine* i *such that* $w_t(i) < w_{t'}(i)$ (w_t *is* strictly behind $w_{t'}$).

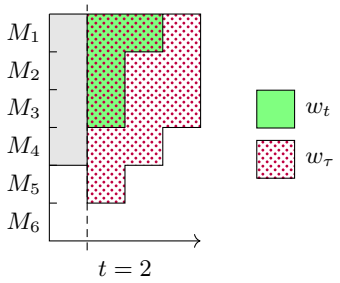


Figure 3.4 – The schedule profile w_t of EFT-Min at time t (in green), just before the adversary sends m new jobs. w_t is strictly behind the stable profile w_τ we want to reach (in purple).

The proof consists in two phases. First, we show that when the schedule profile is strictly behind w_τ, there exists a future time such that the schedule profile is closer to w_τ (Theorem 3.15). Second, we show that at any time, either we can find a past time such that the schedule profile exceeds w_τ, or the current schedule profile is behind w_τ (Theorem 3.16).

Before we dive into the proof, we start with the following lemma, which proves that for any t, w_t is a non-increasing function. This will be of particular importance when proving Theorem 3.15.

Lemma 3.14. *At any time t and for all machines i such that $1 \leq i < m$, $w_t(i+1) \leq w_t(i)$.*

Proof: Let us proceed by induction.

Base case ($t = 0$). No job arrived yet, so $w_0(i) = 0$ for all machines i.

Induction step. Now we assume that for a given time t, $w_t(i+1) \leq w_t(i)$ for all machines i such that $1 \leq i < m$. By contradiction, suppose there exists i such that $w_{t+1}(i+1) > w_{t+1}(i)$. We begin by showing that, as a consequence, only one job can have been scheduled on machine $i+1$ at time t, which will lead to a contradiction.

Let j be the last allocated job on machine $i+1$. By induction hypothesis, we know that $w_t(i+1) \leq w_t(i)$, and we assumed that $w_{t+1}(i+1) > w_{t+1}(i)$, thus j has been scheduled at a time comprised between t and $t+1$ (let $t+\varepsilon$ denote this specific time).

If we had $w_{t+\varepsilon}(i) > w_{t+\varepsilon}(i+1)$, then j could not be the last allocated job on $i+1$ at time $t+1$, because we assumed that $w_{t+1}(i+1) > w_{t+1}(i)$, and EFT-Min is an immediate dispatch algorithm. Therefore, we necessarily had $w_{t+\varepsilon}(i) \leq w_{t+\varepsilon}(i+1)$ at time $t+\varepsilon$, just before scheduling j. We can deduce that we have $\mathcal{M}_j = \{i+1, \cdots, i+k\}$, otherwise we would have scheduled j on the less-loaded machine i (then we say that j is of type $i+1$). Furthermore, all machines $i+2, \cdots, i+k$ were at least as much loaded as machine $i+1$ at time $t+\varepsilon$ ($w_{t+\varepsilon}(i') \geq w_{t+\varepsilon}(i+1)$ for all i' such that $i+1 < i' \leq i+k$), otherwise we would not have scheduled job j on machine $i+1$.

By construction of the adversary, the jobs sent before j at time t cannot have been placed on machine $i+1$ because their processing interval starts after $i+1$ (they are of type $\geq i+2$). Moreover, the jobs sent after j at time t cannot have been placed on $i+1$ as well (otherwise, j would not be the last job on $i+1$). Hence, j is the only job scheduled on $i+1$ between times t and $t+1$.

We consider two cases:

(i) First, suppose that $w_t(i+1) = 0$. This means that EFT-Min makes j starting at time t on $i+1$, and then j completes at time $t+1$. We proved that j is the only job that is scheduled on $i+1$ at time t, and as it completes at time $t+1$, we can say that there is no remaining work at this exact time, i.e., $w_{t+1}(i+1) = 0$. This contradicts our hypothesis $w_{t+1}(i+1) > w_{t+1}(i) \geq 0$.

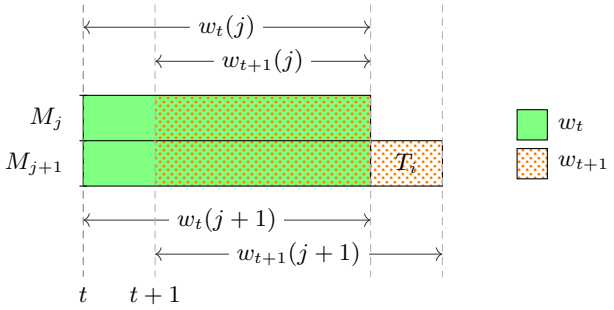


Figure 3.5 – Schedule profiles on M_j and M_{j+1} at times t and $t+1$, under the described hypotheses.

(ii) Now suppose that $w_t(i+1) > 0$. Following our two hypotheses $w_t(i+1) \leq w_t(i)$ and $w_{t+1}(i+1) > w_{t+1}(i)$, and from the consequent fact that exactly one job has been scheduled on $i+1$ between t and $t+1$, the only way to match all these assumptions is when there is as much waiting work on i as on $i+1$ at time t, i.e., $w_t(i) = w_t(i+1)$, and no job is scheduled on i between t and $t+1$. Figure 3.5 helps to visualize the described situation.

But the last job j scheduled on $i+1$ is of type $i+1$, which means that, by construction, at least one job of type i arrived after j at time t. We showed that all machines $i+2, \cdots, i+k$ were at least as much loaded as $i+1$ at time $t+\varepsilon$, thus they were at least as much loaded as i. As a consequence, at least one job must have been scheduled by EFT-Min on i between $t+\varepsilon$ and $t+1$, which is a contradiction. ■

Now we are able to show the first part of our proof: when the schedule profile of EFT-Min is strictly behind the stable profile w_τ, there must exist a future time t' such that the waiting work volume is greater than the current volume, i.e., $\sum_i w_t(i) < \sum_i w_{t'}(i)$.

Lemma 3.15. *At any time t such that $w_t < w_\tau$, there exists a time $t' > t$ such that*

(i) for all t_1 such that $t \leq t_1 < t'$, we have $\sum_i w_{t_1}(i) = \sum_i w_t(i)$, and

(ii) $\sum_i w_t(i) < \sum_i w_{t'}(i)$.

Proof: **Idleness property.** The first thing to notice is that when w_t is empty for a given machine that is not the last one, i.e., there exists $i < m$ such that $w_t(i) = 0$, we know that all subsequent machines have no remaining work to do as well: $w_t(i') = 0$ for all $i' > i$ (Theorem 3.14). When it happens, EFT-Min will not schedule any job on the last machine, because the only eligible job is the first one, which is of type $m-k+1$; that job will be scheduled on the first lightly-loaded compatible machine (with index $\max(i, m-k+1)$) at time t.

Therefore, m new jobs are released by the adversary and at most $m-1$ jobs are processed (no work can be done by the last machine), so we have

$$\sum_i w_{t+1}(i) \geq \sum_i w_t(i) + m - (m-1),$$

and thus $\sum_i w_{t+1}(i) > \sum_i w_t(i)$.

This means that if $w_t(i) = 0$ for some $i < m$, we have $\sum_i w_{t+1}(i) > \sum_i w_t(i)$. The proof is mainly based on this useful property that we call the *Idleness Property*. If the schedule profile is strictly behind

w_τ, we will show that there must exist a plateau on some machines, and this plateau will necessarily propagate on next machines step by step, until we reach a time t such that $w_t(m-1) = 0$.

Existence of a plateau. Now suppose that the schedule profile w_t is strictly behind w_τ ($w_t < w_\tau$). By Definition 3.3, this means that $w_t(i) \leq w_\tau(i)$ for all i, and there is at least one machine i' such that

$$w_t(i') < w_\tau(i'). \tag{3.7}$$

Let i' be the highest index of such a machine. Then we have

$$w_t(i) = w_\tau(i) \text{ for all } i > i', \tag{3.8}$$

and in particular, $w_t(i'+1) = w_\tau(i'+1)$. Let us show that there is a plateau on i' and $i'+1$ at time t, i.e., that $w_t(i') = w_t(i'+1)$. First, note that by definition of w_τ, we have

$$w_\tau(i) = w_\tau(i+1) + 1 \text{ for all } k \leq i < m. \tag{3.9}$$

We have $i' \geq k$, because if $w_t(i) < w_\tau(i)$ for some $i < k$, schedule profiles of all machines $i+1, \cdots, k$ are also strictly behind the stable profile w_τ (by Theorem 3.14 and definition of w_τ), and we defined i' as the highest index. Furthermore, $i' < m$, because we assumed that $w_t(i') < w_\tau(i')$ and by definition, $w_\tau(m) = 0$ ($w_t(m)$ cannot be lower than 0). Therefore,

$$\begin{aligned} w_t(i'+1) \leq w_t(i') &< w_\tau(i') && \text{(by Theorem 3.14 and eq. (3.7))} \\ &< w_\tau(i'+1) + 1 && \text{(by Equation (3.9))} \\ &< w_t(i'+1) + 1 && \text{(by Equation (3.8))} \end{aligned}$$

which gives

$$w_t(i'+1) \leq w_t(i') \leq w_t(i'+1), \tag{3.10}$$

and then $w_t(i') = w_t(i'+1)$.

By definition, $w_\tau(m) = 0$, and as $w_t < w_\tau$, $w_t(m) = 0$. If $i' = m-1$, we have $w_t(i') = w_t(m-1) = w_t(m) = 0$, and by the Idleness Property, the conclusion is immediate. Otherwise, $i' < m-1$, so $w_t(m-1) = w_\tau(m-1) = 1$. By Theorem 3.14, $w_t(i) \geq 1$ for all $i < m$, and then EFT-Min will schedule the first job on the last machine. Overall, m jobs will be processed at time t, and m jobs are sent by the adversary. Therefore, $\sum_i w_{t+1}(i) = \sum_i w_t(i) - m + m = \sum_i w_t(i)$.

Propagation of the plateau. Now we show that the plateau propagates on the next machine in the next step, i.e., as $i' < m-1$, $w_t(i') = w_t(i'+1)$ implies $w_{t+1}(i'+1) = w_{t+1}(i'+2)$.

By Equation (3.8), $w_t(i) = w_\tau(i)$ for all $i > i'$, which means that the first $m - i' - 1$ jobs will be scheduled on their last machine ($\mu_{mt+j} = m - j + 1$ for each $1 \leq j < m - i'$). The corresponding machines will process one job at time t. Thus, $w_{t+1}(i) = w_t(i) - 1 + 1 = w_t(i)$ for all $i > i' + 1$, and in particular,

$$w_{t+1}(i'+2) = w_t(i'+2). \tag{3.11}$$

As $w_t(i') = w_t(i'+1)$, the $(m-i')$-th job will not be scheduled on $i'+1$ (the index of the machine it will be placed on is at most i': $\mu_{mt+m-i'} < i'+1$). All remaining jobs will be scheduled on machines $1, \cdots, i'$, because they are of type $\leq i' - k + 1$. Figure 3.6 shows the propagation process.

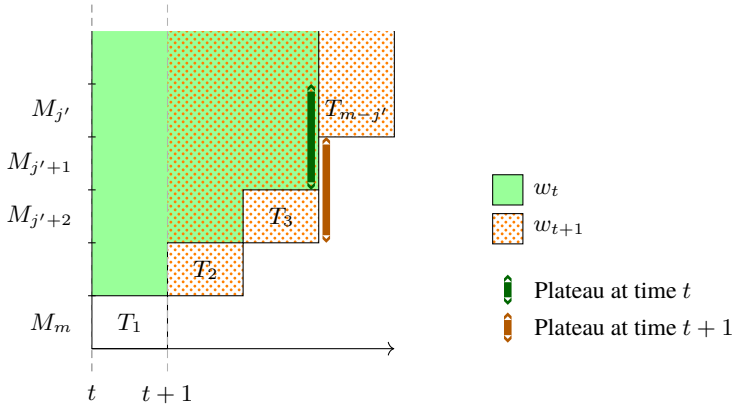


Figure 3.6 – Propagation of the plateau from machines $M_{j'}$ and $M_{j'+1}$ at time t to machines $M_{j'+1}$ and $M_{j'+2}$ at time $t+1$.

Then $i'+1$ does not receive any additional job at time t, but it still processes one job at this time, so we have

$$\begin{aligned} w_{t+1}(i'+1) &= w_t(i'+1) - 1 \\ &= w_\tau(i'+1) - 1 && \text{(by Equation (3.8))} \\ &= w_\tau(i'+2) && \text{(by Equation (3.9))} \\ &= w_t(i'+2) && \text{(by Equation (3.8))} \\ &= w_{t+1}(i'+2). && \text{(by Equation (3.11))} \end{aligned}$$

This shows that the plateau propagates on machines $i'+1$ and $i'+2$ at time $t+1$. By repeating the process, we reach a time at which $i'+1 = m-1$ and $i'+2 = m$, thus $w_t(m-1) = w_t(m) = 0$, and the Idleness Property applies. This concludes the proof. ■

The second phase of our proof consists in showing that either there exists a past time such that the schedule profile exceeds the stable profile w_τ, or the current profile is behind w_τ.

Lemma 3.16. *At any time t, either*

(i) there exists a time $t' \leq t$ such that $w_{t'}(i) > m - k$ for some i, or

(ii) $w_t \leq w_\tau$.

Proof: Let us proceed by induction.

Base case ($t = 0$). Obviously, the base case is true ($w_0 = 0 \leq w_\tau$).

Induction step (Case (i)). First suppose there exists a time $t' \leq t$ such that $w_{t'}(i) > m - k$ for some i. This is obviously still true at time $t+1$.

Induction step (Case (ii)). Now suppose that $w_t \leq w_\tau$ for some t. By contradiction, let us assume that there exists a machine i such that $w_{t+1}(i) > w_\tau(i)$. Combined to the fact that $w_t(i) \leq w_\tau(i)$, we have $w_{t+1}(i) \geq w_t(i) + 1$. Let q denote the number of jobs scheduled on i at time t, such that $w_t(i) + q - 1 = w_{t+1}(i)$. Then, $w_t(i) + q - 1 \geq w_t(i) + 1$, i.e., $q \geq 2$.

So at least 2 jobs must have been scheduled on machine i at time t. Let i be the highest index of such a machine. Two subcases arise:

(a) $i \leq k$. Then by construction $w_\tau(i) = m-k$, and we have $w_{t+1}(i) > w_\tau(i) = m-k$. This proves the induction.

(b) $i > k$. By induction hypothesis, we know that $w_t(m) = 0$ (because $w_\tau(m) = 0$), and by construction, at most one job can be scheduled on the last machine at time t. Therefore, $w_{t+1}(m) = 0$, so $i < m$. Let j be the last allocated job on i, with σ_j its starting time:

$$\sigma_j = t + 1 + w_{t+1}(i) - 1 = t + w_{t+1}(i).$$

Let λ_j be the type of j, i.e., $\mathcal{M}_j = \{\lambda_j, \cdots, \lambda_j + k - 1\}$. As j has been allocated to i, we necessarily have $\lambda_j \leq i \leq \lambda_j + k - 1$.

Suppose $\lambda_j = i$. By construction, all jobs sent before j at time t cannot have been scheduled on i, because their machine interval starts after λ_j. As j is the last job of i, no job sent after j at time t can have been scheduled on this machine. Then j is the only job scheduled on i between t and $t+1$, which contradicts the fact that at least 2 jobs must have been scheduled on i. Hence, $\lambda_j < i$.

Now, as j has been allocated on i and not on $i-1$, we know there was already a job j' on $i-1$ when the scheduling of j occurred, with $\sigma_{j'} = \sigma_j = t + w_{t+1}(i)$.

At time t, just before the adversary releases the m jobs, $i-1$ completes at time $\mathcal{C}_{i-1,mt} = t + w_t(i-1)$. We have $w_t(i-1) \leq w_\tau(i-1)$ (induction hypothesis), and we supposed that $w_{t+1}(i) > w_\tau(i)$. Finally, $w_\tau(i-1) = w_\tau(i) + 1$ (by construction of w_τ). Therefore,

$$\begin{aligned} t + w_t(i-1) &\leq t + w_\tau(i-1) \\ &\leq t + w_\tau(i) + 1 \\ &< t + w_{t+1}(i) + 1 = \sigma_{j'} + 1, \end{aligned}$$

which means that $\sigma_{j'} \geq t + w_t(i-1)$. In other words, j' starts after time $\mathcal{C}_{i-1,mt}$. Hence, the scheduling of j' occurred between t and $t+1$, before the scheduling of j ($t \leq \rho_{j'} < \rho_j < t+1$). Let $\lambda_{j'}$ be the type of j'. We necessarily have $\lambda_{j'} > \lambda_j$.

If $k = 2$, this is a contradiction, because j' cannot have been scheduled on $i-1$ (we proved that $\lambda_j < i$, so $\mathcal{M}_j = \{i-1, i\}$, and then $\lambda_{j'} > i-1$).

If $k > 2$, we deduce that j' has been scheduled on $i-1$ because all machines $i, \cdots, \lambda_{j'} + k - 1$ are planned to finish at or after time $\sigma_{j'}$. In particular, we have

$$\mathcal{C}_{i+1,j'} \geq \sigma_{j'} \geq t + w_{t+1}(i) > t + w_\tau(i).$$

Then,

$$\mathcal{C}_{i+1,j'} - 1 > t + w_\tau(i) - 1 = t + w_\tau(i+1).$$

Moreover, $\mathcal{C}_{i+1,j'} \leq \mathcal{C}_{i+1,m(t+1)} = t + 1 + w_{t+1}(i+1)$. Therefore,

$$t + w_{t+1}(i+1) > t + w_\tau(i+1),$$

i.e., $w_{t+1}(i+1) > w_\tau(i+1)$. This is a contradiction, because we had chosen i to be the highest index such that $w_{t+1}(i) > w_\tau(i)$. This concludes the proof. ■

Proof of Theorem 3.13: To exhibit the lower bound of $m - k + 1$ on the competitive ratio of EFT-Min, we first show there exists a time t such that $w_t = w_\tau$ or $w_t(i) > m - k$ for some i.

For a given time t, we know by Theorem 3.16 that either

(i) there exists a time $t' \leq t$ such that $w_{t'}(i) > m - k$ for some i or

(ii) $w_t \leq w_\tau$.

If Case (i) is true, then we found a time t such that $w_t(i) > m - k$ for some i. If Case (ii) is true, either $w_t < w_\tau$ or $w_t = w_\tau$. Suppose that $w_t < w_\tau$. Then by Theorem 3.15, we can find a future time t' such that $\sum w_{t'}(i) > \sum w_t(i)$. Therefore, while we have $w_t < w_\tau$, we can always find a future time such that the schedule profile is closer to w_τ. If we proceed step by step, we necessarily reach a time t' such that $w_{t'} = w_\tau$. This proves our initial claim.

Now, if $w_t(i) > m - k$ for some t, i, there exists a job j such that $F_j \geq m - k + 1$. If $w_t = w_\tau$ for some t, EFT-MIN will schedule one job on each machine (by definition of the adversary and w_τ). Hence, the k last jobs will be allocated on the k first machines, and they will have a flow time of $m - k + 1$. In any case, we have $F_{\max} \geq m - k + 1$.

On the described instance, at each time step, the optimal strategy consists in scheduling each job of type $\geq k + 1$ on the compatible machine of highest index. This allows reserving the k first machines to the k last jobs, and avoid any delay accumulation. Therefore, for all jobs j, we have $F_j^{\mathrm{OPT}} = 1$, and then $F_{\max}^{\mathrm{OPT}} = 1$. ■

The previous bound on the competitive ratio of EFT-MIN can be extended to the case where EFT uses a random tie-break function RAND, and we call this algorithm EFT-RAND (Algorithm 8). The only condition for Theorem 3.17 to hold is that among a set of candidate machines, the random tie-break function chooses each machine with positive probability, i.e., no machine is systematically discarded when it is a possible candidate.

Algorithm 8 EFT-RAND

```
1: when a new job j is released do
2:     Get U'_j according to completion times of machines M_j (Equation (3.4))
3:     u ← RAND(U'_j)
4:     μ_j ← u
5:     σ_j ← max(r_j, C_{u,j−1})
6:     Update the completion time of machine u
```

Theorem 3.17. *The competitive ratio of* EFT-RAND *is at least* $m-k+1$ *(almost surely) for* $P \,|\, \mathcal{M}_j(k\text{-interval}), \text{online-}r_j, 1 \,|\, F_{\max}$, *where* $1 < k < m$. *In other words, there exists an instance for which we have*

$$\mathbf{P}\left(F_{\max} \geq (m - k + 1)F_{\max}^{\mathrm{OPT}}\right) = 1.$$

Before starting the proof, we define the weighted distance on machine i at time t as

$$\varphi_t(i) = 2^{w_\tau(i)}(m - k + 1 - w_t(i)).$$

For any i_1, i_2 such that $1 \leq i_1 \leq i_2 \leq m$, the partial weighted distance between i_1 and i_2 at time t is defined as

$$\Phi_t(i_1, i_2) = \sum_{i=i_1}^{i_2} \varphi_t(i),$$

and the total weighted distance is denoted by $\Phi_t = \Phi_t(1, m)$. Intuitively, this distance quantifies the proximity between the schedule at time t and a simplified version of the stable schedule profile w_τ. In Theorem 3.18, we show that this distance decreases with t.

Lemma 3.18. *At any time t,*

(i) if there exists a job $j \leq m-k$ released at t and that is not scheduled on its last machine, i.e., $\mu_{mt+j} \neq m-j+1$, then $\Phi_{t+1} < \Phi_t$,

(ii) otherwise $\Phi_{t+1} \leq \Phi_t$.

Proof: **Case (i).** At a given time t, suppose there exists at least one job $j \leq m-k$ released at t and that is not scheduled on its last machine, i.e., $\mu_{mt+j} \neq m-j+1$. Let j be the highest index of such a job. We will study the value of $\Phi_t - \Phi_{t+1}$ in two steps: first, the value of $\Phi_t(1, m-j) - \Phi_{t+1}(1, m-j)$ on machines $1, \cdots, m-j$; second, the value of $\Phi_t(m-j+1, m) - \Phi_{t+1}(m-j+1, m)$ on machines $m-j+1, \cdots, m$.

From 1 to $m-j$. We choose j to be the highest index such that job j is not put on its last machine. This means that all jobs j' such that $j < j' \leq m-k$ are scheduled on their last machine $m-j'+1$, and the last k jobs are scheduled on any of the first k machines (because they are of type 1). In summary, all jobs j' such that $j \leq j' \leq m$ are scheduled on the first $m-j$ machines, and there are $m-j+1$ such jobs.

Any machine i among $1, \cdots, m-j$ can process at most 1 job between t and $t+1$. Let $q_{t,i}$ be the number of jobs released at time t and scheduled on i. Hence, $w_{t+1}(i) \geq w_t(i) - 1 + q_{t,i}$, and then

$$2^{w_\tau(i)}(m-k+1-w_{t+1}(i)) \leq 2^{w_\tau(i)}(m-k+1-w_t(i)+1-q_{t,i}).$$

Therefore, $\varphi_{t+1}(i) \leq \varphi_t(i) + 2^{w_\tau(i)} - 2^{w_\tau(i)} q_{t,i}$. By summing over i, we have

$$\sum_{i=1}^{m-j} \varphi_{t+1}(i) \leq \sum_{i=1}^{m-j} \varphi_t(i) + \sum_{i=1}^{m-j} 2^{w_\tau(i)} - \sum_{i=1}^{m-j} 2^{w_\tau(i)} q_{t,i}.$$

Note that

$$\sum_{i=1}^{m-j} 2^{w_\tau(i)} q_{t,i} \geq \sum_{j'=j}^{m} 2^{w_\tau(\mu_{mt+j'})},$$

as we have shown that at least the last $m-j+1$ jobs released at t are scheduled on the first $m-j$ machines. Then,

$$\Phi_{t+1}(1, m-j) \leq \Phi_t(1, m-j) + \sum_{i=1}^{m-j} 2^{w_\tau(i)} - \sum_{j'=j}^{m} 2^{w_\tau(\mu_{mt+j'})}. \tag{3.12}$$

Now we notice that

$$\begin{aligned}
\sum_{j'=j}^{m} 2^{w_\tau(\mu_{mt+j'})} &= 2^{w_\tau(\mu_{mt+j})} + \sum_{j'=j+1}^{m-k} 2^{w_\tau(\mu_{mt+j'})} + \sum_{j'=m-k+1}^{m} 2^{w_\tau(\mu_{mt+j'})} \\
&= 2^{w_\tau(\mu_{mt+j})} + \sum_{i=k+1}^{m-j} 2^{m-i} + \sum_{i=1}^{k} 2^{m-k} \\
&= 2^{w_\tau(\mu_{mt+j})} + \sum_{i=1}^{m-j} 2^{w_\tau(i)}.
\end{aligned}$$

Finally, by simplifying Equation (3.12),

$$\Phi_t(1, m-j) - \Phi_{t+1}(1, m-j) \geq 2^{w_\tau(\mu_{mt+j})}. \tag{3.13}$$

From $m-j+1$ to m. We saw earlier that the last $m-j+1$ jobs released at time t must have been scheduled on the first $m-j$ machines. We deduce that only the first $j-1$ jobs can have been put on the last j machines. There are more machines than jobs. Therefore, there exists at least one machine i such that $i > m-j$ that did not receive any job at time t. The machine i can process at most one job between t and $t+1$, so we have $w_{t+1}(i) \geq w_t(i) - 1$, and then

$$\varphi_t(i) - \varphi_{t+1}(i) \geq -2^{w_\tau(i)}.$$

In the worst case, all machines i such that $i > m-j$ receive no job. Then we have

$$\sum_{i=m-j+1}^{m} (\varphi_t(i) - \varphi_{t+1}(i)) \geq -\sum_{i=m-j+1}^{m} 2^{w_\tau(i)},$$

and then

$$\Phi_t(m-j+1, m) - \Phi_{t+1}(m-j+1, m) \geq -\sum_{i=m-j+1}^{m} 2^{w_\tau(i)}. \tag{3.14}$$

Now we sum Equations (3.13) and (3.14), and we get

$$\Phi_t - \Phi_{t+1} \geq 2^{w_\tau(\mu_{mt+j})} - \sum_{i=m-j+1}^{m} 2^{w_\tau(i)}.$$

Because $\mu_{mt+j} \leq m-j$, we have $2^{w_\tau(\mu_{mt+j})} \geq 2^{w_\tau(m-j)}$, and as $j \leq m-k$, $2^{w_\tau(m-j)} = 2^j$ and $m-j+1 \geq k+1$. Therefore,

$$\Phi_t - \Phi_{t+1} \geq 2^j - \sum_{i=m-j+1}^{m} 2^{m-i} = 2^j - \sum_{i'=0}^{j-1} 2^{i'} = 2^j - (2^j - 1) = 1,$$

and we conclude that $\Phi_t - \Phi_{t+1} > 0$.

Case (ii). Now suppose that at a given time t, all jobs $j \leq m-k$ released at t are scheduled on their last machine, i.e., $\mu_{mt+j} = m-j+1$.

From 1 to k. Only the last k jobs released at time t can have been put on the first k machines. Moreover, these machines can process at most k jobs between t and $t+1$. Hence,

$$\sum_{i=1}^{k} w_{t+1}(i) \geq \sum_{i=1}^{k} w_t(i) + k - k = \sum_{i=1}^{k} w_t(i),$$

and then

$$2^{m-k} \sum_{i=1}^{k} (m-k+1-w_{t+1}(i)) \leq 2^{m-k} \sum_{i=1}^{k} (m-k+1-w_t(i)),$$

which gives

$$\sum_{i=1}^{k} 2^{w_\tau(i)} (m-k+1-w_{t+1}(i)) \leq \sum_{i=1}^{k} 2^{w_\tau(i)} (m-k+1-w_t(i)).$$

Therefore,

$$\Phi_t(1,k) - \Phi_{t+1}(1,k) \geq 0. \tag{3.15}$$

From $k+1$ to m. All jobs $j \leq m-k$ are put on their last machines. Then all machines $k+1, \cdots, m$ receive exactly one job at time t, and we have $w_{t+1}(i) = w_t(i)$ for these machines, i.e., $\varphi_t(i) - \varphi_{t+1}(i) = 0$.

Hence,

$$\sum_{i=k+1}^{m} (\varphi_t(i) - \varphi_{t+1}(i)) = 0,$$

and then

$$\Phi_t(k+1,m) - \Phi_{t+1}(k+1,m) = 0. \tag{3.16}$$

By summing Equations (3.15) and (3.16), we get $\Phi_t - \Phi_{t+1} \geq 0$. ■

Now we prove that if we have no choice at a given time t (i.e., there is no tie-break) and if all jobs released at this time are put on their last machine, then we have reached a profile that is similar to the stable profile w_τ, where the load of machines decreases with their index.

Lemma 3.19. *At any time t, if $\mu_{mt+j} = m-j+1$ and $|U_{mt+j}| = 1$ for all jobs $j \leq m-k$ released at t, then $w_t(i+1) < w_t(i)$ for all $k \leq i < m$.*

Proof: Suppose that for a given time t, all jobs $j \leq m-k$ are scheduled on their last machine. This means that all machines $k+1, \cdots, m$ receive only one job at time t. Let j be such a job (we have $\mu_{mt+j} = m-j+1$). Suppose that there is no tie for j ($|U_{mt+j}| = 1$). By definition of the tie, we have

$$\mathcal{C}_{m-j+1,mt+j-1} < \mathcal{C}_{i,mt+j-1}$$

for all i such that $m-k-j+2 \leq i < m-j+1$. Moreover, we have $\mathcal{C}_{m-j+1,mt+j-1} = \mathcal{C}_{m-j+1,mt}$ and $\mathcal{C}_{i,mt+j-1} = \mathcal{C}_{i,mt}$, because all jobs $j' < j$ have been put on their last machine $m-j'+1$, and we have $m-j'+1 > m-j+1$.

Hence,

$$\mathcal{C}_{m-j+1,mt} < \mathcal{C}_{i,mt},$$

and then

$$t + w_t(m-j+1) < t + w_t(i),$$

which gives $w_t(m-j+1) < w_t(i)$. In particular, $w_t(m-j+1) < w_t(m-j)$. As this is true for all $1 \leq j \leq m-k$, we have $w_t(i+1) < w_t(i)$ for all $k \leq i < m$. ■

Before starting the proof of Theorem 3.17, we describe the class of random tie-break functions that we consider: RAND corresponds to any randomized policy for which there exists a constant $\theta > 0$ such that the probability to put any job on its last machine is lower than or equal to $1-\theta$, if there exists a tie for this job. In other words, RAND never discards a candidate machine during a tie.

Proof of Theorem 3.17: It is clear from Theorem 3.18 that Φ is non-increasing: at any time t, $\Phi_{t+1} \leq \Phi_t$. Then there are two cases. Either

(i) for all time t, we can find $t' > t$ such that $\Phi_{t'} < \Phi_t$, or

(ii) there exists a time t such that $\Phi_{t'} = \Phi_t$ for all $t' > t$.

Case (i). Suppose that for all t, there exists a future time $t' > t$ such that $\Phi_{t'} < \Phi_t$. As $\Phi_t \in \mathbb{Z}$ for all t, there must exist a time t^* such that $\Phi_{t^*} \leq 0$, i.e., $\sum_i \varphi_{t^*}(i) \leq 0$. Then, there exists at least one i such that $\varphi_{t^*}(i) \leq 0$. By definition, we deduce that $m - k + 1 - w_{t^*}(i) \leq 0$, thus $w_{t^*}(i) \geq m - k + 1$.

The last scheduled job j on i will complete at time $t^* + m - k + 1$, and we have $r_j \leq t^*$. Therefore, $F_{\max} \geq F_j \geq m - k + 1$.

Case (ii). Now suppose that there exists a time t such that $\Phi_{t'} = \Phi_t$ for all future time $t' > t$. By contraposition of Theorem 3.18, for all $t' > t$, we have $\mu_{mt'+j} = m - j + 1$ for all $j \leq m - k$ released at time t', i.e., the first $m - k$ jobs released at each t' are put on their last machine.

We consider first the scenario in which for all $t' > t$, there is at least one job $j \leq m - k$ released at t' for which there is a tie (i.e., $\left|U_{mt'+j}\right| > 1$). Since the first $m - k$ jobs released at each t' are put on their last machine, it implies that RAND has selected the last machine through a tie-break for all $t' > t$. By definition, RAND schedules each such job on any other machine than its last one with a non-zero probability. Therefore, RAND repeatedly makes this decision an infinite number of time with a probability of zero and the initial scenario thus occurs with the same probability.

Then, with probability 1, there exists at least one time $t' > t$ such that $\left|U_{mt'+j}\right| = 1$ for all jobs $j \leq m - k$ released at t'. By Theorem 3.19, we have $w_{t'}(i + 1) < w_{t'}(i)$ for all $k \leq i < m$, i.e., $w_{t'}(k) \geq m - k$. Therefore, there exists a job scheduled on machine k and released before time t' that will necessarily complete at time $t' + m - k$. We conclude that $F_{\max} \geq t' + m - k - (t' - 1) = m - k + 1$. ■

Finally, this result holds for any tie-break function provided that jobs are not anymore of unitary duration.

Theorem 3.20. *The competitive ratio of* EFT *(with any tie-break policy) is at least* $m - k + 1$ *for* $P \,|\, \mathcal{M}_j(k\text{-}interval), online\text{-}r_j \,|\, F_{\max}$.

The proof relies on the same instance as in Theorem 3.13, with some additional jobs with smaller duration. Original jobs from the instance of Theorem 3.13 are called *regular* jobs. Our objective is to enforce the following property.

Property 3.4. *Consider a machine* i *at time* t*, right before the allocation of regular jobs released at* t*. During time interval from* $t - 1$ *to* t*,* i *has* $h \geq 0$ *regular jobs waiting for execution (excluding the eventual one that is already started). These jobs will be completed at time* $t + h + i\delta$.

The value of δ will be set later to a very small value so that

(i) m delays of δ is smaller than the duration of a regular job (1 time unit), and

(ii) the total volume of small jobs can be considered as negligible in the optimal solution.

Once a value of $\delta < 1/m$ is chosen, we set $\varepsilon < \delta/(2m)$. As we will see below, the $i\delta$ delays on each machine allow emulating the original EFT-MIN algorithm, which breaks ties among available machines by choosing the one with minimum index.

We now explain how small jobs are added to the original schedule. Consider any integer time t (including $t = 0$). We have two rounds of small jobs submitted at time t, right before the regular jobs. We consider the set of machines that do not process regular jobs during time interval from $t - 1$ to t (all machines in the case of $t = 0$). Let m_{idle} be the number of such machines.

Intuitively, we first submit m_{idle} small dummy jobs at time t that are scheduled by EFT using its tie-break policy (which we do not control). All dummy jobs have different durations such that there is

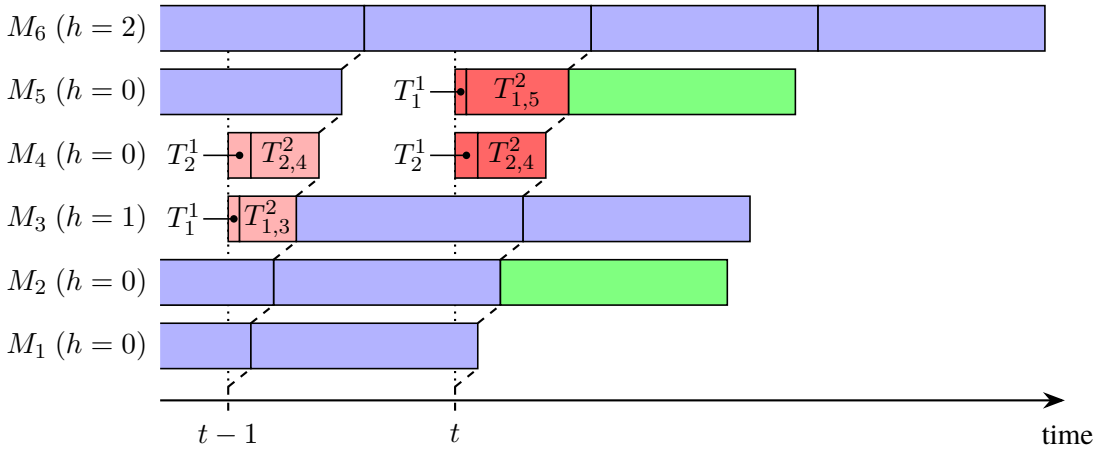


Figure 3.7 – Illustration of the construction of the instance for Theorem 3.20 when adding small jobs at time t. Regular jobs submitted before t are depicted in blue. Small jobs added to ensure the common delay of $i\delta$ are in red (dark red for step t, light red for step $t-1$), and regular jobs submitted at step t are in green. Only two processors are not processing any regular jobs before time t (M_4 and M_5) and require small jobs to ensure the common delay of $i\delta$.

no tie anymore among these machines for the second round. In the second round, we submit jobs whose duration is carefully crafted to ensure that each machine finishes its computation at the prescribed time $t + i\delta$.

First round. We first initialize a counter $c \leftarrow 1$. At time t, while there exists an idle machine $i_c \geq 0$, we submit a job j_c^1 of duration $c\varepsilon$ with an interval covering machine i_c (i.e. $i_c \in \mathcal{M}_{j_c^1}$, for example interval $i_c, \cdots, i_c + k$ if $i_c + k < m$, and $m - k, \cdots, m$ otherwise). We then increment the counter $c \leftarrow c + 1$.

Second round. When all jobs of the first round are submitted and allocated, we submit new jobs based on the allocation of the jobs of the first round. For each $c = 1, \ldots, m_{idle}$, we consider the machine i where the job j_c^1 of the first round has been allocated. We submit a job $j_{c,i}^2$ of duration $i\delta - c\varepsilon$ with an interval covering i (as above).

We now prove that Definition 3.4 is verified at all time, by induction on the time t. Let us first consider the beginning of the schedule ($t = 0$): small jobs are submitted for all idle machines i with $i \geq 0$, before the submission of regular jobs. Each job j_c^1 of the first round must be allocated and started at time $t = 0$ on some idle machine, not necessarily i_c. However, at the end of this first round, all machines must be processing a small job, as the scheduling algorithm never leaves a machine idle when there is some job to perform on it. Jobs submitted during the first round will complete at times $t + c\varepsilon$, with $c = 1, \ldots, m$. Thus, the latest completion time for the first round is equal to $t + m\varepsilon$.

We now move to the second round. Note that since $\varepsilon < \delta/(2m)$, the duration of a job $j_{c,i}^2$ of the second round is greater than $(i - c/(2m))\delta$ and is positive as $c < m$ and $i \geq 1$. Note also that i is the first machine available in the interval of $j_{c,i}^2$. On all other machines, either the small job of the first round completes later, or it has already been allocated a job of the second round, which lasts at least $i\delta - m\varepsilon > m\varepsilon$ and thus will complete later. Hence, job $j_{c,i}^2$ is necessarily allocated to i, and completes at time $t + (c\varepsilon) + (i\delta - c\varepsilon) = t + i\delta$. This proves the property for time $t = 0$.

We now prove the property for $t + 1$, assuming it is correct for t. We consider a machine i, which has $h \geq 0$ regular jobs waiting for execution during interval from $t - 1$ to t and $r \geq 0$ new regular jobs released at time t are allocated to i. We distinguish two cases:

- During interval from t to $t + 1$, i starts a regular job either because it has at least one waiting job

in interval ($h > 1$) or a new job released at time t is allocated to it ($r > 1$). By induction, the machine i will start this job at time $t + i\delta$ and end it at time $t + 1 + i\delta$. Excluding the started job, there remains $h' = h + r - 1 \geq 0$ waiting jobs in interval from t to $t + 1$. All the regular jobs waiting for execution will be completed at time $t + 1 + i\delta + h' = (t + 1) + h' + i\delta$ with $h' \geq 0$. Hence, the property is true at time $t + 1$ for i.

- During interval from t to $t + 1$, i starts no regular job ($h = 0$ and $r = 0$). At time $t + 1$, all machines are either idle like i (when $h = 0$ and $r = 0$) or computing a regular job (allocated before $t+1$). i is allocated a small job j_c^1 in the first round at time $t+1$ (it is available by induction hypothesis) and completes at time $t + 1 + c\varepsilon$. Since there are at most m machines without regular jobs in interval from t to $t + 1$, all small jobs of the first round are completed before or at time $t + m\epsilon < t + \delta$. In the second round, we prove that the job $j_{c,i}^2$ must be allocated on i. As seen before, at time $t + 1 + c\varepsilon$, all idle machines either complete their jobs of the first round later than i or are already computing a job of the second round that completes later. Machines i' with $i' \geq 0$ that are computing regular jobs will be available at the soonest at time $t + i'\delta$ to start a regular job by induction hypothesis. Thus, each machine i' will complete at time $t + 1 + i'\delta$, which is much later than when i completes its job from the first round. Hence, $j_{c,i}^2$ is allocated to i, and completes at time $t + 1 + i\delta$.

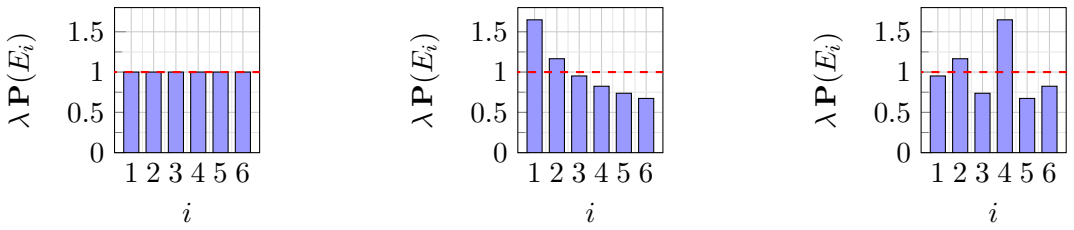
Lemma 3.21. *With the additional (non regular) jobs, the execution of any* EFT *algorithm (with any tie-break policy) follows the original* EFT *policy (with tie-break by selecting the machine with smallest index), up to a delay of $i\delta$ for each machine i.*

This lemma is proven by noticing that compared to the original setting, machines are not available simultaneously for regular jobs, but with a small delay $i\delta$ of increasing value for increasing machine index. Hence, whenever a regular job can be processed on several machines in the original setting, now the EFT policy forces the machine with smaller index to execute it, as it was done in the original EFT policy.

The instance used in the proof of Theorem 3.20 requires at most m^3 steps (each made of m jobs) to reach a maximum flow of $m - k + 1$ for the EFT-Min policy. The modified instance enforces such a maximum flow for a EFT scheduler with any tie-break policy. The total volume of small jobs added to this instance at each time step is bounded by $\sum_{i=1}^{m} i\delta = m(m+1)\delta/2$. Hence the total volume of small jobs during the whole instance is bounded by $m^5\delta/2$. Choosing $\delta = o(1/m^5)$ makes this total volume negligible is front of the duration of a single regular job. We consider an optimal schedule of the original schedule and allocate the additional small jobs to any machine of their interval. The maximum delay for any machine is of order $o(1)$. Hence the maximum flow of this modified optimal algorithm is $1 + o(1)$, which proves the asymptotic competitive ratio of $m - k + 1$.

3.4 A General Method to Bound the Throughput

In this section, we evaluate the relative impact of structured processing set restrictions on the performance of simple scheduling heuristics. We focus on both interval processing sets, because they are used in actual systems [11, 24, 38], and disjoint processing sets, because it is the restrictions for which we have the best, and only, approximation ratio (Theorem 3.11). Moreover, the performance of actual systems are affected by the popularity of requests, which is not uniform, i.e., certain jobs restricted to the same processing set appear more frequently than others. We begin by explaining our model of popularity before developing the process we used to evaluate the theoretical maximum throughput permitted by data item replication.



(a) Uniform case. (b) Worst-case. (c) Shuffled case.

Figure 3.8 – Example of load distribution on a cluster of $m = 6$ machines, with $\lambda = m$, for each case.

Finally, we perform simulations to provide an experimental perspective to the bounds derived in the previous section. All the related code, data and analysis are available online[1].

3.4.1 Modeling Key Popularity

Let us consider a cluster of m machines, where jobs have a unit processing time and are released according to a Poisson process with parameter λ (in other words, λ jobs are released in average at each time unit). The m machines can process at most m jobs at each time unit, hence λ/m measures the average load on the whole cluster, which is loaded at 100% if $\lambda = m$.

For now, suppose that each job can be processed by only one specific machine, i.e., we have $|\mathcal{M}_j| = 1$ for all jobs j. This corresponds to what happens in key-value stores when data items are not replicated: each job j carries a key, which is uniquely associated to a data item in the system, and this data item is held by only one machine of the cluster. Therefore, j has no choice but to be sent and processed on this specific machine.

In practice, some data items are requested more frequently than others during the service lifetime. Depending on the data partitioning and popularity bias on requested keys, some machines will potentially have to process more jobs than others, leading to a biased distribution on machine popularity. Let E_i be the event in which an arbitrary job must be processed by machine i (because it requests a key held by i), which occurs with probability $\mathbf{P}(E_i)$. Thus, $\lambda \mathbf{P}(E_i)$ is the average number of jobs sent on i at each time unit, and measures the load of i. Note that because of the non-uniform popularity bias $\mathbf{P}(E_i)$, the load of a given machine can be greater than 100% (even if the average cluster load is below 100%). In this case, the machine completely saturates as there is no replication.

Let us consider that the machine popularity follows a Zipf distribution, which has been advocated to model popularity distributions [14]. We have $\mathbf{P}(E_i) = \frac{1}{i^s H_{m,s}}$, where $s \geq 0$ is the shape parameter of the distribution and $H_{m,s}$ is the m-th generalized harmonic number of order s. We use s to control the popularity bias: the larger s, the more the popularity heterogeneity increases. In the following, we focus on three specific situations. When $s = 0$, the distribution degenerates to the uniform distribution, i.e., no machine is more popular than another (we call this case the **Uniform case**). When $s > 0$, the Zipf distribution has the particularity to generate a monotonically decreasing load on machines $1, \cdots, m$. This corresponds to a worst case, as the first machines concentrate most of the workload (**Worst-case**). Finally, we randomly permute $\mathbf{P}(E_i)$ to match with more realistic settings (**Shuffled case**). As a realistic bias strongly depends on the dataset and system usage, each permutation is chosen uniformly as we assume no prior knowledge. Figure 3.8 shows an example of load distribution for each case.

[1] https://doi.org/10.6084/m9.figshare.19123139.v1

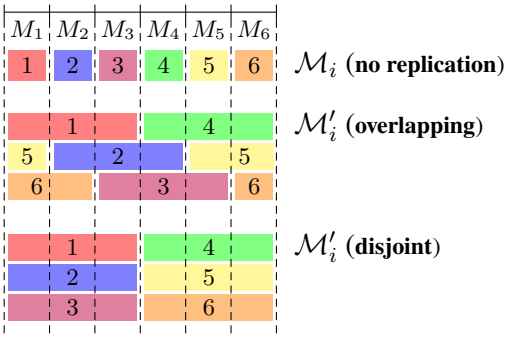


Figure 3.9 – Example of replication strategies in overlapping and disjoint settings, with $k = 3$. For example, suppose that a task T_i is feasible on M_3 only ($\mathcal{M}_i = \{M_3\}$). Then, in overlapping setting (resp. disjoint setting), the new processing set restriction of T_i is $\mathcal{M}'_i = \{M_3, M_4, M_5\}$ (resp. $\mathcal{M}'_i = \{M_1, M_2, M_3\}$).

3.4.2 Finding the Theoretical Maximum Throughput

We want to find the theoretical maximum throughput (that is, finding the maximum value of λ such that the load on each machine is below 100%) one can achieve when data items are replicated across the cluster. Up to now, as we did not consider replication yet, we supposed that each job could only be processed by a single machine (the one holding its requested key). In this case, we clearly have $\lambda \leq 1/\max_i \mathbf{P}(E_i)$.

Let us give more choices to each job by adding more machines to the processing sets $\mathcal{M}_j$. This can be seen as replicating data items. Our goal is to study how extending $\mathcal{M}_j$ under a popularity bias affects performance metrics such as the maximum flow time or the maximum throughput, and how structures in processing sets impact them.

For each job j, we build a new set $\mathcal{M}'_j$ from $\mathcal{M}_j$ by defining a replication strategy. In other words, starting from a set with a single machine u ($\mathcal{M}_j = \{u\}$), we replicate the keys held by u on all machines of $\mathcal{M}'_j$. We focus on strategies that consist in adding $k-1$ machines (with $1 \leq k \leq m$) to the set, such that $\mathcal{M}'_j$ constitutes an interval of size k, i.e., $\mathcal{M}'_j = I_k(u)$. We describe two manners to build $I_k(u)$ from u. Figure 3.9 illustrates these constructions.

Overlapping intervals. There are m distinct overlapping replication intervals of size k, arranged as a ring:

$$I_k(u) = \left\{ i \in M \text{ s.t. } i = (i' - 1) \bmod m + 1 \text{ for all } u \leq i' \leq u + k - 1 \right\}.$$

This constitutes the basic replication strategy of key-value stores. Machines are arranged as a ring, and data items held by a given machine are replicated on the successors of this machine [11, 24]. We have seen in Theorems 3.13, 3.17 and 3.20 that EFT does not always provide a good competitive ratio when minimizing maximum flow time with this structure.

Disjoint intervals. We divide the cluster into $\left\lceil \frac{m}{k} \right\rceil$ disjoint replication intervals of size k:

$$I_k(u) = \{i \in M \text{ s.t. } u' + 1 \leq i \leq \min(m, u' + k)\},$$

where $u' = k \left\lfloor \frac{u-1}{k} \right\rfloor$. This corresponds to the situation seen in Theorem 3.11 and related corollaries. EFT guarantees a good competitive ratio when minimizing maximum flow time with this structure.

After replication, all jobs that could only run on a given machine i can now be processed by any machine of $I_k(i)$. To quantify the gain on maximum cluster load permitted by a given replication strategy,

we solve the following optimization problem modeled as a linear program:

$$\textbf{maximize} \quad \lambda \tag{3.17a}$$

$$\text{subject to} \quad \forall j, \sum_i a_{ij} = \lambda \mathbf{P}(E_j), \tag{3.17b}$$

$$\forall i, \sum_j a_{ij} \leq 1, \tag{3.17c}$$

$$\forall i, j \text{ s.t. } i \notin I_k(j),\ a_{ij} = 0, \tag{3.17d}$$

$$\forall i, j, a_{ij} \geq 0, \tag{3.17e}$$

$$\lambda \geq 0 \tag{3.17f}$$

a_{ij} denotes the average amount of work (in jobs per time unit) that is eventually processed by machine i and that corresponds to jobs originally restricted to machine j. We consider the following constraints:

- The total work corresponding to jobs originally restricted on j is exactly equal to the initial work of j (Equation (3.17b)).
- The average work eventually processed on i does not exceed 1 (Equation (3.17c)).
- We can transfer work from j to i if and only if i belongs to the interval of size k generated from j according to the considered replication strategy, i.e., all jobs that could originally run exclusively on j can now also run on i (Equation (3.17d)).

3.4.3 Experimental Evaluation

In the following experiments, we set the cluster size m to 15, which is a common setup when conducting experiments with scheduling in real key-value store systems [20, 39]. Figure 3.10 shows the result of Linear Program 3.17 as a function of bias s and interval size k, for both previously described replication strategies, in the **Shuffled case** (median over 100 different permutations).

At first glance, it seems that the disjoint strategy is less efficient than the overlapping strategy to cope with high cluster load when non-uniform popularity biases are introduced. For example, for $s = 1$ and $k = 5$, Figure 3.10 indicates that the cluster can theoretically tolerate a maximum load of 100% when intervals overlap, whereas the disjoint strategy allows reaching a maximum load of 70%.

The overlapping strategy superiority is clearly confirmed by Figure 3.11, which shows the gain on the maximum load permitted by overlapping replication intervals over the disjoint strategy. The overlapping strategy allows the cluster to handle loads that are up to 50% higher than the disjoint strategy (e.g., for $s = 1.25$ and $k = 6$), and we can observe a gain up to 35% for common situations in key-value stores, when $0 < s \leq 1.5$ (moderate popularity bias) and $k = 3$ (standard replication factor in most implementations). Note that the popularity bias has obviously no effect when data are fully replicated ($k = m$), and that replication strategies exhibit no difference on the tolerable load when no bias is introduced ($s = 0$).

Now we simulate EFT scheduling on $m = 15$ machines with a popularity bias, on 10 000 generated unitary jobs, which is sufficient to reach a steady state. Figure 3.12 illustrates the impact of both replication strategies on maximum flow time in the EFT-Min scheduler and its counterpart EFT-Max (which selects the candidate machine with highest index). We consider the three cases of popularity bias (in **Worst-case** and **Shuffled case**, we set $s = 1$). We repeat the experiment 10 times, and we take the median among max-flow values. We set $k = 3$ to match with a realistic key-value store system.

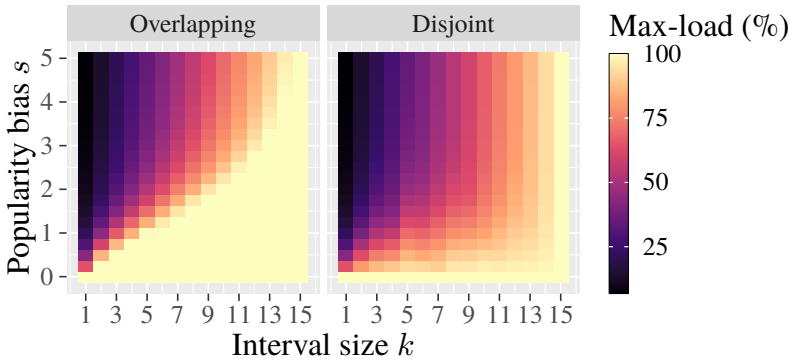


Figure 3.10 – Maximized load obtained by solving Linear Program 3.17, for both overlapping and disjoint strategies, for each $0 \leq s \leq 5$ (by steps of 0.25) and $1 \leq k \leq m$, in the **Shuffled case**.

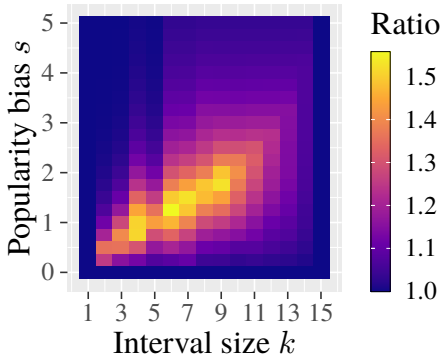


Figure 3.11 – Ratio between the max-load of overlapping and disjoint strategies, for each $0 \leq s \leq 5$ (by steps of 0.25) and $1 \leq k \leq m$, in the **Shuffled case**.

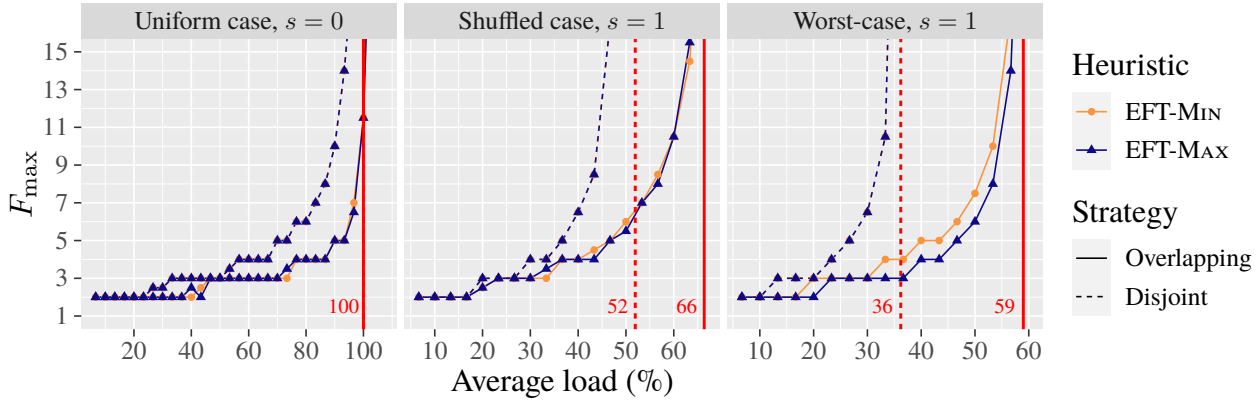


Figure 3.12 – Maximum flow time given by both heuristics EFT-Min and EFT-Max as a function of the average load when $k = 3$, for both the overlapping (solid lines) and disjoint (dashed lines) strategies. Each facet corresponds to a distinct case (for the **Worst-case** and **Shuffled case**, we set $s = 1$). Finally, each vertical red line shows the theoretical maximum load value given by Linear Program 3.17 in the corresponding case.

In the **Uniform case**, no difference is visible between EFT-Min and EFT-Max however, overlapping replication intervals give better results than the disjoint strategy (e.g., for an average cluster load of 90%, EFT exhibits a max-flow of 5 when intervals overlap, whereas it gives a max-flow of 10 with disjoint intervals). When randomly dispatched popularity biases are introduced (**Shuffled case**), we see the relative gain of the overlapping strategy increasing. This is even more obvious when we consider the **Worst-case**. We also see EFT-Max becoming more efficient than EFT-Min for the overlapping strategy, which is consistent with the situation in Theorem 3.13: when breaking a tie, EFT-Min will select the most popular machine, whereas EFT-Max does the opposite (as we are in a worst-case, popularity biases are sorted in decreasing order), leading to a smaller max-flow. However, the gain permitted by the scheduling heuristic is rather marginal compared to the gain allowed by a carefully chosen replication structure.

The replication strategy where intervals overlap, commonly used in key-value stores, exhibits better results than the disjoint strategy when popularity biases are introduced, even if the max-flow of EFT in disjoint setting is bounded (Theorem 3.11). However, there is no efficient worst-case guarantee for the overlapping strategy, as seen in Theorem 3.13. The question of whether there exists a replication strategy giving both good practical results and theoretical guarantees on EFT scheduling remains open.

3.5 Conclusion

4

Partitioning Multi-Get Requests

4.1 Introduction

4.2 Motivation & Model

In the problem of scheduling jobs on unrelated machines (also known as the $R\,||\,C_{\max}$ problem), we are given a set of n jobs $J = \{1, \ldots, n\}$ and a set of m machines $M = \{1, \ldots, m\}$, where each job $j \in J$ has processing time $p_{ij} > 0$ on machine $i \in M$. The objective is to schedule (non-preemptively) the jobs on machines so as to minimize the makespan, that is to say, the maximum completion time of the jobs.

The Restricted Assignment (RA) problem is a special case of $R\,||\,C_{\max}$, where each job $j \in J$ can be processed only on a subset of machines $\mathcal{M}_j \subseteq M$, which we call the *processing set* of j. In this setting, the job j has processing time p_j on machine i if and only if $i \in \mathcal{M}_j$, and $+\infty$ otherwise. The problem is often noted $P\,|\,\mathcal{M}_j\,|\,C_{\max}$ in Graham's notation. A given instance of the RA problem can be represented by a bipartite graph $G = (J \cup M, E)$, where there is an edge (j, i) with weight p_j between a job $j \in J$ and a machine $i \in M$ if and only if $i \in \mathcal{M}_j$.

The $R\,||\,C_{\max}$ problem, and more specifically the RA problem, are well-known NP-hard problems, and it has even been proved that no algorithm can approximate an optimal solution within a factor better than $3/2$ unless P $=$ NP [**lenstra1990**]. Hence, specific cases of the RA problem have also been the subject of extensive research. One possible manner to reduce the complexity of the problem is to bring structure in the processing sets of jobs. For example, a common subproblem consists in solving the RA problem on *nested*[1] processing sets, that is to say, one of the following properties holds for any two jobs $j, j' \in J$: $\mathcal{M}_j \subseteq \mathcal{M}_{j'}$, $\mathcal{M}_{j'} \subseteq \mathcal{M}_j$, or $\mathcal{M}_j \cap \mathcal{M}_{j'} = \emptyset$. An even more specific case is when the processing sets are *inclusive*, i.e., for any two jobs $j, j' \in J$, either $\mathcal{M}_j \subseteq \mathcal{M}_{j'}$, or $\mathcal{M}_{j'} \subseteq \mathcal{M}_j$.

[1] This special case is also known as *laminar* processing sets.

In this paper, we focus on *interval* processing sets, that is a subcase of the RA problem, but more general than the nested case. In this problem, the machines can be rearranged such that the processing sets of jobs consist in contiguous intervals of machines. More formally, let us note (a, b) the interval ranging from machine a (inclusive) to machine b (inclusive, $a \leq b$), and $I_{(a,b)} = \{a, a+1, \ldots, b\}$. In the Restricted Assignment problem on Intervals (RAI), for all jobs $j \in J$, we define $\mathcal{M}_j = I_{(a_j,b_j)}$, where a_j and b_j are respectively the lower and upper bounds of the interval of machines on which the job j can be assigned. In this case, an instance of the problem can be seen as a *convex* bipartite graph.

An applicative context in which the RAI problem arises is when partitioning multi-get requests in a distributed and replicated storage system. Key-value stores are low-latency databases where each data item is associated with a unique key, and a simple read operation consists in retrieving the value that corresponds to a given key [**lakshman2010**, **decandia2007**]. These databases are distributed on several nodes, as the dataset is often too large to fit on a single server. Thus, the dataset is split in several partitions, and each partition is stored on a different server. Moreover, in order to guarantee accessibility of data in case of node failure, each partition is replicated several times on different servers, and the duplicated data on a given node form a *replica* of the original partition. The replication strategy differs from one system to another, but a common and efficient way consists in arranging the servers on a virtual ring, and replicating the partition of each server i on its two successors $i + 1$ and $i + 2$ (modulo the number of servers). In other words, servers are virtually ordered, and each key/value couple is stored on an interval of three different consecutive servers.

Multi-get requests are read operations that involve several keys. These aggregated operations are useful, for instance, to reduce the number of network round-trips between the database and a web-server, where a single HTTP request often requires to retrieve several data items before responding to the client [**reda2017**, **jaiman2020**]. In such a multi-get request, the keys (which are called the *keyset* of the request) may be located in different partitions. Thus, the system must split the request into several subrequests by partitioning the keyset into several *opsets*, and redirect each opset toward the appropriate server (which must hold the keys of the opset, as a replica or not). As we cannot respond before executing all opsets, these opsets must be balanced to guarantee a good response time for the overall multi-get request, i.e., we do not want a few very fast opsets and one that is very slow.

This opset partitioning problem can be seen as the RAI problem, where servers are the machines, and each read operation for a given key is a job, whose processing time is the time required to retrieve the corresponding data item (which depends on the number of bytes that represent the stored item), and the processing set is the interval of servers on which the key is located. An opset is the set of keys whose corresponding jobs are assigned to the same machine. As we seek to minimize imbalance between opsets, we want to minimize the maximum completion time of each opset, which is the makespan. Of course, when partitioning a multi-get request, one can also take into account the current load of each machine by adding m fake jobs whose processing set consists of only one machine.

As the number of jobs and machines are relatively small in this context, a simple solution to this problem could consist in an integer programming formulation, where c is the makespan, and each x_{ij} is a binary variable that indicates whether the job j is assigned to the machine i:

$$
\begin{aligned}
\textbf{minimize} \quad & c \\
\text{subject to} \quad & \forall j \in J, \sum_{i \in M} x_{ij} = 1, \\
& \forall i \in M, \sum_{j \in J} x_{ij} p_j \leq c, \\
& x_{ij} \in \{0, 1\}\,.
\end{aligned}
$$

However, we must solve this problem for each multi-get request in real time, and key-value store systems

are usually dimensioned to handle high throughputs (of the order of thousands of requests per second). This approach is, therefore, clearly not scalable. We need a polynomial, guaranteed (even if not optimal), greedy algorithm, to ensure that the time taken to partition a multi-get request is not greater than the time required to execute the request itself.

Among the interesting, potentially simpler, variants of the RAI problem, we can mention the case with unitary jobs. This corresponds to near-identical requests in the key-value store, which may happen if all data items have similar sizes. Another variant is the case with K types of jobs, which corresponds to the case where we can categorize requests, for example by considering *small* and *large* data items.

4.3 The Restricted Assignment Problem on Intervals

Let us simplify the practical problem for now by considering only regular intervals of machines, i.e., we focus on the standard RAI problem. Lin et al. [**lin2004**] proposed a polynomial algorithm to solve the RAI problem when jobs are unitary. They argue that their algorithm runs in time $O(m(m+n))$, although we found their analysis to be slightly incorrect. We also noticed an error in their proof of optimality. In this section, we give a correct version of their proof, and we generalize their approach to derive the following results:

(i) an optimal algorithm for the RAI problem with unitary jobs, which runs in time $O(m^2 + n \log n + mn)$ (Theorem 4.3), and

(ii) a tight $(2 - 1/m)$-approximation algorithm for the RAI problem with arbitrary jobs, which also runs in time $O(m^2 + n \log n + mn)$ (Theorem 4.4).

Let us introduce Algorithm 9, called Estimated Least Flexible Job (ELFJ), which is directly inspired by Lin et al.'s algorithm. ELFJ takes a parameter λ and builds a schedule that is guaranteed to finish before time λ. In other words, λ denotes an upper bound on the optimal makespan: as λ gets closer to the actual optimal makespan, ELFJ gets closer to an optimal schedule. ELFJ performs two steps: first, it sorts the jobs in non-decreasing order of interval upper bound b_j (in time $O(n \log n)$), and then it assigns jobs on the machines (in time $O(mn)$).

Before starting the optimality analysis, let us introduce some notations and definitions. For any interval of machines (α, β), where $1 \leq \alpha \leq \beta \leq m$, we define $K_{(\alpha,\beta)}$ as the set of jobs whose processing set is included in $I_{(\alpha,\beta)}$, i.e., $K_{(\alpha,\beta)} = \left\{j \in J \text{ s.t. } \mathcal{M}_j \subseteq I_{(\alpha,\beta)}\right\}$. We denote the total processing time of jobs in $K_{(\alpha,\beta)}$ by $w_{(\alpha,\beta)}$, i.e., $w_{(\alpha,\beta)} = \sum_{j \in K_{(\alpha,\beta)}} p_j$. Let $\tilde{w}_{(\alpha,\beta)}$ represent the minimum average work that *any* schedule must perform on machines $\alpha, \ldots, \beta$, i.e.,

$$\tilde{w}_{(\alpha,\beta)} = \frac{w_{(\alpha,\beta)}}{\beta - \alpha + 1},$$

Algorithm 9 ELFJ

```
1: sort jobs in non-decreasing order of b_j
2: for each machine i do
3:     ρ_i ← 0
4:     for each non-assigned job j such that a_j ≤ i ≤ b_j do
5:         if ρ_i + p_j ≤ λ then
6:             assign j to i
7:             ρ_i ← ρ_i + p_j
```

and let $\tilde{w}_{\max}$ be the maximum value of $\tilde{w}_{(\alpha,\beta)}$ over all intervals ($\tilde{w}_{\max} = \max_{1\leq\alpha\leq\beta\leq m}\left\{\tilde{w}_{(\alpha,\beta)}\right\}$). From these definitions, we can easily derive a lower bound on the optimal makespan $C^{\mathrm{OPT}}_{\max}$ for a given instance $\mathcal{I}$ of the RAI problem, as shown by Lin et al. in their original work [**lin2004**].

Lemma 4.1. *The optimal makespan is bounded by $\tilde{w}_{\max}$, i.e., $C^{\mathrm{OPT}}_{\max} \geq \tilde{w}_{\max}$.*

Proof: Let $C^{\mathrm{OPT}}_{(\alpha,\beta)}$ be the maximum completion time of machines $\alpha,\ldots,\beta$ in an optimal schedule. We clearly have $C^{\mathrm{OPT}}_{(\alpha,\beta)} \geq \tilde{w}_{(\alpha,\beta)}$ for any interval (α,β), because all jobs in the set $K_{(\alpha,\beta)}$ must be done between machines α and β. In the best case, the jobs are perfectly balanced on $\beta-\alpha+1$ machines.

Let (a,b) be the interval of machines such that $\tilde{w}_{\max} = \tilde{w}_{(a,b)}$. Then we have $C^{\mathrm{OPT}}_{(a,b)} \geq \tilde{w}_{(a,b)}$. Of course, $C^{\mathrm{OPT}}_{\max} \geq C^{\mathrm{OPT}}_{(a,b)}$, i.e., $C^{\mathrm{OPT}}_{\max} \geq \tilde{w}_{\max}$. ■

Corollary 4.2. *If all processing times are integers, then $C^{\mathrm{OPT}}_{\max} \geq \lceil\tilde{w}_{\max}\rceil$.*

Proof: Suppose that all processing times are integers in the considered instance. Then we have $C^{\mathrm{OPT}}_{(\alpha,\beta)} \geq \lceil\tilde{w}_{(\alpha,\beta)}\rceil$ for any interval (α,β), because jobs are not divisible. The rest of the proof is analogous to the previous lemma. ■

The idea of Lin et al. is to use $\lceil\tilde{w}_{\max}\rceil$ as the value of the parameter λ in ELFJ to get an optimal schedule when jobs are unitary. This means that one must be able to compute $\tilde{w}_{\max}$ in polynomial time in order to apply the algorithm. Suppose for a moment that all processing times are unitary, i.e., for all interval (α,β), $w_{(\alpha,\beta)} = \left|K_{(\alpha,\beta)}\right|$. In the original paper, the authors propose the following procedure. First, for all machine i, construct the sets $A_i = \{j \in J \text{ s.t. } i \leq a_j\}$ and $B_i = \{j \in J \text{ s.t. } b_j \leq i\}$ in time $O(mn)$. Then, for all intervals (α,β), compute $\left|K_{(\alpha,\beta)}\right| = |A_\alpha \cap B_\beta|$. Counting the number of common elements in two sets is clearly not a constant-time operation. Hence, as there are $O(m^2)$ possible intervals, the time complexity of this procedure is at least $O(c(n)\cdot m^2)$, where $c(n)$ is the time complexity of counting common elements in two sets of size $O(n)$. If we recall that the original algorithm performs a sorting operation (in time $O(n\log n)$) and the assignment of jobs to machines (in time $O(mn)$), we conclude that the total complexity $O(m^2+mn)$ given by Lin et al. is underestimated, and we argue that their approach gives in fact an algorithm with time complexity $O(c(n)\cdot m^2 + n\log n + mn)$. Moreover, their computation method is not suitable to the case where processing times are arbitrary.

We present in this section a new procedure to compute $\tilde{w}_{\max}$ in time $O(m^2+n)$ for any instance of the RAI problem with arbitrary processing times. We notice that the set of intervals in a list of m machines can be represented by a graph, where nodes correspond to intervals. For all intervals (α,β) such that $\alpha < \beta$, the node (α,β) is the parent of two children nodes $(\alpha,\beta-1)$ and $(\alpha+1,\beta)$ (see Figure **??**).

Let $J_{(\alpha,\beta)}$ be the set of jobs whose processing set is exactly $I_{(\alpha,\beta)}$, i.e., $J_{(\alpha,\beta)} = \left\{j \in J \text{ s.t. } \mathcal{M}_j = I_{(\alpha,\beta)}\right\}$, and let $v_{(\alpha,\beta)}$ be their total processing time. Then we have a recursive relation between the values $w_{(\alpha,\beta)}$: for a given interval (α,β) that has two children intervals, the work that must be done on machines $\alpha,\ldots,\beta$ includes

(i) $J_{(\alpha,\beta)}$,

(ii) the work that must be done on machines $\alpha,\ldots,\beta-1$,

(iii) the work that must be done on machines $\alpha+1,\ldots,\beta$,

(iv) minus the work that must be done on machines $\alpha+1,\ldots,\beta-1$, as it is included two times in (ii) and (iii).

Then, for any α, β, we have

$$w_{(\alpha,\beta)} = v_{(\alpha,\beta)} + w_{(\alpha,\beta-1)} + w_{(\alpha+1,\beta)} - w_{(\alpha+1,\beta-1)},$$

with the particular case $w_{(\alpha,\beta)} = 0$ if $\alpha > \beta$. The values $v_{(\alpha,\beta)}$ can be pre-computed in time $O(n)$ by scanning jobs, and the computation of the values $w_{(\alpha,\beta)}$ is done in time $O(m^2)$. Thus $\tilde{w}_{\max}$ can be found in time $O(m^2 + n)$ and space $O(m^2)$ (see Algorithm 10).

4.3.1 An Optimal Algorithm for Unitary Jobs

We now prove the optimality of ELFJ when all jobs are unitary and $\lambda = \lceil \tilde{w}_{\max} \rceil$. The principle of the proof comes from the work of Lin et al. [**lin2004**], although we found their demonstration to be incorrect. We know from Corollary 4.2 that the optimal makespan is at least λ. Thus we seek to prove that ELFJ gives a schedule S whose makespan is at most λ.

By contradiction, Lin et al. assume there exists a unitary job j that could not be assigned by ELFJ on any machine before time λ, which means that all machines between a_j and b_j must be full. Then we consider the machine with smallest index $\alpha \leq a_j$ such that all machines between α and b_j are full. Let $\beta = b_j$. Now the goal is to prove that all jobs assigned by ELFJ on machines $\alpha, \alpha+1, \ldots, \beta$ come from the set $K_{(\alpha,\beta)}$; in other words, the processing set of each job assigned on these machines is included in $I_{(\alpha,\beta)}$. Proving this property leads to the conclusion $\lambda < \tilde{w}_{(\alpha,\beta)}$, which is a contradiction because $\lambda = \lceil \tilde{w}_{\max} \rceil$.

To do so, Lin et al. argue that any job j' assigned on a machine between α and β must have $a_{j'} \geq \alpha$, otherwise j' would have been put on $\alpha - 1$ (which is not full), and $b_{j'} \leq \beta$, because jobs have been assigned by non-decreasing order of b_j. This last justification is an error, as highlighted by the following counterexample: suppose that $\alpha = a_j - 1$, and there are λ jobs with interval $\alpha, \alpha+1, \ldots, \beta+1$ (call these jobs the *filling* jobs). The job j must be done in the interval $\alpha+1, \alpha+2, \ldots, \beta$. Then, the filling jobs will be assigned on machine α by ELFJ, even if we have $b_{j'} = \beta + 1 > \beta$ for all filling jobs j', because they are the only jobs that are feasible on α. Therefore, we cannot conclude that all jobs assigned on machines $\alpha, \alpha+1, \ldots, \beta$ come from $K_{(\alpha,\beta)}$.

We present here a constructive proof that also consists in exhibiting a contradiction by finding a machine $\alpha \leq a_j$ such that all jobs assigned between α and β come from $K_{(\alpha,\beta)}$. However, α is more carefully chosen in this new version: we start from the interval (a_j, b_j) and we extend this interval step by step until the appropriate condition is met.

Theorem 4.3. *Let $\lambda = \lceil \tilde{w}_{\max} \rceil$. Then ELFJ (Algorithm 9) is optimal and runs in time $O(m^2 + n \log n + mn)$.*

Proof: The beginning of the proof is similar to the one of Lin et al. By contradiction, suppose that ELFJ does not give a feasible schedule with makespan at most λ. Let j_0 be one of the non-assigned jobs. Then, as all jobs are unitary and λ is an integer, all machines in $\mathcal{M}_{j_0}$ must finish at least at time λ. Let $\beta = b_{j_0}$, and let $\gamma \leq a_{j_0}$ be the smallest machine index such that all machines between γ and β complete at time λ. This means that the machine $\gamma - 1$ completes before time λ if $\gamma > 1$.

Now our goal is to find a machine α between γ and a_{j_0} such that all jobs assigned on machines $\alpha, \alpha+1, \ldots, \beta$ come from the set $K_{(\alpha,\beta)}$. The process here is constructive. For the first step, let j be a job assigned on a machine between a_{j_0} and β. Then, we have $b_j \leq \beta$, otherwise j_0 would have been scheduled instead of j. Now there are two cases: either we have $a_j \geq a_{j_0}$ for all j assigned between a_{j_0} and β, or $a_j < a_{j_0}$ for at least one job j assigned between a_{j_0} and β.

If the first case holds, then we set $\alpha = a_{j_0}$ and we are done: all jobs assigned between α and β have a processing set include in $I_{(\alpha,\beta)}$. If the second case holds, let us choose such j with the smallest a_j (then

$a_j < a_{j_0}$), and let us call this job j_1. Now we proceed to the next step. If j_1 has been assigned between a_{j_0} and β, it means that $b_j \leq b_{j_1} \leq \beta$ for all jobs j assigned on machines $a_{j_1}, a_{j_1}+1, \ldots, a_{j_0}-1$, otherwise we would have scheduled j_1 instead. Moreover, we have two cases again: either we have $a_j \geq a_{j_1}$ for all j assigned between a_{j_1} and $a_{j_0}-1$, or $a_j < a_{j_1}$ for at least one job j assigned between a_{j_1} and $a_{j_0}-1$.

In the first case, we set $\alpha = a_{j_1}$ and we are done. Otherwise, we choose j with the smallest a_j, we call this job j_2 and we proceed to the next step by repeating the same reasoning.

To conclude, note that we have $a_j \geq \gamma$ for all jobs j assigned on a machine whose index is greater than or equal to γ, otherwise j would have been put on machine $\gamma - 1$, as it completes before time λ. By applying the described process iteratively, we inevitably reach a step k where there cannot exist a job j such that $a_j < a_{j_k}$, and we set $\alpha = a_{j_k}$.

Therefore, there exist $\alpha \leq \beta$ such that

(i) $j_0 \in K_{(\alpha,\beta)}$,

(ii) machines $\alpha, \alpha+1, \ldots, \beta$ complete at time λ, and

(iii) all jobs assigned on machines $\alpha, \alpha+1, \ldots, \beta$ belong to $K_{(\alpha,\beta)}$.

Then we have

$$w_{(\alpha,\beta)} \geq (\beta - \alpha + 1)\lambda + 1 > (\beta - \alpha + 1)\lambda,$$

i.e., $\lambda < \tilde{w}_{(\alpha,\beta)}$, which is a contradiction. Hence, ELFJ gives a schedule feasible in λ time units, which means that $C_{\max}^{\mathrm{OPT}} \leq \lambda$. By Corollary 4.2, we also know that $C_{\max}^{\mathrm{OPT}} \geq \lambda$. We conclude that $C_{\max}^{\mathrm{OPT}} = \lambda$, thus ELFJ is optimal. Moreover, as demonstrated earlier, the computation of λ is done in time $O(m^2+n)$, and ELFJ runs in time $O(n \log n + mn)$, which gives a total time complexity of $O(m^2 + n \log n + mn)$. ■

In this proof, we avoid the error from Lin et al. by making sure that $b_j \leq \beta$ for all jobs j assigned between either a_{j_0} and β, or between a_{j_1} and β, or between a_{j_2} and β, etc. In the previous counterexample, we would have stopped at $\alpha = a_j$ and $\beta = b_j$.

4.3.2 An Approximation for Arbitrary Processing Times

ELFJ is optimal for unitary jobs. It may well be applied to non-unitary jobs, but does not produce an optimal schedule. We prove here that it is however an approximation algorithm, with a small adaptation, for arbitrary processing times. In the following, we note $p_{\max}$ the maximum processing time among all jobs.

Theorem 4.4. *Let $\lambda = \tilde{w}_{\max} + \left(1 - \frac{1}{m}\right) p_{\max}$. Then ELFJ (Algorithm 9) is a $(2 - 1/m)$-approximation algorithm and runs in time $O(m^2 + n \log n + mn)$.*

Proof: Suppose by contradiction that ELFJ does not give a feasible schedule with makespan at most λ. Let j_0 be the first non-assigned job. Then all machines in $\mathcal{M}_{j_0}$ must finish after time $\lambda - p_{j_0}$, otherwise we would have assigned j_0. Let $\beta = b_{j_0}$, and let $\gamma \leq a_{j_0}$ be the smallest machine index such that all machines between γ and β complete after time $\lambda - p_{\max}$. This means that the machine $\gamma - 1$ completes at or before time $\lambda - p_{\max}$ if $\gamma > 1$. Hence, we have $a_j \geq \gamma$ for all jobs j assigned on a machine whose index is greater than γ, otherwise j would have been assigned on $\gamma - 1$ by ELFJ, as $p_j \leq p_{\max}$ (by definition of $p_{\max}$).

Now let $S(a, b, t)$ be the set of jobs assigned by ELFJ between machines a and b, and scheduled to start at or before time t ($S(a, b, t) = \emptyset$ if $a > b$). We can see this set $S(a, b, t)$ as a work area, whose minimal shape is the rectangle delimited by a, b and t. Our goal is to prove that there exists a machine α

between γ and a_{j_0} such that all jobs in the set $S(\alpha, a_{j_0}-1, \lambda-p_{\max}) \cup S(a_{j_0}, \beta, \lambda-p_{j_0})$, whose minimal work area can be represented by two adjacent rectangles, come from the set $K_{(\alpha,\beta)}$, which includes all jobs whose processing set is in the interval (α, β). The Figure **??** highlights the work areas of interest.

To do so, we adapt the constructive process that we used in the previous proof. Let us prove that there exists a non-empty set of machines $u_1 > u_2 > \cdots > u_x$ between a_{j_0} and γ such that

- for all u_k, $b_j \leq \beta$ for all jobs $j \in S(u_k, a_{j_0}-1, \lambda-p_{\max}) \cup S(a_{j_0}, \beta, \lambda-p_{j_0})$, and
- for all $u_{k<x}$, there exists $j \in S(u_k, a_{j_0}-1, \lambda-p_{\max}) \cup S(a_{j_0}, \beta, \lambda-p_{j_0})$ such that $\gamma \leq a_j < u_k$, and
- $a_j \geq u_x$ for all $j \in S(u_x, a_{j_0}-1, \lambda-p_{\max}) \cup S(a_{j_0}, \beta, \lambda-p_{j_0})$.

Base case ($u_1 = a_{j_0}$). Let $j \in S(a_{j_0}, \beta, \lambda-p_{j_0})$ be a job assigned between a_{j_0} and β, and starting at or before $\lambda - p_{j_0}$. We have $b_j \leq b_{j_0} = \beta$, otherwise the job j_0 could have been scheduled instead of job j. Then, either $a_j \geq a_{j_0}$ for all $j \in S(a_{j_0}, \beta, \lambda-p_{j_0})$, or there is $j \in S(a_{j_0}, \beta, \lambda-p_{j_0})$ such that $\gamma \leq a_j < a_{j_0}$. In the first case, we set $x = 1$ and we are done. In the second case, we proceed to the next step.

Induction step. Suppose that $b_j \leq \beta$ for all $j \in S(u_k, a_{j_0}-1, \lambda-p_{\max}) \cup S(a_{j_0}, \beta, \lambda-p_{j_0})$. Moreover, suppose there exists $j_1 \in S(u_k, a_{j_0}-1, \lambda-p_{\max}) \cup S(a_{j_0}, \beta, \lambda-p_{j_0})$ such that $\gamma \leq a_{j_1} < u_k$ at step k. Let us choose j_1 such that a_{j_1} is minimal, and let $u_{k+1} = a_{j_1}$.

Now let $j_2 \in S(u_{k+1}, u_k-1, \lambda-p_{\max})$ be any job assigned between machines u_{k+1} and $u_k - 1$. We have $b_{j_2} \leq b_{j_1}$, otherwise the job j_1 would have been scheduled instead of job j_2. Hence, $b_{j_2} \leq \beta$ (by induction hypothesis), thus $b_j \leq \beta$ for all $j \in S(u_{k+1}, a_{j_0}-1, \lambda-p_{\max}) \cup S(a_{j_0}, \beta, \lambda-p_{j_0})$, because $S(u_{k+1}, u_k-1, \lambda-p_{\max}) \cup S(u_k, a_{j_0}-1, \lambda-p_{\max}) = S(u_{k+1}, a_{j_0}-1, \lambda-p_{\max})$.

Then, either $a_{j_2} \geq u_{k+1}$ for all $j_2 \in S(u_{k+1}, u_k-1, \lambda-p_{\max})$, or there exists $j_2 \in S(u_{k+1}, u_k - 1, \lambda-p_{\max})$ such that $\gamma \leq a_{j_2} < u_{k+1}$. In the first case, we conclude that $a_j \geq u_{k+1}$ for all $j \in S(u_{k+1}, a_{j_0}-1, \lambda-p_{\max}) \cup S(a_{j_0}, \beta, \lambda-p_{j_0})$, because we have chosen j_1 in a way that a_{j_1} is minimal (thus $a_j \geq a_{j_1} = u_{k+1}$ for all $j \in S(u_k, a_{j_0}-1, \lambda-p_{\max}) \cup S(a_{j_0}, \beta, \lambda-p_{j_0})$). Hence, we set $x = k+1$ and we stop there. In the second case, we proceed to the next step.

Therefore, we proved that we can find a machine u_x such that $a_j \geq u_x$ and $b_j \leq \beta$ for all $j \in S(u_x, a_{j_0}-1, \lambda-p_{\max}) \cup S(a_{j_0}, \beta, \lambda-p_{j_0})$. In other words, all jobs in the set $S(u_x, a_{j_0}-1, \lambda-p_{\max}) \cup S(a_{j_0}, \beta, \lambda-p_{j_0})$ come from the set $K_{(u_x,\beta)}$.

Recall that all machines $a_{j_0}, a_{j_0}+1, \ldots, \beta$ finish after time $\lambda-p_{j_0}$, and by construction, all machines $u_x, u_x+1, \ldots, a_{j_0}-1$ finish after time $\lambda - p_{\max}$, because $u_x \geq \gamma$. Thus we set $\alpha = u_x$, and we have

$$w_{(\alpha,\beta)} > (\beta-\alpha+1)(\lambda-p_{\max}) + (\beta-a_{j_0}+1)(\lambda-p_{j_0}-(\lambda-p_{\max})) + p_{j_0},$$

which gives the following inequality:

$$\lambda < \tilde{w}_{(\alpha,\beta)} - \frac{p_{j_0}}{\beta-\alpha+1} - \frac{(\beta-a_{j_0}+1)(p_{\max}-p_{j_0})}{\beta-\alpha+1} + p_{\max}.$$

As $\beta - a_{j_0} + 1 \geq 1$ and $\beta - \alpha + 1 \leq m$, we have $\frac{\beta-a_{j_0}+1}{\beta-\alpha+1} \geq \frac{\beta-a_{j_0}+1}{m} \geq \frac{1}{m}$. Moreover, $p_{\max} - p_{j_0} \geq 0$, then

$$\lambda < \tilde{w}_{(\alpha,\beta)} - \frac{p_{j_0}}{\beta-\alpha+1} - \frac{1}{m}(p_{\max}-p_{j_0}) + p_{\max},$$

thus,

$$\lambda < \tilde{w}_{(\alpha,\beta)} + \left(1-\frac{1}{m}\right)p_{\max} + \left(\frac{1}{m} - \frac{1}{\beta-\alpha+1}\right)p_{j_0}.$$

Finally, we have $\frac{1}{\beta-\alpha+1} \geq \frac{1}{m}$, i.e., $\frac{1}{m} - \frac{1}{\beta-\alpha+1} \leq 0$, and $p_{j_0} \geq 0$. Therefore,

$$\lambda < \tilde{w}_{(\alpha,\beta)} + \left(1 - \frac{1}{m}\right) p_{\max},$$

which is a contradiction.

Hence, ELFJ gives a schedule that is feasible in time λ, i.e., $C_{\max} \leq \lambda$. By Lemma 4.1, we have $C^{\mathrm{OPT}}_{\max} \geq \tilde{w}_{\max}$, and obviously, $C^{\mathrm{OPT}}_{\max} \geq p_{\max}$, so $\lambda \leq (2 - 1/m) \cdot C^{\mathrm{OPT}}_{\max}$. We conclude that $C_{\max} \leq (2 - 1/m) \cdot C^{\mathrm{OPT}}_{\max}$.

Note that this approximation ratio is tight. One may easily consider an instance with one job of size 2 and $2(m-1)$ unitary jobs (all feasible on all machines), i.e., $\lambda = 2(2-1/m)$. ELFJ will keep scheduling unitary jobs on the first machine until it reaches makespan $2(2 - 1/m)$, whereas the optimal makespan is 2. ■

4.3.3 An Optimal Algorithm for 2-Stacked Intervals

4.4 A General Framework for Circular Intervals

4.4.1 Introducing Circular Intervals

In the standard RAI problem, machines are linearly arranged, that is to say, they are numbered from 1 to m and virtually placed on a line. As we have seen in the introduction, databases often organize machines in a virtual ring, where the machines able to answer a query for a particular key are arranged as a consecutive subset of this ring. We generalize here the notion of interval to take into account this setting by introducing the notion of *circular* intervals. Machines are now virtually arranged on a circle. In addition to regular intervals (a, b) (with $a \leq b$), we introduce *circular* intervals such that $a > b$. In this case, the corresponding set $I_{(a,b)}$ includes machines $a, a+1, \ldots, m$ and machines $1, 2, \ldots, b$, i.e., we have

$$I_{(a,b)} = \begin{cases} \{a, a+1, \ldots, b\} & \text{if } a \leq b, \\ \{1, 2, \ldots, b\} \cup \{a, a+1, \ldots, m\} & \text{otherwise.} \end{cases}$$

Note that we clearly cannot always rearrange machines to transform an instance with circular intervals to an instance without circular intervals. Consider the instance with 3 machines and 3 jobs with processing sets $\mathcal{M}_1 = \{1, 2\}$, $\mathcal{M}_2 = \{2, 3\}$ and $\mathcal{M}_3 = \{3, 1\}$: any permutation of the machines will exhibit exactly one circular interval. Figure **??** illustrates the generalization of the RAI problem to circular intervals. By extension, we call this generalized problem the Restricted Assignment problem on Circular Intervals (RACI).

Definition 4.1. *The interval* (a_g, b_g) precedes *the interval* (a_h, b_h) *if and only if* $a_g \leq a_h$ *and* $b_g \leq b_h$. *In this case, we note* $(a_g, b_g) \preceq (a_h, b_h)$.

For a given instance, let Z^* be the set of circular intervals that are associated to at least one job ($Z^* = \{(a_j, b_j)$ s.t. $j \in J$ and $a_j > b_j\}$). In this section, we restrict ourselves to instances where the previously-defined relation $\preceq$ is a total order on Z^*. In other words, for any $g, h \in Z^*$, we cannot have $I_g \subset I_h$ or $I_h \subset I_g$. This constitutes a particular case of RACI, but it is still a more general case than RAI. Moreover, we assume that there are K types of jobs, and each job of type k has processing time $p(k)$. We introduce in this section a general procedure that solves the RACI problem for these restricted instances, assuming that one already knows an optimal algorithm $\mathcal{A}$ for the standard RAI problem with K job types.

Theorem 4.5. *Let $\mathcal{A}$ be an optimal algorithm for the RAI problem with K job types that runs in time $O(f(n))$, where f is a polynomial. Then there exists a procedure that solves the corresponding RACI problem on ordered circular intervals in time $O(n^K f(n))$.*

We begin with a few definitions. Then we present the procedure, before proving our result.

Preliminaries. Let J^* be the subset of jobs whose processing set is a circular interval, i.e., $J^* = \{j \in J \text{ s.t. } a_j > b_j\}$, and we note $n^* = |J^*|$. We call J^* the *circular* jobs. We also partition J^* into K subsets $J_1^*, \ldots, J_K^*$, such that all jobs in J_k^* are of type k, and we note $n_k^* = |J_k^*|$.

Moreover, in a given schedule, we say that a circular job j assigned between a_j (inclusive) and m (inclusive) is a *left* job. Equivalently, a circular job j assigned between 1 (inclusive) and b_j (inclusive) is a *right* job. This means that a schedule S implicitely defines a partition of each set J_k^* into two subsets G_k and D_k, where G_k contains γ_k left jobs, and D_k contains δ_k right jobs. Figure **??** shows an example of such schedule.

We present here the intuition on how to compute an optimal schedule by considering all possible partitions of jobs with circular processing sets into left and right jobs. For the moment, we simplify the problem by considering only one type of jobs. A schedule defines a partition of jobs J^* into left and right jobs, which means that we need to find *how many* jobs in J^* should be assigned to the left or to the right. Thus, assume that we know that r jobs of J^* must be assigned to the right in an optimal schedule. Intuitively, the r circular jobs with rightmost intervals should be put on the right, and the remaining jobs of J^* should be put on the left. For example, consider only the small jobs in the instance of Figure **??**. If we suppose that $r = 5$ (arbitrarily), then we guess that the 2 red jobs and the 3 green jobs should be put on the right (i.e., between machines 1 and 3), and the 2 blue jobs should be put on the left (i.e., between machines $m-2$ and m), as the red and green intervals are more on the right than the blue interval. We introduce below the notion of *right-sorted* schedules that captures this intuition, and we will prove later that there always exists optimal schedules that have this property.

Definition 4.2. *A schedule S is* right-sorted *if and only if for each type k, the property $(a_j, b_j) \preceq (a_{j'}, b_{j'})$ holds for any jobs $j \in G_k$ and $j' \in D_k$.*

We denote the set of all possible schedules for a given instance $\mathcal{I}$ by $\mathcal{S}(\mathcal{I})$ ($\mathcal{I}$ is omitted when it is clear from the context). Let $\mathbf{R}^K$ be the set of all vectors $\mathbf{r} = (r_1, \cdots, r_K)$ such that r_k is an integer and $0 \leq r_k \leq n_k^*$ for all k. For a given vector $\mathbf{r} \in \mathbf{R}^K$, we call $\mathcal{S}_\mathbf{r}$ the subset of schedules $\mathcal{S}$ that put exactly r_k jobs of type k on the right, and $n_k^* - r_k$ jobs of type k on the left. Recall that $C_{\max}^{\text{OPT}}$ denotes the optimal makespan among all schedules $\mathcal{S}$. We define analogously $C_\mathbf{r}^{\text{BEST}}$ as the *best possible makespan* among schedules $\mathcal{S}_\mathbf{r}$. Note that the subsets $\mathcal{S}_\mathbf{r}$ define a partition of $\mathcal{S}$, and thus

$$C_{\max}^{\text{OPT}} = \min_{\mathbf{r} \in \mathbf{R}^K} \left\{ C_\mathbf{r}^{\text{BEST}} \right\}. \tag{4.1}$$

Optimal procedure. We introduce a polynomial procedure $\phi_\mathbf{r}$ that transforms any instance $\mathcal{I}$ of the (ordered) RACI problem into another instance $\mathcal{I}' = \phi_\mathbf{r}(\mathcal{I})$ that does not include any circular interval. We prove later the following two properties on the obtained instance:

(i) Applying an optimal algorithm $\mathcal{A}$ to $\mathcal{I}'$ produces a valid solution for $\mathcal{I}$.

(ii) The makespan of this solution is at most $C_\mathbf{r}^{\text{BEST}}$.

Given these properties and Equation (4.1), we can find an optimal solution for $\mathcal{I}$ by performing an exhaustive search of the best vector $\mathbf{r} \in \mathbf{R}^K$. For a given instance $\mathcal{I}$, the function $\phi_\mathbf{r}$ works as follows:

1. Sort jobs J^* by non-increasing order of b_j, and sort jobs with identical b_j by non-increasing order of a_j. Note that this corresponds to sorting jobs by non-increasing order of $\preceq$. As $\preceq$ is a total order on Z^*, all jobs are comparable.

2. For each type k, set $a_j = 1$ for the r_k first jobs of J^*_k, and $b_j = m$ for the $n^*_k - r_k$ other jobs.

Let $\mathcal{S}^{\preceq}_{\mathbf{r}}$ be the subset of schedules $\mathcal{S}_{\mathbf{r}}$ that are right-sorted. The proof of the two properties of interest is structured as follows. As $\mathcal{A}$ is optimal for the standard RAI problem, we know that it finds one of the best schedules among $\mathcal{S}(\phi_{\mathbf{r}}(\mathcal{I}))$. Hence, we will prove two lemmas. On the one hand, we show in Lemma 4.6 that the set $\mathcal{S}(\phi_{\mathbf{r}}(\mathcal{I}))$ is exactly the same as the set of right-sorted schedules $\mathcal{S}^{\preceq}_{\mathbf{r}}(\mathcal{I})$ for the initial instance. On the other hand, we show in Lemma 4.7 that there always exists a right-sorted schedule that has the best possible makespan.

Lemma 4.6. *For any $\mathbf{r} \in \mathbf{R}^K$, we have $\mathcal{S}(\phi_{\mathbf{r}}(\mathcal{I})) = \mathcal{S}^{\preceq}_{\mathbf{r}}(\mathcal{I})$.*

Proof: Let $\mathbf{r}$ be an arbitrary vector of $\mathbf{R}^K$. First we show that $\mathcal{S}(\phi_{\mathbf{r}}(\mathcal{I})) \subseteq \mathcal{S}^{\preceq}_{\mathbf{r}}(\mathcal{I})$. Let $S \in \mathcal{S}(\phi_{\mathbf{r}}(\mathcal{I}))$. By definition of $\phi_{\mathbf{r}}$, for all types k, there are $n^*_k - r_k$ jobs in S that were circular jobs in the initial instance $\mathcal{I}$ and that are on the left (similarly, there are r_k jobs in S that were circular and that are on the right). Moreover, the circular jobs have been sorted in $\phi_{\mathbf{r}}$, which means that for all k, we have $(a_j, b_j) \preceq (a_{j'}, b_{j'})$ for any $j \in G_k$ and $j' \in D_k$ in S. In other words, S is right-sorted, and thus belongs to $\mathcal{S}^{\preceq}_{\mathbf{r}}$.

Now we show that $\mathcal{S}^{\preceq}_{\mathbf{r}}(\mathcal{I}) \subseteq \mathcal{S}(\phi_{\mathbf{r}}(\mathcal{I}))$. By definition of $\mathcal{S}^{\preceq}_{\mathbf{r}}$, in any schedule $S \in \mathcal{S}^{\preceq}_{\mathbf{r}}(\mathcal{I})$, for all types k, we have $(a_j, b_j) \preceq (a_{j'}, b_{j'})$ for any $j \in G_k$ and $j' \in D_k$. Moreover, there are exactly $n^*_k - r_k$ jobs in G_k and r_k jobs in D_k. Thus, S is clearly a valid solution for $\phi_{\mathbf{r}}(\mathcal{I})$ and belongs to $\mathcal{S}(\phi_{\mathbf{r}}(\mathcal{I}))$. ■

Lemma 4.7. *For any $\mathbf{r} \in \mathbf{R}^K$, there exists a right-sorted schedule $S \in \mathcal{S}^{\preceq}_{\mathbf{r}}$ that has the best possible makespan $C^{\text{BEST}}_{\mathbf{r}}$.*

Proof: Let $\mathbf{r}$ be an arbitrary vector of $\mathbf{R}^K$. Let $S \in \mathcal{S}_{\mathbf{r}}$ be a schedule that has the best possible makespan $C^{\text{BEST}}_{\mathbf{r}}$. If S is right-sorted, we are done. Otherwise, there necessarily exists a type k such that two jobs $j \in G_k$ and $j' \in D_k$, scheduled in S, are not sorted according to $\preceq$, i.e., we have $(a_j, b_j) \not\preceq (a_{j'}, b_{j'})$. In other words, either $a_j > a_{j'}$, or $b_j > b_{j'}$. We know that $\preceq$ is a total order on Z^*, which means that if $a_j > a_{j'}$, then we necessarily have $b_j \geq b_{j'}$. In a similar way, if $b_j > b_{j'}$, then we necessarily have $a_j \geq a_{j'}$. This means that even if j is a left job and j' is a right job in S, there is "more room" to put j on the right side and j' on the left side of their respective interval. Moreover, as j and j' have the same type k, they have identical processing times. Hence, we can clearly swap j and j' in S without changing the makespan of S.

By repeatedly swapping non-sorted jobs of the same type, we reach another schedule S' that has the same makespan than S, that also belongs to $\mathcal{S}_{\mathbf{r}}$, and that is right-sorted. ■

Now we are able to conclude. By hypothesis, we know that $\mathcal{A}$ finds a schedule with the smallest makespan among $\mathcal{S}(\phi_{\mathbf{r}}(\mathcal{I}))$. By Lemma 4.7, we also know that there exists at least one schedule in $\mathcal{S}^{\preceq}_{\mathbf{r}}(\mathcal{I})$ that has the best possible makespan $C^{\text{BEST}}_{\mathbf{r}}$. Therefore, we deduce by Lemma 4.6 that:

- the solution given by $\mathcal{A}$, when applied to $\phi_{\mathbf{r}}(\mathcal{I})$, belongs to $\mathcal{S}^{\preceq}_{\mathbf{r}}(\mathcal{I})$, which means that it also belongs to $\mathcal{S}_{\mathbf{r}}(\mathcal{I})$, i.e., it is a valid solution for $\mathcal{I}$ (Property (i)), and
- the solution given by $\mathcal{A}$, when applied to $\phi_{\mathbf{r}}(\mathcal{I})$, has makespan $C^{\text{BEST}}_{\mathbf{r}}$ (Property (ii)).

It follows that, for any instance $\mathcal{I}$, we can find the best possible schedule among $\mathcal{S}_{\mathbf{r}}(\mathcal{I})$ for any vector $\mathbf{r} \in \mathbf{R}^K$ in polynomial time, as the transformation procedure $\phi_{\mathbf{r}}$ and the algorithm $\mathcal{A}$ are themselves polynomial.

Moreover, for all vectors $\mathbf{r} \in \mathbf{R}^K$, we have $r_k \leq n_k^* \leq n$ for all k. Thus, the number of possible vectors $\mathbf{r}$ is bounded by $O(n^K)$, i.e., we can find an optimal schedule for any instance $\mathcal{I}$ by searching over all possible vectors in time $O(n^K f(n))$, assuming that we know a polynomial algorithm $\mathcal{A}$ that runs in time $O(f(n))$ when applied to $\phi_{\mathbf{r}}(\mathcal{I})$. This concludes the proof of Theorem 4.5.

We now study two special cases where this procedure can be applied: the adaptation of an existing dynamic programming algorithm for K job types and the ELFJ algorithm presented above. In the latter case, we are able to largely reduce the complexity compared to Theorem 4.5, as we achieve for ELFJ on circular intervals the same complexity as ELFJ on regular intervals.

4.4.2 An Optimal Procedure for K Job Types

4.4.3 A Dynamic Program for K Job Types

We illustrate how our framework can be successfully applied to derive a polynomial algorithm for the RACI problem on intervals of equal length and K job types. Wang et al. [**wang2016**] showed how to solve the corresponding problem on non-circular intervals with a dynamic program. For completeness, we recall their solution in the following.

Let n_k be the number of jobs of type k ($1 \leq k \leq K$), and let us sort jobs by non-decreasing value of b_j. Suppose that λ is a value that represents a hard deadline for all jobs. Define $F_i(s_1, s_2, \ldots, s_K) = 1$ if and only if it is feasible, for all types k, to schedule s_k jobs of type k on machines $1, 2, \ldots, i$ such that the makespan is at most λ, and $F_i(s_1, s_2, \ldots, s_K) = 0$ otherwise.

Let $F_0(0, \ldots, 0) = 1$, and $F_i(s_1, s_2, \ldots, s_K) = 1$ if and only if there exist $s'_1 \leq s_1, s'_2 \leq s_2, \ldots, s'_K \leq s_K$ such that:

(i) $F_{i-1}(s'_1, s'_2, \ldots, s'_K) = 1$,

(ii) for each k, the next $s_k - s'_k$ jobs of type k in the sorted set contain the machine i in their processing set, and

(iii) $\sum_{k=1}^{K} (s_k - s'_k) \cdot p(k) \leq \lambda$.

Then we have $F_m(n_1, n_2, \ldots, n_K) = 1$ if and only if there exists a schedule feasible in time λ. For a given value of λ, an array with all values of F can be computed in time $O(mn^{2K})$. Finally, the optimal value of λ can be found by performing a binary search. Thus the overall complexity of the algorithm is $O(mn^{2K} \log \sum p_j)$.

By using our framework, adapting this approach to the RACI problem is straightforward. Let $\mathcal{A}$ be the dynamic program described above. By Theorem 4.5, we know that we can find an optimal schedule for any instance in time $O(n^K f(n))$, where $f(n)$ is the complexity of $\mathcal{A}$. Therefore, the time complexity of the derived algorithm is $O(mn^{3K} \log \sum p_j)$.

4.4.4 Revisiting the Unitary Job Case

We proved in Theorem 4.3 that ELFJ is an optimal algorithm for the standard RAI problem on unitary jobs, which runs in time $f(n) = O(m^2 + n \log n + mn)$. Recall that ELFJ consists in 3 distinct steps:

1. computing the optimal makespan value, in time $O(m^2 + n)$,
2. sorting the jobs, in time $O(n \log n)$,
3. performing the actual job assignment, in time $O(mn)$.

By applying our framework around ELFJ, and because we have only one type of jobs in this specific case, we know from Theorem 4.5 that we can solve the generalized problem on ordered circular intervals in time $O(nf(n)) = O(m^2n + n^2 \log n + mn^2)$. We now show how to improve the complexity of this solution, as stated in the following theorem.

Theorem 4.8. *The ordered RACI problem with unitary jobs can be solved in time* $O(m^2+n\log n+mn)$.

Proof: The basic idea is to extract and reorganize some internal computation steps from the exhaustive search procedure to avoid doing any redundant work. We first observe that the only step of ELFJ that actually depends on knowing an optimal number r of right jobs (in the set of circular jobs) is the computation of the optimal makespan. Once we know the best values of r and λ, the sorting and job assignment steps are straightforward. Thus we know that we can easily refine the complexity to $O(n(m^2 + n) + n \log n + mn) = O(m^2n + n^2)$.

To reduce further the complexity, we notice that we do not really need to recompute the matrix w from the beginning (in time $O(m^2 + n)$) for each possible value of r in order to find the minimum makespan. We remark that there are two kinds of non-circular intervals: the ones that may result from cutting a circular interval (a, b) in two subintervals (a, m) and $(1, b)$, which we call the *outside* intervals, and the ones that cannot, which we call *inside* intervals. In other words, outside intervals are all non-circular intervals of the form $(1, x)$ or (x, m), with $1 \leq x \leq m$, and inside intervals are all non-circular intervals of the form (x, y) with $1 < x \leq y < m$. When representing the non-circular interval hierarchy as a lattice graph, the outside intervals are in fact all the nodes on the sides of the lattice, and the inside intervals are the others, as shown in Figure **??**.

Recall that $w_{(\alpha,\beta)}$ represents the total work of all non-circular jobs whose interval is included in (α, β). When we update our guess on the optimal number of right jobs, we transform the instance by shrinking the intervals of circular jobs: if a job j is a right job, we keep the right part of the interval, i.e., the subinterval $(1, b_j)$, and if it is a left job, we keep the left part of the interval, i.e., the subinterval (a_j, m). This means that the only values of w that may change when we transform the instance are the ones that are associated to the outside intervals. All other values remain unchanged, no matter how we partition the circular jobs.

Hence, we can decompose the computation of $\tilde{w}_{\max}$ in two steps. First, compute the value

$$\tilde{w}_{\max}^{inside} = \max_{1<\alpha\leq\beta<m} \left\{\tilde{w}_{(\alpha,\beta)}\right\},$$

which represents the maximum value of $\tilde{w}$ among all inside intervals. This value does not depend on r, and can be computed only once. Second, compute the value

$$\tilde{w}_{\max}^{outside} = \max_{1\leq x\leq m} \left\{\max\left(\tilde{w}_{(1,x)}, \tilde{w}_{(x,m)}\right)\right\},$$

which represents the maximum value of $\tilde{w}$ among all outside intervals. We clearly have

$$\tilde{w}_{\max} = \max\left(\tilde{w}_{\max}^{inside}, \tilde{w}_{\max}^{outside}\right).$$

In other words, each time we update our guess on r, we only need to recompute the value of $\tilde{w}_{\max}^{outside}$, which can be done in time $O(m)$ as there are exactly $2m - 1$ outside intervals. Thus, we can search for the minimum value of $\max\left(\tilde{w}_{\max}^{inside}, \tilde{w}_{\max}^{outside}\right)$ by pre-computing $\tilde{w}_{\max}^{inside}$, and then trying each possible value of r by updating only $\tilde{w}_{\max}^{outside}$.

The only remaining question is how we know which values to update in the matrix w when we make a new guess on r. We avoid recomputing the values that are associated to inside intervals. We can also

avoid recomputing the value $w_{(1,m)}$, as it is always exactly equal to n, and we have $\tilde{w}_{(1,m)} = n/m$. Recall that the set of circular jobs is sorted by decreasing order of $\preceq$ (by definition of $\phi_{\mathbf{r}}$), and for a given number r, we know that the first r jobs in the sorted set are right jobs. We set $r = 0$ and we pre-compute w, $\tilde{w}_{\max}^{inside}$ and $\tilde{w}_{\max}$. Then we loop over the sorted set of circular jobs by adding them progressively on the right side, i.e., for each job $j \in J^*$, we add 1 to $w_{(1,\beta)}$ for all $b_j \leq \beta < m$ and we subtract 1 to $w_{(\alpha,m)}$ for all $1 < \alpha \leq a_j$. The full procedure is given in Algorithm 11.

We conclude that the RACI problem with ordered circular intervals and unitary jobs can be solved in time $O(m^2 + n \log n + mn)$, or $O(n \log n)$ if we assume that m is fixed. ■

4.5 Conclusion

In this paper, we improved prior work done on the Restricted Assignment problem on Intervals by giving a generalized version of an existing algorithm. Our version solves the problem with unitary jobs to optimality in polynomial time $O(m^2 + n \log n + mn)$, and we proved that it also provides a $(2 - 1/m)$-approximation in the general case. Moreover, we extended the RAI problem to *circular* intervals that corresponds to common use cases in distributed databases and is thus of particular practical interest. We proposed a general framework that, given an optimal algorithm for the RAI problem with at most K job types and running in time $O(f(n))$, computes an optimal solution for the RAI problem with circular intervals (RACI) in time $n^K f(n)$. This enabled us to revisit the initial algorithm for the RAI problem with unitary jobs to derive an optimal algorithm for the RACI problem with unitary jobs, also running in time $O(m^2 + n \log n + mn)$.

We identified several challenges for further explorations. First, it is unknown if there exists a reasonably efficient algorithm for the RAI problem with two types of jobs, which would be very suitable for multi-get request partitioning in key-value store systems, as workloads often exhibit long-tailed distributions, with lots of small jobs and some that are a lot bigger. Second, it would be interesting to know if there is an efficient (i.e., usable in practice) approximation algorithm that improves on the $2 - 1/m$ guaranteed factor. Finally, we identified that instances of the RACI problem where some circular intervals are strictly included into other circular intervals are a lot more challenging to solve than the ordered case. The unitary jobs variant is polynomial, but it is still unknown if this is the case when considering at least two job types.

Algorithm 10 Computing $\tilde{w}_{\max}$ in time $O(m^2 + n)$

$\tilde{w}_{\max} \leftarrow 0$
for each $0 \leq \alpha \leq \beta \leq m$ **do**
    $v_{(\alpha,\beta)} \leftarrow 0$
for each job j **do**
    $v_{(a_j,b_j)} \leftarrow v_{(a_j,b_j)} + p_j$
for all l from 0 to $m-1$ **do**
    for all a from 1 to $m-l$ **do**
        $b \leftarrow a + l$
        $w_{(a,b)} \leftarrow v_{(a,b)} + w_{(a,b-1)} + w_{(a+1,b)} - w_{(a+1,b-1)}$
        $\tilde{w}_{(a,b)} \leftarrow \frac{w_{(a,b)}}{b-a+1}$
        if $\tilde{w}_{(a,b)} > \tilde{w}_{\max}$ **then**
            $\tilde{w}_{\max} \leftarrow \tilde{w}_{(a,b)}$

Algorithm 11 Computing r and $\tilde{w}_{\max}$ in time $O(m^2 + n \log n + mn)$

sort circular jobs by decreasing order of $\preceq$
transform circular jobs as left jobs and pre-compute w, $\tilde{w}_{\max}^{inside}$ and $\tilde{w}_{\max}$
$r_{cur} \leftarrow 0$
for each circular job $j \in J^*$ **do**
    $\tilde{w}_{\max}^{outside} \leftarrow \tilde{w}_{(1,m)}$
    for each β from b_j to $m-1$ **do** ▷ Add j on the right
        $w_{(1,\beta)} \leftarrow w_{(1,\beta)} + 1$
        $\tilde{w}_{(1,\beta)} \leftarrow \frac{w_{(1,\beta)}}{\beta}$
        if $\tilde{w}_{(1,\beta)} > \tilde{w}_{\max}^{outside}$ **then**
            $\tilde{w}_{\max}^{outside} \leftarrow \tilde{w}_{(1,\beta)}$
    for each α from 2 to a_j **do** ▷ Remove j from the left
        $w_{(\alpha,m)} \leftarrow w_{(\alpha,m)} - 1$
        $\tilde{w}_{(\alpha,m)} \leftarrow \frac{w_{(\alpha,m)}}{m-\alpha+1}$
        if $\tilde{w}_{(\alpha,m)} > \tilde{w}_{\max}^{outside}$ **then**
            $\tilde{w}_{\max}^{outside} \leftarrow \tilde{w}_{(\alpha,m)}$
    $r_{cur} \leftarrow r_{cur} + 1$
    $\tilde{w}_{cur} \leftarrow \max\left(\tilde{w}_{\max}^{inside}, \tilde{w}_{\max}^{outside}\right)$
    if $\tilde{w}_{cur} < \tilde{w}_{\max}$ **then** ▷ Update minimum makespan
        $r \leftarrow r_{cur}$
        $\tilde{w}_{\max} \leftarrow \tilde{w}_{cur}$

5

Implementing and Evaluating Scheduling Strategies

5.1 Introduction

5.2 Scheduling in Persistent Key-Value Stores

5.2.1 Overview

A key-value store is a simple NoSQL database where each data item is bound to a unique key. The simple API (i.e., without secondary indexes) and lack of integrated support for complex queries (e.g., no joins) allow implementations of key-value stores that scale remarkably well horizontally, i.e., they are easily able to include more nodes if the workload requires it. In distributed key-value stores, keys are dispatched on servers using a partitioning scheme based on consistent hashing. As the same hash function is used across the cluster, each server can know where a data item associated with a given key is stored by simply hashing its key. In addition, data items are replicated according to a replication factor to preserve availability in the presence of faults. The typical replication factor in large-scale key-value stores is 3, which means that each data item is stored on 3 different servers.

In *persistent* key-value stores, in contrast with *in-memory* key-value stores, data is eventually saved on disk. This adds a level of complexity: as random I/O operations are slow, key-value stores typically employ special data structures, called Log-Structured Merge (LSM) trees, that temporarily perform write operations in memory and periodically flush data to disk [**o1996log**]. This allows bulk write operations that reduce wear on disks, in particular SSDs, and significantly improves write throughput. Read operations, on the other hand, may either result in a cache hit (when the key is still represented in memory) or

a cache miss, in which case costly disk reads are necessary. In this paper, we focus on one such persistent and distributed key-value store, namely Apache Cassandra, where each server plays two roles:

A. It receives client requests. For a given client request that is received by a server, we say that this server is the *coordinator* for this request.

B. It executes requests. When a coordinator receives a client request, it computes the set of servers able to execute the request. These servers are called *replicas*. The coordinator (i) decides which subset of these replicas will execute the request, (ii) forwards the request, (iii) awaits the response, and (iv) replies to the client.

The number of chosen replicas may vary according to the nature of the request and its *consistency level*, which corresponds to the number of replicas that must acknowledge this request before it is considered successful. A write operation is always processed on all replicas even if its corresponding consistency level is lower than the replication factor (however, we do not necessarily wait for the response of all replicas). This makes scheduling in general less imperative for write requests, as they must be executed by all replicas anyway, and each write operation is very fast thanks to the LSM tree structure that absorbs most I/O bottlenecks. A read operation, on the other hand, is executed on the exact number of replicas that corresponds to its consistency level and is considered unsuccessful if the responses are inconsistent. In this work, we set the consistency level to 1, which is arguably the most common setting for read-dominated workloads. Moreover, we focus on a single-datacenter cluster of static size: all servers are geographically located in the same place, they are linked by a high-performance local network (in the same rack), and there is no dynamic reconfiguration of scale-out/scale-in operations.

Within each server in Apache Cassandra, the parallel execution of request coordination and execution using multiple cores relies on a Staged Event-Driven Architecture (SEDA) [**welsh2001**]. Each type of operation is processed by a different pool of worker threads (also called a *stage*). For instance, the reception and handling of client requests (as part of the coordinator role) are processed by the Native-Transport-Requests thread pool, whereas read request reception and execution (replica role) are processed by the ReadStage thread pool. Additional stages are dedicated to the execution of write requests, internode messaging protocol, data migration, tracing, etc. Scheduling requests in Apache Cassandra is a two-step operation. First, the coordinator must choose which replica the request should be sent to (then, the request is sent to this replica that inserts it into its local operation queue). This step is called *replica selection*. Second, the chosen replica must decide in which order its pending read operations should be processed. This step is called *local scheduling*. Note that, for a given request, the coordinator and the replica may sometimes be the same server. Figure **??** presents a more thorough breakdown of the steps involved in the scheduling of a read request:

1. A client request reaches a server, which becomes the coordinator for this request.
2. A worker thread from the Native-Transport-Requests thread pool picks the request and infers the set of replicas that are able to execute it.
3. **The coordinator chooses a replica according to a replica selection strategy.**
4. The coordinator forwards the request to the replica.
5. The request reaches the replica, which pushes it into its local queue dedicated to read operations.
6. **Worker threads from the** ReadStage **thread pool process the local queue in a specific order according to a local scheduling strategy.**

7. The request is executed by a worker thread, which reads data (in memory, if present in the cache, or on disk).

8. The replica sends data to the coordinator.

9. The coordinator responds to the client.

Steps 1-4 and 9 happen on the coordinator, whereas steps 5-8 happen on the replica itself. We highlight in boldface the key scheduling operations: replica selection at step 3 and local scheduling at step 6.

5.2.2 Challenges

Various scheduling strategies have been proposed in the literature to improve the overall performance of Apache Cassandra [**jiang2022ams**, 39, 20, 36, 19]. By studying and comparing these papers, we observe that designing and implementing new solutions poses several challenges.

First, scheduling strategies often need information (e.g., the current sizes of the request queues or the current service rates at all replicas) on the cluster state in order to compute a score for each replica. The more accurate this information is, the more representative the score will be of the server health, and therefore the better the scheduling decisions will be. Unfortunately, key-value stores are subject to the usual constraints of distributed systems, namely that each server cannot know the exact state of other machines, as measurements must take place at a bounded pace and information takes time to propagate over the network. This means that algorithmic decisions (such as scheduling of read requests) can only be made with partial and out-of-date knowledge of the cluster condition. Efficiently leveraging such stale data is challenging. For example, the default replica selection algorithm of Apache Cassandra frequently causes *herd behaviors*, i.e., situations where all coordinators periodically select the same, supposedly most-suited server, leading to load oscillations [39].

Another difficulty is that the workload, i.e., the flow of requests reaching the key-value store system at runtime, is generally unknown beforehand. This is a problem, as various workload characteristics have direct implications on the behavior of a scheduling strategy. For example, one such characteristic is the distribution of data item sizes. Existing workloads may be homogeneous, where data items are all of a similar size, or heterogeneous, e.g., with sizes exhibiting a power law or a bimodal distribution. Another example is the distribution of key popularities: this is the statistical distribution of key access frequencies in client requests. Many more characteristics could be extracted from real traces, such as temporal patterns, the correlation between size and popularity, and reuse periods between keys, among others [2].

We also identify a reproducibility concern, coming from the lack of a common baseline on which strategies may be properly compared [**bajpai2017challenges**]. Existing proposals typically implement new algorithms in different versions of Apache Cassandra. When code artifacts are publicly available (which is not systematic), this requires transferring the implementation of a prior solution in the more recent codebase, which is a cumbersome task and implies potential incompatibilities with newer components. When code artifacts are not publicly available, this requires building state-of-the-art strategies from their general description, which is far from being ideal and prone to errors, as implementation details are often overlooked in scientific papers. Moreover, the software configuration is usually not indicated, and the hardware configuration is rarely similar. This makes directly comparing published performance figures particularly unreliable.

Finally, diving into the Apache Cassandra codebase may be intimidating (it contains almost 500 000 lines of Java code), and scheduling-related code is dispatched across many different packages. Testing new solutions is, as a result, a time-consuming task for newcomers, especially when they want to

determine if a particular idea is efficient and worth paying the associated communication, storage, or complexity cost.

5.3 Introducing Hector

5.3.1 Overview

We unify the scheduling-related components of Apache Cassandra (version 4.2) into a coherent framework called Hector[1]. The API of this framework is simple enough to let any user implement new scheduling strategies without knowing the details of the entire codebase. Moreover, we introduce features that are not present by default in Apache Cassandra and that help designing more powerful and sophisticated policies. Our approach enhances the workflow of comparing strategies under specific assumptions by providing a common baseline. As each component is configurable from a single control point, this also enables users to more quickly identify the best setting for their use-case. We present in this section the general components of Hector and we give some examples of implemented scheduling strategies.

5.3.2 Modular Components

The API of Hector is a set of general interfaces. The user implements these interfaces and specifies with which parameters they must be loaded using a configuration file. Hector takes the responsibility of instantiating the components in the correct order and connecting them together when starting the system. We describe the different components in what follows.

Replica selection. This is the step in which the coordinator chooses the set of replicas that are considered to be most suited for the current request. In Apache Cassandra, this module has access to the full list of replicas and the mapping between keys and replicas. When presented with a target key, the module must return a list of servers in decreasing order of priority. The n first servers from this list are contacted, where n is the consistency level. When reading from a single replica (i.e., the consistency level is 1), the module typically returns a list of replicas and only the first one is contacted. In a nutshell, the ordering step constitutes the essence of replica selection.

We find that the programming interfaces of Apache Cassandra lack two essential features to make better scheduling choices, as illustrated by Figure **??**. First, sorting is made without knowing any characteristics of the current request. This makes fine-granularity decisions, e.g., workload-aware choices, nearly impossible. Second, it is not possible to include additional data in the routed request to guide subsequent steps such as local scheduling, for example by specifying a priority score calculated by the coordinator node. In Hector, defining a new replica selection strategy simply consists in extending an abstract class and implementing a sorting function. This function takes the unordered list of replicas and the current request as parameters and must return an ordered list of replicas. Of course, it may also leverage external information (being built by other modules of Hector), as well as internal information (being built in the replica selector instance itself). In addition to request identification, Hector provides interfaces to include custom data in the request to be transferred over the network and retrieved later on replicas.

Local scheduling. As explained in Section **??**, the system is highly concurrent and divides its execution model into various stages. Stages are a general abstraction in the execution model, meaning that each stage simply handles a linked queue of self-contained runnable objects and thus does not know about the nature of the operations it is responsible for. Moreover, the queue instantiation is hard-coded, which makes it impossible to associate different local scheduling policies to different stages.

[1] https://anonymous.4open.science/r/hector

In Hector, we generalize the stage concept by setting the queue as a parameter in the class definition. We also augment the runnable operation objects to include various information about the current request, making any queue implementation aware of the current request characteristics. Moreover, any data added by the replica selection component (on the coordinator) may be retrieved to help taking local scheduling decisions. The local scheduler instance can also rely on external (e.g., data that it gets from replica selection) and internal information to make better ordering decisions.

State propagation. Getting the instantaneous state of remote servers is unfortunately not possible in distributed systems. However, even an out-of-date view can be of interest when talking scheduling decisions. This is why some existing proposals monitor server state characteristics (e.g., queue sizes, average service time, or number of I/Os). This data often forms the basis of replica scoring decisions [39]. The challenge comes from the channels by which we retrieve information: one must periodically transmit values of interest at a sufficiently high rate to take advantage of fairly recent information.

In Hector, each server holds its own copy of a *cluster state* data structure. This cluster state consists of a list of *endpoint state* entries. Each endpoint state corresponds to a specific server in the cluster (including the current host) and maps a value to a property of interest, which we call a *fact*. Let us describe the building process for the cluster state, which is summarized in Figure **??**. The definition of a fact i consists of 4 functions:

- The measurement M_i defines how to retrieve the value to send over the network.
- The serializer S_i (resp. deserializer D_i) encodes (resp. decodes) a value to a byte buffer.
- The aggregator A_i combines received values into a unique value to save in the endpoint state. This component is useful to define custom aggregation operators, e.g., (weighted) moving averages.

The state propagation module extends the internode messaging service of Apache Cassandra. This means that the user may include state values in any message, being a message purposely built for state propagation, or an already-existing message (piggybacking). Hector examines the previously-defined facts and executes the corresponding measurement functions to get raw state values on the host, which are then gathered as *state feedback*. When the state feedback is ready, a timestamp is added and data is transformed into a byte buffer through the fact serializer. Moreover, the byte buffer is prefixed by the fact identifier to know how to deserialize it in the future. Then, the state feedback bytes are added to the message before being sent over the network. When the packet is received, the message handler proceeds to decode the byte buffer and retrieves the state feedback. Each included value is added to the local endpoint state that corresponds to the message sender by applying the aggregation operation of the fact. Note that we must be careful when dealing with ordered values: the high concurrency of Apache Cassandra implies that some feedback f_a from a given endpoint may arrive before feedback f_b from the same endpoint, whereas f_a contains values measured *after* values of f_b. This is why we associate a timestamp with each feedback. As we only compare feedback coming from the same peer, timestamp values stay comparable, and we may choose to discard values that are older than the most recent processed feedback.

Workload oracle. Although they are generally unknown (or known with little precision), workload characteristics such as the distribution of data item sizes or key access frequencies have a direct influence on the efficiency of scheduling strategies. Being able to predict these characteristics is a clear advantage when designing policies.

In Hector, oracles are the components that give information about these characteristics to other modules. According to the use-cases, there are various ways an oracle can build this information. Of course, the simplest situation is when the characteristics are known beforehand: the oracle may for example load data in memory from static files describing the workload. However, in more common situations, the oracle will have to learn the workload at runtime. This can be done through machine learning techniques,

statistical inference, probabilistic data structures such as Bloom filters [20], etc. In order to ease the evaluation process, each oracle instance is assigned a unique identifier. In this way, other components such as replica selection or local scheduling are not tied to a specific oracle definition, and the user may switch between different oracle implementations without modifying the calling component.

5.4 Scheduler Implementations

We demonstrate in this section the flexibility of Hector components by implementing state-of-the-art policies such as Dynamic Snitching [**ds2012**] and C3 [39], and we show that it is also easy to test new ideas by introducing two novel replica selection/local scheduling strategies that we named *Popularity-Aware* and *Random Multi-Level*.

Dynamic Snitching. This is the default replica selection strategy in Apache Cassandra [**ds2012**]. Dynamic Snitching is based on replica scoring. Each coordinator measures service time for each sent request and maintains a history of these measurements for each server in the cluster. The coordinator assigns a score to each replica by computing an Exponentially Weighted Moving Average (EWMA) of recorded latencies. When processing a request, it selects the replica with the current lowest score. A parallel process updates scores every 100 milliseconds to avoid being in the critical path of query service, and another process resets scores every 10 minutes to allow slow servers to recover. We reimplement Dynamic Snitching as a replica selection module in Hector.

C3 (scoring-only). The C3 replica selection algorithm was proposed to overcome some weaknesses of Dynamic Snitching [39]. This strategy aggregates information on the cluster state by including values of interest in the responses of each replica, such as the read operation queue size q and the average service rate μ. We easily reimplement this process using the state propagation module of Hector: we measure the queue size q and the number of completed operations w in the `ReadStage` thread pool and we transmit this information. In the aggregation step, we compute the average service rate as $\mu = \frac{w-w'}{t}$, where w' and t are respectively the previously-received value of w and the time elapsed since the last update, and we add q and μ in corresponding moving averages. C3 also maintains the current count c of remote pending requests for each replica, and an history of observed latencies R, which are finally used to compute a score according to the cubic function $\bar{R} - \bar{\mu}^{-1} + \bar{\mu}^{-1}(1 + cm + \bar{q})^3$, where $\bar{R}$, $\bar{\mu}^{-1}$ and $\bar{q}$ are respectively the EWMAs of observed latencies, service times, and queue sizes, while m is the number of servers. Here, using a cubic term aims to penalize servers with longer queues, which is expected to lead to better balancing. In the original proposal, the replica scoring is coupled to a rate limiting process, which monitors the health of each replica and limits the sending rate towards a replica when it is suspected to be overloaded. For the sake of simplicity, we do not implement this part in this example, although it would be easy to integrate in Hector without any conflict with existing components.

Popularity-Aware. Workloads often exhibit biased popularity distribution on partition keys. We design a new replica selection strategy that is able to learn and leverage this distribution. The popularity of a key (at a given time) is defined as the ratio between the number of accesses for this key and the total number of requests. We implement a workload oracle that maintains a histogram of key accesses. When a request is received by the coordinator, it asks the oracle to record a new access, which is done by first hashing the key to a positive long integer and incrementing the corresponding access count in a map. By hashing the key, we limit the memory footprint of the data structure (at a small cost on the precision of the distribution). For example, recording accesses to a dataset that comprises 1 billion keys requires at most 16 GB of memory. We also ensure that the map implementation guarantees atomic, lock-free increments to enable efficient concurrent mutations.

We schedule requests according to the popularity value. The idea is to maximize the benefit we

obtain from caching at the different servers, while balancing the load of serving popular content over multiple servers. When a key is considered popular, the probability to find the corresponding data item in the cache of any of the replicas holding it is high. Therefore, all replicas are expected to be able to respond without performing costly disk-read operations, and the best choice is to spread the load over these replicas. On the other hand, requests for unpopular keys will take advantage of the cache memory more if they are always executed on the same server, in order to avoid the eviction of the corresponding data items. In summary, if the popularity of a key is above a user-defined threshold, we schedule the request according to a round-robin strategy; otherwise, we always schedule the request on the same (first) replica. For example, with 1 million keys and a popularity distribution following a Zipf law with parameter 1.5, setting a threshold of 10^{-5} leads to 0.1% of keys marked as popular when the learning process has converged.

First-Come First-Served. This is the default local scheduling strategy in Apache Cassandra. Read operations are simply stored in a wait-free concurrent linked queue, and there is no priority mechanism. This is strictly equivalent to the standard First-Come First-Served algorithm.

Random Multi-Level. Priority queues are a common solution to execute operations in a statically-defined order. However, existing standard implementations need thread synchronization when used in a highly concurrent environment, which may degrade overall throughput. Inspired by the work on Rein [36], we emulate a priority queue, while avoiding any related thread contention, with the following randomized process. We define a Random Multi-Level (RML) queue as an ordered list of n wait-free concurrent linked queues. Each sub-queue q is associated a weight $w_q = \alpha^{n-q+1}$, i.e., the first queue has weight α^n, the second queue has weight α^{n-1}, and so on, where α is a user-defined positive coefficient. A given operation entering a sub-queue q has a priority value equal to $n - q + 1$. In other words, operations entering the first sub-queue will have the highest priority, whereas operations entering the n-th sub-queue will have the lowest priority. The RML queue is processed by generating a random integer x between 0 and the sum of weights $\sum_{q=1}^{n} w_q$, and dequeuing the first sub-queue q' such that $x \leq \sum_{q=1}^{q'} w_q$ (if q' is empty, we generate another random number and repeat the process). For example, with $n = 2$ and $\alpha = 2$, the first sub-queue would have priority 4 and the second sub-queue would have priority 2. In other words, the first sub-queue would be treated first with probability $2/3$. For reasonable values of n and α, the probability to treat each sub-queue is high enough to ensure that no starvation occurs.

We implement a priority-based local scheduling policy by retrieving a priority value from each request and pushing read operations in an RML queue. This priority value is defined during the replica selection step on the coordinator server by leveraging the data injection and transmission mechanism of Hector. In this manner, we improve the flexibility of the scheduling process, as requests are locally executed according to a custom order that is directly decided by the coordinator. For example, we may consider that each priority value depends on the size of the requested data item, i.e., requests for data items whose size is below a given threshold must be processed with higher priority. However, this implies that we must be able to estimate the size of requested data items. In this paper, for illustration purposes, we use a simple workload oracle that is able to extract the size of a data item from the corresponding partition key, but a real use-case would require a more sophisticated approach (e.g., Bloom filters [20]).

5.5 Experimental Evaluation

We evaluate Hector through several experiments on a real cluster. We compare the cost incurred by the generalization of scheduling components and newly-introduced features in the framework to the unmodified system, in particular in terms of overhead on throughput and latencies. We also illustrate the possibilities of Hector by showing how it enables easy comparisons of different approaches in scheduling

Table 5.1 – Absolute and relative (average) differences on the observed maximum attainable throughput of vanilla Apache Cassandra and Hector.

	Cassandra	Hector	Abs. diff.	Rel. diff.
Throughput	632 911 ops/s	640 205 ops/s	7294 ops/s	1.15%

requests with various assumptions on the workload and the environment.

We run all experiments on the large-scale experiment platform Grid'5000 [**balouek2012adding**]. We use 15 identical servers located in the same geographic cluster, each equipped with a 18-core Intel Xeon Gold 5220 (2.20 GHz) CPU and 96 GiB of RAM. Data is stored on each machine on a 480 GiB SATA SSD device. Servers are interconnected by 25 Gbps Ethernet, and they all run Debian 11 GNU/Linux. The system runs on Java 11 with the CMS garbage collector. The replication factor is set to 3: each data item is written on 3 replicas, but each read request is processed by only one of these 3 replicas.

As we do not have access to real datasets and traces, we evaluate the system with synthetic workloads. Yahoo's Cloud Serving Benchmark (YCSB) is a commonly used tool to generate such workloads [**cooper2010benchmarking**]. However, we find that YCSB lacks several features for evaluating modern database systems (e.g., advanced workload customization, modular architecture, and reproducibility), and falls behind more recently developed tools. NoSQLBench is one such tool, which permits to tune advanced characteristics of the workload and which takes care of many pitfalls related to benchmarking practice [**nb**]. We run NoSQLBench on additional nodes, located in the same rack as the Hector cluster. We make sure that we bring enough concurrency, and we systematically check that we do not overload the CPU on client nodes to ensure that they are not the bottlenecks in the experiments. In each run, we select a number of keys to store at least 150 GiB of data at each server, a dataset that does not fully fit in memory.

The first set of results is dedicated to the evaluation of Hector itself, as we want to make sure that it behaves identically to Apache Cassandra in the nominal use-case. Then, we illustrate how Hector enables comparing replica selection and local scheduling strategies under various assumptions.

No overhead. We check that the additional features of Hector do not introduce performance overhead compared to the unmodified Apache Cassandra (noted as "vanilla" in what follows). We make sure that both systems are identically configured, and we run them in the same conditions. The replica selection strategy is Dynamic Snitching, and the local scheduling strategy is First-Come First-Served. Key access distribution follow a Zipf law with parameter 0.9, corresponding to a biased popularity. For instance, with 100 keys, the first key is requested with probability 15.5%, the second key with probability 8.3%, and so on, until the 100th key with probability 0.2%. We also bring some heterogeneity: $1/2$ of the values in the dataset have a size of 1 kB, $1/3$ have a size of 10 kB, and the last $1/6$ have a size of 100 kB. This setting simulates a common type of read-only workload to benchmark Apache Cassandra. Moreover, no additional data is transmitted through the state propagation module in Hector, and there is no workload oracle.

Figure **??** shows the maximum attainable throughput in each system while Figure **??** presents the mean, the median, 90th and 99th percentiles of the latency when read requests arrive according to a given rate (200 000 and 500 000 operations per second, which in this case correspond respectively to 30% and 80% of the maximum capacity). Each experiment runs for 10 minutes and is repeated 10 times (each value represents the mean among the runs, and error bars indicate the standard error on the mean). Tables 5.1 and 5.2 summarize the absolute and relative (average) differences between vanilla Apache Cassandra and Hector.

We see that both systems seem to behave identically. Hector attains a slightly better throughput

Table 5.2 – Absolute and relative (average) differences on the observed latency of vanilla Apache Cassandra and Hector, with two different arrival rates.

Stat.	Arrival rate	Cassandra	Hector	Abs. diff.	Rel. diff.
Mean	200 kops/s	0.429 ms	0.435 ms	0.006 ms	1.38%
	500 kops/s	0.670 ms	0.675 ms	0.005 ms	0.72%
Median	200 kops/s	0.352 ms	0.355 ms	0.003 ms	1.02%
	500 kops/s	0.515 ms	0.518 ms	0.003 ms	0.56%
P90	200 kops/s	0.698 ms	0.703 ms	0.005 ms	0.67%
	500 kops/s	0.946 ms	0.953 ms	0.007 ms	0.72%
P99	200 kops/s	1.007 ms	1.011 ms	0.004 ms	0.43%
	500 kops/s	5.214 ms	5.227 ms	0.013 ms	0.25%

Table 5.3 – Throughput (operations per second) of DS, C3 and PA under two key popularity distributions.

Stat.	Key popularity	DS	C3	PA
Mean	$\mathrm{Zipf}(0.1)$	157 282	358 680	984 252
	$\mathrm{Zipf}(1.5)$	1 193 317	1 212 121	1 194 743
Median	$\mathrm{Zipf}(0.1)$	157 356	1 025 641	1 000 000
	$\mathrm{Zipf}(1.5)$	1 204 819	1 219 512	1 190 476
Min	$\mathrm{Zipf}(0.1)$	156 494	156 985	862 069
	$\mathrm{Zipf}(1.5)$	1 123 595	1 162 790	1 162 790
Max	$\mathrm{Zipf}(0.1)$	158 227	1 086 956	1 041 666
	$\mathrm{Zipf}(1.5)$	1 265 822	1 250 000	1 250 000

(in average 1.15% higher) than Apache Cassandra; the errors indicate that this difference is clearly not significant. Moreover, even if Hector shows higher latencies for the considered statistics, the absolute differences stay in the microsecond scale, with a maximum relative difference of 1.38%. Again, according to the standard error, this difference is not significant. In other words, no performance overhead is observed in Hector for the standard use-case, and we consider the framework to be a stable baseline that we can trust to evaluate and compare various scheduling strategies.

Cache-locality effects. We illustrate how different key access patterns may influence scheduling behavior. In particular, we show that a strategy that correctly leverages the Linux page cache clearly outperforms other algorithms under some assumptions on the distribution of key popularities. We compare 3 replica selection strategies: Dynamic Snitching (DS), which is the default algorithm implemented in Apache Cassandra; C3, which is a proposal coming from the literature [39]; and Popularity-Aware (PA), which is a new algorithm that we propose in this paper (see Section **??**. We run two experiments for 100 minutes, with different assumptions on the key popularity distribution: first, a Zipf law with parameter 0.1, which is a quasi-uniform distribution, and second, a Zipf law with parameter 1.5, which is heavily-skewed and right-tailed. All data items are 1 kB in size, and FCFS is used as the local scheduling policy.

Figure **??** measures the attainable throughput over time for each case, and Table 5.3 summarizes the values. This results in a significant difference between strategies when the popularity skew is small (i.e., $\mathrm{Zipf}(0.1)$): for instance, PA shows a maximum throughput that is 6.58 times higher than the maximum throughput that is obtained with DS. Interestingly, we see that C3 seems to learn how to handle the

workload over time, and is able to attain the same performance as PA after 50 minutes. When the skew is heavier (i.e., $\mathrm{Zipf}(1.5)$), all strategies perform the same.

A possible explanation of these results is the cache management of the operating system that occurs on replicas. When a key is accessed, the key-value store starts by looking in the memtable, i.e., the internal data structure of the LSM tree that stores data temporarily in memory. If the key is not found, it must look for the data in the SSTable[2] files that are stored on-disk. The cache management of these files is entirely delegated to the operating system. Thus, if a key is frequently accessed, the Linux page cache keeps the corresponding SSTable file in memory, and the read operation is fast; otherwise, the file must be loaded before reading data, which is a slow operation. As Popularity-Aware tries to maximize the cache hit ratio, it rarely loads data directly from the disk, and thus performs better than other strategies. When the popularity skew is heavy, the cache hit ratio becomes naturally higher, and all strategies are able to achieve similar throughput.

To confirm this explanation, we plot the volume of data that is read on-disk over time on each replica in Figure **??**. Light areas show the range between the minimum and maximum volume of data that are read on each replica, and the points represent the averages. First, we observe a clear visual correlation between both figures: the higher the throughput, the lower the volume of data that is read. Moreover, we see that for a quasi-uniform popularity distribution, PA rarely reads data directly on-disk (less than 1 MB/s), which indicates that it makes a very efficient use of the Linux page cache indeed. On the contrary, DS reads more than 300 MB of data per second in average, with a peak at more than 400 MB per second for one replica in the cluster.

Heterogeneous scheduling. Finally, we show how local scheduling may help improving performance in case of limited parallelism. We emulate a scenario where a slow storage device limits the number of concurrent read operations to avoid putting too much pressure on the disk. For illustration purposes, we set the maximum number of concurrent read operations to 4, and we compare two local scheduling policies from Section **??**, First-Come First-Served (FCFS) and Random Multi-Level (RML), under the hypothesis that the dataset is heterogeneous, that is to say, it is composed of small data items (1 kB) and large data items (1 MB). We assume that $3/4$ of the dataset is small, as it is often observed that heterogeneous datasets are mostly composed of small items in realistic workloads. For the RML strategy, we set the number of sub-queues to 2, and the priority coefficient α to 10. The first sub-queue is dedicated to read operations for small items (with priority 10^2) and the second sub-queue for large items (with priority 10). In this way, we expect that requests for large items will not block requests for small items, resulting in better overall performance. The key popularity distribution is uniform, and DS is the replica selection strategy. For each experiment, we gradually increase the arrival rate of read operations from 10 kops/s to 60 kops/s (10-minute increments), and we observe how the system handles the pressure.

Figure **??** shows the attainable throughput as a function of the arrival rate for each strategy. Horizontal lines represent the saturation throughput for each strategy. Light bars indicate that the system is saturating, and the request generator cannot reach the wanted arrival rate without causing request timeouts. Figure **??** presents the mean, the median, 90th and 99th percentiles of the latency distribution, also according to the arrival rates. The lines stop when the system reaches the saturation point.

We see from these results that RML outperforms FCFS in the considered scenario, as it is able to support a higher arrival rate (more than 50 kops/s, compared to 40 kops/s for FCFS). Moreover, we see that RML is able to achieve excellent median latencies, even when the arrival rate saturates the system (less than 5 ms with an arrival rate of 50 kops/s). However, the last percentiles are higher (almost 60 ms for the 90th percentile and 500 ms for the 99th percentile), which is due to the lower priority of requests for large items. Figure **??** compares the average latency of requests for small and large items. When

[2]Sorted Strings Table (SSTable) is a format that maps data to keys, and these keys are kept sorted. This is the standard solution to store data in persistent key-value stores.

saturating, the latency of requests for small items increases to about 65% of the latency of requests for large items when FCFS is the local scheduling policy. This is not the case for RML, which, when reaching saturation, is able to keep the latency of requests for small items to about 30% of the latency of requests for large items. As small items are in majority in the dataset, and RML treats them in priority, they are not awaiting in the queue and the overall performance is thus better, at the expense of slower requests for large items.

5.6 Conclusion

Conclusion

Bibliography

[1] S Anand et al. “Minimizing maximum (weighted) flow-time on related and unrelated machines”. In: *Algorithmica* 77.2 (2017), pp. 515–536.

[2] Berk Atikoglu et al. “Workload analysis of a large-scale key-value store”. In: *ACM SIGMETRICS Performance Evaluation Review* 40.1 (2012), pp. 53–64.

[3] Michael A. Bender. “New algorithms and metrics for scheduling”. PhD thesis. Harvard University, 1998.

[4] Michael A Bender, Soumen Chakrabarti, and Sambavi Muthukrishnan. “Flow and Stretch Metrics for Scheduling Continuous Job Streams”. In: *ACM-SIAM Symposium on Discrete Algorithms*. 1998, pp. 270–279.

[5] Peter Brucker, Bernd Jurisch, and Andreas Krämer. “Complexity of scheduling problems with multi-purpose machines”. In: *Annals of Operations Research* 70 (1997), pp. 57–73.

[6] Peter Brucker and Svetlana A Kravchenko. “Scheduling jobs with equal processing times and time windows on identical parallel machines”. In: *Journal of Scheduling* 11.4 (2008), pp. 229–237.

[9] Chandra Chekuri, Sanjeev Khanna, and An Zhu. “Algorithms for minimizing weighted flow time”. In: *Proceedings of the thirty-third annual ACM symposium on Theory of computing*. 2001, pp. 84–93.

[10] Jeffrey Dean and Luiz André Barroso. “The tail at scale”. In: *Communications of the ACM* 56.2 (2013), pp. 74–80.

[11] Giuseppe DeCandia et al. “Dynamo: amazon’s highly available key-value store”. In: *ACM SIGOPS operating systems review* 41.6 (2007), pp. 205–220.

[12] Diego Didona and Willy Zwaenepoel. “Size-aware Sharding For Improving Tail Latencies in In-memory Key-value Stores”. In: *16th USENIX Symposium on Networked Systems Design and Implementation*. NSDI. 2019, pp. 79–94.

[13] Maciej Drozdowski. *Scheduling for parallel processing*. Vol. 18. 2009.

[14] Dror G Feitelson. *Workload modeling for computer systems performance evaluation*. Cambridge University Press, 2015.

[15] Naveen Garg and Amit Kumar. “Minimizing average flow-time: Upper and lower bounds”. In: *48th Annual IEEE Symposium on Foundations of Computer Science (FOCS’07)*. IEEE. 2007, pp. 603–613.

[16] Ronald L Graham. “Bounds for certain multiprocessing anomalies”. In: *Bell System Technical Journal* 45.9 (1966), pp. 1563–1581.

[17] Ronald Lewis Graham et al. “Optimization and approximation in deterministic sequencing and scheduling: a survey”. In: *Annals of discrete mathematics*. Vol. 5. 1979, pp. 287–326.

[18] Leslie A Hall. “A note on generalizing the maximum lateness criterion for scheduling”. In: *Discrete applied mathematics* 47.2 (1993), pp. 129–137.

[19] Vikas Jaiman, Sonia Ben Mokhtar, and Etienne Rivière. "TailX: Scheduling Heterogeneous Multiget Queries to Improve Tail Latencies in Key-Value Stores". In: *IFIP International Conference on Distributed Applications and Interoperable Systems*. DAIS. 2020, pp. 73–92.

[20] Vikas Jaiman et al. "Héron: Taming Tail Latencies in Key-Value Stores under Heterogeneous Workloads". In: *37th Symposium on Reliable Distributed Systems*. SRDS. IEEE. 2018, pp. 191–200.

[21] Hans Kellerer, Thomas Tautenhahn, and Gerhard Woeginger. "Approximability and nonapproximability results for minimizing total flow time on a single machine". In: *SIAM Journal on Computing* 28.4 (1999), pp. 1155–1166.

[22] John Klein et al. "Performance evaluation of NoSQL databases: a case study". In: *Proceedings of the 1st Workshop on Performance Analysis of Big Data Systems*. 2015, pp. 5–10.

[23] Svetlana A Kravchenko and Frank Werner. "Preemptive scheduling on uniform machines to minimize mean flow time". In: *Computers & Operations Research* 36.10 (2009), pp. 2816–2821.

[24] Avinash Lakshman and Prashant Malik. "Cassandra: a decentralized structured storage system". In: *ACM SIGOPS Operating Systems Review* 44.2 (2010), pp. 35–40.

[25] Eugene L Lawler and Jacques Labetoulle. "On preemptive scheduling of unrelated parallel processors by linear programming". In: *Journal of the ACM* 25.4 (1978), pp. 612–619.

[26] Arnaud Legrand, Alan Su, and Frédéric Vivien. "Minimizing the stretch when scheduling flows of divisible requests". In: *Journal of Scheduling* 11.5 (2008), pp. 381–404.

[27] Jan Karel Lenstra, AHG Rinnooy Kan, and Peter Brucker. "Complexity of machine scheduling problems". In: *Annals of discrete mathematics*. Vol. 1. 1977, pp. 343–362.

[28] Stefano Leonardi and Danny Raz. "Approximating total flow time on parallel machines". In: *Journal of Computer and System Sciences* 73.6 (2007), pp. 875–891.

[29] Joseph Y-T Leung and Chung-Lun Li. "Scheduling with processing set restrictions: A literature update". In: *International Journal of Production Economics* 175 (2016), pp. 1–11.

[30] Joseph Y-T Leung and Chung-Lun Li. "Scheduling with processing set restrictions: A survey". In: *International Journal of Production Economics* 116.2 (2008), pp. 251–262.

[31] Muthucumaru Maheswaran et al. "Dynamic mapping of a class of independent tasks onto heterogeneous computing systems". In: *Journal of parallel and distributed computing* 59.2 (1999), pp. 107–131.

[32] Monaldo Mastrolilli. "Scheduling to minimize max flow time: Off-line and on-line algorithms". In: *International Journal of Foundations of Computer Science* 15.02 (2004), pp. 385–401.

[34] Shanmugavelayutham Muthukrishnan et al. "Online scheduling to minimize average stretch". In: *40th Annual Symposium on Foundations of Computer Science*. IEEE. 1999, pp. 433–443.

[35] Michael L Pinedo. *Scheduling*. Vol. 29. 2012.

[36] Waleed Reda et al. "Rein: Taming tail latency in key-value stores via multiget scheduling". In: *12th European Conference on Computer Systems*. EuroSys. 2017, pp. 95–110.

[37] Barbara Simons. "Multiprocessor scheduling of unit-time jobs with arbitrary release times and deadlines". In: *SIAM Journal on Computing* 12.2 (1983), pp. 294–299.

[38] Roshan Sumbaly et al. "Serving large-scale batch computed data with project Voldemort". In: *FAST 2012*. 2012, p. 18.

[39] Lalith Suresh et al. “C3: Cutting tail latency in cloud data stores via adaptive replica selection”. In: *12th USENIX Symposium on Networked Systems Design and Implementation*. NSDI. 2015, pp. 513–527.