Solving the Restricted Assignment Problem to Schedule Multi-get Requests in Key-Value Stores

Louis-Claude Canon, Anthony Dugois, Loris Marchal

To cite this version:

Louis-Claude Canon, Anthony Dugois, Loris Marchal. Solving the Restricted Assignment Problem to Schedule Multi-get Requests in Key-Value Stores. 30th European Conference on Parallel and Distributed Processing, Aug 2024, Madrid, Spain. pp.195-209, 10.1007/978-3-031-69577-3_14 . hal-04784268

HAL Id: hal-04784268
https://hal.science/hal-04784268v1

Submitted on 14 Nov 2024

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

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

Solving the Restricted Assignment Problem to Schedule Multi-Get Requests in Key-Value Stores

Louis-Claude Canon[1], Anthony Dugois[1], and Loris Marchal[2]

[1] FEMTO-ST Institute, Univ. Bourgogne Franche-Comté, CNRS
Besançon, France
{louis-claude.canon,anthony.dugois}@femto-st.fr
[2] LIP, ENS Lyon, CNRS
Lyon, France
loris.marchal@ens-lyon.fr

Abstract. Modern distributed key-value stores, such as Apache Cassandra, enhance performance through *multi-get requests*, minimizing network round-trips between the client and the database. However, partitioning these requests for appropriate storage server distribution is non-trivial and may result in imbalances. This study addresses this optimization challenge as the Restricted Assignment problem on Intervals (RAI). We propose an efficient $(2 - 1/m)$-approximation algorithm, where m is the number of machines. Then, we generalize the problem to the Restricted Assignment problem on Circular Intervals (RACI), matching key-value store implementations, and we present an optimal $O(n \log n)$ algorithm for RACI with fixed machines and unitary jobs. Additionally, we obtain a $(4-2/m)$-approximation for arbitrary jobs and introduce new heuristics, whose solutions are very close to the optimal in practice. Finally, we show that optimizing multi-get requests individually also leads to global improvements, increasing achieved throughput by 27%–34% in realistic cases compared to state-of-the-art strategy.

Keywords: Key-Value Stores · Multi-Get · Scheduling · Restricted Assignment · Intervals · Approximation

1 Introduction

Many theoretical scheduling problems capture the essence of practical challenges in modern distributed systems. Among those, NoSQL databases such as distributed key-value stores (e.g., Dynamo [5] and Apache Cassandra [9]), which spread data over several servers and map items to unique keys, became central components in the architecture of online cloud applications, thanks to their excellent performance and capability to scale linearly with the dataset. They are often subject to high throughput, and must therefore be able to serve requests with low latency to meet user expectations. Hence, the proper scheduling of these requests is of paramount importance, and has a direct effect on the overall observed performance of the system [13]. Their wide adoption in the industry

has led to the development of numerous optimization techniques, in particular to mitigate the well-known tail latency problem [4].

The API of modern distributed key-value stores offer various operations to interact with the dataset, among which single reads and writes are the most common. As the dataset is usually replicated on several servers (in order to ensure accessibility of data in case of node failure), each key is accessible at different *replica* servers, which unlocks the possibility to execute the corresponding read operation on any of these replicas. Most web-services often need to retrieve several data items to perform their own calculations. Thus, some APIs, such as Rein [12], provide a special type of operations called *multi-get* requests, which permit to retrieve several items from a given key set in a single round-trip. When executing such a multi-get request, the key-value store needs to partition the requested key set into several sub-operations at the destination of storage servers, and it should carefully balance the keys between these sub-operations in order to respond as quickly as possible. For example, TailX achieves better performance on heterogeneous workloads than a basic priority-based mechanism by taking into account an estimation of the actual service time of read operations [8].

In this paper, we show how the partitioning and scheduling of a multi-get request may be seen as the so-called Restricted Assignment problem, whose objective is to schedule jobs to machines in such a way that the makespan (i.e., maximum completion time) is minimized, with the additional constraint that a given job can be processed only by a particular subset of machines. Unfortunately, this problem is strongly **NP**-hard. Even though there is no polynomial algorithm with an approximation ratio better than $3/2$ unless $\mathbf{P} = \mathbf{NP}$ [10], algorithms with an approximation ratio of 2 or better have been proposed [10,6,7]. Moreover, the actual variant of the Restricted Assignment problem that applies to multi-get request partitioning is easier than the general problem. This enables us to develop low-cost, guaranteed algorithms, giving good results in practice.

Contributions. We express in Section 2 the partitioning of multi-get requests as the Restricted Assignment problem on Intervals (RAI) and extend in Section 3 an efficient algorithm proposed by Lin et al. [11] to the case with arbitrary jobs. We show that it is a $(2-1/m)$-approximation, running in time $O(m^2+n\log n+mn)$, where m is the number of machines and n is the number of jobs (Theorem 1). Then, in Section 4, we further generalize the RAI problem to the Restricted Assignment problem on Circular Intervals (RACI) by allowing intervals that may begin at the end of the list of machines and go back to the start, which match the usual replication strategy of distributed key-value stores [5,9]. We iterate over ELFJ to develop a new $(4-2/m)$-approximation algorithm called Double ELFJ (DELFJ) with the same time complexity (Theorem 4) in Section 5. Finally, we derive two heuristics from DELFJ and evaluate their performance in simulations in Section 6. We find that, for individual multi-get requests, the solutions given by our heuristics largely improve from simple system-like greedy solutions and remain very close to the optimal (with a median ratio to the optimal of 1.031), and that the throughput of a series of multi-get requests is also increased by our heuristics.

2 Applicative Context & Formal Model

In this section, we introduce the partitioning of multi-get requests and give the formal definition of the corresponding scheduling problem.

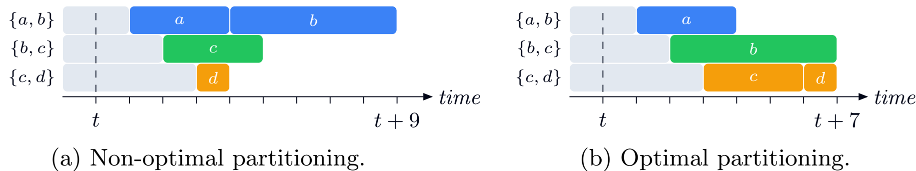
Partitioning Multi-Get Requests in Key-Value Stores. Key-value stores are low-latency databases where each data item is associated with a unique key [9,5]. In these systems, a *get* (or *read*) operation consists in retrieving the value that corresponds to a given key, whereas a *put* (or *write*) operation consists in adding a new association between a value and a key. As it is too large to fit on a single server, the overall dataset is split into several data 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 on different physical servers. Although the replication strategy differs from one system to another, a common and practical way consists in arranging the servers on a virtual ring, and replicating the partition of each server i on its $k-1$ successors $i+1, i+2, \cdots, i+k-1$ (modulo the number of servers m), where k is a small integer ($k = 3$ is a common value). In other words, servers are virtually ordered, and each key/value couple is stored on an interval of k different consecutive servers. The duration of a *get* operation depends on the size of the value being retrieved: even if most values are small, few values with a large size represent a substantial share of the total service time [4]. To some extent, value sizes can be estimated and used for scheduling optimization [8].

In contrast with *single* get operations, *multi-get* requests involve several keys. Such aggregated operations are useful, for instance, to reduce the number of network round-trips between a web-service and the database, as a single end-to-end request often requires to retrieve several data items before responding to the client [12,8]. In a multi-get request, the requested keys (which constitute the *key set* of the request) may be located in different data partitions, which are physically stored on different servers. Thus, the contacted server must split the multi-get request into several sub-requests: each sub-request contains a subset of the initial keys and is redirected towards a storage server holding these keys.

Partitioning a request into sub-requests can be seen as a scheduling problem, where servers correspond to machines, and each single get operation corresponds to a job, whose processing time is the time required to retrieve the data item from the store. Each job may be processed only by a subset of machines, which corresponds to the physical servers on which the requested key is located. The objective is to minimize the response time, that is, the largest completion time of all sub-requests, which corresponds to the maximum completion time of jobs, as illustrated in Figure 1.

In key-values stores, any node may be used to query the database, hence many node may independently send jobs on worker node in a distributed work. However, tools have been proposed to ensure that all nodes have an up-to-date view of the system to make informed scheduling decisions [1].

The Restricted Assignment Problem. In the problem of scheduling jobs on unrelated machines (also known as the $R \,||\, C_{\max}$ problem in Graham's classifi-



(a) Non-optimal partitioning.

(b) Optimal partitioning.

Fig. 1: Two possible partitions of the same multi-get request released at time t containing keys $\{a, b, c, d\}$. Gray areas represent earlier work on each server. Subset of values stored on each server are written on the left.

cation), we are given a set of n jobs $J = \{1, \cdots, n\}$ and a set of m machines $M = \{1, \cdots, m\}$, where each job $j \in J$ has a 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. We focus on a special case of this problem, called the Restricted Assignment (RA) problem and noted $P \,|\, \mathcal{M}_j \,|\, C_{\max}$, 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.

As the RA problem is known to be **NP**-hard in the strong sense, variants have been studied where some structure is brought in the processing sets of jobs. In this paper, we focus on *interval* processing sets. Let us note $\langle a, b\rangle$ the interval[3] ranging from machine a (inclusive) to machine b (inclusive, $a \leq b$). The Restricted Assignment problem on Intervals (RAI) defines for each job $j \in J$ its processing set as $\mathcal{M}_j = \langle a_j, b_j\rangle$. As a generalization of the classical makespan problem $P \,||\, C_{\max}$, the RAI problem, noted $P \,|\, \mathcal{M}_j(\mathit{interval}) \,|\, C_{\max}$ in Graham's classification, remains **NP**-hard in the strong sense.

3 An Algorithm for the Restricted Assignment Problem on Regular Intervals

We focus here on the standard RAI problem $P \,|\, \mathcal{M}_j(\mathit{interval}) \,|\, C_{\max}$, for which Lin et al. [11] have proposed a polynomial algorithm when jobs are unitary. In this section, we generalize their approach to derive a tight $(2-1/m)$-approximation algorithm for the RAI problem with arbitrary jobs, running in time $O(m^2 + n \log n + mn)$ (Theorem 1).

We introduce Algorithm 1, called Estimated Least Flexible Job (ELFJ), which generalizes Lin et al.'s algorithm. ELFJ takes a time λ as parameter and builds a schedule that is guaranteed to finish before this time. In other words, λ denotes an upper bound on the optimal makespan, i.e., the better the quality of the bound, the closer ELFJ gets to an optimal schedule. The algorithm performs two steps. First, it sorts the jobs in non-decreasing order of interval upper bound

[3] We will extend the interval definition later, thus we do not use the common notations of integer intervals.

Algorithm 1 ESTIMATED LEAST FLEXIBLE JOB (ELFJ)

Input: jobs J, machines M and makespan λ
Output: an assignment μ

```
1: sort jobs in non-decreasing order of b_j
2: for all machines i ∈ M do
3:     δ_i ← 0
4:     for all unassigned jobs j ∈ J such that i ∈ ⟨a_j, b_j⟩ do
5:         if δ_i + p_j ≤ λ then
6:             μ_j ← i
7:             δ_i ← δ_i + p_j
8: return μ
```

b_j (in time $O(n \log n)$). Second, it greedily assigns jobs on machines (in time $O(mn)$), and returns an assignment vector μ, where μ_j denotes the machine on which job j is assigned. In the following, we explain how to choose λ to get various guarantees on the quality of the schedule.

Let us start with some notations and definitions. For any interval of machines $\langle \alpha, \beta \rangle$, where $1 \leq \alpha \leq \beta \leq m$, we define $K_{\langle \alpha,\beta \rangle}$ as the set of jobs whose processing set is included in $\langle \alpha, \beta \rangle$, i.e., $K_{\langle \alpha,\beta \rangle} = \{j \in J \text{ s.t. } \mathcal{M}_j \subseteq \langle \alpha, \beta \rangle\}$. We denote the total processing time of jobs in $K_{\langle \alpha,\beta \rangle}$ by $w_{\langle \alpha,\beta \rangle}$, i.e., $w_{\langle \alpha,\beta \rangle} = \sum_{j \in K_{\langle \alpha,\beta \rangle}} p_j$. Let $\tilde{w}_{\langle \alpha,\beta \rangle}$ represent the minimum average work that *any* schedule must perform on machines $\alpha, \cdots, \beta$, i.e.,

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

and let $\tilde{w}_{\max}$ be the maximum value of $\tilde{w}_{\langle \alpha,\beta \rangle}$ over all intervals (that is, $\tilde{w}_{\max} = \max_{1 \leq \alpha \leq \beta \leq m} \left\{ \tilde{w}_{\langle \alpha,\beta \rangle} \right\}$). From these definitions, one may easily see that $\tilde{w}_{\max}$ is a lower bound on the optimal makespan $C_{\max}^{\text{OPT}}$ for a given instance $\mathcal{I}$ of the RAI problem. Let $\langle a, b \rangle$ be an interval of machines such that $\tilde{w}_{\max} = \tilde{w}_{\langle a,b \rangle}$. Then, in the best case, all jobs $K_{\langle a,b \rangle}$ are perfectly balanced on the interval $\langle a, b \rangle$ and finish no earlier than time $\tilde{w}_{\langle a,b \rangle}$. Note that if jobs are unitary, the lower bound can be refined to $\lceil \tilde{w}_{\max} \rceil$.

In the original paper, Lin et al. show that setting λ to $\lceil \tilde{w}_{\max} \rceil$ in ELFJ produces an optimal schedule when jobs are unitary. They also provide a procedure to compute $\tilde{w}_{\max}$ in this specific case. With a similar approach, we show in this section how to choose λ to get a guaranteed approximation ratio when processing times are arbitrary, and we develop a new procedure to find $\tilde{w}_{\max}$ efficiently in the general case.

Computing $\tilde{w}_{\max}$ for Arbitrary Jobs. We provide 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 $\langle \alpha, \beta \rangle$ such that $\alpha < \beta$, the node $\langle \alpha, \beta \rangle$ is the parent of two children nodes $\langle \alpha, \beta - 1 \rangle$ and $\langle \alpha + 1, \beta \rangle$. Let $J_{\langle \alpha,\beta \rangle}$ be the set of jobs whose processing set is exactly $\langle \alpha, \beta \rangle$, i.e., $J_{\langle \alpha,\beta \rangle} = \{j \in J \text{ s.t. } \mathcal{M}_j = \langle \alpha, \beta \rangle\}$, and let $v_{\langle \alpha,\beta \rangle}$ be their to-

tal processing time. We have a recursive relation between the values $w_{\langle\alpha,\beta\rangle}$: for a given interval $\langle\alpha,\beta\rangle$ that has two children intervals, the work $K_{\langle\alpha,\beta\rangle}$ includes the work $J_{\langle\alpha,\beta\rangle}$, the work $K_{\langle\alpha,\beta-1\rangle}$, and the work $K_{\langle\alpha+1,\beta\rangle}$, minus the work $K_{\langle\alpha+1,\beta-1\rangle}$, as it is included both in $K_{\langle\alpha,\beta-1\rangle}$ and $K_{\langle\alpha+1,\beta\rangle}$. Then, for any α,β, we have

$$w_{\langle\alpha,\beta\rangle} = v_{\langle\alpha,\beta\rangle} + w_{\langle\alpha,\beta-1\rangle} + w_{\langle\alpha+1,\beta\rangle} - w_{\langle\alpha+1,\beta-1\rangle},$$

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

An Approximation for Arbitrary Jobs. When jobs have arbitrary processing times, ELFJ does not produce an optimal schedule anymore. However, we show here that, subject to a small adaptation on the value of λ, it still gives a guaranteed solution in this more general case. In the following, $p_{\max}$ denotes the maximum processing time among all jobs of the instance.

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

We give here a sketch of the proof, and refer the reader to the linked research report for a detailed version [2]. Suppose that ELFJ does not produce a feasible schedule in time λ, i.e., there exists at least one job that is unassigned at the end of execution. Let j_0 be the first one. As a consequence, all machines $a_{j_0},\cdots,b_{j_0}$ must finish after time $\lambda - p_{j_0}$, and we know that $b_j \le b_{j_0}$ for all jobs j assigned on these machines (otherwise, j_0 would have been assigned by ELFJ). Let $\gamma \le a_{j_0}$ be the first machine such that all machines $\gamma,\cdots,b_{j_0}$ finish after time $\lambda - p_{\max}$. As a consequence, we know that $a_j \ge \gamma$ for all jobs j assigned on these machines (otherwise, they would have been assigned before γ). The critical step of the proof is now to show that there exists a machine α between γ and a_{j_0} such that

Algorithm 2 Computing $\tilde{w}_{\max}$

```
1: v_⟨α,β⟩ ← 0 for all 0 ≤ α ≤ β ≤ m
2: for all jobs j ∈ J do
3:     v_⟨a_j,b_j⟩ ← v_⟨a_j,b_j⟩ + p_j
4: w̃_max ← 0
5: for all l from 0 to m − 1 do
6:     for all a from 1 to m − l do
7:         b ← a + l
8:         w_⟨a,b⟩ ← v_⟨a,b⟩ + w_⟨a,b−1⟩ + w_⟨a+1,b⟩ − w_⟨a+1,b−1⟩
9:         w̃_⟨a,b⟩ ← w_⟨a,b⟩/(b − a + 1)
10:        if w̃_⟨a,b⟩ > w̃_max then
11:            w̃_max ← w̃_⟨a,b⟩
```

all jobs assigned on machines $\alpha, \cdots, b_{j_0}$ by ELFJ come from the set $K_{\langle \alpha, b_{j_0} \rangle}$. If this is the case, then we necessarily have $w_{\langle \alpha, b_{j_0} \rangle} > (b_{j_0} - \alpha + 1)(\lambda - p_{\max}) + (b_{j_0} - a_{j_0} + 1)(p_{\max} - p_{j_0}) + p_{j_0}$, which leads to the contradiction $\tilde{w}_{\max} < \tilde{w}_{\langle \alpha, b_{j_0} \rangle}$. To find such a machine α, we begin with the interval $\langle a_{j_0}, b_{j_0} \rangle$: if $a_j \geq a_{j_0}$ for all jobs j assigned on these machines, then $\alpha = a_{j_0}$ and we stop here; otherwise, we consider next the interval $\langle a_{j_1}, b_{j_0} \rangle$, where j_1 is a job assigned on $\langle a_{j_0}, b_{j_0} \rangle$ such that a_{j_1} is minimal. We repeat the process until we find α.

4 A General Framework for Circular Intervals

In this section, we present a generalization of the RAI problem to so-called *circular intervals*, which match the usual replication strategy of key-value stores.

Introducing Circular Intervals. Machines are linearly arranged in the standard RAI problem. In contrast, distributed key-value stores organize machines in a virtual *ring*, where the machines able to answer a query for a particular key are consecutively arranged in this ring. Thus, in addition to *regular* intervals $\langle a, b \rangle$ (with $a \leq b$), we introduce here *circular* intervals such that $a > b$. In this case, the corresponding set $\langle a, b \rangle$ includes machines $a, a+1, \cdots, m$ and machines $1, 2, \cdots, b$, i.e., $\langle a, b \rangle = \{a, a+1, \cdots, b\}$ if $a \leq b$, and $\langle a, b \rangle = \{1, 2, \cdots, b\} \cup \{a, a+1, \cdots, m\}$ otherwise. By extension, we call this generalized problem the Restricted Assignment problem on Circular Intervals (RACI).

For two given intervals $\langle a_g, b_g \rangle$ and $\langle a_h, b_h \rangle$, we say that $\langle a_g, b_g \rangle$ *precedes* $\langle a_h, b_h \rangle$ if and only if $a_g \leq a_h$ and $b_g \leq b_h$, and we note $\langle a_g, b_g \rangle \preceq \langle a_h, b_h \rangle$. For a given instance, let Z^* be the set of circular intervals that are associated to at least one job ($Z^* = \{\langle a_j, b_j \rangle \text{ s.t. } j \in J \text{ and } a_j > b_j\}$). We restrict ourselves to instances where the relation $\preceq$ is a total order on Z^*. In other words, for any $\langle a_g, b_g \rangle, \langle a_h, b_h \rangle \in Z^*$, we cannot have $\langle a_g, b_g \rangle \subset \langle a_h, b_h \rangle$. Although it is a particular case of RACI, it remains more general than RAI. Moreover, we assume that there are K types of jobs, and each job of type k has processing time $p(k)$.

An Optimal Procedure for K Job Types. We now introduce a general procedure that solves the RACI problem for the described restricted instances, assuming that one already knows an optimal algorithm $\mathcal{A}$ for the standard RAI problem with K job types.

Theorem 2. *Let $\mathcal{A}$ be an optimal algorithm for the RAI problem with K job types running in time $O(f(n))$. Then there exists a procedure solving the corresponding RACI problem on totally ordered circular intervals in time $O(n^K f(n))$.*

We begin with a few definitions, before giving the procedure and a quick sketch of the proof. Let J^* be the subset of jobs whose processing set is a circular interval ($J^* = \{j \in J \text{ s.t. } a_j > b_j\}$). We call J^* the *circular* jobs, and by extension, the jobs $J \setminus J^*$ are called *regular* jobs. We also partition J^* into K subsets $J_1^*, \cdots, J_K^*$, 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 the last machine m (inclusive) is a *left* job. Equivalently, a circular job j assigned between the first machine 1 (inclusive) and b_j (inclusive) is a *right* job. Thus, a schedule π implicitly defines a partition of each set J^*_k into two subsets G_k and D_k, where G_k (resp. D_k) contains only left (resp. right) jobs. Figure 2 shows an example of such schedule.

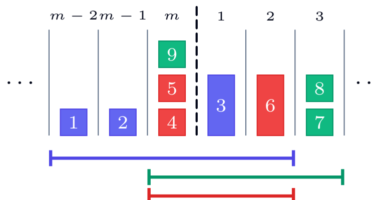


Fig. 2: Example of circular jobs in a schedule. Colors denote common processing intervals. Jobs 1, 2, 4, 5, 9 are left jobs. Jobs 3, 6, 7, 8 are right jobs. Jobs 1, 2, 4, 5, 7, 8, 9 are of type 1 (with $p(1) = 1$). Jobs 3 and 6 are of type 2 (with $p(2) = 2$). We have $G_1 = \{1, 2, 4, 5, 9\}$, $D_1 = \{7, 8\}$, $G_2 = \emptyset$, and $D_2 = \{3, 6\}$.

For now, consider only one type of jobs ($K = 1$). The intuition to compute an optimal schedule is to find which jobs in J^* should be assigned to the left or to the right. 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, while the others should be put on the left. For example, consider only the small jobs in Figure 2. If we suppose that $r = 5$ (arbitrarily), then we guess that the 2 red jobs and the 3 green jobs should be put between machines 1 and 3, and the 2 blue jobs should be put between machines $m-2$ and m, as the red and green intervals are "more on the right" than the blue interval. To capture this intuition, we say that a schedule is *right-sorted* if and only if for all types k, the property $\langle a_j, b_j\rangle \preceq \langle a_{j'}, b_{j'}\rangle$ holds for any jobs $j \in G_k$ and $j' \in D_k$. In the linked report [2], we prove that, by successively swapping jobs of the same type, a non-right-sorted optimal schedule can always be transformed into another optimal solution that is right-sorted, i.e., there always exists at least one optimal right-sorted schedule.

For any vector $\mathbf{r} = (r_1, \cdots, r_K)$ such that $0 \leq r_k \leq n^*_k$ for all k, we introduce the polynomial function $\phi_{\mathbf{r}}$ that transforms any instance $\mathcal{I}$ of the (totally ordered) RACI problem into another instance $\mathcal{I}' = \phi_{\mathbf{r}}(\mathcal{I})$ that does not include any circular interval:

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, effectively removing circular intervals.

For a given instance $\mathcal{I}$ of RACI, let $\Pi(\mathcal{I})$ denote the set of all possible schedules, and for a given $\mathbf{r}$, let $\Pi_{\mathbf{r}}(\mathcal{I})$ be the subset 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 $\Pi(\mathcal{I})$. We define analogously $C_{\mathbf{r}}^{\text{BEST}}$ as the *best possible makespan* among schedules $\Pi_{\mathbf{r}}(\mathcal{I})$. As the subsets $\Pi_{\mathbf{r}}(\mathcal{I})$ define a partition of $\Pi(\mathcal{I})$, we have $C_{\max}^{\text{OPT}} = \min_{\mathbf{r}} \left\{ C_{\mathbf{r}}^{\text{BEST}} \right\}$. In the report [2], we prove the following two statements: applying an optimal algorithm to $\phi_{\mathbf{r}}(\mathcal{I})$ produces a valid solution for $\mathcal{I}$, and the makespan of this solution is at most $C_{\mathbf{r}}^{\text{BEST}}$. This implies that we can find an optimal solution for $\mathcal{I}$ by performing an exhaustive search of the best vector $\mathbf{r}$, proving Theorem 2. The first statement comes from the fact that any schedule for $\phi_{\mathbf{r}}(\mathcal{I})$ is right-sorted with r_k circular jobs on the right for all k, which means that it necessarily belongs to $\Pi_{\mathbf{r}}(\mathcal{I})$. The second statement comes from the fact that there always exists at least one optimal right-sorted schedule, thus, applying an optimal algorithm on $\phi_{\mathbf{r}}(\mathcal{I})$ will necessarily produce a schedule having the best possible makespan among $\Pi_{\mathbf{r}}(\mathcal{I})$.

Revisiting the Unitary Job Case. We study the application of this procedure on the ELFJ algorithm presented in Section 3. We show how to largely reduce the complexity compared to Theorem 2. ELFJ is an optimal algorithm for the standard RAI problem on unitary jobs, which consists in 3 distinct steps: computing the optimal makespan, in time $O(m^2 + n)$; sorting the jobs, in time $O(n \log n)$; 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 2 that we can solve the generalized problem on totally ordered circular intervals in time $O(m^2 n + n^2 \log n + mn^2)$. The following theorem states that we can improve this solution even further by reducing its worst-case time complexity.

Theorem 3. *The totally ordered RACI problem with unitary jobs can be solved in time $O(m^2 + n \log n + mn)$.*

The proof of Theorem 3 explains how to eliminate any redundant work when applying the procedure from Section 4. In particular, finding the correct number of circular jobs that should be put on the right is needed only for the computation of the optimal makespan. The rest of the algorithm can then be processed only once. Furthermore, we can reduce the complexity when computing $\tilde{w}_{\max}$ by relying on a memoization matrix. The interested reader can find the complete proof in the linked research report [2].

5 An Approximation for the Restricted Assignment Problem on Circular Intervals

In this section, we introduce an approximation algorithm to assign jobs on circular intervals, based on the following intuition: under certain conditions, it is possible to split the problem into two sub-problems, such that each of them consider only regular intervals. On each of these sub-problems, we can use the

$(2 - 1/m)$-approximation algorithm presented in Section 3 to get a guaranteed solution.

We consider jobs whose processing set is a circular interval, that is, $J^* = \{j \in J \text{ s.t. } a_j > b_j\}$, and we define the smallest "left" index of these intervals, namely $z_{left} = \min_{j \in J^*} \{a_j\}$, as well as their largest "right" index $z_{right} = \max_{j \in J^*} \{b_j\}$. We assume in this section that the "leftmost" circular interval does not intersect the "rightmost" circular interval, that is, $z_{left} > z_{right}$. This assumption holds in particular for intervals of size k if and only if $m \geq 2(k-1)$, as $z_{left} \geq m - (k-1) + 1$ and $z_{right} \leq k - 1$, i.e., $z_{left} - z_{right} \geq m - 2k + 3$.

Algorithm 3 DOUBLE ELFJ (DELFJ)

Input: jobs J and machines M
Output: an assignment μ

1: $J^* \leftarrow \{j \in J \text{ s.t. } a_j > b_j\}$
2: $\mu \leftarrow$ apply ELFJ on jobs $J \setminus J^*$
3: $z_{left} \leftarrow \min_{j \in J^*} \{a_j\}$
4: **for** all jobs $j \in J^*$ **do**
5: $\quad a_j \leftarrow a_j - z_{left} + 1$ ▷ shift left
6: $\quad b_j \leftarrow b_j + m - z_{left} + 1$ ▷ shift left
7: $\mu^* \leftarrow$ apply ELFJ on jobs J^*
8: **for** all jobs $j \in J^*$ **do**
9: $\quad \mu_j \leftarrow ((\mu^*_j + z_{left} - 2) \bmod m) + 1$ ▷ shift right
10: **return** μ

The proposed algorithm, named DOUBLE ELFJ (DELFJ) and presented in Algorithm 3, works as follows: regular jobs are first allocated on machines using the $(2 - 1/m)$-approximation algorithm ELFJ presented in Section 3. To allocate the remaining jobs (from J^*), we use the same algorithm. However, ELFJ only handles regular intervals. Hence, we first shift all intervals so that the leftmost circular intervals start on machine 1 before applying ELFJ (see

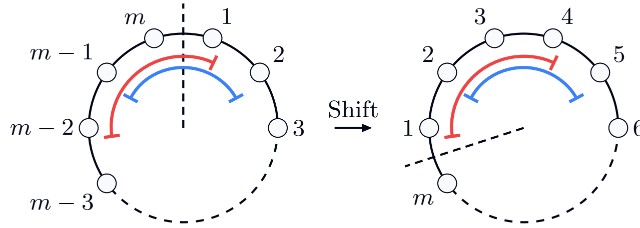


Fig. 3: Shifting the circular intervals "to the left" to transform them into regular intervals. In this example, there are two circular intervals (red and blue). Moreover, $z_{left} = m - 2$ and $z_{right} = 2$. The shifted machine corresponding to the machine with index i has index $i - z_{left} + 1$ if $i \geq z_{left}$, $i - z_{left} + 1 + m$ otherwise.

Figure 3), and we shift back the allocation in the end. Thanks to our assumption on z_{left} and z_{right}, we know that shifting the initially circular intervals will result in all these intervals becoming regular. As the two categories of jobs are allocated separately using a $(2-1/m)$-approximation, we obtain a $(4-2/m)$-approximation algorithm, as stated in the following theorem (see full proof in the research report [2]).

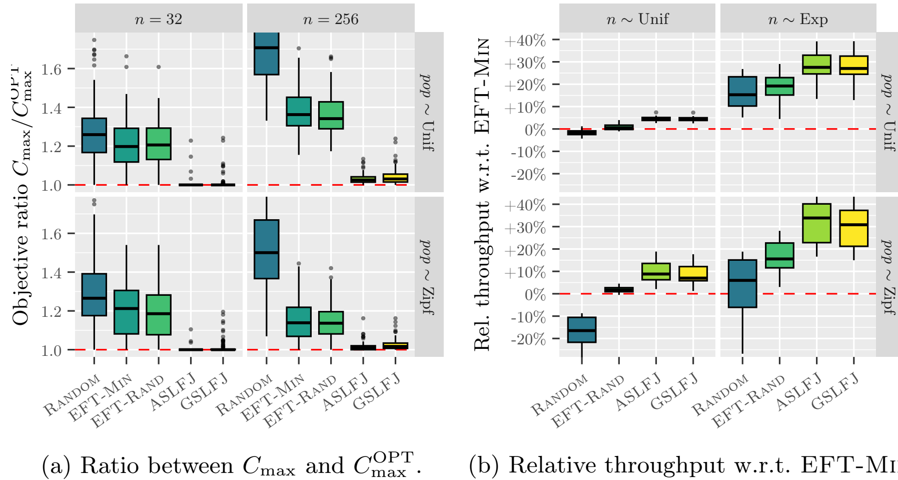
Theorem 4. *Double ELFJ (Algorithm 3) is a tight $(4-2/m)$-approximation algorithm provided that $z_{left} > z_{right}$.*

6 Experimental Evaluation

We now derive a new heuristic from our guaranteed algorithm DELFJ to partition multi-get requests, and we perform a series of experiments to evaluate its practical performance.

Introducing the DSLFJ Heuristic. The drawback of DELFJ is that it uses ELFJ as a sub-algorithm: in each round, it keeps putting jobs on the same machine until it reaches λ, which differs from the optimal by a factor $2-1/m$. Our heuristic, called Double Searched LFJ (DSLFJ), also assigns regular jobs in the first round and circular jobs in the second, but uses a different sub-algorithm to do so. This variant no longer computes an approximated objective value, but instead progressively searches for a feasible makespan by successively applying ELFJ, starting from $\lceil \tilde{w}_{\max} \rceil$. The searching procedure directly depends on how the makespan λ grows through iterations: a slow progression will yield a better final objective, but the worst-case time complexity will necessarily be higher. In the following, we consider two variants of DSLFJ, according to the growing function of λ: Arithmetically-Searched LFJ (ASLFJ), which increments λ by 1 in each iteration, and Geometrically-Searched LFJ (GSLFJ), which doubles λ in each iteration. Their time complexities are respectively $O(m^2 + n \log n + mn \cdot p_{\max})$ and $O(m^2 + n \log n + mn \cdot \log p_{\max})$.

Experimental Settings. We test the quality of ASLFJ and GSLFJ in simulations. The key-value store is characterized by a number of machines m and a replication factor k, which defines the size of each interval of machines. We generate a dataset of 100 000 keys, and we uniformly assign each key to a random machine. Each key κ is associated a corresponding service time t_κ, which is drawn from an exponential distribution with mean 12. The processing time of each job is set to the service time of the corresponding requested key. Each multi-get request is parameterized by the number of keys n that are requested, and the chosen keys that are drawn according to a given popularity distribution. In the following, we consider the uniform distribution (each key has the same probability of being chosen) and the Zipf distribution (with bias 1.0), which is the default in most benchmarks [3]. We compare our heuristics with the following algorithms: Random, which randomly assigns each job to a compatible machine; EFT-Min, which assigns each job to the first compatible machine that



(a) Ratio between $C_{\max}$ and $C_{\max}^{\mathrm{OPT}}$. (b) Relative throughput w.r.t. EFT-MIN.

Fig. 4: On the left, we plot the ratio between the makespan $C_{\max}$ given by each heuristic and the optimal makespan $C_{\max}^{\mathrm{OPT}}$ in different settings (the lower the better). On the right, we plot the ratio between the saturating throughput given by each heuristic on 1000 multi-get requests and the saturating throughput given by EFT-MIN in different settings (the higher the better).

completes the job the earliest; EFT-RAND, which is EFT-MIN with a randomized tie-breaking rule. EFT-MIN is a strategy that actual key-value stores tend to use, even if it is never perfectly implemented in practice due to the usual constraints of distributed systems [13]. When the instance size is not prohibitive, we also compare our heuristics with the optimal solution of a Mixed Integer Linear Program (MILP) solver.

Results. We evaluate the response time of individual requests, and the maximum attainable throughput of the system on a saturating stream of requests. In both experiments, we set the number of machines to $m = 48$ and the replication factor to $k = 3$ (common value in practical systems [9]). The popularity distributions of keys are uniform ($pop \sim$ Unif, top row) and Zipf's law with bias 1.0 ($pop \sim$ Zipf, bottom row).

Response time of individual requests. In Figure 4a, we schedule one multi-get request made of several jobs, and we measure the ratio between the makespan $C_{\max}$ computed by each heuristic and the optimal makespan $C_{\max}^{\mathrm{OPT}}$ computed by the MILP solver. We consider multi-get requests of size $n = 32$ (medium size, left column) and $n = 256$ (large size, right column). Each setting is simulated 100 times. We observe that ASLFJ and GSLFJ give close-to-optimal solutions in the considered settings: the median ratio of ASLFJ (resp. GSLFJ) is at most 1.025 (resp. 1.031), whereas EFT-MIN systematically has a median ratio between 1.139 and 1.362. Moreover, by counting the number of times each heuristic gives the best solution for each instance, we find that ASLFJ gives

the best solution in 99% of the 400 tested cases. Comparatively, without taking ASLFJ into account, GSLFJ gives the best solution in 94% of the cases, whereas EFT-Min is the best only in 5.25% of the cases, and even gives the worst solution in 16.25% of the cases. This confirms that GSLFJ provides a good trade-off between quality and time complexity. Overall, the proposed heuristics give close-to-optimal response time, where EFT-Min is between 15% and 35% slower on average.

Saturating throughput of a stream of requests. In Figure 4b, we test whether optimizing each individual request has an impact on the throughput. We schedule a workload of 1000 multi-get requests and measure the finishing time of the last request to complete. The saturating throughput is defined as the number of requests in the workload divided by this last finish time. In this figure, we plot the ratio between the saturating throughput of each heuristic and the one of the baseline EFT-Min. We make the size of multi-get requests vary according to a uniform distribution between 1 and 256 ($n \sim$ Unif, left column) and an exponential distribution with mean 32 ($n \sim$ Exp, right column). We use this last setting as a realistic workload where small multi-get requests are a lot more probable than large ones. Each experiment is repeated 20 times. We observe that ASLFJ and GSLFJ improve the maximum attainable throughput in all tested settings. However, the improvement is more significant when the size of multi-get requests follows an exponential distribution. When keys have the same probability of being requested (top row), the median saturating throughput of ASLFJ (resp. GSLFJ) is greater than the one of EFT-Min by 4.3% (resp. 4.3%) if $n \sim$ Unif, whereas it is greater by 27.5% (resp. 27%) if $n \sim$ Exp. For a Zipf popularity distribution (bottom row), the median saturating throughput of ASLFJ (resp. GSLFJ) is greater than the one of EFT-Min by 8.8% (resp. 7%) if $n \sim$ Unif, whereas it is greater by 33.9% (resp. 30.8%) if $n \sim$ Exp. We noticed that ASLFJ and GSLFJ were particularly efficient for small multi-get requests (i.e., they find an optimal solution quasi-systematically when $n \leq 10^2$), which are in majority if $n \sim$ Exp. Over the 80 tested workloads, ASLFJ gives the best results in 86.25% of the cases. When ASLFJ is not taken into account, GSLFJ gives the best results in 97.5% of the cases.

Overall, we notice that our heuristics not only improve the response time of individual requests, but also improve the maximum load that the system is able to cope with. This is a non-trivial and interesting conclusion since throughput optimization is similar to load-balancing, which is usually an orthogonal objective to optimizing the individual performance of requests. Depending on the distribution of request sizes and key popularities, the improvement in throughput goes from 27% to 34% in realistic cases.

7 Conclusion

In this paper, we tackle the multi-get request partitioning problem that arises is modern key-value stores by modeling this as a scheduling problem, namely

the Restricted Assignment problem on Intervals (RAI), and proposing approximation algorithms and heuristics to solve it. We first exhibit a $(2-1/m)$-approximation algorithm, and we further extend the RAI problem to *circular* intervals, which fit the configuration of actual replicated key-value stores. In this setting, we propose 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 RACI problem in time $O(n^K f(n))$. This enables us to revisit an optimal algorithm for the RAI problem when jobs are unitary to solve the corresponding RACI problem in time $O(m^2 + n \log n + mn)$. Moreover, we derive a new $(4-2/m)$-approximation algorithm in the general case, which we use as a basis to design new practical heuristics to partition multi-get requests. We evaluate these heuristics through extensive simulations, and we show that they not only improve the response time of individual multi-get requests compared to simple greedy strategies, leading to close-to-optimal allocations, but are also able to increase the maximum attainable throughput of the system by 27%–34% in realistic cases.

As a future work, the next step would be to implement and evaluate our heuristics in a real key-value store, e.g., Apache Cassandra. On the theoretical side, it remains unknown if there exists an efficient approximation algorithm for the particular instances of RACI where circulars intervals are not necessarily totally ordered, i.e., a given circular interval may be strictly included into another. Moreover, we conjecture that there exists an efficient approximation algorithm for RACI that improves on the $4-2/m$ guaranteed factor.

References

1. Ben Mokhtar, S., Canon, L., Dugois, A., Marchal, L., Rivière, E.: A scheduling framework for distributed key-value stores and its application to tail latency minimization. J. Sched. **27**(2), 183–202 (2024)
2. Canon, L.C., Dugois, A., Marchal, L.: Solving the restricted assignment problem to schedule multi-get requests in key-value stores (extended version). Research report (2024), https://hal.science/hal-04516752
3. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with ycsb. In: ACM symposium on Cloud computing. pp. 143–154 (2010)
4. Dean, J., Barroso, L.A.: The tail at scale. Communications of the ACM **56**(2), 74–80 (2013)
5. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: amazon's highly available key-value store. In: SOSP 2007. pp. 205–220 (2007)
6. Ebenlendr, T., Krcál, M., Sgall, J.: Graph balancing: a special case of scheduling unrelated parallel machines. In: SODA. vol. 8, pp. 483–490 (2008)
7. Glass, C.A., Kellerer, H.: Parallel machine scheduling with job assignment restrictions. Naval Research Logistics **54**(3), 250–257 (2007)
8. Jaiman, V., Mokhtar, S.B., Rivière, E.: Tailx: Scheduling heterogeneous multiget queries to improve tail latencies in key-value stores. In: IFIP DAIS. pp. 73–92 (2020)

9. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review **44**(2), 35–40 (2010)
10. Lenstra, J.K., Shmoys, D.B., Tardos, É.: Approximation algorithms for scheduling unrelated parallel machines. Mathematical programming **46**(1), 259–271 (1990)
11. Lin, Y., Li, W.: Parallel machine scheduling of machine-dependent jobs with unit-length. European Journal of Operational Research **156**(1), 261–266 (2004)
12. Reda, W., Canini, M., Suresh, L., Kostić, D., Braithwaite, S.: Rein: Taming tail latency in key-value stores via multiget scheduling. In: EuroSys. pp. 95–110 (2017)
13. Suresh, L., Canini, M., Schmid, S., Feldmann, A.: C3: Cutting tail latency in cloud data stores via adaptive replica selection. In: USENIX NSDI. pp. 513–527 (2015)