Improving Performance Predictability in Cloud Data Stores

Vikas Jaiman

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTÉ UNIVERSITÉ GRENOBLE ALPES

Spécialité : **Informatique**

Arrêté ministériel : 25 mai 2016

Présentée par

Vikas JAIMAN

Thèse dirigée par **Vivien QUÉMA, Professeur, Grenoble INP,**
et coencadrée par **Sonia BEN MOKHTAR, Directrice de Recherche, CNRS**

préparée au sein du **Laboratoire d'Informatique de Grenoble (LIG)**
dans **l'École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique (ED MSTII)**

Improving Performance Predictability in Cloud Data Stores

Amélioration de la prédictibilité des performances pour les environnement de stockage de données dans les nuages

Thèse soutenue publiquement le **30 avril 2019**,
devant le jury composé de :

Monsieur Noël DE PALMA
Professeur, Université Grenoble Alpes, Président

Monsieur Laurent RÉVEILLÈRE
Professeur, Université de Bordeaux, Rapporteur

Monsieur Gaël THOMAS
Professeur, Telecom SudParis, Rapporteur

Monsieur Etienne RIVIÈRE
Professeur, UCLouvain, Examinateur

Monsieur Vivien QUÉMA
Professeur, Grenoble INP, Directeur de thèse

Madame Sonia BEN MOKHTAR
Directrice de Recherche, CNRS LIRIS, Co-Encadrant de thèse

To my parents.

Abstract

Today, users of interactive services such as e-commerce, web search have increasingly high expectations on the performance and responsiveness of these services. Indeed, studies have shown that a slow service (even for short periods of time) directly impacts the revenue. Enforcing predictable performance has thus been a priority of major service providers in the last decade. But avoiding latency variability in distributed storage systems is challenging since end user requests go through hundreds of servers and performance hiccups at any of these servers may inflate the observed latency. Even in well-provisioned systems, factors such as the contention on shared resources or the unbalanced load between servers affect the latencies of requests and in particular the tail (95th and 99th percentile) of their distribution.

The goal of this thesis to develop mechanisms for reducing latencies and achieve performance predictability in cloud data stores. One effective countermeasure for reducing tail latency in cloud data stores is to provide efficient replica selection algorithms. In replica selection, a request attempting to access a given piece of data (also called value) identified by a unique key is directed to the presumably best replica. However, under heterogeneous workloads, these algorithms lead to increased latencies for requests with a short execution time that get scheduled behind requests with large execution times. We propose Héron, a replica selection algorithm that supports workloads of heterogeneous request execution times. We evaluate Héron in a cluster of machines using a synthetic dataset inspired from the Facebook dataset as well as two real datasets from Flickr and WikiMedia. Our results show that Héron outperforms state-of-the-art algorithms by reducing both median and tail latency by up to 41%.

In the second contribution of the thesis, we focus on multiget workloads to reduce the latency in cloud data stores. The challenge is to estimate the bottleneck operations and schedule them on uncoordinated backend servers with minimal overhead. To reach this objective, we present TailX, a task aware multiget scheduling algorithm that reduces tail latencies under heterogeneous workloads. We implement TailX in Cassandra, a widely used key-value store. The result is an improved overall performance of the cloud data stores for a wide variety of heterogeneous workloads.

Keywords. Distributed storage, performance, scheduling.

Résumé

De nos jours, les utilisateurs de services interactifs comme le e-commerce, ou les moteurs de recherche, ont de grandes attentes sur la performance et la réactivité de ces services. En effet, les études ont montré que des lenteurs (même pendant une courte durée) impacte directement le chiffre d'affaire. Avoir des performances prédictives est donc devenu une priorité pour ces fournisseurs de services depuis une dizaine d'années. Mais empêcher la variabilité dans les systèmes de stockage distribué est un challenge car les requêtes des utilisateurs finaux transitent par des centaines de servers et les problèmes de performances engendrés par chacun de ces serveurs peuvent influencer sur la latence observée. Même dans les environnements correctement dimensionnés, des problèmes comme de la contention sur les ressources partagés ou un déséquilibre de charge entre les serveurs influent sur les latences des requêtes et en particulier sur la queue de leur distribution (95ème et 99ème centile).

L'objectif de cette thèse est de développer des mécanises permettant de réduire les latences et d'obtenir des performances prédictives dans les environnements de stockage de données dans les nuages. Une contre-mesure efficace pour réduire la latence de queue dans les environnements de stockage de données dans les nuages est de fournir des algorithmes efficaces pour la sélection de réplique. Dans la sélection de réplique, une requête tentant d'accéder à une information donnée (aussi appelé valeur) identifiée par une clé unique est dirigée vers la meilleure réplique présumée. Cependant, sous des charges de travail hétérogènes, ces algorithmes entraînent des latences accrues pour les requêtes ayant un court temps d'exécution et qui sont planifiées à la suite de requêtes ayant des long temps d'exécution. Nous proposons Héron, un algorithme de sélection de répliques qui gère des charges de travail avec des requêtes ayant un temps d'exécution hétérogène. Nous évaluons Héron dans un cluster de machines en utilisant un jeu de données synthétique inspiré du jeu de données de Facebook ainsi que deux jeux de données réels provenant de Flickr et WikiMedia. Nos résultats montrent que Héron surpasse les algorithmes de l'état de l'art en réduisant jusqu'à 41% la latence médiane et la latence de queue.

Dans la deuxième contribution de cette thèse, nous nous sommes concentrés sur les charges de travail multi-GET afin de réduire la latence dans les environnements de stockage de données dans les nuages Le défi consiste à estimer les opérations limitantes et à les planifier sur des serveurs non-coordonnés avec un minimum de surcoût. Pour

atteindre cet objectif, nous présentons TailX, un algorithme d'ordonnancement de tâches multi-GET qui réduit les temps de latence de queue sous des charges de travail hétérogènes. Nous implémentons TailX dans Cassandra, une base de données clé-valeur largement utilisée. Il en résulte une amélioration des performances globales des environnements de stockage de données dans les nuages pour une grande variété de charges de travail hétérogènes.

Mot clés. Stockage distribué, performance, planification.

Acknowledgements

I would like to express my gratitude towards my thesis supervisor, Prof. Vivien Quéma who has been incredibly helpful throughout the long process of developing the work in this thesis. I would like to thank him for his guidance, constant feedback and encouragement throughout this work. I have learned a lot from him that helped me to mature academically.

I would like to thank my co-supervisor Dr. Sonia Ben Mokhtar for mentoring me during my PhD. Even though we were not in the same city but our regular long Skype discussions to discuss research ideas encouraged me to work harder. She was always willing to hear my half baked ideas and provided excellent guidance and feedback which allowed me to explore new ideas.

A special thanks to Prof. Etienne Rivière for giving me the opportunity to work at UCLouvain, Belgium during my PhD and provided me with much needed guidance along the way. I really enjoyed the time spent over there. He was always willing to discuss new ideas. It helped me to put the thesis in better shape.

Prof. Noël de Palma, for having done me the honor of chairing the jury of this thesis. I would like to thank the other members of my committee as well: Laurent Réveillère and Prof. Gaël Thomas; for agreeing to serve as the reviewer of my thesis and the time and effort they have dedicated to the revision of this thesis. Their experience and valuable feedback significantly helped in improving the final version of this work.

I also extend my sincere thanks to Dr. Lydia Y. Chen for her valuable inputs during my research and collaboration on my paper. Her extensive knowledge of systems helped me to improve my work.

This thesis would not have happened without the financial support of the LIG, University Grenoble Alpes and UCLouvain, Belgium especially project fundings from OCCIWARE, EBSIS, and DIONASYS.

I would like to thank my ERODS team members: Thomas, Renaud, Vania, Nicolas, Didier, Ahmed, Amadou, Hugo, Jeremie, Jaafar, Matthieu, Vincent, Albin, Maha, Nicolas, Soulaimane and Christopher who provided me a very beautiful working atmosphere during my PhD. Our chit-chat during lunch and coffee breaks made my PhD life easier. Also, my colleagues from INGI team at UCLouvain, Belgium: Raziel and Sana; with whom I spent memorable time in Belgium.

I would like to thank Abhinav with whom I spent countless hours by discussing

research problems over the tea break. It was indeed an enjoyable time. My friends: Shweta, Seema, Mishra, Preeti, Siddartha, Karthik and Amrit without them PhD life would have turned different. They were beside me, encouraging me and make me feel like home.

Finally, I would like to thanks my parents for their endless love and constant support during all these years. Without them, it would have been an impossible task to complete.

Preface

This thesis presents the research conducted in the ERODS team at Laboratoire d'Informatique de Grenoble, to pursue a Ph.D. in Computer Science from the doctoral school "Mathématiques, Sciences et Technologies de l'Information, Informatique (ED MSTII)" of the Université Grenoble Alpes, France. My research activities have been supervised by Prof. Vivien Quéma (ERODS / Grenoble INP) and Dr. Sonia Ben Mokhtar (LIRIS / CNRS). Some of the works presented in this thesis have been done in collaboration with Prof. Etienne Rivière, INGI department of Université catholique de Louvain (UCLouvain), Belgium.

This thesis studies performance issues in key-value stores and provide algorithms for improving performance under heterogeneous workloads.

While the second part of the thesis is currently under submission, contributions related to first part led to the following publications:

- 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 Proceedings of the IEEE 37th Symposium on Reliable Distributed Systems (SRDS), Salvador, Brazil* Oct 2018, pp. 191–200.

Contents

List of Figures

List of Tables

1

Introduction

Contents

Today, users of online services such as e-commerce, web search have increasingly high expectations on the performance and responsiveness of these services [3, 4]. Services which give responses quickly are more fluid and natural to users than services which take time. A slow service directly impacts the revenue of its provider, even if the performance drop is only for a short period of time [5]. Amazon has reported a latency loss of 100ms causes loss of 1% in sales [6]. Similarly, Google reported that a slowdown from 100ms to 400ms in page load time for Google search results led to a decrease in user searches from 0.2% to 0.6% [4]. Therefore this underlines the fact that performance has a major impact on revenues. Enforcing *predictable* performance is a challenging task even for well-provisioned systems. End-user requests go through hundreds of servers. Performance hiccups at any of these servers may dramatically inflate the observed latency for some of these requests. For instance, measurements from a real Google service [5] running in a cluster of 2,000 servers show that if one in 100 user requests gets slow (e.g., has a 1 second latency) while handled by one server that is collecting responses from 100 other servers in parallel, then 63% of user requests will take more than one second to execute. This problem is commonly known as the *tail latency* problem where a fraction of latency measurements (such as 95th percentile, 99th percentile and more) incurred longest delay. For example, 99th percentile latency defines the smallest latency of the 1% of largest latency values. Several studies [4,5,7,8] show that in distributed environment latency distributions exhibit long-tail behaviors (95^{th}, 99^{th} percentile etc.) and they affect the performance of large-scale systems.

Managing tail latency is one of the primary challenges for large-scale Internet services. Today's modern data centers facing poor user experiences particularly interactive services such as social networks and web search due to long tail behaviors.

However, to keep the tail latency low and guaranteeing predictable latency is a very challenging task for service providers since the size of the system scales up when the overall use increases [5, 7]. It is very difficult and impractical to eliminate all sources of latency variability in a distributed environment where resources are being shared [8]. This includes global resource sharing, maintenance activities, queuing, power limits, garbage collection, energy management of latency variability which skewed the performance [5]. Several approaches have been proposed to reduce tail latency for the different components of large-scale distributed systems [2, 7, 9, 10, 11, 12, 13, 14, 15, 16]. They include techniques such as reissuing requests, using preferential resource allocation, leveraging parallelism for individual requests or sending redundant requests.

Of the services used for building cloud applications, storage plays a fundamental role for overall services tail latencies. Key-value stores are the dominant class of storage solutions in this context. They emerged as a fundamental building block for cloud applications to provide better scalability and performance in terms of microseconds of latency and throughput. Replica selection strategies and scheduling algorithms are often used to reduce tail latencies in key-value stores. We find that the impact of these algorithms have been overlooked while working under heterogeneous workloads. Therefore, we motivate our approach towards following directions: i) Improve the performance through proposing better replica selection algorithms under heterogeneous workloads, and ii) Proposing better scheduling strategies for multiget requests under heterogeneous workloads.

1.1 Thesis Objectives

In this thesis, we seek to improve the performance of cloud data stores that facilitates online services. In particular, we narrowed our scope towards improving the performance of key-value stores under heterogeneous workloads. Our objectives can be summarized to:

- ***Objective 1:*** *Towards designing replica selection algorithms to reduce tail latencies under heterogeneous workloads.*

  Replica selection algorithms are often used to reduce tail latencies in key-value stores. In a key-value store, each piece of data or value is replicated on multiple servers for fault-tolerance. Replica selection strategies can help reducing tail latency when the performance of these servers differ. Specifically, a request attempting to access the value for a given key can be directed to the presumably best replica, i.e. the one that is expected to serve the request with the smallest

latency. State-of-the-art replica selection algorithms that are Cassandra dynamic snitching [17] and C3 [1] have been designed for workloads where requests access values of the same size. An analysis of real-life key-value stores' workloads (e.g., of Facebook's Memcached deployment [18]) shows that this assumption does not hold, as user requests typically access values of sizes ranging from 1 KB to few MBs. We show in this thesis that under such workloads, the tail latency of a key-value store using these algorithms increases dramatically compared to a scenario where all values have the same size. The increase in tail latency ranges from $\times 10$ when clients access 1% of large values to $\times 126$ when this proportion reaches 20%.

The reason why existing algorithms do not perform well under heterogeneous workloads is that fast requests accessing small values can get stuck behind slow ones accessing large values. This can dramatically increase the latency of fast requests, a phenomenon known as *head-of-line-blocking* [5]. Selecting the best replica for a request, while preventing fast requests from being stuck behind slow ones, requires addressing two challenges. First, when a request for a key arrives at the entry-point of a key-value store, the size of the corresponding value is not known. A first challenge is thus to be able to predict this size based on the key with minimal operational overhead. Assuming that the size of values can be correctly estimated at request time, a second challenge is to be able to choose a replica that can prevent the head-of-line-blocking scenario. We present Héron, a replica selection algorithm that solves these two challenges.

- ***Objective 2:*** *Scheduling multiget requests in key-value stores under heterogeneous workloads.*

  We focus on the problem of performance bottlenecks in cloud data stores while scheduling multiget requests under heterogeneous workloads. In multiget requests, it is very difficult to deliver consistent latency for interactive services due to the varying degree of request fan-out. If the requests are with a larger fan-out, it is most likely that it would be affected by long tail [5, 19]. Therefore to serve high fan-out user requests or request asking for several keys data elements, multiple operations are batched together. A multiget request finishes when all of its operations complete. Therefore, the response time of a request depends on the response time of slowest operation in that multiget request. When the requests come, the key question is which dispatching policy should we use to route incoming requests on the servers? This policy is known as *task assignment policy*. By using better task assignment policy we aim to reduce median and tail latencies.

  Also, requests that are coming on the system have multiple operations with the varied number of keys and value sizes. Server exhibits high tail latencies for these workloads due to the different execution time of operations. Therefore,

it's very difficult to coordinate the operations which go on different servers and the problem becomes more difficult due to the fact that some operation has different value size. Thus the challenge is to schedule the operations in a way that synchronize the approximate execution time and complete all the operations at the same time. Our aim to design an efficient scheduling algorithm which can estimate the bottleneck operations and schedule them on uncoordinated backend servers in such a way that creates minimal overhead and reduce latencies at the tail. We present TailX, a task aware multiget scheduling algorithm that solve these challenges.

1.2 Contributions

The main contribution of this thesis is the design and implementation of Héron and TailX. Héron is a replica selection algorithm for key-value stores where replica diversity is available. Héron improve tail latency in key-value store under heterogeneous workloads. TailX is a task-aware multiget scheduling algorithm which reduces performance bottlenecks under heterogeneous workloads. Here is the summary of thesis contributions:

1. **Taming tail latency through replica selection in key-value stores:** We address the problem of tail latency under heterogeneous workloads in key-value stores. Replica selection algorithms are used to reduce tail latency in key-value stores. We highlight the challenges involved in designing replica selection algorithms. We present **Héron**, a replica selection algorithm that reduces tail latencies under heterogeneous workloads. We implement it as part of Cassandra, a widely used key-value store. In summary, this thesis makes the following contributions:

    - We expose the challenges involved in the design of replica selection algorithms under heterogeneous workloads.
    - We propose a novel replica selection algorithm, i.e., Héron, under heterogeneous workloads, implement it as part of Cassandra a widely used key-value store and assess its performance compared to C3 [1] and DS [20] on a real cluster of machines and with realistic workloads.
    - We evaluate Héron through extensive experiments. Results show that Héron reduces tail latency without sacrificing median latency in all the used YCSB workloads compared to both DS and C3.

2. **Task-aware Scheduling for Tail Latencies in Key-Value Stores:** We address the problem of performance bottlenecks in cloud datastores in the case of scheduling multiget requests under heterogeneous workloads. Multiget scheduling is used to achieve low tail latency. We present **TailX**, a task aware multiget scheduling algorithm that reduces tail latencies under heterogeneous workloads. Here are the design contributions of TailX:

- We expose the challenges involved in the design of task-aware scheduling algorithm under heterogeneous workloads.
- We propose a novel replica selection algorithm, i.e., TailX, under heterogeneous workloads, implement it as part of Cassandra a widely used key-value store and assess its performance compared to Rein [16] on a real cluster of machines and with realistic workloads.
- We evaluate TailX through extensive experiments. Results show that TailX reduces tail latency without sacrificing median latency in all the used YCSB workloads compared to Rein.

1.3 Thesis Organization

Chapter 2 discusses the background setting related to this thesis along with the details of the target problem. It describes the key parameters used in this thesis for improving performance in cloud data stores. It describes the steps involved for replica selection algorithm in key-value stores. Afterwards, it describes the execution steps of multiget scheduling in key-value stores.

Chapter 3 details the challenges of designing a replica selection algorithm in key-value stores under heterogeneous workloads. It defines the problem of tail latency in key-value stores by highlighting the potential of the replica selection algorithm. It presents the design and implementation of Héron, a replica selection algorithm and its key components. Further, it discusses the performance evaluation of Héron with variable configurations of a synthetic dataset and real workloads. Afterwards, it discusses the work related to this thesis. It discusses the replica selection algorithms which are used to reduce tail latency in cloud data stores and Geo-distributed systems. Also, it discusses existing work on reducing tail latency in cluster environments.

Chapter 4 focuses on multiget requests in key-value stores. It details the challenges of designing a multiget scheduling algorithm in key-value stores under heterogeneous workloads. Next, it discusses the work related to task-aware scheduling. It discusses the multiget scheduling in cloud data stores and describes the state-of-the-art algorithms in cloud data stores. Afterwards, it presents the design and implementation of TailX, a multiget scheduling algorithm and it's key components. Further, it discusses the performance evaluation of TailX with variable configurations of a synthetic dataset.

Chapter 5 concludes the thesis and outlines future work.

2

Background and Problem Definition

Contents

Cloud computing is a distributed computing paradigm which is based on the fundamental principle of "reusability of IT capabilities". In cloud computing, systems are connected in a private or public network to provide subscription-based services in terms of infrastructure, platform and softwares. It offers consumers a beneficial way to acquire and manage these IT resources in a distributed environment. Through this, cloud computing reduces the cost consumption of application hosting, computation, maintenance, elastic provisioning, content storage in a significant way. In 2018, a study on cloud computing [21], provides an interesting comparison between cloud hosting and enterprises. It says 77% of the enterprises have at least one application or a portion of their infrastructure in a cloud. Also, 76% of the enterprises seek to accelerate their

service delivery for their cloud apps and platforms. Therefore, cloud providers concentrate on better Quality of service (QoS) to provide good performance guarantees. QoS is achieved by many factors including flexibility, performance, availability, robustness, usability, security and many more. In this thesis, we are focusing on the performance of these services.

Of the services used for building cloud application, storage plays a fundamental role in overall services. Specifically, we focus to design an efficient cloud data stores that can provide better performance guarantees. Key-Value Stores are the dominant class of storage solutions in this context and are the focus of this thesis.

The remaining of this chapter is structured as follows. We first detail the key factors of key-value stores through which better performance can be achieved (§2.1). Next, we discuss the tail latency in key-value stores and detail the working of replica selection strategies and task aware scheduling (§2.2). Afterward, we discuss the heterogeneous workloads in key-value stores (§2.3). Finally we conclude the chapter (§2.4).

2.1 Key-value stores

Today key-value stores are very popular among storage solutions due to elasticity, scalability, and ease of deployment compared to traditional database systems. They emerged as an alternative to traditional database systems due to the limitations of predefined schema, scalability and data structures. In key-value stores, there is no schema constraint and they can handle unstructured data such as numbers, texts, audios, videos, email, images, XML and many more. Values are accessed via a key with a GET request. Therefore it is more flexible and an application has complete control over the value. Additionally, they provide a high throughput rate and have low latencies for get/put workloads.

2.1.1 Data model

A key-value store maintains a number of tables, each mapping a collection of keys to values of arbitrary size. A partitioning scheme distributes keys between servers as described in §2.1.2. To enforce fault tolerance, key-value stores use data replication which we will discuss in §2.1.4. To retrieve a value for a given key, a client sends a request to one of the nodes in the system, which then redirects this request to one of the servers holding a replica. The retrieved value depends on the consistency model followed by the key-value stores. It ranges from eventual consistency to serializability as discussed in §2.1.5.

2.1.2 Distributed hash table

Distributed Hash Table (DHT) is widely used by distributed storage systems. It's a structured peer to peer overlay that provides the look-up service for key-value pairs. It's

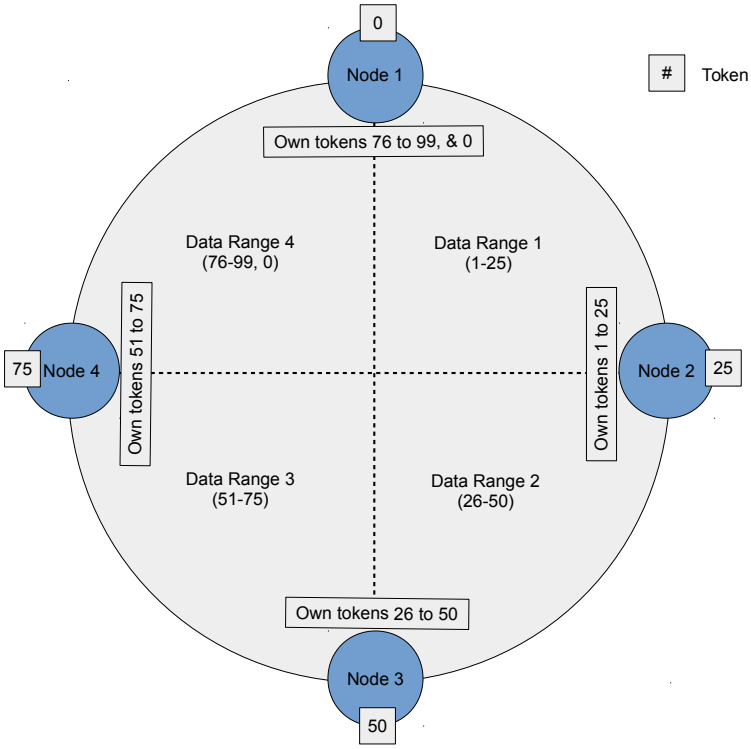


Figure 2.1 – Distributed hash table implementation.

similar to a hash table in which each participating node efficiently retrieves the value by given key. For data lookup, instead of connecting to the centralized server, DHT offers a scalable alternative which distributes lookup on a number of peers without the need for central coordination. This avoids the problem of the bottleneck at the central coordinating node.

Any read/write operation should locate the node containing that key. Each node has some range of keys along with the information of key range on other nodes. Whenever a request comes to a node, that node forwards the request to the nearest node containing that key according to the look up table. For example, in Cassandra, by doing internal gossip between nodes, each node eventually has state information of every other node. As seen in figure 2.1, whole token range is divided into number of nodes which are in a cluster. Each node holds a hash based token range which is well balanced among nodes. By default, MD5 hashing is used for even key distribution among nodes which leads to well-balanced storage distribution. Keys are converted into the hash-based token using a consistent hashing algorithm. When the request comes to a node, partitioner [22] converts the key into a hash token. It then go clockwise into the ring until it picks the corresponding node that is holding the data on the basis of token range. Afterward, based on the replication factor and replication strategy which we will discuss in later sections, rest of the replica locations are identified.

2.1.3 Latency

Online services are latency critical and a very small delay directly impacts the revenue. Amazon has reported a latency loss of 100ms causes loss of 1% in sales [6]. To provide good QoS there is a Service Level Agreement (SLA) between the cloud provider and users which defines the scope of the service. To accomplish this task, Service Level Objective (SLO) are used to define the exact metrics that are being used to provide the service.

We take the example of a real Google service [5] running in a cluster of 2,000 servers show that if one in 100 user requests gets slow (e.g., has a 1 second latency) while handled by one server that is collecting responses from 100 other servers in parallel, then 63% of user requests will take more than one second to execute. Even for the services where one in 10000 requests gets slow, it see almost one in five user requests taking more than 1 second. This problem is commonly known as the *tail latency* problem. Tail latency is a very important metrics which defines the worst delays in cloud data stores. Tail latency is the fraction of latency measurements (such as 95th percentile, 99th percentile and more) which incurred longest delay. For example, 99th percentile latency defines the smallest latency of the 1% of largest latency values.

2.1.4 Data replication

Data replication creates a copy of the data or subset of a data and stores to another physical machine (called replica) in same datacenter or another physical location. Data replication is used to provide more data availability and scalability in cloud data stores. In general, data is replicated on different datacenters, racks, and disks. There is a possibility of data loss in the case of network failures, server outage etc. To overcome this, data is recovered from other replicas and request is served to the client. For this, a well-designed replication protocol is needed. Also to improve scalability replication techniques are used. If the data is properly distributed and managed then concurrent clients can access the same data at the same time from different replicas without creating any bottlenecks.

Replication strategies are not only used to achieve availability but also to provide scalability to the systems. It increases the system performance in order to achieve a better response time. However, if only one copy of data is scattered among servers then it leads to a lack of availability and robustness. Therefore by adding more servers and replicating data among those servers, the system is capable of handling more requests and become more scalable.

2.1.5 Data consistency

In general, it's not necessary to get the most recent value of data on every replica all the time. Normally for any read or write operation, client application configures the consistency level that shows how consistent the requested data must be. The decision is

based on the response time and accuracy of data. It determines the number of replicas on which a read/write operations must succeed before returning an acknowledgment to the client. For example, in Cassandra, depends on the consistency level, corresponding replicas return the data and it asks remaining replicas for checksums only. To do this, a snitch function decides in which rack and datacenter are both written to and read from. A user can specify the desired consistency level as ONE, QUORUM, ALL and many more[1].

One. It returns the response from the closest replica as determined by the snitch. It is a weak consistency model which dominates when there is a requirement of "fast access".

ALL. It returns the response once all the replicas have responded to a client. It is a strong consistency model where only one consistent state can be seen but this may cost high latency.

QUORUM. It returns the response once a quorum of the replicas has responded to the client. This consistency model also suffers from performance issues due to getting replies from multiple replicas.

How Quorum is calculated? Quorum is calculated based on the replication factor (RF)

quorum = (sum_of_replication_factors / 2) + 1

sum_of_replication_factors = datacenter1_RF + datacenter2_RF + . . . + datacentern_RF

2.2 Tail latency in key-value stores

2.2.1 Replica selection

Key-value stores use replica selection mechanism to curb the tail latency. Normally data is replicated on multiple servers in key-value stores. For replica selection, when a request comes to a client, it chooses the best replica out of multiple replica servers. This is done by considering the performance of each replica server in the datacenter i.e. client will choose the replica which serves the request in minimum time. In the following sections, we see the replica deployment strategies and working strategy of a replica selection algorithm in key-value stores.

2.2.1.1 Replica deployment strategies

Normally data is stored on multiple replicas and a replication strategy determines the nodes where the replica should be placed. A user can define the *replication factor*

[1]We are following the consistency level 1 in the rest of our work. It means that data is retrieved from a single replica.

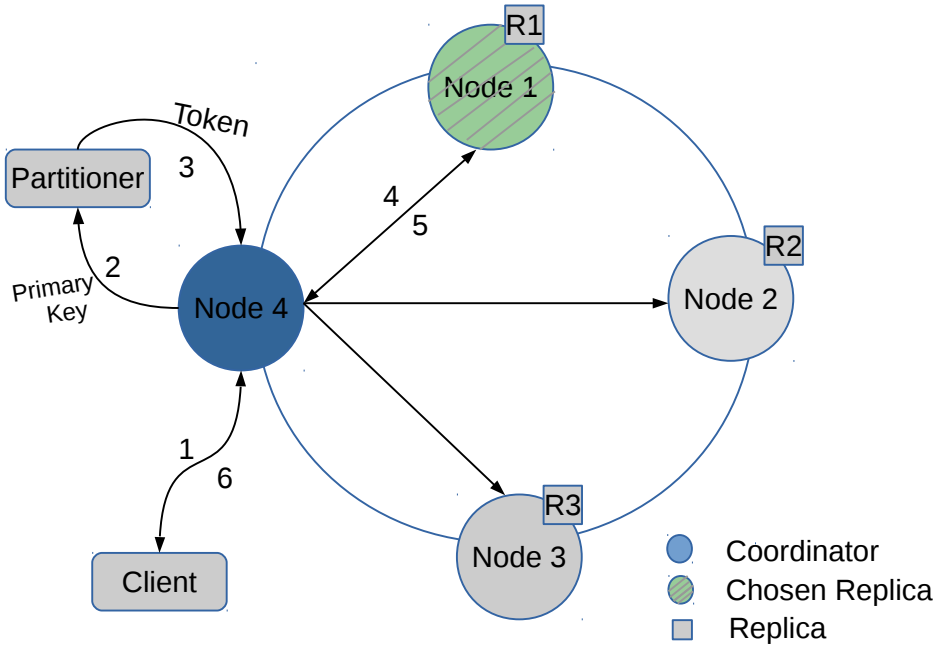


Figure 2.2 – Replica selection in Cassandra.

for each piece of data which is being duplicated within a system[2]. In our work, we are using replication factor as 3 i.e. each piece of data is available on 3 replica servers. Here we discuss the replica deployment strategy followed by Cassandra through which data can be replicated on nodes[3]. We keep Cassandra as a reference point for our study. Since among all the key-value stores, Cassandra [17] implements the most efficient of existing replica selection algorithms (see Table 2.1).

1. **SimpleStrategy:** This strategy is used only for one rack and single datacenter. It places the first replica on a node which is determined by partitioner (eg. Murmur3 [22]). Additional replicas will be placed on next nodes clockwise in the ring.

2. **NetworkTopologyStrategy:** This strategy is used when a cluster is deployed across multiple datacenters. It specifies the number of replicas in each datacenter. It tries to place the replicas on different racks as there is a possibility of failure of rack due to power, cooling or network issues.

[2]A data can be replicated in a single datacenter or multiple datacenters. In our work, we are considering the replication in a single datacenter.

[3]In our work, we are following the Simple strategy for deployment of replicas in a single datacenter.

Cassandra [17]	Choose replica based on history of read latencies
MongoDB [23]	Select nearest node using network latency
OpenStack Swift [24], Apache Accumulo [25]	Read from a single node, choose a new replica in case of a node failure
Riak [26]	Recommend to use an external load balancer

Table 2.1 – Replica selection strategies (adapted from [1]).

2.2.1.2 Replica selection in key-value stores

We take the example of Cassandra [17] in Figure 2.2. Each value is by default replicated to three replica servers (in the example, let us consider a value v replicated on R1, R2, and R3). The request initially arrives at a node (step 1) that will act as its *coordinator*. Depending on the implementation of the coherence mechanisms, the coordinator has to fetch the value from one or multiple replicas. We consider the most-common configuration where the coordinator waits only for the response from a *single* replica. The coordinator asks the partitioner to hash the key (step 2) and gets a token in return (step 3). The coordinator can then identify the first node holding the key/value pair (i.e., R1). The coordinator identifies all the nodes holding the key/value pair based on the replication strategy. Here, replicas of a given pair are placed in successive nodes clockwise. The coordinator uses a *replica selection* algorithm to select the best replica for executing the query (step 5), and replies to the client (step 6).

2.2.2 Task-aware scheduling

In task aware scheduling, a task decomposes into tens to hundreds of sub-tasks. These tasks comprise multiple flows and go through different parts of the network in a different time. Network resource allocation mechanisms treat all these flows in isolation, therefore optimize only flow level metrics. A slow sub-task can slowdown the overall response time. A task is considered as complete when all of its sub-tasks complete. Therefore, this has motivated efforts to schedules the requests in a task-aware fashion. Task-aware scheduling reduces average and tail latency by grouping flows of a task and schedules them together.

For understanding multiget request scenario, we take the example of Cassandra [17] which is a widely used key-value store. As seen in Figure 2.3, node 1, 2 and 3 are holding the values of keys (A,B), (C,D) and (E,F) respectively. A client sends a multiget request mget(A, B, C) to one of the nodes and ask for the value v for corresponding keys A, B, C. The request initially arrives at a node and the node acts as a *coordinator* (step 1). In order to fetch a value, the coordinator asks the partitioner to hash the requested key (step 2) and gets as a response a token corresponding to that key (step 3). Using the obtained token of keys, the coordinator can identify the first node containing the value

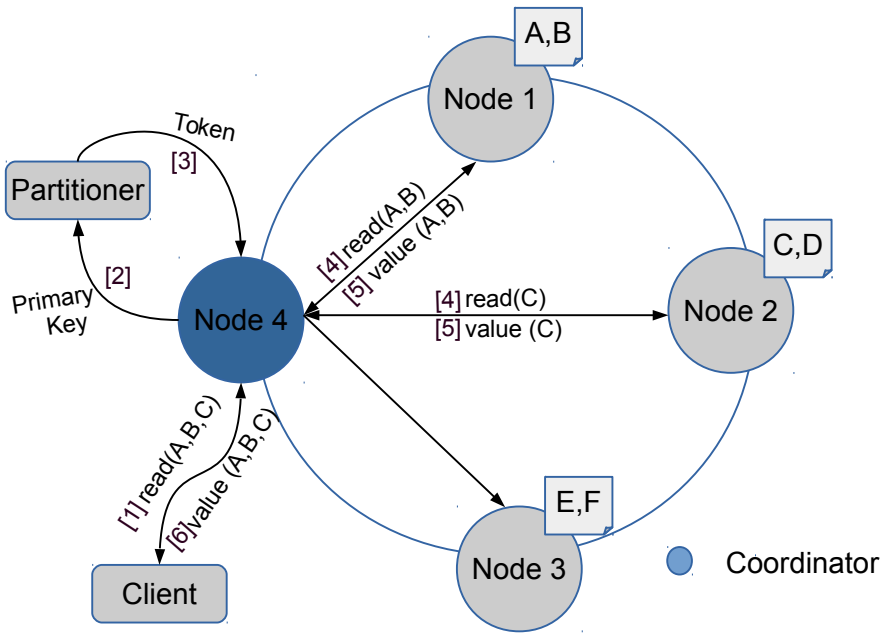


Figure 2.3 – Multiget requests in key-value stores.

A, B and the second node containing the value C (step 4). This mapping is possible because the overall range of hashed keys (or tokens) is split equally among nodes using a distributed hash table as explained in § 2.1.2. The corresponding nodes return the value to the coordinator (step 5). Finally, coordinator returns the response to the client (step 6).

2.3 Workload heterogeneity is the norm

In today's key-value stores, workloads which are coming on a system are heterogeneous in nature. The value sizes which are stored in these key-value stores can span an order of magnitude. As seen in figure 2.4 a value can be binary, text, image or small video. These value sizes vary from few bytes to MBs. For instance, a workload analysis at Facebook's memcached key-value store [18] shows that stored value sizes have large variations. The value size distribution is heavy-tailed in some of the pools at a memcached key-value store. A similar degree of variability is shown in WikiPedia [27] and Flickr [28] datasets where value size varies from kBs to MBs.
The value sizes are highly skewed towards smaller size but very few large value sizes consume a large share of computational resources [5]. Therefore, to optimize the performance of key-value stores ideally we would like to assign jobs to servers according to the size of jobs. Conventionally this may entail minimizing the tail latency or the server can be well balanced [29].

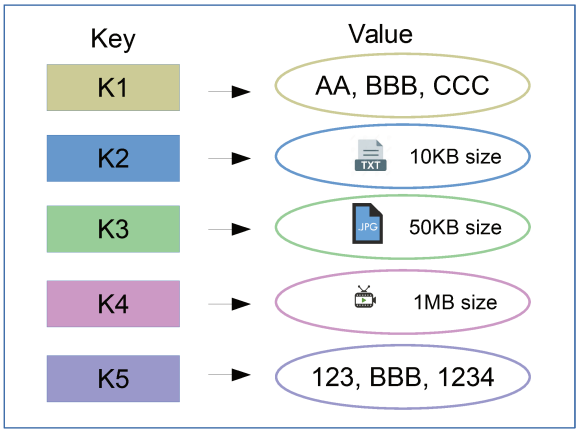


Figure 2.4 – Key-Value Table.

2.4 Summary

In this chapter, we discuss the background in key-value stores. We discuss the problem of tail latency in cloud data stores and why it's important to curb this latency. We discuss the workload heterogeneity in these systems and how it affects the tail latency of the system. Further, we discuss the replica selection approach and its effectiveness to deal with tail latency. Finally, we discuss the multiget requests in key-value stores.

3

Latency Aware Algorithm for Replica Selection

Contents

In this chapter, we present Héron [30], a replica selection algorithm that reduces tail latencies under heterogeneous workloads. Héron predicts which requests will require significant time by keeping track of the keys corresponding to large values. It does so using a Bloom filter at each server. Once a request has been identified as accessing a small (respectively, a large) value, it applies an appropriate replica selection algorithm, which avoids head-of-line-blocking.

Héron can be applied to any low latency key-value store where value replication is enabled. Among such stores, Cassandra [17] implements the most efficient of existing replica selection algorithms (see Table 2.1). It is therefore the best reference point for our study. We implemented Héron in Cassandra and compared its performance to state-of-the-art algorithms (e.g., dynamic snitching [17] – DS for short – and C3 [1]) in a public cluster of 15 machines on Grid5000 [31].

We evaluate Héron with four datasets, two synthetic datasets including one based on access statistics from Facebook [18] and two real datasets from Flickr [28] and Wiki-Media [27]. We use YCSB [32] to generate different requests workloads, allowing to evaluate the system under different read/write ratios. Our results show that Héron improves tail latency over state-of-the-art algorithms without compromising median latency.

The remaining of this chapter is structured as follows. We first detail the major replica selection algorithms proposed in the literature (§3.1). Next, we further explain the limitations of the C3 and DS replica selection strategies in presence of heterogeneous workloads (§3.2). Afterwards, we present a detailed description of Héron (§3.3) and present its implementation and performance evaluation (§3.4). Next, we detail the state-of-the-art related to replica selection and tail latency improvement in distributed systems (§4.2). Finally, we conclude the chapter (§3.6).

3.1 Replica selection in key-value stores

Two major replica selection algorithms have been proposed in the literature. The details are as follows:

Dynamic snitching By default, in Cassandra, replica selection is done using *dynamic snitching* (DS for short) [17]. The performance of read requests from various replicas is monitored over time and the best replica is selected based on its recent performance history. DS maintains a score for each replica that is updated every 100 ms. All replica scores are reset every 10 minutes, to allow replicas to possibly recover. The limitation of this approach is that it is solely based on replicas' past read performance, without considering the forthcoming load at each replica (i.e., its queue size). This may lead to overloading recently-fast replicas and to the appearance of bottlenecks in the system, ultimately impacting tail latency.

The C3 algorithm C3 [1] is a replica selection algorithm that improves over DS by handling service time variations among replicas. C3 computes replica scores based on both service time and queue size. This score is then used by a request coordinator for choosing the replica that is expected to better help reducing the request waiting time. Additionally, to avoid overloading a given replica queue (e.g., because the

replica is fast and thus simultaneously selected by several coordinators), C3 uses a rate control mechanism at each replica to limit the arrival of requests. Results show that C3 significantly improves tail latencies compared to DS [1].

3.2 Performance of DS and C3 under heterogeneous workloads

Several studies, including the analysis of a Facebook Memcached deployment [33], show that workloads are heterogeneous: key-value stores contain values of sizes ranging from a few bytes (e.g., text messages) to MB (e.g., photos or videos). Our analysis of the Flickr [28] and WikiMedia [27] datasets confirms this trend. Figure 3.1 shows the CDF of value sizes for the two datasets, ranging from a few bytes to MBs.

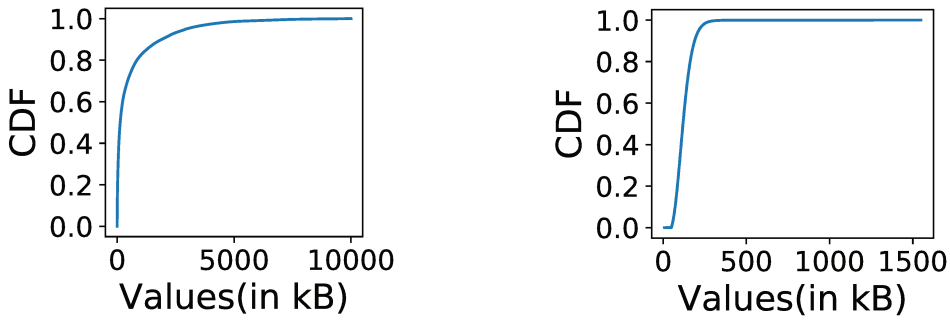


Figure 3.1 – CDF of value sizes for WikiMedia (left) and Flickr (right) datasets.

We illustrate the limitations of C3 and DS in presence of heterogeneous workloads with an example in Figure 3.3. In figure (a), three replica servers A, B and C currently have a service time of 2 ms, 3 ms and 1 ms, respectively, for requests of values size 1 KB. For the sake of simplicity, let us assume that the large request depicted with a large square and a dark color has a size of 512 KB and that small requests depicted with small rectangles and a fair color have a size of 1 KB. C3 will select replica server A for subsequent requests since it only considers the queue size and service time for estimating the fastest replica (queue size $\times$ service time of replica A is smaller than queue size $\times$ service time for the other replicas). DS only considers the past read performance (measured every 100 ms). Let us assume that in the recent past replica A only processed small requests. In that case, DS will select this replica to process the subsequent requests. As a result, under both algorithms, the latency of the next small request will be of (X+2) ms where $X \gg 2$: X ms waiting for the large request to be processed, and 2 ms for processing the small request (figure (b)).

In an ideal scenario, the next small request should complete in a much shorter time: it should not be stuck behind the large request at replica server A. This means that,

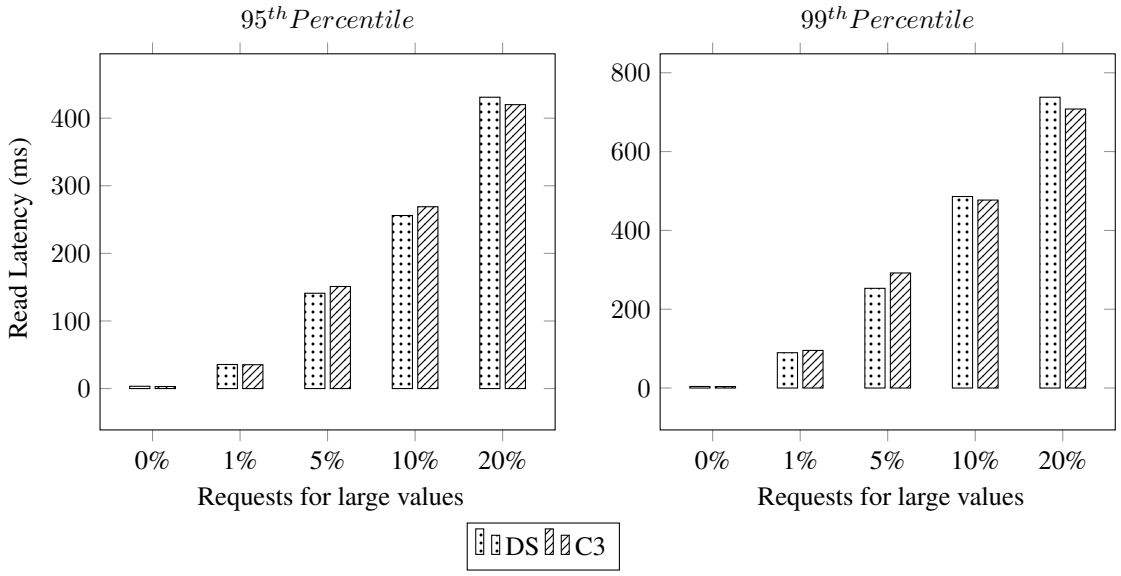


Figure 3.2 – Tail latency when varying the proportion of requests for large values with DS and C3.

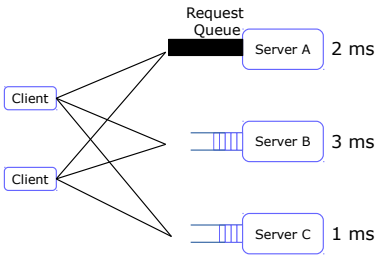
while processing a large request, a server should no longer be selected to process small requests. These small requests should be scheduled instead on the following best replica based on its score (service time $\times$ queue size). In the example of Figure 3.3, the next small requests should be sent to replica server C, which would yield a latency of 5 ms (figure (c)).

More practically, to study the impact of heterogeneous workloads on DS and C3, we use a synthetic workload containing small (1 KB) and large (512 KB) values. We use YCSB [32] to generate a read-heavy workload. We vary the proportion of large values from 0% to 20%. Our experimental settings are detailed in Section 3.4. Figure 3.2 presents the 95^{th} and 99^{th} percentile of the read latency distribution. We observe a significant increase in tail latency when increasing the proportion of large values for both DS and C3. For the 95^{th} percentile, the increase ranges from $\times 10$ when there are 1% of requests for large values, to $\times 126$ when there are 20% of requests to large values.

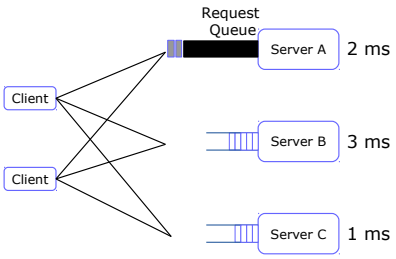
3.3 Héron design and implementation

This section presents Héron, a replica selection algorithm 1 that aims at reducing tail latency under heterogeneous workloads.

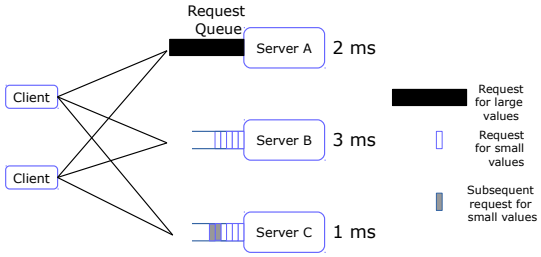
Figure 3.4 presents the architecture of Héron and how it handles client requests. When a request from a client reaches a coordinator, it first goes through a replica scoring module (step 1). The coordinator retrieves the set of replicas storing the requested value by querying the partitioner. The replica scoring module ranks these replicas based on periodic feedback on their latest service time and queue size. In parallel, the request



(a) An example scenario where a large request has been scheduled on replica server A. The question is: where should subsequent small requests be scheduled?



(b) C3 and DS schedule the subsequent requests to replica server A which results the requests get stuck behind the large one on replica server A.



(c) Shows how Héron dynamically blocks the replica server A for small period of time until the latter finishes the processing of the large request. Meanwhile, subsequent small requests are scheduled on replica server C.

Figure 3.3 – An example scenario illustrating the limitations of state-of-the-art solutions.

is sent to the Bloom filter manager (step 2). This module estimates whether a given request will access a small or a large value. It uses a Bloom filter that keeps track of large requests. The replica selection module (step 3) uses the input from both modules to select the replica that is expected to serve the request faster.

We first detail the challenges (§3.3.1) and then present the mechanisms for load estimation among servers (§3.3.2), followed by size estimation using Bloom filters

Algorithm 1: On Request Arrival (Request *req*, Replicas *R*, Bloom filter Manager *bF*)

```
1  repeat
2      R ⟵ sort(R)
3      for Server s_i in R do
4          if req.key is in bF then
5              if s_i is available then
6                  send(req,s_i)
7                  put s_i on busy
8              else
9                  repeat
10                     go to next server s_i in R
11                 until s_i is available
12                 send(req,s_i)
13         else
14             if s_i is available then
15                 send(req,s_i)
16             else
17                 repeat
18                     go to next server s_i in R
19                 until s_i is available
20                 send(req,s_i)
21         if s_i is busy then
22             wait until s_i is available
23 until req is sent
```

(§3.3.3) and replica selection (§3.3.4).

3.3.1 Challenges

This section details the challenges associated with reducing tail latencies under heterogeneous workloads, and the design space for building a key/value store with this goal.

Dealing with heterogeneous requests first requires being able to distinguish requests accessing small values from requests accessing large values. This must be done by the coordinator node, but this node only knows the key that is requested. It does not hold the value and must therefore use a specific mechanism to estimate the category of size the value belongs to. Under the high performance constraints of key-value-stores, this mechanism must be cost-effective.

Once the coordinator is able to distinguish between requests for small and large

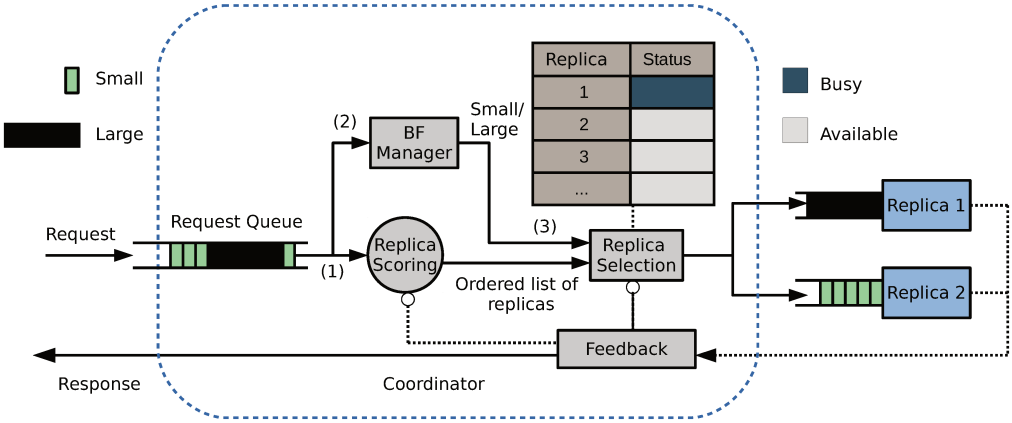


Figure 3.4 – Operating principle of Héron.

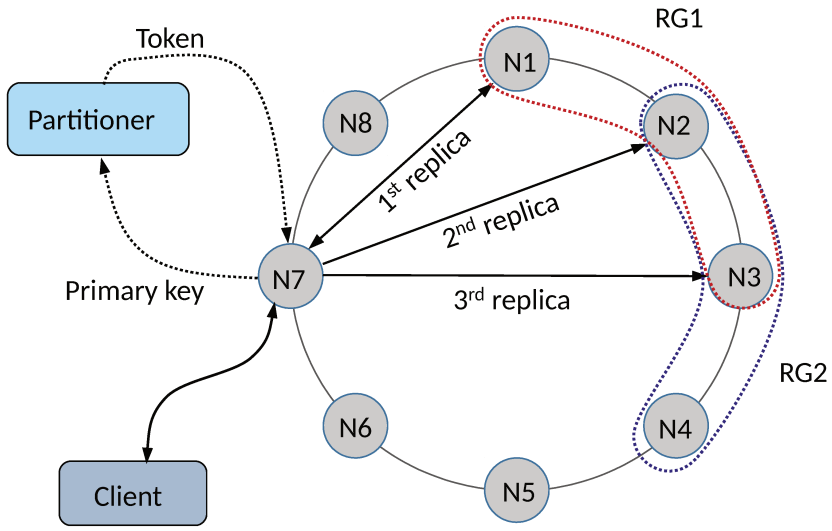


Figure 3.5 – Cassandra placement strategy.

values, it must make appropriate scheduling decisions, i.e. decide which of the replicas will serve in minimum latency while handling the request. Both static and dynamic scheduling strategies can be considered.

Static scheduling strategies permanently assign one or multiple servers to handle requests accessing small (or large) values. While this has been shown to be effective in other contexts (e.g., scheduling in high load situations where large request overwhelm a server [2, 34]), such solutions are not appropriate in key- value stores due to the

overlapping between token ranges. Let us for instance consider the cluster depicted in Figure 3.5 where the first replica of each value is determined by the partitioner (e.g. Murmur3 [22]) and the rest of the replicas are placed on next nodes clockwise in the ring. In this case, statically reserving a replica on each group to handle small requests (e.g., node N1 in the group RG1, node N2 in the group RG2, etc.) may lead to a situation where all replicas are reserved to serve small requests, which leads to the starvation for large requests. To avoid this problem, Héron uses dynamic assignment policy (explained in§ 3.3.4) for placing the jobs on nodes in which the reservation of replicas is made dynamic.

With a *dynamic* scheduling strategy, a particular replica is reserved at runtime and only for a small amount of time to handle requests for large values, leaving the others available for handling requests to small values as shown in figure 3.5. Since requests for large values create more head-of-line-blocking rather than requests for small values. Compared to a static assignment policy, this approach has the benefit of blocking replicas only temporarily. With dynamic scheduling, a request can be handled by any replica of the value. Two main approaches are available for selecting this replica, priority-based and FIFO algorithms. Priority-based algorithms, such as shortest job first [35] give the priority to fastest jobs over slower ones. In our context, this means giving priority to requests accessing small values over requests accessing large values. This approach is not ideal for a key-value store. It creates an imbalance between the requests of different clients, and clients accessing larger values can potentially observe a significant increase in latency. FIFO algorithms process requests in their order of arrival, regardless of the size of the access values. They do not incur fairness or consistency problems and are therefore more appropriate for key/value stores. The challenge however is to offer a FIFO scheduling algorithm that avoids the head-of-line-blocking problem.

3.3.2 Load estimation

Héron includes a replica scoring mechanism similar to the one used by C3 [1] to estimate the load among servers as described in figure 3.6. Coordinators periodically collect as scoring metric for each server the product of its average service time and its queue size. Replicas with a lower score are better candidates for serving incoming requests, if all requests are for value of the same size. The difference with C3 is that Héron does not rely on a control flow mechanism to balance the load between servers. Instead, Héron uses differentiated scheduling in which a dynamic assignment policy is used to schedule requests as described in the following.

3.3.3 Value size estimation

The objective of Héron is to process fast and slow requests in a way that avoids head-of-line-blocking. To reach this objective, Héron needs to predict whether a request will access a large (slow request) or a small value (fast request). The size threshold

between these two types of requests is application-specific. We therefore assume that an application and database administrator will be able to set a threshold value THR_L according to the data distribution over her database. For instance, for a database containing values ranging from 1 KB to 512 KB, the system designer may rely on an evaluation of the average read time latencies of these values as shown by Figure 3.7. This figure shows that a request accessing a 512 KB value is 32 times slower than a request accessing a 1 KB value, and that there is a significant gap in latency between values of sizes 256 KB and 512 KB. In this example, the system designer may set the threshold parameter THR_L to 256 KB.

Héron uses Bloom filters to keep track of keys corresponding to large values as depicted in figure 3.8.

Bloom filters [36] are space-time efficient probabilistic data structures that allow performing set-membership queries (i.e., testing whether a given item belongs to a set). A Bloom filter is a vector of m bits initially set to 0, with an associated set of k hash functions (with $k \ll m$). Inserting an element in a Bloom filter is done by hashing the element (in our context the key contained in the request) using the k hash functions and setting the corresponding bit positions to 1. Testing the presence of an element in a Bloom filter is done by hashing the element using the k hash functions and testing whether *all* corresponding bit positions are set to 1. Querying a Bloom filter may lead to false positives but will never lead to false negatives. The false positive rate depends on the size of the vector, the number of hash functions and the maximum number of elements to be inserted in the set. Héron sets these values so as to maintain the false positive rate below 0.1%.

Constructing the bloom filter

We update the Bloom filter when new large values get inserted in the database. The

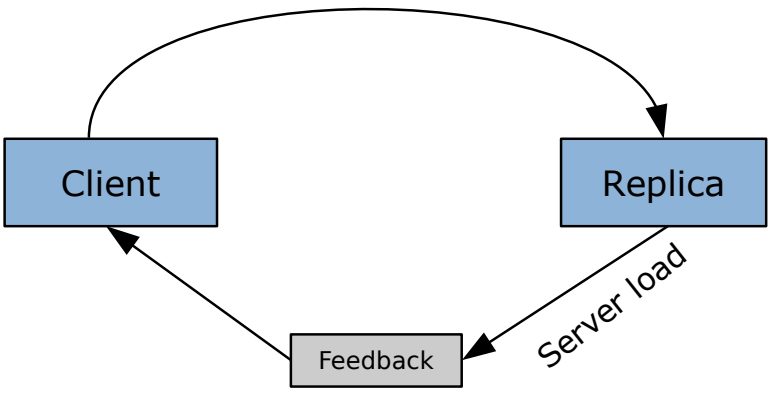


Figure 3.6 – Load Estimation in Héron.

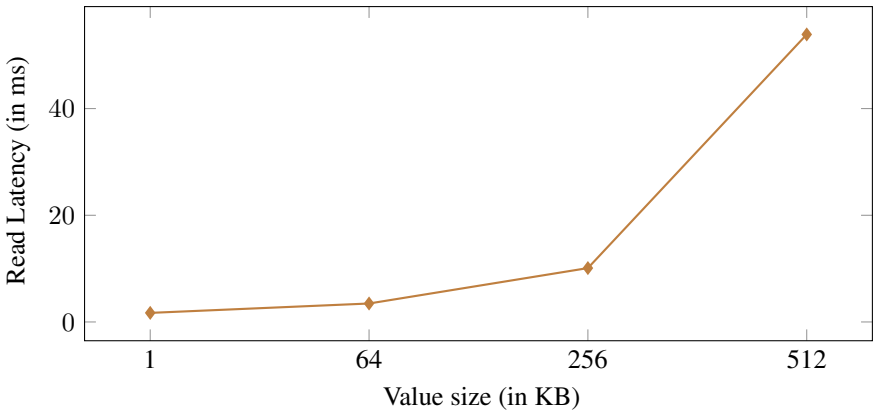


Figure 3.7 – Example of requests read latencies.

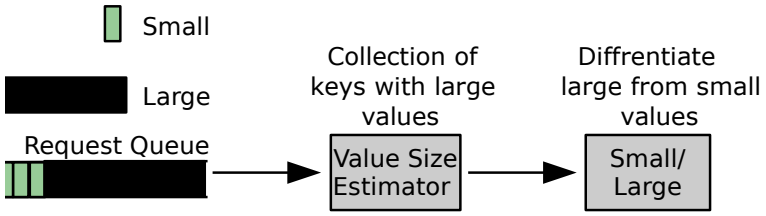


Figure 3.8 – Value Size Estimation

filter is built at each of the replicas. For each write request, if the value size exceeds THR_L, the key is inserted into the Bloom filter. The Bloom filter is also updated when an existing small value is replaced by a value whose size exceeds THR_L.

We observe that in workloads such as the one from Facebook [18], different tables have different value distribution patterns. Some may contain only small values and be relatively homogeneous, while others are highly heterogeneous. To account for this fact, Héron allows administrators to set policies that disable the use of the Bloom filters for tables that have less than a configurable proportion of large values, as observed from the collected statistics, and avoid paying the overhead of querying the Bloom filter when it is not necessary.

Synchronizing the bloom filter between nodes

The addition of information to the filter is performed at all of the replicas for a given key. The construction of a common global filter across all coordinators requires aggregating the filters from all replicas, i.e. keeping the result of the logical `or` operation between

all filters. Due to the append-only nature of this construction, and to the idempotency of the aggregation, there is no need to complex synchronization involving a consensus protocol. Héron disseminates updates in an asynchronous, gossip-like, way. When the local filter is modified by a given node, upon the addition of a new large value or the replacement of a small value by a large one, it is piggybacked on the write acknowledgment sent to the coordinator. Coordinators gradually construct the global filter by interacting with storage servers, and merging newly-set bits to their local filter.

Handling deletions and growth

Deletion of large values (or their replacement by small values) would require removing their keys from the filter aggregated by coordinators. Bloom filters do not allow this operation, as un-setting the bits for the corresponding key comes with the risk of un-setting bits set for keys of other large values still present in the store. Handling deletions with a compact membership representation is actually only possible using more complex data structures, such as counting Bloom filters [37] or counting quotient filters [38]. These data structures have higher costs in memory and computation. More importantly, they are less amenable to the simple, asynchronous synchronization that Héron uses: their aggregation would require more costly consistency maintenance for coordinators.

Another linked issue is that of filters that happen to be insufficiently large after the growth of the dataset. In this case, the rate of false positives increases and the system is at risk of incorrectly considering too many keys as being associated with large values. Again, dynamically-resizable compact membership representations such as incremental Bloom filters [39] or counting quotient filters [38] can address this issue, but they also come at the cost of additional complexity in particular for aggregation and querying. As a result, Héron does not implement such features but relies on periodic system updates as detailed next.

Periodic system updates

Our system includes a number of updates that take place periodically, and using a low-priority background task. These tasks include the *regeneration* of Bloom filters and their synchronization, the periodic updates of table statistics and possibly the updates on the threshold for large values. Specifically, the datastore is periodically analyzed and the distribution of value sizes is updated. Using the gathered data, each storage node computes a new filter for the values it owns and whose size exceeds the threshold. The parameters of this filter (size, number of hash functions) may change according to some administrator-defined policy, e.g. if the size of the store exceeds the value initially estimated. Coordinators aggregate filters for several generations and start using the latest generation as soon as they have received an update for it from all

storage servers. Similarly, the list of tables that contain a given proportion of large requests is updated. Finally, if the distribution of values changes dramatically, the administrator gets informed and may possibly decide to adjust the threshold for large values accordingly.

3.3.4 Replica selection module

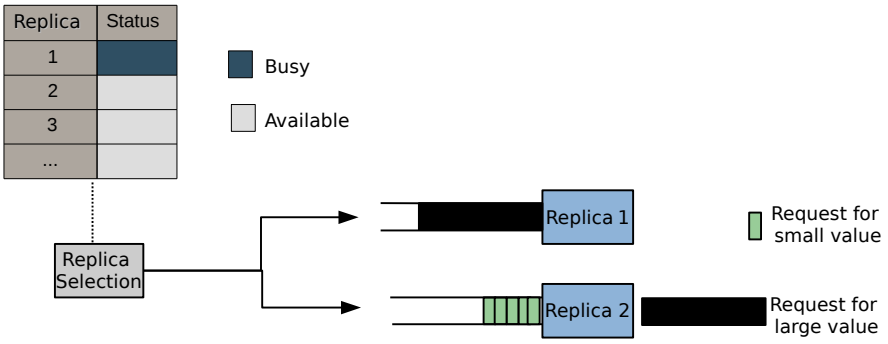


Figure 3.9 – Replica selection in key-value stores.

The replica selection selects the replica that is expected to serve an incoming request faster than the other replicas. The working mechanism of replica selection is depicted in figure 3.9. It uses three types of information: (i) whether the request is expected to access a small or a large value, as provided by the Bloom filter manager; (ii) the relative score of the servers holding replicas for that key, as provided by the replica scoring module; (iii) whether these replicas are currently handling a request for a large value or not. The last information is maintained over time by the replica selection module. Specifically, the initial status of a replica having no request to process is set to *available*. As long as this replica processes requests accessing small values, its status remains *available*. Instead, when a request accessing a large value is scheduled on a given replica, its status becomes *busy*. The latter comes back to the *available* status when the processing of the large request is over.

Scheduling of requests tagged as large

If the request is tagged as large by the Bloom filter manager and if there exists a replica R whose status is *available*, then the request is sent to R and R's status becomes *busy*. If there is more than one replica whose status is *available*, the replica selection module

uses the scores provided by the replica scoring module to chose the one that is expected to be the fastest. After finishing processing the request, the replica selection module updates the status of R to *available*. If there is no *available* replica, then the request is blocked until at least one of the replicas becomes *available*.

Scheduling of requests tagged as small

If the request to schedule is tagged as small by the Bloom filter manager, the replica manager uses the first replica server whose status is *available* from the ordered list provided by the replica scoring module. In this situation, the replica's status remains unchanged since short jobs are not expected to affect tail latency by head-of-line blocking.

Synchronizing information about replicas' status

A coordinator relies on its local knowledge of the replica status for making scheduling decisions. This information is synchronized in a best-effort manner between replicas. Specifically, the propagation of this information leverages existing communication between servers and coordinators for requesting queues sizes and service times for replica scoring. Storage servers include their current status with all exchanged messages and coordinators update their replica status table accordingly when receiving these messages. This asynchronous and best-effort propagation may lead to inconsistent views between coordinators about the status of a given storage server. For instance, a given coordinator may schedule a request to a given replica and locally update the latter's status to *busy*. At the same time, another coordinator can schedule small requests into this same replica because it did not yet receive the feedback that this node was busy. In this situation, the head-of-line-blocking problem may occur but only for a limited period of time. We consider this to be an acceptable compromise as a fully consistent exchange of information about replica status would greatly impair the horizontal scalability of the key/value store and its overall performance. Our evaluation shows that Héron still drastically reduces tail latencies despite the potential inconsistencies in replica state tables.

3.4 Evaluation

We implement Héron as an extension of the Cassandra [17] key/value store. We evaluate its effectiveness in reducing tail latency using both synthetic datasets generated using the Yahoo Cloud Serving Benchmark (YCSB) [32], and real datasets from WikiMedia [27] and Flickr [28]. We compare the reduction in tail latency with DS [17] and C3 [1]. We conduct the experiments on a public high-performance cluster representative of current cloud data centers hardware [31]. We analyze the impact of three types of

heterogeneous workloads, i.e., read-only (100% read), read-heavy (95% read-5% write) and update-heavy (50% read-50% write), with varying ratios of requests for large values and varying large value sizes.
Our evaluation aims at answering the following questions:

1. How does Héron compare to C3 and DS when the dataset contains a mix of small and large values? (§3.4.2.1)
2. How is the performance of Héron impacted by the proportion of large values in the key-value store? (§3.4.2.2)
3. How does Héron perform when the proportion of read vs write requests varies? (§3.4.2.3)
4. Is the impact of Héron confirmed with real datasets? (§3.4.3)

We start this section by presenting our evaluation setup (§3.4.1) before presenting our results (§3.4.2 and 3.4.3).

3.4.1 Experimental setup

Experimental platform. We evaluate Héron on Grid5000 [31]. We use 15 servers equipped with 2 Intel Xeon E5-2630 v3 CPUs (8 cores per CPU), 128 GB of RAM, and 2 558 GB HDDs. Servers are interconnected by a 10Gbps Ethernet network and run the Debian 8 GNU/Linux OS.
Cassandra configuration. We use a replication factor of 3, which means that each value is available on three replica servers. We consider a write-all read-from-one coherency mechanism in which consistency is achieved by reading from a single replica and not from a quorum. Each experiment involves 2 million requests accessing small and large values. We systematically check that Cassandra achieves its maximum throughput (i.e. that we use enough clients to saturate the system). The measured peak throughput depends on the proportion of requests for large values in the system. Figure 3.10 shows the maximal achieved throughput across all experiments and value sizes in this section.

When there are no requests to large values, the throughput peaks at $\approx$ 72000 requests/sec. It reduces to $\approx$ 2200 requests/sec when 20% of the requests are for large values.
Synthetic datasets. We use the industry standard Yahoo Cloud Serving Benchmark (YCSB) [32] to generate synthetic workloads. YCSB originally only generates workloads with values of a single size. We modified its source code to generate a configurable proportion of large and small values. The size of small and large values is also configurable. The distribution of access frequency for all stored values (small and large) follows a Zipfian distribution with a Pareto index of $\alpha = 0.99$. This means informally that the most popular value is almost twice as popular as the second-most-popular value, and so on with decreasing popularities.

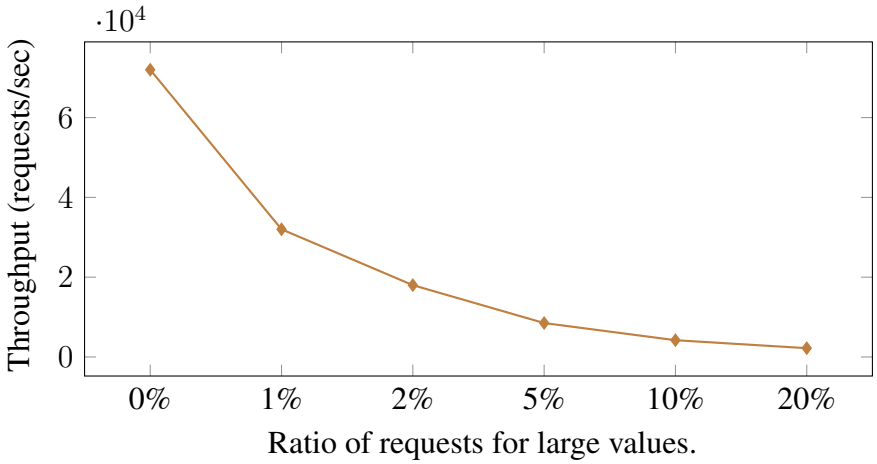


Figure 3.10 – Maximum throughput attained across all scenarios.

With a replication factor of 3 and 15 servers, each storage node holds about 170 GB of data.

Real datasets. We evaluate Héron with the publicly available Flickr [28] and WikiMedia [27] datasets, which contain 1 million images and 225 thousand images, respectively. The CDF of image sizes on these datasets is shown by Figure 3.1. We use YCSB [32] to generate workloads based on these datasets and considered the 10% largest values of each dataset as large in all scenarios. The access frequency distribution for stored values (small and large) follows the same Zipfian distribution as for the synthetic workloads.

3.4.2 Héron on variable configurations of the synthetic dataset

We evaluate Héron along three dimensions of heterogeneous workloads, i.e., the varying size of large values (§3.4.2.1), the ratio of requests for large values (§3.4.2.2), and types of workloads (§3.4.2.3).

We compile the absolute latency values measurements of Héron in Table 3.1. We will refer to this table on the three subsequent sections. We present the relative improvements between the different systems in Figures 3.11, 3.12 and 3.13.

3.4.2.1 Varying the size of large values

We start by studying the impact of the size of large values. We fix the proportion of requests to large values to 10%. We vary the size of large values between 64 KB, 128 KB, 256 KB and 512 KB. By looking at the table of absolute values (Table 3.1), we first note that, when the size of values increases in the database (e.g., 64 KB values versus 512 KB values), tail latencies are higher because the system takes more time to execute the overall workloads.

Moreover, in the case of values of size 512 KB, the probability of running into head

Dataset	Update Heavy		Read Heavy		Read Only	
	$95^{th}\%ile$	$99^{th}\%ile$	$95^{th}\%ile$	$99^{th}\%ile$	$95^{th}\%ile$	$99^{th}\%ile$
64 KB	12.5 ms	27.8 ms	21.2 ms	33.7 ms	24 ms	41.8 ms
128 KB	25.2 ms	41.3 ms	51 ms	101 ms	47 ms	86 ms
256 KB	50 ms	81 ms	95 ms	160 ms	105 ms	209 ms
512 KB	130 ms	218 ms	196 ms	340 ms	199 ms	336 ms
1%	19.1 ms	57.2 ms	35.2 ms	89.6 ms	35.3 ms	96.2 ms
2%	38.2 ms	92.7 ms	58.8 ms	114.6 ms	65 ms	134.2 ms
5%	77.6 ms	140 ms	121 ms	216 ms	139 ms	272 ms
10%	130 ms	218 ms	196 ms	340 ms	199 ms	336 ms
20%	183 ms	310 ms	336 ms	549 ms	349 ms	574 ms

Table 3.1 – Héron absolute performance measurements.

of line blocking in Héron is higher than in the case of 64 KB. This is not surprising as requests for large values occupy servers for a longer period of time. In order to have a closer look at the numbers, we summarized the results for the read-heavy workload in Table 3.2. This table shows the improvement provided by Héron over C3 and DS for the median and 99th percentile latencies. From this table we can observe that Héron improves the median latency by up to 36% over C3 and DS respectively. Furthermore, Héron improves the tail by up to 30% over DS and 28% over C3.

Dataset	Improvement over DS		Improvement over C3	
	$Median$	$99^{th}\%ile$	$median$	$99^{th}\%ile$
64 KB	+19.23%	+11.6%	+20.74%	+9.13%
128 KB	+17.58%	+13.67%	+19.35%	+10.61%
256 KB	+23.12%	+17.52%	+27.64%	+14.89%
512 KB	+36.75%	+30.04%	+36.02%	+28.72%

Table 3.2 – Improvement of Héron over Median and Tail Latency with different large request sizes

We present the percentage of improvement of Héron over DS and C3 for read-heavy workloads in Figure 3.11. For instance, the improvement of Héron over DS for large values of size 64 KB is 9% for the 95^{th} percentile, meaning that the tail latency at this percentile of Héron is 91% that of DS under the same conditions. The read latency with Héron is lower than the read latency of both C3 and DS in all tested configurations. More importantly, the improvement increases with the value sizes, i.e., Héron achieves higher latency reduction than DS and C3.

For large value sizes of 64 KB, 128 KB, and 256 KB, the improvements over C3 and DS are relatively modest, ranging between 8%-10% and 12%-17% for 95^{th} and 99^{th} percentiles tail latency, respectively. For large values sizes of 512 KB, Héron improves the tail latency by up to 30% over DS and 28% over C3. This is explained by the fact that Héron is able to mitigate the waiting time of a request for small values by avoiding scheduling them behind requests for large values. As C3 and DS are oblivious to the size of the requested value, not only requests for small but also request for large values can be scheduled behind requests for large values, significantly degrading tail latency. Our measurements further show that the improvement brought by Héron for median latency ranges from 1.5 ms to 13 ms for large value sizes ranging from 64 KB to 512 KB.

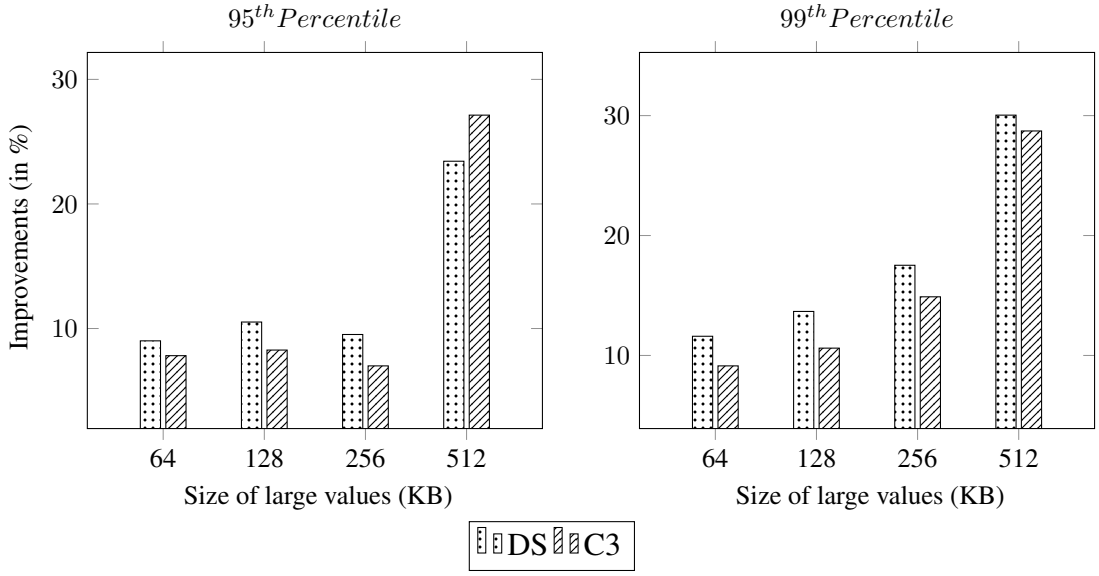


Figure 3.11 – Improvement of tail latency with Héron for different sizes of large values.

3.4.2.2 Varying the proportion of requests for large values

We study the impact of the proportion of requests for large value on system performance. We fix the size of large values to 512 KB and we vary their proportion from 1% to 20% for all three types of workloads. From the absolute values reported in Table 3.1, for both the 95^{th} and 99^{th} percentiles and for all three workload types we observe that the latency increases with the percentage of requests for large values, which is expected as these requests take longer to execute.

We present the improvement of Héron over DS and C3 for read heavy workloads in Figure 3.12. We can observe that Héron outperforms DS and C3 in all configurations. Further, we observe that the effectiveness of Héron increases with the percentage of

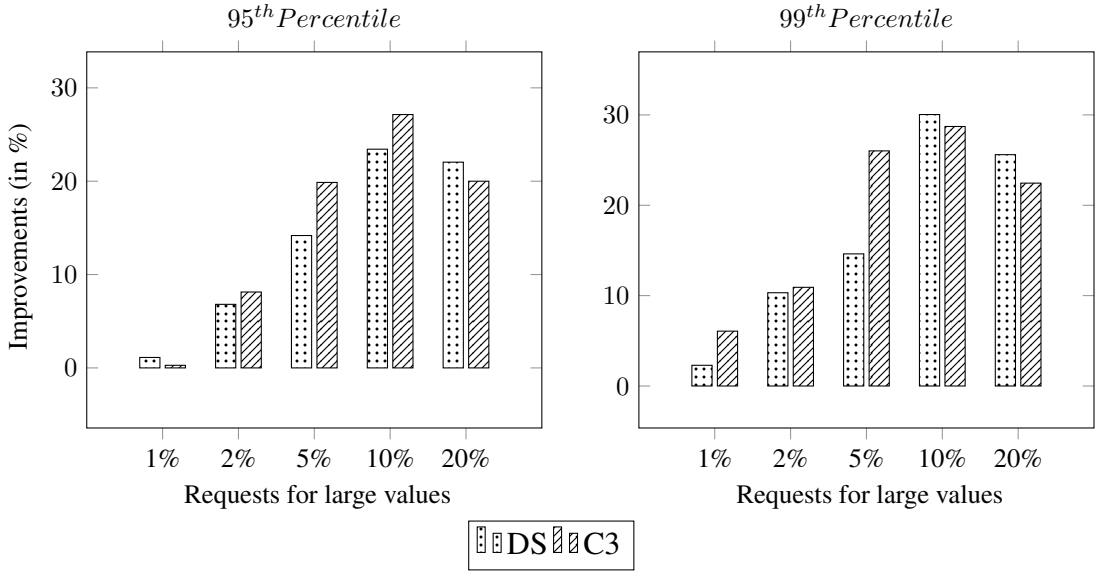


Figure 3.12 – Improvement of Héron for different proportion of request for large values.

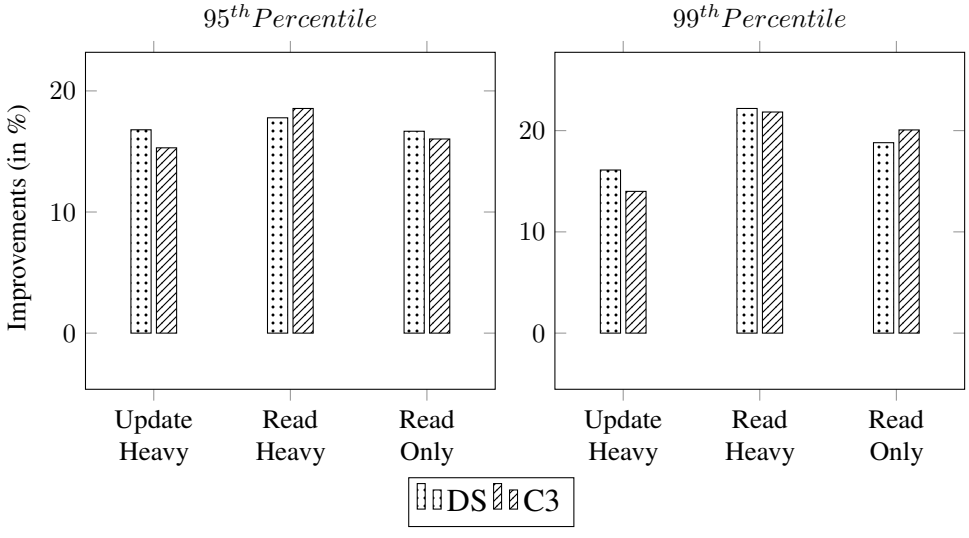


Figure 3.13 – Improvement of Héron over the average of all heterogeneous workloads.

requests for large values. With 10% of requests for 512 KB values, Héron achieves an improvement of around 25%; however the improvement slightly drops in the case of 20% of such requests. This can be explained by the fact that the probability that all servers are blocked by a request for a large value is higher in the case of 20% of such requests than the same probability in the other configurations. In this situation, Héron loses part of its ability to dispatch the requests agilely among servers. In the following we zoom into each specific percentage of requests for large values.

95-5. In this experiment, 5% of the values in the database are large ones. Héron shows similar improvement over DS, i.e., roughly 15% for the 95^{th} and 99^{th} percentile of tail latency. Compared to C3, Héron achieves better gain, i.e., roughly 26% improvement for the 99^{th} percentile.

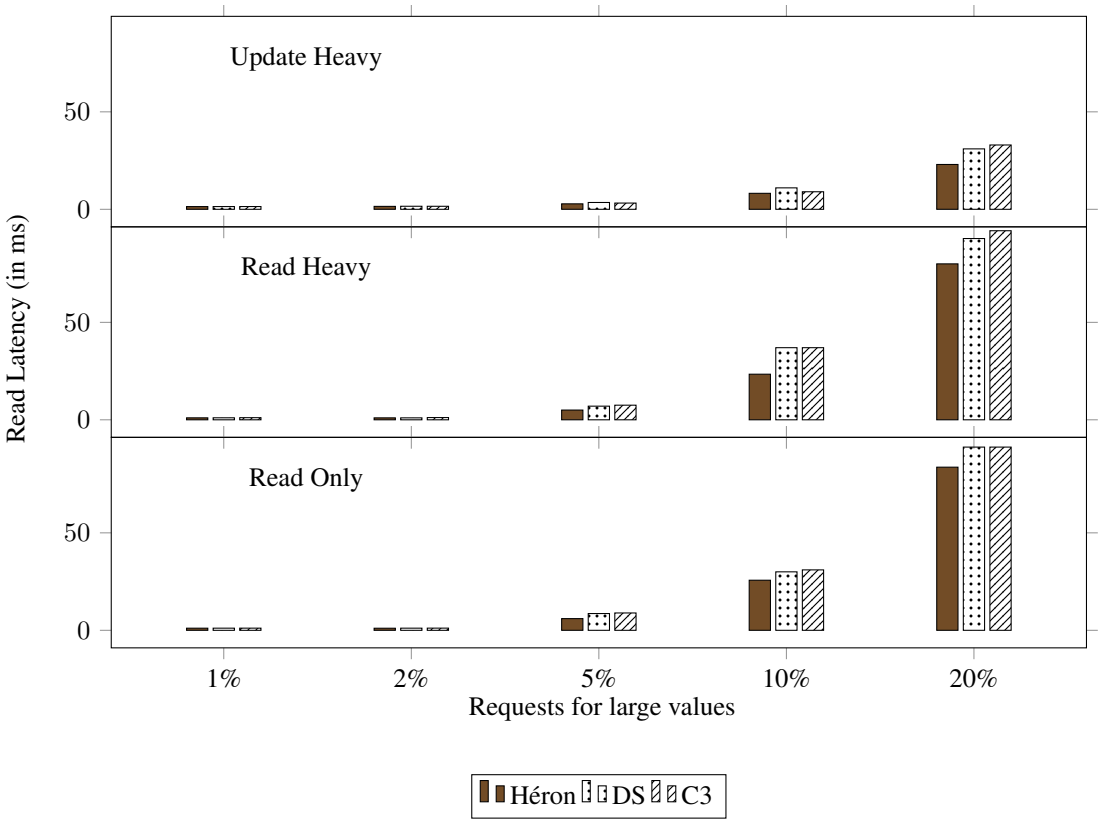


Figure 3.14 – Median latency over different proportions of request for large values.

90-10. In this experiment, 10% of the values of the database are large ones. Héron shows similar improvement over C3, i.e., around 27% for the 95^{th} percentile. Against DS, Héron achieves a slightly better gain for the 99^{th} percentile than for the 95^{th} percentile, i.e., roughly 23% v.s. 28%. In terms of absolute latency, say for the 99^{th} percentile, it is 340 ms for Héron but roughly 486 and 477 ms for DS and CS, respectively. Héron achieves the best performance for this workload, i.e., 10% of requests for 512 KB large values, where there is a sufficient number of available servers for Héron to schedule requests for large values without blocking incoming requests.

80-20. In this experiment, 20% of the overall values are large ones. Héron shows consistent improvement over C3 and DS for both the 95^{th} and 99^{th} percentiles, i.e., roughly 23% and 20%. This increases up to 41% in case of the 99.9^{th} percentile tail latency. In terms of absolute latency, for instance for the 95^{th} percentile, it is 336 ms for Héron but roughly 431 and 420 ms for DS and CS, respectively. The gap is even more significant for the 99^{th} percentile, where the observed latencies for Héron, DS, C3 are 549 ms, 738 ms, and 708 ms, respectively.

In addition to tail latency, we also report the absolute values of the median latency under the three workloads (Figure 3.14). We observe that Héron has a comparable median latency to the one of DS and C3 when the proportion of requests for large values is small ($1 - 5\%$). When the proportion of requests for large values exceeds 5% we can observe that Héron improves over DS and C3 by up to 35%.

In summary, Héron is particularly more effective in reducing tail latency when the percentage of requests for large values is higher. Compared to C3 and DS, Héron reduces tail latencies by up to 41% without compromising median latency in all the considered synthetic workloads.

3.4.2.3 Consolidated performance improvement

We analyze the average improvement over 8 heterogeneous read-only, read heavy, and update heavy workloads. Figure 3.13 shows the average improvement over all configurations (proportion of requests for large values, and different large value sizes, as detailed in Table 3.1). Héron achieves the highest improvement for read heavy workloads $\approx 23\%$ and the smallest improvement for update heavy workloads $\approx 14\%$. Héron takes scheduling decisions for read requests only, as write requests have to reach all replicas in all cases. It can better cut the tail latency for workloads that contain higher percentages of read requests, such as read-only and read-heavy workload. On the other hand, reading requests take longer time than updating requests in our particular Cassandra setting (as also illustrated in Table 3.1), as writes can be buffered to memory while reads most often have to reach the servers' disks. Hence, a higher percentage of read requests can further stress the system. The average response time of the read-only

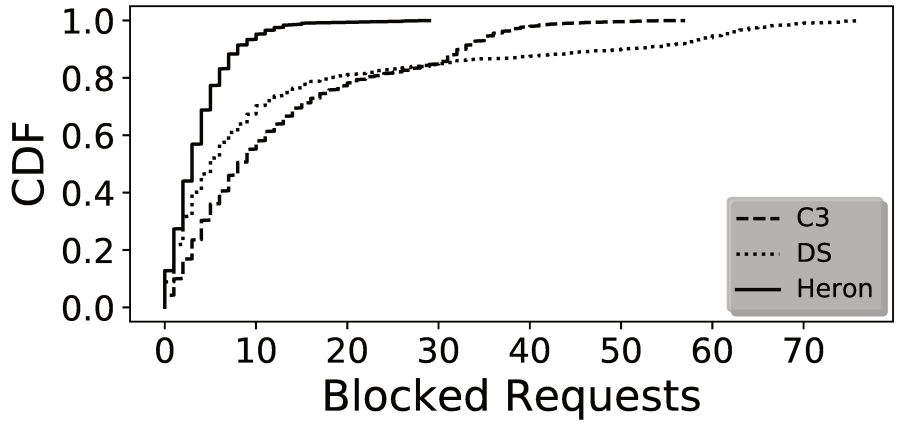


Figure 3.15 – Impact of head-of-line-blocking when small requests are queued behind large requests.

workload is therefore higher than the one of the read heavy workload.

Finally, we show the consolidated observation of the average number of requests for small values scheduled behind requests for large values over all experiments in Figure 3.15. This confirms our initial intuition that C3 avoids overwhelming well-performing replicas with requests compared to DS, but will let requests accumulate behind requests for large values, leading to a large number of blocked requests in most cases. On the other hand, the adaptive strategy used by Héron allows limiting the number of blocked requests in the queues of all replicas, limiting the impact of head-of-line blocking.

3.4.3 Real workloads

We now proceed to evaluate the performance of Héron in reducing tail latency under value sizes distributions obtained from two datasets from WikiMedia and Flickr. The distribution of these sizes is shown by Figure 3.1. We generate three access workloads, update-heavy, read-heavy and read-only, using the same proportions of accesses as for the previous experiments. For each dataset, we compute the median, 95^{th}, 99^{th}, and 99.9^{th} percentile latencies for Héron, DS and C3.

Figure 3.16 present the results for the WikiMedia trace. Héron yields the lowest latency for almost all combinations of latency metrics and workload types, whereas the performance of DS and C3 vary across different combinations. Similar to the synthetic case, absolute latency improvements of Héron are more significant for the higher tail latency, e.g., 99^{th} percentile. Moreover, Héron achieves the best gain for the read-heavy

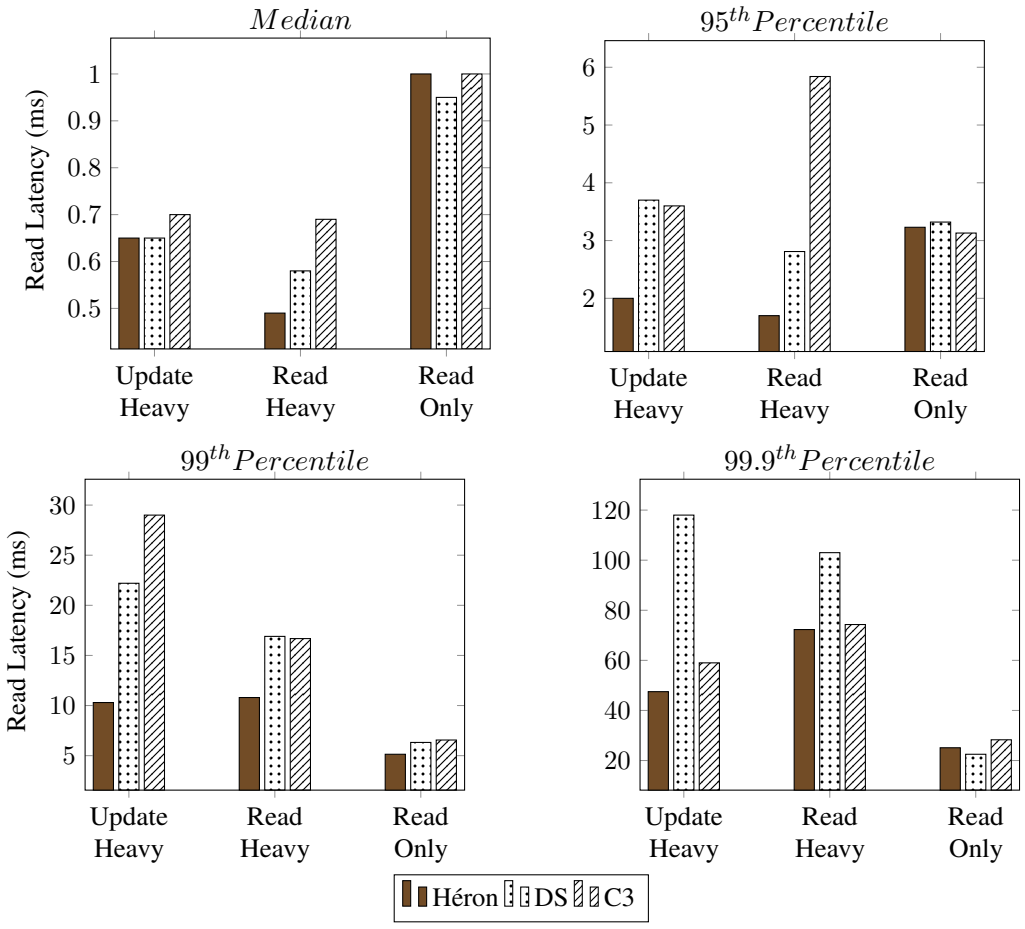


Figure 3.16 – WikiMedia Dataset

workloads, up to 70%. The only case where Héron has inferior performance compared to C3 and DS is for the median of the read only workload. The latency of Héron remains around 1 ms. Actually, Héron achieves rather minor performance improvements in both median and tail latency against DS and CS for read only workload in this dataset.

We present the results for the Flickr dataset in Figure 3.17. We first note that these results show different latency characteristics from WikiMedia, i.e., lower tail latency, though their median latency is in a similar range. This can be explained by the fact that the Flickr dataset value size distribution has a shorter tail, that is, the maximal value for sizes is smaller than for the WikiMedia dataset. Here, Héron achieves the lowest latency for almost all combinations of latency metrics and workload types, except for the 99.9 percentile of read-heavy workloads. The overall improvement compared to C3 and DS is also less significant than with the WikiMedia dataset. Héron is particularly designed to handle the heterogeneous workloads that have a high variance across requests' sizes. When the sizes of requested values have lower variability, the impact of size-aware

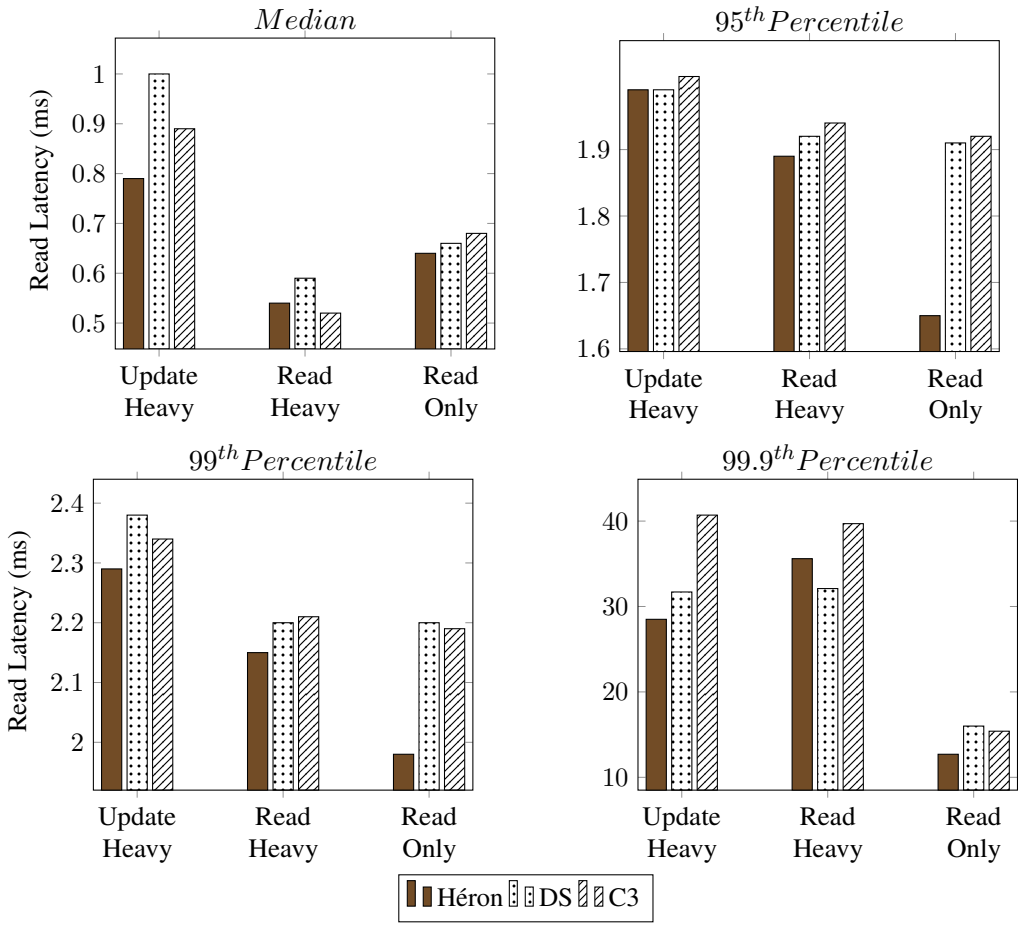


Figure 3.17 – Flickr Dataset

scheduling becomes less visible, even if they are still present. Different from the synthetic data set and WikiMedia, Héron achieves the best median and tail latency improvement for update heavy workloads, compared against read only and read-heavy workloads.

3.5 Related Work

In this section, we will discuss the literature on performance issues in cloud data stores. Prior work has been focused on reducing tail latency through replica selection algorithms. Also, there has been numerous work done in the area of Geo-distributed systems. Apart from this, in cluster environments, a lot of work has been done on proposing better scheduling policies, resource allocation problems and effect of workload heterogeneity on latencies. In this section, we summarize the research contributions related to our work.

3.5.1 Reducing tail latency through replica selection

In the literature, a lot of work has been done on reducing tail latency through replica selection algorithms. Here, we discuss some state-of-the-art algorithms.

3.5.1.1 Dynamic snitching

Dynamic snitching (DS for short) [20] is a replica selection algorithm that is used by Cassandra [17]. All the nodes in a Cassandra cluster organize themselves into a ring based distributed hash table. Every replica in a cluster has some *score*. A *score* is calculated by the integration of one or more metrics. It defines how fast a replica can serve the request. Thus, a replica with the best score serves the next incoming request in minimum latency. In the case of Cassandra, it uses a combination of request latency and estimated load on replicas. It uses normalized smoothed latency and the normalized load of replicas for scoring. To calculate a textitscore, the performance of read requests from various replicas is monitored over time and the best replica is selected based on its recent performance history.

In *dynamic snitch*, snitch is a function that helps to decide which datacenter and rack a read or write should be written and read from. It has to be dynamic since cassandra asks the concerned node for data and the rest of the replicas reply with checksums only. Therefore, among multiple replicas, only one replica (in the case of consistency 1) returns the data. Here, role of *dynamic snitching* comes into play where it has to choose the best replica among multiple replicas to minimize the latency. In *dynamic snitch*, it chooses the best replica in the datacenter based on the network distance and replicas' past read performance.

Scoring of replica is performed periodically at a regular interval. For instance, DS maintains a score for each replica that is updated every 100 ms. The order of replicas is maintained between re-computations and utilized for each request coming on the system in that time interval. Each replica does independent measurements, therefore has its own local view of the network delays and loads on other replicas. All replica scores are reset every 10 minutes, to allow replicas to possibly recover.

The limitation of this approach is that it is solely based on replicas' past read performance, without considering the forthcoming load at each replica (i.e., its queue size). This may lead to overloading recently-fast replicas and to the appearance of bottlenecks in the system, ultimately impacting tail latency.

3.5.1.2 The C3 algorithm

C3 [1] is a replica selection algorithm that improves over DS by handling service time variations among replicas. Due to periodic garbage collection, maintenance activities, periodic background tasks, servers in cloud environments exhibit performance fluctuations in service time. To overcome this, C3 computes replica scores based on both service time and queue size. The score is piggybacked as feedback in response to the coordinator. This score is used by a request coordinator for choosing the replica that is expected to serve the request in minimum waiting time.

C3 tries to avoid *herd* behaviors where a number of clients try to send to the same fast replica server. To overcome this, C3's design has following key components:

1. **Replica ranking:**

   Every replica server sends the product of queue size and service time as feedback by piggybacking in each response to the client. Client keep Exponentially Weighted Moving Averages (EMWA) of queue size and service time to smoothen the signal. Afterwards, client rank and choose the best replica according to the score.

   Since there is a possibility of delayed feedback from the replicas, clients calculate the queue size estimate of each replica by calculating the instantaneous count of its outstanding requests (os_s). Outstanding requests are those for which a response is yet to be received.

2. **Rate control mechanism:**

   Through replica selection, we can't identify the replica server capacity for fulfilling client demands. Sometimes replicas overwhelmed with an excessive number of client requests. To avoid overwhelming of a given replica queue (e.g., because the replica is fast and thus simultaneously selected by several coordinators), C3 uses a rate control mechanism at each replica to limit the arrival of requests. To estimate server capacity, since clients need to adapt performance fluctuations across servers, C3 uses CUBIC congestion-control scheme [40].

   Every client maintains a token based rate limiter for each server through which it limits the requests sent to the server. This rate is termed as *sending-rate* ($srate$). Also, the client tracks the responses received from the replica servers at each interval. This rate is termed as *receive-rate* ($rrate$). By Cubic rate adaption function client compares $srate$ and $rrate$ for server s. If the receive rate is lower than the sending rate then the client decreases the sending rate. Similarly, if the receive rate is higher than the sending rate then client increases the sending rate.

C3 is implemented in Cassandra on a cluster of 15 m1.xlarge instances of Amazon EC2. C3 uses Yahoo Cloud Serving Benchmark (YCSB) [32] to generate datasets and run the workloads. To generate datasets, C3 inserts 500 million 1 KB size records

through YCSB and runs 10 million operations with Zipfian distribution. Results show that C3 significantly improves tail latencies compared to DS [1]. Since the incoming request size was assumed to be the same, C3 does not perform well with heterogeneous workloads (figure 3.2).

3.5.1.3 Replica selection in geo-distributed systems

Geo-distributed systems are another area of research in replica selection algorithms where a client sends a request to the nearest replica among multiple datacenters to minimize network delay. Nowadays many cloud providers have deployed their datacenters in multiple geographic regions. To improve QoS, data is replicated on multiple datacenters. Data is retrieved by the client from the nearest datacenter to minimize the latency. Most of the cloud providers often face performance issues due to the poor selection of servers [41]. Since all the datacenters differ in their service availability, cost of usage, performance etc.

To select the best replica, Geo-replicated data stores continuously monitor the network and system state of all replicas. Performance in these systems improves by choosing a better choice of replica selection algorithms. Systems like Cassandra and MongoDB uses such algorithms in Geo-distributed systems.

To choose the replica, the first step is to evaluate all replica metrics such as load measurements and latency. The second step is to set the score of each replica based on the measurements done in the first step and define the *goodness* of a replica. The third step is to filter the replicas that do not have the requested data since sometimes not all replicas have the required data to perform a certain query. Afterwards, the final step is to select the subset of replica based on the consistency requirements.

In Cassandra, as explained in the previous section (§3.5.1.1) it uses dynamic snitching to choose the nearest replica. Normally the scoring of replicas is done on the basis of physical locations. Therefore, replicas which are on the same rack and in the same datacenter would be preferred. After that replica scoring mechanism is followed for replica selection in the datacenter. Similarly, in MongoDB network RTT is considered to select the replicas. All the replicas that are farther from the default threshold of 15 ms from the nearest are sorted out from the list and finally, one random replica is selected from the list. Unfortunately, internet traffic, routing decisions changes very frequently which results in the latency and loss rate change dramatically.

In Recent work [41], the author presented GeoPerf which automates the process of testing the best replica selection algorithm with the help of symbolic execution and lightweight modeling. Symbolic execution replaces input values by symbolic variables rather than numeric values of an application. The main purpose of symbolic execution is to test and validate all operations by traversing all possible code paths and generate

test cases for encountered errors.
In Geo-distributed replica selection algorithms, symbolic execution identifies test inputs (i.e. latency measurements) and traverse all possible code paths. Afterwards, the symbolic execution engine examines code paths and look for bugs in an effort to find a case where a given algorithm doesn't perform well compared to the referenced one. However, finding a case is still difficult since it needs to symbolically execute the entire distributed system and requires inside knowledge of the system, code modifications, and computation resources. However, symbolic execution doesn't have a notion of continuous time, therefore, it is difficult to evaluate replica selection choices.

Apart from this, Pisces [42] (Predictable Shared Cloud Storage), a key-value storage service provides per-tenant fair resource allocation. Pisces uses replica selection in a weight sensitive manner to make the fairness easier. It distributes load over replicas by getting implicit feedback of per node-latencies.

3.5.2 Taming latency in cluster environments

Besides replica selection, other mechanisms have been proposed to reduce tail latency. Dean and Barroso [5] analyze the reasons for latency variability and describe a set of tail-latency tolerance techniques implemented in Google's large scale systems. They discuss short term adaptations in which they focused on hedged requests where a user sends a request to multiple replicas and uses the results by the replica which responds first. A client cancels remaining requests once the response is received. Though these adaptations typically add unnecessary load on servers.

CosTLO [43] reduces high latency variance by issuing requests redundantly. D-SPTF [44] and RobinHood [9] adapt caching mechanisms to reduce tail latency. Maintaining low tail latency is difficult since incoming requests are complex and consist of multiple operations. RobinHood [9] reallocates cache resources from cache-rich (backends which don't affect request tail latency) to cache-poor (backends which affect request tail latency) to maintain low tail latency. By doing this, it improves overall tail latency and it has very little overhead. In D-SPTF [44], a request is sent to a server and if the data is in a cache, it will respond otherwise the request will be forwarded to the replicas.

Zoolander [12] is a key-value store that serves low latency with strict service level objectives (SLOs). It uses replication for predictability and uses redundant accesses to mask outlier response times. When system resources are under-utilized, it uses an analytical model to scale-out through replication. Similarly, when they are heavy-utilized it uses traditional approaches to scale-out. Zoolander provides accurate predictions to find better replication policies. By doing so, zoolander improves latency for key-value stores.

Mantri [45] tries to resolve the problem of skew-tolerance where some slower tasks in

the data-parallel application can delay the overall job completion time. Mantri uses network-aware task placement and protects the task output based on cost-benefit analysis. Through this, Mantri reduces job completion time by 20% on production cluster that supports Bing.

Bobtail [8] tries to reduce the tail latency in cloud environments. It shows that virtualization in Amazon EC2 increases the tail latency by a factor of two to four. It states that the root cause of increasing response time is a property of nodes rather than the network. Bobtail detects and avoid the bad behaving VMs without penalizing node instantiation. Li et al. [46] explore the hardware, OS and application-level causes behind tail latencies. Their work explain why queuing delay increases the latency whereas parallelism improve the tail latency. Through extensive experiments, authors improved Memcached tail latency with a two orders of magnitude.

Taming latency via resource allocation. Several users or tenants share same physical server or network infrastructure in the cloud to use common services. However, tenants often face performance fluctuations. Therefore, to improve the performance predictability for a shared storage system, Pisces [42] (Predictable Shared Cloud Storage), a key-value storage service provides per-tenant fair resource allocation. Each tenant global weight is set according to the tenant's service level objectives (SLO). Pisces does the fair sharing between the tenants in a datastore. PARDA [47] also focus on the problem of storage bandwidth sharing to give fairness between the clients. It uses IO latency to observe load and uses a FAST-TCP control mechanism to limit the number of IO requests per storage client. Prioritymeister [11] combine priorities and token-bucket rate-limiter to provide tail-latency QoS for bursty workloads in shared-networked storage. mClock [48] discusses the IO resource allocation in a hypervisor which provides per-VM QoS.

Cake [49] is a reactive feedback control scheduler for distributed storage environments to achieve high throughput and bounded latency. It considers a two-level scheduling approach for shared storage systems where first-level schedulers control the consumption to individual resources such as CPU, disk etc. These schedulers do various tasks such as split large requests into small chunks, limit the number of outstanding requests and provide mechanisms for differentiated scheduling. Second-level schedulers run a slower feedback loop that adjusts resource allocations at the first-level schedulers and maps high-level SLO requirements. To conclude, Cake allows latency sensitive workloads while ensuring that tail latency SLOs are met.

3.6 Summary

In this chapter, we addressed the problem of tail latency under heterogeneous workloads in key-value stores through replica selection. We study the approaches that focus on the performance in cloud data stores. For replica selection algorithms, C3 and DS are two major replica selection algorithms in literature for key-value stores. An in-depth study of these algorithms has highlighted the fact that these algorithms don't perform well under heterogeneous workloads. We proposed Héron, a replica selection algorithm that deals with requests accessing large values by avoiding the head-of-line-blocking of requests accessing small requests behind these requests. The result is an improved overall performance of the key-value-store for a wide variety of heterogeneous workloads. Our experiments with heterogeneous YCSB workloads in a Cassandra based implementation showed that Héron outperforms state-of-the-art algorithms (C3 and DS), reducing tail latencies by up to 41% and reducing the median latency by up to 31%.

4

Task-aware Scheduling for Improving Tail Latencies

Contents

4.1 Problem definition

In this chapter, we consider the question of performance bottlenecks in cloud data stores in the case of scheduling multiget requests under heterogeneous workloads. It is very difficult to deliver consistent latency for interactive services due to the varying degree of request fan-out i.e. each request has different number of sub-request. If the requests are with a larger fan-out, it is most likely that it would be affected by long tail latencies [5, 19]. Therefore, to serve high fan-out user requests or request asking for several keys data elements, multiget APIs batch multiget read operations in key-value stores [33, 50, 51, 52]. A multiget request finishes when all of its operations complete. Therefore, the response time of a request depends on the response time of slowest operation in that multiget request.

In practice, multiget requests vary in the number of accessed keys and value size. A workload analysis at Facebook [33] shows that a request contains an average 24 keys while 5% of the requests contain more than 95 keys. Another statistics from SoundCloud trace presented in Rein [16] shows a heavy-tailed distribution of keys in which 40% of the requests involve multiple keys with an average size of 8.6 keys and the maximum number of keys reaches up to $\sim$2000 keys. Similarly, a production workload analysis at Facebook [18] for key-value stores show that value size typically ranging from a few bytes to MBs. Such variability in requests requires an efficient scheduling algorithm that is not affected by bottleneck operations on backend servers.

Several approaches [1, 16, 19, 30, 50, 51, 53] have been used to reduce the latency in large scale distributed systems. These approaches include changing the level of multiplexing in the network to avoid head-of-line-blocking [19], scheduling based on bottleneck completion time [53] and client-side priority assignment on the basis of bottleneck operations [16].

Processing multiget requests on backend servers pose many challenges. First, in an online setting when a request is issued service demand is unknown for that request i.e. we do not have any prior knowledge of the request and the request is processed in non-clairvoyant fashion [54]. Therefore, it is very difficult to schedule a request by just knowing its key. A first challenge is thus to be able to predict the value size based on the key with minimal operational overhead. Then if we assume that value size is correctly estimated, the second challenge is to coordinate the operations (*sub-requests*) which go on different servers. The problem becomes more challenging when keys access heterogeneous value sizes. Therefore, another challenge is to provide an efficient scheduling algorithm that can estimate the bottleneck operations and schedule them on uncoordinated backend servers in such a way that creates minimal overhead and reduce latencies at the tail.

Potential gains In order to understand the problem, we consider the example given by Figure 4.1. A multiget request mget(A, B, C) is coming on the system which requests values of keys A, B, C. Each key of a multiget request might have different value size which results in the different completion time of each multiget request. Therefore, the completion time of a multiget request depends on the number of operations and the value sizes associated with each operation. In the example (left figure), three servers 1, 2, 3 are currently holding the value of keys (A, B), (C, D) and (E, F, G, H, I) respectively. For the sake of simplicity, we assume that all the servers having the service time of 1 operation per unit time for a small value and 5 unit time for a large value. In the figure, a small box represents a request to a small value and large rectangle boxes (i.e. D) represent requests to large values. A request is divided into number of sub-requests called *opset* according to the data stored on the servers. In the figure 4.1, for a request mget(A,B,C), (A,B) and (C) are the two opsets. Considering the given scenario, mget(A, B, C), mget(D, E), mget(F, G) and mget(H, I) will complete in 2, 6, 3 and 5 time units respectively with an average response time of 4 time units (*left sub-figure*).

In an ideal scenario, it would be beneficial to consider the bottlenecks between the opsets and procrastinate some operations before scheduling it on the server. To reach this objective, we calculates the delay allowance of each opset by considering their approximate total execution time and procrastinate some opset. Since a multiget request will complete when all of its operations complete. Therefore, in mget(D, E) request D take 6 unit time whereas request E will take 1 unit time. Thus, we have delay allowance of 5 unit time for request E. We procrastinate this opset to get the benefit of delay allowance and let other requests execute in that time. In this scenario, mget(A, B, C), mget(D, E), mget(F, G) and mget(H, I) will complete in 2, 6, 2 and 4 time unit respectively with an average response time of 3.5 time unit (*right sub-figure*).

Contributions. We present TailX, a task aware multiget scheduling algorithm that reduces tail latencies under heterogeneous workloads i.e. i) multiget requests of different value sizes where values are very large in some multiget requests ii) multiget requests of different number of operations where number of operations (*sub-requests*) in some multiget requests are very large.

We first devise a method to estimate the approximate response time. To overcome this, we use a *size estimation module* to provide a estimation of response time. It keeps track of the keys that corresponds to the large value. Once a request has been identified as accessing a small (large) value, TailX calculates the delay allowance of operations by which some operations can procrastinate for some time to give better flexibility for other operations to execute. It inserts the delay allowance in each bottleneck operations as metadata. Delay allowance is calculated on the basis of the completion time of each operation set that goes on the different servers. Our insight is to consider the number of operations and value size to calculate the approximate execution time. Afterwards,

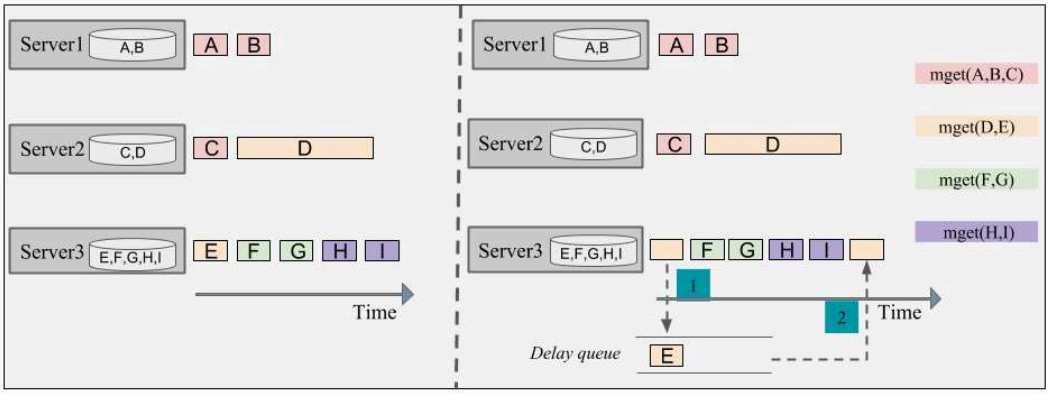


Figure 4.1 – An example scenario. *Left:*Requests assigned to server facing delayed response time. *Right:*Procrastinate opsets into delay queue to take benefits of delay allowance

operations that are finishing earlier can wait for other operations which are taking time since a multiget request finishes when all of its operations complete. In procrastinate time, some other requests can be executed thus decrease latencies at the tail. To take advantage of procrastinate time, rest of the subsequent operations schedule on the storage server.

To demonstrate the feasibility of TailX, we implement it in Cassandra [17] — a popular key-value store which is widely used by service providers. We compared TailX with Rein [16], a state-of-the-art algorithm in a cluster of 16 machines on Grid'500 [31]. We evaluate TailX on variable configurations of a synthetic dataset which is inspired by Facebook [18]. We use YCSB [32] to generate workloads which contains various proportions of accessed keys and value size. We evaluate TailX with synthetic dataset where 20% of multiget requests contain 100 operations (*long request*) whereas 80% of multiget requests contain 5 operations (*short request*). In another evaluation, we consider the dataset with each multiget size of 20 operations in which 80% multiget requests contain 1 KB value size whereas rest 20% multiget request contains 10% operations of large value (i.e. 2 MB) size. We vary the proportion of value size from 10% to 50%. We show that compares to Rein, TailX improves the median latency by 75% as well as tail latency by up to 70%.

The remaining of this chapter is structured as follows. We first detail the state-of-the-art related to task aware scheduling in distributed systems (§4.2). Next, we explain the challenges to design a scheduling algorithm for multiget requests under heterogeneous workloads (§4.3). Next, we present a detailed description of TailX and

its key components (§4.4). Afterwards, we present its implementation and performance evaluation (§4.5). Finally, we conclude the chapter (§4.6).

4.2 Related Work

Workloads which are coming on system have a great impact on latency. Before discussing the state-of-the-art work, here we discuss some statistics of production workloads.

Web workloads. Atikoglu et al. [18] described the workload analysis of a Memcached [33] traffic at Facebook. This is perhaps the most detailed work yet for the analysis of key-value store workloads. It studies 284 billion requests over a period of 58 days for five different Memcached use cases. Several features of the workloads were outlined by the authors but we concentrate our study on value size distribution. It presents the CDFs of value size in different Memcached pools. ETC pool is the largest and most heterogeneous value size pool where value sizes vary from few bytes to MBs. It represents the general cache usage of multiple applications. In ETC, 40% of the requests contain value size up to 11 bytes while few values range up to 1 MB. These kinds of workloads leave less cache space for smaller values. Though requests having small values dominate all workloads in number and overall weight.
All workloads in the paper exhibit long-tail distribution and a small portion of keys are appearing in most of the requests. Repeating keys provide the opportunity to cache them in the first place. Some keys are repeating a number of times. For example, in ETC pool, 50% of ETC keys appear in only 1% of requests i.e. they do not repeat many times. Instead, in some pools, keys are repeated in millions of requests per day.

Another statistics from the analysis of a 30-minute production trace from SoundCloud [16] in which multiget requests exhibit variations in key popularity and their sizes. It shows the heavy-tailed distribution of multiget requests in which $\sim$40% of requests involve more than one key while the average size is 8.6 keys. The maximum size of a multiget request reaches up to $\sim$2000 keys.

The trace shows the heavy-tailed distribution of key access frequency in which most keys are accessed once but few keys are accessed up to 1000 times. This shows the variations in multiget requests which are coming on systems.

Task scheduling for large computation jobs is another area of related research. A task is characterized into two features: i) the task size and ii) number of flows (sub-tasks) per task. These two features are very critical while improving the performance in data-parallel clusters. Task-aware scheduling is categorized as follows:

4.2.1 Network-specific

Orchestra [55] enables global control to improve performance across multi-node transfers. The Orchestra architecture contains a cross-transfer scheduling policy that prioritizes ad-hoc queries over batch jobs. It uses weighted fair sharing where each transfer is assigned a weight and each link in the network is shared proportionally to the weight of the transfer. Since scheduler without apriori knowledge compromise on performance to avoid head-of-line-blocking.

Baraat [19] is a decentralized task-aware scheduling system which dynamically changes the level of multiplexing in the network to avoid head-of-line-blocking. It uses task arrival time to assign a globally unique identifier and put a priority for each task. All flows of a task use this priority irrespective of the network they traverse. Therefore, based on the identifier, switches can make consistent decisions even though they are not in coordination and improves the chances that all flows of a task make progress together. Since task flows in a serial fashion, hence they do similar behavior across the network. It avoids the problem of centralized scheduling such as scalability or fault-tolerance etc.

Varys [53] is another coflow scheduling system that decrease communication time for data-intensive jobs and provide predictable communication time. It uses *Smallest-Effective-Bottleneck-First* (SEBF) and *Minimum-Allocation-for-Desired-Duration* (MADD) heuristics for guaranteed coflow completions in a timely manner. In SEBF, scheduling of a coflow is based on it's bottleneck completion time. In MADD, it slow down all the flows to match the longest flow i.e. the flow that takes longest completion time to finish. It schedules the tasks in FIFO order such that small tasks are not starved behind large tasks. Varys assumes complete prior knowledge of coflow characteristics such as a number of flows, their sizes etc.

Aalo [56] is another scheduling policy that improves performance in data-parallel clusters without prior knowledge. Aalo separate coflows into a number of priority queues. Aalo's non-clairvoyant scheduler is starvation free which performs prioritization across queues and schedules coflow in FIFO order. It makes coflows practical in presence of task failures and mitigation techniques.

To improve the performance in datacenters, pFabric [57] decouples flow scheduling from rate control mechanisms. Each packet has a priority and whenever a port is idle, the packet which has the highest priority is dequeued and sent out.

4.2.2 Redundancy-specific

Redundancy is a powerful technique which is used to reduce the latency in large scale distributed systems. In redundancy, clients initiate an operation multiple times on multiple servers. The operation which completes first is considered and rest of them is

discarded. If a client duplicates the request and considers the response which receives earlier, then this scenario decreases the mean latency but system utilization has doubled.

Vulimiri et al. [58] characterize the scenarios where redundancy improves latency even under exceptional conditions. It introduces a queuing model which gives an analysis of system utilization and server service time distribution. It says if we assume client-side replication cost is low then server-side below threshold replication always improves mean latency. Also, if there is high variability in service-time distribution then performance achievement would be higher in this case.

Sparrow [10], a stateless distributed scheduler that adapts the power of two choices technique [59] by selecting two random servers. It put the tasks on the server which has fewer queued tasks. Sparrow [10] uses batch sampling where instead of sampling each task it places m tasks of a job on least loaded randomly selected servers. This approach performs better for parallel jobs since they are sensitive to tail task wait time. It also includes the *late binding* since the power of two choices suffers from performance issues due to considering the server queue length which is a poor indicator of wait time. By *late binding*, it delays the tasks assigned to machines until worker machines are ready to run the task. Overall, all these approaches improve load-balance.

4.2.3 Task-aware schedulers

There has been numerous work on heterogeneous workloads which typically consists of many numbers of short jobs and a very few long jobs. Large jobs are those which consume the bulk of resources whereas short jobs are latency sensitive. Long jobs can sustain long latencies but they suffer from poor scheduling placements. Therefore efficient scheduling mechanisms are needed to improve performance and decrease latency in these systems. As seen in table 4.1, top 10% of the Google trace jobs account for 83.65% of task-seconds (product of a number of tasks and average task duration). Their average task duration is 7.34 times higher than the remaining 90% of the jobs. This kind of pattern emerges in rest of the traces.

Hawk [2] and Eagle [34] are two systems proposing a hybrid scheduler that schedules jobs according to their sizes. In Hawk [2], long jobs are scheduled using a centralized scheduler while small jobs are scheduled in a fully distributed way. Since long jobs are in fewer number than small jobs, their centralized scheduling allows a good placement of jobs without introducing coordination bottlenecks. It reserves a portion of a cluster to run exclusively for small jobs and remaining part of a cluster is left for running large jobs. Short jobs can be scheduled on any server. This allows short tasks to take advantages of idle servers which are available. If long tasks are scheduled on any server, there might be a possibility of head-of-line-blocking. Apart from this, Hawk uses randomized work stealing to allow idle nodes to steal short tasks that are

Workloads	% Long Jobs	% Task-Seconds
Google 2011	10.00%	83.65%
Cloudera-b 2011	7.67%	99.65%
Cloudera-c 2011	5.02%	92.79%
Cloudera-d 2011	4.12%	89.72%
Facebook 2010	2.01%	99.79%
Yahoo 2011	9.41%	98.31%

Table 4.1 – Fraction of long jobs out of total jobs in heterogeneous workloads consuming bulk of resources (adapted from [2]).

queued behind long tasks. To differentiate between long and short tasks, it uses an estimated runtime of jobs. This estimated runtime is compared against a threshold value. Jobs whose task runtime are below the threshold value are considered as short jobs and scheduled in a distributed way. The threshold value is based on the statistics of past jobs.

Eagle [34] is another hybrid scheduler, which schedules short and long jobs same as Hawk [2] to avoid *head-of-line-blocking*. A job is a set of tasks that can run in parallel on several worker nodes. Every task of a job is assigned to worker nodes to schedule a job. The completion time of a job is the time when all the task finishes. Therefore, it is the maximum completion time of any task in a job. Identification of long and short jobs are classified by the average execution time of the tasks. If the average execution time of tasks falls above (below) the threshold, then the job is considered as long (short).

It introduces sticky batch probing to achieve better job scheduling. In sticky batch probing, scheduler places a probe on a node and it stays there until all the tasks of a job are completed. It uses Least Work Left (LWL) scheduling policy to schedule long jobs. Finally, Eagle is implemented as a part of Spark-plugin and it improves the performance by avoiding head-of-line-blocking.

Omega [60] is a shared-state scheduler in which a separate centralized resource manager maintains a shared scheduling state. In general, worker node or distributed scheduler update this state. Based on the shared state, schedulers make a scheduling decision and update the state. In Omega, there is no central resource allocator. Therefore all the resource allocation decision happened in the schedulers themselves. There is a master copy of the resource allocations called *cell state*, given to each scheduler. Each scheduler has the local private copy of cell state which is used to make scheduling decisions. The scheduler can see the entire cell state and has all the permission to claim any resources. Once a placement decision is made, scheduler updates the shared copy of the cell state.

4.2.4 Request reissues and parallelism

Kwiken [7] optimizes the end-to-end latency using a DAG of interdependent jobs. It further uses latency reduction techniques such as request reissues to improve the latency of request-response workflows. Also, there has been work done on request parallelism.

Haque et al. [14] propose solutions for decreasing tail latencies by dynamically increasing the parallelism of individual requests in interactive services. But parallelizing each request is challenging since service demand of a request is unknown during its arrival. Therefore it is very difficult to parallelize requests without knowing the service demand. Apart from this, parallelizing short requests don't improve tail latency. To overcome this, *Few-to-Many* (FM) selectively parallelizes the long running requests since that are the ones contributing the most to the tail latency. FM scheduler determines how much parallelism it needs to add based on individual system load and request progress. Therefore, it improves the state-of-the-art predictor in terms of accuracy and efficiency to support selective parallelization. FM parallelizes long requests on all servers thus also including slow servers.

Recent efforts [15, 61] show that it is challenging to schedule tasks during the arrival of variable size jobs. These works try to predict the long-running queries and parallelize them selectively. Instead of targeting the more general problem of predicting job sizes, which in some cases involves costly computations.

Jeon et al. [15] focus on the parallelizing long running queries which are few compared to the short ones. It aims to achieve consistent low response time for web search engines. It mentions a statistics from Bing which shows that 85% of the queries take less than 15ms while few queries are very long which take up to 200ms. Specifically, the average execution time is 13.47 ms while 99th percentile latency is 200 ms which is 15 times more than average latency. Therefore, parallelizing long running queries is a fair solution to reduce query execution time. To achieve this, an effective parallelization scheme and efficient prediction of query execution time are needed with accuracy. For prediction, it improves the term and query features of the predictor. It also does query rewriting to improve accuracy. It uses predicted query execution time to parallelize long-running queries.

Apart from this, there has been numerous work on size aware scheduling. Authors [62] proposed *unfair scheduling* to improve the performance of web servers. The idea is to give priority to requests with the short remaining time.

4.2.5 Multiget scheduling

In key-value stores, multiget scheduling is a common pattern for scheduling requests efficiently. Systems like Cassandra [17], MongoDB [23] offer such algorithms in these

systems. Rein [16] uses a multiget scheduling algorithm to schedule the multiget request in a fashion that can reduce median as well as tail latency. In the next section, we discuss the Rein [16] multiget scheduling.

4.2.5.1 Rein scheduling

When a multiget request comes to a server it splits into the group of sub-requests called *opsets* on the basis of hashing mechanism 2.1.2 followed by the nodes. Normally all the nodes have some hash range and keys split according to that hash range. Next, it predicts the bottleneck opset i.e. opset which creates head-of-line-blocking. Estimation of bottleneck opset is based on the number of operations in it. Opset which has the highest number of operations is considered as bottleneck opset. Therefore, there is a probability of more than one bottleneck opset if they have the same number of operations.

Rein uses a client-side priority assignment on the basis of bottleneck opset. The idea is to prioritize operations with shorter execution time compared to larger ones. This reduces the head-of-line-blocking of requests and improves latencies. It inserts the priority corresponds to the number of operations in bottleneck opset to prefer the opsets with shorter bottlenecks. Each priority is inserted as meta-data in each operation of a given opset. Thus, all operations of an opset have the same priority. Afterwards, each opset is sent to the corresponding server responsible for that data. Scheduling of each operation is done on the server side.

In high-level view, each server servers the requests according to their assigned priority. Since a simple priority queue might suffer from starvation, it uses two policies which include the Shortest Bottleneck First (SBF) and Slack-Driven Scheduling (SDS). The details are as follows:

1. **Shortest bottleneck first:**
   In SBF, every operation of a multiget request has a priority which corresponds to the cost of the bottleneck opset. The intuition is to prioritizes requests which have smaller bottlenecks to minimize head-of-line-blocking. This scheme is similar to Shortest Job first (SJF), apart from the request completion time that is determined by the last operation of the request. It favors smaller multigets compare to the large ones and schedule them ahead of larger multigets.

2. **Slack driven scheduling:**
   In SDS, it assigns the priority for every operation x of a non-bottleneck opset O as $(cost(x) + slack(x))/size(O)$. It deprioritizes the operations based on how long they can afford to be slacked. This policy uses server capacity efficiently by prioritizing servicing requests compare to the bottle-necking the request.

The canonical priority queue implementation has some significant performance drawback due to lock contention in multi produce/consumer settings. Therefore, Rein uses

multiple queues at each server to make it more efficient. At backend nodes, there is a multilevel queue with K FIFO queues $Q = \{Q_1, Q_2, \ldots, Q_K\}$ to serve the requests with priorities. In multilevel queue, requests execute according to their assigned priorities. However, each queue has different dequeue rates in a multilevel queue to avoid starvation due to priority scheduling. A queue is assigned with highest queue rate w_{Q_1} and successive queues have the lower rates in a similar fashion. The scheduler uses Deficit Round Robin (DRR) scheduling for dequeuing operations with assigned dequeue rates. The higher priority queue has higher dequeue rate while lower priority queue has lower dequeue rate. To assign priority of a multiget request, Rein calculates the cost of each bottleneck opset B and opset with higher cost are assigned to lower priority. For non-bottleneck opset op, it calculates the ratio between the cost of non-bottleneck and bottleneck opset and loop over all the queues to minimize the absolute difference between cost ratio and dequeue rate ratios. w_B is the rate of the queue to which bottleneck opset B is assigned. It is defined as below:

$$Q_{min} = \underset{q \in Q}{argmin} \parallel \frac{cost(op)}{cost(B)} - \frac{w_q}{w_B} \parallel$$

Rein is implemented in Cassandra, a widely used key-value store. It has been evaluated on 16 m3.xlarge AWS EC2 instances. To generate the workloads, YCSB [32] is used and run on a separate node. The consistency level is set to 1. For workloads, it uses SoundCloud trace which is not publicly available. For synthetic dataset, it uses constant multiget size of 50. Results show that Rein significantly improves median, as well as tail latencies, compared to other state-of-the-art algorithms. It demonstrates that distributed scheduling provide beneficial outputs in the context of key-value stores and without requiring coordination. Rein can be applied to any key-value stores such as Memcached [33] and MongoDB [23] or any other where same scheduling heuristics can be applied.

4.3 Challenges

Before going into detail of TailX, we address the challenges to design such scheduling algorithms. Here are the challenges associated with multiget scheduling under heterogeneous workloads:

4.3.1 Scheduling without complete knowledge is hard

Scheduling multiget request in a storage system is challenging. Requests that are coming on a system have multiple operations with varying number of keys and value sizes [5, 16, 18]. Server exhibits high tail latencies for these workloads due to the different execution time of operations. In an online setting, when a request is issued to a key-value store, the service demand or response time of the request is unknown.

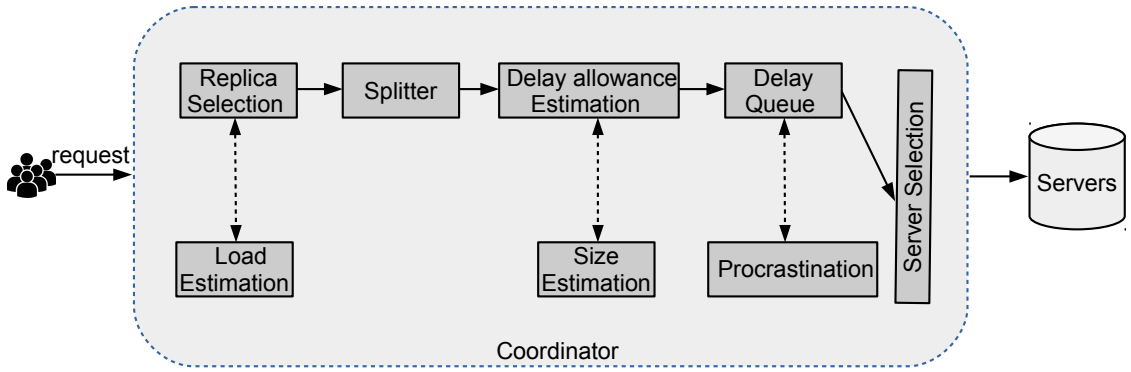


Figure 4.2 – Overview of TailX.

To provide better scheduling algorithm, it is required to know the service demand. So, the first challenge to address is: how to predict the approximate response time of a request by knowing only its key? Further, considering the high-performance constraints of key-value stores, the mechanism set up to estimate the size of a given value must be extremely efficient. The problem is challenging since scheduling is done in non-clairvoyant fashion [54].

4.3.2 Need for coordination

Then, we assume that the approximate response time of a given request can be correctly estimated at request time, the second challenge to address is: how to provide an efficient scheduling algorithm which can estimate the bottleneck operations and schedule them on uncoordinated backend servers in such a way that creates minimal overhead and reduces tail latencies. Since all the operations have to wait for the slowest operation, the scheduling algorithm should schedule the operations in a way that synchronize the approximate execution time and complete at the same time.

4.4 TailX design and implementation

An overview of the architecture of TailX scheduling is given by Figure 4.2. In the figure, when a request is issued, the node acts as a coordinator. Coordinator first does the *replica selection* where it selects the best replica out of total target replicas based on the past read performance of replica servers. An appropriate replica selection mechanism (Dynamic snitching [20]) is applied to select the best replica.

Afterwards, the request goes to a *splitter* where it is split into sub-requests (opsets) by a partitioner (Murmur3 [22]). This splitting is based on the hash based token range holding by the server as explained in chapter 2.1.2. This splitting of keys is the same as used in Cassandra for distribution of keys. The number of operations and value sizes associated with keys vary in these opsets. Therefore, to correctly estimate the total execution time of each opset, one needs to identify the operations which are taking a long time. Therefore, to identify these operations, it passes through *size estimation* module. The objective of this module is to estimate whether a given operation will access a small or a large value. It keeps track of keys that associated with large values and store the keys of those operations.

Once the value size of an operation is identified, *delay allowance estimation* module estimates the cost of each opset i.e. approximate total execution time and calculate the approximate delay allowance occurred by each opset. This delay allowance is inserted as metadata in each operation of an opset. After delay allowance assignment, opsets go through the *delay queue*. The objective of this step is to procrastinate each opset which has delay allowance and let other requests execute in that time. If an operation has delay allowance then it inserted in a delay queue with given procrastinating time. The operations reside in the delay queue until the given procrastinate time expires.

Finally, operations go to the required server which is holding the data. Once the operations finish, they return the data to the coordinator.

We present in the following sections the details of all proposed modules. First, we present the replica selection mechanism based on the load estimated among servers (§4.4.1). Next, we describe the request splitting based on the data storage (§4.4.2). Afterwards, we explain the delay allowance policies including delay estimation of operations and scheduling mechanism (§4.4.3). Finally, we explain the server selection (§4.4.4).

4.4.1 Load estimation and replica selection

The operations of a multiget request select the target replicas according to the hash-based mechanism followed by the replica server (details in 2.1.2). The number of replicas depends on the replication factor followed by the storage systems. Afterwards, a replica selection algorithm (Dynamic snitching [17] which considers past read performance of the replicas) is applied for scoring the replicas and a faster replica is chosen to complete the operation. The role of this component is to select the replica that is expected to serve a given request faster than other replicas.

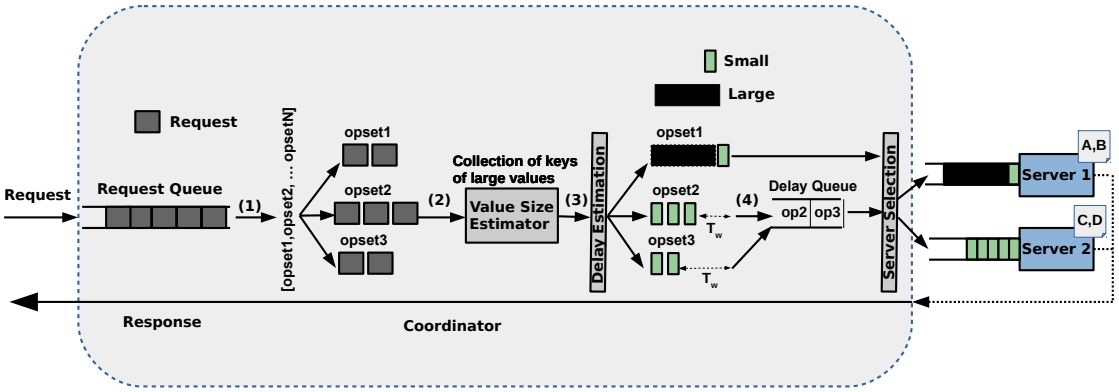


Figure 4.3 – Operating principle of TailX scheduling.

4.4.2 Request splitting

In a key-value store, all storage nodes are divided into hash based token ranges. After selecting the intended replica, request splits into number of sub-requests called *opsets* according to the partitioner (e.g. Murmur3 [22]). Each opset goes to a different replica server and contains a varied number of operations with different value size. Our goal is to schedule the operations in a way that can complete each opset at the approximately same time. This gives better flexibility to other requests to execute in that time.

4.4.3 Delay allowance policies

The algorithms for delay allowance policies are described in Algorithm 2 and Algorithm 3. The role of these algorithms is to procrastinate the opsets which are finishing earlier than the other opsets.

Every opset has different completion time due to the variations in value size and number of operations in it. Therefore, some operations of an opset have to wait for bottleneck operations. This results in increasing the latency of the overall request.

To overcome this situation, the delay allowance module calculates the cost of each opset (*opcost*) i.e. opset execution time on the server. Calculation of the opset cost is based on the value size estimation since we need to know the number of operations for large values (N_L) in each opset. The operations of large values are the sole reason of inflating the operation cost. Therefore, we match the keys of large value to the keys stored in Bloom filter [36] (step 4 of Algorithm 2). Next, it calculates opset cost of each opset based on the request service time for small value (T_S) and request service time for large value (T_L) (step 5 of Algorithm 2). Afterwards, it calculates delay allowance T_w

((step 8 of Algorithm 2)) and tag the allowance to each opset. Finally, it procrastinates operations which has delay allowance (step 11 of Algorithm 2) otherwise send the opset to the corresponding replica server. In Algorithm 3, if the delay allowance time has finished then the request is dequeued and sent to the corresponding replica server.

4.4.3.1 Delay allowance estimation

The role of delay estimation is to estimate the approximate execution cost of each opset and calculate the approximate delay allowance which can be occurred at each opset. Calculation of delay allowance is based on the value size estimation of each operation.

Value size estimation. An important question that TailX addresses is to determine whether an operation will access a large or a small value. In this context, one may start by wondering how to set a threshold (say THR_L) such as values above this threshold are considered as large by TailX. We assume that this choice is application dependent and that it is up to the database administrator to set the value of THR_L according to the data distribution over her database. For instance, for a database containing text messages (in the order of few KBs), photos (starting from hundreds of KBs), a database manager may decide to consider as large all the values corresponding to photos and thus setting THR_L to few hundred KBs. Above this value, all requests will be considered as large by TailX.

Considering a given value for THR_L, TailX uses Bloom filters to keep track of keys corresponding to large values. A Bloom filter is a vector of m bits initially set to 0, with an associated set of k hash functions (generally $k \ll m$). Inserting an element in the Bloom filter is done by hashing the element (in our context the key contained in the request) using the k hash functions and setting the corresponding bit positions to 1. Testing the presence of an element in the Bloom filter is done similarly by hashing the element using the k hash functions and testing whether all the corresponding bit positions are set to 1. Querying a Bloom filter may lead to false positive answers but will never lead to false negative ones. The false positive rate depends on the size of the vector, the number of used hash functions and the maximum number of elements to be inserted in the set.

After identifying keys which corresponds to the large value, it calculates the opset cost i.e. how much time the opset will take to execute. To estimate the opset cost ($opcost$), it calculates the service time of operations for large values (T_L) and small values (T_S). Afterwards, it multiplies them by their respective number of operations to get the overall cost of the opset.

Further, it calculates the delay allowance (T_w) for each opset. Delay allowance is calculated on the basis of cost difference of maximum opset cost ($opcost_{max}$) and cost

of opset for which we are calculating the delay allowance. It means every opset has the allowance time in which it can wait and let other operations to complete.

4.4.3.2 Delay scheduling

The role of a delay queue is to procrastinate the opset which has some delay allowance. This gives better flexibility for other queries to execute in the delay allowance time.

Delay queue design Delay queue (Q_d) is an unbounded blocking queue implemented in Java for opsets which have delayed allowance. The idea of delay queue is to procrastinate some operations. An element can be taken out once the delay has expired. The element which is at the head of the queue has the expired delay furthest in the past.

Scheduling of requests which has delay allowance If the request is tagged by delay allowance ($T_w > 0$) during delay estimation then the request will be sent to delay queue. The scheduler adds the system current time in the delay allowance i.e. procrastinate time (T_d), which helps to correctly estimate the procrastinated opset.

Scheduling of requests with zero delay allowance If the request is tagged by delay allowance ($T_w == 0$) during delay estimation then the request will be sent directly to the server without delay. Since these are the requests which take time to execute and don't offer any allowance for slacking that opset.

4.4.4 Server selection

Finally, operations are sent to the intended server directly or after completion of the procrastination time.

4.5 Evaluation

We implement TailX as an extension of Cassandra [17], a very popular key-value store. We evaluate its effectiveness in reducing tail latency using synthetic dataset generated using the Yahoo! Cloud Serving Benchmark (YCSB) [32]. We compare different latency percentiles, particularly the tail, under TailX, against state-of-the-art algorithm i.e. Rein. We conduct extensive experiments on Grid'5000 [31], exploring the impact of varying ratios of multiget request sizes and their value sizes. Overall, our evaluation answers the following questions:

1. How is the performance of TailX impacted by the multiget request sizes in the key-value store? (§4.5.2.1)

Algorithm 2: Opset delay allowance algorithm

Data: $ksName$ = keyspace name, K = set of keys, CF = tablename, op = opset, $opcost_{max}$= max opset cost, req = multiget request, $opsets$ = set of opsets, N_L = set of keys correspond to large values in an opset, Q_d = delay queue, BF = bloom filter;

Input: req ($ksName, K, CF$);

Output: Procrastinated opsets.

```
1  begin
2      opcost_max = 0;
3      for op ∈ opsets do
           /* Calculate number of keys correspond to large values in
              an opset                                              */
4          N_L := {opr ∈ op | match(BF, opr.key) = 1};
           /* Calculate opset cost                                  */
           // T_L= request service time (in nanosec) for large value
           // T_S= request service time (in nanosec) for small value
           // opsize = number of keys in an opset
5          opcost = T_L * |N_L| + T_S * (opsize − |N_L|);
           /* Calculate max opset cost                              */
6          opcost_max = max(opcost, opcost_max);
7      for op ∈ opsets do
           /* Calculate delay allowance                             */
8          T_w = opcost_max − op.opcost;
           /* Tag T_w to each opset                                 */
9          tag(T_w, op);
           /* Calculate procrastinating time                        */
           // T_current = current system time
10         T_d ⟵ T_current + T_w;
11         if op.T_w > 0 then
               /* insert opset in delay queue                       */
12             Q_d.enqueue(op, T_d);
13         else
14             send op to corresponding replica;
```

Algorithm 3: Opset dequeue algorithm

```
1  begin
2      while Q_d ≠ ∅ && T_current − T_d ≥ 0 do
3          deque from Q_d;
4          send op to corresponding replica;
```

2. How is the performance of TailX impacted by the proportion of large values in the key-value store? (§4.5.2.2)

We start this section by presenting our evaluation setup (§4.5.1) before presenting our results (§4.5.2.1) and (§4.5.2.2).

4.5.1 Experimental setup

Experimental Platform. We evaluate TailX on Grid'5000 [31]. We use a 16 node cluster in which each machine is equipped with 2 Intel Xeon X5570 CPUs (4 cores per CPU), 24GB of RAM and a 465GB HDD. The machines are running the Debian 8 GNU/Linux operating system.

Configuration. We evaluate TailX in Cassandra. We used the industry standard Yahoo! Cloud Serving Benchmark (YCSB) [32] to generate datasets and run our workloads. As YCSB only generates a single value size datasets for each given client, we modified its source code to allow generation of mixed size datasets. Specifically, for mixed size workloads, we kept the proportion of large values compared to small values the same. For generating client workloads, we configured YCSB on a separate node.

Moreover, in all the generated workloads, the access pattern of stored values (whether small or large) follows a Zipfian distribution (with a Zipfian parameter ρ=0.99). To have an idea of the size a given synthetic dataset, we insert 20 million of small records (1KB size) and 100K of large records (2 MB size). This approximately represents $\tilde{4}$1GB of data per node. We kept the replication factor as 3 which means each piece of value is available on 3 servers. Each measurement involves 1 million or 10 million requests and is repeated 5 times. Each multiget request access various operations with different value sizes. We test the cluster of its maximum attainable throughput and kept the 75% system load for all our experiments.

4.5.2 TailX on variable configurations of the synthetic dataset

We evaluate in this section the effectiveness of TailX along different dimensions of heterogeneous workloads, i.e., the impact of *long* operations on multiget requests (§4.5.2.1), the impact of operations correspond to large values (§3.4.2.2).

4.5.2.1 Impact of multiget requests containing large number of operations

To study the impact of the proportion of long multiget requests (i.e. multiget request size is large) on the system performance, we fix the size of multiget request as 100 and short multiget request to 5. We keep the ratio of long multiget request to 20% i.e. for each 100 multiget requests, 80 multiget are of size 5 and 20 multiget are of size 100.

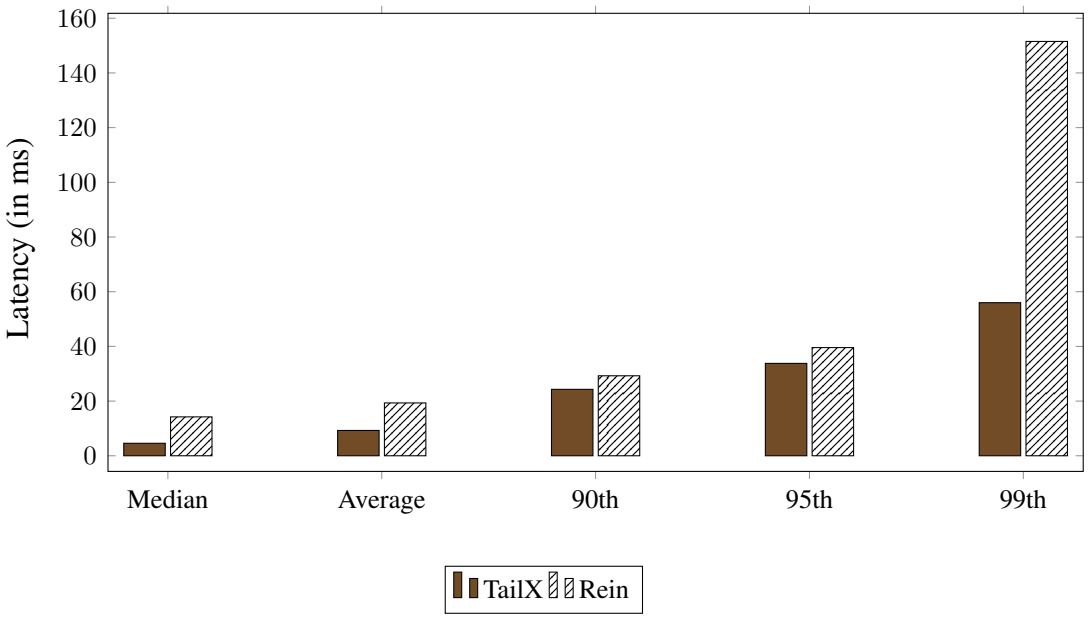


Figure 4.4 – Improvement of TailX over latency with different multiget request sizes (80% multiget of size 5 and 20% multiget of size 100) for 1 million operations

Through this, we can see the impact of long multiget over short multiget requests.

We present the improvement of TailX over Rein for 1 million operations and 10 million operations in Figure 4.4 and 4.5 respectively. Figure 4.6 shows the different latency percentiles to give a closer look in system. In this experiment, we start by generating datasets in which each multiget request contains 1KB values.

Results show that TailX reduces the tail latencies over Rein by up to 63% while reducing the median latency by up to 71%. TailX achieves a better gain for median latency compare to tail latency. In terms of absolute latency (for 1 million operations), say for 99th percentile, it is 56 ms for TailX but roughly 152 ms for Rein respectively. For median latency, absolute value is 4.57 ms for TailX whereas it is around 14.3 ms for Rein.

4.5.2.2 Impact of multiget requests having keys of large value sizes

To study the impact of the proportion of large requests (request having large value i.e. 2 MB) on the system performance, we fix the size of multiget request as 20. We keep the percentage of large multiget requests as 20% and vary the proportion of large values.

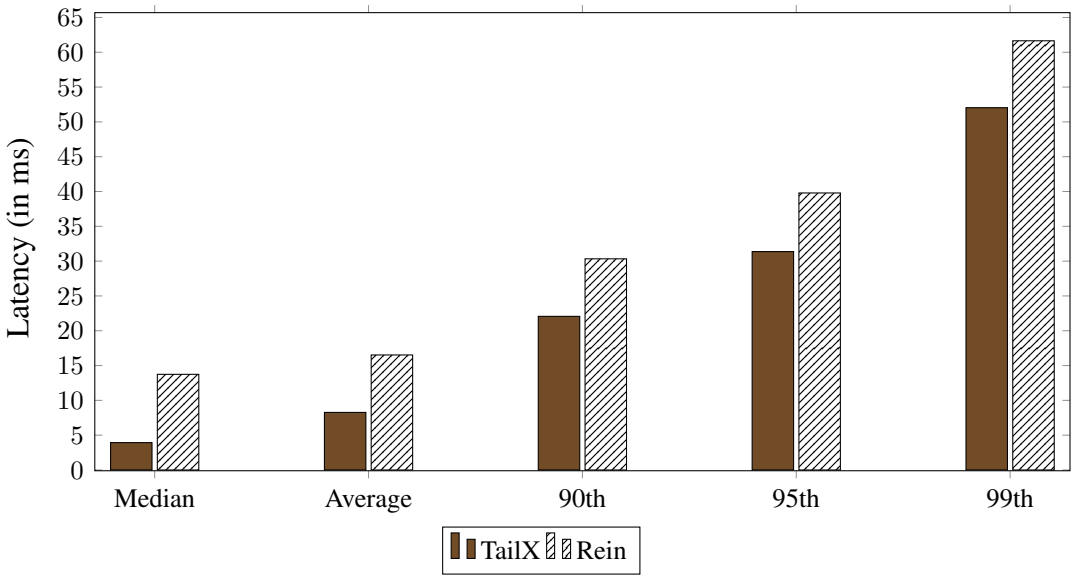


Figure 4.5 – Improvement of TailX over latency with different multiget request sizes (80% multiget of size 5 and 20% multiget of size 100) for 10 million operations

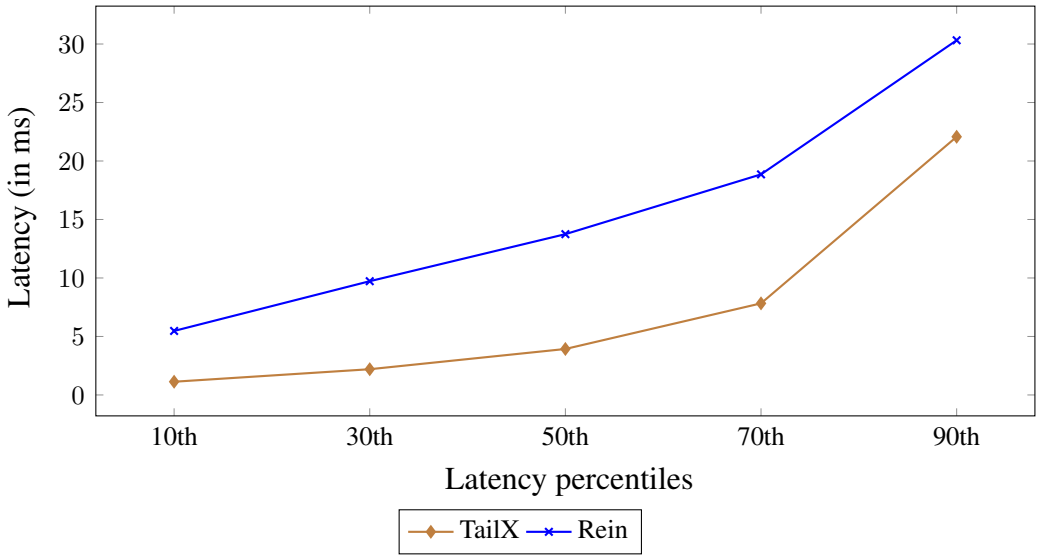


Figure 4.6 – Analysis of different latency percentiles for different multiget request sizes (80% multiget of size 5 and 20% multiget of size 100) for 10 million operations

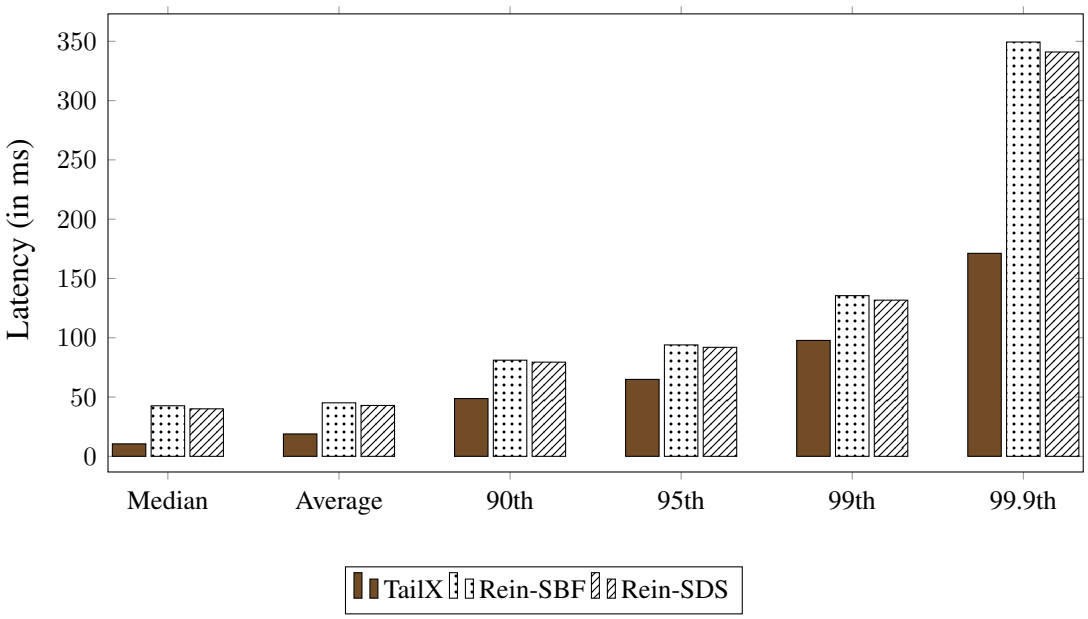


Figure 4.7 – Improvement of TailX over latency with different multiget request value sizes (80% multiget requests have small values (1 KB) and rest 20% multiget requests have 10% of large values) for 1 million operations

Varying proportion of large value sizes. We vary the proportion of large value from 10% to 50% in a multiget request. As specified before, these variations are only for 20% of multiget requests. We present the latency reduction of TailX over Rein for 1 million operations. In the following, we zoom into the specific percentage of large value sizes.

Multiget of 10% large values. In this experiment, 20% of each multiget contains 10% of large values. Figure 4.7 and 4.8 show the improvement of TailX over Rein i.e., 30% latency reduction in 95th and 99th percentiles. TailX achieves a better gain for median latency compare to tail latency, i.e., roughly 75% v.s. 30%. In terms of absolute latency, say for 99th percentile, it is 97 ms for TailX but roughly 135 ms for Rein respectively. For median latency, absolute value is 11 ms for TailX whereas it is around 43 ms for Rein.

Multiget of 20% large values. Figure 4.9 and 4.10 show the improvement of TailX over Rein i.e., 40% and 45% latency reduction in 95th and 99th percentiles respectively. TailX achieves a better gain for median latency compare to tail latency, i.e., roughly 56% v.s. 45%. In terms of absolute latency, say for 99th percentile, it is 112 ms for TailX but roughly 203 ms for Rein respectively. For median latency, absolute value is 8 ms for TailX whereas it is around 18 ms for Rein.

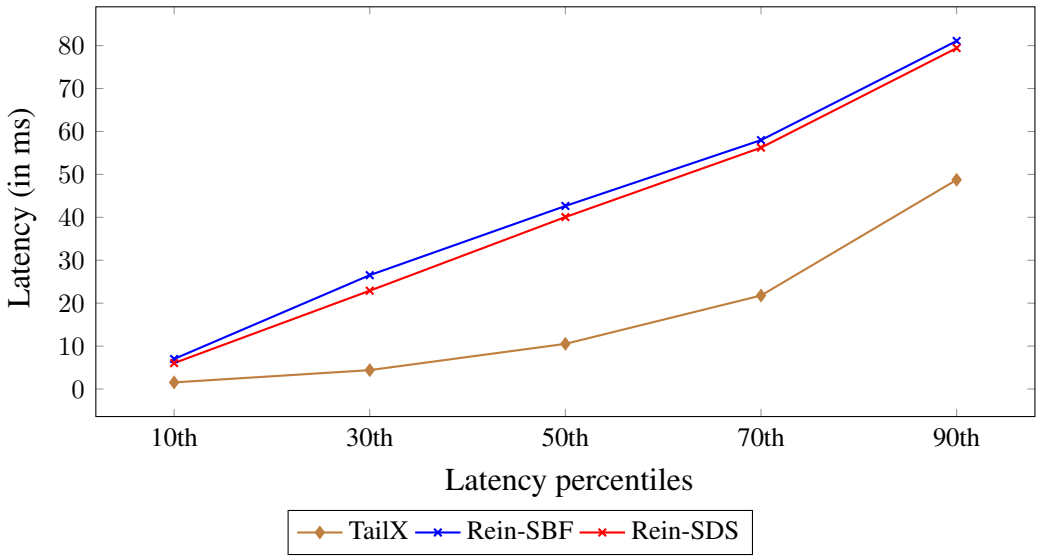


Figure 4.8 – Analysis of different latency percentiles for different multiget request value sizes (80% multiget requests have small values (1 KB) and rest 20% multiget requests have 10% of large values) for 1 million operations

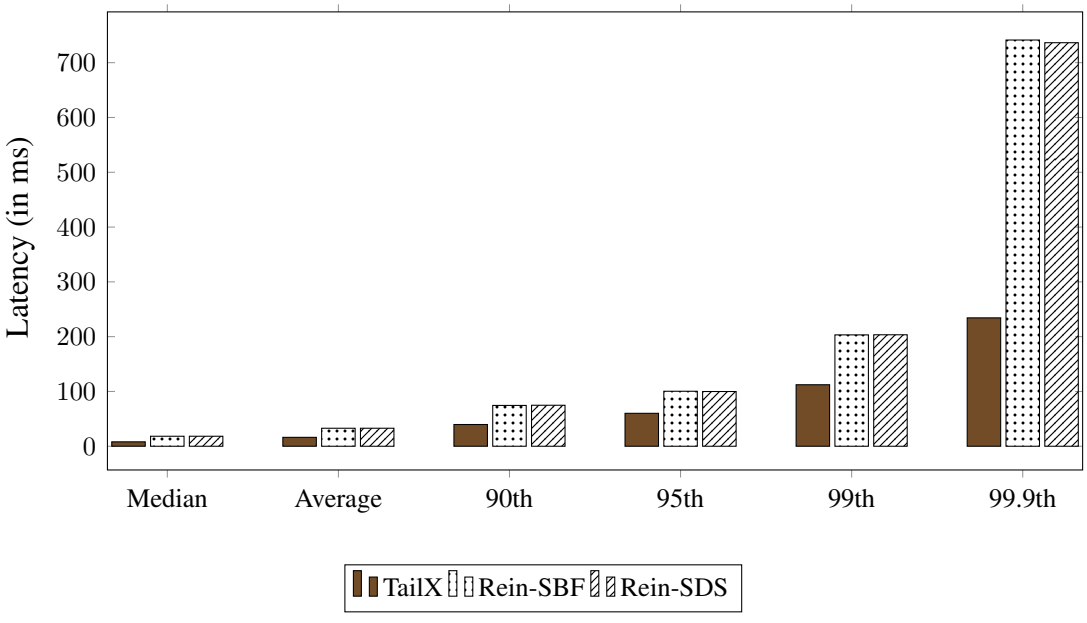


Figure 4.9 – Improvement of TailX over latency with different multiget request value sizes (80% multiget requests have small values (1 KB) and rest 20% multiget requests have 20% of large values) for 1 million operations

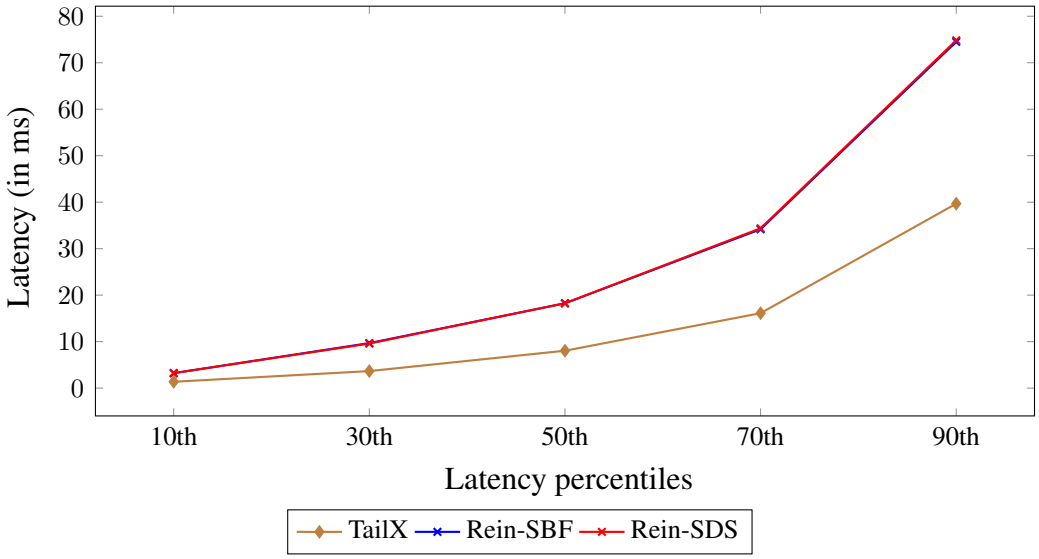


Figure 4.10 – Analysis of different latency percentiles for different multiget request value sizes (80% multiget requests have small values (1 KB) and rest 20% multiget requests have 20% of large values) for 1 million operations

Multiget of 50% large values. Figure 4.11 and 4.12 show the improvement of TailX over Rein i.e., 18% and 27% latency reduction in 95th and 99th percentiles respectively. TailX achieves a little less gain for median latency compare to tail latency, i.e., roughly 13%. In terms of absolute latency, say for 99th percentile, it is 109 ms for TailX but roughly 150 ms for Rein respectively. For median latency, absolute value is 5.9 ms for TailX whereas it is around 6.76 ms for Rein.

Summarizing, TailX outperforms Rein in all the configurations. TailX is effective when there are some long requests in the systems. Also, effectiveness of TailX can be seen when some multiget requests have some percentage of large values. Overall in these configurations, TailX reduces the median latency up to 75% and tail latency by up to 70%.

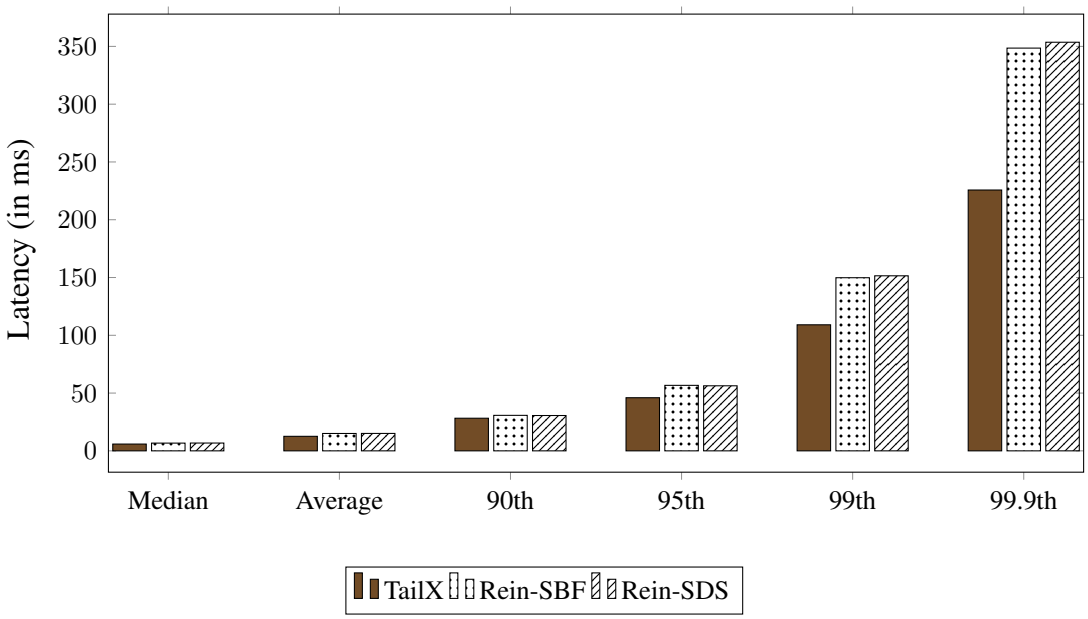


Figure 4.11 – Improvement of TailX over latency with different multiget request value sizes (80% multiget requests have small values (1 KB) and rest 20% multiget requests have 50% of large values) for 1 million operations

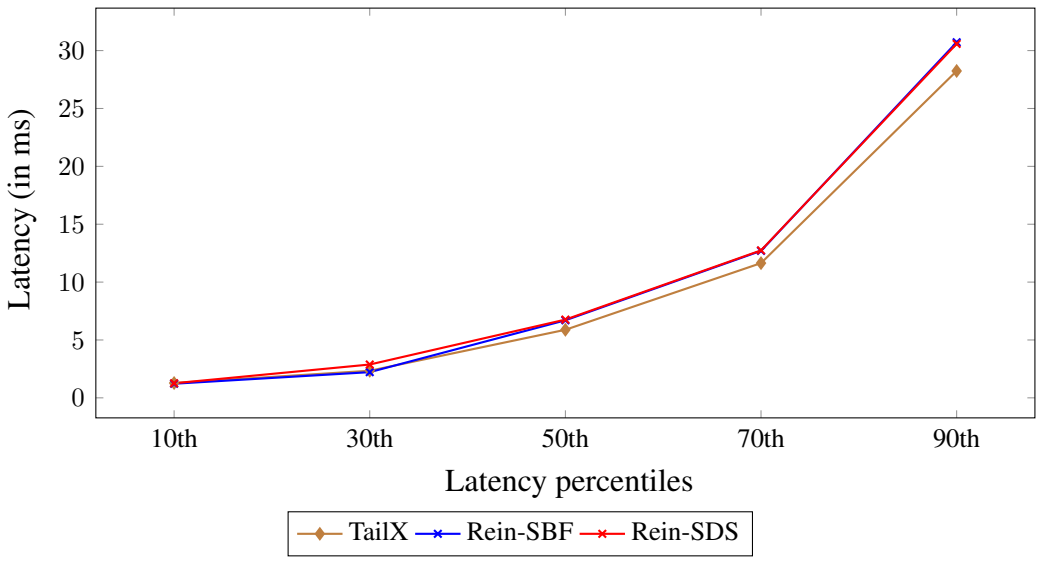


Figure 4.12 – Analysis of different latency percentiles for different multiget request value sizes (80% multiget requests have small values (1 KB) and rest 20% multiget requests have 50% of large values) for 1 million operations

4.6 Summary

In this chapter, we addressed the problem of median as well as tail latency in key-value stores under heterogeneous workloads for multiget requests. We study the approaches that focus on the performance in cloud data stores. For multiget scheduling, an in-depth study of Rein has highlighted the fact that it doesn't perform well under heterogeneous workloads. Specifically, we design TailX, a multiget scheduling algorithm that effectively deals with heterogeneous multiget requests. The result is improved overall performance of the key-value store for a wide variety of heterogeneous workloads. Specifically, our experiments under heterogeneous YCSB workloads in a Cassandra based implementation shows that TailX outperforms Rein and reduces the tail latencies by up to 70% while reducing the median latency by up to 75%.

5

Conclusions

Contents

This thesis started with the aim of improving the performance of cloud data stores. Over the course of this thesis, we addressed the set of challenges faced in the way of improving the performance of cloud data stores and reduces latencies in these systems. In this chapter, we summarize the key aspects of improving the performance in key-value stores by stating how this goal is achieved (§5.1). We share the lessons which we learned during the thesis (§5.2). Finally, we conclude the chapter with some possible future directions in which the work can be extended (§5.3).

5.1 Summary

The main contributions of this thesis are as follows:

1. **Taming tail latencies in key-value stores under heterogeneous workloads:** In chapter 3, we address the problem of tail latency under heterogeneous workloads in key-value stores. Based on the previous work, we make the key observation that replica selection algorithms are useful to reduce tail latency in key-value stores. At the same time, we highlight the challenges involved in designing better replica selection algorithms. We present **Héron**, a replica selection algorithm that reduces tail latencies under heterogeneous workloads. We implement it as part of Cassandra, a widely used key-value store. The result is improved overall

performance of the key-value-store for a wide variety of heterogeneous workloads. Here are the design contributions of Héron:

- Héron includes a replica scoring mechanism similar to the one used by C3 [1] to estimate the load among servers. It periodically collects as scoring metric for each server the product of its average service time and its queue size.
- In an online setting, we don't have apriori knowledge of requests coming on system i.e. we don't know the service demand of the request. Therefore it is very difficult to schedule the request by knowing only its key. Héron predicts which requests will require significant time by keeping track of the keys corresponding to large values.
- Once a request has been identified as accessing a small (respectively, a large) value, it applies an appropriate replica selection algorithm, which avoids head-of-line-blocking.

2. **Task-aware scheduling for improving tail latencies in key-value stores:** In chapter 4, we address the problem of performance bottlenecks in cloud datastores in the case of scheduling multiget requests under heterogeneous workloads. Multiget scheduling is used to achieve low tail latency. We aim to design an efficient scheduling algorithm which can estimate the bottleneck operations and schedule them on uncoordinated backend servers in such a way that creates minimal overhead and reduce latencies at the tail. We present **TailX**, a task aware multiget scheduling algorithm that reduces tail latencies under heterogeneous workloads. We implement it as part of Cassandra, a widely used key-value store. The result is improved overall performance of the key-value-store for a wide variety of heterogeneous workloads. Here are the design contributions of TailX:

   - In an online setting, we don't have apriori knowledge of requests coming on system i.e. we don't know the service demand of the request. Therefore it is very difficult to schedule the request by knowing only its key. TailX predicts which requests will require significant time by keeping track of the keys corresponding to large values.
   - TailX is able to adopt varying workload conditions and can estimate the bottleneck operations.
   - TailX uses a novel approach to schedule the operations on uncoordinated backend servers in such a way that create minimal overhead and reduce latencies at the tail.

5.2 Lessons Learned

Achieving better performance in key-value stores require better algorithms with minimum overhead. It requires solving complex mechanism of distributed architecture and considering the incoming load coming on the system.

In our work, we consider latency as one of the main factors which affects the overall performance of key-value stores. In order to achieve low latency, incoming requests should be scheduled on uncoordinated backend servers in a way that creates minimal overhead. This depends on the value size of the incoming requests. We learn that value size has a great impact on the latency of requests, which leads to head-of-line-blocking if not scheduled properly. Therefore we try to avoid head-of-line-blocking by separating requests with small and large value sizes. Thus, we improved the overall performance and reduces latency at the tail. Apart from value size, it is needed to estimate the load among servers which helps to schedule the requests in an efficient manner.

For task aware scheduling in key-value stores, we learn that it is better to procrastinate some operations of a request which complete very early compared to the operations which take longer time. Therefore it leaves the room for executing the next incoming requests, which leads to improving the latency.

5.3 Future Directions

In spite of contributions of this thesis in regard to performance improvisation in cloud data stores, there are a number of open research challenges that required to be addressed to make further advancement in the area. Some of the research directions are identified as below:

Improving performance under heterogeneous workloads

There is a lot of work remaining for improving performance in key-value stores under heterogeneous workloads. In our thesis, we have worked on single and multi GET requests. Our work in Chapter 3 and Chapter 4 detailed the problem under heterogeneous workloads. Next, we seek to consider the request models where we want to see the performance results with a different ratio of the request for large and small values and different request distribution such as uniform, zipfian bimodal etc. Also, a number of operations in a request greatly impact the latency where we want to see the impact of a heavy request (with more number of keys) on light request (less number of keys). Apart from this, in today's scenario, there are some high priority jobs which should be complete before any other requests. It is interesting to see how these request models impact the performance in key-value stores. To summarize, heterogeneous workloads have a great impact on latency. Therefore a thorough performance evaluation is needed to improve the latency in these systems.

Geo-distributed replica selection

The current work is focused on improving the latency in a single data center. There is a lot of room for improvement in replica selection algorithms for geo-distributed systems. In the literature, a lot of work has been done in the area of geo-distributed replica selection. It is interesting to see the latency measurements for replica selection algorithms under heterogeneous workloads in geo-distributed systems. Some work [41] has been done on selecting the better replica selection algorithm but there is no sufficient literature which shows the improvement in replica selection algorithm in geo-distributed systems.

Scheduling in multi-datacenter with different consistency models

In this thesis, we consider a single datacenter with consistency one. We seek to answer the question where heterogeneous workloads coming on the multi-datacenter. In general, a client is getting replies with different consistency as one or quorum. We believe quorum consistency will have a different impact on replica selection algorithms as well as on task aware scheduling. Since in a quorum consistency model we will have replies from more than half of the replica and it would be interesting to see how these algorithms perform under such consistency levels. Furthermore, such consistency levels with heterogeneous workloads led to complex the scenario. We believe this opens up the research directions under such circumstances.

Bibliography

[1] L. Suresh, M. Canini, S. Schmid, and A. Feldmann, "C3: Cutting tail latency in cloud data stores via adaptive replica selection," in *NSDI*, 2015.

[2] P. Delgado, F. Dinu, A.-M. Kermarrec, and W. Zwaenepoel, "Hawk: Hybrid datacenter scheduling," in *USENIX ATC*, 2015.

[3] E. Schurman and J. Brutlag, "Performance related changes and their user impact," in *Velocity: web performance and operations conference*, 2009.

[4] J. Brutlag, "Speed matters for google web search," *Google. June*, 2009.

[5] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, 2013.

[6] G. Linden, "Make data useful," https://sites.google.com/site/glinden/Home/StanfordDataMining.2006-11-28.ppt.

[7] V. Jalaparti, P. Bodik, S. Kandula, I. Menache, M. Rybalkin, and C. Yan, "Speeding up distributed request-response workflows," in *SIGCOMM*, 2013.

[8] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey, "Bobtail: Avoiding long tails in the cloud," in *NSDI*, 2013.

[9] D. S. Berger, B. Berg, T. Zhu, S. Sen, and M. Harchol-Balter, "Robinhood: Tail latency aware caching – dynamic reallocation from cache-rich to cache-poor," in *OSDI*, 2018.

[10] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: Distributed, low latency scheduling," in *SOSP*, 2013.

[11] T. Zhu, A. Tumanov, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger, "Prioritymeister: Tail latency qos for shared networked storage," in *SoCC*, 2014.

[12] C. Stewart, A. Chakrabarti, and R. Griffith, "Zoolander: Efficiently meeting very strict, low-latency SLOs," in *ICAC*, 2013.

[13] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca, "Jockey: Guaranteed job latency in data parallel clusters," in *EuroSys*, 2012.

[14] M. E. Haque, Y. h. Eom, Y. He, S. Elnikety, R. Bianchini, and K. S. McKinley, "Few-to-many: Incremental parallelism for reducing tail latency in interactive services," in *ASPLOS*, 2015.

[15] M. Jeon, S. Kim, S.-w. Hwang, Y. He, S. Elnikety, A. L. Cox, and S. Rixner, "Predictive parallelization: Taming tail latencies in web search," in *SIGIR*, 2014.

[16] W. Reda, M. Canini, L. Suresh, D. Kostić, and S. Braithwaite, "Rein: Taming tail latency in key-value stores via multiget scheduling," in *EuroSys*, 2017.

[17] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, 2010.

[18] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *SIGMETRICS*, 2012.

[19] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron, "Decentralized task-aware scheduling for data center networks," in *SIGCOMM*, 2014.

[20] B. Williams, "Dynamic snitching in Cassandra: past, present, and future," http://www.datastax.com/dev/blog/dynamic-snitching-in-cassandra-past-present-and-future, 2012.

[21] "CloudComputing Survey," https://www.idg.com/tools-for-marketers/2018-cloud-computing-survey/, 2018.

[22] "Partitioners," https://docs.datastax.com/en/cassandra/3.0/cassandra/architecture/archPartitionerAbout.html.

[23] "Mongodb," https://www.mongodb.com/.

[24] "Openstack swift," https://docs.openstack.org/swift/latest/.

[25] "Apache accumulo," https://accumulo.apache.org/.

[26] "Riak Load Balancing and Proxy Configuration," http://docs.basho.com/riak/1.4.0/cookbooks/Load-Balancing-and-Proxy-Configuration/.

[27] "Wikimedia downloads," http://download.wikimedia.org/.

[28] M. J. Huiskes and M. S. Lew, "The MIR Flickr retrieval evaluation," in *MIR*, 2008.

[29] M. Ould-Khaoua, G. Min, and N. Thomas, "Performance analysis and evaluation of parallel, cluster, and grid computing systems comparing job allocation schemes where service demand is unknown," *Journal of Computer and System Sciences*, 2008.

[30] V. Jaiman, S. B. Mokhtar, V. Quéma, L. Y. Chen, and E. Rivière, "Héron: Taming tail latencies in key-value stores under heterogeneous workloads," in *SRDS*, 2018.

[31] D. Balouek, A. Carpen Amarie, G. Charrier, F. Desprez, E. Jeannot, E. Jeanvoine, A. Lèbre, D. Margery, N. Niclausse, L. Nussbaum, O. Richard, C. Pérez,

F. Quesnel, C. Rohr, and L. Sarzyniec, “Adding virtualization capabilities to the Grid’5000 testbed,” in *Cloud Computing and Services Science*, 2013.

[32] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, “Benchmarking cloud serving systems with YCSB,” in *SoCC*, 2010.

[33] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, “Scaling Memcache at Facebook,” in *NSDI*, 2013.

[34] P. Delgado, D. Didona, F. Dinu, and W. Zwaenepoel, “Job-aware scheduling in Eagle: Divide and stick to your probes,” in *SoCC*, 2016.

[35] J. Lenstra, A. R. Kan, and P. Brucker, “Complexity of machine scheduling problems,” in *Studies in Integer Programming*, 1977.

[36] B. H. Bloom, “Space/time trade-offs in hash coding with allowable errors,” *Communications of the ACM*, 1970.

[37] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, “An improved construction for counting bloom filters,” in *European Symposium on Algorithms*, 2009.

[38] P. Pandey, M. A. Bender, R. Johnson, and R. Patro, “A general-purpose counting filter: Making every bit count,” in *SIGMOD*, 2017.

[39] F. Hao, M. Kodialam, and T. V. Lakshman, “Incremental bloom filters,” in *INFOCOM*, 2008.

[40] S. Ha, I. Rhee, and L. Xu, “Cubic: A new tcp-friendly high-speed tcp variant,” *SIGOPS Oper. Syst. Rev.*, 2008.

[41] K. Bogdanov, M. Peón-Quirós, G. Q. Maguire, Jr., and D. Kostić, “The nearest replica can be farther than you think,” in *SoCC*, 2015.

[42] D. Shue, M. J. Freedman, and A. Shaikh, “Performance isolation and fairness for multi-tenant cloud storage,” in *OSDI*, 2012.

[43] Z. Wu, C. Yu, and H. V. Madhyastha, “Costlo: Cost-effective redundancy for lower latency variance on cloud storage services,” in *NSDI*, 2015.

[44] C. R. Lumb, R. Golding, and G. R. Ganger, “D-SPTF: Decentralized request distribution in brick-based storage systems,” in *ASPLOS*, 2004.

[45] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, “Reining in the outliers in map-reduce clusters using mantri,” in *OSDI*, 2010.

[46] J. Li, N. K. Sharma, D. R. K. Ports, and S. D. Gribble, “Tales of the tail: Hardware, OS, and application-level sources of tail latency,” in *SoCC*, 2014.

[47] A. Gulati, I. Ahmad, and C. A. Waldspurger, “PARDA: Proportional allocation of resources for distributed storage access,” in *FAST*, 2009.

[48] A. Gulati, A. Merchant, and P. J. Varman, "mclock: Handling throughput variability for hypervisor io scheduling," in *OSDI*, 2010.

[49] A. Wang, S. Venkataraman, S. Alspaugh, R. Katz, and I. Stoica, "Cake: Enabling high-level slos on shared storage systems," in *SoCC*, 2012.

[50] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "Mica: A holistic approach to fast in-memory key-value storage," in *NSDI*, 2014.

[51] B. Fan, D. G. Andersen, and M. Kaminsky, "Memc3: Compact and concurrent memcache with dumber caching and smarter hashing," in *NSDI*, 2013.

[52] G. Ananthanarayanan, A. Ghodsi, A. Warfield, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica, "Pacman: Coordinated memory caching for parallel jobs," in *NSDI*, 2012.

[53] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient coflow scheduling with varys," in *SIGCOMM*, 2014.

[54] R. Motwani, S. Phillips, and E. Torng, "Non-clairvoyant scheduling," in *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA, 1993.

[55] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing data transfers in computer clusters with orchestra," in *SIGCOMM*, 2011.

[56] M. Chowdhury and I. Stoica, "Efficient coflow scheduling without prior knowledge," in *SIGCOMM*, 2015.

[57] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "pfabric: Minimal near-optimal datacenter transport," in *SIGCOMM*, 2013.

[58] A. Vulimiri, P. B. Godfrey, R. Mittal, J. Sherry, S. Ratnasamy, and S. Shenker, "Low latency via redundancy," in *CoNEXT*, 2013.

[59] M. Mitzenmacher, "The power of two choices in randomized load balancing," *IEEE Transactions on Parallel and Distributed Systems*, 2001.

[60] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: Flexible, scalable schedulers for large compute clusters," in *EuroSys*, 2013.

[61] M. Jeon, Y. He, H. Kim, S. Elnikety, S. Rixner, and A. L. Cox, "TPC: Target-driven parallelism combining prediction and correction to reduce tail latency in interactive services," in *ASPLOS*, 2016.

[62] M. Harchol-Balter, B. Schroeder, N. Bansal, and M. Agrawal, "Size-based scheduling to improve web performance," *ACM TOCS*, 2003.