Numéro National de Thèse : xxxxxx

THÈSE DE DOCTORAT DE L'UNIVERSITÉ DE LYON

opérée par

l'École Normale Supérieure de Lyon

École Doctorale n^o^ 512

École Doctorale en Informatique et Mathématiques de Lyon

Spécialité : Informatique

présentée et soutenue publiquement le 28 septembre 2023, par :

Anthony Dugois

Scheduling in Distributed Storage Systems

Ordonnancement dans les systèmes de stockage distribués

Devant le jury composé de :

Oliver Sinnen	Associate Professor, Univ. of Auckland	Rapporteur
Safia Kedad-Sidhoum	Professeur des Universités, CNAM	Rapportrice
Denis Trystram	Professeur des Universités, Grenoble INP	Examinateur
Sara Bouchenak	Professeur des Universités, INSA Lyon	Examinatrice
Emmanuel Jeannot	Directeur de Recherche, Inria	Examinateur
Loris Marchal	Chargé de Recherche, CNRS	Directeur de thèse
Louis-Claude Canon	Maître de Conférence, Univ. de Franche-Comté	Co-encadrant de thèse

Table of Contents

Introduction

The continuous increase in the number of Internet users (over 5 billion in 2023, representing approximately two-thirds of the global population), combined with the decrease in the costs of production and maintenance of physical storage resources (in the 2000s, a terabyte cost up to $4,000, compared with $15 in 2022 [85]), accompanied by fast-changing usage habits, has led to the emergence of a period referred to as the *Zettabyte Era*[1], corresponding to an explosion in the quantity of data produced by humanity. Although precise amounts vary depending on the sources, it is estimated that less than 20 zettabytes of data were stored worldwide in 2015, a number that could be multiplied by 8 by 2025. This growth comes from multiple sources and areas of activity, including entertainment, artificial intelligence, smart devices, or scientific experimentation. For example, experiments related to the Large Hadron Collider (LHC) at CERN are known to produce several tens of petabytes[2] of data in a year [22]. These numbers are even more impressive considering that data is often duplicated for durability and availability reasons.

This evolution rapidly required the development and implementation of storage solutions suited to such large amounts of data. Historically, the most widely used systems were Relational Database Management Systems (RDBMS), renowned for their ease of use through the SQL query language, their expressiveness, and their robustness in a wide range of situations. However, they have shown limitations when it comes to evolving and meeting new challenges. Indeed, the storage of ever-increasing quantities of data soon necessitated the adoption of distributed models, so that data can be spread over several machines. The relational model and the robustness guarantees offered by RDBMS are hardly compatible with such an architecture, i.e., it is much harder to scale them up by adding machines to the system (*horizontal* scalability) than by simply improving the capabilities of the host machine (*vertical* scalability). Horizontal scalability quickly appeared to be essential, as multi-site deployment became widely adopted in cloud architectures. In addition, RDBMS offer many features (e.g., joins, secondary indexing, multi-column indexing, etc.) and integrity guarantees (atomicity, consistency, isolation, durability) that are sometimes unnecessary, and which have the effect of limiting the system's performance, availability, and scalability. Finally, the rigidity of the relational model is not always suited to the data to be stored, which can be difficult to structure in tabular form, especially when the schema is subject to frequent changes (user-generated content, time series from heterogeneous equipment, etc.). These limitations have led to the emergence of new storage solutions, known as NoSQL, which have rapidly established themselves as serious alternatives to RDBMS [68]. These systems can offer a wide range of data models, more or less adapted to specific use cases: key/value-oriented databases, document-oriented databases, column-oriented databases, graph-oriented databases, time-series-oriented databases, and so on. In NoSQL databases, the underlying data structures do not suffer from the same limitations as the relational model, and possess, for example, much more interesting availability and scalability properties, to the detriment of certain guarantees on data consistency in the event of temporary failures [1].

NoSQL databases have thus been able to meet a wide range of needs, to the point of becoming essential components in modern infrastructures. As storage, processing, and usability requirements continue to evolve, many improvements have been incorporated over the years, making these systems more and more powerful, but also more and more complex in terms of design. This complexity is accompanied

[1]A zettabyte is equivalent to 10^{21} bytes (1,000 billion gigabytes).
[2]A petabyte is equivalent to 10^{15} bytes (1 million gigabytes).

by increasing difficulty in configuring, maintaining, and controlling them efficiently, particularly under varying and dynamic workloads. A wide variety of variability sources, such as workload heterogeneity, random network fluctuations, transient hardware failures, execution of background processes, poor I/O management, etc., favor load imbalances and head-of-line blocking situations, where queued operations are blocked by other operations in progress, with the effect of worsening the latency of requests. This lack of control makes it very difficult to predict the behavior of storage systems in certain scenarios, and their limits are not always well understood. Ultimately, unexpected slowdowns or even unavailability periods may appear. In the case of service-based architectures, where the response to an external request may require dozens, or even hundreds of service calls, the slowdown of a single internal request has a direct impact on the overall latency for the final user. In other words, even if each service is dimensioned to respond very quickly in 99.9% of cases, slowing down only one request out of 1000 can severely degrade the latency for a majority of users. This problem, commonly referred to as the tail latency problem, is particularly present in key/value-oriented databases (otherwise known as key-value stores), which form the application context of this thesis.

Even if it is sometimes possible to reduce it, a natural variability remains inherent to distributed systems, and maintaining control over the induced side effects is essential. In the case of key-value stores, this implies continuously monitoring the state of the system's resources to be able to detect situations of imbalance or slowdown, and dynamically adapt the scheduling of requests accordingly. Of course, the underlying complexity of the systems involves strong constraints on this scheduling, and the design of an efficient (or even *guaranteed*) scheduler is a particularly difficult problem. In addition, the theoretical performance limits of key-value stores are poorly understood. Several authors have attempted to model these systems in the form of stochastic queuing networks, with the objective, for example, of predicting their average response time [47]. Such an approach quickly showed limitations, as the analytical resolution of these models was in the vast majority of cases out of reach, forcing the authors to resort to costly simulations for systems of non-trivial size.

This thesis proposes an alternative direction based on scheduling theory. The first goal is to establish general theoretical guarantees, possibly of various kinds, on different objective functions specific to key-value stores. The second goal entails designing formal and practical tools to assist researchers in reliably evaluating various characteristics of these storage systems. On a more general level, the third and final goal is to provide guiding principles for future optimizations. Rather than attempting to model key-value stores in their entirety, we focus on the essential aspects for understanding their behavior, so as to capture the intrinsic difficulty of scheduling under their specific constraints without being overwhelmed by too many parameters. The aspects in question are as follows:

- Heterogeneity and semi-clairvoyance of request service times. The natural variability of the system and the non-uniform distribution of the size of the stored data result in highly heterogeneous query execution times. Some techniques can be used to estimate these execution times in advance, but the residual imprecision means that they cannot be known with absolute certainty.

- Unpredictable arrival of requests over time. Requests randomly arrive in the system, and their number can greatly vary depending on time of the day. Scheduling decisions must therefore be made in real time, dynamically, and without knowledge of the future.

- Strong data locality constraint. For reasons of availability in the event of failure, data is replicated on several machines, but not everywhere, as the volume is too large to fit on a single machine. Each request can therefore be executed by a subset of the system's machines only.

- Non-uniform distribution of key access frequencies. Some stored items are particularly popular, while others are rarely requested. In this case, some machines are more in demand, implying a natural imbalance in the system.

- Non-migratory, non-preemptive model. Once a request has been assigned to a machine, it cannot be moved elsewhere, and must be executed to completion. This constraint stems from the fact that, in practice, the benefits of migrating and/or preempting a query during execution is systematically counterbalanced by the cost incurred by the migration or preemption process itself.

- Bound on the response time. Requests must be executed within a given timeframe (as short as possible), to guarantee a reasonable latency for the final user.

This enables us to define models that are simplified in regard to reality, but sufficiently expressive to characterize the main difficulties of scheduling in key-value stores. We study these models using classical combinatorial optimization tools (complexity, optimality analysis, approximation algorithms, competitive analysis) and empirical techniques (simulations, heuristics) to complement the analytical results. Moreover, we conduct experimental evaluations on actual systems to validate findings from the study of formal models. In addition to gaining a better understanding of the underlying performance limits of key-value stores, the aim is to guide the design of efficient schedulers in practice, as well as to develop methods for quantitatively assessing their quality. The main contributions of each chapter are summarized below.

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 basics of scheduling theory (i.e., definitions, Graham's notation, complexity and reductions, online scheduling, as well as some classical results). Then, we review the existing literature on scheduling problems that are relevant to our work. We focus mainly on flow time minimization problems.

Chapter 2: Scheduling Requests in Distributed Key-Value Stores [C3, R2, P3]

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 scheduling problem that consists in executing a stream of requests on servers so as to minimize the maximum weighted flow time $\max w_j F_j$ of these requests, under strong locality constraints due to the availability of data. One may easily see through simple reductions to classical results (e.g., the well-known MAKESPAN problem $P \,||\, C_{\max}$) that this more general scheduling problem is already $\mathbf{NP}$-hard. Thus, we begin to explore simpler variants in order to discover where the line stands between tractable and intractable problems in the considered settings. We derive several optimality and approximation results for the offline version of the problem, that is to say, when all requests are known a priori. Moreover, by considering the variant where preemption is allowed, we leverage an existing result of the literature to show how to find a lower bound on the maximum weighted flow time in any instance of the problem. This enables us to compare and evaluate practical scheduling heuristics according to a common baseline through extensive simulations at the end of the chapter.

Chapter 3: Bounds and Inapproximability under Replicated Datasets [C1, R1]

In typical key-value store systems, each partition of the dataset is replicated on a few machines, 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 so-called processing set restrictions 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 (e.g., response time and throughput) of the system. We consider the online model in which requests randomly arrive in the system over time. Prior work showed how a strong lower bound in $\Omega(m)$ (where m is the total number of machines) may be obtained on the maximum flow time in the general case when the processing sets are completely arbitrary. We argue that this model does not match actual distributed storage systems, which follow structured replication schemes. In particular, it was unknown whether adding more constraints in the processing sets could also lead to such pessimistic bounds. We answer this question in the first part of this chapter by showing that the considered structure has a lot of impact on the achievable competitive ratio. In the second part, we design an exact method to compute the theoretical maximum attainable throughput under a given distribution of key access frequencies and for a given replication strategy. We show through a small case study how this method can be used to compare the performance enabled by different replication schemes.

Chapter 4: Partitioning and Balancing Multi-Get Requests on the Cluster [P1]

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 sub-requests, each of which must be processed by a specific machine 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 sub-request to be late compared to the others. This can be seen as the RESTRICTED ASSIGNMENT problem on intervals of machines (sometimes abbreviated RAI), which constitutes a well-studied 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, which can be used to model more accurately 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 distinct classes according to their processing times.

Chapter 5: Implementing and Evaluating Scheduling Strategies [C2]

In this final chapter, we take a more experimental approach and study a real 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 and carefully designed on the basis of prior work and lessons learned in the previous chapters. The goals of Hector include facilitating the design and evaluation of scheduling algorithms for key-value stores, as well as providing a common baseline for comparing different solutions and improving the evaluation step. 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 find 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.

Résumé français

L'augmentation continue du nombre d'utilisateurs d'Internet (plus de 5 milliards en 2023, représentant environ deux tiers de la population mondiale), conjointe à la diminution des coûts de production et de maintenance des moyens de stockage physique (dans les années 2000, le téraoctet coûtait jusqu'à 4 000 dollars, contre 15 dollars en 2022 [85]) et accompagnée d'une évolution rapide des usages, a conduit à l'émergence d'une période qualifiée de *Zettabyte Era* (« ère du zettaoctet[3] »), correspondant à une explosion de la quantité de données produites par l'humanité. Bien que les chiffres précis varient selon les sources, on estime que moins de 20 zettaoctets de données étaient stockées dans le monde en 2015, alors que ce chiffre pourrait être multiplié par 8 en 2025. Cette croissance provient de multiples sources et domaines d'activité, parmi lesquels le divertissement, l'intelligence artificielle, les équipements domotiques ou l'expérimentation scientifique. Par exemple, les expériences liées au Grand collisionneur de hadrons (LHC) du CERN sont connues pour produire plusieurs dizaines de pétaoctets[4] de données sur une année [22]. Ces chiffres sont d'autant plus impressionnants que les données sont souvent dupliquées pour des raisons de durabilité et de disponibilité.

Cette évolution a rapidement nécessité le développement et la mise en place de solutions de stockage adaptées à une telle quantité de données. Historiquement, les systèmes les plus largement répandus étaient les Systèmes de Gestion de Bases de Données Relationnelles (SGBDR), réputés faciles d'utilisation (grâce au langage de requêtes SQL), efficaces et robustes dans un grand nombre de situations. Ils ont cependant montré leurs limites lorsqu'il s'est agit de les faire évoluer pour répondre aux nouveaux enjeux. En effet, le stockage de quantités toujours plus importantes de données a vite nécessité de passer sur des modèles distribués, afin de pouvoir répartir ces données sur plusieurs machines. Or, le modèle relationnel et les garanties de robustesse offertes par les SGBDR sont difficilement compatibles avec une telle architecture, c'est-à-dire qu'il est beaucoup plus difficile de les faire passer à l'échelle en ajoutant des machines au système (scalabilité *horizontale*) qu'en améliorant simplement les capacités de la machine hôte (scalabilité *verticale*). La scalabilité horizontale s'est vite imposée comme incontournable, le déploiement multi-sites se généralisant avec les architectures de type *cloud*. De plus, les SGBDR proposent un grand nombre de fonctionnalités (jointures, indexation secondaire, indexation multi-colonnes, etc.) et des garanties d'intégrité (atomicité, cohérence, isolation, durabilité) parfois non nécessaires, et qui ont pour effet de limiter les performances, la disponibilité et les capacités de passage à l'échelle du système. Enfin, la rigidité du modèle relationnel n'est pas toujours adaptée aux données à stocker, qui peuvent être difficiles à structurer sous forme tabulaire, notamment lorsque le schéma est sujet à des évolutions fréquentes (contenu généré par l'utilisateur, séries temporelles issues d'équipements hétérogènes, etc.). Ces limitations ont conduit à l'émergence de nouvelles solutions de stockage, dites NoSQL, qui se sont rapidement imposées comme des alternatives sérieuses aux SGBDR [68]. Ces systèmes peuvent proposer des modèles de données très divers, plus ou moins adaptés selon les cas d'utilisation : bases de données orientées clé/valeur, bases de données orientées document, bases de données orientées colonnes, bases de données orientées graphe, bases de données orientées séries temporelles, etc. Les structures de données sous-jacentes utilisées par ces systèmes ne souffrent pas des mêmes limitations que le modèle relationnel, et possèdent par exemple des propriétés de disponibilité et de passage à l'échelle bien plus intéressantes,

[3]Un zettaoctet équivaut à 10^{21} octets (1000 milliards de gigaoctets).
[4]Un pétaoctet équivaut à 10^{15} octets (1 million de gigaoctets).

au détriment de certaines garanties sur la cohérence des données en cas de défaillance momentanée d'une partie du système [1].

Les systèmes NoSQL ont donc permis de répondre à de nombreux besoins, jusqu'à devenir des composants essentiels dans les infrastructures modernes. Les besoins de stockage, de traitement et d'utilisation ne cessant pas d'évoluer, de nombreuses optimisations ont été apportées au fil du temps, rendant ces systèmes de plus en plus performants, mais aussi de plus en plus complexes en termes de conception. Cette complexité s'accompagne d'une difficulté croissante à les configurer, les maintenir et les contrôler de manière efficace, en particulier sous des charges de travail variées et dynamiques. En effet, des sources de variabilité très diverses, telles que l'hétérogénéité de la charge de travail, les fluctuations aléatoires du réseau, les défaillances matérielles momentanées, l'exécution de processus d'arrière-plan, la mauvaise gestion des entrées/sorties, etc., favorisent les déséquilibres de charge et les situations de *head-of-line blocking*, où des opérations en file d'attente sont bloquées par d'autres opérations en cours d'exécution, causant une augmentation de la latence de certaines requêtes. Ce manque de contrôle rend très difficile la prédiction du comportement des systèmes de stockage dans certains cas de figure, et leurs limites ne sont pas toujours bien comprises, conduisant finalement à des ralentissements ou même des indisponibilités inattendues. Dans le cas des architectures à base de services, où la réponse à une requête utilisateur peut nécessiter plusieurs dizaines, voire plusieurs centaines ou même milliers de requêtes internes, le ralentissement d'une seule de ces requêtes impacte directement la latence globale pour l'utilisateur. En d'autres termes, même si chaque service est dimensionné pour répondre très rapidement dans 99.9% des cas, le ralentissement d'une requête sur 1000 peut en fait fortement dégrader la latence pour une majorité d'utilisateurs. Ce problème, couramment nommé *problème de tail latency* [34], apparaît notamment dans les bases de données orientées clé/valeur (autrement appelées *key-value stores*), qui constituent le contexte applicatif de cette thèse.

Même s'il est parfois possible de la réduire, une certaine variabilité reste inhérente aux systèmes distribués. À défaut de l'éliminer complètement, il devient alors primordial de garder le contrôle sur les effets induits par cette variabilité naturelle. Dans le cas des *key-value stores*, cela passe par une surveillance continue de l'état des ressources du système, afin de pouvoir détecter les situations de déséquilibre ou de ralentissement, et d'adapter dynamiquement l'ordonnancement des requêtes en conséquence. Bien sûr, la complexité sous-jacente des systèmes induit des contraintes fortes sur cet ordonnancement, et la conception d'un ordonnanceur efficace (ou même *garanti*) est un problème particulièrement difficile. En outre, les limites théoriques de performance des *key-value stores* sont mal connues. Plusieurs auteurs ont tenté de modélisé ces systèmes sous forme de réseaux de files d'attente stochastiques, dans l'optique, par exemple, de prédire leur temps de réponse moyen [47]. Cette approche a rapidement montré ses limites, la résolution analytique de ces modèles étant dans la grande majorité des cas hors de portée, contraignant ainsi les auteurs à recourir à des simulations coûteuses pour des systèmes de taille non triviale.

Dans cette thèse, nous proposons une direction alternative basée sur la théorie de l'ordonnancement. Le premier objectif est d'établir des garanties théoriques générales, possiblement de plusieurs natures, sur différents critères d'optimisation spécifiques aux *key-value stores*. Le second objectif consiste à développer des outils formels et pratiques afin de faciliter l'évaluation de différentes caractéristiques de ces systèmes de stockage. D'une manière plus globale, le troisième et dernier objectif est de fournir des principes permettant de guider le développement d'optimisations futures. Plutôt que de chercher à modéliser les *key-value stores* dans leur intégralité, nous nous concentrons sur les aspects essentiels à la compréhension de leur comportement, de manière à capturer la difficulté intrinsèque de l'ordonnancement sous contraintes, sans pour autant être submergés par un nombre trop important de paramètres. Les aspects en question sont les suivants :

- Hétérogénéité et semi-clairvoyance du temps de service des requêtes. La variabilité naturelle du système ainsi que la distribution non-uniforme de la taille des données stockées entraînent une forte hétérogénéité des temps d'exécution des requêtes. Certaines techniques permettent d'estimer ces temps d'exécution à l'avance, mais l'imprécision résiduelle ne permet pas de les connaître avec absolue certitude.

- Arrivée imprévisible des requêtes au cours du temps. Les requêtes arrivent dans le système de manière aléatoire et leur nombre peut varier fortement en fonction du moment de la journée. Les décisions d'ordonnancement doivent donc être prises en temps réel, dynamiquement et sans connaissance du futur.

- Contrainte forte de localité des données. Pour des raisons de disponibilité en cas de panne, les données sont répliquées sur plusieurs machines, sans pour autant bien sûr être dupliquées partout, le volume de données étant trop important pour tenir sur une seule machine. Chaque requête ne peut donc être exécutée que sur un sous-ensemble des machines du système.

- Distribution non-uniforme des fréquences d'accès aux données. Certaines données stockées sont particulièrement populaires, tandis que d'autres ne sont presque jamais demandées. Quelques machines sont alors plus sollicitées, impliquant un déséquilibre naturel du système.

- Modèle non-migratoire et non-préemptif. Une fois qu'une requête a été affectée à une machine, elle ne peut plus être déplacée ailleurs et doit être exécutée jusqu'à son terme. Cette contrainte provient du fait qu'en pratique, le bénéfice gagné à migrer ou préempter une requête en cours d'exécution est systématiquement contrebalancé par le coût induit par la migration ou la préemption elle-même.

- Borne sur le temps de réponse. Les requêtes doivent être exécutées dans un délai imparti (le plus court possible), afin de garantir une latence raisonnable pour l'utilisateur.

Ceci nous permet de définir des modèles simplifiés par rapport à la réalité, mais suffisamment expressifs pour caractériser les principales difficultés de l'ordonnancement dans les *key-value stores*. Nous étudions ces modèles à l'aide des outils classiques en optimisation combinatoire (complexité, optimalité, algorithmes d'approximation, analyse de compétitivité) ainsi que de techniques empiriques (simulations, heuristiques) venant compléter les résultats analytiques. De plus, nous menons des évaluations expérimentales sur des systèmes réels afin de valider les résultats provenant des modèles formels. En plus de mieux comprendre les limites sous-jacentes quant aux performances de ces systèmes, cette recherche de garanties a également pour but de guider la conception d'ordonnanceurs efficaces en pratique, ainsi que de développer des méthodes permettant d'évaluer quantitativement la qualité de ces derniers. Les contributions principales de chaque chapitre sont résumées ci-dessous.

Chapitre 1 : État de l'art

Dans ce premier chapitre, nous présentons le contexte de notre étude et nous introduisons quelques notions de base. En particulier, nous décrivons l'architecture générale des *key-value stores* et nous expliquons plus en détails les principaux défis inhérents à ces systèmes. Nous rappelons également quelques bases de la théorie de l'ordonnancement (définitions, notation de Graham, complexité et réductions, ordonnancement en ligne, ainsi que quelques résultats classiques). Ensuite, nous passons en revue la littérature existante sur les problèmes d'ordonnancement en rapport avec notre étude. Nous nous concentrons en particulier sur les problèmes de minimisation du temps de réponse.

Chapitre 2 : Ordonnancement de requêtes dans les *key-value stores* [C3, R2, P3]

Après avoir introduit le fonctionnement des *key-value stores* et les bases de la théorie de l'ordonnancement, nous concevons le cadre théorique que nous utiliserons dans la suite de ce manuscrit. Nous formulons le problème d'ordonnancement qui consiste à exécuter un flux de requêtes sur des serveurs de manière à minimiser le temps de réponse maximum pondéré $\max w_j F_j$ de ces requêtes, sous contraintes fortes de localité dues à la disponibilité des données. Il est aisé de déduire, par des relations de réductions depuis des résultats classiques (par exemple, le fameux problème $P \,||\, C_{\max}$), que ce problème d'ordonnancement plus général est déjà $\mathbf{NP}$-difficile. Nous commençons donc par explorer des variantes plus simples afin de découvrir où se situe la ligne de démarcation entre les problèmes « faciles » (pour lesquels il existe un algorithme dont le temps est borné par un polynôme en la taille de l'instance considérée) et « difficiles » (au sens de la théorie de la complexité) dans les contextes considérés. Nous dérivons plusieurs résultats d'optimalité et d'approximation pour la version *offline* du problème, c'est-à-dire dans l'hypothèse (non-réaliste) où toutes les requêtes sont connues à l'avance. De plus, en considérant la variante où la préemption est autorisée, nous exploitons un résultat de la littérature pour montrer comment trouver une borne inférieure sur le temps de réponse maximum de n'importe quelle instance du problème. Ceci nous permet, en fin de chapitre, de comparer et d'évaluer des heuristiques d'ordonnancement pratiques à partir d'une base commune, au travers de simulations.

Chapitre 3 : Bornes et inapproximabilité sous données répliquées [C1, R1]

Dans les systèmes de *key-value stores*, chaque partition du jeu de données est répliquée sur un petit nombre de machines, ce qui signifie qu'une requête donnée ne peut pas être exécutée par n'importe quelle machine (l'ensemble des machines capables d'exécuter une requête donnée est appelé « ensemble de traitement » pour cette requête). Nous poursuivons l'exploration du cadre théorique décrit dans le chapitre précédent en introduisant des restrictions sur les ensembles de traitement dans nos problèmes d'ordonnancement. Le but est de modéliser la réplication limitée des données dans les *key-value stores* et de comprendre comment différents schémas de réplication peuvent impacter les performances (temps de réponse et débit) du système. Pour ce faire, nous considérons le modèle *online* dans lequel les requêtes arrivent aléatoirement dans le système au cours du temps. Des travaux antérieurs ont montré comment une borne inférieure forte en $\Omega(m)$ (où m est le nombre total de machines) peut être obtenue sur le ratio de compétitivité lié au temps de réponse maximum dans le cas général, lorsque les ensembles de traitement sont complètement arbitraires. Nous montrons que ce modèle ne correspond pas aux systèmes réels, qui suivent des schémas de réplication structurés et déterministes. Jusqu'ici, déterminer si l'ajout de contraintes supplémentaires dans les ensembles de traitement pouvait également conduire à de telles bornes pessimistes restait une question ouverte. Dans la première partie de ce chapitre, nous répondons à cette question en montrant que la structure considérée a en effet un impact important sur le ratio de compétitivité atteignable. Dans la seconde partie, nous concevons une méthode exacte permettant de calculer le débit maximum théoriquement atteignable en fonction de la stratégie de réplication et de la distribution des fréquences d'accès. Nous montrons ensuite au travers d'une petite étude de cas comment cette méthode peut être utilisée pour comparer certains des différents schémas de réplication étudiés préalablement.

Chapitre 4 : Partitionnement et équilibrage de requêtes *multi-get* sur le cluster [P1]

Ce chapitre est consacré à l'étude d'un type spécifique de requêtes, à savoir les requêtes *multi-get*, capables de lire plusieurs données à la fois dans un système de *key-value store*. À son entrée dans le système, une requête *multi-get* doit être divisée en un ensemble de sous-requêtes, chacune devant être traitée par un serveur spécifique du cluster. Ce processus de partitionnement doit être effectué de manière à minimiser le temps de réponse global de la requête *multi-get*, c'est-à-dire qu'une sous-requête ne doit pas être trop lente par rapport aux autres. L'autre avantage d'un bon partitionnement est qu'il permet de répartir équitablement la charge sur les machines. Ceci peut être vu comme le problème d'assignation restreinte sur des intervalles de machines, qui constitue un problème d'ordonnancement bien connu (parfois abrégé « problème RAI »). En étendant des travaux existants, nous concevons des algorithmes efficaces et garantis pour des variantes de ce problème, et nous généralisons le modèle formel en introduisant la notion d'intervalles « circulaires », plus réalistes quant à la stratégie de réplication utilisée dans les *key-value stores*. Dans ce cadre, nous proposons une méthode générale pour calculer le temps de réponse optimal de toute instance du problème RAI avec intervalles circulaires, à la condition que les requêtes puissent être classées en (au plus) K classes distinctes.

Chapitre 5 : Implémentation et évaluation de stratégies d'ordonnancement [C2]

Dans ce dernier chapitre, nous adoptons une approche plus expérimentale en nous concentrant sur un système de *key-value store* persistent, distribué et répliqué. Nous commençons par une étude de cas d'Apache Cassandra, un *key-value store* standard de l'industrie largement utilisé en production. Nous détaillons comment les requêtes sont effectivement ordonnancées dans un tel système, et nous identifions plusieurs difficultés connexes. Nous présentons ensuite Hector, un framework modulaire construit par-dessus Apache Cassandra, dont le but est de faciliter la conception d'algorithmes d'ordonnancement, ainsi que d'améliorer l'étape d'évaluation en fournissant une base logicielle commune permettant de comparer différentes solutions. Nous décrivons les différents composants d'Hector et nous illustrons leur utilisation au travers de plusieurs implémentations. Ensuite, nous menons une série d'expériences pour vérifier qu'Hector lui-même n'apporte pas de surcoût significatif au système, et nous montrons comment évaluer divers aspects de performance liés à l'ordonnancement dans les *key-value stores*. Nous illustrons par exemple que dans certaines conditions sur la charge de travail, l'utilisation effective du cache du système d'exploitation peut améliorer significativement le débit du système. Un autre exemple est que le réordonnancement local des requêtes sur chaque machine peut améliorer le temps de réponse global lorsque la distribution des tailles des données stockées est hétérogène.

1
Preliminaries & Related Work

This first chapter introduces the general notions needed for a thorough understanding of this thesis, and surveys related work from the scientific literature. It starts with an overview of a specific kind of distributed storage systems, namely key-value stores, which constitute the applicative context throughout the manuscript (Section 1.1). We also review the latest advances on several aspects of key-value stores in Section 1.2. Then, we introduce the basic concepts of scheduling theory in Section 1.3, and we recall some classical results. Finally, we explore related work on scheduling in Section 1.4, with a particular focus on response time minimization in various settings.

1.1 Introduction to Key-Value Stores

We begin with a general overview of key-value stores (Section 1.1.1), before delving deeper into their distributed architecture (Section 1.1.2) and persistence mechanisms (Section 1.1.3).

1.1.1 Overview

Modern large-scale applications are increasingly built on top of service-oriented cloud architectures, where several hundreds of services are deployed on tens of thousands of machines and interact with each other (see Figure 1.1). These services may be categorized as stateless, that is to say, they simply perform memoryless computations and/or aggregate data from other services, or as stateful, that is to say, they maintain a state that may be requested or updated through given operations. Of course, the memory is often not sufficient to store the state of a data-intensive stateful service, in which case it traditionally relies on a dedicated storage system. According to the type of data that must be stored by the service, several kinds of storage systems may be used, such as relational databases, graph databases, document stores, etc. Among these, key-value stores became central components in most production systems throughout the years, as they are particularly suited to the storage of very large amounts of unstructured data. These systems are often referred to as NoSQL (Not only SQL) databases, as opposed to traditional Relational Database Management Systems (RDBMS). Various key-value store implementations have been proposed over the years, such as Amazon's Dynamo [35], Facebook's Cassandra [66] (now maintained

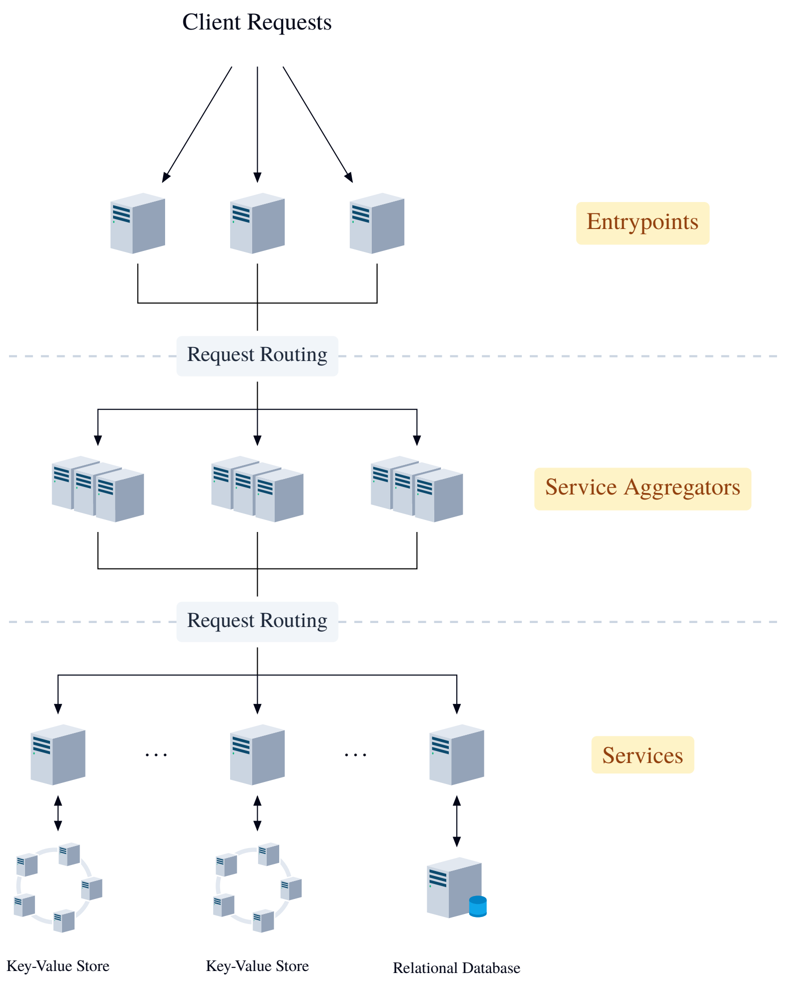
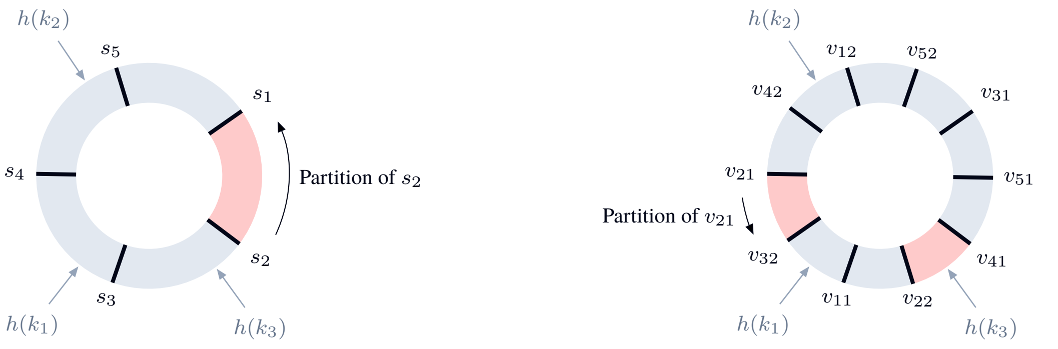


Figure 1.1 – Example of a service-oriented cloud architecture. Client requests reach an entrypoint and are routed towards a service aggregator, which gathers all needed data from individual services. Key-value stores are central components of many data-intensive services in modern large-scale applications.

by the Apache Software Foundation), LinkedIn's Project Voldemort [96], Riak KV [63], Redis [28], Memcached [59], etc.

In a nutshell, a key-value store identifies each piece of data with a unique key, and operates on the dataset through a set of simple operations, the two most important ones being the read operation (i.e., `read(key)`), which reads the data item corresponding to the provided key, and the write operation (i.e., `write(key,value)`), which inserts a new entry in the dataset. The actual programming interface differs between implementations, but most key-value stores provide similar sets of operations.

Because of their apparent simplicity, key-value stores are extremely versatile systems. Their use cases include, but are not limited to, recording user data [45], storing activity logs [27], monitoring data in scientific projects [91], or storing statistics [102]. They are particularly useful when dealing with data that do not need to be structured. Although this prevents supporting complex queries (e.g., joins) as traditional RDBMS, it makes them able to scale remarkably well horizontally, i.e., adding new nodes to the cluster when the workload requires it is easy. This also unlocks handling very high throughputs (of the order of 10^6 requests per second in some cases) and storage capacities (from some gigabytes to



(a) Consistent hashing (standard). (b) Consistent hashing (with virtual nodes).

Figure 1.2 – Dataset partitioning over consistent hashing. The output of the hash function h is represented as a ring. In the standard case (left), each physical server s_i is assigned a single position on this ring, and stores the keys k_j whose hash is between its position and the position of its predecessor. When using virtual nodes (right), each physical server s_i is assigned several positions (two in this example, v_{i1} and v_{i2}) on the ring. In both cases, the red areas represent the dataset physically stored on server s_2.

several petabytes of data). Moreover, key-value stores are resilient to failures, and they are designed to be highly available, i.e., any read or write operation remains feasible under network partitions, at the cost of strong consistency, which is often preferred over availability in relational databases.

1.1.2 Distributed Architecture

Let us now focus on the design of distributed key-value stores. Various components are necessary to achieve the excellent properties described in the previous section, and we explain some of them in order to give a better understanding of the architecture of such complex systems. We do not aim at being exhaustive, and we refer the interested reader to prior work for more details [35, 66].

Partitioning. One of the main challenges in the design of a distributed storage system is to decide how to partition the dataset over the available servers, in such a way that scale-in/scale-out operations remain feasible without moving too much data. In fact, this is a well-known problem in RDBMS, in which achieving efficient partitioning is often very difficult due to the relational nature of the data. Key-value stores do not suffer from this issue, although we must be careful on the strategy to adopt. A naive approach for assigning a key to a server of the cluster would be using a hash function and applying a modulo operation on the number of servers (i.e., the chosen server for key k would be $1 + h(k) \bmod m$, where h is the hash function and m is the number of servers). However, adding or removing servers in this case implies migrating a large amount of data between servers, as we get $h(k) \bmod m \neq h(k) \bmod (m{+}1) \neq h(k) \bmod (m-1)$ for most keys k.

To avoid this issue, key-value stores most often rely on consistent hashing. Here, the output of the hash function h is treated as a ring, that is to say, the largest hash value wraps around to the smallest hash value (this ring is sometimes called the *token space*), and each server is assigned a fixed position (a *token*) on this ring. By hashing a key with h, we also get its position on the ring. Then, each key/value couple is assigned to the first server whose position is greater than or equal (walking clockwise) to the position of the key. In this way, each server is responsible for the data partition standing between its position and the position of its direct predecessor. Figure 1.2a illustrates this process. Note that adding (or removing) a server to the cluster now only affects its predecessor and successor on the ring, and thus only a small fraction of the dataset must be migrated when scaling the cluster.

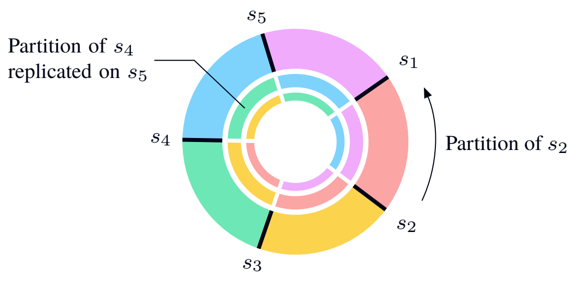


Figure 1.3 – Usual replication scheme of data partitions.

Still, the standard consistent hashing approach presents drawbacks. First, the dataset is often not uniform, and this leads to unbalanced partitions among servers. Second, this does not take into account the possible heterogeneity of the servers, which may have different storage capacities. According to system implementations, two solutions have emerged:

1. Assigning multiple positions to each server on the ring. This is the historical approach taken by Dynamo [35], for instance. A server is now represented by a number of *virtual nodes*, and these virtual nodes are assigned positions on the ring instead of the server. Each server thus becomes responsible for several data partitions (the ones that correspond to its virtual nodes), as shown in Figure 1.2b.

2. Analyze load information and move servers on the ring to balance data on the cluster. This is the historical approach taken by Cassandra [66], for instance.

The first solution has the advantage that adding a server in the cluster now leads to migrating data from more physical servers, with the effect to automatically balance the load more evenly. In the same manner, removing a server from the cluster leads to dispersing its data among the remaining servers. Finally, the number of virtual nodes of a given server may be adapted according to its storage capacity. However, it has been found that there exist trade-offs between the increased operational maintainability permitted by virtual nodes and the availability of data during failures [29], which makes the second solution a good alternative, as we keep full control on the data migration and balancing process. Nowadays, the latest version of Apache Cassandra, for instance, uses a combination of both approaches and makes the number of virtual nodes configurable (from 1, which corresponds to standard consistent hashing, to 16) [30].

Replication. Key-value stores provide high availability and fault tolerance by replicating data partitions over several servers. Each data item is replicated at k different servers, which are called the *replicas*. Moreover, we say that k is the *replication factor*. In most systems, the replication factor is unique and identical for all items of a given instance, although it is configurable when starting the system, and is often set to 3.

The replication is done according to a *replication strategy* (also called a *replication scheme*), which defines how the k copies of a given item are distributed among the servers. The usual strategy simply consists in choosing the k clockwise successor servers on the ring. With standard consistent hashing, this means that a server is responsible for the region of the dataset starting from it and ending at its k-th predecessor (walking counter-clockwise), as shown in Figure 1.3. In this example, the data partition initially stored on server s_2 (red partition) is replicated on servers s_3 and s_4, and the data partition initially stored on server s_3 (yellow partition) is replicated on servers s_4 and s_5. Hence, server s_4 is responsible for the region between its position (inclusive) and the position of server s_1 (exclusive).

Note that when using virtual nodes, walking clockwise on the ring could result in meeting several virtual nodes belonging to the same physical server, which would lead to a replication factor lower than

k for some data items. Thus, the replication strategy is adapted, in order to skip the virtual nodes whose corresponding server has already been encountered.

Request execution. Usually, key-value stores are leaderless systems, which means that any server is able to receive client requests. The server that receives a given request is called the *coordinator* for this request, and is responsible for routing the request to the servers that store a copy of the requested data items. When a server is assigned a position (or several, if using virtual nodes) on the token space, it broadcasts this information to the other servers of the cluster. This means that the coordinator is always able to determine the replicas corresponding to a given request, because the hash function used for partitioning is unique among servers, and the replication strategy is deterministic. After hashing the key of the request, the coordinator walks clockwise on the ring until it finds the responsible server for this key, and deduces the replicas from the replication strategy.

A well-known trade-off in storage system design stands between consistency and availability. Indeed, high availability of data cannot be achieved without relaxing consistency guarantees [1], which means that the replicas may temporarily diverge from each other for a common key/value couple. Such systems are sometimes called *eventually consistent*, as the replicas converge to the same state eventually. Still, in order to provide a consistent view of the data to the clients, key-value stores define what is called a *consistency level* for each request, which is the number of replicas that must acknowledge the request before the coordinator considers it successful. Thus, when receiving a write request, the coordinator forwards it to all replicas (because we want them to eventually store the new value), but does not necessarily await confirmation from all these replicas before responding to the client. Similarly, when receiving a read request, the coordinator redirects it to a subset of the replicas that corresponds to the wanted consistency level, and compares the responses. If these responses match, the consistency level is respected and the coordinator returns the value to the client. Otherwise, the request is considered unsuccessful.

1.1.3 Local Persistence of Data

According to the use-case, key-value stores may be specialized as *in-memory* or *disk-based* (equivalently called *persistent*). The former (e.g., Redis, Memcached) are optimized for small datasets that can fit in memory, and are usually used as caches, session stores, message queues or for real-time applications. In general, they are not designed to save data on disk, or are able to do so under low throughputs. On the other hand, persistent key-value stores (e.g., Dynamo, Cassandra) deal with huge amounts of data, and are heavily optimized to handle very high write throughputs. However, they are usually slower at reading data than in-memory key-value stores, as they must sometimes perform costly disk accesses.

Modern persistent key-value stores are most often based on a special data structure, namely the Log-Structured Merge tree (LSM tree), which consists of an in-memory balanced tree in which writes are performed and kept ordered according to the keys. Then, when the tree reaches a certain size threshold, it is flushed to disk and replaced by a new empty tree. The file created in the process is called a Sorted String Table (SSTable), and contains data ordered by keys. A corresponding index is also saved on disk for efficient lookups. Note that SSTables are immutable files, which means that an update of a given key-value pair simply results in a new occurrence being appended. Deletions are handled similarly by setting a special *tombstone* value for the corresponding key. A background process punctually merges SSTables together in order to reduce the number of files and remove obsolete data.

When performing a read operation, the key-value store first looks into the in-memory tree, which is almost instantaneous. If the key is not found, it looks in the SSTables saved on disk, starting with the most recent one. Several optimizations are possible to speed up the process, such as using Bloom filters to avoid unnecessary file lookups, or using caches to keep the most frequently accessed data in memory.

1.2 Related Work on Key-Value Stores

As being central components of modern computing architectures (from web applications to scientific analysis), key-value stores have been the subject of a huge amount of industrial and academic research since their apparition in the nominal paper of Amazon's Dynamo system [35]. In addition to the continuous optimization of various components over the years, their massive usage motivated researchers to better understand the behavior of these somewhat complex systems. In Section 1.2.1, we review the various techniques proposed in the literature that are related to the pure optimization of various metrics in key-value stores. Then, we focus in Section 1.2.2 on the works that aim at measuring, analyzing and predicting the behavior of key-value stores, for instance through statistical analysis or formal modeling.

1.2.1 Optimizing Performance in Key-Value Stores

Online applications are used by many users dealing with ever-increasing amounts of data, under high expectations in terms of service responsiveness. Not meeting these expectations can have a significant impact on the business of a company. For instance, experiments conducted by Google researchers showed that a 400 ms increase in the latency of search requests for 6 weeks resulted in a 0.6% decrease in the number of daily searches [26]. Given the order of magnitude of the number of requests at Google, this represents a drop of several millions of searches per day. Similar effects have been observed with the Bing search engine, where a 2-second slowdown resulted in a 4.3% decrease in revenue per user [98]. Considerable attention has been given, therefore, to the optimization and performance predictability of distributed systems, including key-value stores.

The most common metrics of interest are the throughput of the system, measured by the number of treated requests per second, and the latency of requests, which corresponds to the time spent from the launching of a request to the reception of its response. In particular, it is well-known that key-value stores are heavily subject to high values in the last percentiles of latency distribution, which can be very problematic. As explained in the previous section, serving a client request in large-scale applications usually requires fetching multiple data items from several services and aggregating the results. As a consequence, the overall latency of the client request is dominated by the slowest of these internal requests. Hence, even if a very small fraction ($< 1\%$) are slow, the fact that a single client request requires fetching many data items makes it likely that at least one of these internal requests will dramatically increase the overall latency. This is what is called the *tail latency* problem [34]: slowing a small fraction of requests may degrade the Quality of Service for most users. It has been observed, for instance, that the 99th percentile of latency distribution is several orders of magnitude higher than the median latency. The tail latency problem is extremely challenging to eliminate, as it has many, often unpredictable, sources in complex systems.

A lot of optimization techniques have been proposed in the literature, with a different focus on the considered objective (increase the throughput of the system, decrease the average latency, mitigate the tail latency problem, etc.). In the following, we propose to review some of the most prominent ones.

Request redundancy. Vulimiri et al. [99] proposed to duplicate requests to mitigate the tail latency problem. The idea is to initiate independently several copies of the same request and use the first result which completes. Of course, this also multiplies the load on the system: if each request is sent twice, the load is doubled, although the number of clients has not increased. Through the analysis of a queueing model and simulations, the authors characterize the conditions under which request redundancy remains beneficial and improves latency without overloading the system. They find that, in a disk-based key-value store system, duplicating requests results in a decrease on both the mean latency (25-33%) and tail latency (up to 50%) when the load stays below 30%. Taking this idea further, Wu et al. [104] proposed

CosTLO, a system that is able to combine several forms of redundancy and adapt the issued requests to meet a given latency variance without increasing costs too much.

Resource sharding. For in-memory key-value stores, Didona et al. [37] introduced size-aware sharding, where requests are assigned to cores according to the size of the item associated with the key. This results in requests for small and large items being sent to disjoint subsets of cores, avoiding head-of-line-blocking (that is to say, a request for a small item being queued behind a request for a large item). The number of cores dedicated to each subset is dynamically adjusted to load balance the system over time. Size-aware sharding enables $20\times$ improvement on the 99th percentile of latency distribution compared to the fastest state-of-the-art competitor.

Replica selection. In replicated key-value stores, the dataset is duplicated to ensure availability and fault-tolerance, which implies that several servers are able to process a given read operation. This gives the opportunity to select the server that is expected to perform the best. Thus, Suresh et al. [97] proposed C3, a replica selection algorithm that adapts itself to the performance fluctuations across servers by computing a health score for each replica and slowing down the sending rate in case of server overloading. By implementing C3 in Cassandra, the authors demonstrate that their solution results in a $3\times$ improvement on the tail latency compared to the default replica selection algorithm. It also increases the throughput by up to 50%. Later, Jaiman et al. [54] showed that C3 may suffer from the heterogeneity of the workload, and presented Héron, another replica selection algorithm that avoids head-of-line-blocking by keeping track of large items using Bloom filters. Héron achieves up to 40% improvement on both the median and tail latency compared to C3.

Multi-get requests. Another mitigation technique consists in batching single read operations into so-called multi-get requests, in order to reduce the natural variability that arises with large numbers of requests and that exacerbates the tail latency problem. This also increases the network efficiency by reducing the number of round-trips. However, a multi-get request must be perfectly balanced, as its service time is equal to its slowest read operation. Reda et al. [89] proposed Rein, a scheduler that is able to identify the bottleneck of a given multi-get request and that assigns different priorities to the contained operations to improve response time. Compared to the default First-Come First-Served policy, the priority-based scheduler Rein reduces the median latency by $1.5\times$ and the 99th percentile latency by $1.9\times$. Under heterogeneous workloads, in which the dataset is composed of small and large items and multi-get requests consist in a varying number of operations, Jaiman et al. [55] proposed TailX, a scheduler that is able to perform better than Rein by taking into account an estimation of the actual service time of read operations. Their evaluation shows a 75% improvement on the median latency and a 70% improvement on tail latency compared to Rein.

Popularity-aware replication. The typical workload of a key-value store is usually skewed, i.e., some data items are accessed much more frequently than others. This creates hotspots in the cluster that may lead to overloading some servers and increase the tail latency. Moreover, maintaining replicas for rarely accessed data items can be seen as a waste of storage and bandwidth resources. These observations lead Cavalcante et al. [31] to propose PopRing, a replica placement strategy based on a genetic algorithm that takes into account the popularity of data items to reduce load imbalance, with the additional goal of minimizing the cost of reconfiguration (i.e., data movement). Their simulations suggest that PopRing could reduce the load imbalance up to 52% compared to the baseline OpenStack-Swift, while incurring a reconfiguration of only 6% of the dataset.

Data structures. Key-value stores use efficient data structures to perform read and write operations, the two most common ones being the LSM tree and the B-tree. However, even if these structures are known to be very efficient, the scale at which key-value store operate is so extreme that they still may become a bottleneck, for instance because of high data write amplifications or too large index set. To address these issues, Wu et al. [103] proposed a new data structure, called LSM-trie and based on a prefix tree, which re-organizes data during compactions (i.e., the merging of SSTables) much more efficiently than the traditional LSM tree. The authors obtain higher write and read throughputs (20× and 10×, respectively) than the state-of-the-art system LevelDB. Similarly, Papagiannis et al. [86] used a variant of a B-tree, called B^ε-tree, that improves on CPU efficiency and throughput compared to LSM trees, which may incur CPU overhead due to their compaction process.

CPU scheduling. An orthogonal concern to throughput and latency is the energy efficiency of the system. The workloads often exhibit temporal patterns, with periods of high activity followed by periods of under-utilization, which offer opportunities for power savings in servers. However, Asyabi et al. [4] argue that the common techniques such as Dynamic Voltage and Frequency Scaling (DVFS) or CPU idle-state mechanisms are not well suited for key-value stores, as the periods of lower activity still require to handle a relatively high rate of traffic, which prevents the system from entering low-power states. Instead, the authors propose Peafowl, a CPU scheduler that bypasses the kernel thread scheduler and that voluntarily unbalance the load among cores to allow some of them to enter low-power states. By monitoring the load, Peafowl is able to dynamically scale the number of active cores during low- and medium-utilization periods, and reduces the energy consumption by up to 40-50% compared to state-of-the-art systems.

Due to the large amount of literature, it is difficult to present an exhaustive list of optimization techniques for key-value stores in this thesis, but the reader may also be interested in other proposals such as hybrid scheduling [36], I/O scheduling [9], or cache warming [87].

1.2.2 Measuring, Analyzing and Predicting Behavior of Key-Value Stores

Parallel to the optimization techniques presented in the previous section, some work has been dedicated to the modeling and performance prediction of key-value stores and their workloads. A greater comprehension of these systems' behavior would enable novel strategies for optimization and a better understanding of their limits. At the moment, authors' attempts in this area include a wide variety of approaches, ranging from queueing models to machine learning techniques.

One of the very first efforts in trying to improve the understanding of key-value stores came from Atikoglu et al. [5], who were the first to perform a large-scale measurement study of a production workload, by collecting the traces of a Memcached (in-memory) deployment. By analyzing various aspects, such as the request rate, the size of the data items, or temporal patterns, the authors were able to propose a representative statistical model. The key findings include a higher-than-expected read/write ratio (of the order of 30:1), power-law distributions for item sizes, a dominant proportion of small items in the requested data, and a skewed key popularity distribution (for instance, 50% of the keys appear in only 1% of all requests in some cases).

Another important aspect in characterizing the performance of key-value stores is the ability to properly measure and benchmark the representative metrics in a reproducible way, which is also a necessity to validate the formal models. To this end, Cooper et al. [33] proposed the Yahoo! Cloud Serving Benchmark framework (YCSB), a generic benchmarking tool for key-value stores that embeds a synthetic workload generator. Since its introduction, YCSB has been widely adopted by the community and used in many studies, for instance to compare the performance of different key-value stores under various configurations [47].

Furthermore, various proposals have been made to model key-value stores more formally, in order to predict their performance and to derive stronger guarantees on their behavior. In addition to their benchmark study, Gandini et al. [47] present a queueing network model where each server is modeled as two queues (one for the CPU, and another for the disk). The model is able to capture the average throughput and mean latency for read and write requests. However, the authors made some simplifying assumptions, such that the absence of coordinator nodes, no communication latency, and approximation of replication with fork-join nodes, which limit the accuracy of the model. Dipietro et al. [38] takes the queueing model one step further by considering local requests (i.e., requests being directly performed by the receiving server) and remote requests (i.e., requests being forwarded to another server), which better captures the behavior of a disk-based key-value store and achieve predictions with a relative error of less than 10% compared to experimental measurements. Other works considered Petri Nets as a modeling formalism. For instance, Osman et al. [84] proposed a Queueing Petri Net model to study the effect of replication and consistency level on performance, although their model sometimes show relative errors of up to 40% compared to measurements. Similarly, Huang et al. [53] used a Coloured Petri Net model to find the best configuration settings according to hardware and workload characteristics. Karniavoura et al. [60] take a measurement-based approach to predict the performance of the NoSQL system MongoDB from various parameters (e.g., cluster size, request rate, read/write ratio, hardware configuration, etc.), applying 3 different regression techniques: Multivariate Adaptive Regression Splines (MARS), Support Vector Regression (SVR) and Artificial Neural Networks (ANN). After training, their evaluation shows that the MARS model achieves the best accuracy, with an average successful prediction rate of 98%. As an orthogonal objective, and after having observed in practice that the relaxed eventual consistency of the dataset is often very close to strong consistency in highly available key-value stores, Liu et al. [77] proposed a probabilistic model checking approach to analyze quantitatively the guarantees we may obtain on this critical aspect.

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 problems have been studied. In this introduction, we intend to provide the non-expert reader with the necessary theoretical background to understand the thesis. We explain the basic concepts of scheduling theory (Section 1.3.1), Graham's problem classification (Section 1.3.2) and we present some classical results on complexity and approximation of these scheduling problems (Section 1.3.3). Then we complete the section with a focus on online scheduling (Section 1.3.4), which constitutes an important topic of 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 perform a computation, or, in a non-computing context, maximizing the usage of classrooms 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 (Single-machine scheduling)**.** *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 $C_1 = 5$, $C_2 = 6$*

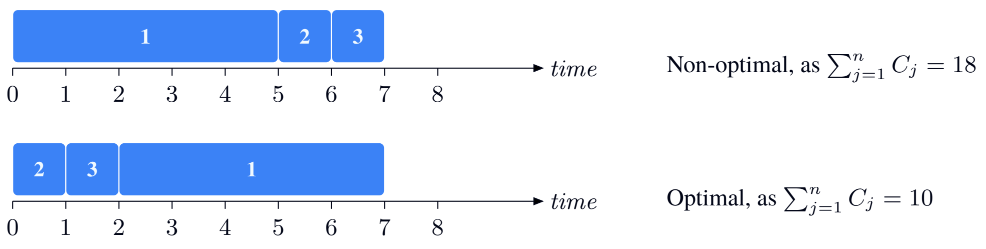


Figure 1.4 – Gantt charts for the schedules described in Example 1.1.

and $C_3 = 7$*, which gives a total completion time of* $5 + 6 + 7 = 18$*. We can 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 a total completion time of 10.*

A convenient representation of a schedule is a Gantt chart, as illustrated in Figure 1.4, which lists jobs (horizontal blue bars, the length of each bar corresponding to the job processing time) to be performed by machines (y-axis) over time (x-axis).

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* for short, is minimized.

Example 1.2 (MAKESPAN problem)**.** *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 is *unrelated* scheduling. 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 (Unrelated problem)**.** *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 2. In this case, an optimal solution could consist in scheduling job 1 on machine 1, job 2 on machine 1 and job 3 on machine 2.

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. [51]. 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 exist many such properties in the literature. We describe the ones that are used in this thesis, and refer the reader to classical scheduling books for more details and examples [39, 88]. The property $p_j = p$ 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, possibly on a different machine; r_j stands for release times, 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, many criteria exist in the literature. We describe in Table 1.1 the various functions that are used throughout this thesis.

Here are some examples of scheduling problems, with their corresponding $\alpha \,|\, \beta \,|\, \gamma$ notation:

- $1 \,||\, \sum C_j$: minimize the average completion time of jobs on a single machine. This corresponds to the basic problem of Example 1.1.
- $P \,||\, C_{\max}$: minimize the maximum completion time of jobs on identical machines (this is the Makespan problem).
- $Q \,|\, p_j = p, r_j \,|\, C_{\max}$: minimize the makespan of jobs on uniform machines, with homogeneous processing times and arbitrary release times.
- $R \,||\, C_{\max}$: minimize the maximum completion time of jobs on unrelated machines (this is the Unrelated problem).

For classical scheduling problems, the reader may refer to the Scheduling Zoo (`http://schedulingzoo.lip6.fr`) for inspecting the associated state-of-the-art results.

Notation	Name	Formal Definition
$C_{\max}$	makespan	$\max_{1\leq j\leq n}\{C_j\}$
$\max w_jC_j$	weighted makespan	$\max_{1\leq j\leq n}\{w_jC_j\}$
$\sum C_j$	average completion time	$\sum_{j=1}^{n} C_j$
$\sum w_jC_j$	average weighted completion time	$\sum_{j=1}^{n} w_jC_j$
$F_{\max}$	maximum flow time	$\max_{1\leq j\leq n}\{C_j - r_j\}$
$\max w_jF_j$	maximum weighted flow time	$\max_{1\leq j\leq n}\{w_j(C_j - r_j)\}$
$\sum F_j$	average flow time	$\sum_{j=1}^{n} C_j - r_j$
$\sum w_jF_j$	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	$\sum_{j=1}^{n}\frac{C_j-r_j}{p_j}$

Table 1.1 – Objective functions appearing in this thesis.

1.3.3 Complexity and Approximation

An important property of computational problems is their complexity (in time and/or space). A decision problem P for which a polynomial-time algorithm exists is said to be in the class **P** (i.e., the running time of the algorithm is bounded by a polynomial function in the size of the input), and may be qualified as *easy*, or *tractable*. On the other hand, a decision problem P for which no polynomial-time algorithm is known may possibly be proven **NP**-hard (i.e., each problem Q in **NP** is polynomially reducible to P), in which case it is said to be *hard*, or *intractable*. An **NP**-hard problem is **NP**-complete if it also belongs to **NP**. An optimization problem is said to be polynomial or **NP**-hard according to the complexity of its associated decision problem.

Scheduling problems are a particular class of combinatorial optimization problems, and their complexity is highly dependent on the considered constraints. For instance, the problem $1\,||\sum w_jC_j$ is a problem that is solvable by a polynomial-time algorithm called Weighted Shortest Processing Time (WSPT), also known as Smith's rule [95], which consists in scheduling the jobs j in non-increasing order of w_j/p_j. On the other side, the Makespan problem is **NP**-hard, even on only two machines, as there exists a polynomial reduction from the Partition problem [71].

There are many polynomial reductions between scheduling problems, and Graham's classification helps to derive the complexity of a problem by comparing it to the complexity of another problem through so-called *elementary* reductions. An elementary reduction is a polynomial reduction between two instances of a given characteristic. For instance, the constraint $p_j = 1$ is a particular case of $p_j = p$, which is itself a particular case of arbitrary processing times. Hence, the scheduling problem $P\,|\,p_j = p\,|\sum w_jC_j$ is at least as hard as the problem $P\,|\,p_j = 1\,|\sum w_jC_j$, and we can directly deduce that there exists a polynomial reduction from the latter to the former. Similarly, there are elementary reductions between objective functions, e.g., $\sum w_jC_j$ generalizes $\sum C_j$, as any instance of $1\,||\sum C_j$ also constitutes an instance of $1\,||\sum w_jC_j$ by setting $w_j = 1$ for all jobs j. Some elementary reductions are given in Figure 1.5.

Nowadays, many scheduling problems of practical interest have been proven to be **NP**-hard, i.e., it is unlikely that a polynomial-time algorithm exists for them. However, several strategies have been developed to get around this difficulty and find good-enough solutions in a reasonable amount of time:

- Restricting to particular cases of the **NP**-hard problem, for which polynomial-time algorithms are

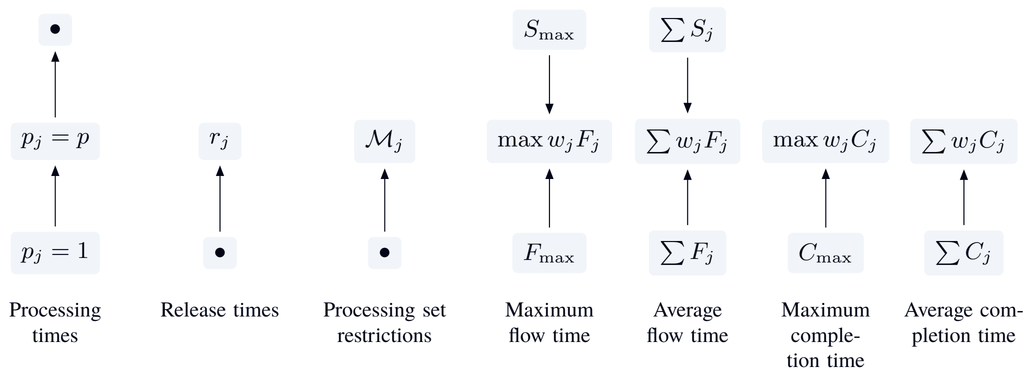


Figure 1.5 – Examples of elementary reductions. $A \to B$ means that A is polynomially reducible to B. A sign • indicates no constraint for the given characteristic.

more likely to exist.

- Using exact methods (e.g., branch-and-bound, dynamic programming, etc.) for small instances.
- Find a close-to-optimal solution, with a guarantee on its quality for any instance of the problem.

This last strategy has given rise to the field of approximation theory, which aims at designing guaranteed approximation algorithms for **NP**-hard optimization problems. The goal is to find a solution in polynomial time, giving an objective value whose difference with an optimal solution is bounded by a constant factor. We recall some basic definitions in the following.

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. In other words, for all inputs $\mathcal{I}$,*

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

where

- *$f^{\mathcal{A}}(\mathcal{I})$ is the objective value of the solution returned by $\mathcal{A}$ for input $\mathcal{I}$,*
- *$f^{\mathrm{OPT}}(\mathcal{I})$ is the objective value of an optimal solution for input $\mathcal{I}$, and*
- *$\rho \geq 1$.*

We say that ρ is the approximation ratio *of $\mathcal{A}$. Note that the definition may be easily adapted to maximization problems.*

When analyzing an approximation algorithm, ideally we want to know the best possible approximation ratio this algorithm enables to reach, in order to have an accurate idea of the difference it has with an optimal solution. The concept of tightness has been introduced to characterize this idea.

Definition 1.2 (Tightness)**.** *Suppose that an algorithm $\mathcal{A}$ is a ρ-approximation for a given problem. The approximation ratio of $\mathcal{A}$ is said to be* tight *if*

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

If the previous condition holds and there exists no instance $\mathcal{I}$ such that $f^{\mathcal{A}}(\mathcal{I}) = \rho \cdot f^{\mathrm{OPT}}(\mathcal{I})$, the approximation ratio of $\mathcal{A}$ is said to be asymptotically tight.

Remark that we may now define the notion of approximation scheme as following. For a given minimization problem, a Polynomial-Time Approximation Scheme (PTAS) is a $(1+\varepsilon)$-approximation algorithm for this problem that runs, for any fixed $\varepsilon > 0$, in a time polynomial in the size of the input but not necessarily in $1/\varepsilon$ (e.g., $O(n^{1/\varepsilon})$). A Fully Polynomial-Time Approximation Scheme (FPTAS) is a particular case of a PTAS that runs in a time also polynomial in $1/\varepsilon$ (e.g., $O(n^2(1/\varepsilon))$).

Approximation theory has been successfully applied to many scheduling problems over the years. A famous result is Graham's bound for the Makespan problem, which states that any greedy, list-scheduling algorithm (i.e., scheduling each job on the least-loaded machine) has a tight approximation ratio of $2 - \frac{1}{m}$, where m is the number of machines [50]. This result may be refined by considering the Longest Processing Time first rule (LPT), which is a list-scheduling algorithm that schedules jobs in non-increasing order of their processing time. The approximation ratio of LPT is $\frac{4}{3} - \frac{1}{3m}$ for the Makespan problem [50].

1.3.4 Online Scheduling

Many scheduling problems may be qualified as *online*, i.e., the complete input cannot be known in advance, either because jobs arrive one by one as a list, without any notion of time, or because they arrive as a continuous stream over time. In the former model, each job must be scheduled before the next one to appear (this is sometimes called the *online-over-list* model). In the latter model, each job is unknown until its release time, but may not necessarily be scheduled before the next one to appear (this is sometimes called the *online-over-time* model), which is noted as $online\text{-}r_j$ instead of r_j in Graham's notation. Moreover, some properties of a job may remain unknown until its full completion (e.g., its exact processing time), in which case the model is characterized as *non-clairvoyant*.

Even under such difficult constraints, we still want to be able to build schedules whose objective measure is as close as possible to an optimal solution, similarly to what is done in classical approximation theory. Competitive analysis [94] constitutes the standard approach. The idea is to compare the performance of an *online* algorithm, which must make decisions without knowing the full input in advance, to that of an optimal *offline* algorithm, which knows all jobs and their characteristics. Analogous to a ρ-approximation algorithm, we may define the notion of ρ-competitive algorithm.

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* ***offline*** *objective value. In other words, for all inputs $\mathcal{I}$,*

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

where

- *$f^{\mathcal{A}}(\mathcal{I})$ is the objective value of the online solution returned by $\mathcal{A}$ for input $\mathcal{I}$,*
- *$f^{\mathrm{OPT}}(\mathcal{I})$ is the objective value of an optimal offline solution for input $\mathcal{I}$, and*
- *$\rho \geq 1$.*

We say that ρ is the competitive ratio *of $\mathcal{A}$. This definition may also be adapted to maximization problems.*

Remark that a ρ-competitive algorithm is necessarily a ρ-approximation algorithm (although the converse is not necessarily true). The concept of tightness exposed in Definition 1.2 may also be adapted to competitive analysis.

Objective γ	Job charac. β	Env. $\alpha = 1 \longrightarrow$	$\alpha = P \longrightarrow$	$\alpha = Q \longrightarrow$	$\alpha = R$
$\sum w_j F_j$	$\mathcal{M}_j$	?	+	+	+
	$\bullet$	?	**NP**-hard [25]	+	+
	$\mathcal{M}_j, r_j, p_j = 1$	-	p. solvable [23]	?	?
	$r_j, p_j = p$	-	p. solvable [15, 24]	?	$\emptyset$
	$r_j, pmtn$	**NP**-hard [65]	+	+	+
$\sum F_j$	r_j	**NP**-hard [71]	+	+	+
	$\mathcal{M}_j$	-	-	-	p. solvable [25]
	$\bullet$	-	-	-	-
	$\mathcal{M}_j, r_j, p_j = 1$	-	-	?	?
	$r_j, p_j = p$	-	p. solvable [92]	?	$\emptyset$
	$r_j, pmtn$	p. solvable [8]	**NP**-hard [16]	+	+
	$\mathcal{M}_j, pmtn$	-	p. solvable [23]	?	**NP**-hard [93]
	$r_j, p_j = p, pmtn$	-	-	p. solvable [64]	$\emptyset$
$\sum S_j$	r_j	**NP**-hard [70]	+	+	+

Table 1.2 – Complexity of average flow minimization problems. Arrows are polynomial reduction relationships ($A \to B$ means that A is a special case of B). A sign $\bullet$ indicates no particular job characteristics. A sign **+** (resp. **-**) means that the problem is **NP**-hard (resp. polynomially solvable) via the reduction relationship. Incompatible problem designations are noted $\emptyset$.

Graham's bound for list-scheduling algorithms may be extended to the online setting of the Makespan problem, as an arbitrary list-scheduling algorithm treats jobs one by one without any particular priorization rule. Thus, the competitive ratio of any list-scheduling algorithm is at most $2 - \frac{1}{m}$ for the Makespan problem in the online-over-list model, and this ratio is tight [50]. On the contrary, LPT is clearly not a valid online algorithm, as it requires sorting jobs by their processing times. This implies the necessity to know the entire instance in advance.

When studying online scheduling problems, another natural question that arises is the intrinsic limit that the online constraint aspect imposes on the performance of online algorithms. Formalizing this idea leads to the notion of *lower bound* on the competitive ratio that may be achieved for a given problem.

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

$$f^{\mathcal{A}}(\mathcal{I}) \geq B \cdot f^{\text{OPT}}(\mathcal{I}).$$

We say that B is a lower bound *on the competitive ratio for P. If there exists a B-competitive online algorithm $\mathcal{A}$ for the problem P, then B is said to be a* tight *lower bound, and $\mathcal{A}$ is said to be* optimal.

For instance, several lower bounds have been established for the online Makespan problem, e.g., Faigle et al. [44] showed that no deterministic online algorithm can achieve a competitive ratio better than $3/2$ on 2 machines, $5/3$ for 3 machines, and $1 + 1/\sqrt{2}$ for $m \geq 4$ machines. In this sense, Graham's list-scheduling is optimal for the Makespan problem on 2 and 3 machines.

1.4 Related Work on Scheduling

Given the amount of prior work done on scheduling problems, we treat only the subset of the literature that is most relevant to our work. As we mainly focus on response time of key-value stores throughout the manuscript, the scheduling problems we are interested in are those that minimize the flow time of jobs. In the following, we review the work on the minimization of average (weighted) flow time, which has been extensively studied in both offline and online settings (Section 1.4.1). Then, we treat the minimization of maximum (weighted) flow time, which seems to be a more difficult problem (Section 1.4.2). The last section is dedicated to scheduling problems with processing set restrictions, as this constraint is of particular importance to model replication in key-value stores (Section 1.4.3). We provide a summary of the main results of the literature review in Tables 1.2 to 1.5.

1.4.1 Minimization of Average Flow Time

The flow time F_j of a job j is defined as the difference between its completion time C_j in a given schedule and its release time r_j. This corresponds to the time spent by a job in the system, and may be seen as the response time of a request in the context of key-value stores. Thus, optimizing the mean response time of requests can be modeled as the minimization of their average flow time $\sum F_j$. This objective function has been well-studied since the first formalization of scheduling theory principles. In most settings, minimizing the average flow time is in fact equivalent to minimizing the average completion time, as $\sum_j F_j = \sum_j C_j - r_j = \sum_j C_j - \sum_j r_j$, and $\sum_j r_j$ is a constant of the input instance of the considered problem.

Minimizing the average flow time on a single machine (i.e., $1 \,|\, r_j \,|\, \sum F_j$) is already $\mathbf{NP}$-hard, as demonstrated early by Lenstra et al. [71] using a reduction from the 3-Partition problem. The problem becomes easier when job preemption is allowed, as the well-known Shortest Remaining Processing Time first strategy (SRPT) is optimal in this case [8]. The preemptive parallel case remains $\mathbf{NP}$-hard [16]. Another way to make the problem tractable is to consider fixed processing times. For instance, Simons [92] presents an optimal algorithm for $P \,|\, r_j, p_j = p \,|\, \sum C_j$ that runs in time $O(n^3 \log\log n)$, where n is the total number of jobs, and Baptiste [15] and Brucker et al. [24] extend the result to the weighted case. Another notable result is the linear formulation given by Kravchenko et al. [64] for the preemptive problem on uniform machines.

Obviously, given the difficulty of the problem with arbitrary processing times, minimizing the average flow time has been approached with approximation algorithms and competitive analysis. Thus, the single-machine non-preemptive problem has been shown to admit an $O(\sqrt{n})$-approximation algorithm by resolving preemptions from a preemptive solution given by SRPT [61]. The authors also demonstrate that no polynomial-time algorithm can approximate the problem within $\Omega(n^{1/2-\varepsilon})$ for any $\varepsilon > 0$, unless $\mathbf{P} = \mathbf{NP}$, by using a reduction from the 3-Dimensional Matching problem. In the online setting, the First-Come First-Served strategy (FCFS), which consists in treating jobs in order of arrival, is Δ-competitive [70], with Δ being the ratio between the maximum processing time and the minimum processing time of the jobs (i.e., $\Delta = \frac{\max_j p_j}{\min_j p_j}$). Becchetti et al. [17] also give an $O(\log n)$-competitive algorithm for the preemptive problem. Similar results have been proven in the parallel case. Leonardi et al. [73] give an $O\left(\sqrt{\frac{n}{m}} \log \frac{n}{m}\right)$-approximation algorithm, where m is the total number of machines, although they also show a lower bound of $\Omega(n^{1/3-\varepsilon})$ on the approximability of the problem. Moreover, with preemptions allowed, the authors prove that SRPT is $O\left(\log(\min(\frac{n}{m}, \Delta))\right)$-competitive, and they derive two lower bounds ($\Omega\left(\log \frac{n}{m}\right)$ and $\Omega\left(\log \Delta\right)$) on the competitive ratio of any randomized online algorithms. Interestingly, a similar competitive ratio can be achieved when allowing local preemptions but not job migrations, that is to say, any interrupted job must be resumed on the same machine it was

Objective γ	Env. α	Job charac. β	Results	Ref.
$\sum w_j F_j$	1	$r_j, pmtn$	$O\left(\log^2 \Delta\right)$-competitive algorithm	[32]
			Lower bound in 1.618 (deterministic)	[32]
			Lower bound in $4/3$ (randomized)	[32]
			$O\left(\log \frac{\max w_j}{\min w_j}\right)$-competitive algorithm	[13]
			$O\left(\log n + \log \Delta\right)$-approximation algorithm	[13]
	P	$r_j, pmtn$	Lower bound in $\Omega\left(\min\left\{\sqrt{\Delta}, \sqrt{\frac{\max w_j}{\min w_j}}, \sqrt[4]{\frac{n}{m}}\right\}\right)$ (randomized)	[32]
$\sum F_j$	1	r_j	Δ-competitive algorithm	[70]
			Non-approximable within $\Omega(n^{1/2-\varepsilon})$	[61]
			$O\left(\sqrt{n}\right)$-approximation algorithm	[61]
		$r_j, pmtn$	Optimal algorithm	[8]
			$O\left(\log n\right)$-competitive algorithm	[17]
	P	r_j	Non-approximable within $\Omega(n^{1/3-\varepsilon})$	[73]
			$O\left(\sqrt{\frac{n}{m}} \log \frac{n}{m}\right)$-approximation algorithm	[73]
		$r_j, pmtn^*$	$O\left(\min(\log \Delta, \log n)\right)$-competitive algorithm	[6]
		$r_j, pmtn$	$O\left(\log(\min(\frac{n}{m}, \Delta))\right)$-competitive algorithm	[73]
			Lower bound in $\Omega\left(\log \frac{n}{m}\right)$ (randomized)	[73]
			Lower bound in $\Omega\left(\log \Delta\right)$ (randomized)	[73]
			$O\left(\log n \min(\log \Delta, \log \frac{n}{m})\right)$-competitive algorithm	[17]
	$P \mid \mathcal{M}_j$	$r_j, pmtn^*$	$O\left(\log \Delta\right)$-approximation algorithm	[48]
			No bounded competitive ratio	[48]
$\sum S_j$	1	r_j	Δ^2-competitive algorithm	[70]
		$r_j, pmtn$	2-competitive algorithm	[81]
	P	$r_j, pmtn^*$	17.32-competitive algorithm	[32]
		$r_j, pmtn$	14-competitive algorithm	[81]
			9.82-competitive algorithm	[32]

Table 1.3 – Various results on average (weighted) flow minimization. $P \mid \mathcal{M}_j$ denotes parallel machines with processing set restrictions, which is a particular case of unrelated machine environment, i.e., we have $P \rightarrow Q \rightarrow R$ and $P \rightarrow P \mid \mathcal{M}_j \rightarrow R$. $pmtn^*$ denotes non-migratory preemption. n is the number of jobs and m is the number of machines. Δ denotes the ratio between the largest and the smallest processing time.

started on [6].

For the generalized weighted case, even the preemptive problem is $\mathbf{NP}$-hard on a single-machine [65]. Still, several authors propose approximations. For instance, Chekuri et al. [32] present an $O\left(\log^2 \Delta\right)$-competitive algorithm in the semi-online setting (i.e., their algorithm needs to know the ratio Δ at the beginning), and a $(2+\varepsilon)$-approximation that runs in quasi-polynomial time. They also give lower bounds on the competitive ratio of deterministic and randomized online algorithms (1.618 and $4/3$, respectively). Bansal et al. [13] propose an $O\left(\log \frac{\max w_j}{\min w_j}\right)$-competitive algorithm and an $O\left(\log n + \log \Delta\right)$-approximation.

In addition to $\sum F_j$, another special case of the general weighted problem has received a significant attention: the average stretch $\sum S_j$, the stretch S_j of a job j, also called the *slowdown*, being defined as its flow time divided by its processing time (i.e., $S_j = \frac{C_j - r_j}{p_j}$). The single-machine problem remains $\mathbf{NP}$-hard, although the preemptive case becomes easier to approximate. For instance, SRPT is 2-competitive on a single machine, and 14-competitive in the parallel case [81]. Chekuri et al. [32] improve the result with a 9.82-competitive algorithm, as well as a 17.32-competitive algorithm when local preemptions are allowed but not migrations. In the non-preemptive setting, FCFS has been shown to be Δ^2-competitive [70].

1.4.2 Minimization of Maximum Flow Time

The second objective related to response time optimization is the minimization of the maximum flow time $F_{\max}$, as introduced by Bender et al. [19]. According to the authors, this criterion may be more appropriate than the average flow time in certain cases, as the latter is subject to starvation (minimizing the average flow time may lead to execute small jobs in priority, delaying large jobs indefinitely). They prove that FCFS is $(3 - 2/m)$-competitive for the non-preemptive parallel problem (i.e., $P \,|\, r_j \,|\, F_{\max}$), and as a corollary, it is optimal on a single machine. Mastrolilli [80] prove that FCFS achieves the same competitive ratio in the preemptive case. Ambühl et al. [2] give a $(2-1/m)$-competitive algorithm, which matches the lower bound and makes it the best possible online algorithm for the preemptive problem. On uniform machines, Bansal et al. [12] provide a 13.5-competitive algorithm, and two PTAS are derived by Bansal [11] and Mastrolilli [80].

In addition to the maximum flow time, Bender et al. [19] also introduced the maximum stretch $S_{\max}$, and proved that no polynomial-time algorithm can approximate $1 \,|\, r_j \,|\, S_{\max}$ within a factor $\Omega(n^{1-\varepsilon})$ for any $\varepsilon > 0$, unless $\mathbf{P} = \mathbf{NP}$. They also exhibit a FPTAS for the preemptive case, and they derive an

Objective γ	Job charac. β	Env. $\alpha = 1 \longrightarrow$	$\alpha = P \longrightarrow$	$\alpha = Q \longrightarrow$	$\alpha = R$
$\max w_j F_j$	r_j	+	+	+	+
	$r_j, pmtn$	-	-	-	p. solvable [70]
	•	p. solvable [52]	+	+	+
$F_{\max}$	r_j	p. solvable [19]	+	+	+
$S_{\max}$	r_j	$\mathbf{NP}$-hard [19]	+	+	+
	•	-	$\mathbf{NP}$-hard [20]	+	+

Table 1.4 – Complexity of maximum flow minimization problems. Arrows are polynomial reduction relationships ($A \rightarrow B$ means that A is a special case of B). A sign • indicates no particular job characteristics. A sign **+** (resp. **-**) means that the problem is $\mathbf{NP}$-hard (resp. polynomially solvable) via the reduction relationship. Incompatible problem designations are noted $\emptyset$.

Objective γ	Env. α	Job charac. β	Results	Ref.
$\max w_j F_j$	P	r_j	Lower bound in $\Omega\left(\frac{\max w_j}{\min w_j}\right)$	[12]
	R	r_j, *pmtn*	Optimal algorithm	[67, 65, 70]
$F_{\max}$	P	r_j	$(3 - 2/m)$-competitive algorithm	[19]
			Lower bound in $2 - 1/m$	[2]
		r_j, *pmtn*	$(3 - 2/m)$-competitive algorithm	[80]
			$(2 - 1/m)$-competitive algorithm	[2]
			Lower bound in $2 - 1/m$	[2]
	$P \mid \mathcal{M}_j$	r_j	Lower bound in $\Omega(m)$	[3]
	Q	r_j	13.5-competitive algorithm	[12]
	R	r_j	$O(\log n)$-approximation algorithm	[14]
			PTAS in $n^{O(m/\varepsilon)}$	[11]
			FPTAS in $O\left(nm(n^2/\varepsilon)^m\right)$	[80]
$S_{\max}$	1	r_j	Δ-competitive algorithm	[70]
			$(\frac{1}{2}\Delta(\sqrt{5} - 1) + 1)$-competitive algorithm	[40]
			Lower bound in $\frac{1}{2}\Delta(\sqrt{5} - 1) + 1$	[40]
			Non-approximable within $\Omega\left(n^{1-\varepsilon}\right)$	[19]
		r_j, *pmtn*	$O\left(\sqrt{\Delta}\right)$-competitive algorithm	[19]
			PTAS	[19]
	P	r_j	$(2\Delta + 1)$-competitive algorithm	[90]
			Lower bound in $\Omega(\Delta)$	[12]

Table 1.5 – Various results on maximum (weighted) flow minimization. $P \mid \mathcal{M}_j$ denotes parallel machines with processing set restrictions, which is a particular case of unrelated machine environment, i.e., we have $P \to Q \to R$ and $P \to P \mid \mathcal{M}_j \to R$. n is the number of jobs and m is the number of machines. Δ denotes the ratio between the largest and the smallest processing time.

$O\left(\sqrt{\Delta}\right)$-competitive algorithm from the Earliest Deadline First strategy (EDF), which schedules jobs by increasing order of deadline. FCFS is also shown to be Δ-competitive on a single-machine, and a lower bound of $\frac{1}{2}\Delta^{\sqrt{2}-1}$ is derived by Legrand et al. [70]. Saule et al. [90] improve this result by showing a lower bound of $\frac{1}{2}(\Delta+1)$ on a single machine and $\frac{1}{2}(\frac{\Delta}{m+1}+1)$ in the parallel case, in addition to a $(2\Delta+1)$-competitive algorithm. Dutot et al. [40] close the online single-machine problem by proving a lower bound of $\frac{1}{2}\Delta(\sqrt{5}-1)+1$, which is tight.

Note that a breakthrough has been made for the preemptive problem by Legrand et al. [70], who provide a polynomial-time algorithm to solve the offline minimization of maximum weighted flow time on unrelated machines. Essentially, this closes any complexity question on the minimization of maximum flow time when preemptions are allowed.

1.4.3 Scheduling with Processing Set Restrictions

We complete our review on scheduling with problems involving processing set restrictions. An important consequence of replication in key-value stores is that a given request must be processed by a machine that holds the requested key. In scheduling literature, the terminology is not yet fixed for this kind of constraint, which is known as restricted assignment, multipurpose machines, processing set restrictions or even eligibility constraints. In this thesis, we stick with *processing set restrictions*, and we note $\mathcal{M}_j$ the corresponding constraint in Graham's notation.

The great majority of problems involving such constraints focus on makespan minimization in various settings, and they have been heavily surveyed [75, 69, 74]. In fact, the problem $P\,|\,\mathcal{M}_j\,|\,C_{\max}$ is also known as the Restricted Assignment problem, which has received a significant attention, as it captures the essence of many practical problems. It is a subcase of the more general Unrelated problem ($R\,||\,C_{\max}$), for which a famous 2-approximation algorithm, based on linear programming, has been proposed by Lenstra et al. [72]. The authors also considered the Restricted Assignment problem and proved that no polynomial algorithm may give an approximation better than $3/2$, unless $\mathbf{P}=\mathbf{NP}$. A PTAS has recently been derived for this problem, which approximates an optimal solution within a factor $11/6+\varepsilon$ in time $O\left((n+m)^{O(1/\varepsilon\log(n+m))}\right)$ [56]. Various subcases of the Restricted Assignment problem have also been considered in the literature. For instance, Ebenlendr et al. [42] studied the Graph Balancing problem, which corresponds to the Restricted Assignment problem where each job may be processed by at most two different machines. They show the $3/2$-hardness of the problem, and provide a $7/4$-approximation algorithm. List-scheduling, which is a famous $(2-1/m)$-approximation for the problem $P\,||\,C_{\max}$, has also been proved to give the same guarantee at the conditions that the processing sets form a laminar set family and jobs are initially sorted by non-decreasing size of their processing set [49]. On the negative side, Maack et al. [78] proved that no PTAS exists when processing sets consist in contiguous intervals of machines unless $\mathbf{P}=\mathbf{NP}$. Interestingly, there is a PTAS when all intervals are overlapping without any strict inclusion, i.e., for any two jobs j, j', $\mathcal{M}_j \not\subset \mathcal{M}_{j'}$ and $\mathcal{M}_{j'} \not\subset \mathcal{M}_j$ [100]. There is also a PTAS for other variants [43].

Some authors also studied the Restricted Assignment problem with limited heterogeneity in processing times. Jansen et al. [57] proved that even when considering only two possible processing times, there is no algorithm giving an approximation better than $4/3$. However, in the specific case where $p_j \in \{1,2\}$, there is a $3/2$-approximation algorithm [49]. By approaching the Restricted Assignment problem as a matching problem, Biró et al. [21] derive a $(2-1/2^k)$-approximation when $p_j \in \left\{1,2,\cdots,2^k\right\}$ for all jobs. When jobs are unitary, the Restricted Assignment problem becomes simpler, and it is possible to find optimal schedules in time $O\left(n^3\log n\right)$ by coupling a binary search procedure to a network flow formulation of the problem [76].

Furthermore, some results have been derived for other objective functions. Anand et al. [3] give a lower bound of $\Omega\left(m\right)$ on the competitive ratio of any online algorithm trying to minimize the maximum flow time when general processing set restrictions are considered. Brucker et al. [23] used a routine based on the minimum cost matching problem to solve the problems $Q\,|\,\mathcal{M}_j, p_j\ =\ 1\,|\sum w_jU_j$ and $P\,|\,\mathcal{M}_j, r_j, p_j = 1\,|\sum w_jU_j$ in polynomial time.

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 request scheduling in distributed key-value stores, which we begin to explore as a latency-minimization problem. The optimization of the average response time is well-understood, in both stochastic and deterministic settings [88], and key-value stores show excellent empirical results for this specific metric [62]. However, key-value stores are well-known to suffer from the tail latency problem, where a small fraction of slow requests affect the majority of the workload [34]. We choose to model the problem as an optimization problem where the objective is to minimize the maximum response time among requests (Section 2.2). An equivalent formulation consists in computing a schedule where each request is answered within a minimal time span. Our 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 exactly make the problem intractable (Section 2.3). 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 (Section 2.4). Finally, we explore request scheduling from a more empirical approach in Section 2.5, by simulating a distributed key-value store and comparing several heuristics on the basis of this common lower bound.

2.2 Modeling Key-Value Stores

We propose a formal scheduling framework for key-value stores. This general model is common to all chapters, with some slight variations. 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.

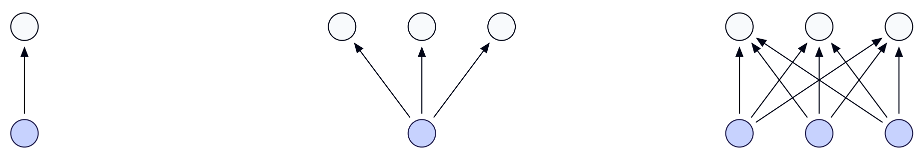
Scheduler model. A *scheduler* is a component that assigns jobs to machines. In *distributed* key-value stores, the cluster is composed of machines that receive and treat client requests. These requests constitute the jobs of the system and are assigned to machines by schedulers. Thus, if we denote the 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, because key-value stores are generally leaderless distributed systems where each machine is able to receive and schedule requests in the cluster (these are the coordinators, as previously explained in Chapter 1). In other words, in distributed key-value stores, each machine is also a scheduler, and we have $c = m$. Nonetheless, this model is 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 costs of communication and synchronization between schedulers. Given the difficulty of the last model, and considering the natural complexity of key-value stores, we choose in this thesis 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



(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 (bottom nodes) represent schedulers, and gray circles (top nodes) 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 particular case of the second model, which is itself a particular case of the third model.

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. That being said, we will also consider the first model as a particular case of the second model in some theorems, and the last model 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 responding with the corresponding value to the initiator of the request, whereas processing a *write* request (e.g., `INSERT` or `DELETE` operations) consists in modifying the stored value corresponding to the key. Note that persistent key-value stores are usually extremely efficient at processing write 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 replica holding the corresponding value, the scheduler must choose where the request should be handled (if the consistency level, as defined in Chapter 1, is set to a value that is lower than the replication factor, which is almost always the case). This enables the system to balance the load between machines. On the other hand, write requests must be processed by all replicas 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. Indeed, read-dominated workloads are the most common in practice [5], and read requests are the most critical to optimize for mitigating the tail latency problem in such situations. Moreover, we consider that the consistency level is set to 1, i.e., each read request is treated by a single machine. This is a common choice in practice, as it enables the system to achieve the best performance for read-dominated workloads.

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 replica holding the value corresponding to the specified key. Moreover, there is no notion of precedence relation. Following Graham's notation, for each job j, we denote its processing time by p_j, and we note $\mathcal{M}_j$ the subset of machines storing the requested value. This subset $\mathcal{M}_j$ is called the *processing set* of j.

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 time* 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 [54]).

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. For a continuous stream of requests, this can be expressed as the total number of processors times the inverse of the average response time of a request, which means that maximizing the average throughput may be seen as minimizing the average response time.

- The tail latency, corresponding to the last percentiles in the distribution of response times of the requests, which should be minimized. Different percentiles are considered among authors in the literature, the most frequent being the 90th, 95th, 99th and 99.9th percentiles.

Minimizing the average response time has been the subject of a huge amount of work, both in the context of key-value stores and scheduling theory. Several techniques have been proposed to analyze this metric, from the complexity analysis of related scheduling problems [88, 23, 24, 64] to the design of approximations and competitive online algorithms [19, 81, 61, 70, 73, 32, 48].

On the contrary, few papers have focused on tail latency minimization from the point of view of scheduling theory in the context of key-value stores. Expressing this criterion is not straightforward in deterministic scheduling, and we propose the following approach. If we take a step back, our goal 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. 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 the user's tolerance for the response time of a service is higher when a process considered to be heavy is in progress (downloading a video, in contrast with sending a text message, for example), 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 to $\max_j \{w_j F_j\}$, and simply noted $\max w_j F_j$. 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 as they become more sensitive to scheduling decisions.

2.3 Scheduling Problem and Relaxed Variants

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:

- Jobs are heterogeneous, i.e., each job j has an arbitrary processing time p_j.

- Jobs have processing set restrictions, i.e., each job j can be processed by a subset of machines $\mathcal{M}_j$ only (with $\mathcal{M}_j \subseteq M$).

- 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 migrated nor preempted.

- There are no simultaneous executions, i.e., two different jobs cannot be executed at the same time on the same machine.

Problem	Relaxation	Class	Approximation	Ref.
$R \,\vert\vert\, C_{\max}$	w_j, r_j	**NP**-hard	2	Lenstra et al. [72]
$1 \,\vert\vert\, \max w_j C_j$	$\mathcal{M}_j, r_j$	**P**		Theorem 2.1
$P \,\vert\vert\, \max w_j C_j$	$\mathcal{M}_j, r_j$	**NP**-hard	$2 - 1/m$	Theorem 2.3
$Q \,\vert\, p_j = p \,\vert\, \max w_j C_j$	$\mathcal{M}_j, r_j, p_j$	**P**		Theorem 2.2
$1 \,\vert\, r_j \,\vert\, F_{\max}$	$\mathcal{M}_j, w_j$	**P**		Bender et al. [19]
$P \,\vert\, r_j, p_j = p \,\vert\, F_{\max}$	$\mathcal{M}_j, w_j, p_j$	**P**		Theorem 2.4
$P \,\vert\, r_j \,\vert\, F_{\max}$	$\mathcal{M}_j, w_j$	**NP**-hard	$3 - 2/m$	Bender et al. [19]

Table 2.1 – Computational complexity of relaxed variants.

We note the problem $P \mid \mathcal{M}_j, \mathit{online}\text{-}r_j \mid \max w_j F_j$ in Graham's classification (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 π 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.

The offline problem is trivially **NP**-hard by reduction to the $P \,\vert\vert\, C_{\max}$ problem, which is itself **NP**-hard (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 $P \,\vert\vert\, C_{\max}$, can be solved). Thus, finding an optimal solution for any instance is unlikely to be possible in polynomial time. In the following, we 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.

Algorithm 1 MaxWeight

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

We can show by a swapping argument that MaxWeight is optimal on a single server.

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

Proof: Let π^{OPT} be an optimal schedule for a given instance of the problem. We may consider without loss of generality that π^{OPT} is compact, i.e., there is no idle time between jobs. There are two possibilities: either all jobs are ordered by non-increasing weight in π^{OPT}, which corresponds to MaxWeight, or we can find two consecutive jobs j and k in π^{OPT} such that $w_j \leq w_k$.

In the second case, the contribution of j and k 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 them, 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., $\mathcal{C}' \leq \mathcal{C}$. Note that swapping j and k does not change the completion time of any other job, because j and k were consecutive in π^{OPT}.

Hence, if two consecutive jobs are not ordered by non-increasing weight in π^{OPT}, we can switch them without increasing the objective $\max w_j C_j$. By repeating the operation job by job, we transform π^{OPT} in another optimal schedule where jobs are sorted by non-increasing w_j, which corresponds to MaxWeight. The conclusion follows. ■

By extending the previous proof, we show that MaxWeight also solves the uniform case when all jobs have the same processing time p.

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

Proof: For a given instance of the problem, let π^{OPT} be an optimal, compact schedule with two jobs j and k such that $w_j < w_k$ and where k completes at time $C_k = C_j + c$ with $c > 0$ (possibly on a different 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, $\mathcal{C}' < \mathcal{C}$. Note that swapping j and k does not change the completion time of any other job, because all jobs have the same processing time p.

Hence, we can transform π^{OPT} in another optimal schedule by swapping repeatedly non-sorted jobs. 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 $c > 0$). ■

Unfortunately, the result does not extend to the parallel case with arbitrary processing times. Nevertheless, MaxWeight still gives a guaranteed approximation of an optimal solution.

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

Proof: Although the bound has been established by Hall [52], we give a simpler proof here, and we also demonstrate that the approximation is tight. Let us consider a MaxWeight schedule π and an optimal schedule π^{OPT}.

Let u be the job for which $w_u C_u = \max_j \{w_j C_j\}$, i.e., the job that reaches the objective in π. We remove from π and π^{OPT} all jobs j such that $w_j < w_u$, which does not change the objective value in π, because those jobs are all scheduled to start after the job u and removing them does not change the schedule of the remaining jobs. Note that u is necessarily the last job to complete in π, as all jobs starting before u have a higher weight and u reaches the maximum weighted makespan. Moreover, removing jobs can only decrease the objective value 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 $C_{\max} \leq \left(2 - \frac{1}{m}\right) C^*_{\max}$ [50], where $C_{\max}$ is the completion time of the last job in π, i.e., $C_{\max} = C_u$. Let k be the last completed job in π^{OPT}, such that $C_k^{\text{OPT}} = C_{\max}^{\text{OPT}}$. This makespan is bounded by the optimal one for the partial schedule (i.e., $C^*_{\max} \leq C_k^{\text{OPT}}$). Therefore,

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

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 $\max w_j C_j \leq (2 - 1/m) \max w_j C_j^{\text{OPT}}$.

We prove that it is asymptotically tight by considering the class of instances, parameterized by an arbitrary positive value W, with m machines and $n = m(m-1)+1$ jobs, whose weights and processing times are as follows:

- $w_j = W + 1, p_j = 1$ 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 $\max w_j C_j^{\text{OPT}} = m(W+1)$. The approximation ratio $(2 - 1/m)W/(W+1)$ tends to $2 - 1/m$ as $W \to \infty$, and the conclusion follows. ■

It is also known that $P \mid r_j, p_j = p \mid F_{\max}$ can be solved in polynomial time [92], although the proposed algorithm is quite complex, as it solves a more general problem. We show that processing jobs in their order of arrival (i.e., the First Come First Served policy) is sufficient to solve $P \mid r_j, p_j = p \mid F_{\max}$.

Theorem 2.4. *First Come First Served (FCFS) solves $P \mid r_j, p_j = p \mid F_{\max}$ in polynomial time.*

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 FCFS policy: two jobs starting simultaneously on two machines may be allocated on different machines by OPT and FCFS, but it does modify neither their completion time nor the completion times of other jobs, because all jobs have the same processing time p. Otherwise, let j and k be two jobs in π^{OPT} such that $r_j \leq r_k$, and where j starts after k (i.e., $\sigma_j \geq \sigma_k$). The job j can be on any machine, as well as the job k. Thus, $\sigma_j + p \geq \sigma_k + p$, and then $C_j \geq C_k$, as $p_j = p_k = p$.

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, which 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, $\mathcal{F}' \leq \mathcal{F}$. Note that swapping j and k does not modify the flow time of any other job, because all jobs have the same processing time p.

Hence, we can transform π^{OPT} in another optimal schedule by swapping repeatedly non-sorted jobs, and the conclusion follows. ■

2.4 Bounding the Objective of Each Instance

The general scheduling problem we are interested in, with heterogeneous processing times, processing sets, release times and no preemption ($P \mid \mathcal{M}_j, r_j \mid \max w_j F_j$) 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 sub-problem that captures the whole set of solutions of the offline problem. More formally, let $\mathcal{P}$ be the general and difficult scheduling problem $P \mid \mathcal{M}_j, r_j \mid \max w_j F_j$, and let $\Pi(\mathcal{I})$ be the set of valid solutions for a given instance $\mathcal{I}$. We want to find a tractable scheduling problem $\mathcal{Q}$ such that $\Pi(\mathcal{I}) \subseteq \Pi(\phi_{\mathcal{P}\to\mathcal{Q}}(\mathcal{I}))$ for any instance $\mathcal{I}$ of the problem $\mathcal{P}$, where $\phi_{\mathcal{P}\to\mathcal{Q}}$ is a polynomial-time procedure that transforms any instance of the problem $\mathcal{P}$ into an instance of the relaxed problem $\mathcal{Q}$. Then, if we note $f(\pi)$ the objective value of a solution π, we necessarily have

$$\mathcal{L}(\mathcal{I}) \leq \min_{\pi\in\Pi(\mathcal{I})} \{f(\pi)\}$$

for any instance $\mathcal{I}$ of the problem $\mathcal{P}$, where $\mathcal{L}(\mathcal{I}) = \min_{\pi\in\Pi(\phi_{\mathcal{P}\to\mathcal{Q}}(\mathcal{I}))} \{f(\pi)\}$ 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 $\mathcal{Q}$ 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 $\mathcal{Q}$ so that the difference $\min_{\pi\in\Pi(\mathcal{I})} \{f(\pi)\} - \mathcal{L}(\mathcal{I})$ is not too large.

Legrand et al. [70] solved the scheduling problem $R \,|\, r_j, pmtn \,|\, \max w_j F_j$ in polynomial time by expressing the model as a linear program. This problem is very similar to the one we are interested in, as the platform relies on unrelated machines, which, by definition, generalizes our multipurpose machines environment ($P \,|\, \mathcal{M}_j \,|\, \max w_j F_j$ is a special case of $R \,||\, \max w_j F_j$) [75]. 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: it is solvable in polynomial time through the procedure given by Legrand et al. [70], the identity function is a trivial transformation procedure between instances of the two problems, and 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 systematically resumed on the same machine it was initially assigned on.*

However, we show that the non-migratory variant of the preemptive problem is $\mathbf{NP}$-complete. The proof of this result consists in a reduction from the MAKESPAN problem, which is itself $\mathbf{NP}$-complete [71].

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* $\mathbf{NP}$*-complete.*

Proof: We prove the $\mathbf{NP}$-completeness of this problem by reduction from $P \,||\, C_{\max}$, which is well-known to be $\mathbf{NP}$-complete [71]. Obviously, NONMIGRATORY belongs to $\mathbf{NP}$, as any solution is verifiable in polynomial time. Let us now prove that NONMIGRATORY is $\mathbf{NP}$-hard.

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 $R\,|\,r_j, pmtn\,|\,\max w_jF_j$ given by Legrand et al. [70] consists in performing a binary search on a linear program, followed by the schedule reconstruction scheme given by Lawler et al. [67]. 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 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 $E(f) = \{r_1, \cdots, r_n\} \cup \{d_1(f), \cdots, d_n(f)\}$. 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.

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 $f = \frac{w_jw_{j'}}{w_{j'}-w_j}(r_{j'} - r_j)$, 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 corresponding feasible preemptive schedule. If this is the case, we want to find such value f that is minimum. 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 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)).

$$\begin{aligned}
\textbf{minimize}\quad & f && \text{(2.1a)}\\
\text{subject to}\quad & \forall j, \sum_{it} x_{ijt} = 1, && \text{(2.1b)}\\
& \forall i,t, \sum_{j} x_{ijt} p_{ij} \leq e_{t+1}(f) - e_t(f), && \text{(2.1c)}\\
& \forall j,t, \sum_{i} x_{ijt} p_{ij} \leq e_{t+1}(f) - e_t(f), && \text{(2.1d)}\\
& \forall i,j,t, x_{ijt} = 0 \text{ if } t_{r_j}(f) > t, && \text{(2.1e)}\\
& \forall i,j,t, x_{ijt} = 0 \text{ if } t_{d_j}(f) \leq t, && \text{(2.1f)}\\
& \lambda_1 \leq f \leq \lambda_2 && \text{(2.1g)}
\end{aligned}$$

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 given by Linear Program 2.1.

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 duration of execution for job j 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 the current iteration.

We now compute the duration 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. The duration 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

Given the difficulty of the general problem $P \mid \mathcal{M}_j, r_j \mid \max w_j F_j$, it seems difficult to derive guaranteed strategies that are usable in practice. Nonetheless, we are now equipped with a tractable lower bound for any instance, which unlocks the possibility to quantitatively estimate the quality of a scheduling heuristic. In this section, we switch to a more empirical approach. 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 scheduling heuristics, both coming from the literature and original work (Section 2.5.2), and we compare them with each other in simulations on the basis of the previously-described lower bound (Section 2.5.3).

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:

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

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

These components constitute 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 (this is called the *replica selection* step). Finally, the replica dequeues local operations and eventually executes the request (this is called the *local execution* step).

Moreover, we consider from now on that each processing set $\mathcal{M}_j$ follows the typical replication strategy of key-value store implementations as described in Chapter 1, that is, the cluster is a circular, ordered set of machines, and each set $\mathcal{M}_j$ can be seen as an interval of k machines. In other words, $\mathcal{M}_j = \{1 + (i - 1 + x) \bmod m\}_{0 \leq x < k}$ for all jobs j, where i is the first machine storing the value that is requested by j.

Finally, 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, the information about the load of a given machine will often be slightly out of date, as propagation of data through the network takes time. Similarly, the processing times of jobs are not exact, as the size of the requested value cannot be always known by the coordinator for large datasets. Practical systems generally employ an approximation, e.g., by categorizing values according to their size (*small* and *large* values) through the use of Bloom filters [54]. 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.

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 [97].

Héron. We also consider an omniscient version of the heuristic used by Héron [54]. 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 [97]. The threshold is chosen according to the wanted proportion of large requests in the workload.

Earliest Finish Time (EFT). Let $\textsc{Avail}(i)$ denote the earliest time when 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 $\textsc{Avail}$ 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 and split them into "large" and "small" 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 the size-aware sharding technique [37]). 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, as mentioned in the previous section), 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 Bloom filters, in a similar fashion to Héron [54].

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 [79] inspired this strategy. Let $\mathcal{F}$ be the function giving the estimated weighted flow $\mathcal{F}(i,j) = w_j(\max(r_j, \textsc{Avail}(i)) + p_j - r_j)$ 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

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

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: $s = \arg\max_{j \in Q} \textsc{Suf}(j)$. The chosen replica is $\rho(s)$: $r = \rho(s) = \arg\min_{i \in \mathcal{M}_j} \mathcal{F}(i, s)$.

Algorithm 2 SUFFERAGE-SW

```
1:  repeat every $q$ time units
2:      for all $j \in Q$ do
3:          $\rho(j) \leftarrow \arg\min_{i \in \mathcal{M}_j} \mathcal{F}(i, j)$
4:          $\rho'(j) \leftarrow \arg\min_{i \in \mathcal{M}_j \setminus \rho(j)} \mathcal{F}(i, j)$
5:          $\textsc{Suf}(j) \leftarrow \mathcal{F}(\rho'(j), j) - \mathcal{F}(\rho(j), j)$
6:      while $Q$ is not empty do
7:          $s \leftarrow \arg\max_{j \in Q} \textsc{Suf}(j)$
8:          Schedule $s$ on $\rho(s)$
9:          $Q \leftarrow Q \setminus \{s\}$
10:         Update $\rho$, $\rho'$ and $\textsc{Suf}$
```

Algorithm 3 MAXMIN-SW

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

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

MAXMIN-SW (MSW). This strategy is inspired from the Max-Min heuristic [79]. We build a matrix MAT whose rows are machines and columns are requests from Q, where

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

The best weighted flow of request j is $\mathcal{F}_{best}(j) = \mathcal{F}(\rho(j), j) = \min_{i \in M} \textsc{Mat}[i, j]$. 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: $r = \arg\min_{i \in M} \textsc{Mat}[i, s]$.

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

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

Heuristic	Knowledge	Type	Time 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	$O(n^2 m)$
MSW	AVAIL, p_j, r_j	Centralized	$O(n^2 m)$

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. $p_j \geq \omega$ means that the heuristic is able to know the category (small or large) of each request. n is the number of requests in Q and m is the total number of servers.

responsible for a limited number of keys. We consider the following local policies.

FIRST IN FIRST OUT (FIFO). This strategy is commonly used as a local scheduling policy in key-value stores (e.g., Apache Cassandra [66]). 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, because the difference $t - r_j$ will keep increasing over time. Note that when coupled with the stretch metric ($w_j = 1/p_j$), MWF tends to favor 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 sizes characterizations [46]. 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

Heuristic	Knowledge	Time 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 $\mathcal{Q}$.

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 [66, 35]. 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*.

We now describe the results of our simulations, before discussing them in the next paragraph.

Results. Figure 2.2 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 2.3 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.

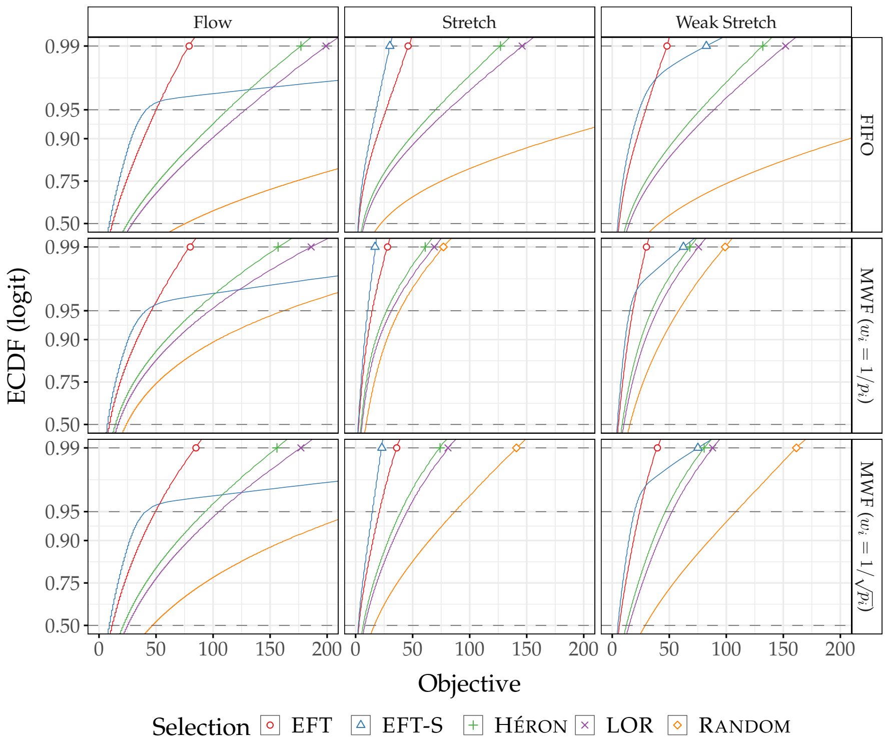


Figure 2.2 – ECDF of flow, stretch and weak stretch metrics given by each combination of distributed selection and execution heuristics in steady-state over 120 seconds, under average load of 90%.

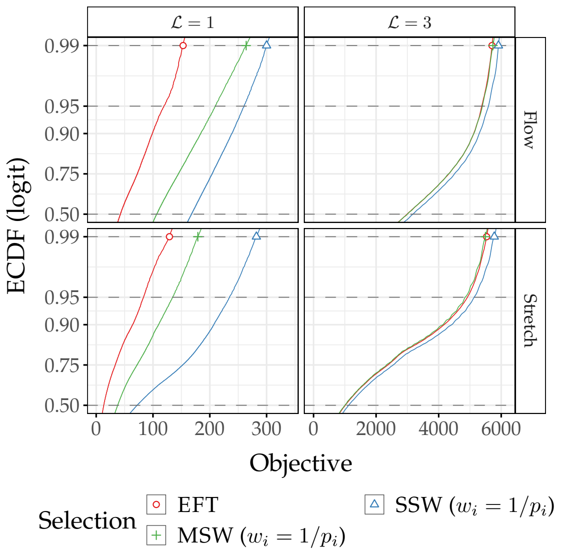


Figure 2.3 – ECDF of flow and stretch metrics given by centralized heuristics SSW and MSW combined to a local FIFO execution in a burst over 3 seconds, under average loads of 100% and 300%.

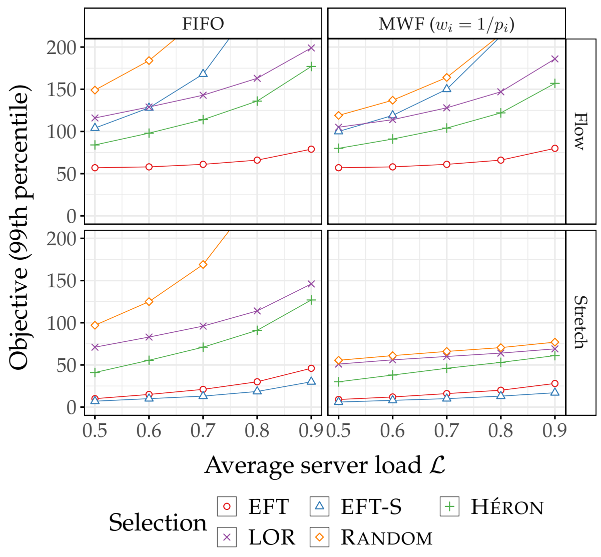


Figure 2.4 – 99th quantile of flow and stretch metrics for each combination of selection/execution heuristics in steady-state over 120 seconds, under average loads ranging from 50% to 90%.

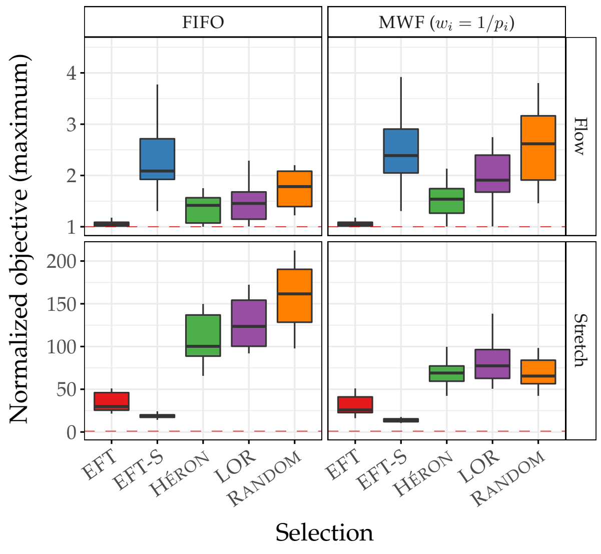


Figure 2.5 – Distributions of normalized flow maximums and stretch maximums for each combination of selection/execution heuristics. Data items are requested with a load of 90%, and the 10 different scenarios consist of 1200 requests.

Figure 2.4 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 weak stretch case $w_j = 1/\sqrt{p_j}$, as it exhibits performance always worse than the stretch case $w_j = 1/p_j$. The simulations run for 120 seconds.

Another comparison of online heuristics is shown in Figure 2.5, where we normalize the objective $\max w_j F_j$ given by each heuristic with the lower bound introduced in Section 2.4. Each boxplot[2] represents the distribution of these normalized maximums among 10 different scenarios, for each combination of strategies. Horizontal red bars help to locate the lower bound. Data items are requested with a load $\mathcal{L} = 0.9$, and the 10 scenarios are solved over 1200 requests.

Discussion. The first thing to note in Figures 2.2 to 2.5 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 2.2. This highlights the fact that some properties of the cluster and the workload are more suitable to taming tail latency; in particular, knowing the current load of a server, and thus its earliest available time, allows implementing the EFT strategy and getting very close to the lower bound (Figure 2.5).

Figure 2.5 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 2.4.

For the stretch metric, where latencies are weighted by processing times, EFT-S performs even better than EFT (Figures 2.2 and 2.4), 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 2.2, 2.4 and 2.5 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 [19]. It is confirmed by our observations, as MWF worsen the tail latency.

When a burst occurs, Figure 2.3 shows the value of our window-based heuristics. Interestingly, these replica selection strategies do not benefit a lot from centralized and global information about the workload, and are not even effective for realistic load values. When the average load exceeds 300% ($\mathcal{L} \geq 3$) we see that ECDF of EFT and SSW or MSW are similar, but the window-based heuristics never outperform EFT. This seems to confirm that EFT is a close-to-optimal strategy in average, as additional information do not allow improving performance.

[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.

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 $\mathbf{NP}$-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 (original ones or coming from prior work and real implementations) that are evaluated through discrete-event simulations and properly compared on the basis of the theoretical lower bound. We confirm that replica selection is a critical step in the scheduling process. In particular, EFT gives excellent results on the tail latency and achieves maximum response times that are very close to the lower bound. Moreover, we highlight the importance of the local execution policy, which can be tuned to further improve performance.

In the next chapter, we study the online version of the problem, where scheduling algorithms discover the instance over time, and have to make decisions without knowing the future. We focus particularly on the impact of the replication strategy on the performance of the system, which we choose to evaluate through competitive analysis.

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 of balancing 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 (corresponding 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 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 constitute a subset of online algorithms that schedule jobs as soon as they are released, and (iii) Earliest Finish Time algorithms, which constitute a subset of immediate dispatch algorithms that systematically schedule jobs on the machine that finishes the earliest (with specific tie-break strategies), and which provided excellent empirical results at the end of Chapter 2. 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 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, we show that the replication strategy has also a significant impact on this 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 $J = \{j\}_{1 \le j \le n}$ on m identical machines $M = \{i\}_{1 \le i \le m}$ in order to minimize the maximum weighted flow time $\max w_j F_j$ under the following set of constraints:

- jobs have heterogeneous processing times p_j,
- jobs have processing set restrictions $\mathcal{M}_j$,
- jobs have release times r_j,
- 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 $P \mid \mathcal{M}_j, \textit{online-}r_j \mid \max w_j F_j$. 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. [3], which prove that there is a lower bound of $\Omega(m)$ on the competitive ratio of any online algorithm for $P \mid \mathcal{M}_j, \textit{online-}r_j \mid F_{\max}$. However, several authors introduced variants where the subsets $\mathcal{M}_j$ are not arbitrarily defined anymore [75, 74], as realistic applications often exhibit specific structures in the processing sets. For instance, a common replication strategy in existing key-value stores, as seen in Chapters 1 and 2, consists in placing machines on a clockwise virtual ring and replicating the dataset of a given machine on its two direct neighbors [35, 66]. In this case, each processing set can be seen as an interval of three consecutive machines.

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.

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. Moreover, for each compatible structure, we define a corresponding variant, noted $\mathcal{M}_j(k\text{-}*)$, where processing sets have a fixed size k, i.e., $|\mathcal{M}_j| = k$ for all jobs j.*

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

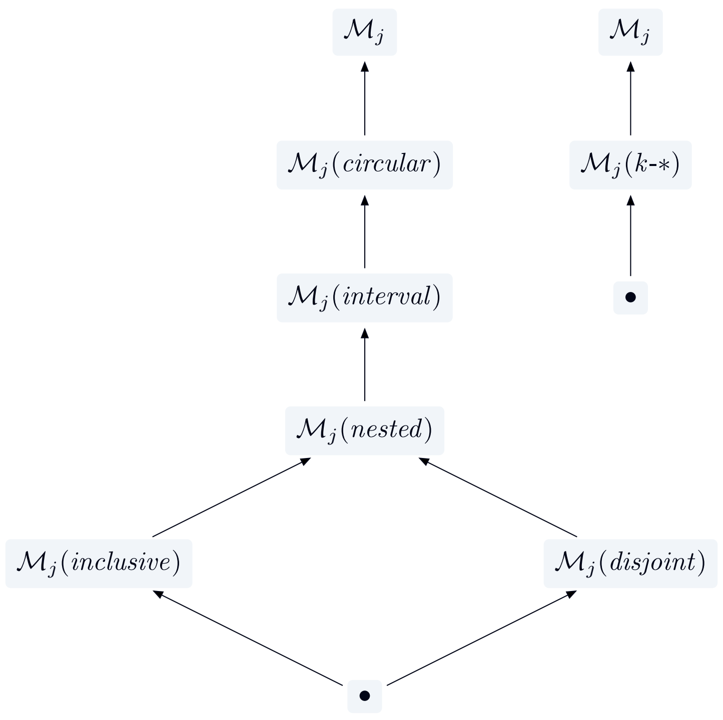
$\mathcal{M}_j(interval)$. *Identical to $\mathcal{M}_j(circular)$, with the constraint that $a_j \leq b_j$ for all jobs j.*

$\mathcal{M}_j(nested)$. *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 $\mathcal{M}_j \cap \mathcal{M}_{j'} = \emptyset$.*

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 3.1 gives the reduction graph of all previously-defined structures.



$$\begin{tikzcd}
 & \mathcal{M}_j & & \mathcal{M}_j \\
 & \mathcal{M}_j(circular) \arrow[u] & & \mathcal{M}_j(k\text{-}*) \arrow[u] \\
 & \mathcal{M}_j(interval) \arrow[u] & & \bullet \arrow[u] \\
 & \mathcal{M}_j(nested) \arrow[u] & & \\
\mathcal{M}_j(inclusive) \arrow[ur] & & \mathcal{M}_j(disjoint) \arrow[ul] & \\
 & \bullet \arrow[ul] \arrow[ur] & &
\end{tikzcd}$$

Figure 3.1 – 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.

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

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. [3]
$P \mid \mathcal{M}_j(\textit{nested}), \textit{online-}r_j, p_j = 1 \mid F_{\max}$	*Any online*	$\geq \frac{1}{3} \lfloor \log_2(m) + 2 \rfloor$	Theorem 3.4
$P \mid \mathcal{M}_j(k\textit{-interval}), \textit{online-}r_j, p_j = p \mid F_{\max}$	*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
$P \mid \mathcal{M}_j(k\textit{-size}), \textit{online-}r_j, p_j = p \mid F_{\max}$	*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	$\leq 3 - 2/ \max_j \{\lvert\mathcal{M}_j\rvert\}$	Corollary 3.12
$P \mid \mathcal{M}_j(k\textit{-disjoint}), \textit{online-}r_j \mid F_{\max}$	EFT	$\leq 3 - 2/k$	Corollary 3.12
$P \mid \mathcal{M}_j(k\textit{-interval}), \textit{online-}r_j, p_j = 1 \mid F_{\max}$	EFT-Min	$\geq m - k + 1$	Theorem 3.13
$P \mid \mathcal{M}_j(k\textit{-interval}), \textit{online-}r_j, p_j = 1 \mid F_{\max}$	EFT-Rand	$\geq m - k + 1$	Theorem 3.18
$P \mid \mathcal{M}_j(k\textit{-interval}), \textit{online-}r_j \mid F_{\max}$	EFT	$\geq m - k + 1$	Theorem 3.21

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.

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 Earliest Finish Time-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* $1 \mid online\text{-}r_j, p_j = 1 \mid \max w_j F_j$.

Proof: Consider 10 jobs with the following weights and release times:

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

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 or 10, then the adversary stops there, and $\mathcal{A}$ is clearly not optimal, as the objective is 10. 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 (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 (the objective is 12 at best). Hence, the competitive ratio of $\mathcal{A}$ is always strictly greater than 1. ■

Even if there is no optimal online algorithm for this problem, obtaining a guaranteed competitive ratio could still be possible, for instance by considering the following algorithm, called MaxFlow (Algorithm 4), which schedules the job with the highest weighted flow at each time step.

Algorithm 4 MaxFlow

```
1: when the single machine is idle at time $t$ do
2:     Execute an available job $j$ whose weighted flow (i.e., $w_j(t+1-r_j)$) is the highest
```

However, we show that, although very intuitive, this algorithm does not lead to a guaranteed ratio.

Theorem 3.2. *The competitive ratio of* MaxFlow *(Algorithm 4) is arbitrarily large for* $1 \mid online\text{-}r_j, p_j = 1 \mid \max w_j F_j$*, that is to say, we can always build an instance reaching a given ratio* ρ.

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 $h \geq 0$ 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

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

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

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

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 t such that $\rho \leq t < n$, 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 $\tau+1 < j \leq \tau+\rho$, we have

$$w_j(\tau+1-r_j) \leq \left(1+\frac{1}{\rho}\right)^{j-\tau-1} w_{t+2}(\tau+1-j+\rho).$$

Let us show that

$$\left(1+\frac{1}{\rho}\right)^{j-\tau-1} w_{t+2}(\tau+1-j+\rho) < w_{t+2}\rho$$

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, $w_\rho F_\rho^{\text{OPT}} = \rho(C_\rho - r_\rho) = \rho n$. For job n, $w_n F_n^{\text{OPT}} = w_n$. We prove that job n attains the maximum weighted flow, that is to say, we prove that $w_n \geq \rho n$, and we proceed 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

$$\rho(\rho+1) + \rho n_2 + \frac{n_2^2}{2} \geq \rho n.$$

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:

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

where $\gamma \approx 0.577$ is the Euler-Mascheroni constant. Let N denote this last bound, i.e., $N = \rho(\log(\rho) + \gamma + 1/(2\rho) + 1)$. 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. ■

We also derive a lower bound on the competitive ratio of any online algorithm when jobs have arbitrary processing times. In this case, the ratio is at least $\Delta + 1$, where Δ is the ratio between the maximum processing time and the minimum processing time of the jobs.

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

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 $1 \mid online\text{-}r_j \mid \max w_j F_j$ 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, $w_1F_1 = \sigma_1 + a$ and $w_2F_2 \geq W(\sigma_1+a+b-(\sigma_1+\varepsilon)) = W(a+b-\varepsilon)$. Therefore,

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

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 $w_2F_2^{\mathrm{OFF}} = Wb$, and then job 1 at time $\sigma_1 + \varepsilon + b$, which gives

$$\begin{aligned}w_1F_1^{\mathrm{OFF}} &= \sigma_1 + \varepsilon + b + a\\ &< a - \varepsilon + \varepsilon + b + a = 2a + b.\end{aligned}$$

As we have chosen W such that $W = 2a/b + 1$, we have $Wb = 2a + b$. Hence, $w_1F_1^{\mathrm{OFF}} < Wb$, and then $\max w_jF_j^{\mathrm{OFF}} = w_2F_2^{\mathrm{OFF}} = Wb$.

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 $w_1F_1^{\mathrm{OFF}} = a$, and then job 2 at time $r_2 = \sigma_1 + \varepsilon \geq a$, which gives $w_2F_2^{\mathrm{OFF}} = Wb$. We have $a < Wb$, hence, $\max w_jF_j^{\mathrm{OFF}} = w_2F_2^{\mathrm{OFF}} = Wb$.

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

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

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

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

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

From the previous results, it seems that minimizing the maximum weighted flow time is difficult even on a single machine. Thus we remove the weights in the objective function, in order to study the maximum flow time $F_{\max}$. We also move to the parallel environment with processing set restrictions.

The following result is inspired from the proof of Anand et al. [3], who show that the competitive ratio of any online algorithm is at least $\Omega(m)$ (where m is the total number of machines) for $P \,|\, \mathcal{M}_j, \textit{online-}r_j, p_j = 1 \,|\, F_{\max}$. As their analysis involves reordering the machines periodically, the considered processing sets do not exhibit any particular structure. We derive a similar result for the subproblem that consists in minimizing the maximum flow time when processing sets have a nested structure and jobs are unitary (i.e., $P \,|\, \mathcal{M}_j(\textit{nested}), \textit{online-}r_j, p_j = 1 \,|\, F_{\max}$), by showing that the competitive ratio of any online algorithm is at least $\frac{1}{3}\lfloor \log_2(m) + 2 \rfloor$ in this case.

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* $P \mid \mathcal{M}_j(nested), online\text{-}r_j, p_j = 1 \mid F_{\max}$.

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 algorithm, and let F be an arbitrary number such that $F \geq \log_2(m) + 2$. We build the following adversary. 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 $\{u_0, u_0+1, \cdots, u_0+s_0-1\}$), which we denote by $I(u_0, s_0)$, where $u_0 = 1$ and $s_0 = m$. The adversary submits s_0 unitary jobs at time t_0, with processing set $I(u_0, s_0)$. Let $\mathcal{J}_{1,0}$ denote this set of jobs. Next, for each machine $i \in I(u_0, s_0)$, the adversary releases one unitary job at each time $t_0, t_0+1, \cdots, t_0+F-1$ and feasible only on the machine i. Let $\mathcal{J}_{2,0}$ denote this second set of jobs. Note that at time $t_0 + F - 1$, algorithm $\mathcal{A}$ should have completed the jobs of $\mathcal{J}_{1,0}$, otherwise the maximum flow time is greater than $\log_2(m) + 2$.

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

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

(i) $s_h = \frac{m}{2^h}$, and

(ii) there are at least hs_h uncompleted jobs on $I(u_h, s_h)$ at time t_h before sending $\mathcal{J}_{1,h}$ and $\mathcal{J}_{2,h}$, i.e., $|\mathcal{J}_{0,h}| \geq hs_h$.

For the base case ($h = 0$), we have $s_0 = \frac{m}{2^0} = m$, and $\mathcal{J}_{0,h} = \emptyset$, so there is no completed job on $I(1, m)$ at time 0 before sending $\mathcal{J}_{1,0}$ and $\mathcal{J}_{2,0}$. Now assume that $s_h = \frac{m}{2^h}$ is true at a certain step h. At step $h+1$, we have $s_{h+1} = \frac{1}{2}s_h$ by definition, so $s_{h+1} = \frac{1}{2} \cdot \frac{m}{2^h} = \frac{m}{2^{h+1}}$, which proves Statement (i). Suppose that there are at least hs_h uncompleted jobs on $I(u_h, s_h)$ at time t_h, i.e., $|\mathcal{J}_{0,h}| \geq hs_h$. Then we send $\mathcal{J}_{1,h}$ and $\mathcal{J}_{2,h}$, which means that there are at least

$$hs_h + s_h + Fs_h - Fs_h = (h+1)s_h$$

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

We stop when we reach the step h such that $s_h = 1$. This means that $\frac{m}{2^h} = 1$, i.e., $h = \log_2(m)$. Therefore, there remains at least $hs_h = \log_2(m)$ uncompleted jobs on an interval of size 1 at time t_h, plus 1 job of $\mathcal{J}_{1,h}$ and 1 job of $\mathcal{J}_{2,h}$, 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 $0 \leq h < \log_2(m)$, in executing all jobs of $\mathcal{J}_{1,h}$ on the subinterval $I(u_h, s_h) \setminus I(u_{h+1}, s_{h+1})$, for a max-flow of 3: jobs of $\mathcal{J}_{1,h}$ are scheduled first (with flow 2), followed by jobs of $\mathcal{J}_{2,h}$, which have a flow at most 3. The conclusion follows. ■

A simpler result can be obtained for the case where the processing sets are intervals of fixed size k, and jobs have a common processing time p. Note that, similarly to $\mathcal{M}_j(nested)$, $\mathcal{M}_j(k\text{-}interval)$ is another particular case of $\mathcal{M}_j(interval)$. We prove that the competitive ratio of any online algorithm is at least 2 when p tends to $+\infty$.

Theorem 3.5. *As $p \to +\infty$, the lower bound on the competitive ratio of any online algorithm tends to 2 for $P \mid \mathcal{M}_j(k\text{-}interval), online\text{-}r_j, p_j = p \mid F_{\max}$.*

Proof: Let $\mathcal{A}$ be an arbitrary online algorithm, and consider an instance with $m = 4$ machines. At time 0, the adversary sends one job with processing time p and processing set $\mathcal{M}_1 = \{2, 3\}$. Now there are two cases. Either $\mathcal{A}$ executes this job on machine 2, or on machine 3. 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 $\sigma_1 < p$.

Now suppose that $\mathcal{A}$ executes job 1 on machine 2. Then the adversary sends two jobs at time $\sigma_1 + 1$ with processing time p and with processing set $\mathcal{M}_2 = \mathcal{M}_3 = \{1, 2\}$. $\mathcal{A}$ 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 maximum 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 maximum 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 and 4. ■

3.3.2 Lower Bounds for Immediate Dispatch Algorithms

The lower bound on the competitive ratio can be largely increased when considering immediate dispatch algorithms. We first study $\mathcal{M}_j(inclusive)$. We show in Theorem 3.6 that restricting to this processing set structure enables to derive a lower bound of $\lfloor \log_2(m) + 1 \rfloor$ on the competitive ratio of any immediate dispatch algorithm. This is also true for $\mathcal{M}_j(nested)$ and $\mathcal{M}_j(interval)$, as they generalize $\mathcal{M}_j(inclusive)$.

Theorem 3.6. *As $p \to +\infty$, the lower bound on the competitive ratio of any immediate dispatch algorithm tends to $\lfloor \log_2(m) + 1 \rfloor$ for $P \mid \mathcal{M}_j(inclusive), online\text{-}r_j, p_j = p \mid F_{\max}$.*

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 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 $\log_2(m)$. 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

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

The optimal strategy consists in scheduling each set $\mathcal{J}_\ell$ on machines $\mathcal{M}^{(\ell)}\setminus\mathcal{M}^{(\ell+1)}$, for a maximum 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. *As $p\to+\infty$, the lower bound on the competitive ratio of any immediate dispatch algorithm tends to $\lfloor\log_k(m)\rfloor$ for $P\mid\mathcal{M}_j(k\text{-size}), \text{online-}r_j, p_j=p\mid F_{\max}$.*

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 build the following adversary. For each ℓ such that $1\le\ell\le\log_k(m)$, 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 $\log_k(m)p$, and the maximum flow time is $\log_k(m)p-(\log_k(m)-1)$. Note that

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

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

Algorithm 5 FIFO

```
Require: Global first-come first-served queue $Q$
Input: Incoming jobs $j$
Output: Allocated machines $\mu_j$ and starting times $\sigma_j$
1: when a new job $j$ is released do
2:     $enqueue(j, Q)$
In parallel, do:
1: when a set of machines $U$ become idle at time $t$ do
2:     $j \leftarrow dequeue(Q)$
3:     if $j \neq nil$ then
4:         $u \leftarrow$ BreakTie($U$)
5:         $\mu_j \leftarrow u$
6:         $\sigma_j \leftarrow t$
```

3.3.3 Upper Bounds for Earliest Finish Time

First In First Out (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). We move our focus to the Earliest Finish Time (EFT) scheduler (see Algorithm 6), which pushes each released job on the machine that is scheduled to finish the earliest. We show in this section that both schedulers are equivalent on any instance of the problem $P \mid online\text{-}r_j \mid F_{\max}$. 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, EFT 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 relevant for distributed systems, whereas FIFO is more suited 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 the subset of 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 $C_{j'} = \sigma_{j'} + p_{j'}$, which is the completion time of j' in π. By convention, $\mathcal{C}_{i,0} = 0$ for all i. 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

Algorithm 6 EFT

```
Input: Incoming jobs $j$
Output: Allocated machines $\mu_j$ and starting times $\sigma_j$
1: when a new job $j$ is released do
2:     Get $U_j$ according to completion times of machines $M$ (Equation (3.3))
3:     $u \leftarrow \textsc{BreakTie}(U_j)$
4:     $\mu_j \leftarrow u$
5:     $\sigma_j \leftarrow \max(r_j, \mathcal{C}_{u,j-1})$
6:     Update the completion time of machine $u$
```

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

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).

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 $\sigma_{h+1}^{\text{EFT}} = \max\left(r_{h+1}, \mathcal{C}_{u,h}^{\text{EFT}}\right)$. On the other hand, at time $\max\left(r_{h+1}, \min_i\left\{\mathcal{C}_{i,h}^{\text{FIFO}}\right\}\right)$, 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 $\sigma_{h+1}^{\text{FIFO}} = \max\left(r_{h+1}, \mathcal{C}_{u',h}^{\text{FIFO}}\right)$.

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 jobs 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. ■

Note that FIFO has been proven to be $(3 - 2/m)$-competitive when minimizing maximum flow time on parallel machines without any processing set restriction [18, 19, 80], which also makes it optimal on a single machine. For completeness, we give the proof of this result, which implies by Proposition 3.8 that EFT is also $(3 - 2/m)$-competitive for this problem.

Theorem 3.9 (Bender et al. [19])**.** FIFO *is* $(3 - 2/m)$*-competitive for* $P \mid online\text{-}r_j \mid F_{\max}$.

Let $\mathcal{J}_t$ denote the set of jobs released before or at time t and not yet started in a given schedule π, 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 job 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 $F_{\max,j}$. Let π^{FIFO} and π^{OPT} respectively denote a FIFO and an optimal schedule.

Lemma 3.10. *For all jobs j, $W_{r_j}^{\text{FIFO}} \leq W_{r_j}^{\text{OPT}} + (m-1)p_{\max,j}$.*

Proof: Let us proceed by induction on jobs. In the base case ($j = 1$), all machines are idle at time r_1, and we have $W_{r_1}^{\text{FIFO}} = W_{r_1}^{\text{OPT}} = p_1$. Now 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}. In this case, we have

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

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

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

(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 $W_{r_{j+1}}^{\text{FIFO}} \leq p_{j+1} + (m-1)p_{\max,j}$.

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 $W_{r_{j+1}}^{\text{OPT}} \geq p_{j+1}$. Therefore,

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

■

Proof of Theorem 3.9: Let us start with two lower bounds for $F^{\mathrm{OPT}}_{\max,j}$ (for any job j):

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

and

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

Lower Bound 3.5 is immediate. For Lower Bound 3.6, we see that we necessarily need to execute a volume of work $W^{\mathrm{OPT}}_{r_j}$ after time r_j, and this work completes at time $r_j + W^{\mathrm{OPT}}_{r_j}/m$ in the best case. Let j' be the last job (possibly j) that completes in this volume of work. As j was the last release job among $W^{\mathrm{OPT}}_{r_j}$, we have $r_{j'} \leq r_j$, thus $F^{\mathrm{OPT}}_{j'} = r_j + W^{\mathrm{OPT}}_{r_j}/m - r_{j'} \geq W^{\mathrm{OPT}}_{r_j}/m$.

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 $r_j + W^{\mathrm{OPT}}_{r_j}/m$.

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

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

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

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

The case of disjoint processing sets is particular. We may apply a competitive algorithm independently on each unique 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 $\mathcal{A}$ to each instance $\mathcal{I}_u$. By definition of the competitive ratio of $\mathcal{A}$, we have $F_{\max}^{\mathcal{A}}(\mathcal{I}_u) \leq f(|\mathcal{M}_u|)F_{\max}^{\mathrm{OPT}}(\mathcal{I}_u)$, where OPT is an optimal offline strategy. As $\mathcal{I}_u$ is a subinstance of $\mathcal{I}$, we also have

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

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, $F_{\max}^{\mathcal{A}}(\mathcal{I}_u) \leq f(|\mathcal{M}_u|)F_{\max}^{\mathrm{OPT}'}(\mathcal{I})$, and

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

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* $P \mid \mathcal{M}_j(\textit{disjoint}), \textit{online-}r_j \mid F_{\max}$ *and* $(3-2/k)$*-competitive for* $P \mid \mathcal{M}_j(k\textit{-disjoint}), \textit{online-}r_j \mid F_{\max}$.

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

3.3.4 Lower Bounds for Earliest Finish Time

Although EFT is a guaranteed immediate dispatch algorithm when processing sets are disjoint, it turns out that this is not the case when these processing sets are fixed-size intervals of machines, even when jobs are unitary. We prove in the following that the competitive ratio of EFT is at least $m - k + 1$ in a variety of settings.

To exhibit this result, we first need to focus on a specific tie-break function. We start by studying the MIN policy: 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.

Algorithm 7 EFT-MIN

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

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

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., $\mathcal{M}_j = \{\lambda, \cdots, \lambda + k - 1\}$. Let us build the following instance (we illustrate an EFT-MIN schedule of this instance in Figure 3.2). At each integer time step, we send m jobs such that:

(i) for all $1 \leq j \leq m - k$, the job j is of type $m - k - j + 2$ (blue job in Figure 3.2), and

(ii) for all $m - k < j \leq m$, the job j is of type 1 (red job in Figure 3.2).

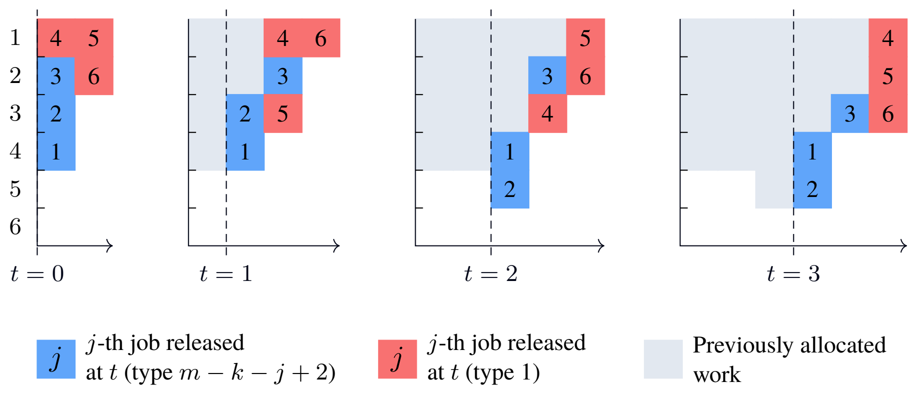

1
2
3
4
5
6
$t = 0$
$t = 1$
$t = 2$
$t = 3$
j-th job released at t (type $m-k-j+2$)
j-th job released at t (type 1)
Previously allocated work

Figure 3.2 – An EFT-Min schedule of the instance from time $t = 0$ to $t = 3$, for $m = 6$ and $k = 3$. Colored jobs are released in-order at each time t.

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 next m jobs, mt jobs have already been scheduled in π^{EFT}, and each machine i thus completes at time $\mathcal{C}_{i,mt}$ (by definition in Equation (3.3)). 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 release of the next m jobs at time t. 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

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

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 machines i,

(ii) $w_t \leq w_{t'}$ if $w_t(i) \leq w_{t'}(i)$ for all machines 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 *$w_{t'}$).*

Definition 3.4. *For any t, we say that w_t* has a plateau *from machine i_1 to machine i_2 if and only if $w_t(i) = w_t(i_1) = w_t(i_2)$ for all machines i such that $i_1 \leq i \leq i_2$.*

Figure 3.3 shows an example of a schedule profile w_t. The proof of Theorem 3.13 consists in two phases. First, we show that when the current schedule profile is strictly behind the stable profile w_τ, there exists a future time such that the schedule profile is closer to w_τ (Lemma 3.16). Second, we show that at any time step, either we can find a past time such that the schedule profile exceeds the stable profile w_τ for at least one machine, or the current schedule profile is behind w_τ (Lemma 3.17). Before we dive into the proof, we start with the following two lemmas, which will be of particular importance when proving Lemma 3.16.

Lemma 3.14 (Non-increasing function). *For any time t, w_t is a non-increasing function, i.e., $w_t(i+1) \leq w_t(i)$ for all machines i such that $1 \leq i < m$.*

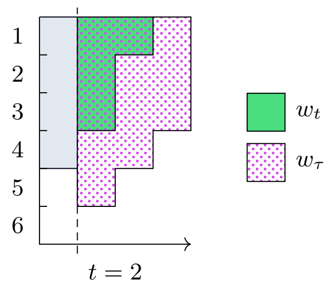


Figure 3.3 – The schedule profile w_t of EFT-Min at time t (in green), just before the adversary sends m new jobs. w_t is strictly behind the stable schedule profile w_τ we want to reach (in purple). Moreover, w_t has a plateau from machine 2 to machine 3, and a plateau from machine 4 to machine 6.

Proof: Let us proceed by induction on t. The base case ($t = 0$) is direct since no job has arrived yet, so $w_0(i) = 0$ for all machines i. 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 $w_{t+1}(i+1) > w_{t+1}(i)$ at time $t+1$. 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' denote this specific time.

If we had $w_{t'}(i) > w_{t'}(i+1)$, 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 $w_{t'}(i) \leq w_{t'}(i+1)$ at time t', 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' (i.e., $w_{t'}(i') \geq w_{t'}(i+1)$ 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 $i+1$ (their type is at least $i+2$). 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 job j starting at time t on machine $i+1$, 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 $w_{t+1}(i+1) > w_{t+1}(i) \geq 0$.

(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 machine $i+1$ between times 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 3.4 helps to visualize the described situation.

But the last job j scheduled on machine $i+1$ 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', thus they were at least as much loaded as i. As a

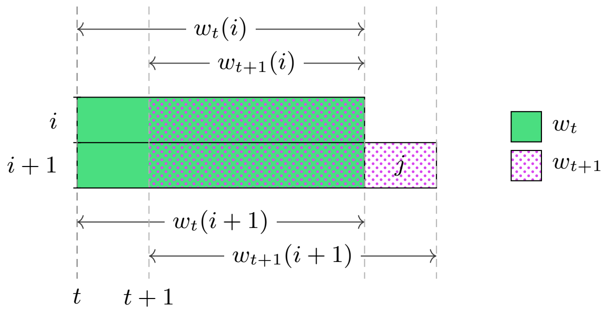


Figure 3.4 – Schedule profiles on machines i and $i+1$ at times t and $t+1$, under the described hypotheses.

consequence, at least one job must have been scheduled by EFT-Min on machine i between times t' and $t+1$, which is a contradiction. ■

Lemma 3.15 (Idleness property). *If $w_t(i) = 0$ for some $t, i < m$, we have $\sum_i w_{t+1}(i) > \sum_i w_t(i)$.*

Proof: If 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, i.e., $w_t(i') = 0$ for all $i' > i$ (as w_t is non-increasing, by Lemma 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 $m-k+1$ and will be scheduled on the first lightly-loaded compatible machine (with index $\max(i, m-k+1)$).

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

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 3.16. *For any time t such that $w_t < w_\tau$, there exists a time $t' > t$ such that*

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

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

Proof: If the schedule profile is strictly behind w_τ, we will show that there must exist a plateau somewhere, and this plateau will necessarily propagate on next machines step by step, until we reach a time t such that $w_t(m-1) = 0$. Then we will be able to apply the idleness property.

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.

Existence of a plateau. As $w_t \leq w_\tau$ and i' is the highest machine index such that $w_t(i') < w_\tau(i')$, we necessarily 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 $i < k$, the schedule profiles of all machines $i+1, \cdots, k$ are also strictly behind the stable profile w_τ (by Lemma 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,

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

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 of Lemma 3.15, the conclusion is immediate. Otherwise, $i' < m-1$, so $w_t(m-1) = w_\tau(m-1) = 1$. By Lemma 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 their type is at most $i' - k + 1$. Figure 3.5 shows the propagation process.

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

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

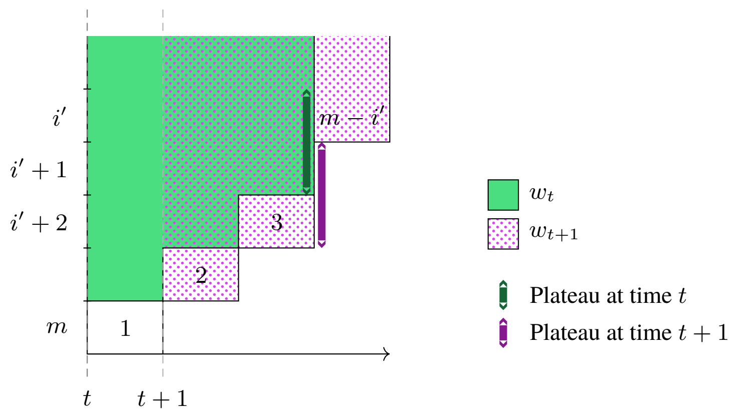


Figure 3.5 – Propagation of the plateau from machines i' and $i'+1$ at time t to machines $i'+1$ and $i'+2$ at time $t+1$.

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 of Lemma 3.15 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 $w_\mathcal{T}$, or the current profile is behind $w_\mathcal{T}$.

Lemma 3.17. *For any time t, either*

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

(ii) $w_t \leq w_\mathcal{T}$.

Proof: Let us proceed by induction on t. Obviously, the base case ($t=0$) is true, as $w_0 = 0 \leq w_\mathcal{T}$. Now we have two cases in the induction step.

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

Case (ii). Now suppose that $w_t \leq w_\mathcal{T}$ for some time t. By contradiction, let us assume that there exists a machine i such that $w_{t+1}(i) > w_\mathcal{T}(i)$. Combined to the fact that $w_t(i) \leq w_\mathcal{T}(i)$, 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 $w_\mathcal{T}(i) = m-k$, and we have $w_{t+1}(i) > w_\mathcal{T}(i) = m-k$. This proves the induction.

(b) $i > k$. By induction hypothesis, we know that $w_t(m) = 0$ (because $w_\mathcal{T}(m) = 0$), 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 machine 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 machine i, because their machine interval starts after λ_j. As j is the last job of machine 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 times 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 job j has been allocated on machine i and not on machine $i-1$, 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, job j' starts after time $\mathcal{C}_{i-1,mt}$. Hence, the scheduling of j' occurred between times t and $t+1$, before the scheduling of job 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 job j' has been scheduled on machine $i-1$ 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 machine index such that $w_{t+1}(i) > w_\tau(i)$. This concludes the proof. ■

We are now able to prove the full theorem.

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 machine i. For a given time t, we know by Lemma 3.17 that either

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

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

If Case (i) is true, we are done. If Case (ii) is true, either $w_t < w_\tau$ or $w_t = w_\tau$. If $w_t = w_\tau$, we are done. Otherwise, if $w_t < w_\tau$, we know we can find a future time t' such that $\sum_i w_{t'}(i) > \sum_i w_t(i)$ (by Lemma 3.16). 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 last k jobs will be allocated on the first k 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 whose type is at least $k + 1$ 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 conclusion follows. ■

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 3.18 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 $\mathcal{M}_j$ (Equation (3.4))
3:     $u \leftarrow \textsc{Rand}(U'_j)$
4:     $\mu_j \leftarrow u$
5:     $\sigma_j \leftarrow \max(r_j, \mathcal{C}_{u,j-1})$
6:     Update the completion time of machine $u$
```

Theorem 3.18. *The competitive ratio of* EFT-Rand *is at least $m - k + 1$ (almost surely), where $1 < k < m$, for $P \mid \mathcal{M}_j(k\text{-interval}), \text{online-}r_j, p_j = 1 \mid F_{\max}$. 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 Lemma 3.19, we show that this distance decreases with t.

Lemma 3.19. *For any time t,*

(i) if there exists a job j such that $1 \leq j \leq m - k$ released at time 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: Let us prove the two cases separately.

Case (i). At a given time t, suppose there exists at least one job j (such that $1 \leq j \leq m-k$) released at time 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 3.20. *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 machines 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 3.18, 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, in case of a tie including the last machine, the probability to put a job on it is lower than or equal to $1-\theta$. In other words, Rand always has a non-zero probability to choose a machine different from the last one during a tie.

Proof of Theorem 3.18: It is clear from Lemma 3.19 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 Lemma 3.19, 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., $|U_{mt'+j}| > 1$). 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 makes the decision of selecting the last machine during a tie 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 $|U_{mt'+j}| = 1$ for all jobs $j \leq m - k$ released at t'. By Lemma 3.20, 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 $t' + m - k$, and we have $F_{\max} \geq t' + m - k - (t' - 1) = m - k + 1$. The conclusion follows. ■

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

Theorem 3.21. *The competitive ratio of* EFT *(with any tie-break policy) is at least* $m - k + 1$ *for* $P \mid \mathcal{M}_j(k\text{-interval}), \textit{online-}r_j \mid F_{\max}$.

Proof: 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. Note that, in this proof, machine indices are reversed for convenience. Our objective is to enforce the following property.

Property 3.5. *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 no tie anymore among these machines for the second round. In the second round, we submit jobs whose

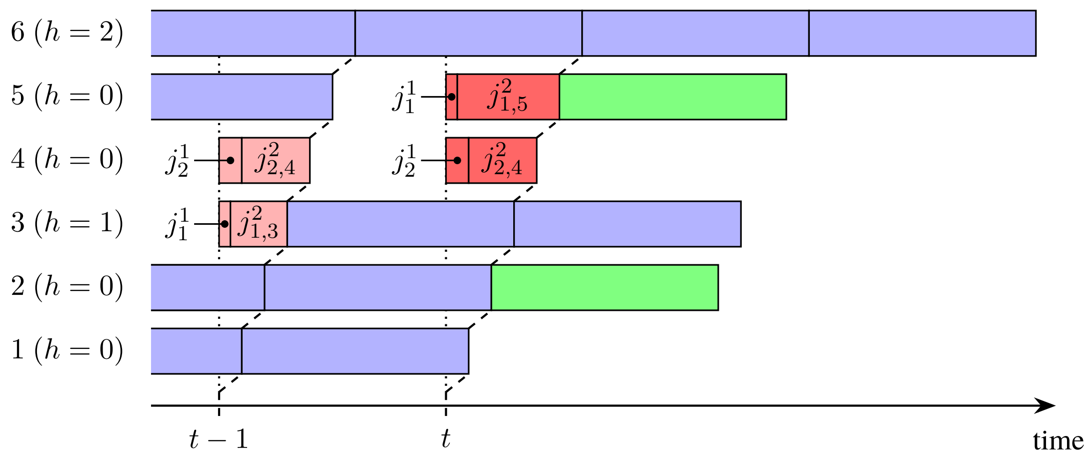


Figure 3.6 – Illustration of the construction of the instance for Theorem 3.21 when adding small jobs at time t. Regular jobs submitted before t are depicted in blue. Small jobs added to ensure the common delay of $i\delta$ are in red (dark red for step t, light red for step $t-1$), and regular jobs submitted at step t are in green. Only two machines are not processing any regular jobs before time t (machines 4 and 5) and require small jobs to ensure the common delay of $i\delta$.

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 Property 3.5 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 $t + 1 + i\delta$.

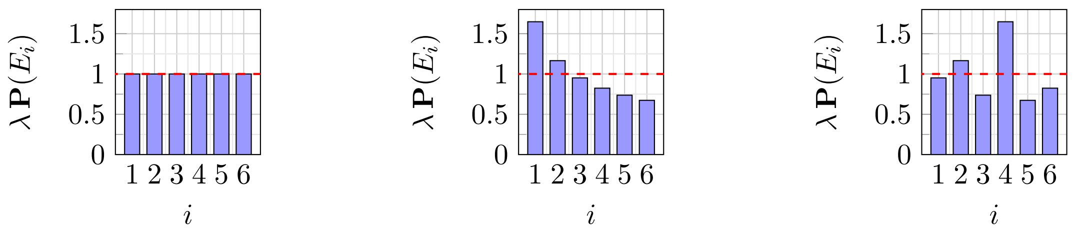
Lemma 3.22. *With the additional (non regular) jobs, the execution of any* EFT *algorithm (with any tie-break policy) follows the original* EFT *policy (with tie-break by selecting the machine with smallest index), up to a delay of $i\delta$ for each machine i.*

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 3.21 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 maximum achievable throughput of the system and the practical performance of simple scheduling heuristics. We focus on circular interval processing sets, because they are used in actual systems [35, 66, 96], and disjoint processing sets, because it is the restriction for which we have the best, and only, approximation ratio (Theorem 3.11). Moreover, the performance of actual systems is affected by the popularity of requests, which is not uniform, that is to say, certain jobs restricted to the same processing set appear more frequently than others. We begin by modeling popularity before explaining the method we used to evaluate the theoretical maximum throughput enabled by data item replication. Finally, we perform



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

Figure 3.7 – Example of load distribution on a cluster of $m = 6$ machines, with $\lambda = m$, for each case.

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

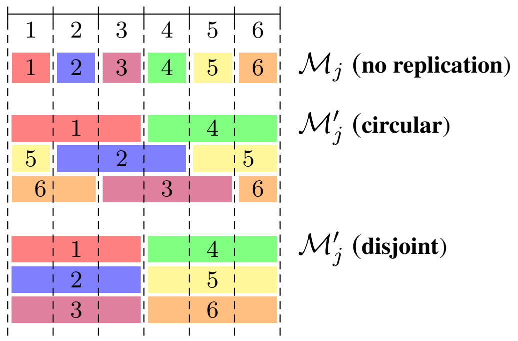
Let us consider a cluster of m machines, where jobs have a unit processing time and are released according to a Poisson process with parameter λ, which implies that λ jobs are released in average at each time unit. The m machines can process at most m jobs at each time unit, hence λ/m measures the average 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 [46]. 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 of generating a monotonically decreasing load on machines $1, \cdots, m$. This corresponds to the 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 3.7 shows an example of load distribution for each case.

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



1 2 3 4 5 6
$\mathcal{M}_j$ (no replication)
$\mathcal{M}'_j$ (circular)
$\mathcal{M}'_j$ (disjoint)

Figure 3.8 – Example of replication strategies in overlapping and disjoint settings, with $k = 3$. For example, suppose that a job j is feasible on machine 3 only ($\mathcal{M}_j = \{3\}$). Then, in overlapping setting (resp. disjoint setting), the new processing set restriction of j is $\mathcal{M}'_j = \{3, 4, 5\}$ (resp. $\mathcal{M}'_j = \{1, 2, 3\}$).

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 3.8 illustrates these constructions.

Overlapping intervals. There are m distinct overlapping replication intervals of size k, arranged as a ring, which corresponds to the $\mathcal{M}_j(k\text{-}circular)$ restriction:

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

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 [35, 66]. We have seen in Theorems 3.13, 3.18 and 3.21 that EFT does not always provide a good competitive ratio when minimizing maximum flow time with the non-circular version of this structure.

Disjoint intervals. We divide the cluster into $\left\lceil \frac{m}{k} \right\rceil$ disjoint replication intervals of size k, which corresponds to the $\mathcal{M}_j(k\text{-}disjoint)$ restriction:

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

where $u' = k \left\lfloor \frac{u-1}{k} \right\rfloor$. Theorem 3.11 and its corollary give related results, i.e., 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 maximum cluster load permitted by a given replication strategy for a specific popularity distribution, we solve the following optimization problem modeled as a linear program:

$$
\begin{aligned}
\textbf{maximize} \quad & \lambda && \text{(3.17a)}\\
\text{subject to} \quad & \forall v, \sum_u a_{uv} = \lambda\,\mathbf{P}(E_v), && \text{(3.17b)}\\
& \forall u, \sum_v a_{uv} \leq 1, && \text{(3.17c)}\\
& \forall u, v \text{ s.t. } u \notin I_k(v),\ a_{uv} = 0, && \text{(3.17d)}\\
& \forall u, v, a_{uv} \geq 0, && \text{(3.17e)}\\
& \lambda \geq 0 && \text{(3.17f)}
\end{aligned}
$$

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

- The total work corresponding to jobs originally restricted on u is exactly equal to the initial work of v (Equation (3.17b)).
- The average work eventually processed on u does not exceed 1 (Equation (3.17c)).
- We can transfer work from v to u if and only if u belongs to the interval of size k generated from v according to the considered replication strategy, i.e., all jobs that could originally run exclusively on v can now also run on u (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 scheduling experiments in real key-value store systems [54, 97]. Figure 3.9 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 3.9 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 3.10, 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 and without a popularity bias, on 10 000 generated unitary jobs, which is sufficient to reach a steady state. Figure 3.11 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 the 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.

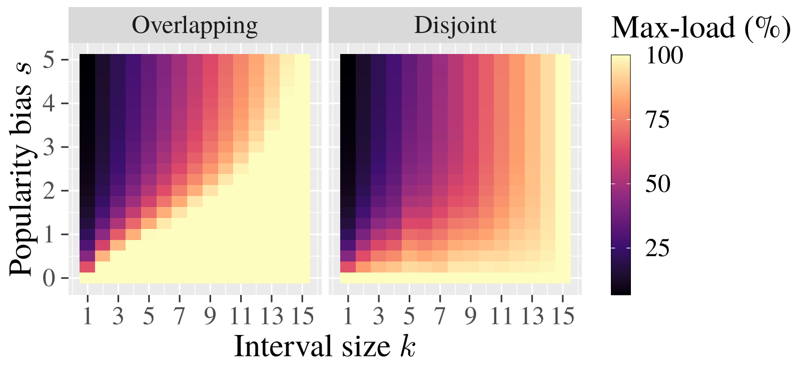


Figure 3.9 – Maximized load obtained by solving Linear Program 3.17, for both overlapping and disjoint strategies, for each $0 \leq s \leq 5$ (by steps of 0.25) and $1 \leq k \leq m$, in the **Shuffled case**.

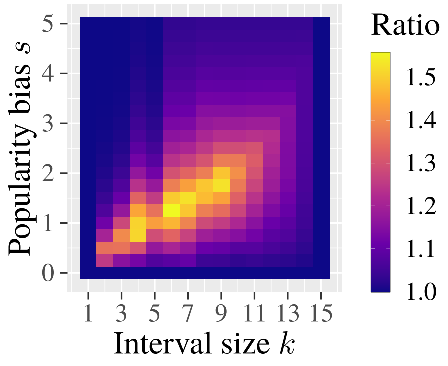


Figure 3.10 – Ratio between the max-load of overlapping and disjoint strategies, for each $0 \leq s \leq 5$ (by steps of 0.25) and $1 \leq k \leq m$, in the **Shuffled case**.

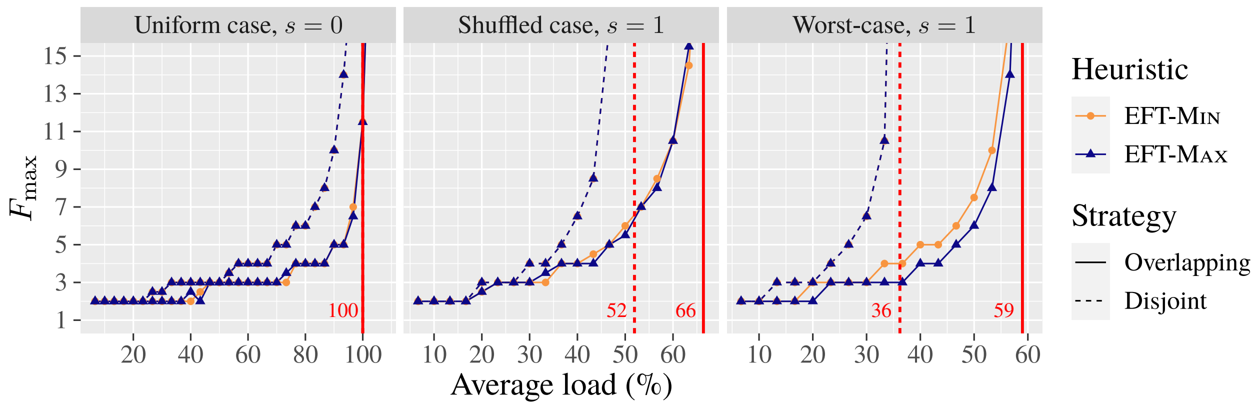


Figure 3.11 – Maximum flow time given by both heuristics EFT-Min and EFT-Max as a function of the average load when $k = 3$, for both the overlapping (solid lines) and disjoint (dashed lines) strategies. Each facet corresponds to a distinct case (for the **Worst-case** and **Shuffled case**, we set $s = 1$). 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. Knowing if there exists a replication strategy giving both good practical results and theoretical guarantees on EFT scheduling remains an open question.

3.5 Conclusion

In this chapter, we demonstrated that the replication strategy has a significant impact on the guarantees we may obtain on the maximum response time and the attainable average throughput of key-value stores. In particular, we showed that a very reasonable scheduling algorithm such as EFT, whose competitive ratio is guaranteed to be lower than $3 - 2/m$ without processing set restriction and lower than $3 - 2/k$ on fixed-size disjoint sets, can have poor performance when used with the common replication scheme of key-value stores where processing sets consist of fixed-size intervals. However, when considering the maximum attainable throughput, we showed through an exact method that the disjoint structure is less efficient than overlapping intervals when key access frequencies follow a Zipf law under various settings. Indeed, the maximum supported load of the cluster is up to 50% higher when intervals overlap.

In the next chapter, we continue investigating the interval structure of the processing sets in the problem. However, we shift our focus from the response time of the complete workload to the load-balancing of a specific type of requests, called *multi-get* requests, that are able to retrieve several data items in a single round-trip and have been the subject of recent advances in key-value store architectures.

4

Partitioning and Balancing Multi-Get Requests on the Cluster

4.1 Introduction

A typical use-case of key-value stores is a webservice that must retrieve several data items to respond to a client request. Of course, the number of needed items differs between client requests, and goes from one to several thousands. To reduce the number of network round-trips between the webservice and the storage system, the read operations that are performed for a single client request may be aggregated into a special kind of request, called a *multi-get* request, which consists in retrieving several data items at once in the store [89, 55]. As the dataset is distributed on several servers in the key-value store, the multi-get request must be split into sub-requests that will be sent to the appropriate servers. This *partitioning* of multi-get requests is the subject of this chapter.

The first section (Section 4.2) is dedicated to a thorough explanation of the structure of multi-get requests, and why the partitioning step is important to guarantee a good response time for the overall multi-get request. Then, we introduce the RESTRICTED ASSIGNMENT problem, which is a well-known algorithmic problem that we can use to model the partitioning and load-balancing of multi-get requests more formally. In Section 4.3, we refine this problem to a special case, called the RESTRICTED ASSIGNMENT problem on intervals of machines (the RAI problem, for short), which is more suited to the context of key-value stores. Based on an existing algorithm, we derive an efficient greedy algorithm for the RAI problem with unitary jobs, and we generalize it to a $(2 - 1/m)$-approximation algorithm for arbitrary processing times. Then, we generalize the problem even further to the case of *circular* intervals of machines (Section 4.4), which matches exactly the typical replication strategy of key-value stores. We give a general procedure that, when used in conjunction with a polynomial-time algorithm for the standard RAI problem, computes an optimal solution for the RAI problem with circular intervals and K job types,

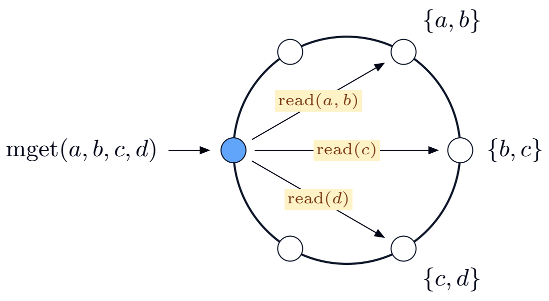


Figure 4.1 – Example of a multi-get request. The keyset $\{a, b, c, d\}$ is partitioned into three opsets ($\{a, b\}$, $\{c\}$ and $\{d\}$), which are sent to the appropriate servers.

where K is an arbitrary integer. This allows us, for instance, to derive an efficient and optimal solution for the RAI problem with circular intervals and unitary jobs.

4.2 Motivation & Model

In this section, we introduce the Restricted Assignment problem, which arises in the context of multi-get requests in key-value stores. We begin with an explanation of the applicative structure of this special kind of requests in Section 4.2.1, and we present the formal problem in Section 4.2.2.

4.2.1 Multi-Get Requests in Key-Value Stores

In key-value stores, 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 a webservice and the database, as a single request often requires to retrieve several data items before responding to the client [89, 55]. In such a multi-get request, the requested keys (which constitute the *keyset* of the request) may be located in different data partitions, which are stored on different servers. Thus, the coordinator (i.e., the receiving server) must split the multi-get request into several sub-requests, each sub-request being redirected towards the appropriate storage server. In other words, the keyset must be partitioned into several subsets (one per sub-request), and each subset must include keys that are located on the same server. These subsets are called the *opsets* of the multi-get request. Choosing these opsets is a crucial step, because the key-value store cannot respond before gathering all requested data items (i.e., executing all sub-requests). We do not want a few very fast sub-requests, and one that is very slow. The opsets must therefore be well-balanced to guarantee a good response time for the overall multi-get request. Figure 4.1 gives an example of execution of a multi-get request in a distributed key-value store.

Choosing the opsets may be seen as a scheduling problem, where servers are the machines, and each single read operation for a given key is a job j, whose processing time p_j is the time required to retrieve the corresponding data item (which depends on the number of bytes that represent the stored item). Moreover, each job has its own processing set restriction $\mathcal{M}_j$, which corresponds to the servers on which the requested key is located. Then, partitioning the jobs on machines in the context of the scheduling problem is equivalent to choosing the opsets of the multi-get request, and minimizing the maximum completion time of jobs is equivalent to minimizing imbalance between opsets. Of course, when balancing a given 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. In Figure 4.2, we show how a partitioning of a multi-get request may be suboptimal.

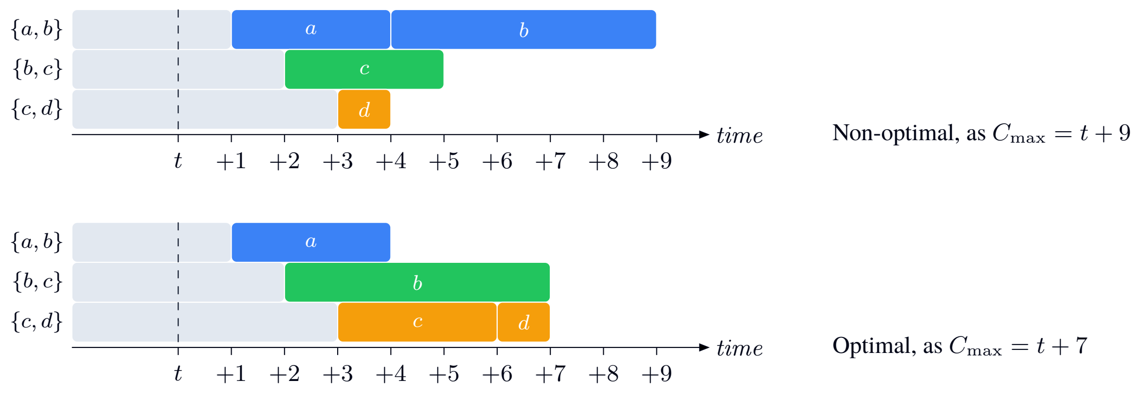

{a, b}
{b, c}
{c, d}
a
b
c
d
time
t +1 +2 +3 +4 +5 +6 +7 +8 +9
Non-optimal, as $C_{\max} = t + 9$
Optimal, as $C_{\max} = t + 7$

Figure 4.2 – Two possible choices of partitioning of a multi-get request released at time t. The gray areas are fake jobs that represent the current load of each machine. In the first case, the opsets are $\{a, b\}$, $\{c\}$ and $\{d\}$, which causes imbalance. In the second case, the opsets are $\{a\}$, $\{b\}$ and $\{c, d\}$, which is the best possible choice for the current multi-get request.

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 a variable representing the makespan to minimize, and each x_{ij} is a binary variable that indicates whether the job j is assigned to the machine i:

$$\begin{aligned}
\textbf{minimize} \quad & c && \text{(4.1a)}\\
\text{subject to} \quad & \forall j \in J, \sum_{i \in M} x_{ij} = 1, && \text{(4.1b)}\\
& \forall i \in M, \sum_{j \in J} x_{ij} p_j \leq c, && \text{(4.1c)}\\
& x_{ij} \in \{0, 1\}\,. && \text{(4.1d)}
\end{aligned}$$

Equation (4.1b) ensures that each job is assigned to exactly one machine, and Equation (4.1c) ensures that the completion time of each machine is at most c. However, we must solve this problem for each multi-get request, 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, as solving an Integer Linear Program is a costly operation that would be usable for a single multi-get request, but not for a continuous stream to treat in real-time. We need instead a polynomial, guaranteed (even if not optimal), and ideally 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. In order to find such an efficient algorithm, we will work from the fact that the formulated scheduling problem corresponds to the well-known RESTRICTED ASSIGNMENT problem, which we describe in the following section.

4.2.2 The Restricted Assignment Problem

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 $J = \{1, \cdots, n\}$ and a set of m machines $M = \{1, \cdots, m\}$, 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, i.e., 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 is called 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 noted $P\,|\,\mathcal{M}_j\,|\,C_{\max}$ in Graham's notation.

The $R\,||\,C_{\max}$ problem, and more specifically the RA problem, are well-known $\mathbf{NP}$-hard problems, and it has even been proved that no algorithm can approximate an optimal solution within a factor better than $3/2$ unless $\mathbf{P} = \mathbf{NP}$ [72]. Hence, specific cases of the RA problem have also been the subject of extensive research. As shown in the previous chapter, one possible manner to reduce the complexity of the problem is to bring structure in the processing sets of jobs. In this paper, we focus on *interval* processing sets, that is a particular case of the RA problem where the machines can be rearranged such that the processing sets of jobs consist in contiguous intervals of machines. More formally, let us note $\langle a, b\rangle$ the interval[1] ranging from machine a (inclusive) to machine b (inclusive) such that $a \le b$, and $I_{\langle a,b\rangle} = \{a, a+1, \cdots, b\}$. In the Restricted Assignment problem on Intervals (RAI), for all jobs $j \in J$, we define $\mathcal{M}_j = I_{\langle a_j,b_j\rangle}$, 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.

The RAI problem, which is noted $P\,|\,\mathcal{M}_j(\mathit{interval})\,|\,C_{\max}$ in Graham's classification, is still NP-hard (as it is a generalization of $P\,||\,C_{\max}$). However, several guarantees can be obtained. In the following sections, we study slightly simpler variants, such as the case with unitary jobs, which corresponds to near-identical requests in the key-value store. This may happen in some homogeneous workloads where all requested data items have similar sizes. Another variant is the case with K types of jobs, which corresponds to a discrete categorization of the requests, for example by considering *small* and *large* data items.

4.3 Algorithms for the Restricted Assignment Problem on Intervals

Let us focus on the standard RAI problem for now. Lin et al. [76] proposed a polynomial algorithm to solve this 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 $\langle\alpha, \beta\rangle$, where $1 \le \alpha \le \beta \le m$, we define $K_{\langle\alpha,\beta\rangle}$ as the set of jobs whose processing set is included in $I_{\langle\alpha,\beta\rangle}$, i.e., $K_{\langle\alpha,\beta\rangle} = \left\{j \in J \text{ s.t. } \mathcal{M}_j \subseteq I_{\langle\alpha,\beta\rangle}\right\}$. We denote the total processing time of

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

Algorithm 9 ELFJ

```
1: sort jobs in non-decreasing order of $b_j$
2: for each machine $i$ do
3:     $\delta_i \leftarrow 0$
4:     for each non-assigned job $j$ such that $a_j \leq i \leq b_j$ do
5:         if $\delta_i + p_j \leq \lambda$ then
6:             assign $j$ to $i$
7:             $\delta_i \leftarrow \delta_i + p_j$
```

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 ($\tilde{w}_{\max} = \max_{1\leq\alpha\leq\beta\leq m}\left\{\tilde{w}_{\langle\alpha,\beta\rangle}\right\}$). From these definitions, we can easily derive a lower bound on the optimal makespan $C^{\text{OPT}}_{\max}$ for a given instance $\mathcal{I}$ of the RAI problem, as shown by Lin et al. [76] in their original work. We give two versions of the lemma (the first one holds in the general case and is used for the approximation algorithm, whereas the other one holds only when processing times are integers and is used for the optimal algorithm).

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

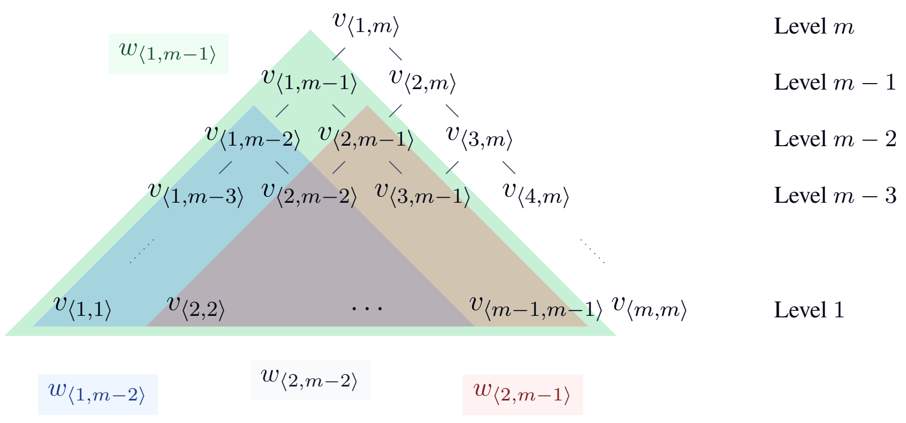
Proof: Let $C^{\text{OPT}}_{\langle\alpha,\beta\rangle}$ be the maximum completion time of machines $\alpha, \cdots, \beta$ in an optimal schedule π^{OPT}. We clearly have $C^{\text{OPT}}_{\langle\alpha,\beta\rangle} \geq \tilde{w}_{\langle\alpha,\beta\rangle}$ for any interval $\langle\alpha,\beta\rangle$,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

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

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

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

The idea of Lin et al. [76] 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 intervals $\langle\alpha,\beta\rangle$, $w_{\langle\alpha,\beta\rangle} = \left|K_{\langle\alpha,\beta\rangle}\right|$. In the original paper, the authors propose the following procedure. First, for all machines 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 $\langle\alpha,\beta\rangle$, compute $\left|K_{\langle\alpha,\beta\rangle}\right| = |A_\alpha \cap B_\beta|$. 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.



$v_{\langle 1,m\rangle}$
Level m
$w_{\langle 1,m-1\rangle}$
$v_{\langle 1,m-1\rangle}$
$v_{\langle 2,m\rangle}$
Level $m-1$
$v_{\langle 1,m-2\rangle}$
$v_{\langle 2,m-1\rangle}$
$v_{\langle 3,m\rangle}$
Level $m-2$
$v_{\langle 1,m-3\rangle}$
$v_{\langle 2,m-2\rangle}$
$v_{\langle 3,m-1\rangle}$
$v_{\langle 4,m\rangle}$
Level $m-3$
$v_{\langle 1,1\rangle}$
$v_{\langle 2,2\rangle}$
$\cdots$
$v_{\langle m-1,m-1\rangle}$
$v_{\langle m,m\rangle}$
Level 1
$w_{\langle 1,m-2\rangle}$
$w_{\langle 2,m-2\rangle}$
$w_{\langle 2,m-1\rangle}$

Figure 4.3 – Interval hierarchy represented as a lattice graph. Each node represents an interval $\langle x, y\rangle$ and is labeled with the value $v_{\langle x,y\rangle}$. Nodes are organized by level, where nodes on level h represent intervals of size h, e.g., if $m = 3$, the node on level m is the interval of size 3, nodes on level $m-1$ are intervals of size 2, and nodes on level $m-2$ are intervals of size 1.

Instead, we present 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 $\langle \alpha+1, \beta\rangle$ (see Figure 4.3). Let $J_{\langle\alpha,\beta\rangle}$ be the set of jobs whose processing set is exactly $I_{\langle\alpha,\beta\rangle}$, i.e., $J_{\langle\alpha,\beta\rangle} = \left\{ j \in J \text{ s.t. } \mathcal{M}_j = I_{\langle\alpha,\beta\rangle} \right\}$, and let $v_{\langle\alpha,\beta\rangle}$ be their total processing time. Then 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 that must be done on machines $\alpha, \cdots, \beta$ includes

(i) the work $J_{\langle\alpha,\beta\rangle}$,

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

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

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

Then, for any machines 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 particular case $w_{\langle\alpha,\beta\rangle} = 0$ if $\alpha > \beta$. The values $v_{\langle\alpha,\beta\rangle}$ can be pre-computed in time $O(n)$ by scanning jobs linearly, and the computation of the 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 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. [76], although we found their demonstration to be incorrect. We know from Lemma 4.2 that the optimal makespan is at least λ. Thus we seek to prove that ELFJ gives a schedule whose makespan is at most λ.

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

```
 1: $\tilde{w}_{\max} \leftarrow 0$
 2: for each $0 \leq \alpha \leq \beta \leq m$ do
 3:     $v_{\langle \alpha,\beta \rangle} \leftarrow 0$
 4: for each job $j$ do
 5:     $v_{\langle a_j,b_j \rangle} \leftarrow v_{\langle a_j,b_j \rangle} + p_j$
 6: for all $l$ from 0 to $m-1$ do
 7:     for all $a$ from 1 to $m-l$ do
 8:         $b \leftarrow a + l$
 9:         $w_{\langle a,b \rangle} \leftarrow v_{\langle a,b \rangle} + w_{\langle a,b-1 \rangle} + w_{\langle a+1,b \rangle} - w_{\langle a+1,b-1 \rangle}$
10:         $\tilde{w}_{\langle a,b \rangle} \leftarrow \frac{w_{\langle a,b \rangle}}{b-a+1}$
11:         if $\tilde{w}_{\langle a,b \rangle} > \tilde{w}_{\max}$ then
12:             $\tilde{w}_{\max} \leftarrow \tilde{w}_{\langle a,b \rangle}$
```

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 $\alpha, \alpha+1, \cdots, \beta$ come from the set $K_{\langle \alpha,\beta \rangle}$. In other words, the processing set of each job assigned on these machines is included in $I_{\langle \alpha,\beta \rangle}$. Proving this property leads to the conclusion $\lambda < \tilde{w}_{\langle \alpha,\beta \rangle}$, 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 $\alpha, \alpha+1, \cdots, \beta+1$ (call these jobs the *filling* jobs). The job j must be done in the interval $\alpha+1, \alpha+2, \cdots, \beta$. 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 $\alpha, \alpha+1, \cdots, \beta$ come from $K_{\langle \alpha,\beta \rangle}$.

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 $K_{\langle \alpha,\beta \rangle}$. However, α is more carefully chosen in this new version: we start from the interval $\langle a_j, b_j \rangle$ 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 $O(m^2 + n \log n + mn)$, where m is the number of machines and n is the number of jobs.*

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 $\alpha, \alpha+1, \cdots, \beta$ come from the set $K_{\langle \alpha,\beta \rangle}$. 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 included in $I_{\langle\alpha,\beta\rangle}$. 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 $a_{j_1}, a_{j_1}+1, \cdots, a_{j_0}-1$, 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) $j_0 \in K_{\langle\alpha,\beta\rangle}$,

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

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

Then we have

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

i.e., $\lambda < \tilde{w}_{\langle\alpha,\beta\rangle}$, which is a contradiction. Hence, ELFJ gives a schedule feasible in λ time units, which means that $C_{\max}^{\text{OPT}} \leq \lambda$. By Lemma 4.2, we also know that $C_{\max}^{\text{OPT}} \geq \lambda$. We conclude that $C_{\max}^{\text{OPT}} = \lambda$, 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 $p_{\max}$).

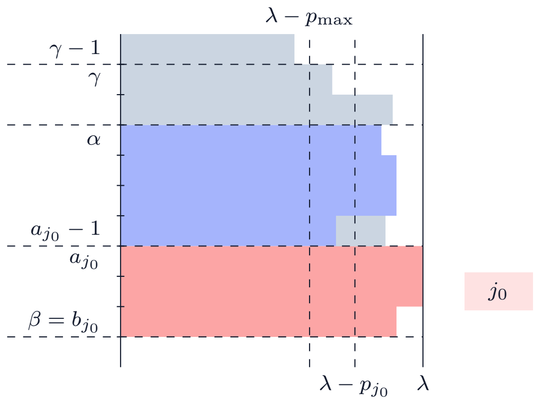


Figure 4.4 – Work areas between machines α and β. The blue area is $S(\alpha, a_{j_0} - 1, \lambda - p_{\max})$. The red area is $S(a_{j_0}, \beta, \lambda - p_{j_0})$. Gray areas are the other jobs. We seek to prove that the blue and red areas are made of jobs included in $K_{\langle\alpha,\beta\rangle}$.

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 $K_{\langle\alpha,\beta\rangle}$, which includes all jobs whose processing set is in the interval $\langle\alpha, \beta\rangle$. Figure 4.4 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 $K_{\langle u_x, \beta \rangle}$.

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

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

which gives the following inequality:

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

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

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

thus,

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

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,

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

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 $C_{\max}^{\text{OPT}} \geq \tilde{w}_{\max}$, and obviously, $C_{\max}^{\text{OPT}} \geq p_{\max}$, so $\lambda \leq (2 - 1/m) C_{\max}^{\text{OPT}}$. We conclude that $C_{\max} \leq (2 - 1/m) C_{\max}^{\text{OPT}}$.

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. ■

4.4 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. We formally introduce these circular intervals in Section 4.4.1. Then, we present a general procedure to solve the RAI problem with circular intervals (Section 4.4.2), and we give two examples of applications of this procedure in Sections 4.4.3 and 4.4.4.

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, distributed key-value stores often organize machines in a virtual ring, where the machines able to answer a query for a particular key are

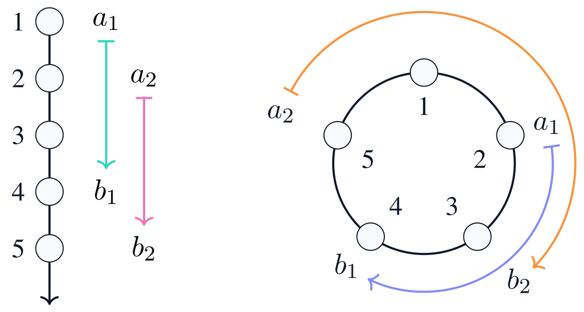


Figure 4.5 – Comparison between instances of the standard RAI problem (on the left) and its generalization to circular intervals (on the right).

consecutively arranged in this ring. We generalize here the notion of interval to take into account this setting by introducing *circular* intervals. Machines are now virtually arranged on a circle. In addition to regular intervals $\langle a, b\rangle$ (with $a \leq b$), we introduce circular intervals such that $a > b$. In this case, the corresponding set $I_{\langle a,b\rangle}$ includes machines $a, a+1, \cdots, m$ and machines $1, 2, \cdots, b$, i.e., we have

$$I_{\langle a,b\rangle} = \begin{cases} \{a, a+1, \cdots, b\} & \text{if } a \leq b, \\ \{1, 2, \cdots, b\} \cup \{a, a+1, \cdots, m\} & \text{otherwise.} \end{cases}$$

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 4.5 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 $\langle a_g, b_g\rangle$* precedes *the interval $\langle a_h, b_h\rangle$ if and only if $a_g \leq a_h$ and $b_g \leq b_h$. In this case, 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 ($Z^* = \{\langle a_j, b_j\rangle$ 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 $p(k)$.

4.4.2 An Optimal Procedure for K Job Types

We introduce in this section 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 4.5. *Let $\mathcal{A}$ be an optimal algorithm for the RAI problem with K job types that runs in time $O(f(n))$. 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.

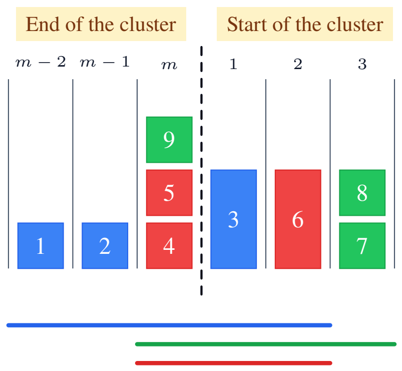


Figure 4.6 – Example of circular jobs in a schedule. Colors denote jobs with common processing sets. Jobs 1, 2, 4, 5, 9 are left jobs, whereas jobs 3, 6, 7, 8 are right jobs. Moreover, there are two types of jobs. Jobs 1, 2, 4, 5, 7, 8, 9 are of type 1 (with $p(1) = 1$), whereas jobs 3 and 6 are of type 2 (with $p(2) = 2$). Thus, in this example, $G_1 = \{1, 2, 4, 5, 9\}$, $D_1 = \{7, 8\}$, $G_2 = \emptyset$, and $D_2 = \{3, 6\}$.

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 $J^*_1, \cdots, J^*_K$, 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 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 π 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 4.6 shows an example of such a 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 4.6. 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 π is* right-sorted *if and only if for each type 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 $j' \in D_k$.*

We denote the set of all possible schedules for a given instance $\mathcal{I}$ by $\Pi(\mathcal{I})$ ($\mathcal{I}$ 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 $\Pi_\mathbf{r}$ the subset of schedules Π 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 $C^{\text{OPT}}_{\max}$ denotes the optimal makespan among all schedules Π. We define analogously $C^{\text{BEST}}_\mathbf{r}$ as the *best possible makespan* among schedules $\Pi_{\mathbf{r}}$. Note that the subsets $\Pi_{\mathbf{r}}$ define a partition of Π, and thus

$$C_{\max}^{\mathrm{OPT}} = \min_{\mathbf{r} \in \mathbf{R}^K} \left\{ C_{\mathbf{r}}^{\mathrm{BEST}} \right\}. \tag{4.2}$$

Optimal procedure. We introduce a polynomial function $\phi_{\mathbf{r}}$ 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. In a nutshell, this function chooses, for each type k, the r_k jobs to put on the right. We prove later the following two statements 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 $C_{\mathbf{r}}^{\mathrm{BEST}}$.

Given these statements and Equation (4.2), 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 $\Pi_{\mathbf{r}}^{\preceq}$ be the subset of schedules $\Pi_{\mathbf{r}}$ that are right-sorted. The proof of the two statements 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 $\Pi(\phi_{\mathbf{r}}(\mathcal{I}))$. Hence, we will prove two lemmas. On the one hand, we show in Lemma 4.6 that the set $\Pi(\phi_{\mathbf{r}}(\mathcal{I}))$ is exactly the same as the set of right-sorted schedules $\Pi_{\mathbf{r}}^{\preceq}(\mathcal{I})$ 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* $\Pi(\phi_{\mathbf{r}}(\mathcal{I})) = \Pi_{\mathbf{r}}^{\preceq}(\mathcal{I})$.

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

Now we show that $\Pi_{\mathbf{r}}^{\preceq}(\mathcal{I}) \subseteq \Pi(\phi_{\mathbf{r}}(\mathcal{I}))$. By definition of $\Pi_{\mathbf{r}}^{\preceq}$, in any schedule $\pi \in \Pi_{\mathbf{r}}^{\preceq}(\mathcal{I})$, for all types k, we have $\langle a_j, b_j \rangle \preceq \langle a_{j'}, b_{j'} \rangle$ 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, π is clearly a valid solution for $\phi_{\mathbf{r}}(\mathcal{I})$ and belongs to $\Pi(\phi_{\mathbf{r}}(\mathcal{I}))$. ■

Lemma 4.7. *For any* $\mathbf{r} \in \mathbf{R}^K$*, there exists a right-sorted schedule* $\pi \in \Pi_{\mathbf{r}}^{\preceq}$ *that has the best possible makespan* $C_{\mathbf{r}}^{\mathrm{BEST}}$.

Proof: Let $\mathbf{r}$ be an arbitrary vector of $\mathbf{R}^K$. Let $\pi \in \Pi_{\mathbf{r}}$ be a schedule that has the best possible makespan $C_{\mathbf{r}}^{\mathrm{BEST}}$. If π 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 π, are not sorted according to $\preceq$, i.e., we have $\langle a_j, b_j \rangle \npreceq \langle a_{j'}, b_{j'} \rangle$. 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 π, 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 π without changing the makespan of π.

By repeatedly swapping non-sorted jobs of the same type, we reach another schedule π' that has the same makespan than π, that also belongs to $\Pi_{\mathbf{r}}$, 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 $\Pi(\phi_{\mathbf{r}}(\mathcal{I}))$. By Lemma 4.7, we also know that there exists at least one schedule in $\Pi_{\mathbf{r}}^{\preceq}(\mathcal{I})$ that has the best possible makespan $C_{\mathbf{r}}^{\text{BEST}}$. Therefore, we deduce by Lemma 4.6 that:

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

It follows that, for any instance $\mathcal{I}$, we can find the best possible schedule among $\Pi_{\mathbf{r}}(\mathcal{I})$ for any vector $\mathbf{r} \in \mathbf{R}^K$. 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 an 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.

4.4.3 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. [100] 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 $1, 2, \cdots, i$ 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) $\sum_{k=1}^{K} (s_k - s'_k) p(k) \leq \lambda$.

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 $O(f(n))$ is the time complexity of $\mathcal{A}$. Therefore, the time complexity of the derived algorithm is $O(mn^{3K} \log \sum p_j)$.

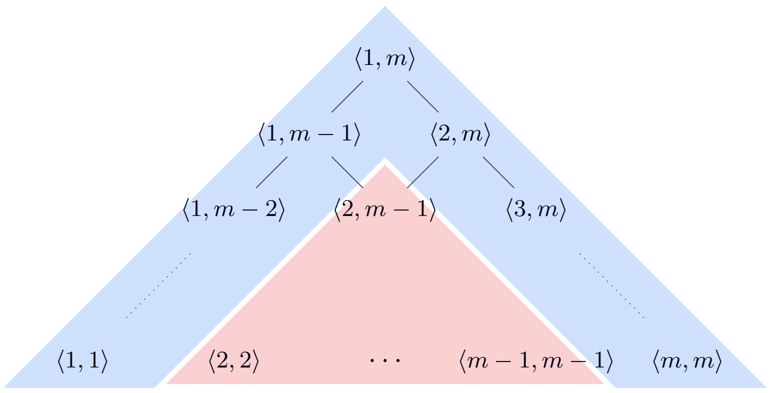


Figure 4.7 – Outside and inside intervals in the lattice graph representation. Outside intervals (blue area) are of the form $\langle 1, x\rangle$ or $\langle x, m\rangle$, with $1 \leq x \leq m$, and inside intervals (red area) are of the form $\langle x, y\rangle$, with $1 < x \leq y < m$.

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 $O(f(n)) = O(m^2 + n\log n + mn)$. 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 $O(nf(n)) = O(m^2 n + n^2 \log n + mn^2)$. 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 $O(n(m^2 + n) + n \log n + mn) = O(m^2 n + n^2)$.

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 $\langle a, b\rangle$ in two subintervals $\langle a, m\rangle$ and $\langle 1, b\rangle$, 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 $\langle 1, x\rangle$ or $\langle x, m\rangle$, with $1 \leq x \leq m$, and inside intervals are all non-circular intervals of the form $\langle x, y\rangle$ 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 4.7.

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

```
1: sort circular jobs by decreasing order of $\preceq$
2: transform circular jobs as left jobs and pre-compute $w$, $\tilde{w}_{\max}^{\textit{inside}}$ and $\tilde{w}_{\max}$
3: $r_{cur} \leftarrow 0$
4: for each circular job $j \in J^*$ do
5:     $\tilde{w}_{\max}^{\textit{outside}} \leftarrow \tilde{w}_{\langle 1,m \rangle}$
6:     for each $\beta$ from $b_j$ to $m-1$ do                    ▷ Add $j$ on the right
7:         $w_{\langle 1,\beta \rangle} \leftarrow w_{\langle 1,\beta \rangle} + 1$
8:         $\tilde{w}_{\langle 1,\beta \rangle} \leftarrow \frac{w_{\langle 1,\beta \rangle}}{\beta}$
9:         if $\tilde{w}_{\langle 1,\beta \rangle} > \tilde{w}_{\max}^{\textit{outside}}$ then
10:            $\tilde{w}_{\max}^{\textit{outside}} \leftarrow \tilde{w}_{\langle 1,\beta \rangle}$
11:    for each $\alpha$ from 2 to $a_j$ do                        ▷ Remove $j$ from the left
12:        $w_{\langle \alpha,m \rangle} \leftarrow w_{\langle \alpha,m \rangle} - 1$
13:        $\tilde{w}_{\langle \alpha,m \rangle} \leftarrow \frac{w_{\langle \alpha,m \rangle}}{m-\alpha+1}$
14:        if $\tilde{w}_{\langle \alpha,m \rangle} > \tilde{w}_{\max}^{\textit{outside}}$ then
15:            $\tilde{w}_{\max}^{\textit{outside}} \leftarrow \tilde{w}_{\langle \alpha,m \rangle}$
16:    $r_{cur} \leftarrow r_{cur} + 1$
17:    $\tilde{w}_{cur} \leftarrow \max\left(\tilde{w}_{\max}^{\textit{inside}}, \tilde{w}_{\max}^{\textit{outside}}\right)$
18:    if $\tilde{w}_{cur} < \tilde{w}_{\max}$ then                ▷ Update minimum makespan
19:        $r \leftarrow r_{cur}$
20:        $\tilde{w}_{\max} \leftarrow \tilde{w}_{cur}$
```

Recall that $w_{\langle \alpha,\beta \rangle}$ represents the total work of all non-circular jobs whose interval is included in $\langle \alpha, \beta \rangle$. 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 $\langle 1, b_j \rangle$, and if it is a left job, we keep the left part of the interval, i.e., the subinterval $\langle a_j, m \rangle$. 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

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

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

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

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

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

In other words, each time we update our guess on r, we only need to recompute the value of $\tilde{w}_{\max}^{\textit{outside}}$, 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 $\max\left(\tilde{w}_{\max}^{\textit{inside}}, \tilde{w}_{\max}^{\textit{outside}} \right)$ by pre-computing $\tilde{w}_{\max}^{\textit{inside}}$, and then trying each possible value of r by updating only $\tilde{w}_{\max}^{\textit{outside}}$.

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 $w_{\langle 1,m\rangle}$, as it is always exactly equal to n, and we have $\tilde{w}_{\langle 1,m\rangle} = n/m$. 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, $\tilde{w}^{inside}_{\max}$ 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 $w_{\langle 1,\beta\rangle}$ for all $b_j \leq \beta < m$ and we subtract 1 to $w_{\langle \alpha,m\rangle}$ 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 chapter, starting from the applicative context of multi-get requests in key-value stores, 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, and 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 $O(m^2 + n \log n + mn)$.

The next and final chapter of this thesis is dedicated to a scheduling framework that we implemented in a real key-value store system. After the theoretical approach, we will now treat the problem of scheduling in key-value store from a more practical and experimental point of view, partly based on the lessons learned from the previous chapters.

5
Implementing and Evaluating Scheduling Strategies

5.1 Introduction

On the practical side, several strategies have been proposed in the literature to improve request scheduling in distributed and persistent key-value stores. However, we observe that authors of these proposals often make different assumptions on the workload and the execution environment, and that the proposed solutions are often implemented in different versions of the system. This prevents any fair comparison between results of different publications. Moreover, code artifacts, as well as the used hardware/software configurations, are almost never made available, which raises a reproducibility concern. The major difficulty comes from the fact that, even if the proper scheduling of requests in distributed key-value stores has been demonstrated to unlock performance improvements, most industrial systems do not provide much configuration on this aspect, and do not allow users to easily switch between various policies. In this chapter, we propose a solution to this problem by introducing Hector, a framework extending the industry-standard key-value store Apache Cassandra, to enable experimenters to implement and evaluate a large variety of scheduling algorithms through several modules designed from principles learned in the literature and previous chapters of this thesis.

In Section 5.2, we detail the common mechanisms of request scheduling in key-value stores through the example of Apache Cassandra. We identify several challenges related to practical scheduling in these systems, such as the difficulty to gather useful information on the cluster and the workload. To address these problems, we present Hector in Section 5.3, an extension of Apache Cassandra that provides modular components to implement, integrate and evaluate new scheduling policies without having to dive in the complex code base. After describing the various features of Hector, we give some examples of possible implementations in Section 5.4. Finally, we assess in Section 5.5 that Hector does not introduce any significant overhead by itself compared to Apache Cassandra, and we illustrate how it can be used to compare different solutions under various assumptions on the workload and the execution environment.

5.2 Scheduling in Persistent Key-Value Stores

In Section 5.2.1, we give a technical overview of request scheduling in the popular key-value store Apache Cassandra. We then discuss in Section 5.2.2 the challenges that arise when trying to implement, evaluate and compare scheduling policies in this kind of distributed system.

5.2.1 Overview of Apache Cassandra

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 dynamically include more machines if the workload requires it. As explained in Chapter 1, in distributed key-value stores, keys are dispatched on servers using a partitioning scheme based on consistent hashing. The same hash function is used across the cluster, which makes each server able to 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, persistent 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 [83]. 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. For a write request, this means storing data in the LSM tree. For a read request, this means retrieving data from disk or memory and sending it back to the client.

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 chapter, we focus on read-dominated workloads to properly highlight the effects of

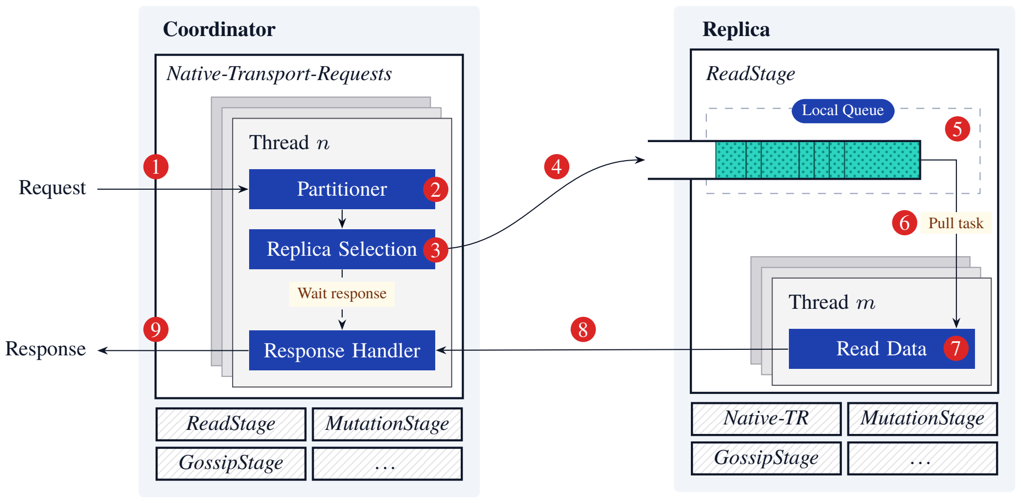


Figure 5.1 – Scheduling of a read request in Apache Cassandra.

scheduling on the overall performance. We set the consistency level of read requests to 1, which means that only one replica is chosen to execute a given request, which is arguably the most common setting for such workloads.

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) [101]. 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, as part of the 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*. Although this step is not really exploited in Apache Cassandra, we highlighted in Chapter 2 that it may be of particular importance in request scheduling. Note that, for a given request, the coordinator and the replica may sometimes be the same server. Figure 5.1 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 of Scheduling in Apache Cassandra

Various scheduling strategies have been proposed in the literature to improve the overall performance of Apache Cassandra [97, 54, 89, 55, 58]. 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, the more representative the score of the server health, and therefore the better the scheduling decisions. 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 [97].

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 [5].

We also identify a reproducibility concern, coming from the lack of a common baseline on which strategies may be properly compared [7]. 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 the Scheduling Framework Hector

In an effort to address these challenges, we unify the scheduling-related components of Apache Cassandra (version 4.2) into a coherent framework called Hector[1]. This framework is simple enough to let any user implement new scheduling strategies without knowing the details of the entire codebase. In fact, the API simply consists of a set of general interfaces. The user implements these interfaces and specifies with which parameters they must be loaded using a configuration file. Then, Hector takes the responsibility of instantiating the components in the correct order and connecting them together when starting the system. Among these components, we also introduce additional 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.

Note that Hector is a scheduling-centric framework, and as such, it does not modify the behavior of other components of Apache Cassandra. In other words, the orthogonal mechanisms that are access control, recovery from failures, or dynamic scale-in/scale-out of the cluster are not impacted by our framework. For instance, we ensured that Hector did not prevent the automatic reconfiguration of replicas when adding or removing a machine from the cluster.

However, to avoid diving in too much complexity in the rest of this chapter, we focus on the case of a single-datacenter cluster of static size, where all servers are geographically located in the same place, linked by a high-performance local network, and without situations of scale-in/scale-out reconfiguration.

We now introduce the main components of Hector.

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 5.2. 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.

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

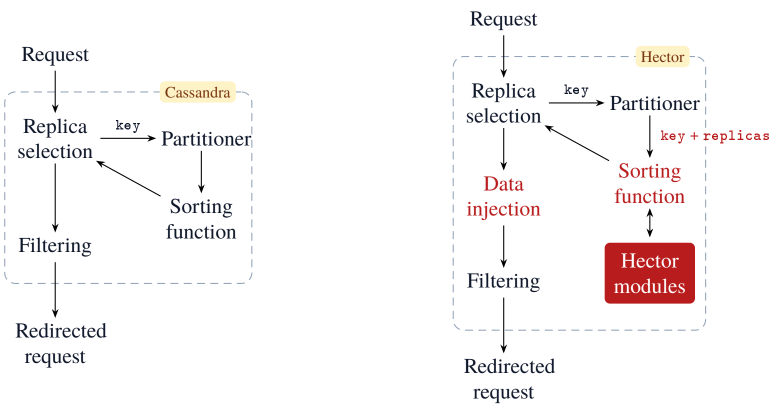


Figure 5.2 – Replica selection on coordinators in Apache Cassandra (left) and Hector (right). Additional features of Hector are highlighted in red.

Local scheduling. As explained in Section 5.2, Apache Cassandra 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.

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 taking 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 [97]. 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, without overloading the network.

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 5.3. 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

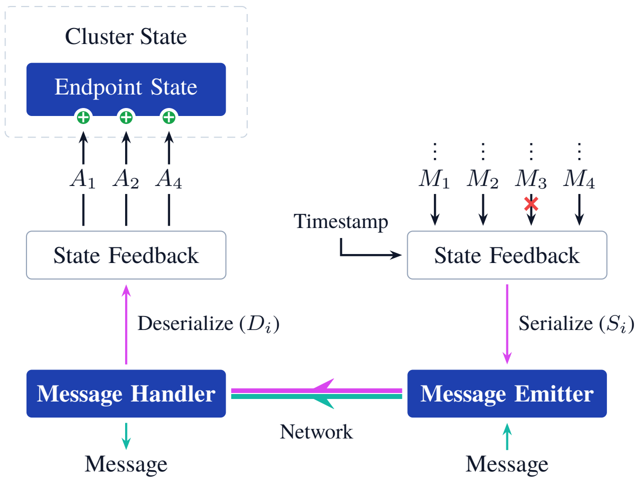


Figure 5.3 – State propagation module of Hector. M_i, A_i, S_i, and D_i are respectively the measurement component, the aggregation component, the serializer, and the deserializer of fact i. The message emitter and message handler represent any internode communication, e.g., a periodic broadcast or a request response. The user may filter the transmitted values. In this example, the fact 3 is not included in the packet.

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 [54], 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 in Hector

We demonstrate in this section the flexibility of Hector components by implementing state-of-the-art policies such as Dynamic Snitching [41] and C3 [97], and we show that it is also easy to test new ideas by introducing two novel replica selection/local scheduling strategies that we name *Popularity-Aware* and *Random Multi-Level*.

5.4.1 Replica Selection

We present 3 replica selection strategies: Dynamic Snitching and C3, which are existing and well-tested policies, and Popularity-Aware, which is a new policy that we introduce in this work.

Dynamic Snitching. This is the default replica selection strategy in Apache Cassandra [41]. 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 [97]. 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 64-bit integer and incrementing the corresponding access count in a map. Instead of directly using the key string (which may be long), hashing limits the memory footprint of the data structure, at a small cost on the precision of the distribution in case of collision. For example, recording accesses to a dataset that comprises 1 million keys requires at most 16 MB 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 one 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.

5.4.2 Local Scheduling

Now we present 2 local scheduling strategies: First-Come First-Served, which is the default policy in most key-value stores, and Random Multi-Level, which is a new policy that we introduce in this work.

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 [89], 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. We define the priority of a given operation entering a sub-queue q to be 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, based on the fact that avoiding putting requests for small values behind requests for large values may have a positive effect on performance, as shown in Chapter 2, 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 [54]).

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 requests with various assumptions on the workload and the environment.

5.5.1 Platform and Configuration

We run all experiments on the large-scale experiment platform Grid'5000 [10]. 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 [33]. 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 [82]. 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.

5.5.2 Results

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 (also 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

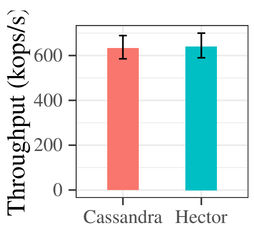


Figure 5.4 – Maximum attainable throughput ($\times 10^3$ operations per second) for Apache Cassandra and Hector. Keys are accessed according to a Zipf law with parameter 0.9 and data item sizes range from 1 to 100 kB. The replica selection strategy is Dynamic Snitching and the local scheduling policy is First-Come First-Served.

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

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

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 5.4 shows the maximum attainable throughput in each system while Figure 5.5 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 (in average 1.15% higher) than Apache Cassandra. The error bars 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

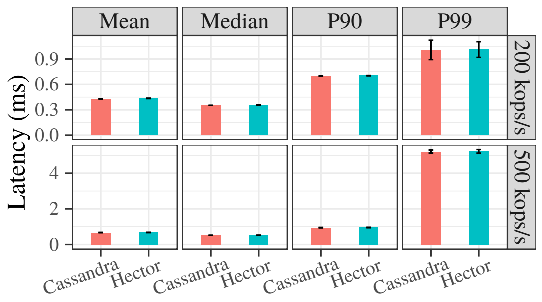


Figure 5.5 – Latency (milliseconds) for Apache Cassandra and Hector, with two different arrival rates (200 000 and 500 000 operations per second).

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

Apache Cassandra; C3, which is a proposal coming from the literature [97]; and Popularity-Aware (PA), which is a new algorithm that we propose in this paper (see Section 5.4). 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 5.6 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

[2]Sorted Strings Table (SSTable) is a format that maps data to keys, and these keys are kept sorted. This is the standard

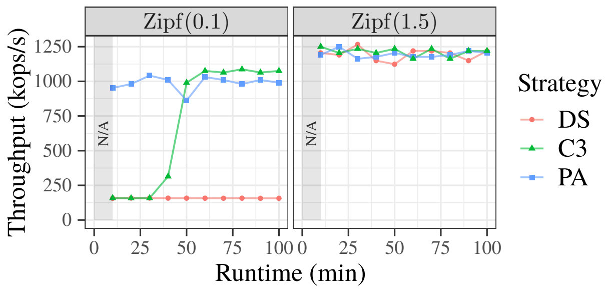


Figure 5.6 – Maximum attainable throughput ($\times 10^3$ operations per second) over time for each replica selection strategy under two key popularity distributions. Higher is better. $\mathrm{Zipf}(0.1)$ corresponds to a quasi-uniform distribution, whereas $\mathrm{Zipf}(1.5)$ corresponds to a heavily-skewed, right-tail distribution. Each data item has a fixed size of 1 kB.

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

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

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 5.7. 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 5.4, First-Come First-Served (FCFS) and Random Multi-Level (RML), under the

solution to store data in persistent key-value stores.

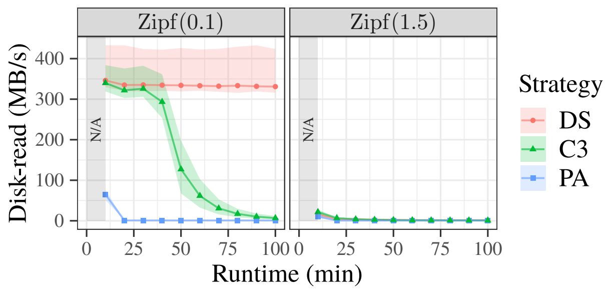


Figure 5.7 – Amount of data read on-disk (megabytes per second) over time for each replica selection strategy under two key popularity distributions. Lower is better.

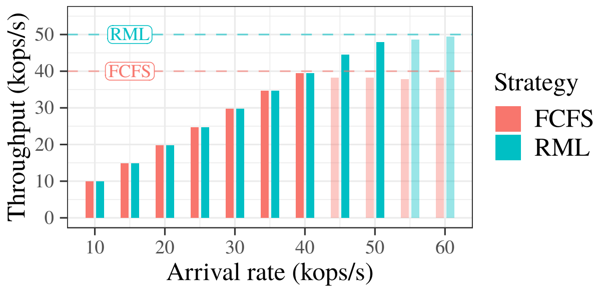


Figure 5.8 – Attainable throughput ($\times 10^3$ operations per second) as a function of the arrival rate for each local scheduling strategy. Higher is better. Keys are accessed according to a uniform distribution. Data item sizes are small (1 kB, 75% of data items) or large (1 MB, 25% of data items). Servers have limited parallelism for read operations (up to 4 threads). Light bars indicate that the system saturates.

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 5.8 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 5.9 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 5.10 compares the average latency of requests for small and large items. When 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

In this chapter, we proposed Hector, a framework built on top of Apache Cassandra to ease the implementation and evaluation of scheduling policies. Hector also constitutes a stable baseline to properly compare the performance of different strategies with identical assumptions. For instance, we were able

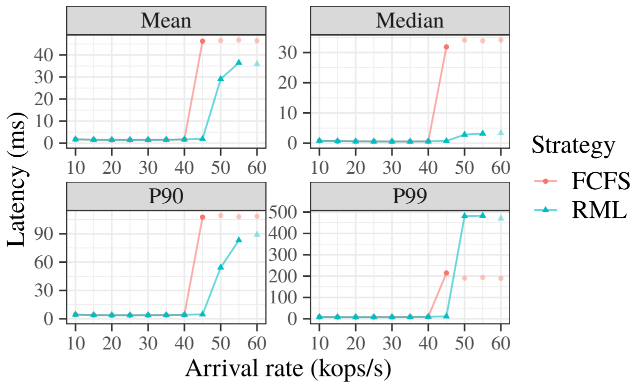


Figure 5.9 – Latency (milliseconds) as a function of arrival rate for each local scheduling strategy. Lower is better.

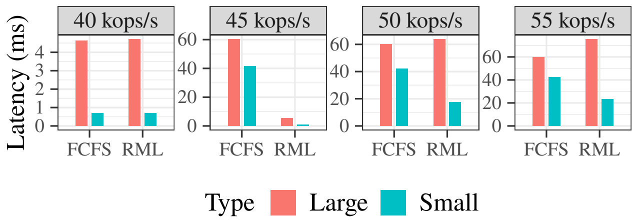


Figure 5.10 – Average latency of requests for large (left bars) and small (right bars) items, for both strategies.

to rebuild previously-proposed algorithms in Hector, and compare their performance to two novel algorithms, called Popularity-Aware and Random Multi-Level, under specific hypotheses on the workload and the environment. When key popularity is moderately biased, we found that PA is able to achieve a throughput that is more than 6 times higher than the maximum throughput of Dynamic Snitching, and converges faster than C3. In the case of limited parallelism and heterogeneous dataset, RML handles the load better than FCFS and keeps a low median latency (under 5 ms), even when the system starts to saturate. Moreover, we show that the various components introduced in Hector do not bring additional overhead on the overall performance of the system.

Conclusion

In this thesis, we studied scheduling in key-value stores with the objectives to (i) understand better their theoretical limits, (ii) equip researchers with tools to evaluate them both formally and practically, and (iii) provide guidings for future optimizations. We summarize our main contributions below.

Theoretical bounds on metrics (Chapters 2 and 3). The first step in establishing guarantees was to define a formal scheduling framework for key-value stores, which we modeled in Chapter 2 as a non-preemptive scheduling problem with processing set restrictions (representing replication), release times (representing a random stream of requests), and heterogeneous processing times (representing non-identical data items), with the objective of minimizing the maximum weighted flow time. We then studied the complexity of the offline problem by relaxing some constraints, in order to identify the most challenging ones. In particular, we showed that, although the preemptive problem has been shown to be tractable [70], the non-migratory problem remains $\mathbf{NP}$-hard. Restricting processing times to be identical also makes the problem easier, as we were able to derive optimal and efficient algorithms for such cases. Moreover, processing set restrictions clearly constitute one of the most challenging aspects of the problem. In Chapter 3, we studied the impact of replication on the theoretical guarantees that may be derived from the online version of the problem. Although the case of arbitrary processing sets has been shown to admit a strong lower bound in $\Omega\left(m\right)$ on the competitive ratio [3], we refined the result by bringing structures, matching better the replication schemes in actual key-value stores. We established bounds on the response time for three different classes of algorithms (online algorithms, immediate dispatch algorithms, and EFT algorithms), and highlighted the impact of the replication strategy on the guarantees that may be obtained. A particularly meaningful result is that the EFT algorithm, which turned out to be a good heuristic when empirically evaluated in Chapter 2, may or may not be guaranteed within an acceptable factor of the optimal solution, depending on the way fixed-size replication intervals are defined.

Formal and practical evaluation tools (Chapters 2, 3 and 5). In addition to formal guarantees, we proposed several tools to evaluate various aspects of scheduling in key-value stores. On the theoretical side, we provided an exact method, described in Chapter 3, to study the maximum attainable throughput of a key-value store under skewed key accesses and for a given replication scheme. We were able to show that, in theory, overlapping replication intervals may enable a throughput almost 50% higher than the one obtained with non-overlapping intervals. In Chapter 2, we built a discrete-event simulator of a key-value store, with the objective of comparing scheduling heuristics from an empirical point of view, under different assumptions on the workload. Although being simplified compared to real-world systems, the simulator is able to capture the general behavior of a given heuristic, in order to evaluate ideas before implementing them in a real system, and also constitute a bridge between the theoretical and practical aspects of scheduling, as the full control over the simulator enables testing conditions that cannot be easily obtained in an experimental environment. A good illustration of this is the comparison of several heuristics over a common lower bound in Chapter 2. Finally, based on our observations on existing proposals and from the previous chapters, we proposed in Chapter 5 a scheduling-centric framework called Hector, which extends the industry-standard key-value store Apache Cassandra, with the objective of facilitating the implementation and the evaluation of new scheduling strategies. We illustrated the flexibility of our framework by implementing several policies, both coming from the literature or original ones, and we showed that Hector does not bring any significant overhead compared to the original system.

Guidings for future optimizations (Chapters 2, 4 and 5). We provided a number of guidings throughout the thesis for future optimizations of key-value stores. In the simulations performed in Chapter 2, we exhibited the efficiency of EFT as a heuristic when mitigating the tail latency, and we showed that local scheduling may also have a significant impact on performance, especially when considering the stretch metric. This guided us to implement a local scheduling module in Hector (Chapter 5), and we validated the idea by showing that our policy RML is able to significantly increase the throughput of the system and diminish the latency of requests under certain conditions. Moreover, considering the presence of skewed key accesses (similarly to Chapter 3), we proposed the PA replica selection policy, which leverages the page cache of the operating system to mitigate the impact of costly disk-reads. Finally, Chapter 4 may be seen as a full set of guidings for the optimization of multi-get requests in key-value stores. We studied the partitioning of such requests with the objective to optimize the load balancing. Starting from the Restricted Assignment problem, we extended an existing result of the literature on the specific case of interval processing sets to derive low-cost optimal and approximation algorithms. Going one step further, we generalized the problem by introducing circular intervals to match the actual replication scheme of key-value stores, and we gave a general procedure to find an optimal solution when jobs may be categorized into K classes according to their processing times. In particular, we showed that the optimal result that we obtained in the non-circular case may be adapted to the circular case without increasing the time complexity of the algorithm.

Short-Term Perspectives

In the short-term, we plan to extend the results and tools presented in each chapter:

- In Chapter 2, the lower bound derived from the literature presents no formal guarantee on its tightness. We only tested its quality empirically, but it is still unknown if the difference between its value and the optimal solution is small enough for any instance. Moreover, it would deserve more investigations to find a possibly closer lower bound. On the practical side, although our key-value store simulator turned out to be already a useful tool for evaluating scheduling heuristics, it would be interesting to extend it to support more features, such as dynamic reconfigurations in case of failures or scale-in/scale-out operations, multi-get requests, temporal patterns, etc.

- In Chapter 3, it remains unknown if there exists a replication scheme that permits reaching a high throughput and for which we may find a competitive algorithm with respect to the response time. A possible approach could be to consider the space of all possible replication schemes for a given number of machines m, and systematically apply the exact method we developed to find the maximum attainable throughput. Of course, this represents a huge number of combinations (in the order of 2^{2^m}), even for small m, but it could be possible to set high-level restrictions, e.g., limiting the subsets of machines to a common fixed size. Once the best schemes are found, competitive analysis could be performed to derive the corresponding bounds on the response time.

- In Chapter 4, we saw that there exists an optimal algorithm for the RAI problem with unitary jobs, and a $(2 - 1/m)$-approximation algorithm for the general case. An interesting intermediary case is the problem with two classes of jobs. Key-value store workloads often exhibit long-tailed distributions, with a majority of short requests and some long ones. It has been proven that this problem is in $\mathbf{P}$, although the algorithm is very difficult to implement. We conjecture that there exists simpler polynomial-time algorithms for slightly relaxed versions of the problem (e.g., when intervals can overlap only a limited number of times). Moreover, it would also be interesting to find a better approximation ratio than $2 - 1/m$ in the arbitrary case, analogously to the MAKESPAN problem.

- In Chapter 5, we presented Hector, which features several experimental modules for various aspects of scheduling. We plan to extend the framework with multi-get capabilities, in order to be able to implement partitioning policies and evaluate them in a realistic environment. In particular, this would permit testing our guaranteed algorithms from Chapter 4. Furthermore, Hector could be used to perform a systematic and exhaustive study of scheduling policy combinations under various workloads, in order to find the best strategies for a given use-case.

Long-Term Perspectives

In the long-term, various directions could be explored. We detail here the most promising ones to our opinion, in the sense that they could lead to general results that may be applied to other systems than key-value stores.

Scheduling problems constrained by SLOs. Nowadays, many system providers offer so-called Service-Level Objectives (SLOs) to their customers, which consist in guaranteeing a certain level of Quality of Service (QoS) according to a given performance measure (e.g., availability, response time, throughput, etc.). In this thesis, we focused on the minimization of the maximum response time of requests, which corresponds to a very specific SLO, i.e., if the maximum response time of a schedule is $F_{\max}$, then it is guaranteed that each job will be completed within $F_{\max}$ time units. Generally speaking, a SLO on the response time is defined as following: "for a given proportion α of jobs, the response time is guaranteed to be less than a given threshold τ". Then, we may define the corresponding parameterized scheduling problem as follows. We fix the deadline $d_j = r_j + \tau$ for all jobs (or $d_j = r_j + \tau/w_j$ in the weighted case), and we denote by $\sum U_j$ the number of late jobs in a given schedule. The objective becomes minimizing τ under the constraint that $\sum U_j < (1 - \alpha)n$, where n is the total number of jobs. We expect the case $\alpha < 1$ to be easier than the case $\alpha = 1$ treated in this manuscript.

Problem classification in online analysis. Different classes of online algorithms (immediate dispatch, clairvoyant, randomized, etc.) seem to have different properties according to the considered problem. For instance, we saw in Chapter 3 that optimal or guaranteed immediate dispatch algorithms may exist for a particular problem, while it can be proven that there are no such algorithm for another given problem. Can we classify problems according to the classes of online algorithms that they admit, and can we derive general results on this classification? At first sight, the answer seems to be positive: for a given problem, a negative result for the general class of online algorithms necessarily applies to the more restricted class of immediate dispatch algorithms. Similarly, a positive result for the class of non-clairvoyant algorithms necessarily applies to the class of clairvoyant algorithms. Much less obvious relationships could exist between problems in the online framework.

DSL for scheduling in replicated systems. The general principles of scheduling in key-value stores that we used to build Hector in Chapter 5 could be taken one step further, in order to be applied more generally to distributed systems that replicate their data. The idea would be to abstract these principles and define a Domain-Specific Language (DSL), which would adapt to any compatible system through specific bindings. The DSL would be in charge to translate the high-level scheduling policies into low-level implementations for the target system. With a sufficient level of abstraction, Hector modules (replication selection, local scheduling, state propagation and workload oracle) could be applied to any system that replicate data on several nodes and need to perform specific operations on these data. Key-value stores would consitute only a subset of theses systems, where the operations of interest are simple read and write operations.

Bibliography

[1] Daniel Abadi. “Consistency tradeoffs in modern distributed database system design: CAP is only part of the story”. In: *Computer* 45.2 (2012), pp. 37–42.

[2] Christoph Ambühl and Monaldo Mastrolilli. “On-line scheduling to minimize max flow time: an optimal preemptive algorithm”. In: *Operations Research Letters* 33.6 (2005), pp. 597–602.

[3] S Anand, Karl Bringmann, Tobias Friedrich, Naveen Garg, and Amit Kumar. “Minimizing maximum (weighted) flow-time on related and unrelated machines”. In: *Algorithmica* 77.2 (2017), pp. 515–536.

[4] Esmail Asyabi, Azer Bestavros, Erfan Sharafzadeh, and Timothy Zhu. “Peafowl: In-application cpu scheduling to reduce power consumption of in-memory key-value stores”. In: *Proceedings of the 11th ACM Symposium on Cloud Computing*. 2020, pp. 150–164.

[5] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. “Workload analysis of a large-scale key-value store”. In: *ACM SIGMETRICS Performance Evaluation Review* 40.1 (2012), pp. 53–64.

[6] Baruch Awerbuch, Yossi Azar, Stefano Leonardi, and Oded Regev. “Minimizing the flow time without migration”. In: *SIAM Journal on Computing* 31.5 (2002), pp. 1370–1382.

[7] Vaibhav Bajpai, Mirja Kühlewind, Jörg Ott, Jürgen Schönwälder, Anna Sperotto, and Brian Trammell. “Challenges with reproducibility”. In: *Proceedings of the Reproducibility Workshop*. 2017, pp. 1–4.

[8] Kenneth R Baker. *Introduction to sequencing and scheduling*. John Wiley & Sons, 1974.

[9] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. “SILK+ Preventing Latency Spikes in Log-Structured Merge Key-Value Stores Running Heterogeneous Workloads”. In: *ACM Transactions on Computer Systems* 36.4 (2020), pp. 1–27.

[10] Daniel Balouek, Alexandra Carpen Amarie, Ghislain Charrier, Frédéric Desprez, Emmanuel Jeannot, Emmanuel Jeanvoine, Adrien Lèbre, David Margery, Nicolas Niclausse, Lucas Nussbaum, et al. “Adding virtualization capabilities to the Grid’5000 testbed”. In: *International Conference on Cloud Computing and Services Science*. Springer. 2012, pp. 3–20.

[11] Nikhil Bansal. “Minimizing flow time on a constant number of machines with preemption”. In: *Operations Research Letters* 33.3 (2005), pp. 267–273.

[12] Nikhil Bansal and Bouke Cloostermans. “Minimizing maximum flow-time on related machines”. In: *Theory of Computing* 12.1 (2016), pp. 1–14.

[13] Nikhil Bansal and Kedar Dhamdhere. “Minimizing weighted flow time”. In: *ACM Transactions on Algorithms* 3.4 (2007), p. 39.

[14] Nikhil Bansal and Janardhan Kulkarni. “Minimizing flow-time on unrelated machines”. In: *Proceedings of the forty-seventh annual ACM symposium on Theory of Computing*. 2015, pp. 851–860.

[15] Philippe Baptiste. “Scheduling equal-length jobs on identical parallel machines”. In: *Discrete Applied Mathematics* 103.1-3 (2000), pp. 21–32.

[16] Philippe Baptiste, Peter Brucker, Marek Chrobak, Christoph Dürr, Svetlana A Kravchenko, and Francis Sourd. “The complexity of mean flow time scheduling problems with release times”. In: *Journal of Scheduling* 10.2 (2007), pp. 139–146.

[17] Luca Becchetti and Stefano Leonardi. “Nonclairvoyant scheduling to minimize the total flow time on single and parallel machines”. In: *Journal of the ACM* 51.4 (2004), pp. 517–539.

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

[19] 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.

[20] Anne Benoit, Redouane Elghazi, and Yves Robert. “Max-stretch minimization on an edge-cloud platform”. In: *2021 IEEE International Parallel and Distributed Processing Symposium*. 2021, pp. 766–775.

[21] Péter Biró and Eric McDermid. “Matching with sizes (or scheduling with processing set restrictions)”. In: *Discrete Applied Mathematics* 164 (2014), pp. 61–67.

[22] David Rex Britton and Steve L Lloyd. “How to deal with petabytes of data: the LHC Grid project”. In: *Reports on Progress in Physics* 77.6 (2014), p. 065902.

[23] 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.

[24] 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.

[25] James Bruno, Edward G Coffman Jr, and Ravi Sethi. “Scheduling independent tasks to reduce mean finishing time”. In: *Communications of the ACM* 17.7 (1974), pp. 382–387.

[26] Jake Brutlag. *Speed matters for Google web search*. 2009.

[27] Zhao Cao, Shimin Chen, Feifei Li, Min Wang, and X Sean Wang. “LogKV: Exploiting key-value stores for event log processing”. In: *Proc. Conf. Innovative Data Syst. Res*. 2013.

[28] Josiah L Carlson. *Redis in action*. Manning Publications Co., 2013.

[29] *Cassandra Availability with Virtual Nodes*. https://jolynch.github.io/pdf/cassandra-availability-virtual.pdf. Accessed on 2023-06-11.

[30] *Cassandra Documentation. Production recommendations*. https://cassandra.apache.org/doc/4.1/cassandra/getting_started/production.html. Accessed on 2023-06-11.

[31] Denis M. Cavalcante, Victor A. E. de Farias, Flávio R. C. Sousa, Manoel Rui P. Paula, Javam C. Machado, and José Neuman de Souza. “PopRing: A Popularity-aware Replica Placement for Distributed Key-Value Store”. In: *CLOSER 2018*. 2018, pp. 440–447.

[32] 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.

[33] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. "Benchmarking cloud serving systems with YCSB". In: *Proceedings of the 1st ACM symposium on Cloud computing*. 2010, pp. 143–154.

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

[35] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. "Dynamo: amazon's highly available key-value store". In: *ACM SIGOPS operating systems review* 41.6 (2007), pp. 205–220.

[36] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. "Hawk: Hybrid datacenter scheduling". In: *2015 USENIX Annual Technical Conference*. 2015, pp. 499–510.

[37] 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.

[38] Salvatore Dipietro, Giuliano Casale, and Giuseppe Serazzi. "A queueing network model for performance prediction of apache cassandra". In: (2016).

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

[40] Pierre-François Dutot, Erik Saule, Abhinav Srivastav, and Denis Trystram. "Online non-preemptive scheduling to optimize max stretch on a single machine". In: *Computing and Combinatorics - 22nd International Conference*. Vol. 9797. Lecture Notes in Computer Science. 2016, pp. 483–495.

[41] *Dynamic snitching in Cassandra: past, present, and future*. `https://www.datastax.com/blog/dynamic-snitching-cassandra-past-present-and-future`. Accessed on 2022-11-10. 2012.

[42] Tomás Ebenlendr, Marek Krcál, and Jiri Sgall. "Graph balancing: a special case of scheduling unrelated parallel machines." In: *SODA*. Vol. 8. 2008, pp. 483–490.

[43] Leah Epstein and Asaf Levin. "Scheduling with processing set restrictions: PTAS results for several variants". In: *International Journal of Production Economics* 133.2 (2011), pp. 586–595.

[44] Ulrich Faigle, Walter Kern, and György Turán. "On the performance of on-line algorithms for partition problems". In: *Acta cybernetica* 9.2 (1989), pp. 107–119.

[45] Dietrich Featherston. "Cassandra: Principles and application". In: *Department of Computer Science UIUC* (2010).

[46] Dror G Feitelson. *Workload modeling for computer systems performance evaluation*. Cambridge University Press, 2015.

[47] Andrea Gandini, Marco Gribaudo, William J Knottenbelt, Rasha Osman, and Pietro Piazzolla. "Performance evaluation of NoSQL databases". In: *Computer Performance Engineering: 11th European Workshop, EPEW 2014, Florence, Italy, September 11-12, 2014. Proceedings 11*. Springer. 2014, pp. 16–29.

[48] 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.

[49] Celia A Glass and Hans Kellerer. “Parallel machine scheduling with job assignment restrictions”. In: *Naval Research Logistics* 54.3 (2007), pp. 250–257.

[50] Ronald L Graham. “Bounds for certain multiprocessing anomalies”. In: *Bell System Technical Journal* 45.9 (1966), pp. 1563–1581.

[51] Ronald Lewis Graham, Eugene Leighton Lawler, Jan Karel Lenstra, and AHG Rinnooy Kan. “Optimization and approximation in deterministic sequencing and scheduling: a survey”. In: *Annals of discrete mathematics*. Vol. 5. 1979, pp. 287–326.

[52] Leslie A Hall. “A note on generalizing the maximum lateness criterion for scheduling”. In: *Discrete applied mathematics* 47.2 (1993), pp. 129–137.

[53] Xiangdong Huang, Jianmin Wang, Jialin Qiao, Liangfan Zheng, Jinrui Zhang, and Raymond K Wong. “Performance and replica consistency simulation for quorum-based NoSQL system cassandra”. In: *Application and Theory of Petri Nets and Concurrency: 38th International Conference, PETRI NETS 2017, Zaragoza, Spain, June 25–30, 2017, Proceedings 38*. Springer. 2017, pp. 78–98.

[54] Vikas Jaiman, Sonia Ben Mokhtar, Vivien Quéma, Lydia Y Chen, and Etienne Rivière. “Héron: Taming Tail Latencies in Key-Value Stores under Heterogeneous Workloads”. In: *37th Symposium on Reliable Distributed Systems*. SRDS. IEEE. 2018, pp. 191–200.

[55] 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.

[56] Klaus Jansen and Lars Rohwedder. “A Quasi-Polynomial Approximation for the Restricted Assignment Problem”. In: *SIAM Journal on Computing* 49.6 (2020), pp. 1083–1108.

[57] Klaus Jansen and Lars Rohwedder. “Structured Instances of Restricted Assignment with Two Processing Times”. In: *Conference on Algorithms and Discrete Applied Mathematics*. 2017, pp. 230–241.

[58] Wanchun Jiang, Yujia Qiu, Fa Ji, Yongjia Zhang, Xiangqian Zhou, and Jianxin Wang. “AMS: Adaptive Multiget Scheduling Algorithm for Distributed Key-Value Stores”. In: *IEEE Transactions on Cloud Computing* (2022).

[59] Jithin Jose, Hari Subramoni, Miao Luo, Minjia Zhang, Jian Huang, Md Wasi-ur-Rahman, Nusrat S Islam, Xiangyong Ouyang, Hao Wang, Sayantan Sur, et al. “Memcached design on high performance rdma capable interconnects”. In: *2011 International Conference on Parallel Processing*. IEEE. 2011, pp. 743–752.

[60] Flora Karniavoura and Kostas Magoutis. “A measurement-based approach to performance prediction in NoSQL systems”. In: *2017 IEEE 25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE. 2017, pp. 255–262.

[61] 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.

[62] John Klein, Ian Gorton, Neil Ernst, Patrick Donohoe, Kim Pham, and Chrisjan Matser. “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.

[63] Rusty Klophaus. “Riak core: Building distributed applications without shared state”. In: *ACM SIGPLAN Commercial Users of Functional Programming*. 2010, pp. 1–1.

[64] 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.

[65] Jacques Labetoulle, Eugene L Lawler, Jan Karel Lenstra, and AHG Rinnooy Kan. “Preemptive scheduling of uniform machines subject to release dates”. In: *Progress in combinatorial optimization*. 1984, pp. 245–261.

[66] Avinash Lakshman and Prashant Malik. “Cassandra: a decentralized structured storage system”. In: *ACM SIGOPS Operating Systems Review* 44.2 (2010), pp. 35–40.

[67] 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.

[68] Neal Leavitt. “Will NoSQL databases live up to their promise?” In: *Computer* 43.2 (2010), pp. 12–14.

[69] Kangbok Lee, Joseph Y.-T. Leung, and Michael L. Pinedo. “Makespan minimization in online scheduling with machine eligibility”. In: *Annals of Operations Research* 204.1 (2013), pp. 189–222.

[70] 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.

[71] 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.

[72] Jan Karel Lenstra, David B Shmoys, and Éva Tardos. “Approximation algorithms for scheduling unrelated parallel machines”. In: *Mathematical programming* 46.1 (1990), pp. 259–271.

[73] 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.

[74] 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.

[75] 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.

[76] Yixun Lin and Wenhua Li. “Parallel machine scheduling of machine-dependent jobs with unit-length”. In: *European Journal of Operational Research* 156.1 (2004), pp. 261–266.

[77] Si Liu, Son Nguyen, Jatin Ganhotra, Muntasir Raihan Rahman, Indranil Gupta, and José Meseguer. “Quantitative analysis of consistency in NoSQL key-value stores”. In: *Quantitative Evaluation of Systems: 12th International Conference, QEST 2015, Madrid, Spain, September 1-3, 2015, Proceedings 12*. Springer. 2015, pp. 228–243.

[78] Marten Maack and Klaus Jansen. “Inapproximability results for scheduling with interval and resource restrictions”. In: *arXiv preprint arXiv:1907.03526* (2019).

[79] Muthucumaru Maheswaran, Shoukat Ali, Howard Jay Siegel, Debra Hensgen, and Richard F Freund. “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.

[80] 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.

[81] Shanmugavelayutham Muthukrishnan, Rajmohan Rajaraman, Anthony Shaheen, and Johannes E Gehrke. “Online scheduling to minimize average stretch”. In: *40th Annual Symposium on Foundations of Computer Science*. IEEE. 1999, pp. 433–443.

[82] *NoSQLBench Docs*. `https://docs.nosqlbench.io`. Accessed on 2022-12-11.

[83] Patrick O’Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O’Neil. “The log-structured merge-tree (LSM-tree)”. In: *Acta Informatica* 33 (1996), pp. 351–385.

[84] Rasha Osman and Pietro Piazzolla. “Modelling replication in nosql datastores”. In: *Quantitative Evaluation of Systems: 11th International Conference, QEST 2014, Florence, Italy, September 8-10, 2014. Proceedings 11*. Springer. 2014, pp. 194–209.

[85] *Our World in Data. Historical cost of computer memory and storage*. `https://ourworldindata.org/grapher/historical-cost-of-computer-memory-and-storage`. Accessed on 2023-06-20.

[86] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. “Tucana: Design and implementation of a fast and efficient scale-up key-value store”. In: *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. 2016, pp. 537–550.

[87] Antonis Papaioannou and Kostas Magoutis. “Addressing the read-performance impact of reconfigurations in replicated key-value stores”. In: *IEEE Transactions on Parallel and Distributed Systems* (2021).

[88] Michael L Pinedo. *Scheduling*. Vol. 29. 2012.

[89] Waleed Reda, Marco Canini, Lalith Suresh, Dejan Kostić, and Sean Braithwaite. “Rein: Taming tail latency in key-value stores via multiget scheduling”. In: *12th European Conference on Computer Systems*. EuroSys. 2017, pp. 95–110.

[90] Erik Saule, Doruk Bozdağ, and Ümit V Çatalyürek. “Optimizing the stretch of independent tasks on a cluster: From sequential tasks to moldable tasks”. In: *Journal of Parallel and Distributed Computing* 72.4 (2012), pp. 489–503.

[91] Alexandru D Sicoe, Giovanna Lehmann Miotto, Luca Magnoni, Serguei Kolos, and Igor Soloviev. “A persistent back-end for the ATLAS TDAQ online information service (P-BEAST)”. In: *Journal of Physics: Conference Series*. Vol. 368. 1. 2012, p. 012002.

[92] 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.

[93] René Sitters. “Two NP-hardness results for preemptive minsum scheduling of unrelated parallel machines”. In: *International Conference on Integer Programming and Combinatorial Optimization*. Springer. 2001, pp. 396–405.

[94] Daniel D Sleator and Robert E Tarjan. “Amortized efficiency of list update and paging rules”. In: *Communications of the ACM* 28.2 (1985), pp. 202–208.

[95] Wayne E Smith. *Various optimizers for single-stage production*. Tech. rep. CALIFORNIA UNIV LOS ANGELES NUMERICAL ANALYSIS RESEARCH, 1955.

[96] Roshan Sumbaly, Jay Kreps, Lei Gao, Alex Feinberg, Chinmay Soman, and Sam Shah. “Serving large-scale batch computed data with project Voldemort”. In: *FAST 2012*. 2012, p. 18.

[97] Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. “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.

[98] *Velocity and the Bottom Line*. http://radar.oreilly.com/2009/07/velocity-making-your-site-fast.html. 2009.

[99] Ashish Vulimiri, Philip Brighten Godfrey, Radhika Mittal, Justine Sherry, Sylvia Ratnasamy, and Scott Shenker. “Low latency via redundancy”. In: *9th ACM conference on Emerging networking experiments and technologies*. CoNext. ACM. 2013, pp. 283–294.

[100] Chao Wang and René Sitters. “On some special cases of the restricted assignment problem”. In: *Information Processing Letters* 116.11 (2016), pp. 723–728.

[101] Matt Welsh, David E. Culler, and Eric A. Brewer. “SEDA: An Architecture for Well-Conditioned, Scalable Internet Services”. In: *Proceedings of the 18th ACM Symposium on Operating System Principles, SOSP*. 2001, pp. 230–243.

[102] Lili Wu, Roshan Sumbaly, Chris Riccomini, Gordon Koo, Hyung Jin Kim, Jay Kreps, and Sam Shah. “Avatara: OLAP for web-scale analytics products”. In: *Proceedings of the VLDB Endowment* 5.12 (2012), pp. 1874–1877.

[103] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. “LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data Items”. In: *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 2015, pp. 71–82.

[104] Zhe Wu, Curtis Yu, and Harsha V Madhyastha. “Costlo: Cost-effective redundancy for lower latency variance on cloud storage services”. In: *12th USENIX Symposium on Networked Systems Design and Implementation*. NSDI. 2015, pp. 543–557.

Publications

For each reference, authors appear in alphabetical order.

Articles in International Conferences

[C1] Louis-Claude Canon, Anthony Dugois, and Loris Marchal. "Bounding the Flow Time in Online Scheduling with Structured Processing Sets". In: *2022 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2022*. IEEE, 2022, pp. 683–693.

[C2] Louis-Claude Canon, Anthony Dugois, Loris Marchal, and Etienne Rivière. "Hector: A Framework to Design and Evaluate Scheduling Strategies in Persistent Key-Value Stores". In: *52nd International Conference on Parallel Processing, ICPP 2023*. To appear. 2023.

[C3] Sonia Ben Mokhtar, Louis-Claude Canon, Anthony Dugois, Loris Marchal, and Etienne Rivière. "Taming Tail Latency in Key-Value Stores: A Scheduling Perspective". In: *27th International Conference on Parallel and Distributed Computing, Euro-Par 2021*. Vol. 12820. Lecture Notes in Computer Science. Springer, 2021, pp. 136–150.

Research Reports

[R1] Louis-Claude Canon, Anthony Dugois, and Loris Marchal. *Bounding the Flow Time in Online Scheduling with Structured Processing Sets (extended version)*. Research Report RR-9446. Inria, 2022, pp. 1–35.

[R2] Sonia Ben Mokhtar, Louis-Claude Canon, Anthony Dugois, Loris Marchal, and Etienne Rivière. *Taming Tail Latency in Key-Value Stores: a Scheduling Perspective (extended version)*. Research Report. Inria, 2021.

Preprints

[P1] Louis-Claude Canon, Anthony Dugois, and Loris Marchal. "Low-Cost Algorithms for the Restricted Assignment Problem on Intervals of Machines". Planned to be submitted to IPDPS 2024.

[P2] Louis-Claude Canon, Anthony Dugois, Mohamad El Sayah, and Pierre-Cyrille Héam. "A Markov Chain Monte Carlo Approach to Cost Matrix Generation for Scheduling Performance Evaluation". Planned to be submitted to Future Generation Computer Systems.

[P3] Sonia Ben Mokhtar, Louis-Claude Canon, Anthony Dugois, Loris Marchal, and Etienne Rivière. "A Scheduling Framework for Distributed Key-Value Stores and its Application to Tail Latency Minimization". Under review in Journal of Scheduling.