Fall 12-2018

Reducing the Tail Latency of a Distributed NoSQL Database

Jun Wu

REDUCING THE TAIL LATENCY OF A DISTRIBUTED NOSQL DATABASE

by

Jun Wu

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfillment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Professor Lisong Xu

Lincoln, Nebraska

Dec, 2018

REDUCING THE TAIL LATENCY OF A DISTRIBUTED NOSQL DATABASE

Jun Wu, M.S.

University of Nebraska, 2018

Adviser: Dr. Lisong Xu

The request latency is an important performance metric of a distributed database, such as the popular Apache Cassandra, because of its direct impact on the user experience. Specifically, the latency of a read or write request is defined as the total time interval from the instant when a user makes the request to the instant when the user receives the request, and it involves not only the actual read or write time at a specific database node, but also various types of latency introduced by the distributed mechanism of the database. Most of the current work focuses only on reducing the average request latency, but not on reducing the tail request latency that has a significant and severe impact on some of database users. In this thesis, we investigate the important factors on the tail request latency of Apache Cassandra, then propose two novel methods to greatly reduce the tail request latency. First, we find that the background activities may considerably increase the local latency of a replica and then the overall request latency of the whole database, and thus we propose a novel method to select the optimal replica by considering the impact of background activities. Second, we find that the asynchronous read and write architecture handles the local and remote requests in the same way, which is simple to implement but at a cost of possibly longer latency, and thus we propose a synchronous method to handle local and remote request differently to greatly reduce the latency. Finally, our experiments on Amazon EC2 public cloud platform demonstrate that our proposed methods can greatly reduce the tail latency of read and write requests of Apache Cassandra.

DEDICATION

This thesis is dedicated to my family, for their continuous and warm encouragement and support through my way to the degree.

ACKNOWLEDGMENTS

I would first like to thank my thesis advisor Dr. Lisong Xu of Department of Computer Science & Engineering at University of Nebraska Lincoln. His excellent forethought and guidance inspire me a lot. He is always there if I encountered any problem. Without him, it could not be possible for me to complete this thesis. No words could describe how grateful I am for having him as my advisor.

Meanwhile, I would also like to thank my committee members, Dr. Hongfeng Yu and Dr. Qiben Yan for serving as my committee members. Greatly thanks them for generously offering their time, support, and guidance throughout the preparation and review of my thesis.

Lastly, I would like to say thanks to my family and my girlfriend. They offer unconditional support and provide solid backing. When I was down, they'll always be there and cheer me up. Without support from them, I could not be who am I today. Lastly, I would like to thank all my colleagues and my friends. We talk, we play together, we help each other, we share tears and laughter. My life could not be so wonderful without them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Relational database management systems (RDBMS) [20] have been traditionally used for organizing data. A RDBMS allows users to create, query, update, and delete databases. Hence, users can manage databases with the four corresponding basic operations: create, read, update and delete (known as CRUD) [38]. However, in the new ear of the Internet, challenges arise for the RDBMS with the need to manage very large data sets, which we call big data [44]. Because RDBMS were originally designed to work on a single machine as a server, the performance comes to a bottleneck to handle large data sets [26]. Hence, NoSQL (Not-only-SQL, also called non SQL) databases [45] are designed to solve this problem, while providing great performance.

As of today, NoSQL databases have been widely used for storing large data. These databases include Apache Cassandra [1], Apache HBase [3], MongoDB [11] etc. Compared with traditional RDBMS, like MySQL [12], Oracle [13], NoSQL systems are more powerful. Firstly, NoSQL systems are normally not centralized and distributed applications [33]. They are designed in a masterless fashion and there are no master-slave concepts proposed

in RDBMS. This property makes NoSQL systems easier to scale up horizontally [54]. Basically, we only need to add more machines/nodes to the systems and they can spread data automatically across multiple nodes. This could linearly improve the system capacity. Secondly, compared with RDBMS, the NoSQL systems support not only the structured data, but also the semi-structured and unstructured data. Hence, the NoSQL systems can support various types of data, like video, email, texts etc, while RDBMS can only handle clearly defined data [57]. Meanwhile, for RDBMS, the schema should be predefined before any operation, while NoSQL system can deal with dynamic schema. This provides more convenience to our current daily usage. Finally, for RDBMS, they are prone to server failures, because if the master node fails, the whole system will fail. However, NoSQL systems spread data among multiple nodes, even if one or more nodes go down, they still could provide continuous availability to the users [58].

Among these NoSQL systems, Apache Cassandra is one of the most representative ones. It's free and open-sourced [62]. Currently, it is in use at eBay, Instagram, Netflix and many other companies [2]. In Cassandra, there is no master-slave concept and every node in the cluster is identical. Normally, data is stored in multiple nodes, working as replicas for fault-tolerance. Meanwhile, replication [59] across multiple data centers is also supported to geographically distribute data. Even an entire data center goes down, Cassandra is still able to provide service to users. So there is no single point of failure for the whole system [33]. Also, its elastic characteristic makes it easy to scale the system, by adding more new nodes. This could linearly improve the read and write throughput, without interruption to the existing applications [23].

The performance of Cassandra has a great impact on the user experience, especially the performance of the basic operations, such as write, read and delete. Among the performance metrics, the response time [32] is the most intuitive one for the users. Formally, it's defined as the request latency or end-to-end latency, which is normally

the time duration from the time when an user sends out a request to the time when the user gets the operation results. Google [29] suggests that the request latency should be no more than 100 ms to provide satisfactory user experience. Otherwise, the degraded performance could directly impact revenues for the companies [22].

Therefore, in this thesis, we focus on analyzing the request latency problem [39] in Cassandra.

1.2 Problem studied in this thesis

There are two types of request latency. One is the average latency or the mean latency, and the other is the percentile latency, such as the 99th percentile latency or even 99.9th percentile latency [65]. A recent measurement study [36] shows that the 95th (99th) percentile latency of over 30% of examined services could be three times or even five times of the median latency. Another measurement results of a Google service [29] show that the 99th percentile latency is 10 ms for a single random request.

However, the 99th percentile latency for all the requests to finish is 140 ms, while for 95% of the requests, that latency is only 70 ms. This means for the latency, 95% of the requests are doubled, due to waiting for the remaining 5% of the requests. These results clearly show the importance of the percentile latency, which normally is called the tail latency [69]. Thus, in this thesis, we study how to reduce the tail latency. However, this could be quite challenging because of various reasons.

Firstly of all, Cassandra is a multi-tiered, distributed system, where a read request experiences more complex processes/stages than a write request. For example, a read request may experience the READ stage, READ RESPONSE stage, READ REPAIR stage in the whole read process [61]. Generally, reading data from the cluster may involve multiple nodes. For example, it may have connections among coordinator node, non-coordinator

node and the client machine. Nearly every node could potentially impact the tail latency performance.

Meanwhile, on each server, multiple applications are competing for the limited system resources, like CPU cores, memory and disk [42]. Even for a single application, these resources are shared among multiple stages. At the same time, as a distributed system, the nodes in Cassandra also need to compete for the network resources, such as the shared file system and the bandwidth of the network. This also adds unpredictable factors to the system [63].

Furthermore, for each server, background activities happen irregularly, which may also have unpredictable influence on the performance of the servers. Internally, this includes periodic garbage collection [43] activities in garbage-collective programming languages, such as Java. In the full Garbage Collection, applications may experience stop-the-world phenomenon. This could stop any other applications which are running on the Java Virtual Machine (JVM) platform [56]. Only after the completion of the full Garbage Collection, other processes could resume to normal status. Also, background data compaction could also happen during user's requests. Data compaction is utilized to compact multiple SSTables to single one, which could incur a large number of disk I/O operations [40].

Finally, the hardware power of each server is limited [29]. Such limitations make it impossible to provide consistent high performance for Cassandra.

In summary, the complex design of Cassandra makes latency appear in multiple stages. The unpredictable background activities in each node make it hard to select the best replica. Meanwhile, limited hardware and network resources lead to the bottleneck for the system performance. All these factors contribute to the high latency in Cassandra cluster, especially the tail latency. The ultimate goal of this thesis is to study the impact of these factors on the tail latency and propose techniques to reduce the tail latency of

the whole cluster.

1.3 Contributions

This thesis makes the following contributions.

Firstly, we conduct a comprehensive measurement study to understand the reasons for the long tail latency. Specifically, we measure the queue size in the READ stage, the service time for each read request, the corresponding response time back to the coordinator node, and the waiting time in the queue that plays a very important part in the latency when the dataset size is large. For the queue size and service time, they're measured both locally and remotely at their coordinator nodes. For the local read in coordinator nodes, these metrics are measured locally, while for the remote read for non-coordinator nodes, these metrics are measured in the non-coordinator nodes and piggy-backed to the coordinator node through the response packet.

Second, we find that the selection of the replica to read data has a big impact on the tail latency, because a better replica could reduce not only the local latency at the replica, but also the overall latency of the whole database, because of the better load balance among the nodes in the database [49]. Therefore, we propose a novel replica selection algorithm to select the best replica to read data from. This new algorithm considers the current status of each node and does not select the replica with busy background activities, such as Garbage Collection and data compaction. Chapter 4 provides more details for this algorithm.

Lastly, we modified the process for the case when local read and remote read exist, which helps reduce latency. In Cassandra, it uses Staged Event Driven Architecture (SEDA) [67], operations like read or write are put into multiple stages. Specifically for the read request, it includes the READ stage, RequestResponse stage, and ReadRepair

stage. These stages are modeled as a thread pool and the requests are executed by threads concurrently. Because of the multi-threading mechanism, Cassandra provides very powerful service in case of a large number of requests. For a read request, it could be served locally by its coordinator node or remotely by a non-coordinator node. For the local service, it's still using the READ stage to execute the local read asynchronously. In this case, when there are multiple requests waiting for available threads, this could lead to a large amount of waiting time for the request. Meanwhile, the switch time of different stages could also add unnecessary latency. Based on this observation, we can reduce this part of waiting and queuing latency, by modifying the local read process, and letting the local read to be executed synchronously in the request thread.

Chapter 2

Background

In this chapter, the basic but important concepts in Cassandra are introduced firstly. Then, we present the flow for both the write path and read path. Finally, we introduce the architecture that Cassandra is built on.

2.1 Key concepts and architecture in Cassandra

Apache Cassandra was originally designed at Facebook. Afterwards, it combines the features from Amazon's DynamoDB [30] and Google's BigTable data model [25]. Since then, it becomes a top-tier project. Essentially, Cassandra is a key-value based database [53], which means that it has a primary key connecting with a set of data. This key is used for data retrieval. According to the famous CAP (Consistency, Availability, Partition tolerance) theorem proposed by Eric Brewer [5], Cassandra typically falls into the AP system. That means Cassandra makes sure that every operation receives a response, no matter whether it's successful or not. Meanwhile, it can continue working, even if part of the system goes down.

However, that does not mean that Consistency is not important in Cassandra. In

contrast, the Consistency level could be tuned by a replication factor and consistency level in Cassandra [18].

In Cassandra, nodes are the basic components to store data. When a client connects to one of the nodes to operate a read or write request, that node is served as the coordinator for the operation. A cluster is a group of nodes, which could be a single node, a single data center or multiple data centers [51].

To share each node's information, Cassandra uses the peer-to-peer communication protocol called Gossip [19]. To assure no single point of failure [46], data is distributed among multiple nodes in the cluster. These nodes are called replica. A replication factor could be set up with various needs. For example, a replication factor of 1 means there is only one copy of each row in one node. A replication factor of 3 means three copies of each row and each copy is in a different node. We can set up the replica replacement strategy to be either SimpleStrategy or NetworkTopologysTrategy [52]. The former is used for single data center, while the latter is used for multiple data centers. A consistency level could also be configured with multiple choices, like ONE, TWO, THREE, etc [66]. It should be less than the number of nodes. Further, it could be set up to ANY, QUORUM. The default consistency level for both read and write requests is 1. For a write operation, that means a write must be written to the CommitLog and MemTable of at least one replica. For a read request, it means the client should get one response from the closest replica.

2.2 Write path in Cassandra

Figure 2.1 shows the operation for the write request in a single data center cluster. The cluster consists of 6 nodes and the replication factor is set to 3. When a client wants to write data to the cluster, it contacts node 1 firstly, which serves as Coordinator node.

Then, three copies of the data are written to nodes 3, 4 and 5. If the consistency level is set to the default value of 1, then only one node will send a write-success response back to the Coordinator. Finally, the Coordinator sends back a write acknowledge to the client, telling it the whole write request is done.

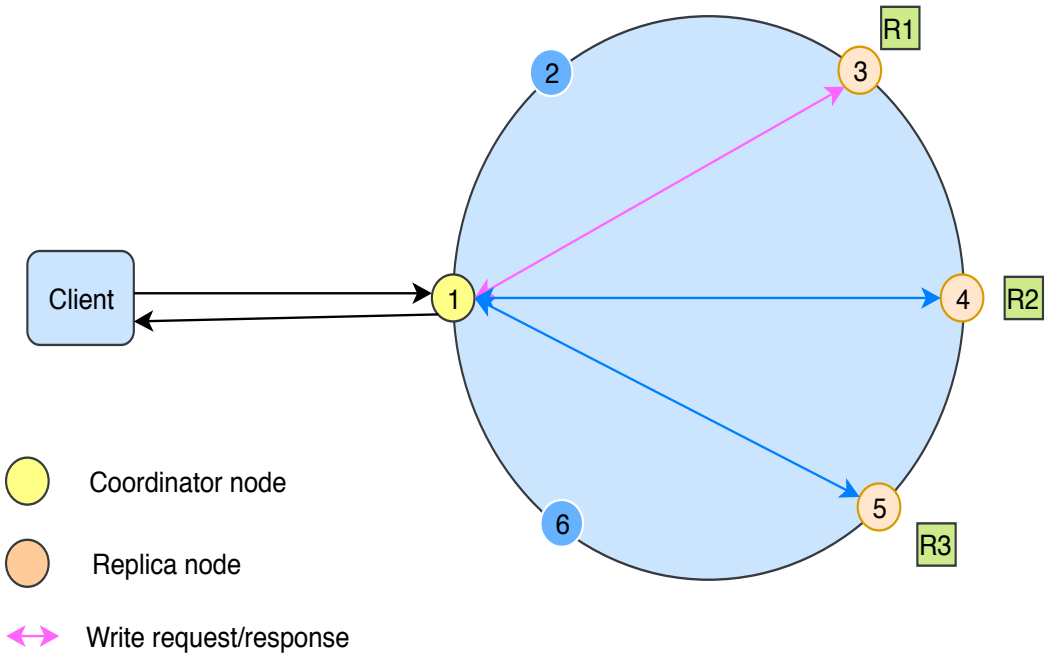

Figure 2.1: Write request in a single data center cluster (adapted from [16])

For a write path in Cassandra, the write request goes through multiple stages. Figure 2.2 shows the full stage. For each column family, it may go to three different layers for storage: MemTable, CommitLog and SSTable (Sorted String Table). MemTable is a special structure in memory. It works similarly to a write-back cache of data, which can be looked up by the primary key. Commit log is a log structure in disk to store data. When new data needs to be written, it will be appended to Commit log, which is for durability. Even if the data in MemTable gets lost, it can still be recovered from the Commit log. Because the size of memory is limited, when the size of the MemTable reaches to its limit, data will be flushed to SSTable. Data in Commit log will be purged after the corresponding

data in MemTable is flushed to an SSTable. SSTable is immutable, which means after writing, it can't be changed. For each SSTable, it maintains a partition index, a partition summary and a Bloom filter. All these structures are used for quicker read. A partition index is a list of partition keys and positions of data on disk. Partition summary is a sample of partition index [10].

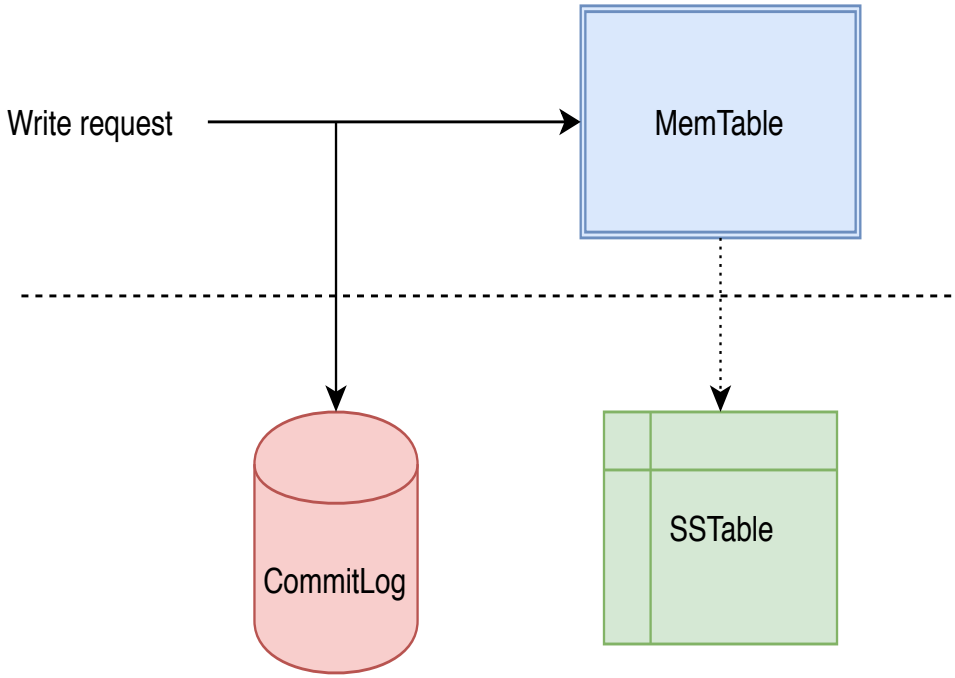

Figure 2.2: Internal write path (adapted from [10])

Meanwhile, as SSTables are immutable, the size of SSTables increases drastically. To save the space and assure updated data, Cassandra provides a mechanism called compaction to compact the SSTable periodically. Essentially, Cassandra compares the timestamps of the data and marks the outdated data that will be deleted later. Then updated data are accumulated into a new SSTalble during compaction.

2.3 Read path in Cassandra

The read path is more complex than the write path. In a single datacenter cluster illustrated in Figure 2.3, the replication factor is 3 and consistency level is one. The closest replica is selected for full reading. Meanwhile, a read repair may happen in the background, based on the read_repair_chance parameter.

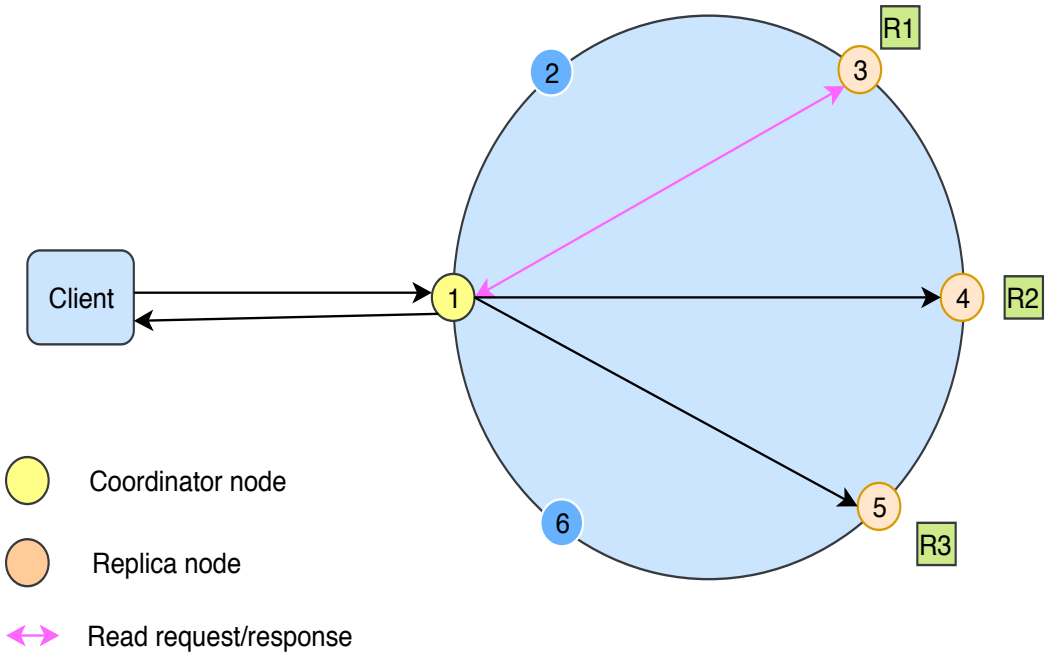

Figure 2.3: Read request in a single data center cluster (adapted from [16])

Figure 2.4 shows the internal read process. When a read request comes in, a node checks the Bloom filter, which is a space-efficient probabilistic data structure designed to check whether an element is not inside a data set [4]. When the Bloom filter shows that data is not in the corresponding SSTable, it skips the scan for this SSTable. Then Cassandra checks the partition key cache and if the entry is found in the cache, it goes to the compression offset map and finds the position that has the data. If it's not found in the cache, Cassandra goes to the partition summary to check the approximate location of

the entry. If the entry is still not found, then it goes to the partition index for full scan and finds the position for the data [9]. After finding the position of the data, then it fetches the data from the disk and returns the result back to the client.

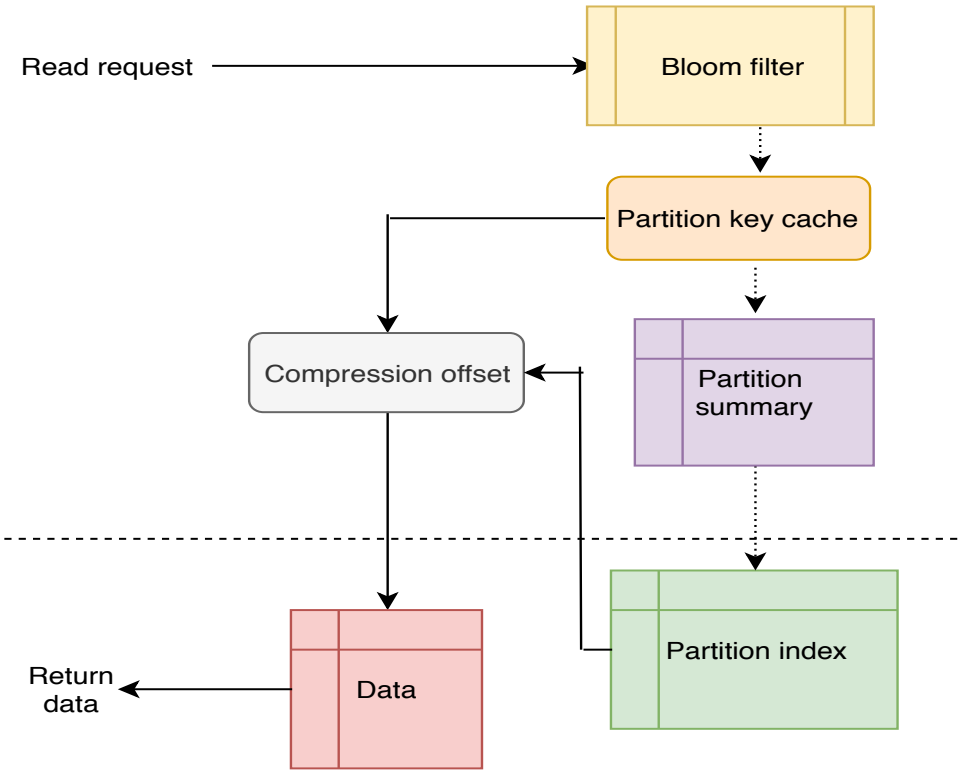

Figure 2.4: Internal read path (adapted from [9])

2.4 Staged Event Driven Architecture (SEDA)

Cassandra is based on a Staged Event Driven Architecture (SEDA) [67]. This architecture is designed for highly concurrent Internet services and is intended to support massive concurrency demands. In SEDA, applications are composed by multiple event-driven stages, which are collected by message queues. This design provides high performance and is robust to dynamic load.

In Cassandra, different events correspond to different stages. For example, for a write request, it goes to Mutation Stage, while for a read request, Read Stage is allocated to handle it. Meanwhile, multiple other stages are included, such as the Request_Response Stage, Internal Stage and Gossip Stage, etc [15]. Each stage has a thread pool and an event queue. The thread pool is to execute the current requests while the incoming request are queued in the event queue. When one of the threads is available, a request is removed from the event queue and moved to the available thread to be executed [14].

To demonstrate the process, take a read request as an example. For instance, node A wants to send out a read request to node B. The whole process may go through the process illustrated in Figure 2.5.

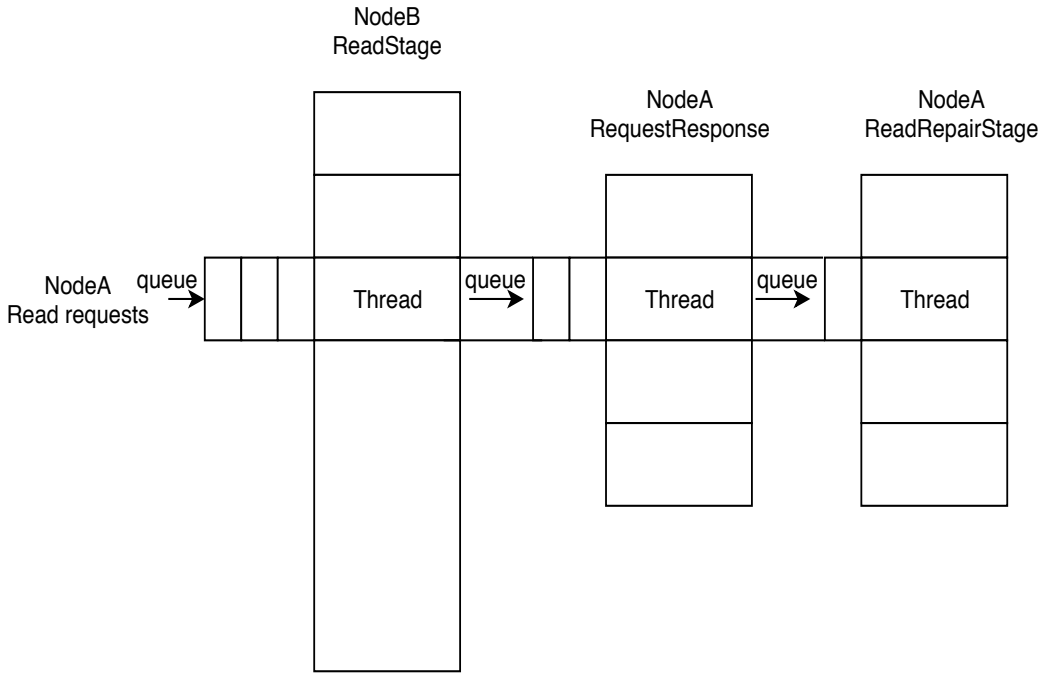

Figure 2.5: A read request stage flow (adapted from [15])

Basically, these are the working flow: (1) Node A sends out a read request to node B. The request is queued in the event queue of node B. (2). If one of the threads in the ReadStage of node B is available, the request from node A is dequeued from the event

queue and executed. (3). When the request has been executed, the result rows are fetched and sent out to the Request Response Stage of node A. (4). Similarly, the thread pool of node A executes the request response from node B and returns the result to the client. (5). Meanwhile, the read repair operation is optional to be processed in node A. Note that in the figure above, the thread pool size for the Read Stage is set to 32, while the size for Request Response Stage and the Read Repair Stage is 4. Actually, these are the default setting for Cassandra and are configurable according to different system requirements [9].

Chapter 3

Related Work

3.1 Research related to tail latency reduction

There has been years of research on the tail latency reduction, especially in the cloud platform. Jeffrey Dean and Luiz Andre Barroso in Google introduced multiple techniques to reduce the latency [29]. They mentioned that there are multiple reasons for latency. High tail latency in individual components of a service could be caused by global resource sharing, maintenance activities, queueing and background activities. They provided high-level solutions based on both the short-term and long-term adaptions. Real time status of the servers are monitored. Then the client avoids the slowest server.

[36] is another paper to analyze the tail latency and provide solutions to the latency reduction. This paper analyzed the latency distribution in each stage of the web service of Bing. It found that the reasons for high latency includes slow servers, network anomalies, complex queries, congestion due to improper load balance, and software artifacts like buffering. Three techniques are developed to resolve the latency issue: reissues, incompleteness, and catch-up. The adaptive reissue is to get the time information from the past queries, and then start a second copy of the request at the optimized time

calculated using the statistics collected. The incompleteness method is to trade off completeness with latency. The catch-up technique allocates more threads to process the slow requests and uses high priority network packets for lagging requests to protect them from burst losses.

A. Vulimiri *et al.* in [64] argued that with the use of redundancy, latency especially the tail latency could be reduced. The paper proposed a very powerful technique through redundancy technique: initiate an operation multiple times, and use the first result which completes. It also introduced a queuing model of the query replication, and analyzed the expected response time as a function of the system utilization and server-side service time distribution. This technique showed great performance improvement on both the mean and tail latency reduction in several systems, such as DNS queries, database servers.

[68] focused on another aspect of tail latency reduction in the cloud platform. It verified with experiments, showing that the poor response times in Amazon EC2 are a property of nodes. The root cause for the long response time on each node is the co-scheduling of CPU-bound and latency-sensitive tasks. Then it implemented a framework called Bobtail to detect the instances of which the sharing processor does not cause extra long tail latency. Then users could use Bobtail to decide on which instance to run their latency-sensitive tasks. The final results showed that with the usage of Bobtail, the 99.9th percentile response time could be reduced by 40 %.

[70] focused on scheduling requests to meet the service level objectives (SLO) for tail latency. It automatically configured workload priorities and rate limits among shared network stages. Through measurements, high priorities were used to provide less latency to the workloads which required low latency. If a workload can meet it's SLO with a given low priority, then this lowest priority is removed from the search. Otherwise, it would iterate on the remaining workloads at the next lowest priority. Through this method, it assured each task is finished within the SLO constraint. Meanwhile, to prevent starvation,

multiple rate limiters were utilized for each workload at each stage.

Sh. Khan and ASML Hoque in [37] indicated that in data center applications, predictability in service time and controlled latency, especially tail latency, were quite important to build high performance applications. It analyzed three data center applications: Memcached, OpenFlow and Web search, to measure the effect of 1) kernel socket handling, NIC interaction, and the network stack, 2) application locks in kernel, and 3) application queuing due to requests queued in the thread pool. It proposed a framework called Chronos to deliver low-latency service in data center applications. Chronos utilized user-level networking APIs to reduce lock contention and perform efficient load balancing. The experiments results showed that, the tail latency of the new Memcached with the Chronos could be reduced by a factor of 20, compared with existing Memcached.

J. Li *et al.* in [41] explored the hardware, OS, and application-level sources of high tail latency in high throughput servers in multi-core machines. It modeled these network services as a queuing system, to establish the best latency distribution. Using fine-grained measurements, it showed that the underlying causes for the high latency include interference from background tasks, request re-submit due to poor scheduling, sub-optimal interrupt routing, etc. To fix those problems, it implemented several mechanisms. For example, it either used real-time scheduling priorities or isolated threads on dedicated cores, to isolate the server from background tasks. The final experiments showed that it reduced the 99.9th percentile latency of Memcached from 5ms to 32us at 75% utilization.

M. Haque *et al.* in [34] proposed a mechanism called Few-to-Many(FM) incremental parallelization, which dynamically increases parallelism to reduce tail latency. It used request service demand profiles and hardware parallelism in an offline phase to compute a policy. This policy is to specify when and how much software parallelism to add. Then during running time, FM adds parallelism. If a request executes for a long time, then FM adds more parallelism. The result showed that FM reduces the 99th percentile response

time up to 26% in Bing and 32% in Lucene.

3.2 Performance research related to Cassandra

Because Cassandra has been popularly used only in recent years, there is very little research on the performance research.

Paper [24] described the architecture of a QoS infrastructure for achieving controlled application performance over Cassandra. It tried to fix the storage configuration problem. Meanwhile, it could also address the dynamic adaption problem, by monitoring service performance at run time and adjusting the short term variations. The core of the architecture is the QoS controller. This controller collected response time and throughput metrics periodically. Then it performed admission control, by estimating overall resource and level of sanctification of requirements.

In Cassandra, mixed query types existed, such as a single query to obtain just a value. Then in this case, a single query needed to wait for the completion of the range query, which took longer time than a single query. This caused the increase of theresponse time. Then S. Fukuda *et al.* in [31] proposed a query scheduling algorithm. It gave the higher priority to a single query and made the query be executed before a range query. For range queries, this algorithm also gave higher priority to queries which require fewer search results. The priority assignments were dynamically changed with the progress of query execution.

In [50], Sh. Nakamura and K.Shudo developed a modular cloud storage called MyCassandra, based on Cassandra. To optimize the read the write performance, it built a storage engine interface, based on different types of storage engines, such as MySQL, Redis. When a query came in, the interface guided the query to different storage engines, based on whether the query is read-optimized or write-optimized. Through this

re-direction, the read and write latency could be reduced by leading the query to the nodes which process faster. The final results showed that MyCassandra could reduce the read latency by 90% and improve the throughput up to 11 times in specific workload.

P. Suresh *et al.* in [60] aimed to cut the tail latency in Cassandra through adaptive replica selection. They proposed a new algorithm called C3. It focused more on the read path and improved the read performance. Two mechanisms were proposed: replica ranking and distributed rate control. For the replica ranking, it modified the existing dynamic snitch algorithm, which only considered the history latency to decide the best replica to read data from. Additionally, C3 measured the queue size, the response time and the service time for each server. Then it piggy-backed these information to the client, to let the client selects the best replica. Meanwhile, it measured the request served in a fixed time interval, then sent it back to the client. Then the client dynamically adjusted the request sending rate. The final result showed improvement on mean, median and tail latency. Specially, for the 99.9% latency, there is a 3 times improvement.

In [35], the authors formalized the write process model of Cassandra and explored two queuing systems: the sending task queue and mutation queue. The sending queue was defined as the queue in the coordinator node and the mutation queue was the queue to wait for mutation in both the coordinator and non-coordinator node. This paper modeled the sending queue as a G/G/1 system and mutation queue as a G/G/S system. Then it measured the inconsistency between two replicas via the departure time difference from the mutation queue. Furthermore, metrics like Request Per Second(RPS) were measured. Then through the measurement of the RPS, the Mutation Threads Number(MTN) was adjusted dynamically. Meanwhile, it also proposed a metric called inherent inconsistency time window(IITW), to measure the replica consistency.

W. Chow and N. Tan in [27] investigated the usage of redundant requests to reduce the read latency in Cassandra. It tested the system via a dynamical threshold when

sending duplicate requests. It also tried to change sending duplicate replies with a small portion. The results showed that these mechanisms performed similar to the average case and performed better in the long tails. Meanwhile, in order to dynamically determine the optimal retry threshold, the paper applied a graphical model to predict the network latency in the system. By exploring the correlation between nodes and time, the model was more accurate from the results.

J. Polo *et al.* in [55] explored the effects which additional consistency guarantees and isolation capabilities may have on the store state of Cassandra. It proposed a new multi-versioned isolation level which provided stronger guarantees. This isolation was implemented in the form of readable snapshots, which included changes in both the data store and the read path of Cassandra. Meanwhile, it implemented a new compaction strategy which compacted the SSTables only within certain boundaries. With this new compaction strategy, only specific SSTables were considered for compaction. The final results showed that it not only improved the throughput, but also the average read latency.

Chapter 4

Problems and Methods

4.1 Problem statements

In Apache Cassandra, data is stored among multiple replicas. To read data from the cluster, the client needs to contact one of the nodes, which works as the coordinator node. The coordinator node is the proxy between the client and the whole cluster. Through the coordinator, the client gets the information of the whole cluster. The client sends a request to the coordinator node and then the coordinator node gets the result or the acknowledge packet back to the client.

During this process, the coordinator node needs to select one or more replicas out of multiple replicas to serve a request. This is based on the consistency level (CL), determined by the users. The CL shows how up-to-date and synchronized of the data on all of the replicas and it could be any number between one and the total number of nodes. Cassandra extends the concept of eventual consistency by providing the tunable consistency level. For example, if we set the CL to be one, then one out of the replicas needs to send an acknowledge packet back to the coordinator node. If it's two, then two out of the replicas need to reply back to the coordinator node. If the CL is large, then the

users could get more reliable data.

However, this could lead to long transmission latency, as the users need to wait for the acknowledge packets from a large number of replicas. On the contrary, if the CL is small, the performance for the latency could be guaranteed. Then the reliability of the data may vary. So the users should trade-off the settings with their own realistic requirement.

In this thesis, we consider the situation where CL is set to be one, which means the coordinator node needs to select one out of multiple replicas. One of the reasons for this choice is that we care more about the performance of latency in this thesis, not the data reliability. So having less replica to select from could reduce the latency greatly. In addition, setting the CL to be one is also a very popular tuning policy, which provides the data to the users in less time. So, to keep the design simple, the CL is set to be one in this thesis and the tunable consistency setting could be left for future work.

While the status of each replica is unknown, then how to select the best replica is a problem. For example, a potential replica may experience an unknown problem and could not serve the request. Or the replica itself may experience some background activities and could only provide very slow service. So this uncertainty of each replica makes the replica selection much more challenging. In this thesis, we care more about the performance of the latency, then the best replica means the one, which could provide the fastest service. In the later section, it shows that the replica selection has great impact on the latency distribution, especially the tail latency. As a distributed system, Cassandra is popularly deployed in cloud environments, such as Amazon AWS or Google Cloud, which experience much more resource contention and other uncertainties. This could further aggravate the performance fluctuation.

Meanwhile, each replica exhibits performance fluctuations over time. This makes the replica selection a challenging problem. At the same time, the coordinator could not always choose the fastest replica to service the requests and this could lead to the

herd behavior problem [48]. Herd behavior means multiple clients send requests to the fastest/best replica, which could cause the load imbalanced and further degrade the server's performance. This could happen subsequently and cause the clients to choose other different nodes and repeat the same process.

In Cassandra, a snitch determines which data center nodes belong to and inform the Cassandra nodes about the network topology. Then the requests could be routed efficiently. Multiple snitches could be chosen according to user's requirements. Among these snitches, one is called Dynamical Snitching, which is designed to make replica selection decisions. Dynamical snitch is wrapped on other snitches by default and working for the read side only. It monitors the read latency for each node and provides this information to the coordinator node. Then the coordinator node attempts to avoid the poorly-performing nodes.

Taking a 6-node Cassandra cluster as an example, as shown in the Figure 4.1, when the client sends a request to the cluster, node1 is selected as the Coordinator node. Node3, node 4, node5 are the three replicas which store the data.

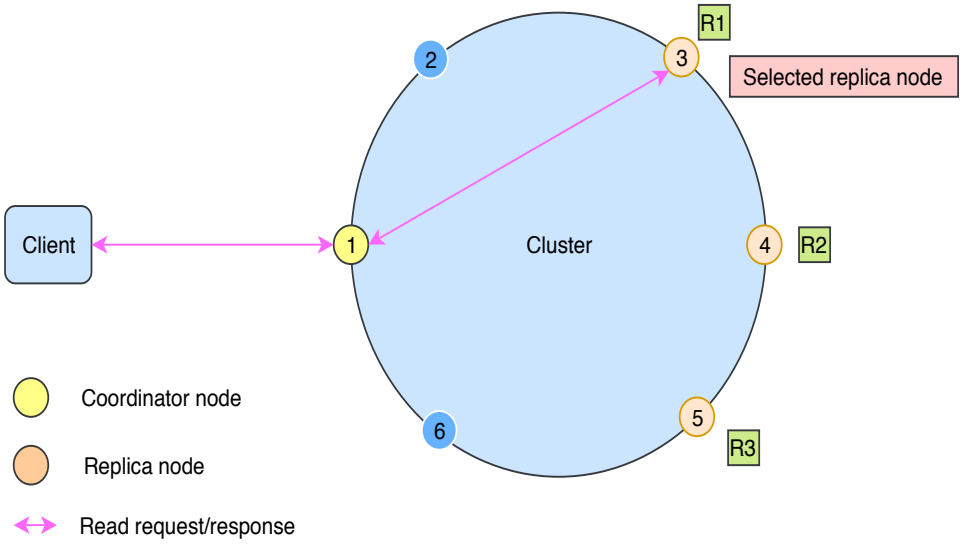

Figure 4.1: Remote read path

Since node1 has the latency information of each replica node, node3 is selected to read data from if it has the lowest latency. We call this process as a remote read.

Also, it could be possible that the coordinator node is one of the replicas. Then definitely we'll choose this node to read data from. This process is called a local read, as shown in the Figure 4.2.

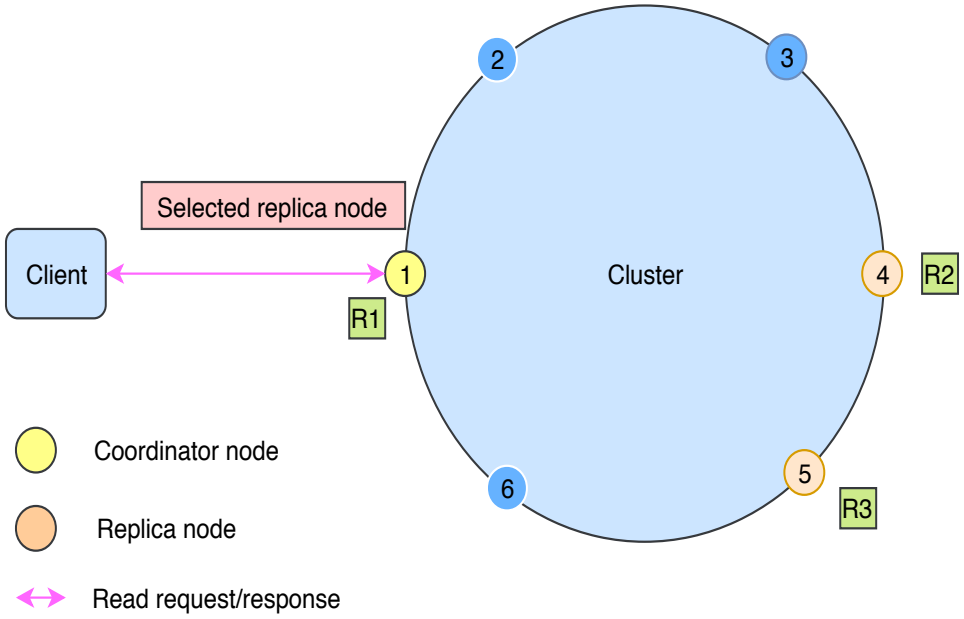

Figure 4.2: Local read path

However, the performance is poor when making replica selection decisions based only on the latency information of each node. In our experiments, we've measured the requests served in a 6-node cluster, which is deployed in the cloud platform of Amazon EC2 (the settings are the same as those used in the experiment result chapter). In particular, we tracked the number of read requests served in each 100ms interval.

The figure below shows that the default Dynamical Snitch is not working well and as a result the load varies a lot. The y-axes shows the number of read requests processed every 100 ms for one node. It ranges from 1 up to 28 requests per 100 ms. Especially

sometimes the number of read requests processed in 100 ms is only 1, which means the node is very busy at that time.

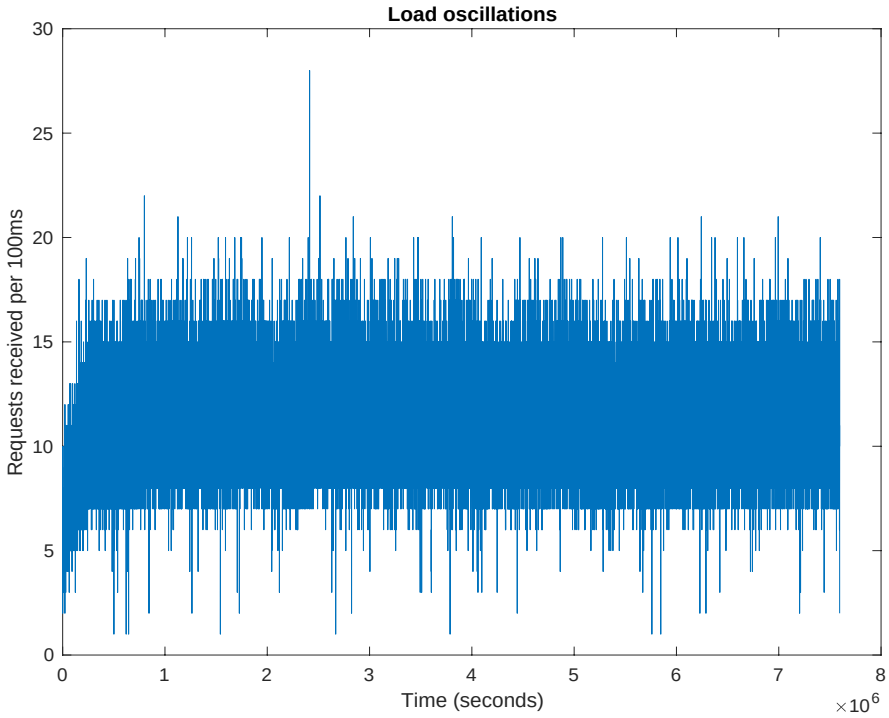

Figure 4.3: Requests received per 100ms

In addition to the vast load variance, other performance also experiences fluctuations, such as the latency, especially the tail latency, like 99th, 99.9th, 99.99th percentile latency. Dean [29] lists various sources for these latency fluctuation on Google cloud service. The phenomenon of long tail latency also happens in Apache Cassandra, because of the reasons mentioned in previous chapters and demonstrated below.

To demonstrate the long tail latency phenomenon in Apache Cassandra, a tool called YSCB (Yahoo! Cloud Serving Benchmark) [28] is used. YCSB is one of the most popular tools to evaluate and benchmark various databases, including both traditional databases and NoSQL databases. In YCSB, multiple workload patterns could be used to generate

customized workloads with different requirements. In our thesis, we care more about the read performance, so we use three workloads: update heavy, read heavy and read only. The operations in update heavy workload include 50% of read and 50% of update. The operations in read heavy workload include 95% of read and 5% of update. The operations in read only workload are all read. These three workloads are generated with the Zipfian or Uniform distribution.

Workloads	Operations	Distributions
A - Update heavy	Read: 50% Update: 50%	Zipfian/Uniform
B - Read heavy	Read: 95% Update: 5%	Zipfian/Uniform
C - Read only	Read: 100%	Zipfian/Uniform

Table 4.1: Workloads in YCSB [28]

With the same settings, we conduct experiments to get the tail latency information through YCSB. In these experiments, 6 nodes form a cluster. SimpleStrategy is used for the replication strategy and the replication factor is set to 3. Zipfian distribution is used for the workloads with 4 Gb data. Table 4.2 shows the latency performance regarding the previous three workloads, including average, min, max, 95th, 99th, 99.9th, 99.99th percentile latency. The unit for latency in the table is milliseconds.

Workloads vs Latency	average	min	max	95th	99th	99.9th	99.99th
A - Update heavy	3.09	0.54	91.19	5.76	8.44	15.56	32.30
B - Read heavy	2.37	0.48	215.30	4.46	5.86	11.62	22.59
C - Read only	2.59	0.44	235.14	4.40	7.54	12.22	33.37

Table 4.2: Latency information for different workload patterns

From Table 4.2, we can see the vast gap between the average latency and the tail latency. Taking the result of read-only workload as an example, minimum latency is 0.44 ms, and the maximum latency goes up to 235.14 ms, which must experience long service time. Meanwhile, the average latency is only 2.59 ms, while the 99.99th percentile

latency is 33.37 ms, which is nearly 10X times of the average latency. Also, the difference between the 95th, 99th, 99.9th values are also large. This long tail latency happens in all three different workloads, which could further influence the end-user delay. Below we investigate the reasons for the long tail latency and seek solutions to effectively reduce the tail latency.

4.2 Problem analysis

In read request processing, the read latency could be divided into several components. Taking a remote read request as an example, when the Coordinator node decides the replica and sends out a read request, there's a request time, which is basically the network latency between the two nodes. Since the request time is relatively small, it can be ignored.

Our experiment showed that the RTT(round-trip time) time between two nodes in the same rack is 0.75 ms in average, with the packet size 64 bytes. The max RTT time is about 1.59 ms, which is still relatively small compared with the whole latency. Actually in our experiments, this part is ignored when it's a local read, since there's no cross node connection. When the request comes to the replica and the replica needs to process the request and retrieve data from its memory or disk, this part is called the request process time. Based on the status of each replica, this process time could vary a lot and contributes majority to the whole latency. Specifically, the process time includes both the waiting time in the queue and the real service time for each request. Figure 4.4 shows the latency components for the remote read operation. We'll analyze each component in the part below.

Finally, after finishing the read process, the replica needs to send the data back to the Coordinator if it's a remote read process. The response time is also one of the important

parts in the latency, since the data we read could be quite large, like in Gigabyte unit.

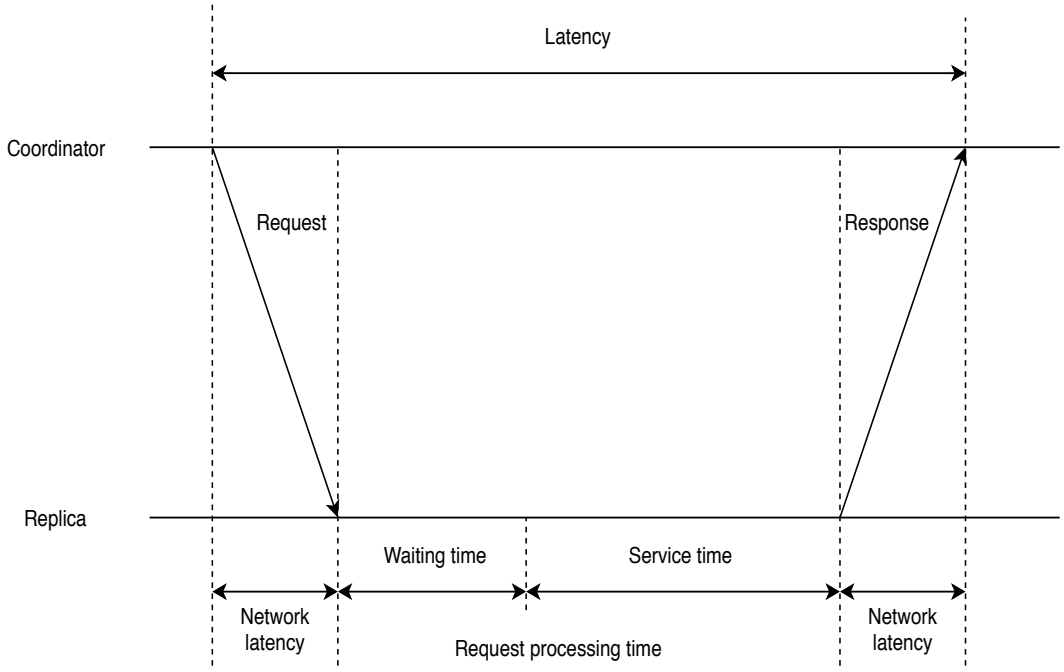

Figure 4.4: Latency components in remote read

To dig deeper into the internal components of the latency issue, we conduct experiments with different settings: small data set and large data set. Here small means the size of the data set (750 MB) is less than the RAM memory (1 GB) of the server. In this case, reading data only needs to access the memory for most situations. Only in very few cases, it needs to read data from disk. Large data set means the data set size (4 GB) is larger than the memory size (1 GB). In this case, it needs to access disk for every read request.

4.2.1 Small data set analysis

First of all, we start our experiments with a small data set, which consists of 0.75 million data of records. The size of each record is 1 KB, and thus the total data size is 750 MB. The total data size is less than the RAM memory of the experimental Amazon EC2 instance, which is 1 GB. Figure 4.5 shows the latency record for each read request in the millisecond

unit. From the figure, we can see the latency varies a lot, from minimum value of 0.26 ms to maximum value of 74.72 ms. The max value 74.72 ms is way out of the latency range and is possible an anomalous value. With experiment going on, the latency for each request could still go as high as 33 ms. Also, some of the requests could be served in more than 10 ms, or even above 20 ms.

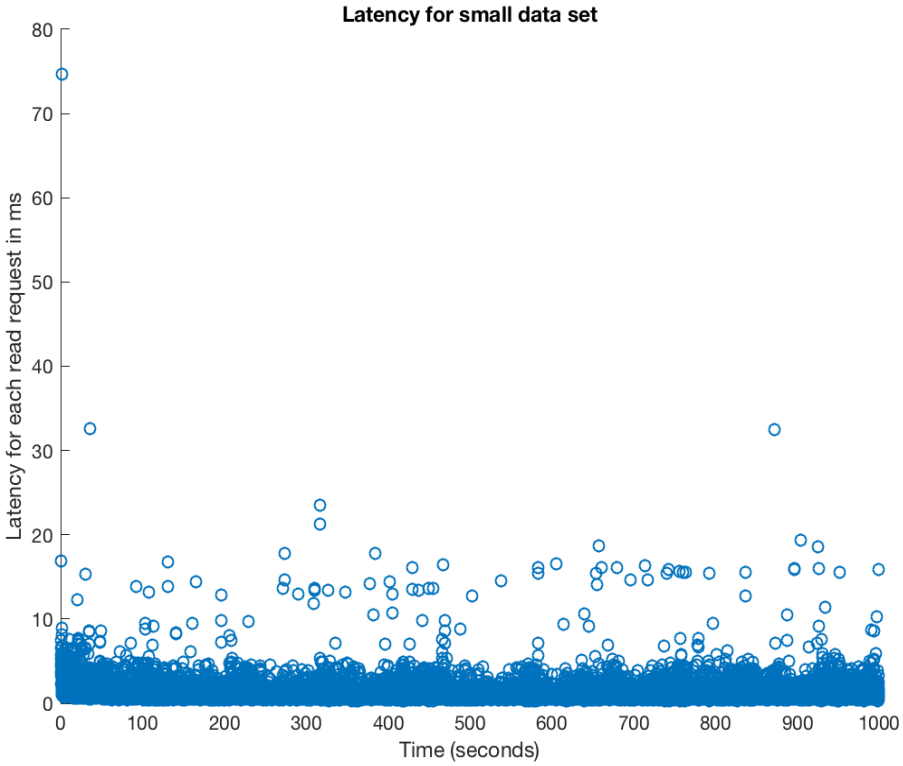

Figure 4.5: Latency for small data set

Furthermore, we plot the ECDF (Empirical cumulative distribution function) of the latency for one of the nodes as in Figure 4.6. The x-axis is the latency value, while the y-axis is the ECDF value. The figure shows that the average latency is about 0.88 ms, while the median latency is 0.68 ms. The 95% percentile latency is about 1.72 ms, which is about 2.5 times the median value. The 99% percentile latency is about 2.82 ms, which is about 4.1 times the median value. The 99.9% percentile latency goes up to 5.39 ms, which

is about 7.9 times the median latency. The statistic shows the big variance of the latency metric, while the tail latency contributes a lot. Also, we found the difference between the high percentile latency is large. For example, the 99.9% percentile latency almost doubles the value of 99% percentile latency and triples the value of 95% percentile latency.

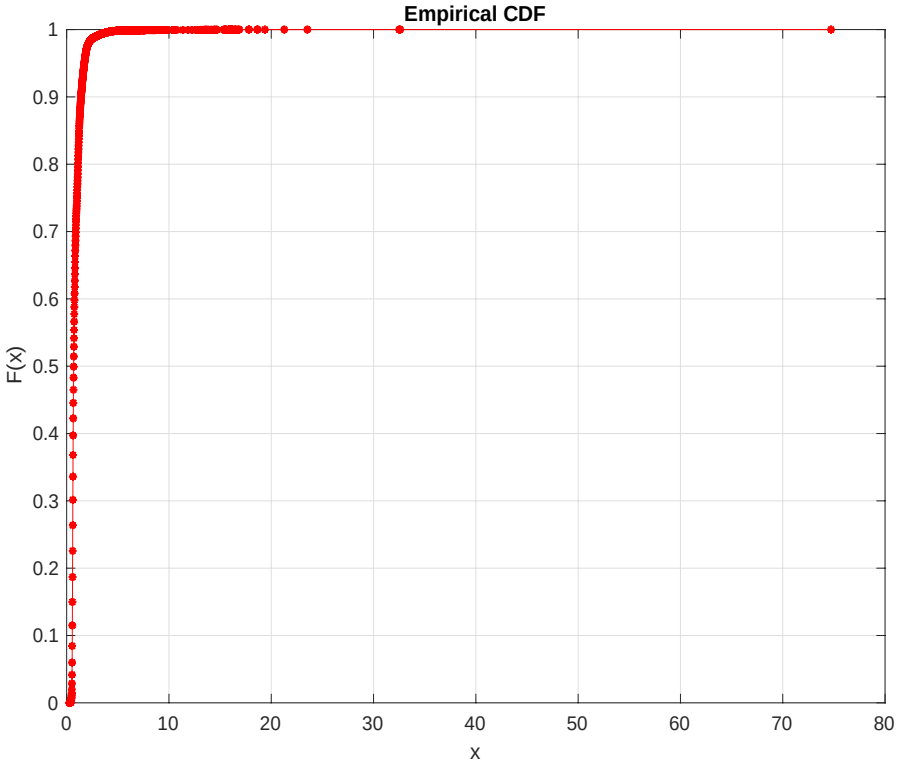

Figure 4.6: Latency ECDF for small data set

Figure 4.7 shows the PMF (Probability mass function) of the latency for the same node with small data set. To make the figure clearer, the zoomed plot has been added in the middle of the figure. From the figure we can see that, most of the latency lies between 0.5 ms to 0.8 ms, with 0.58 ms is the most frequent latency value. This latency is the first peak in the figure. The reason behind the first peak is the read access to the memory. Since the data set size is less than the RAM memory. So for most of the read requests,

they can be served through reading data from memory only. Only for few times, if the data not in the memory, it will need to access the disk.

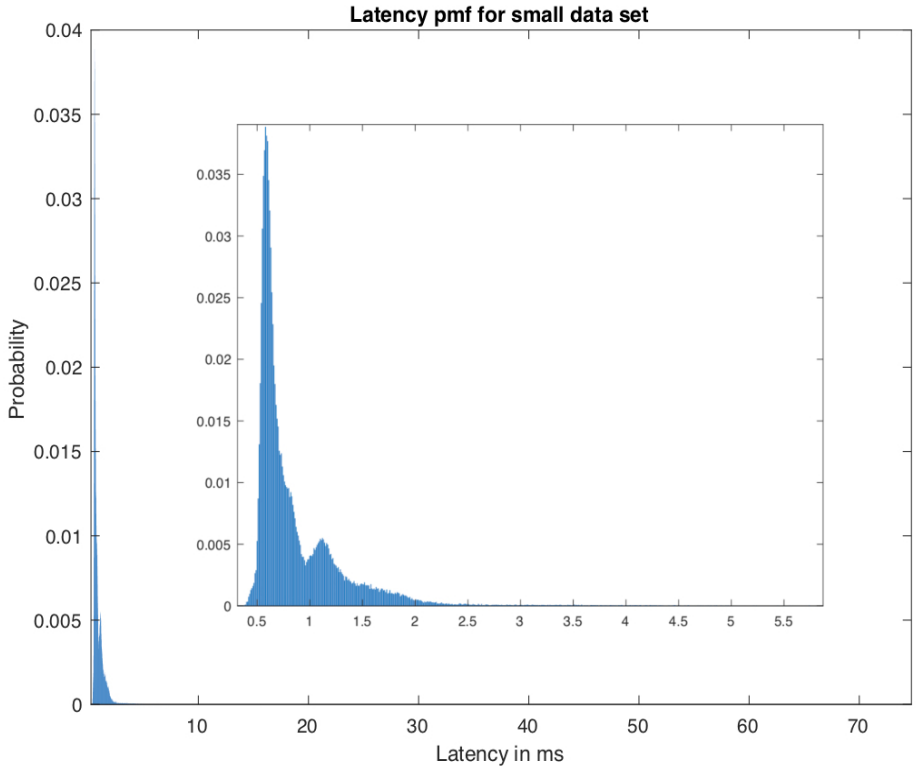

Figure 4.7: Latency PMF for small data set

However, for few times, if reading from the memory fails, it will read data from disk as well. This comes to the second peak in the PMF figure. The latency for the second peak lies between 0.96 ms to 1.24 ms. In this case, reading data needs to access the disk, which causes a longer reading time.

After further experiments, we found that the latency consists of different components. Figure 4.8 shows the latency components breakdown, which includes the waiting time in the queue, response time via the network latency, and service time to serve the request. The figure below shows the average ratio for each component with different workloads. The experiments show the response time is the major component for the latency, which

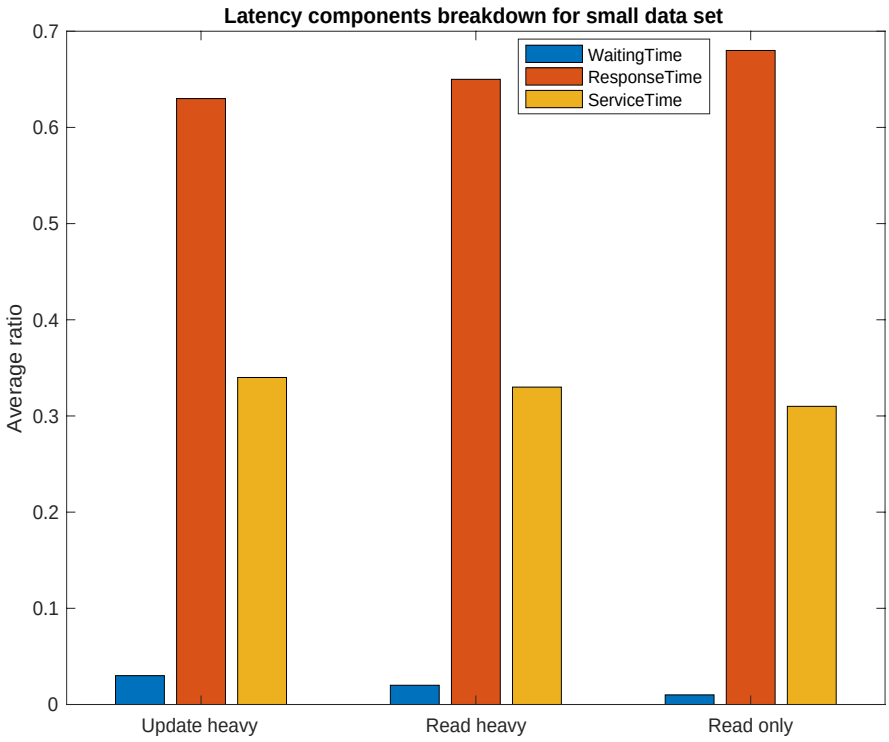

Figure 4.8: Latency components breakdown for small data set

takes more than half of the latency. Also the service time is the second important factor and it takes about 30% of the latency. Meanwhile, the waiting time in the queue only takes about 3% in average. This is because the data set is small, read requests only need to read from memory, which won't take too long to process. Consequently, the network latency part dominates the whole latency.

Figure 4.9 is the service time for each request on one of the nodes in the cluster. From the experiment result, we can see the service time varies a lot, ranging from the minimum value of 0.03 ms to the maximum value of 16 ms, while the average service time is about 0.33 ms.

Figure 4.10 is the ECDF distribution plot for the service time in one of the nodes with the small data set. From the figure, we can see that the average value of the service time

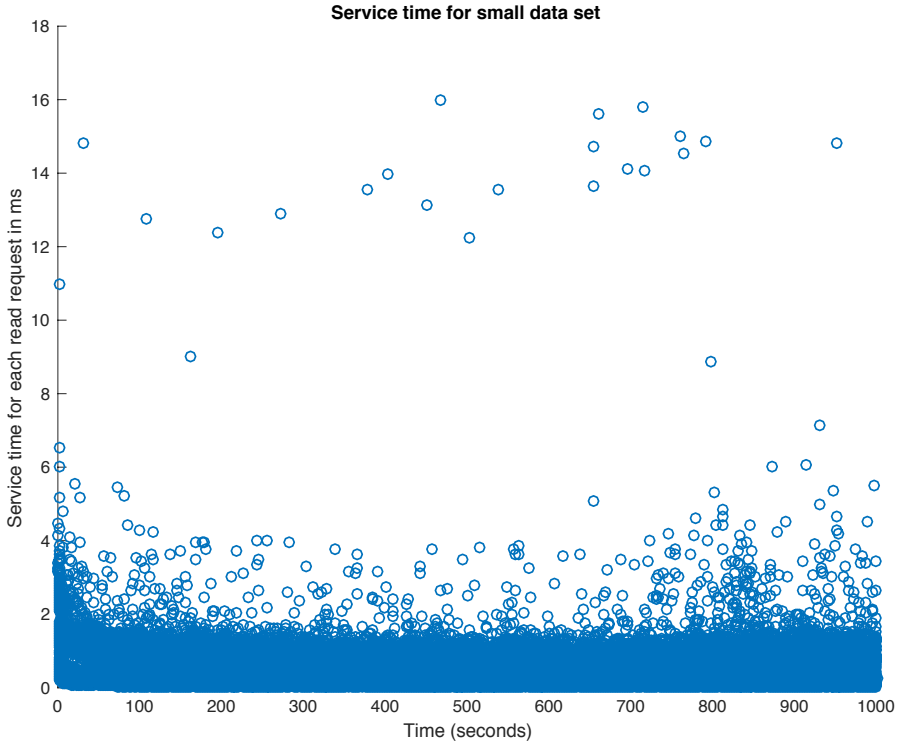

Figure 4.9: Service time for small data set

is 0.33 ms and the median value is 0.22 ms.

However, when coming to the high percentile values, it shows the 95%, 99%, 99.9% percentile values are 1.08 ms, 1.45 ms and 3.51 ms respectively, which is 4.9, 6.6 and 16 times of the median value.

Meanwhile, the big difference between the high percentile values also indicates the fluctuation of the service. The 99.9% percentile value doubles the 99% percentile value, and triples the 95% percentile value.

Figure 4.11 is the PMF for the service time with the small data set. From the statistic, it shows that most of the service time is less than 2 ms. 74.6% of the service time is in the range of 0.11 ms to 0.49 ms, which is the first peak in the figure. The reason for this is because that the size of the data set is smaller than the memory size. So some of the

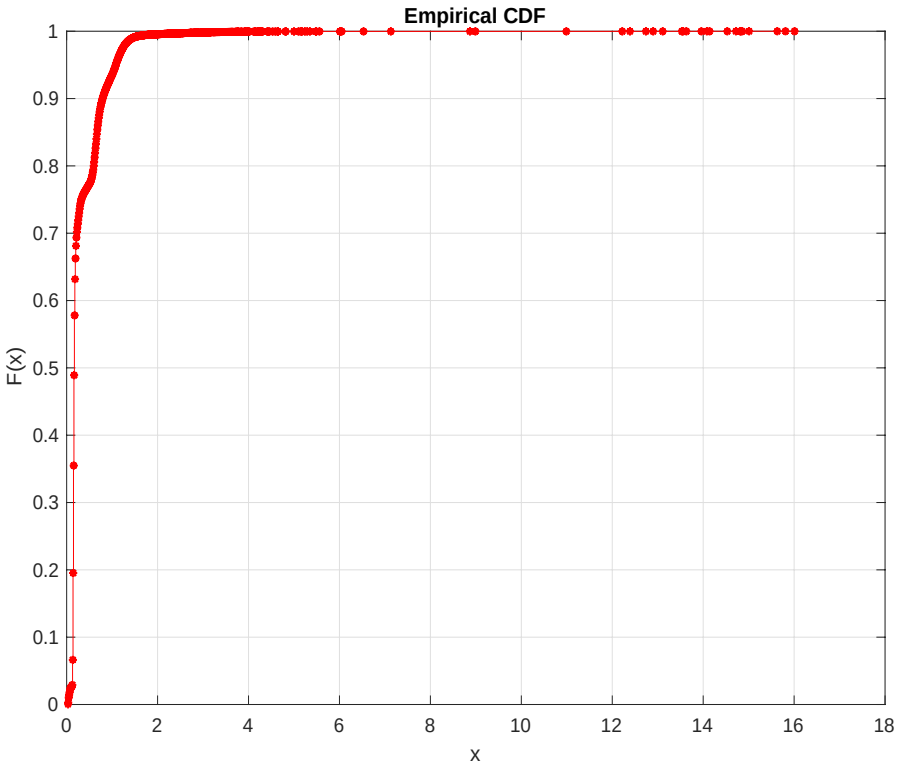

Figure 4.10: Service time ECDF for small data set

read requests need only to access the memory. Another peak is in the range of 0.51 ms to 1.24 ms. Similarly, this peak is occurred because of reading data from disk, when reading data from the memory failed. Accessing data in disk also may cause unpredictable issues. From the figure we can see, the service time could go up to 16 ms. This significantly increases the tail latency.

Because the response time takes the largest portion in the latency for the small data set, we analyze it below.

Figure 4.12 and Figure 4.13 show the ECDF and the PMF plot for response time with the small data set. Even though the response time varies, ranging from the minimum value of 0.12 ms to the maximum value of 66.97 ms, the figure shows that the response time concentrates a lot around the average response time, which is about 0.53 ms. This

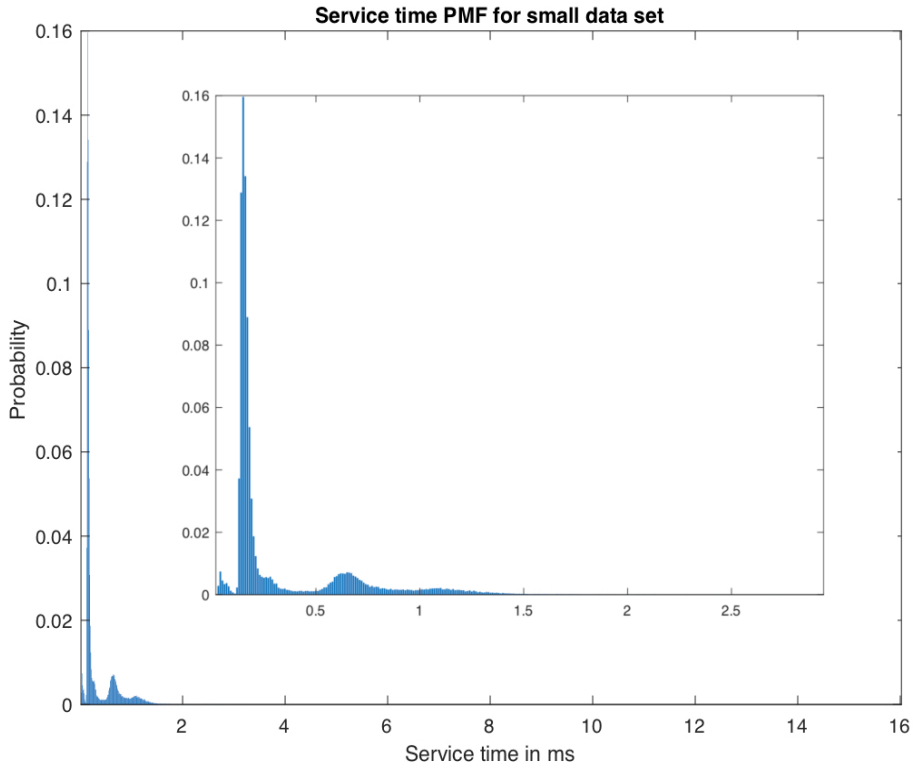

Figure 4.11: Service time PMF for small data set

value matches closely with our previous experiments on the RTT time experiments between two nodes in the same rack.

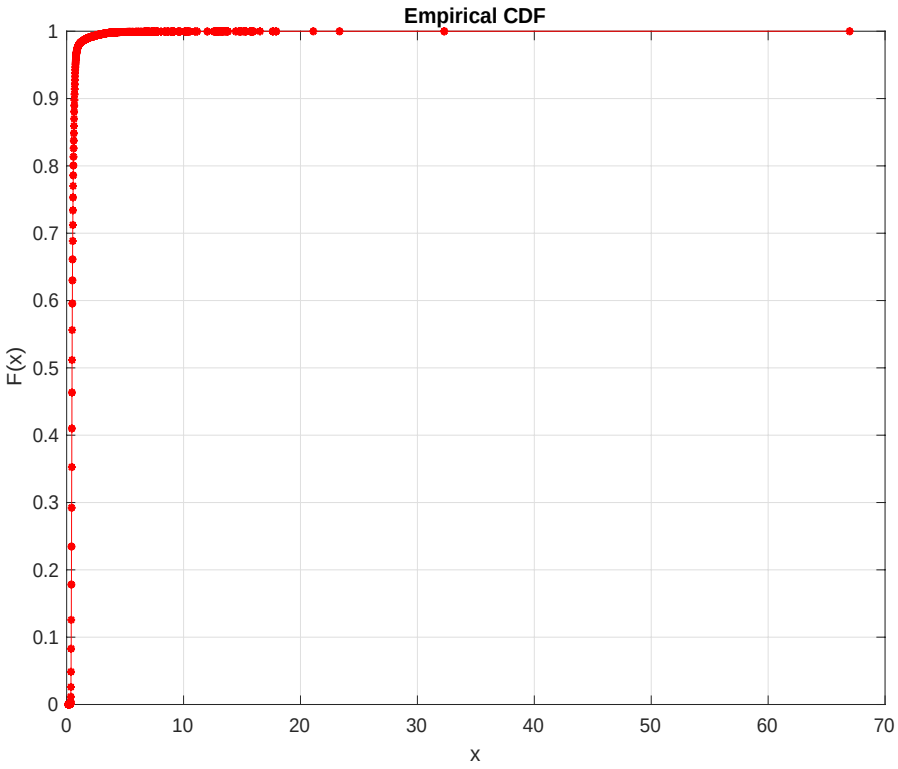

Figure 4.12: Response time ECDF for small data set

Figure 4.12 shows that the median response time is about 0.56 ms, which is very close to the average response time. The 95%, 99%, 99.9% percentile values are 0.75 ms, 1.88 ms and 4.2 ms respectively, which are 1.3, 3.4 and 7.5 times of the median value. Most of the response time is less than 1 ms. Actually from the PMF plot, the response time is closely subject to the discrete Gaussian distribution, with the mean value of 0.56 ms.

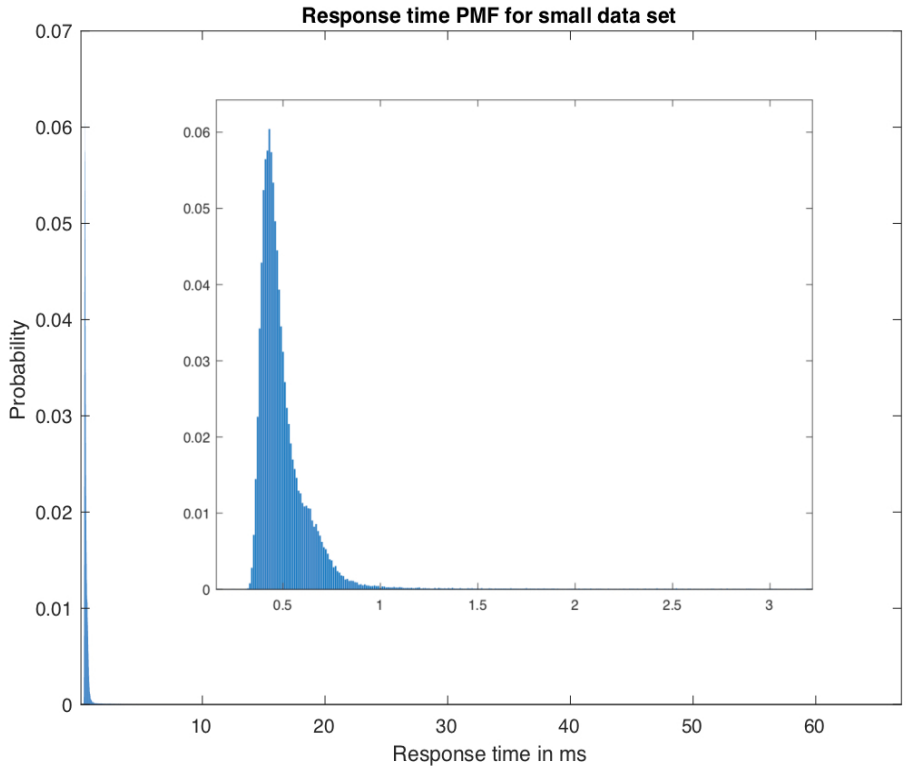

Figure 4.13: Response time PMF for small data set

From the analysis above, we can see for the small data set, the latency distribution mainly depends on service time.

4.2.2 Large data set analysis

We've worked on a large data set with the data size of 4 GB, which is greatly larger than the RAM memory of 1 GB.

For the larger data set, since the data can not be fit into the memory any more, so reading data from the node requires accessing the disk to read the full data set. Figure 4.14 shows the latency for each read request in a node with the millisecond unit. Compared with the small data set results, its value range varies even larger, ranging from the

minimum value of 0.22 ms to the maximum value of 165.18 ms. The figure shows that the peak latency value could go up to hundreds of milliseconds, which is definitely not tolerable in real time applications.

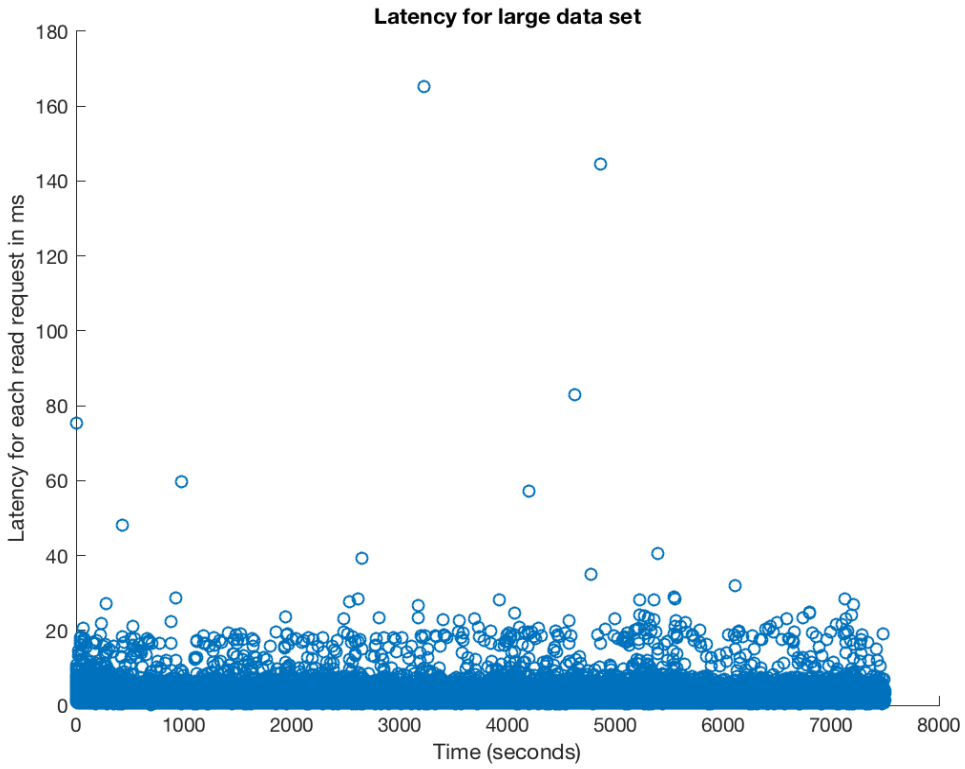

Figure 4.14: Latency for large data set

Figure 4.15 is the ECDF for the latency for one of the nodes with the large data set. It shows that nearly 99% latency value is less than 10 ms, which means most of the requests could be served within 10 ms. The average latency value is about 1.36 ms, while the median latency value is 1.27 ms. The 95%, 99%, 99.9% percentile latency are 2.38 ms, 3.99 ms and 6.81 ms, which are 1.8, 3.2 and 5.4 times of the median latency value respectively. The experiment results also show that the tail latency is pretty long, even the average latency values are a few milliseconds. Meanwhile, the differences between the high

percentile latency value is also very large. For example, the 99.9% percentile latency is about 1.7 times of the 99% percentile latency, and 2.9 times of the 95% percentile latency.

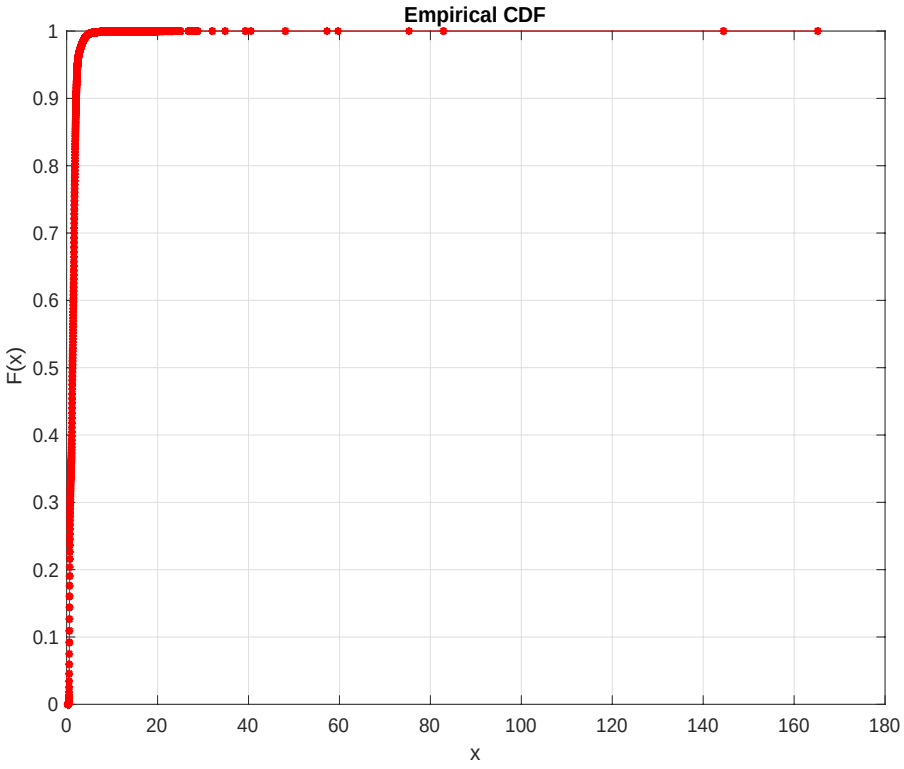

Figure 4.15: Latency ECDF for large data set

Figure 4.16 shows the PMF for the latency. The first peak value of the latency is about 0.62 ms and the second, the third are around 1.18 ms and 1.62 ms respectively. Similarly, the first peak value is the time to read data from the memory. If the data not completely stored in the memory, it will access the disk to read extra data from the disk. Reading data from disk is more time-consuming, which leads to the second peak in the figure. During the reading process, back ground activities like garbage collection, data compaction could happen. This further increases the time to read the whole data. When the background activities happen, the service to the read request could be stopped fully.

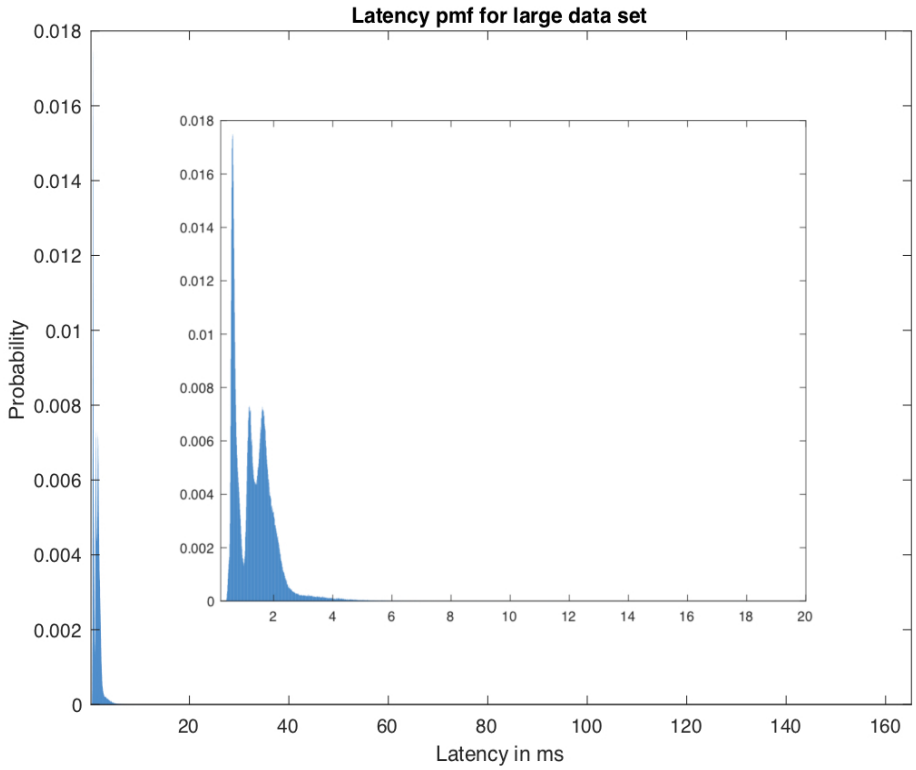

Figure 4.16: Latency PMF for large data set

Similarly, we break down the components for the latency and analyze each component in it. Figure 4.17 shows the average ratio for each component with different distribution workload. From the figure we can see the waiting time takes larger portion in the latency, compared with the small data set experiments. The average ratio of the waiting time over latency goes up to around 8%. Meanwhile, the figure also shows that the service time takes the largest portion in the latency, with the value around 50%. This is because the data size gets larger, then the chance of request congestion increases as well. Also, more background activities happen under-ground, such as garbage collection, and data compaction. These all could make the time to serve the request longer.

Among the latency, we conducted further experiments to measure the service time for each read request and found that the service time also varies a lot with an even larger

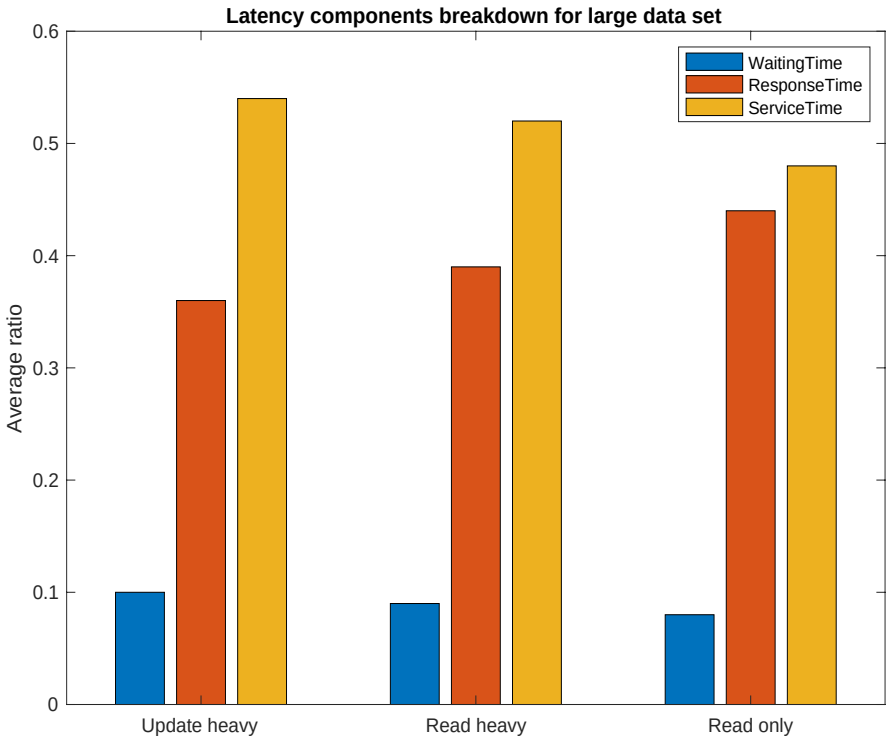

Figure 4.17: Latency components breakdown for large data set

range. Figure 4.18 below shows the measurement of the service time for each read request. It ranges from the minimum value of 0.03ms to the maximum value of 164.41ms, which shows that the peak value could go up to hundreds of milliseconds. Compared with the average service time, it's about 200 times. This high value could contribute greatly to the tail latency.

Similarly, we analyzed the distribution of the service time for the large data set. Figure 4.19 shows the ECDF of service time for the large data set. The average and median values for the service time are 0.79 ms and 0.72 ms respectively, which are very close. Statistic also shows that the 95%, 99% and 99.9% percentile values for the service time are 1.72 ms, 3.04 ms and 4.80 ms respectively, which are 2.4, 4.2, 6.6 times of the median service time. The differences between the high percentile values are also large. The 99.9% percentile

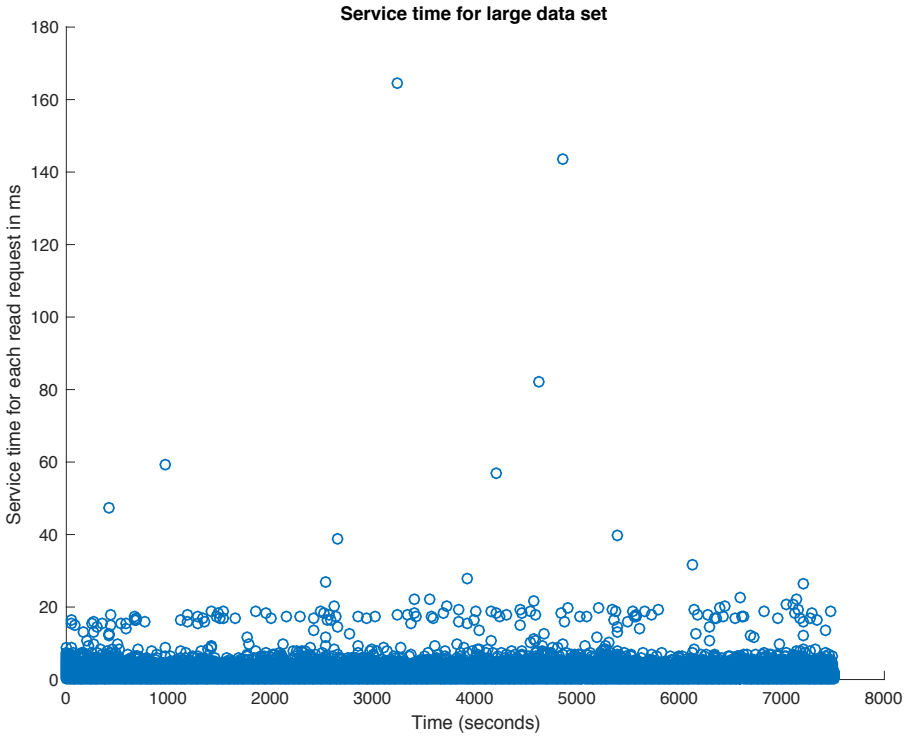

Figure 4.18: Service time for large data set

value is 1.6 times of the 99% percentile value and 2.8 times of the 95% percentile value. This further indicates the service time also contributes a lot to the tail latency, especially the highest value.

Figure 4.20 shows the PMF for the service time with the large data set. Different from the PMF of the small data set, there are three peaks in the figure, similar to the latency distribution. This is because of the same reason as we analyzed before. The first peak is because of the time to read data from the memory, while the second peak is because of the time to access the disk. The reason for this is that the data set is larger than the RAM. In order to read data from the cluster, it must access the disk to read the data in SSTables. The last peak is because of the background activities. If the background activities are active, it stops the node to serve the request completely. For example, the full garbage

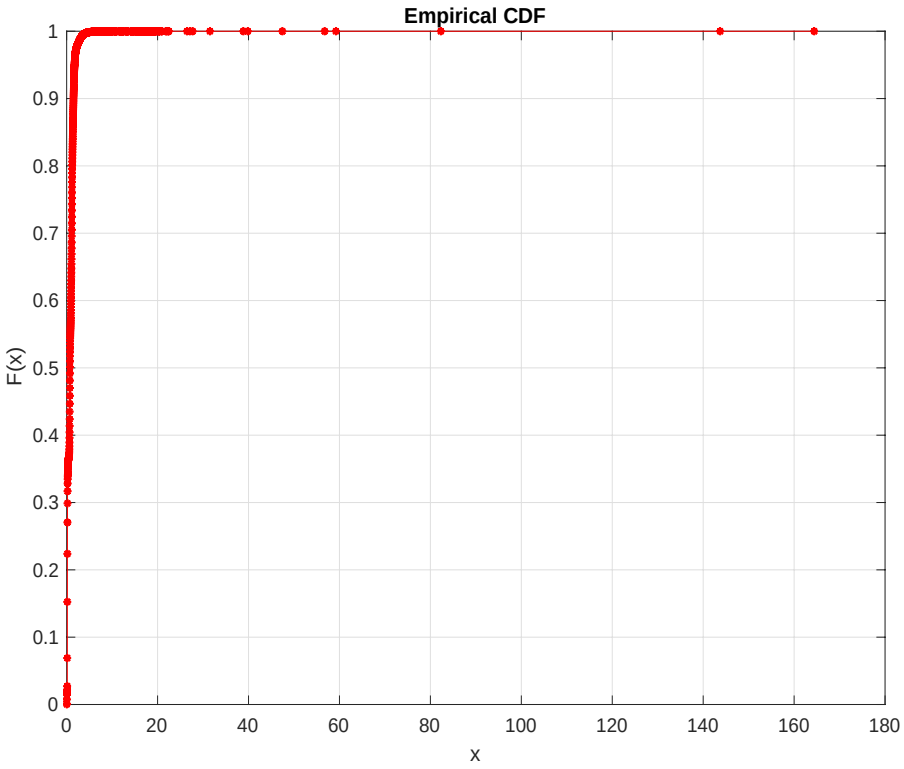

Figure 4.19: Service time ECDF for large data set

collection is a stop-the-world event. When it's triggered, no other process could run at the same time. This will definitely affect the reading process.

Similarly, we find that the distribution of the service time is similar to the distribution of the latency. The figure for the latency breakdown also indicates that the service time plays a very important part in the whole latency, which is more than 50%. As the network latency takes another portion, it's more related to the resource allocation of the network bandwidth, which we cannot control it directly.

However, for the service time, we could choose the node with a less service time. So basically, the service time is a very good metric for the latency performance.

At the same time, we also measured the response time to get the read result back to the coordinator, which is the network transmission latency. In our experiments, if the

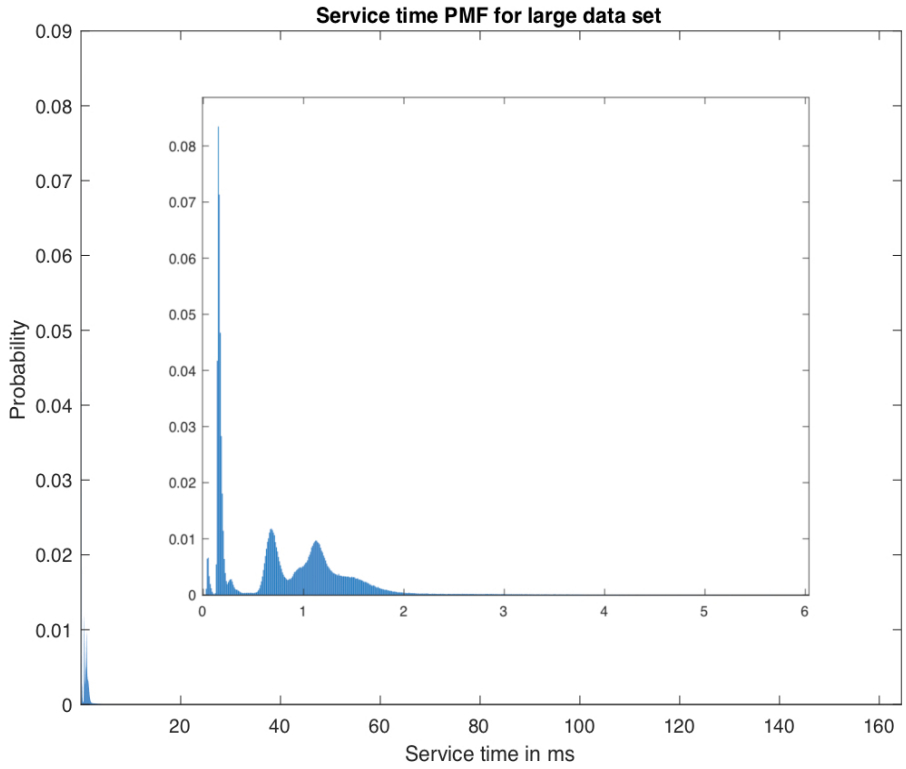

Figure 4.20: Service time PMF for large data set

request is served in a local node, then this part is set to be zero, because in this case, no cross-nodes transmission is needed.

Figure 4.21 and Figure 4.22 are the ECDF and PMF for the response time. The minimum response time and the maximum value are 0.12 ms and 60.48 ms, respectively. From the figures, we can see that the average response time is about 0.55 ms, while the median value is about 0.49 ms. The figure also shows that the 95%, 99%, 99.9% percentile latency are 0.72 ms, 1.65 ms and 4.16 ms, which are 1.5, 3.3 and 8.5 times of the median response time, respectively.

However, the statistic shows that the response time concentrates around the average response time, which is about 0.55 ms. This is very close to the RTT measured in our previous experiments.

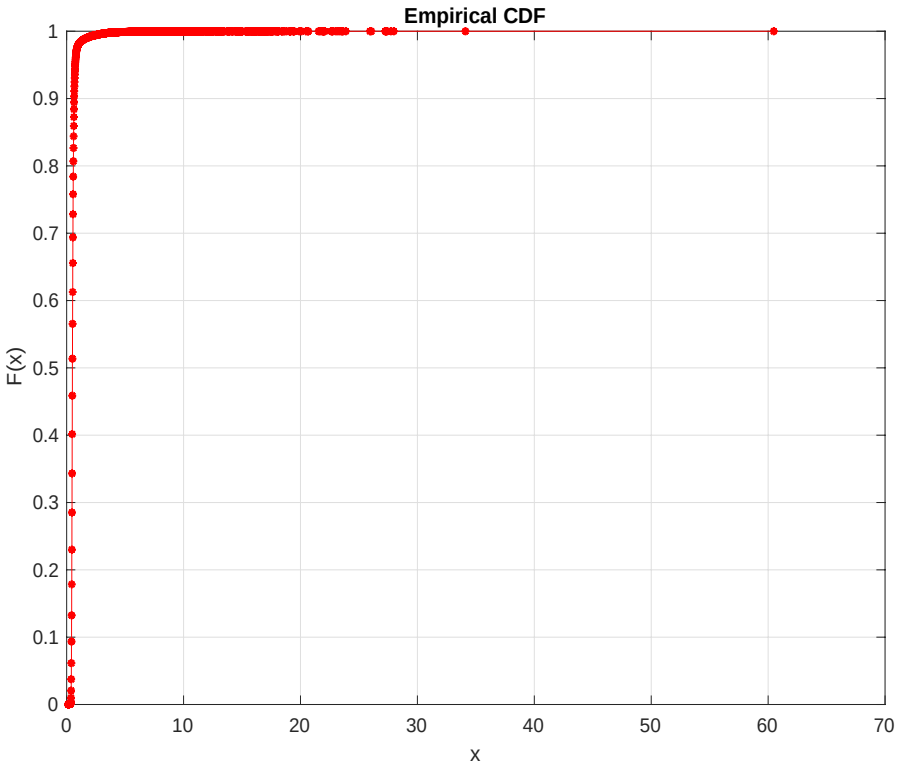

Figure 4.21: Response time ECDF for large data set

Another important part which also contributes to the latency is the waiting time in the queue, especially when the node is busy with handling multiple requests. This waiting time could be a very powerful indicator to the latency performance.

However, in C3[60], it doesn't provide information for this part. To show how the waiting time impacts the latency, we also conduct experiments using the large data set with similar settings above.

Figure 4.23 shows the PMF for the waiting time in the queue. From the experiment results, we can see that the waiting time in the queue is 0.01 ms in most cases, this is because of the powerful multi-threading design of Cassandra.

However, our results show that the waiting time could go up to 23.1 ms, which also contributes a lot to the latency. Therefore, it is important to provide this information to

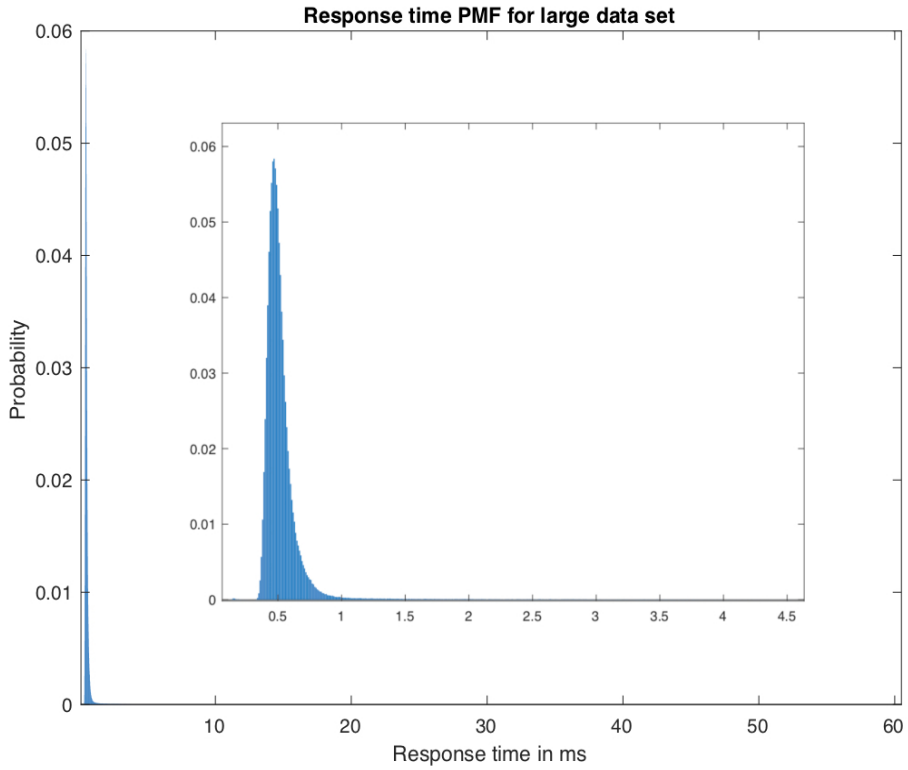

Figure 4.22: Response time PMF for large data set

the coordinator node, so that it does not select a node with a long waiting time as the replica.

4.3 Approach to reduce tail latency

Based on our experiment result and analysis, we propose two approaches to reduce the tail latency: (1) Optimize replica ranking system with a smooth tail value adjustment algorithm; (2) Execute the local read of the coordinator node synchronously, to avoid the unnecessary waiting time.

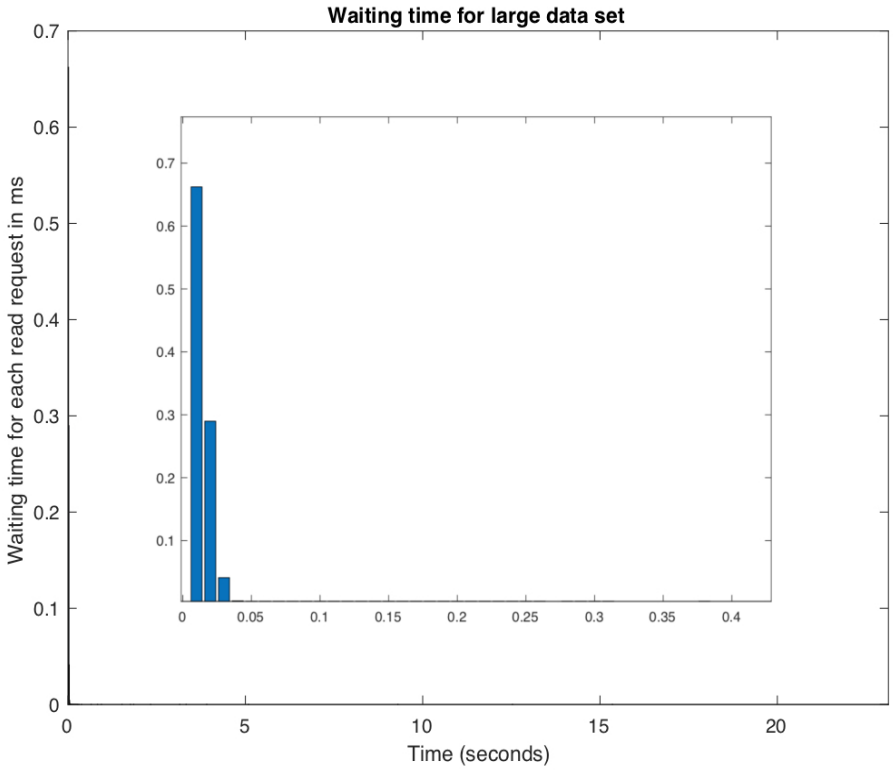

Figure 4.23: Waiting time PMF for large data set

4.3.1 Optimized replica ranking

When a read request comes in, the coordinator decides whether to read data from the local node or a remote node, based on the collected latency information. When multiple replicas exist, the coordinator chooses the node with the least latency for the past requests. This latency information is measured every 100 ms as shown in Figure 4.24. That means every 100 ms, each node calculates the score based on the last request.

However, as analyzed before, this estimation may not show the real accurate status of each node. As the background activities like data compression and garbage collection have a great impact on the latency performance. Similar to the algorithms in c3, we also consider the parameters of network latency, service time and the queue size. These

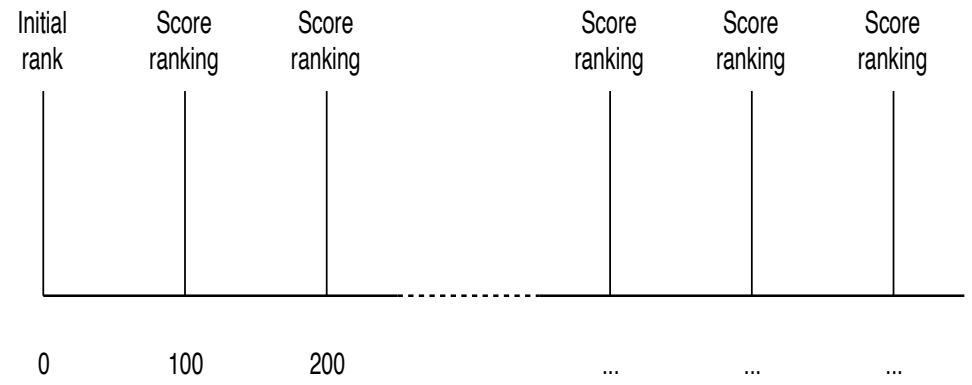

Figure 4.24: Server ranking

parameters play very important roles in the latency. Meanwhile, we also consider the waiting time in the queue.

Here, the network latency is the time interval from the instant when the data is read from the replica to the instant when the data is sent back to the coordinator node. If it's a local read in the coordinator node, then the network latency is 0. Otherwise, it's the time to send the data from the replica to the coordinator node. Queue size is the size of the current request queue in each server. When the queue size is large, the node is very busy. Service time is the time spent to process each request internally. For a read request, it may experience multiple stages in memory, disk or both, which depends on the data set size to read.

For each read request, the replica replies back with the corresponding queue size q and the service time 1/u (the inverse of the service rate) to the coordinator node. Meanwhile, it monitors the network latency between the coordinator node and the replica. The queue size is calculated with Exponentially Weighted Moving Average (EWMA) [8], to smooth the metric.

However, because of multiple servers and time-varying service time, a linear function of q could not effectively reflect the real status of each replica. Similar to the idea in the

paper [21], the replica with a longer queue is penalized. Here we adapt a cubic function of the queue size, which makes a good trade-off between the preferred faster replicas and robustness to time-varying service time.

Algorithm 1 Tail Value Adjustment Algorithm

```
1: procedure TAIL-VALUE-ADJUSTMENT-ALGORITHM(current_value, α)
2:     adjusted_value ← current_value                ▷ Initialize the adjusted value
3:     for every 100 ms do
4:         Get the 90%, 95%, 99%, 99.9% value based on previously collected data
5:         if 99%_percentile_value ≥ α * 90%_percentile_value then
6:             adjusted_value ← 90%_percentile_value
7:         else if 99.9%_percentile_value ≥ α * 95%_percentile_value then
8:             adjusted_value ← 95%_percentile_value
9:     return adjusted_value
```

Both the response time and service time fluctuate a lot, and their tail values are calculated using Algorithms 1 and 2. Algorithm 1 describes how we calculate the tail value of a performance metric, such as the response time or service time. Algorithm 2 describes how we find a specific percentile of a performance metric. In Algorithm 1, we have a list to keep the past values. For every time window, we calculate the long tail percentile value as described in Algorithm 2. We adjust the value based on the differences between those tail percentile values. For example, if the 99% percentile value is larger than α times of the 90% percentile value, this means the tail percentile value is very high and it takes a very long time to process the request. So we set the adjusted value to the 90% percentile value, instead of the most recent value. We adjust the tail values similarly for 99.9% percentile value and 95% percentile value. In our following experiments, if not specified, the value of α is set to be 2.

Based on our previous measurement on the waiting time for the queue, we can see that in most of the cases, the waiting time is relatively small with an average value of 0.02 ms. But for few times, this waiting time could go up to 23 ms. This indicates that

Algorithm 2 Get Percentile Algorithm

```
1: procedure GET-PERCENTILE-ALGORITHM(valuelist, percentile)
2:     valuelist ← sort(valuelist)
3:     index ← (int)Math.ceil((percentile/100) * valuelist.size)
4:     return valuelist.get(index − 1)
```

many requests are waiting in the queue and the node is busy with the requests. To avoid selecting a replica with a long waiting time, we add the waiting time to our score function as well, but with a more sensitive weight β, which is set to 10. This is because in most cases, the waiting time is small. A large weight makes the score more sensitive to the waiting time. In our experiment, the 99.9% percentile waiting time is about 0.06 ms. Even if we multiply the value with β, it's about 0.6 ms, which is close to the average value for the service time and response time. But in a few cases, when the waiting time goes up to 1 ms or more, this value could go to 10 ms, which probably dominates the score function. So based on this mechanism, we indirectly avoid the long waiting time and further reduce the tail latency.

With the above information, the score for each replica is calculated and then ranked accordingly. Below is the score calculation formula for each replica adapted from [60], where N_s is the network latency, u_s is the service rate. These two parameters are calculated using our tail value adjustment algorithm. q_s is the queue size and Ω_s is the waiting time in the queue. These two parameters are calculated using the EWMA with the weighted factor set to be 0.9. β is the weight for the waiting time, which is set to be 10 as we mentioned before.

$$\phi_s = \tilde{N_s} + \bar{q_s}^3 / \tilde{u_s} + \beta * \bar{\Omega_s} \tag{4.1}$$

The above parameters are calculated for each request, and piggybacked to the client.

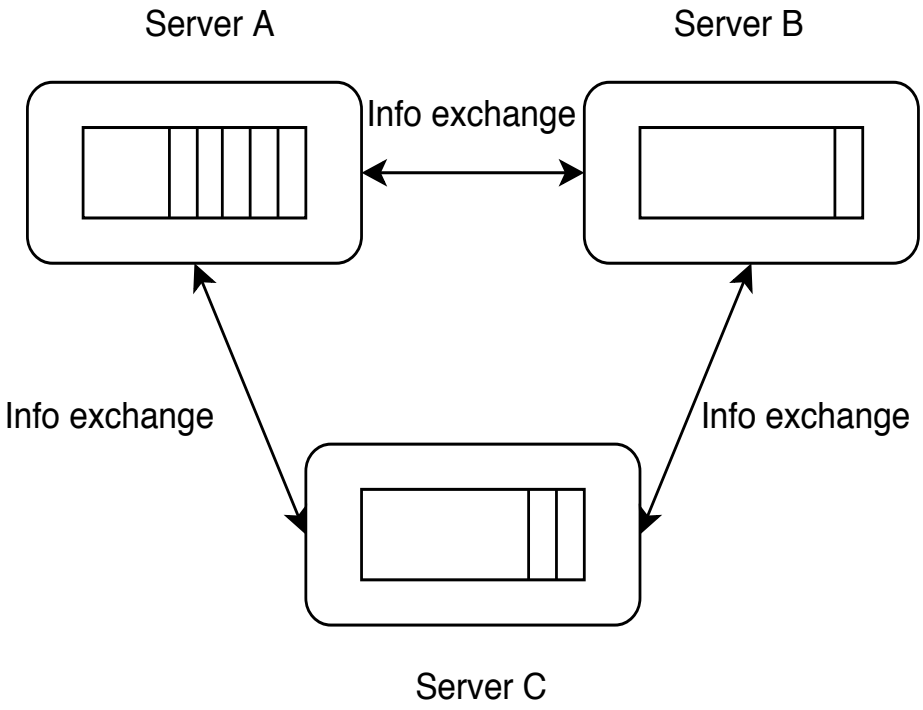

Figure 4.25: Server status exchange

Then the client chooses the best replica, based on the score calculated every 100 ms. Then the read request could be forwarded to the replica with the lowest score. Figure 4.25 shows how the nodes share this information.

4.3.2 Local read improvement at coordinator node

As a large Java-based project, Apache Cassandra uses concurrent programming techniques, to efficiently utilize the system resources, further improve the performance, especially the latency performance.

For each request at the read/write stage, it is placed in the thread pool and executed asynchronously. The default thread pool size for each Cassandra node is 32, which means every time, the node could serve at most 32 concurrent read operations. The multi-threading mechanism greatly improves the system performance of Cassandra. The

thread pool size is adjustable.

However, the design of this parameter should be carefully considered, as a small pool size could lead to poor performance, while a large pool size uses more resources which could also be harmful to the system.

For a node in the cluster, it could be chosen by the client as the coordinator node. If at the same time, this coordinator is also one of the replicas and is chosen for data-reading, then the read requests are served locally and we call these requests local requests. Meanwhile, this node could also receive requests from other chosen coordinator nodes, and we call these requests as remote requests. As a result, both local requests and remote requests co-exist in the thread pool.

As mentioned before, the SDEA design of Apache Cassandra puts different events into different stages. This is true even for local read requests with consistency level one.

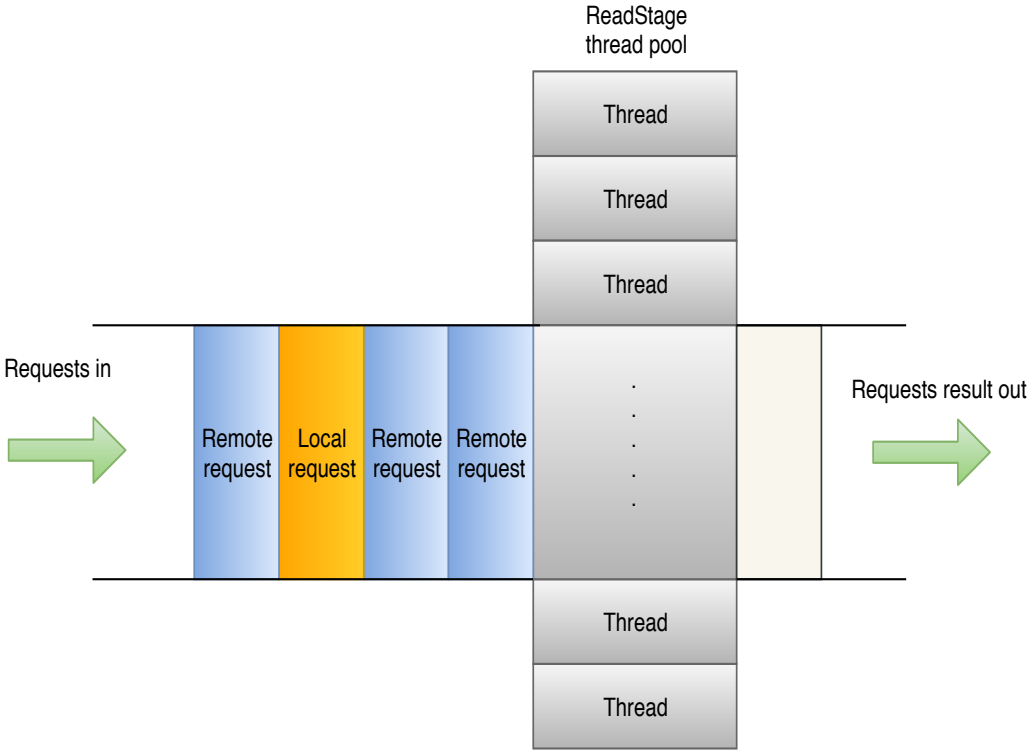

Figure 4.26: Local/remote requests co-exist

As shown in Figure 4.26, when a local read request reaches a coordinator node, the coordinator node puts this local read request into the read stage for asynchronous execution, together with other remote read requests. This local read request needs to be enqueued into the read stage, together with other remote requests. To execute the local request, it needs to wait for the finish of other remote requests queued in-front. This process causes possible waiting time in the queue and potential context switches, which adds unnecessary latency.

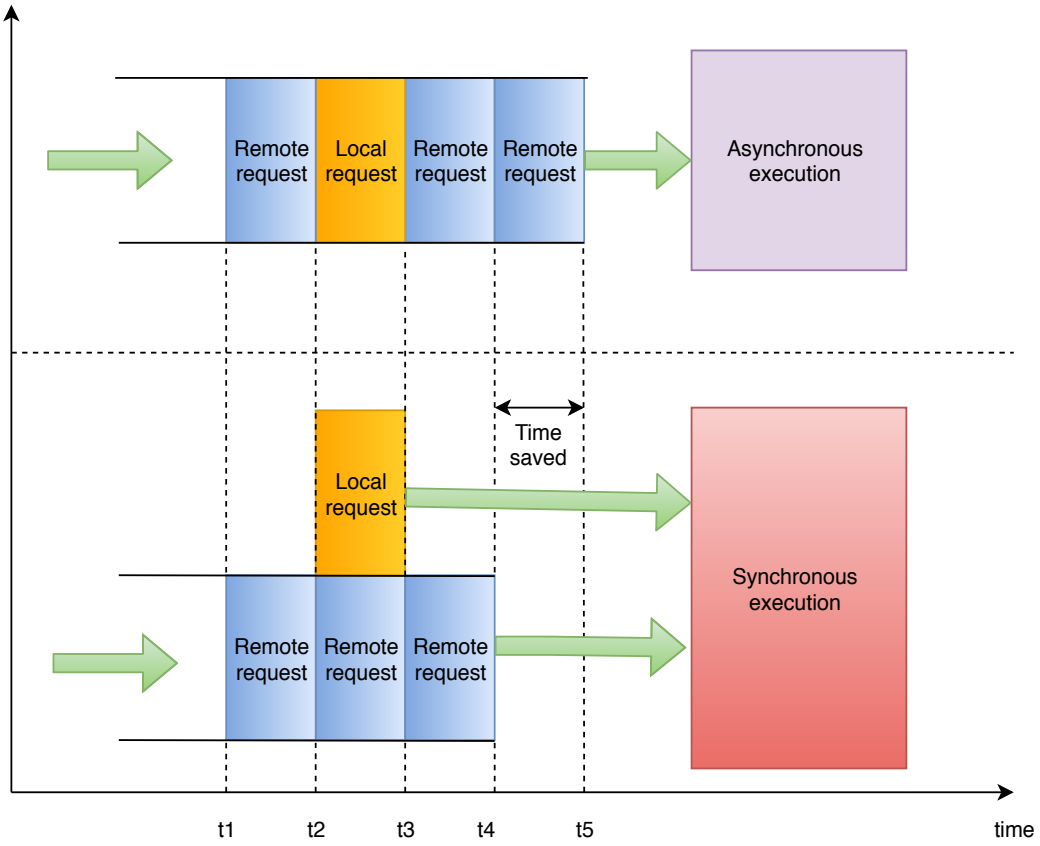

Figure 4.27: Async/sync execution of read requests for Coordinator node

As illustrated in Figure 4.27, to avoid this unnecessary latency and improve throughput, we can simply execute the local read requests at the coordinator node synchronously in the request thread. Since a local request can be served in a very short time, if the

coordinator is not busy when receiving the local request. Through synchronous execution, this could reduce the waiting time in the queue for the local read requests, further improve the overall throughput performance. Since the local read and remote read requests are executed synchronously, this could save potential waiting time in the queue and reduce the read latency further.

However, here we need to note that the distribution of local read and remote read depends on the coordinator selection algorithm. This algorithm is defined in the driver side for load-balancing of the coordinator, which is out of our scope for this thesis.

Chapter 5

Evaluation and Results

5.1 Experiment set up

Our proposed methods are implemented in Apache Cassandra of development version 2.1. The Cassandra nodes are deployed on Amazon EC2 with an instance type of t2.micro (Variable ECUs, 1 vCPUs, 2.5 GHz, Intel Xeon Family, 1 GiB memory, EBS only). The instances and nodes are tuned according to the recommended settings from Datastax [6].

We have a cluster of 6 nodes installed in the same zone of California in Amazon EC2. Because nodes are within the same zone and nodes communicate through the private IPs, we set up the replication strategy to be SimpleStrategy. The replication factor is set to 3. Regarding the workloads, Yahoo Cloud Serving Benchmark(YCSB) [17] is used. YCSB [28] is the most popular benchmark tool to evaluate cloud serving systems. The YCSB version is 0.8 and it's installed in a separate Amazon EC2 instance, with the same system settings with the Cassandra nodes. We insert 400,000 1KB-size records generated by YCSB to the cluster and they serve as the data set to operate on. The keyspace and table for these records are set as name of ycsb and usertable. In each record, ten fields with names ranging from field0 to field9 are modeled. The size for each field is 100 bytes. Meanwhile,

an identified primary key is used, which is a string like user123456.

In the experiments, a read operation reads an entire record from the table, a update operation updates a record by replacing the value of one field. The requests generated by YCSB follow the Zipfian access pattern, which is typically used in photo tagging, session-store and user-profile applications. Each measurement involves 1 million of operations of the workload and is repeated three times. For the experiments below, if not specified, the multiply factor in tail value adjustment is set to 2. The weight to get the EMA value is set to 0.9. Table 5.1 shows the experiments setup and Table 5.2 shows the settings for the Amazon EC2 instances.

Parameter	Value
Amazon EC2 instance type	t2.micro
Cluster size	6
Replication strategy	SimpleStrategy
Replication factor	3

Table 5.1: Experiment parameters

Parameter	Value
Model	t2.micro
vCPU	1
Memory size	1 GB
Storage	EBS-only

Table 5.2: Amazon EC2 instance settings

Two common workload patterns are used to evaluate our new algorithm: update-heavy(50% reads and 50% updates), which is usually used in a session store to record recent actions, read-heavy(95% reads and 5% update), which is typically used for photo tagging and read-only(100% reads), which is commonly used in user profile cache applications [17].

5.2 Experiment Results

To evaluate the performance of the new algorithm, we compare it with the default setting for Apache Cassandra, as well as the C3 algorithm [60]. We focus on the performance of latency, especially the tail latency and the throughput as well. We want to make sure that the new algorithm reduces the tail latency without sacrificing the mean/median latency, and improve the throughput (number of requests per second).

5.2.1 Impact of data set size

In this group of experiments, we study the impact of a small data size (750 MB) with different workloads. The operation distribution follows a Zipfian distribution.

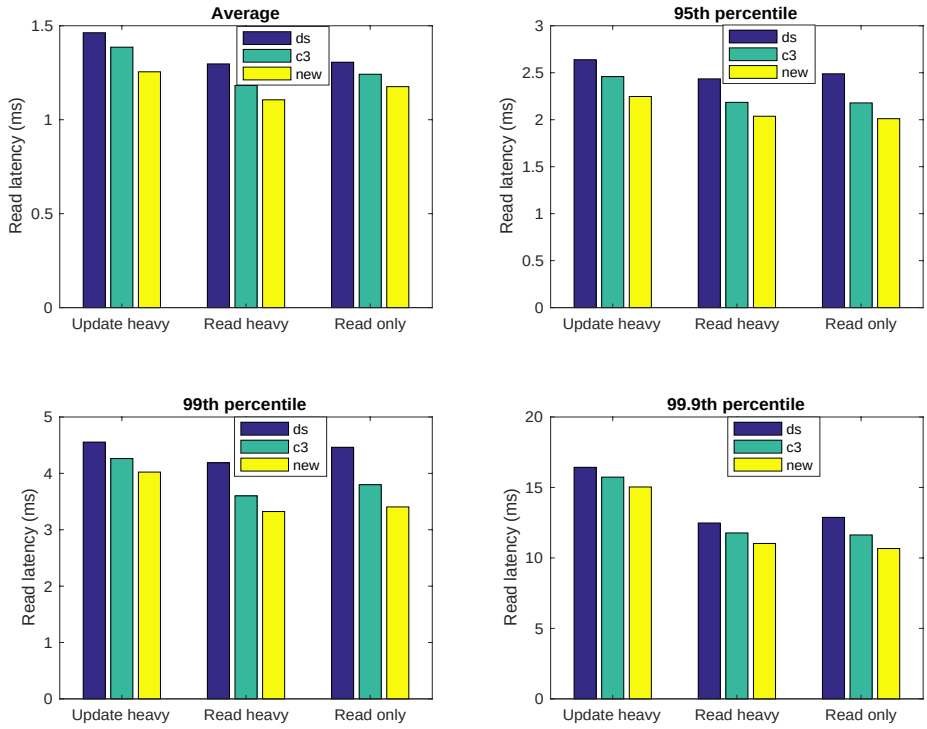

Figure 5.1: Read latency comparison with small data set in Zipfian distribution

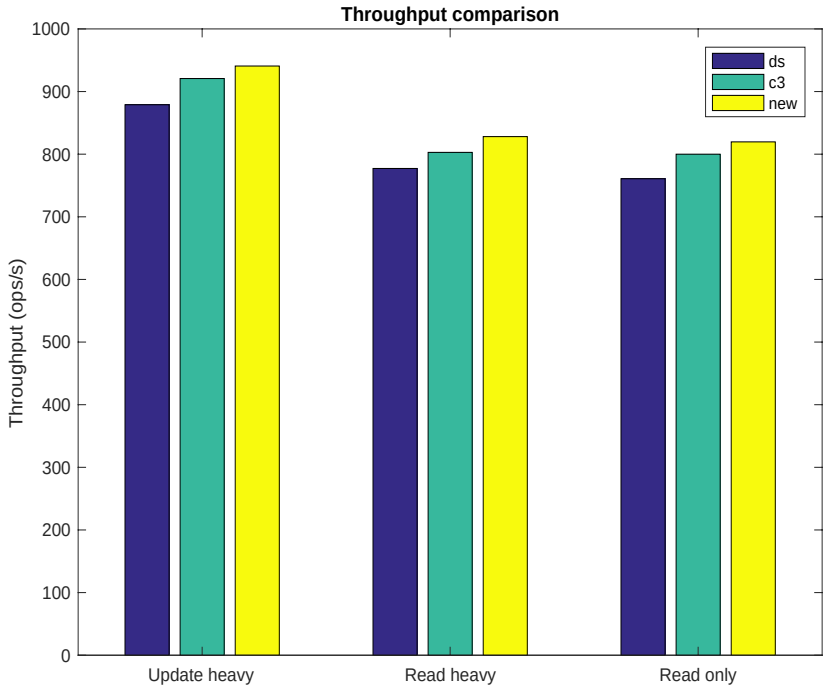

Figure 5.2: Throughput characteristic comparison with small data set in Zipfian distribution

Firstly, we evaluate the read latency. Above are the experiment results for different algorithms, of which ds (dynamic snatch) indicates the default algorithm deployed in Cassandra, c3 is the algorithm developed in the paper [60] and new is the new algorithm proposed in this thesis.

Figure 5.1 shows the read latency comparison with different workload patterns: update-heavy, read-heavy and read-only, same as table 4.1. We can see that for all workload used, the new algorithm reduces the read latency with all different metrics: the average, the 95th, the 99th and the 99.9th percentile latency. Taking the 99th percentile latency metric as an example, for the read only workload, the new algorithm reduces the read latency from 4.5 ms to 3.4 ms, which is about 24% reduction. Even compared with the C3 algorithm, the read latency is reduced by 16%.

Because the data size in the experiments is small, the data might be cached in the memory, and as a result, the performance improvement for the new algorithm is not that significant.

Then we consider the throughput achieved by different algorithms. The throughput indicates the number of requests processed in a second. The higher the throughput is, the better the algorithm performs. The comparison is shown in Figure 5.2. The throughput goes up to 940 ops per second in update heavy, which is nearly 14% improvement over the ds algorithm and 6% improvement over the c3 algorithm. Overall, the throughput performance has been improved, with the usage of the new algorithm.

5.2.2 Impact of request distribution

In this section, we compare the performance with different request distributions: Zipfian and Uniform with a large data set size - 4 GB.

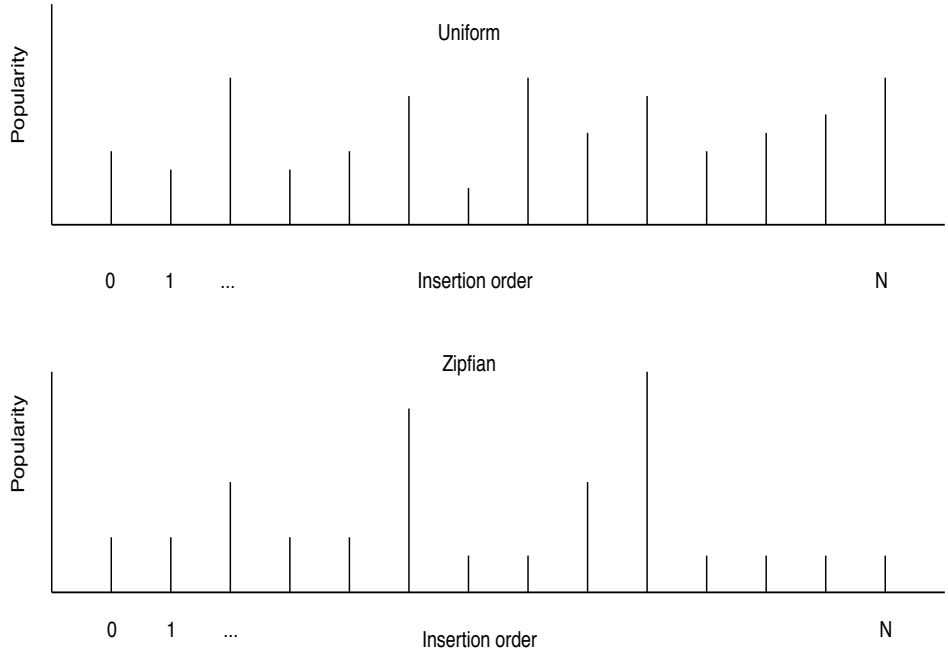

Figure 5.3: Selection results of Uniform and Zipfian [28]

With the Uniform distribution, a record is chosen uniformly at random. Thus, all records in the database has an equal chance to be chosen.

For the Zipfian distribution, when choosing a record, some records have more chance to be chosen. This distribution generates more unexpected situations. Figure 5.3 shows the different results of these two distributions.

Firstly, we compare the read latency with the Zipfian distribution. Figure 5.4 shows the read performance comparison. The figure shows that for the large data set, the read performance varies greatly. That's because the reading operations need to read data from the disk.

However, with our new algorithm, the latency is reduced on all metrics.

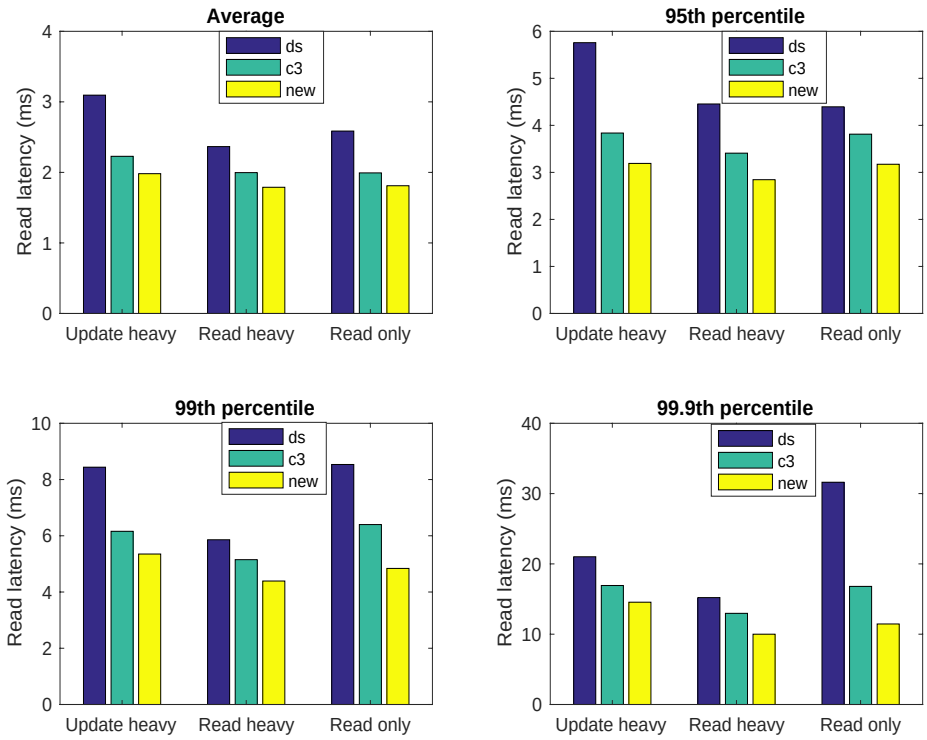

Figure 5.4: Read latency comparison with large data set in Zipfian distribution

For example, for the read-only operation, the 99.9% percentile latency for default

dynamic snitch method goes up to 31.61 ms, which is nearly 10x compared with the average latency. The c3 algorithm reduces the latency to be nearly half with 16.79 ms. Our algorithm, further, reduces it to be 11.45 ms, which is another 32% percent reduction for read latency.

By indicating the busy node and distributing the load among multiple replicas, the new algorithm takes full advantage of available system resources, resulting in an increase in throughput, for all workload patterns. Even in the experiment with mixed read/write requests, it still improves the throughput. The reason lies in that the new algorithm distributes the read requests among multiple replicas more evenly and avoids the busy ones. Then this mechanism indirectly keeps the write requests balanced among multiple replicas.

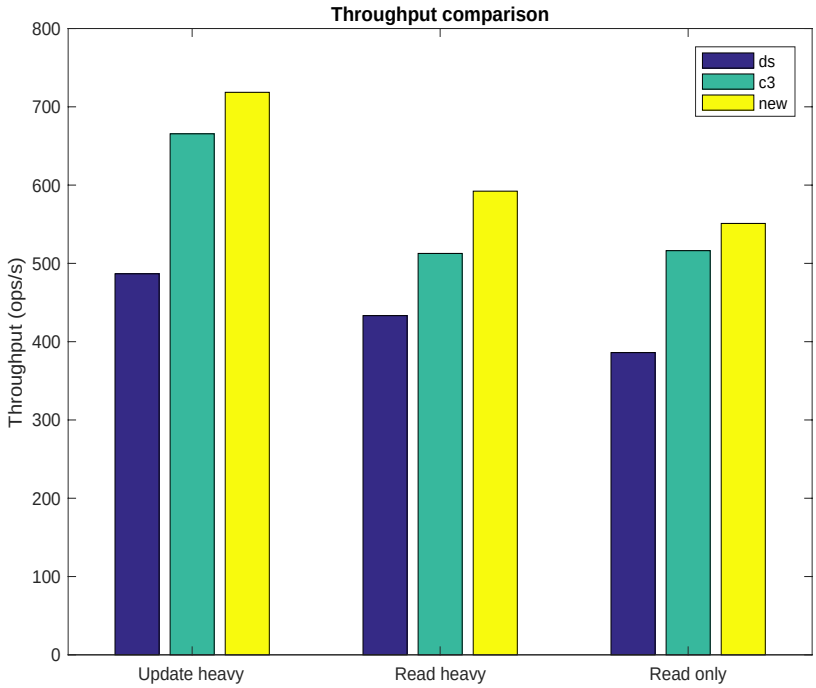

Figure 5.5: Throughput characteristic comparison with large data set in Zipfian distribution

Figure 5.5 shows the throughput performance comparison. From the figure we can see that for the update heavy operation, c3 improves the throughput by 32%, while our algorithm further improves another 7%. For the read heavy operation, c3 improves the throughput by 18%, while our algorithm further improves it for another 16%.

Meanwhile, we compare the performance with the Uniform distribution requests. Figure 5.6 shows the read latency comparison result for the Uniform distribution.

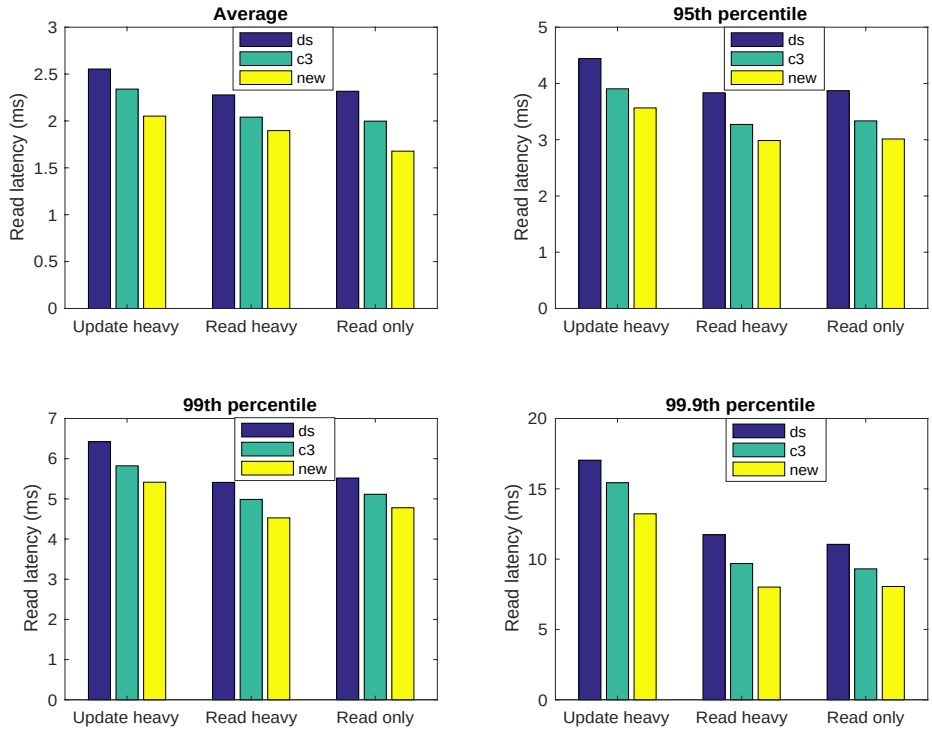

Figure 5.6: Read latency comparison with large data set in Uniform distribution

From the figure we can see that the new algorithm improves the latency for all the percentile metrics. For example, for the 99.9th percentile latency, with the update-heavy workload, the latency for ds is around 17ms, while our new algorithm reduces it to 13 ms, which is nearly 23% improvement.

At the same time, the throughput performance has been compared as well for Uniform distribution requests. Figure 5.7 shows the throughput comparison result. With the more efficient replica selection algorithm, the throughput is improved. Taking read only workload as an example, the throughput is increased from 420 ops/s to nearly 500 ops/s, which is about 19% improvement.

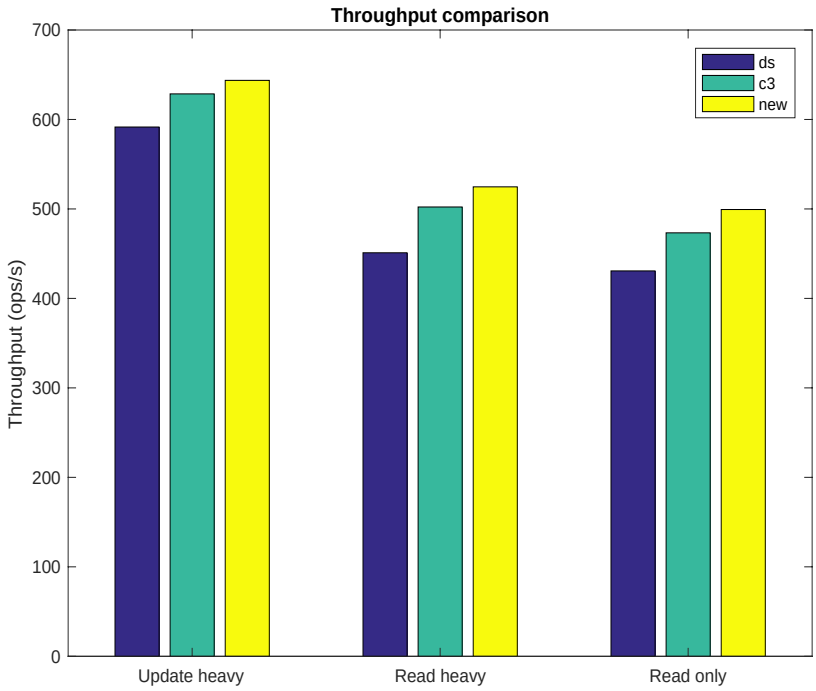

Figure 5.7: Throughput comparison with large data set in Uniform distribution

Figure 5.8 shows the write latency comparison results for the update heavy and read heavy operations with the Zipfian distribution requests.

From the results, we can see that our algorithm also improves the write latency performance, especially the update heavy operation, which includes half write operations and half read operations. For example, for the update heavy workload, the 99.9th percentile latency is reduced from 15.7 ms to 10 ms, which is nearly 36% improvement.

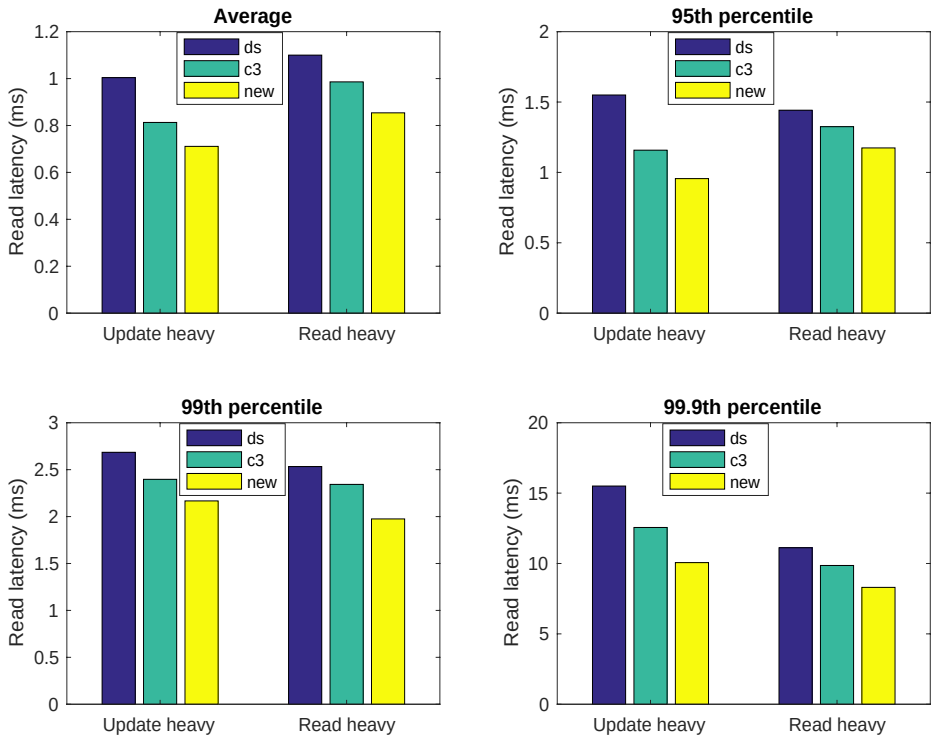

Figure 5.8: Write latency comparison with large data set in Zipfian distribution

Figure 5.9 shows the write latency comparison results for the update heavy and read heavy operations with the Uniform distribution requests. The average latency and 95% percentile latency have very similar results due to the uniform distribution.

However, for 99% percentile and 99.9% percentile latency, we can see that the write latency performance is improved.

5.2.3 Impact of skewed record size

So far, we only consider the fixed-length records. In realistic, the data size could be variables. Since we calculate the score based on per-request in our algorithm, we need to make sure that the varied data size won't affect the performance too much. In this group

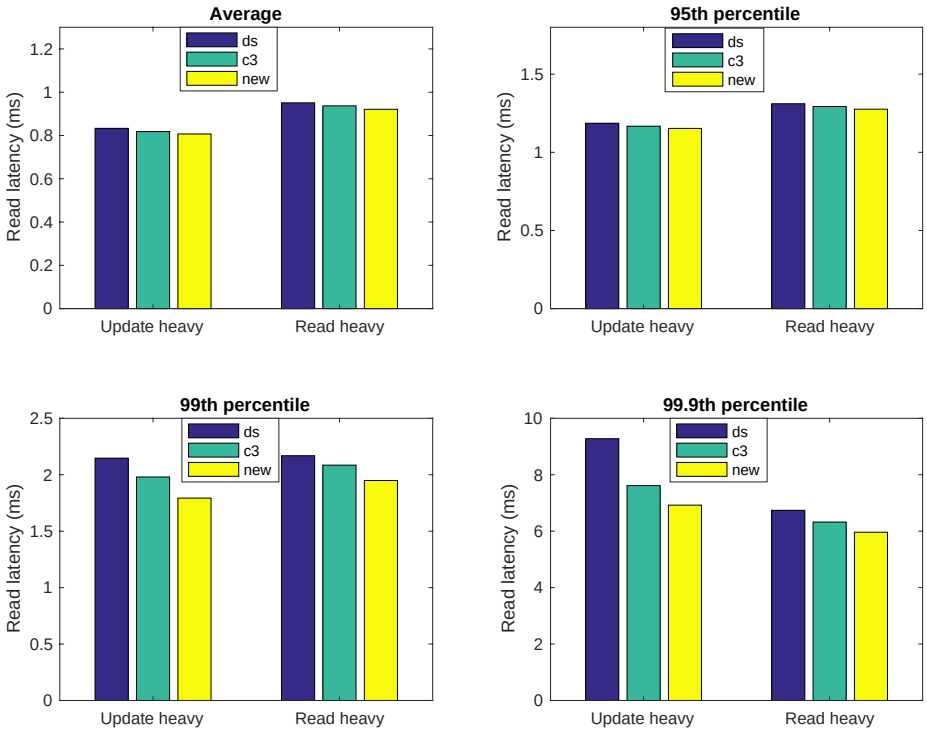

Figure 5.9: Write latency comparison with large data set in Uniform distribution

of experiments, we use YCSB to generate the same size of large dataset. In the dataset, the field size follows Zipfian distribution, which favors the shorter values.

Figure 5.10 shows the read latency for the large data set with the Zipfian distribution workload. The figure shows that even with the skewed dataset size, our new algorithm could improve the latency performance with different percentile values, especially the 99.9% percentile latency. For the read only workload, the 99.9% percentile latency in the traditional DS algorithm goes up to nearly 36 ms. Our new algorithm reduces the latency to be about 15 ms, which is about 2.4 times improvement.

For the skewed dataset size, its uneven characteristic makes the latency more unstable, which indirectly increases the chance of high tail latency. Since the new algorithm makes the metrics more smooth, it reduces the chance of anomalies values. The experiment

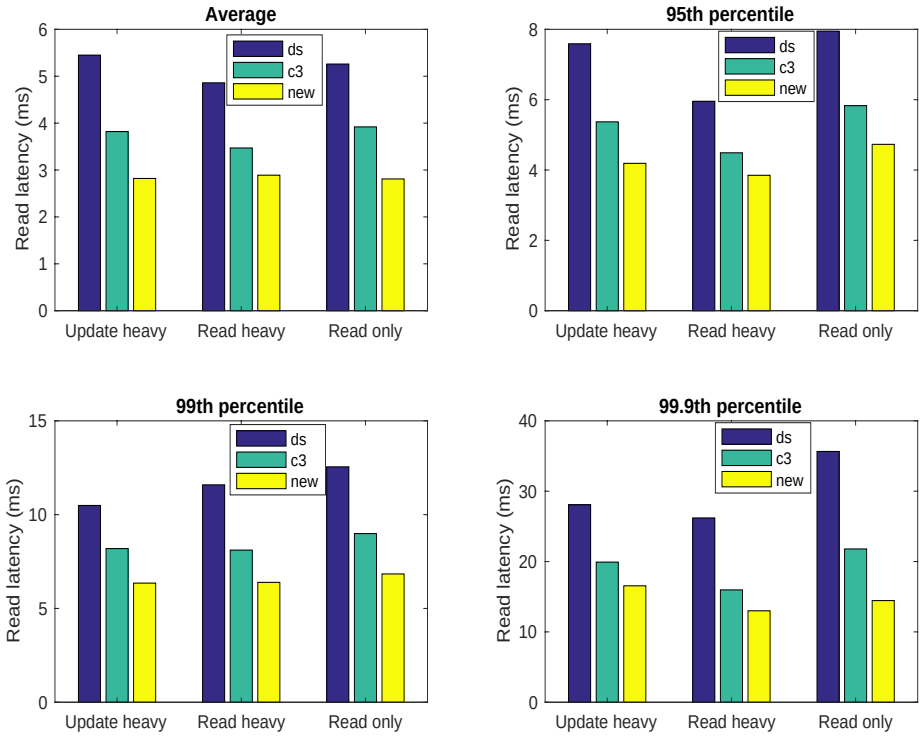

Figure 5.10: Read latency comparison with skewed data size

results also shows that our new algorithm performs well for this case.

5.2.4 Impact of workload on load condition

Furthermore, since we've found the load oscillation issue is severe with the traditional ds algorithm, this further causes the load spikes, which has been shown in the top figure of Figure 5.11. The experiment is the result for the large data set with the Zipfian distribution workload. The y-axis is the request number served per 100 ms for one of the nodes.

Sometimes, the request number served in 100 ms is only 1. This is because the node is very busy and cannot serve any request in the 100 ms time duration.

Through our new algorithm, the load is distributed more evenly among multiple

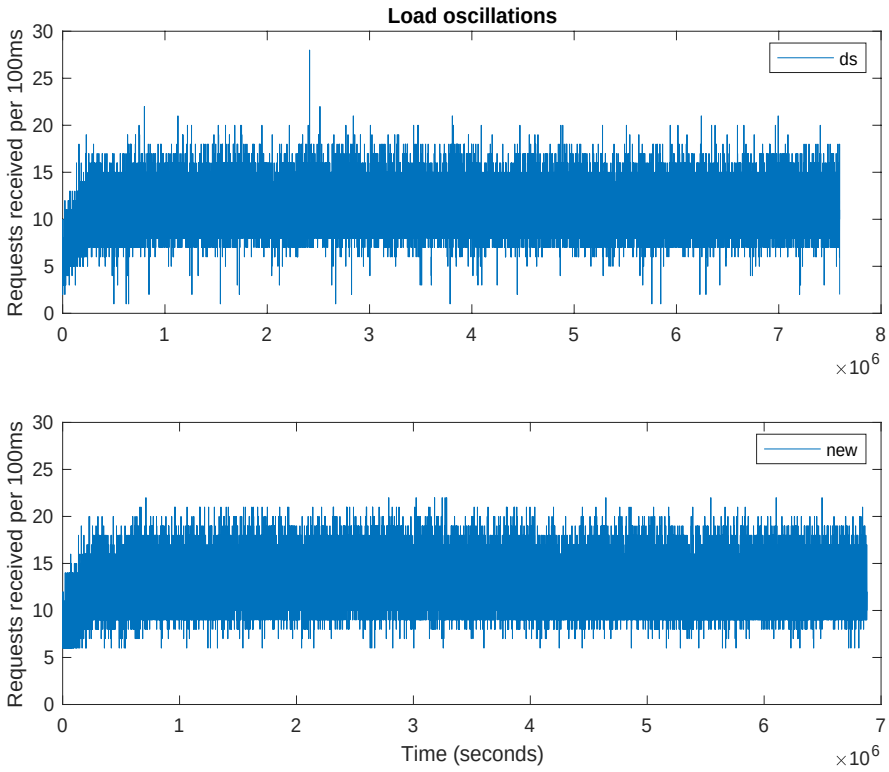

Figure 5.11: Load versus time

nodes. This reduces the number of load spikes, as shown in the bottom figure of Figure 5.11. The result shows that the traditional ds algorithm could only serve 10 requests in average, while the new algorithm could serve up to 13 requests in average per 100 ms. This is about 30% improvement in the load performance.

Also, from the figure we can see, with the new algorithm the node could serve at least about 5-6 requests every 100 ms. Meanwhile, from the x-axis we can also see that the running time is reduced as well.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This thesis measures all the important factors on the tail latency in Apache Cassandra, especially the queue size, service time, response time, and the waiting time in the queue. The distributions of these parameters are fully explored. Specifically, the results show that the service time varies a lot during the whole process, which inspires the deeper research on the background activities, including the Garbage Collection and data compaction. Then a new algorithm is proposed to get the real time statistics of the service time and this information is then piggy-backed to the client. Then based on the information collected, the client avoids selecting busy nodes and sends the request to less-busy nodes.

Local read request are processed at the coordinator node with remote read requests from non-coordinator node. This leads to unnecessary waiting time, queuing time and the stages switching time. Then, the internal process for local read requests is modified to avoid this unnecessary latency in the thread pool queue. This could also help to reduce the latency for each request and improve the throughput for the whole cluster.

Through comprehensive evaluation, the final results show that it greatly improves the

latency performance, including the median, and the percentile tail latency, compared with the base-line Cassandra. Meanwhile, as the algorithm distributes the request more evenly, it also works well with mixed requests, like read-heavy and update-heavy workloads.

6.2 Future Work

There are multiple different aspects which could be improved in our algorithm for future work.

Firstly, Cassandra nodes compute the score for each node in a fixed, discrete interval, which is 100ms in default. Hence, the system may not react to the time-varying performance fluctuations, which may happen when a time scale is less than the fixed interval. Though one may think about is to reduce the interval to re-calculate the score for each node. However, it has been stated that the calculation could be very expensive. So a more complex design should be proposed to consider this offset.

Secondly, Cassandra has its own mechanism called speculative retries to reduce the tail latency. When sending out a read request to a replica, the coordinator waits for the response for a configurable duration. After this, it'll reissue the request to another replica. As the duration is configurable, if it's set to be small, then too many reissued requests could make the system saturated. If it's set to be large, then the effect of this mechanism may not be obvious. So a better design on this duration and the integration with our own algorithm could be further explored.

Meanwhile, the selection of the coordinator node could also be explored, as the coordinator node is working as the proxy between the client and the whole cluster. How to choose the right coordinator is also a big issue. Right now, companies like Datastax and Netflix have their own drivers to Cassandra. For example, in the Datastax Java driver for Cassandra, they implement different policies, such as DCAwareRoundRobinPolicy,

LatencyAwarePolicy, TokenAwarePolicy and WhiteListPolicy [7]. Among all these polices, the LatencyAwarePolicy chooses the coordinator with the least latency. As in Cassandra, the coordinator is chosen randomly right now, then the integration of the previous policy should be of great interest for research.

Lastly, in our thesis, the replica selection only chooses one replica from multiple replicas, which is called eventual consistency [47]. However, to make data more reliable, stronger consistency also needs to be explored, which means to choose more than one replicas. This makes the design more challenging and more complex, and it may require more synchronization among all these replicas.

Bibliography

[1] Apache cassandra. http://cassandra.apache.org/. Accessed: 2015-03-10. 1.1

[2] Apache cassandra use cases. http://www.datastax.com/resources/casestudies. Accessed: 2015-05-03. 1.1

[3] Apache hbase. https://hbase.apache.org/. Accessed: 2015-03-30. 1.1

[4] Bloom filter. https://en.wikipedia.org/wiki/Bloom_filter. Accessed: 2015-04-15. 2.3

[5] Cap theorem. https://en.wikipedia.org/wiki/CAP_theorem. Accessed: 2015-03-10. 2.1

[6] Datastax. http://www.datastax.com/. Accessed: 2015-08-10. 5.1

[7] Datastax load balancing. http://docs.datastax.com/en/developer/java-driver/3.1/manual/load_balancing/. Accessed: 2016-05-15. 6.2

[8] Exponentially weighted moving average. https://en.wikipedia.org/wiki/Moving_average. Accessed: 2016-03-03. 4.3.1

[9] Internal read in cassandra. https://pandaforme.gitbooks.io/introduction-to-cassandra/content/how_is_data_read.html. Accessed: 2015-07-16. (document), 2.3, 2.4, 2.4

[10] Internal write in cassandra. https://teddyma.gitbooks.io/learncassandra/content/model/where_is_data_stored.html. Accessed: 2015-07-15. (document), 2.2, 2.2

[11] Mongodb. https://www.mongodb.com/. Accessed: 2015-03-30. 1.1

[12] Mysql. https://www.mysql.com/. Accessed: 2015-03-30. 1.1

[13] Oracle. https://www.oracle.com/database/index.html. Accessed: 2015-03-30. 1.1

[14] Read request stage flow. https://www.pythian.com/blog/guide-to-cassandra-thread-pools/. Accessed: 2016-02-10. 2.4

[15] Stages in cassandra. https://wiki.apache.org/cassandra/ArchitectureInternals. Accessed: 2015-08-03. (document), 2.4, 2.5

[16] Write path in cassandra. https://docs.datastax.com/en/cassandra/2.1/cassandra/dml/architectureClientRequestsWrite.html. Accessed: 2016-03-10. (document), 2.1, 2.3

[17] Ycsb. https://github.com/brianfrankcooper/YCSB/wiki. Accessed: 2015-07-07. 5.1, 5.1

[18] Daniel Abadi. Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. *Computer*, 45(2):37–42, 2012. 2.1

[19] Leonardo Aniello, Silvia Bonomi, Marta Breno, and Roberto Baldoni. Assessing data availability of cassandra in the presence of non-accurate membership. In *Proceedings of the 2nd International Workshop on Dependability Issues in Cloud Computing*, page 2. ACM, 2013. 2.1

[20] Takao Bakuya and Masato Matsui. Relational database management system, October 21 1997. US Patent 5,680,614. 1.1

[21] Carlos F Bispo. The single-server scheduling problem with convex costs. *Queueing Systems*, 73(3):261–294, 2013. 4.3.1

[22] Jake Brutlag. Speed matters for google web search, 2009. 1.1

[23] Apache Cassandra. Apache cassandra. *Website. Available online at http://planetcassandra.org/what-is-apache-cassandra*, page 13, 2014. 1.1

[24] Maria Chalkiadaki and Kostas Magoutis. Managing service performance in the cassandra distributed storage system. In *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on*, volume 1, pages 64–71. IEEE, 2013. 3.2

[25] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008. 2.1

[26] Min Chen, Shiwen Mao, and Yunhao Liu. Big data: A survey. *Mobile Networks and Applications*, 19(2):171–209, 2014. 1.1

[27] Wesley Chow and Ning Tan. Investigation of techniques to model and reduce latencies in partial quorum systems. 2012. 3.2

[28] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010. (document), 4.1, 4.1, 5.1, 5.3

[29] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, February 2013. 1.1, 1.2, 3.1, 4.1

[30] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. *ACM SIGOPS operating systems review*, 41(6):205–220, 2007. 2.1

[31] Satoshi Fukuda, Ryota Kawashima, Shoichi Saito, and Hiroshi Matsuo. Improving response time for cassandra with query scheduling. In *Computing and Networking (CANDAR), 2013 First International Symposium on*, pages 128–133. IEEE, 2013. 3.2

[32] Hakan Hacigumus, Bala Iyer, and Sharad Mehrotra. Providing database as a service. In *Data Engineering, 2002. Proceedings. 18th International Conference on*, pages 29–38. IEEE, 2002. 1.1

[33] Jing Han, E Haihong, Guan Le, and Jian Du. Survey on nosql database. In *Pervasive computing and applications (ICPCA), 2011 6th international conference on*, pages 363–366. IEEE, 2011. 1.1

[34] Md E Haque, Yuxiong He, Sameh Elnikety, Ricardo Bianchini, Kathryn S McKinley, et al. Few-to-many: Incremental parallelism for reducing tail latency in interactive services. *ACM SIGPLAN Notices*, 50(4):161–175, 2015. 3.1

[35] Xiangdong Huang, Jianmin Wang, Jian Bai, Guiguang Ding, and Mingsheng Long. Inherent replica inconsistency in cassandra. In *Big Data (BigData Congress), 2014 IEEE International Congress on*, pages 740–747. IEEE, 2014. 3.2

[36] Virajith Jalaparti, Peter Bodik, Srikanth Kandula, Ishai Menache, Mikhail Rybalkin, and Chenyu Yan. Speeding up distributed request-response workflows. In *ACM*

SIGCOMM Computer Communication Review, volume 43, pages 219–230. ACM, 2013. 1.2, 3.1

[37] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M Voelker, and Amin Vahdat. Chronos: predictable low latency for data center applications. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 9. ACM, 2012. 3.1

[38] Shahidul Islam Khan and ASML Hoque. A new technique for database fragmentation in distributed systems. *International Journal of Computer Applications*, 5(9):20–24, 2010. 1.1

[39] Saehoon Kim, Yuxiong He, Seung-won Hwang, Sameh Elnikety, and Seungjin Choi. Delayed-dynamic-selective (dds) prediction for reducing extreme tail latency in web search. In *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining*, pages 7–16. ACM, 2015. 1.1

[40] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010. 1.2

[41] Jialin Li, Naveen Kr Sharma, Dan RK Ports, and Steven D Gribble. Tales of the tail: Hardware, os, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14. ACM, 2014. 3.1

[42] Emil C Lupu and Morris Sloman. Conflicts in policy-based distributed systems management. *IEEE Transactions on software engineering*, 25(6):852–869, 1999. 1.2

[43] Martin Maas, Tim Harris, Krste Asanovic, and John Kubiatowicz. Trash day: Coordinating garbage collection in distributed systems. In *HotOS*, 2015. 1.2

[44] James Manyika, Michael Chui, Brad Brown, Jacques Bughin, Richard Dobbs, Charles Roxburgh, and Angela H Byers. Big data: The next frontier for innovation, competition, and productivity. 2011. 1.1

[45] Adam Marcus. The nosql ecosystem. *The Architecture of Open Source Applications*, pages 185–205, 2011. 1.1

[46] Vivek Mishra. *Instant Apache Cassandra for Developers Starter*. Packt Publishing Ltd, 2013. 2.1

[47] Vivek Mishra. *Beginning Apache Cassandra Development*. Apress, 2014. 6.2

[48] Michael Mitzenmacher. How useful is old information? *IEEE Transactions on Parallel and Distributed Systems*, 11(1):6–20, 2000. 4.1

[49] Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001. 1.3

[50] Shunsuke Nakamura and Kazuyuki Shudo. Mycassandra: A cloud storage supporting both read heavy and write heavy workloads. In *Proceedings of the 5th Annual International Systems and Storage Conference*, page 14. ACM, 2012. 3.2

[51] Nishant Neeraj. *Mastering Apache Cassandra*. Packt Publishing Ltd, 2013. 2.1

[52] Nitin Padalia. *Apache Cassandra Essentials*. Packt Publishing Ltd, 2015. 2.1

[53] Rabi Prasad Padhy, Manas Ranjan Patra, and Suresh Chandra Satapathy. Rdbms to nosql: reviewing some next-generation non-relational databases. *International Journal of Advanced Engineering Science and Technologies*, 11(1):15–30, 2011. 2.1

[54] Jaroslav Pokorny. Nosql databases: a step to database scalability in web environment. *International Journal of Web Information Systems*, 9(1):69–82, 2013. 1.1

[55] Jordà Polo, Yolanda Becerra, David Carrera, Jordi Torres, Eduard Ayguadé, Mike Spreitzer, and Malgorzata Steinder. Enabling distributed key-value stores with low latency-impact snapshot support. In *Network Computing and Applications (NCA), 2013 12th IEEE International Symposium on*, pages 65–72. IEEE, 2013. 3.2

[56] Robert F Stärk, Joachim Schmid, and Egon Börger. *Java and the Java virtual machine: definition, verification, validation*. Springer Science & Business Media, 2012. 1.2

[57] Michael Stonebraker. Sql databases v. nosql databases. *Communications of the ACM*, 53(4):10–11, 2010. 1.1

[58] Christof Strauch, Ultra-Large Scale Sites, and Walter Kriha. Nosql databases. *Lecture Notes, Stuttgart Media University*, 2011. 1.1

[59] Robbie Strickland. *Cassandra high availability*. Packt Publishing Ltd, 2014. 1.1

[60] P Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *NSDI*, pages 513–527, 2015. 3.2, 4.2.2, 4.3.1, 5.2, 5.2.1

[61] Isuru Suriarachchi. A survey on seda and using seda to optimize hdfs read operation. 1.2

[62] Clarence JM Tauro, Baswanth Rao Patil, and KR Prashanth. A comparative analysis of different nosql databases on data model, query model and replication model. In *Proceedings of the International Conference on Emerging Research in Computing, Information, Communication and Applications ERCICA 2013, ISBN*, volume 1120603436, 2013. 1.1

[63] Eno Thereska, Brandon Salmon, John Strunk, Matthew Wachs, Michael Abd-El-Malek, Julio Lopez, and Gregory R Ganger. Stardust: tracking activity in a distributed

storage system. In *ACM SIGMETRICS Performance Evaluation Review*, volume 34, pages 3–14. ACM, 2006. 1.2

[64] Ashish Vulimiri, Philip Brighten Godfrey, Radhika Mittal, Justine Sherry, Sylvia Ratnasamy, and Scott Shenker. Low latency via redundancy. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, pages 283–294. ACM, 2013. 3.1

[65] Ashish Vulimiri, Oliver Michel, P Godfrey, and Scott Shenker. More is less: reducing latency via redundancy. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, pages 13–18. ACM, 2012. 1.2

[66] Guoxi Wang and Jianfeng Tang. The nosql principles and basic application of cassandra model. In *Computer Science & Service System (CSSS), 2012 International Conference on*, pages 1332–1335. IEEE, 2012. 2.1

[67] Matt Welsh, David Culler, and Eric Brewer. Seda: an architecture for well-conditioned, scalable internet services. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 230–243. ACM, 2001. 1.3, 2.4

[68] Yunjing Xu, Zachary Musgrave, Brian Noble, and Michael Bailey. Bobtail: Avoiding long tails in the cloud. In *NSDI*, volume 13, pages 329–342, 2013. 3.1

[69] Jeong-Min Yun, Yuxiong He, Sameh Elnikety, and Shaolei Ren. Optimal aggregation policy for reducing tail latency of web search. In *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 63–72. ACM, 2015. 1.2

[70] Timothy Zhu, Alexey Tumanov, Michael A Kozuch, Mor Harchol-Balter, and Gregory R Ganger. Prioritymeister: Tail latency qos for shared networked storage. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14. ACM, 2014. 3.1