On the Importance of Infrastructure-Awareness in Large-Scale Distributed Storage Systems

by

Sajjad Rizvi

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2021

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	Eyal de Lara Professor, Department of Computer Science, University of Toronto
Supervisor(s):	Bernard Wong Associate Professor, Cheriton School of Computer Science, University of Waterloo
	Srinivasan Keshav Professor, Cheriton School of Computer Science, University of Waterloo
Internal Member:	Khuzaima Daudjee Research Associate Professor, Cheriton School of Computer Science, University of Waterloo
Internal Member:	Samer Al-Kiswany Assistant Professor, Cheriton School of Computer Science, University of Waterloo
Internal-External Member:	Wojciech Golab Associate Professor, Department of Electrical and Computer Engineering, University of Waterloo

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

This dissertation includes first-authored and peer-reviewed materials that have appeared in conference proceedings published by the Institute of Electrical and Electronics Engineers (IEEE) and Association for Computing Machinery (ACM).

ACM's policy on the reuse of published materials in a dissertation is as follows:

> *"Authors can include partial or complete papers of their own (and no fee is expected) in a dissertation as long as citations and DOI pointers to the Versions of Record in the ACM Digital Library are included."*

The following list serves as a declaration of the Versions of Record for works included in this dissertation:

Portions of Chapter 1, 2, and 3:

Sajjad Rizvi, Xi Li, Bernard Wong, Fiodar Kazhamiaka, Benjamin Cassell. Mayflower: Improving distributed filesystem performance through SDN/filesystem co-design. In Proceedings of the 2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS), 384-394, Nara, Japan, 2016. DOI: 10.1109/ICDCS.2016.39

Portions of Chapter 1, 2, and 4:

Sajjad Rizvi, Bernard Wong, and Srinivasan Keshav. Canopus: A scalable and massively parallel consensus protocol. In Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies (CoNEXT), 426-438, Incheon, Republic of Korea, 2017. DOI: 10.1145/3143361.3143394

Intellectual property rights of the system presented in Chapter 4 are protected with U.S. Patent 10,848,549, issued on November 24, 2020.

Abstract

Big data applications put significant latency and throughput demands on distributed storage systems. Meeting these demands requires storage systems to use a significant amount of infrastructure resources, such as network capacity and storage devices. Resource demands largely depend on the workloads and can vary significantly over time. Moreover, demand hotspots can move rapidly between different infrastructure locations.

Existing storage systems are largely infrastructure-oblivious as they are designed to support a broad range of hardware and deployment scenarios. Most only use basic configuration information about the infrastructure to make important placement and routing decisions. In the case of cloud-based storage systems, cloud services have their own infrastructure-specific limitations, such as minimum request sizes and maximum number of concurrent requests. By ignoring infrastructure-specific details, these storage systems are unable to react to resource demand changes and may have additional inefficiencies from performing redundant network operations. As a result, provisioning enough resources for these systems to address all possible workloads and scenarios would be cost prohibitive.

This thesis studies the performance problems in commonly used distributed storage systems and introduces novel infrastructure-aware design methods to improve their performance. First, it addresses the problem of slow reads due to network congestion that is induced by disjoint replica and path selection. Selecting a read replica separately from the network path can perform poorly if all paths to the pre-selected endpoints are congested. Second, this thesis looks at scalability limitations of consensus protocols that are commonly used in geo-distributed key value stores and distributed ledgers. Due to their network-oblivious designs, existing protocols redundantly communicate over highly oversubscribed WAN links, which poorly utilize network resources and limits consistent replication at large scale. Finally, this thesis addresses the need for a cloud-specific realtime storage system for capital market use cases. Public cloud infrastructures provide feature-rich and cost-effective storage services. However, existing realtime timeseries databases are not built to take advantage of cloud storage services. Therefore, they do not effectively utilize cloud services to provide high performance while minimizing deployment cost.

This thesis presents three systems that address these problems by using infrastructure-aware design methods. Our performance evaluation of these systems shows that infrastructure-aware design is highly effective in improving the performance of large scale distributed storage systems.

Acknowledgements

First and foremost, *"All praise be to Allah, the praise that is equal to the praise of His favourite angels and His prophets. May Allah send blessings to His select, Muhammad, the seal of prophets, and upon his household, the pure and immaculate"*, (an excerpt from the supplication of the day of A'rafah).

The last few years have been a life-changing and enlightening experience. These years have taught me lessons, and have shown me the aspects of my self and my life that I doubt I would have gained without such experience. In all these years, my supervisors have been a major support and guide. I am thankful to my supervisors, Prof. Bernard Wong and Prof. Srinivasan Keshav, for all their efforts, guidance, and support in achieving my goals.

My wife, daughters, and my parents have been a tremendous support during my stay at Waterloo. They have been a part of all of my achievements and a source of contentment and joy. I also acknowledge the love and support of other family members, who have always been with me in all matters. I would also like to thank my friends, especially Taha Taqvi, Maysum Panju, and Ali Molavi.

I have benefited from numerous people during these years. I am thankful for the members of our research group. The group discussions and feedback from our meetings have helped greatly in improving my work. I am especially thankful to Xinan for his thought provoking discussions. Research community benefits greatly from anonymous reviewers who review research papers and provide valuable insights and feedback. I am thankful to many unknown reviewers who have reviewed my research papers and provided valuable feedback. Also, I am thankful to the management and staff of the University who work hard to provide excellent working conditions and all means for students to succeed, leading them to become a strong pillar to support human society. I also acknowledge the team at Smash.BI Inc. who have provided valuable insights and directions in one of my projects. I also acknowledge the funding agencies who have provided funding for my studies and for the equipment that I have used during my research.

Dedication

To the promised one who

"will fill the earth with justice and equity as it would be filled with tyranny and oppression".

Table of Contents

Chapter 1

Introduction

Big data analytics is pervasive in the modern world where nearly all tools and technologies collectively produce large volumes of data. Organizations use big data applications to collect and analyze data, which reveals insights that are critical for business development and innovation. Processing large volumes of data introduces many challenges such as efficiently storing and managing data across many machines. Big data applications offload data storage and management to scalable *distributed storage systems*, which address the problems of storing and managing large volumes of data.

A distributed storage system typically consists of a cluster of networked machines that work together to provide a unified storage space to applications. Spreading data across machines introduces many challenges such as efficiently utilizing storage capacity, maintaining data consistency and durability despite failures, and providing high scalability. A distributed storage system masks all these challenges from applications, which simplifies applications' design and enable them to evolve without handling storage related problems. Thus, distributed storage systems go hand-in-hand with data processing applications and are a fundamental part of the big data eco-system.

Large scale storage and processing requires provisioning a large infrastructure of servers, storage devices, and a high-speed network. For efficient provisioning and management of a vast infrastructure, big data applications and storage systems are deployed in large datacenters. A typical deployment consists of a mix of storage systems and data processing applications. Multiple storage systems are employed to store different types of data. For example, the data warehousing and analytics infrastructure at Facebook uses a message brokering system for log data, an SQL database for advertisement details, and a distributed filesystem for aggregated data [58]. Similarly, a stream processing pipeline may use a

combination of a message queue, a SQL database, and a distributed filesystem to manage historical data. In addition to storage systems, a diverse set of applications are deployed in datacenters that use the storage systems. For instance, batch processing is performed through Hadoop MapReduce, Hive is used for SQL queries, and Spark streaming can be used to process streaming data.

Different applications and storage systems in a datacenter strive to achieve different performance goals while sharing infrastructure resources. Moreover, they have different workload characteristics and demand different resources from the infrastructure. Some workloads are data intensive and demand high network bandwidth whereas others are latency sensitive and demand high compute resources. Due to sharing infrastructure resources, often applications and storage systems compete for shared resources and stress parts of the infrastructure, which results in resource limitations and lower performance of storage systems. For example, competing for limited network bandwidth can result in congestion in parts of the network, resulting in lower performance of read/write jobs [272, 59]. Therefore, performance of storage systems highly depend on the availability of infrastructure resources and utilizing the available resources effectively.

One way to avoid low performance of storage systems due to resource limitations is to provision enough resources that address the requirements of all applications and storage systems. However, workload characteristics shift over time and demand hotspots keep changing their location within the infrastructure. For example, a storage system for an e-commerce application may have high resource demands only during shopping seasons, or a data item may become popular that shift the location of resource demands within the infrastructure. Addressing resource requirements of all applications and workloads would require significantly over-provisioning resources and maintaining them, which is cost-prohibitive. On the contrary, many infrastructure components, such as the network [92, 29, 108], are commonly oversubscribed to remain cost-effective and increase resource utilization. Therefore, efficiently utilizing available resources is significantly important to achieve high performance in distributed storage systems.

1.1 Designing Infrastructure-Aware Storage Systems

An alternative approach to address competing resource demands is to efficiently utilize available resources. Most storage systems do not use infrastructure resources efficiently as they are designed oblivious to the infrastructure. Without using rich information from infrastructure, a storage system may introduce inefficiencies such as utilizing excessive network bandwidth due to redundant communication over same links. Moreover, an

infrastructure-oblivious system cannot recognize the availability of alternative resources and make use of them, e.g., not taking advantage of multiple network paths between endpoints. In the case of cloud-based storage systems, cloud services have their own infrastructure-specific limitations such as requests throttling and sizes limitations. Therefore, an infrastructure-oblivious storage system would inadequately uses cloud infrastructure when deployed in a cloud.

On the contrary, an infrastructure-aware storage system can use rich information from the infrastructure to make decisions that efficiently use available resources. Infrastructure-awareness can manifest in many different ways such as designing around the underlying network topology, utilizing current network state information, and exploiting the redundancy in the network. In the case of cloud based deployments, applications can take advantage of specific properties of cloud services, e.g., their scalability and fault tolerance, to optimize the storage systems that leverage cloud services for durable storage. Therefore, introducing infrastructure-awareness in storage systems can be instrumental in efficiently utilizing available resources, resulting in higher performance of the storage systems.

As network is often the performance bottleneck for distributed storage systems, many storage systems are designed to be network-aware. Such systems include several distributed filesystems, e.g., GFS and HDFS, and specialized hardware based key-value stores, such as Nessie [49], FaRM [86], and HERD [139]. Network-aware storage systems mostly utilize the locality information of storage nodes in racks, estimate network bandwidth assuming hierarchical topology, and exploit high-speed network interconnect. However, we argue that these are only a few obvious methods of infrastructure-awareness, which are prevalent in existing systems. Other aspects of the infrastructure are commonly neglected in the design of storage systems. Availability of high-bandwidth network within racks, network topology (flat, star, hierarchical, or random), LAN or WAN based deployment, and the knowledge of competing workloads are a few examples of design aspects that the storage systems can use for performance improvements. For cloud infrastructure, storage systems can benefit from the knowledge of the design, features, and limitations of cloud services to improve their performance. In this thesis, we identify three areas where performance can be improved, and we use richer infrastructure-aware methods to improve the performance.

1.2 Challenges in Designing Infrastructure-Aware Systems

An infrastructure-aware system can take advantage of system performance optimizations that are not possible without using specific information from the infrastructure. However, designing infrastructure-aware systems is challenging and may limit a system in several other aspects, some of which are as follows.

Higher complexity: Infrastructure-awareness increases complexity of the system. Increased complexity can be at different levels such as obtaining information from the infrastructure, handling inaccuracies in the information, failure handling, and software design and its maintenance. Existing storage systems tradeoff performance to achieve simpler design and use the underlying infrastructure as a black-box. For example, ensuring specific network path assignments through an SDN controller increases complexity in the system due to additional components and dependencies in the system.

Reduced Accessibility: An infrastructure-aware design requires rich information from the infrastructure, which is not always available. For example, network topology information is not visible to public cloud tenants. Similarly, infrastructure-aware systems may require a significant degree of control over the infrastructure, which may not be available in specific deployment scenarios. For instance, network control plane is not accessible in shared network facilities such as shared compute clusters and public clouds. Therefore, an infrastructure-aware storage system is limited to only the deployments where required information and control are accessible.

Less generality: An infrastructure-aware system is specific and tailored to their target environment. Therefore, infrastructure-aware systems are limited to only specific deployment scenarios as compared to more general systems. Using them in different environment may require re-engineering the system. For example, rack-aware writes in HDFS, which place two replicas in the same rack [117], cannot be used when HDFS is deployed in virtual machines in a public cloud. Two HDFS nodes may be located on the same physical machine, invalidating their fault-domain assumptions, or may not be in the same rack. Therefore, HDFS is best used when deployed in a native environment instead of a cloud environment.

1.3 Contributions

This thesis aims to improve the performance of large-scale distributed storage systems and highlights the significance of using infrastructure-aware design methods in improving their performance. To that end, we develop three systems that used infrastructure-aware design to achieve high performance. We introduce these systems below.

1.3.1 Reads in Storage Systems for Batch Processing

Many data intensive applications use distributed filesystems, such as GFS [105], HDFS [253], and Ceph [287], to exchange data and share state. Therefore, distributed filesystems contribute heavily to the total network traffic in datacenters. For example, some studies show that file reads from distributed filesystems contribute 14% to 31% of overall cross-rack traffic in some datacenters [60]. Due to a large network footprint, reading files over the network becomes highly susceptible to network congestion, resulting in slow read operations.

To avoid network congestion, distributed filesystems employ several techniques such as minimizing remote reads by selecting nearest replica [253]. However, they do not effectively utilize available resources to avoid network congestion as they select replicas using only static network topology information [105, 253, 96]. Network-aware filesystems, such as Sinbad [60] and OctopusFS [138], estimate network bandwidth and select replicas that are accessible through higher bandwidth. However, these systems select only replica locations and leave path assignments for read/write jobs to independent flow schedulers, such as Hedera [10]. An independent flow scheduler can be ineffective if all paths to the pre-selected endpoints are congested. This problem can be avoided by jointly selecting replica and network path through close collaboration between filesystems and the network.

This thesis presents the Mayflower [235] filesystem, which jointly selects read replica and network path to minimize read completion times. Mayflower achieves this through a network/filesystem co-design approach in which both the filesystem and the network share information with each other to achieve their respective performance goals. Mayflower employs a *Flowserver* that runs a heuristic to select the best replica-path combination based on the collected information from the network and the filesystem clients. Flowserver, running as an SDN controller application, monitors the bandwidth utilization at the network edge and models the path bandwidth of the elephant flows in the network. It uses the network model to select the best replica and network path selection for filesystem read requests.

The main advantage of using an SDN/filesystem co-design approach is that it enables the filesystem and the network to take actions collaboratively to mutually improve their performance. For example, when performing a read operation, instead of first selecting a replica based on network information and then choosing a network path connecting the reader and the replica host, Mayflower can simultaneously evaluate all possible paths between the reader and all replicas to select the replica/path combination that minimizes average request completion time.

We present the details of Mayflower in Chapter 3. We make the following contributions in this work:

- We present the design of Mayflower filesystem, which is co-designed with SDN control plane. We present design choices and solve various challenges in co-designing a filesystem with the network. We discover in this research that accounting for the impact of the new read job on the performance of existing in-flight jobs is significant. In Mayflower, we develop a method to estimate the completion time of a new job and its impact on the completion time of the existing jobs. This is non-trivial because it requires combining the information of flows' paths, their remaining sizes, and the link utilization along the flows' paths.

- We show in this research that opportunistically reading from multiple replicas instead of a single replica can accelerate the read jobs significantly. Reading from multiple replicas in parallel can negatively impact the performance of a filesystem and the network if not done carefully. This is because reading from multiple replicas may traverse the same network paths that can reduce the overall available network bandwidth. Moreover, multiple parallel and dependent jobs increases the chances of longer completion times due to stragglers. We present in this work a method that can be used to select the replicas, their network paths, and the size of data that should be read from each replica to minimize the completion time of the file read operations.

- Finally, we implement Mayflower and evaluate its performance over an emulated datacenter network. We compare Mayflower with HDFS and several combinations of static and dynamic replica and network path selection methods. Our experimental results show that Mayflower's network/filesystem co-design approach effectively reduces the read completion times as compared to independently optimizing filesystem and network operations. Compared to Mayflower, existing systems require 1.3x the completion time for file reads using common datacenter workloads.

1.3.2 Consistent Data Replication at Wide Area

Storage systems, such as Spanner [71], BigTable [56], and Etcd [89], replicate data across tens of geo-distributed nodes for high performance and fault-tolerance. Such systems often use consensus protocols, such as Paxos [160, 161] and Raft [210], for data replication with strong consistency. However, leader-based consensus protocols do not scale to a large number of nodes. The leader becomes the bottleneck due to load concentration on the leader as compared to other nodes. Moreover, leaderless protocols, such as EPaxos [195] and Mencius [184], do not use available resources efficiently as they are designed to be oblivious to the underlying infrastructure. They send redundant messages over bandwidth constraint WAN links, which waste CPU and bandwidth resources, limiting them from scaling to a large number of geo-distributed nodes.

This thesis introduces *Canopus*, an infrastructure-aware and scalable consensus protocol for state machine replication in geo-distributed key-value stores. Canopus is built from the ground-up by following the design of modern datacenters and the networking infrastructure. Canopus takes into account three aspects of modern datacenters that enable it to scale to a large number of nodes.

First, Canopus recognizes that machines in datacenters are placed in racks, with each rack hosting tens of machines that are connected through redundant top-of-rack (ToR) switches. The ToR switches provide high bandwidth, low latency, and fault-tolerant connectivity to the nodes within the same rack. Canopus exploits the availability of high performance network within racks to perform heavy communication within racks to reduce communication across racks.

Second, Canopus takes into account the hierarchical design of datacenter networks where bandwidth decreases and latency increases with distance between nodes. Therefore, inter-datacenter communication have high cost as compared to intra-datacenter communication. Therefore, Canopus aligns its communication pattern with the network hierarchy by using an overlay that minimizes communication across datacenters at the expense of more communication within racks.

Finally, we observe that modern datacenters have built in high redundancy of power and networking infrastructure to tolerate power and network hardware failures without service disruptions. Racks are connected through redundant switches and are backed by multiple power sources to eliminate single point of failures. As a result, full-rack failures and network partitions are rare [179, 180, 254, 100]. Leveraging the robustness of power and networking infrastructure in datacenters, Canopus simplifies its design by assuming that complete rack failures and network partitions do not happen. Particularly, a simple

majority of nodes within a rack always stay alive. In the case of catastrophic failures that result in a rack failure or a network partition, the protocol does not compromise safety and halts its execution until the fault is recovered. Thus, Canopus trades-off high availability to achieve high scalability.

Canopus significantly simplifies its design based on the above mentioned information and assumptions about the infrastructure, which makes Canopus easier to implement and reason about. All the nodes in Canopus serve client requests, which distribute the load across the nodes. Canopus organizes the same-rack nodes in a group, and it achieves consensus in multiple rounds. In the first round, the nodes concurrently achieve consensus within groups. In the subsequent rounds, the nodes achieve consensus across groups.

We give the detailed design and evaluation of Canopus in Chapter 4. We make the following contributions in this work:

- We present Canopus, a scalable consensus protocol for consistent data replication in geo-distributed deployments. Canopus simplifies its design by making assumptions and using information specific to the design of modern datacenters and network infrastructure. Through Canopus, we demonstrate the effectiveness of infrastructure-aware design approach to improve the performance of large-scale consensus protocols.
- We observe that batching and pipelining like a TCP protocol are necessary to overcome the WAN latencies and support higher throughput. To achieve high throughput in geo-distributed deployments, we introduce batching and pipelining in Canopus while maintaining the safety of the protocol.
- We corroborate the effectiveness of our approach through a prototype implementation of Canopus and compare its performance against EPaxos and ZooKeeper. We evaluate the systems in two deployment models: a local deployment within a datacenter and a geo-distributed cluster across up to seven regions. Our experimental results show that Canopus can achieve more than 4 million requests with 21 geo-distributed nodes, which is more than 8x higher throughput than that of EPaxos in the same settings. Design of Canopus and its high performance show that the consensus protocols, and consequently the storage systems using them, can benefit from following the infrastructure-aware design approach.

1.3.3 Financial Data Streams Storage in a Public Cloud

In the end, we use infrastructure-aware design to improve the performance of streaming data storage systems. We focus on capital market use cases where financial applications

perform streaming data analysis in a public cloud deployment.

Capital markets produce large volumes of data at a high pace, which is distributed to companies in the form of realtime financial data streams. However, high volume and velocity of market data put significant performance demands on their target storage and processing applications. Collective data rates in market data streams from a few data sources, such as stock exchanges, can exceed tens of millions of messages per second [87, 79]. As a result, storage systems for financial data streams are required to support high data ingestion rates and high scalability. Moreover, financial applications, such as live dashboards and trades monitoring [221], require realtime access to data with low latency, which is typically less than 20 milliseconds [68]. Making it further challenging, applications also perform historical data queries [273, 259, 101] in parallel to realtime queries, which compete with realtime data ingestion and queries for infrastructure resources.

Existing storage systems commonly use Lambda [190] and Kappa [149] stream processing architectures, which use message queues, such as Apache Kafka [20] and Amazon Kinesis [15], to ingest data. However, data ingestion and retrieval through message queues add a significant delay, tens to hundreds of milliseconds [132, 232], which is highly undesirable in many financial market use cases [273]. In addition, historical data queries are supported through separate systems, which increases system complexity and system management overheads [149]. Furthermore, alternative solutions [133, 25] for timeseries data, such as Druid [290], BTrDB [18], and OpenTSDB [213], do not take advantage of the infrastructure specific to financial data streams. As a result, these systems are overly complex and consume more cloud resources than needed.

To address the storage needs of streaming data in capital market uses cases, we build *Kawkab filesystem*, which is purpose-built around cloud infrastructure and financial data streams. Kawkab uses a combination of worker nodes, running on virtual machines in a public cloud, and a cloud storage service to provide a high performance and cost-effective storage system. Worker nodes, consisting of a buffer-cache and locally attached storage, serve as the front-end of the system that perform all filesystem operations, which include data ingestion, indexing, serving realtime and historical data queries, and using cloud storage services for data persistence.

Kawkab achieves high performance by exploiting unique characteristics of financial data streams and cloud infrastructure. In contrast with existing systems, Kawkab eliminates replicating data for fault-tolerance during data ingestion for high performance. Instead, Kawkab leverages the fault-tolerance guarantees provided by market data providers and cloud storage services to avoid losing data during data ingestion. Data providers, such as OPRA [214] distribution network, publish data using redundant streams for high fault-

tolerance, and they retain data at the source for a limited duration, e.g., a trading day [215], which can be accessed through offline channels at additional costs [216, 72]. Based on such fault-tolerance measures taken by financial data providers, Kawkab simply ingests data redundantly from replicated streams, and in the case of data loss during ingestion, it retrieves lost data from the upstream providers. As a result, Kawkab significantly reduces overheads of data replication in the fast path, which results in higher ingestion rates and low data staleness. In addition, it makes Kawkab highly scalable due to cutting dependencies and interactions across Kawkab nodes.

In addition to supporting high ingestion rates and scalability, Kawkab provides low latency reads and historical queries in the same system. Historical queries, which access large amounts of data, interferes with data ingestion and realtime queries if they all are performed on the same node. Kawkab leverages the scalability of cloud storage services to reduce such impact. Applications perform historical queries on separate nodes that are not loaded with performance-critical tasks such as data ingestion and serving realtime queries. If the nodes do not have required data to perform historical queries, they load data from the cloud storage instead of other Kawkab nodes. As a result, burden of historical queries is offloaded to cloud storage and historical queries have least impact on data ingestion and realtime queries.

Finally, to support range queries based on specified time, Kawkab introduces a novel timeseries index. The index is optimized to perform faster search within recently ingested data as our target applications mostly access recent data in small time windows. Moreover, the index is designed to have no impact on data ingestion performance. Furthermore, the index is not required to be fully loaded in memory for updates or search, which enables Kawkab to to support very large files and indexes.

We make the following contributions through this work:

- We present Kawkab storage system to address storage requirements of stream processing applications in capital market use cases. Following the infrastructure-aware design approach, Kawkab eliminates data replication in the fast path to gain high ingestion rates and scalability by exploiting fault-tolerance characteristics of financial data streams. Moreover, Kawkab enables applications to read both realtime and historical data using the same system by leveraging scalability of clouds storage services.

- We develop a novel indexing scheme in Kawkab to support range queries over timeseries data. Indexing in Kawkab is tailored to enable search withing recent data with low latency at the expense of higher latency to read historical data.

- We develop a prototype of Kawkab and evaluate its data ingestion throughput against BTrDB, which is a high performance streaming data storage system. Our evaluation using micro-benchmarks show that Kawkab can ingest 16-byte messages with 1.4x to 5x higher throughput than BTrDB in similar settings. Moreover, applications can read recent data from a Kawkab worker node with minimal request processing overhead. Furthermore, historical queries do not impact the performance of data ingestion and realtime queries when historical queries are performed on a separate node.

1.4 Thesis Organization

The rest of this thesis document is organized as follows. We present background and the related studies to our research in the next chapter (§2). After that, the following three chapters present the details of Mayflower (§3), Canopus (§4), and Kawkab (§5), storage systems. Finally, Chapter 6 presents the conclusion of this research.

Chapter 2

Background and Related Work

Big data applications, such as Hadoop framework and Spark, leverage distributed storage systems to retrieve input data, share data among nodes between different stages of jobs, and store the output data. A distributed storage system provides an abstraction that hides the underlying complexities involved in the storage of large volumes of data across many nodes. Moreover, a distributed storage system involves complex distributed functionality and properties such as data replication, fault-tolerance and recovery, and storage and retrieval of data from remote nodes.

Distributed storage systems fall into many categories, such as block-based stores, object stores, and key-value stores. We focus on the storage systems that are infrastructure-aware, and we divide our discussion into three sections. First, we present the work related to network-aware distributed filesystems that have been developed for batch processing applications (§ 2.1). As we target the filesystems that are deployed within a single datacenter, we do not address geo-distributed filesystems. Next, we discuss network-aware consensus protocols for consistent data replication in geo-distributed deployments (§ 2.2). Later, we discuss the storage systems for streaming data (§ 2.3), focusing on the systems that use a cloud-based storage service for data persistence.

2.1 Network-aware Distributed Filesystems for Batch Processing

Distributed filesystems, such as GFS [105], HDFS [253], and Ceph [287], are widely used to provide storage for batch processing applications. Applications exchange large volumes

Network-Aware Reads and Writes					
Only reads		Only writes		Both reads and writes	
Static	Dynamic	Static	Dynamic	Static	Dynamic
TidyFS	None	None	Sinbad	GFS, HDFS	OctopusFS

Table 2.1: Taxonomy of network-aware distributed filesystems for batch processing within a single datacenter.

of data with filesystems during various stages of data analytics jobs such as map and reduce phases. To provide fault-tolerance, distributed filesystems create several replicas of data and spread them across different parts of the underlying infrastructure that have non-correlated failures, such as multiple racks in a datacenter and/or across datacenters. Replication policies vary with the infrastructure and application requirements. For example, by default, HDFS creates three replicas, one of which it places in the same rack as the data writer. It places the other two replicas on two nodes in a different rack that does not have correlated network and power failures with the first rack [117].

Depending on the location of replicas, there can be multiple redundant paths between the reader and data source. Consequently, there can be several replica and path combinations that can be used for reading data from a remote node. When a file reader is not co-located with data, the reader may access data from a remote replica that requires traversing oversubscribed network paths, resulting in slow read performance.

To avoid network congestion and minimize loading data over the network, job schedulers usually schedule jobs on the machines that already have the required data or that are close to the data. However, co-locating jobs with data is not always possible due to various constraints such as resource limitations and avoiding load imbalance. Some studies show that nearly 15% to 30% of overall cross-rack traffic is due to file read operations from distributed filesystems [60]. Consequently, replica selection is a significant decision that is closely tied to network path selection and creates a dependency between filesystems and the network for high performance. Selecting an overloaded replica or a replica that is reachable only through congested network paths can significantly downgrade reads' performance.

Existing storage systems use network-aware read and write operations to avoid in-network congestion. They use static network information, such as network-topology information, or dynamic information from the network to optimize file operations, such as

current network state information. Below we provide an overview of network-aware distributed filesystems and discuss the limitations of their replica placement and selection methods. Table 2.1 shows a taxonomy of network-aware distributed filesystems based on their static or dynamic network-awareness methods. We focus on only those filesystems that are designed for a single datacenter deployment.

2.1.1 Filesystems with network-aware reads and writes

GFS, HDFS, and OctopusFS use network-aware methods for both read and write operations. However, OctopusFS uses dynamic methods where as GFS and HDFS use static methods for network-awareness. We discuss these systems below.

Google File System: The Google File System (GFS) [105] is a pioneering filesystem for large scale data analytics using Google MapReduce framework. GFS targets for high availability, reliability, and scalability to store large amounts of data.

Many existing filesystems follow the same architecture as GFS. GFS consists of a single *Master* (metadata node), a client library, and many *chunkservers* (data nodes). The master persistently stores the namespace, keeps track of block locations, and resolves conflicts if there is any divergence in replicas due to failures during concurrent write operations. Chunkservers store the data blocks as individual files in their underlying Linux system.

GFS utilizes static network-topology information in its read and write operations. For replica placement, the GFS master selects the locations of each block that balances disk space utilization and maximizes the network bandwidth. For file read operations, each client reads from the closest replica. However, as we show in our Mayflower filesystem (§3.4), closest replica selection is not always the best choice. Always selecting the closest replica makes the system susceptible to network hotspots and congestion.

Hadoop Distributed File System (HDFS): HDFS [253] is a widely used distributed filesystem for big data applications. HDFS provides storage services for many applications in the Apache eco-system. For example, Spark [293], Flink [19], and Hadoop MapReduce [112] use HDFS for data storage.

HDFS follows the same architecture as GFS. Reads and writes in HDFS use static network topology information for replica selection and placement. Although HDFS supports pluggable replica placement policies, its default policy follows the rack-aware approach in which first replica is placed on the same node as the data writer, and the second and third replicas are placed on different nodes in the same remote rack. For file reads, the clients select the nearest replica, breaking ties randomly.

OctopusFS: OctopusFS [138] leverages heterogeneous storage devices and services in the deployment to provide high performance storage service for big data applications. OctopusFS places replicas based on the estimated current network state and I/O load information collected from the data nodes. Its metadata node periodically gathers node statistics from data nodes to build its current view of the system state. The metadata node then uses the dynamic view of the system to select replica-locations that balance disk I/O and network load across the nodes.

For file read operations, OctopusFS selects the replica that results in the highest read throughput, which it estimates by evaluating the throughput bottleneck of all the replicas. To evaluate the read throughput of a node, OctopusFS monitors the replicas' network and IO statistics. However, replica selection based statistics collected from only the endpoints can be ineffective if internal paths of the network are congested. Replica selection in OctopusFS can select the least loaded replica that is only accessible through the congested network paths.

2.1.2 Filesystems with only network-aware writes

Sinbad: GFS and HDFS select nodes for replica placement based on static information from the network (network-topology and placement in racks), which can create load imbalance in the network and face network congestion. Sinbad [60] addresses this problem and provides a mechanism to dynamically select the nodes for replica placement based on the estimated current network state.

Sinbad estimates the current network state based on the edge links utilization information, which its centralized master periodically collects from the data nodes. It combines the edge-links' utilization information in the form of a tree, moving from hosts towards the core layer, to estimate the in-network links' utilization. Based on the estimates, Sinbad greedily selects the nodes that are reachable from the data source with the least network load.

Although Sinbad's replica placement is a significant improvement over other replica placement strategies, its replica placement is sub-optimal due to working independently from the network. In particular, Sinbad does not account for multiple flows in each link and their individual network utilization. Consequently, network estimations in Sinbad can be inaccurate, which can lead to placing replicas in the nodes which are reachable only through congested network paths.

2.1.3 Filesystems with only network-aware reads

TidyFS: TidyFS [96] provides storage for data analytics applications that produce data in stages and can reproduce the data between different stages of a data analysis job. TidyFS follows the same architecture as HDFS but drops various features for simplicity and to achieve high performance. Similar to GFS and HDFS, TidyFS follows the nearest replica selection approach for file read operations. For file writes, TidyFS does not optimize for network bandwidth. Instead, a client writes the first copy of data locally. The system then selects the subsequent replica locations randomly. However, it gives preference to the nodes with most available disk space to balance space across the nodes.

2.1.4 Discussion

Network-aware writes: OctopusFS and Sinbad select replica locations based on dynamic network state estimations, which balance network load and avoid network congestion. However, their network state estimations are based on the information that is collected only from the endpoints. Therefore, their estimates are accurate only when the endpoints are the bottleneck and the network is congestion free. As a result, replica selection in these systems can be sub-optimal when the network is the performance bottleneck rather than the endpoints.

Network-aware reads: GFS, HDFS, and TidyFS follow the topology-aware approach in which the clients select the nearest replica based on network distance. However, if filesystem clients always read files from their closest replicas, large number of concurrent reads from the clients can create network and I/O hotspots, which can result in slow read performance.

Systems that estimate network load based on only endpoint measurements, e.g., OctopusFS, are unable to effectively avoid in-network congestion. These systems select the read replica and offload the network load balancing to traffic engineering approaches such as ECMP, MicroTE [31], and Hedera [10]. However, scheduling flows between the pre-selected replicas reduces the optimization space for the flow schedulers. Moreover, an independent flow scheduler can be ineffective if all paths to the pre-selected endpoints are congested. Therefore, close collaboration between the filesystem and the network control-plane is required to leverage the availability of multiple replica and network path choices.

We address this problem in our Mayflower filesystem [235]. In Mayflower, we jointly select file replica and network path through close coordination between the filesystem and the network control plane. Joint replica and path selection fully exploits the availability

Network-aware consensus protocols	
Single DC	Multi-DC
Giraffe, Hierarchical quorum consensus, AllConcur	Delegator Paxos, WanKeeper, WAN Paxos

Table 2.2: Taxonomy of network-aware consensus protocols.

of multiple replicas and network paths to avoid network congestion and improve file read performance. We present the detailed design and evaluation of Mayflower in Chapter 3.

2.2 Network-Aware Consensus Protocols

In this section, we review scalable consensus and atomic broadcast protocols that can be used in storage systems for consistent data replication and to solve distributed problems such as leader election. Many storage systems use consensus protocols for consistent data replication, e.g., Spanner [71] uses Paxos [160, 161], and Etcd [89] and CockroachDB [66] use Raft [210]. As the underlying consensus protocols can become the performance bottleneck for the storage systems, we follow infrastructure-aware design approach to improve the performance of geo-distributed consensus protocols, which directly improves the performance of the storage systems using them.

We categorize network-aware consensus protocols in two classes based on their ability to scale in geo-distributed deployment, shown in Table 2.2. We first give an overview of the distributed consensus problem. The overview follows the review of WAN-scale network-aware protocols (§ 2.2.2).

2.2.1 Distributed Consensus Problem

Distributed consensus achieves mutual agreement between distributed nodes in the presence of node and communication failures. Distributed consensus is a key building block in distributed systems to develop reliable systems and solve complex problems such as leader

election and membership management. A distributed consensus protocol must satisfy the following safety and liveness properties [223, 150]:

- *Validity:* If all the correct nodes propose the same value v, the correct nodes decide the value v.
- *Agreement:* All the correct nodes must agree on the same value.
- *Termination:* All the correct nodes must eventually decide some value.

A faulty node can have different types of failures [82] such as crash-stop, omission, timing, and Byzantine. Existing protocols mainly assume crash-stop or Byzantine failures. A crashed node ceases to function in any way. A Byzantine node can act arbitrarily and maliciously.

FLP impossibility result: The FLP impossibility result [99] shows that consensus and atomic broadcast are not deterministically solvable in an asynchronous system where even a single node can fail by crashing. There is always a possibility that the protocol will not terminate, which dissatisfies the termination condition. The FLP result is based on the fact that node failures cannot be detected reliably in an asynchronous system. This is because a lost message cannot be distinguished from a slow message and a crashed node cannot be distinguished from a slow node. In practice, consensus protocols use randomness to circumvent FLP result and eventually meet the termination condition. Moreover, consensus protocols use unreliable failure detectors [55] to achieve safety despite failures.

Failure detectors: Several types of failure detectors can be used depending on synchrony assumptions of communication, which can be synchronous, partially synchronous, or asynchronous. In an asynchronous system, an *eventually weak failure detector* ($\diamond W$) is the weakest failure detector that can be used to solve distributed consensus [54, 55]. An eventually weak failure detector satisfies the *weak completeness* and *eventually weak accuracy* properties. Together these properties enable using failure detectors for development and practical use of consensus protocols in an asynchronous environment despite failures.

Atomic broadcast Properties: Consensus and atomic broadcast are equivalent problems [85, 55]. An atomic broadcast protocol ensures that all correct nodes agree on the delivery of the same messages in the same order. An atomic broadcast protocol must satisfy the following properties [113]:

- *Validity:* If a correct node broadcasts a message m, then it eventually delivers m.

- *Agreement:* If a correct node delivers a message m, then all correct nodes eventually deliver m.
- *Integrity:* For any message m, every correct node eventually delivers m (a) at most once, and (b) only if m was previously broadcast by its sender.
- *Total Order:* If correct nodes p and q both deliver messages m_1 and m_2, then p delivers m_1 before m_2 if and only if q delivers m_1 before m_2.

Consensus Protocols

Many consensus protocols have been developed to solve the distributed consensus problem. Paxos [160, 161] is the seminal protocol that has inspired the development of many other consensus protocols. Consensus protocols can be categorized in leader-based or leaderless/multi-leader protocols.

Leader-based consensus protocols: Leader-based consensus protocols elect a centralized leader that lead protocol execution and serve client requests. Raft [210], ZAB [135], and Multi-Paxos [53] are widely used leader-based protocols. However, these protocols are mostly deployed using only a few nodes because of their scalability limitations. Leader-based protocols do not scale to a large number of nodes because the centralized leader performs considerably more work as compared to other nodes in the consensus ensemble. As a result, load concentrates on the leader when the number of nodes increases in the system, which makes the leader the performance bottleneck.

Multi-leader consensus protocols: Leaderless or multi-leader consensus protocols, e.g., Mencius [184], EPaxos [195], and S-Paxos [34], distribute the load across all the nodes in the cluster, which makes them more scalable. However, these protocols do not utilize their resources efficiently as they are designed to be oblivious to the underlying datacenter infrastructure. In a geo-distributed deployment, WAN links that connect datacenters are highly oversubscribed as compared to LAN links within a datacenter. Therefore, cost of sending messages over WAN links across datacenters is much higher, have higher bandwidth limitations and incur higher latency, as compared to sending messages within the same datacenter. However, communications across nodes during protocol execution in existing protocols do not consider the proximity of nodes within and across datacenters. Some of the protocols send equal number of messages to all the nodes in the cluster regardless of their location. As a result, these protocols do not efficiently utilize the network and computational resources, which hinders their ability to scale to a large number of geo-distributed nodes.

In contrast to network-oblivious protocols, network-aware consensus protocols utilize information from the network to achieve higher performance. In this thesis, we focus on the scalability of network-aware consensus protocols that are designed for deployments at global scale. In the following section, we discuss network-aware consensus protocols that are targeted for geo-distributed deployments.

2.2.2 WAN-scale network-aware consensus protocols

In this section, we review network-aware consensus protocols that are designed for geo-distributed deployments.

Delegator Paxos: Delegator Paxos (D-Paxos) [178] targets a geo-distributed model in which each datacenter consists of several nodes. It organizes nodes in the same datacenter as a single logical group, where each group has an elected leader that handles client requests and communicates with other groups. The groups concurrently execute different steps of the protocol in a pipelined fashion, making D-Paxos highly concurrent and achieve high throughput. However, D-Paxos does not fully distribute load across all nodes because the group leaders perform significantly more work as compared to the replicas in their groups. As a result, D-Paxos' throughput can be improved by distributing load across all nodes in the cluster.

WanKeeper: WanKeeper [5] is a distributed coordination service that extends ZooKeeper for larger deployments. The main motivation behind building WanKeeper is to provide a geo-distributed coordination service with high throughput and low latency of client requests. To achieve this goal, WanKeeper logically organizes nodes in a two level hierarchy, which is similar to Delegator Paxos [178]. At the bottom level, it groups the nodes based on regions and partitions the object space across groups. Each region runs ZooKeeper Atomic Broadcast (ZAB) protocol [135] to commit requests. The leaders of the regions make another group of ZAB, forming the top level in the hierarchy. The top-level leader coordinates transactions that are across regions.

Although WanKeeper improves the performance and scalability of ZooKeeper, it has two limitations which limit its scope and scalability. First, it provides causal consistency for writes across regions to achieve node-local reads. This can be a problem for applications that require strong consistency guarantees across regions, e.g., Spanner and Megastore. Second, WanKeeper concentrates more load on the brokers, that limit the number of nodes within a region. In comparison, Canopus provides strong consistency across regions and distributes load across all nodes in the consensus cluster.

WAN Paxos: WAN Paxos (WPaxos) [4] borrows the idea of flexible quorums from Flexible Paxos [121]. It forms the quorums such that the common-case messages are exchanged only between nearby nodes. Like WANKeeper, WPaxos organizes nodes in groups based on their geographical location. It forms quorums across groups such that the fast path of the protocol only touches two groups. The groups can be in the same or in different datacenters. Unlike WANKeeper, each node in WPaxos serves as a leader for a subset of objects. A node becomes the leader for an object by executing the phase-1 protocol of Paxos. After that, the leader only executes phase-2 of Paxos to achieve consensus for that object.

WPaxos works best when there is access locality of objects such that the object space can be partitioned across many groups. In the scenarios where the same object is frequently accessed from different geographical locations, either WPaxos has to frequently elect leaders for each region, resulting in high latency for most of the operations, or it provides low latency for only those requests that originate from the leader's region.

2.2.3 Discussion

Existing geo-distributed network-aware consensus protocols target to achieve high scalability by forming multiple consensus groups and by distributing load across the groups. However, each group has a leader that performs more work as compared to other nodes in the group. As a result, scalability of the consensus protocol is limited by the capabilities of only a few nodes in the cluster. Consequently, the protocols do not effectively utilize all available resources and under-perform than their potential.

To that end, we developed Canopus consensus protocol in this thesis, which addresses the problem of scaling consensus to a large number of geo-distributed nodes. In comparison with existing protocols, Canopus fully distributes load across all nodes in the consensus ensemble. Moreover, Canopus exploits the hierarchical and resilient design of modern datacenters and the underlying network to simplify its design. We further provide the design and evaluation of Canopus in Chapter 4.

2.3 Cloud-based Storage Systems for Financial Data Streams

In this section, we discuss the storage systems designed for realtime data analytics in a public cloud deployment. Realtime data analytics are becoming increasing popular as they

provide time-critical insights extracted from recently generated data. Stream processing applications, such as Spark streaming [22] and Flink [19], process continuously arriving data and generate actionable results that are valid only for a short time window, from milliseconds to a few seconds. As the value of data diminishes over time, realtime analytics demand high performance for all stages in the stream processing pipeline, which include data transformation, storage, processing, and presentation of the analysis results.

A broad range of applications and use cases have been developed based on timeseries data. This thesis focuses on capital market use cases where data originates from *market data providers*, such as stock exchanges and market data vendors, in response to financial activities such as buying and selling of stocks. Market data providers publish continuous streams of messages that is ingested to storage system for realtime and historical data analysis.

Other than performance requirements, managing operational costs is another challenge in capital market use cases. Storing large volumes of data forever incurs high operational costs over the period of time. Several public cloud providers offer scalable and fault-tolerant storage services at low price, e.g., Amazon S3. Therefore, existing cloud storage services can be used for cost-effective storage of large volumes of data. However, cloud storage services are not feasible to ingest data directly from stream sources as data access latency from the cloud services can be very high, which makes them not feasible for realtime data access. Moreover, they do not provide data indexing, which is needed to support time-based fast range queries. Consequently, an intermediate storage system is required that ingest data from data streams, index data for reads and serve reads with low latency, while continuously persisting data using a cloud storage service.

Stream processing architectures: Existing systems commonly use Lambda [190] and Kappa [149] architectures for stream processing. These architectures use message queues, such as Apache Kafka [20] and Amazon Kinesis [15], to ingest data from data sources. Applications fetch data from the message queues for realtime data analytics. Moreover, in Lambda architecture, separate storage systems fetch data from the message queues for persistence and historical data analysis.

Although message-queue systems are highly scalable and fault tolerant, pulling data from these systems for processing incurs high latencies, tens to hundreds of milliseconds [132], which are highly undesirable in capital market uses cases [232, 273]. Moreover, separate systems are required for data indexing and to support parallel access to realtime and historical data. Use of additional systems for storage increase system complexity and maintenance overheads. Therefore, a unified storage system is required that ingest data at high rates and provide realtime and historical data access in the same system.

Below we discuss storage systems for streaming data in the context of capital market use cases and financial data streams. First we give an overview of financial data streams, which follows an overview of cloud infrastructure. We then discuss the storage systems for streaming data that support data persistence using a public cloud service. As we target the system to be deployed in a public cloud, we discuss only those systems that support a public cloud storage service for data persistence.

2.3.1 Financial Data Streams

Capital markets, such as stock exchanges and financial institutions, produce large volumes of messages at a high pace in response to financial transactions such as trading of stocks. Messages representing financial transactions are commonly termed as *market data*. A message, also called a *tick*, contains relevant information about trade of a financial instrument. Stocks, funds, and currencies are a few examples of financial instruments. Most common information about a trade includes the id of traded financial instrument (symbol), ask/bid prices and quantities, and a timestamp indicating the time when when the financial instrument was traded.

Market data from its originating sources is distributed in the form of market data *feeds*, where a feed is a continuous stream of messages from a particular source such as a stock exchange. Messages in feeds are time ordered and contain sequence numbers, which can be used for message ordering and to detect lost messages. *Market data vendors*, such as Operations Price Reporting Authority (OPRA) [214] and Vereinigte Wirtschaftsdienste GmbH (vwd) [283], collect data feeds from multiple sources and redistribute the feeds to their subscribers. The feeds are usually published through a dedicated network using IP multicast [216, 101]. Eventually, data feeds are used in various applications that make use of both realtime and historical data, e.g., competitive pricing analysis, market data terminals [101]. We target to build a storage system that collects market data from market data vendors, being deployed in a public cloud. Applications access market data from the storage system for both realtime and historical data analysis.

Reliability of financial data streams: Market data providers often ensure fault-tolerance at data source layer using two methods. First, data providers publish replicated data streams for high fault-tolerance [216, 262]. Data in both streams are published independently, i.e., failure of one stream does not impact the other stream. Data receivers can receive data from both streams separately for high fault-tolerance. Second, data sources retain recent data for some time, e.g., for a day, and provide separate channels to receivers to playback the streams offline at additional costs [215]. If data is lost during data ingestion

at the receiver end, the receivers can recover lost messages from data providers.

Nonetheless, many financial applications prefer low latency and data staleness over high fault-tolerance [101]. For example, terminals displaying latest stock prices prefer to drop messages rather than displaying staled values. In addition, some companies, such as Smash.BI Inc. [256], target to achieve low latency instead of high fault-tolerance by eliminating data replication during data storage. Instead, they prefer to achieve fault-tolerance by ingesting data from replicated streams and use them as independent streams.

2.3.2 Cloud Storage Services

Cloud providers offer many cost-effective storage services that can be used for persistent data storage. For instance, Amazon offers DynamoDB key-value store, S3 object store, NFSv4-based Elastic File System (EFS), and Elastic Block Store (EBS). Target use cases and storage properties vary with these services, e.g., the cost model, performance, availability guarantees, consistency, and usage limitations are different for EBS, EFS, and S3.

We use Amazon S3 as the persistent storage in our Kawkab storage system because of its high scalability, fault-tolerance, and low cost of storage. Moreover, S3 is globally accessible from all nodes in different regions. Although EBS and EFS offer higher performance, they are not suitable to store financial data streams. EBS volumes cannot be shared among nodes, which either prevents reading same data from different nodes or require provisioning additional EBS volumes to replicate data and make it available to each node. In contrast, each object in Amazon S3 is simultaneously accessible to all nodes in all regions. EFS is best used for large data analytics workloads that require concurrent access of same data among a large number of machines. However, we target applications that mostly perform realtime streaming data analytics instead of frequent batched data analysis at large scale.

2.3.3 Amazon Simple Storage Service (S3)

Amazon Simple Storage Service (S3) is an object store provided in the Amazon cloud infrastructure. S3 is highly scalable (storing trillions of objects), fault-tolerant, and cost effective, which makes it an effective long term solution for persistent storage of streaming data. For high fault-tolerance, S3 replicates data to at least three availability zones, which do not have correlated failures. Moreover, data stored in S3 is accessible simultaneously from multiple regions in a scalable way.

Storage classes in S3: S3 provides several classes of storage that have different price, performance, and fault-tolerance characteristics. S3 standard class is designed to store

frequently accessed data with high throughput and low latency. Although S3 Infrequent-Access (IA) class has a lower cost than the standard class, it restricts that the data has to be stored for at least a month. The glacier class costs least at the expense of high latency of data access. S3 provides a mechanism to automatically move data between these classes based on the access frequency and data age. As online data analytics mostly access recently arrived data, the frequency of accessing particular data decreases with time. Therefore, S3 storage classes provide a low cost storage for historical data that is not accessed for a long period.

Limitations of S3: Storing streaming data in S3 with high performance can be challenging. Financial data streams consist of small messages, ranging from tens to a few hundreds of bytes. However, the pricing model and performance of S3 is optimized for storing large objects. Frequently performing small write operations not only results in low throughput, but also incurs high cost due to the operations-based pricing policy of S3 [136]. Kadekodi *et al.*, [136] have shown that the throughput of writes increases with data size, ranging up to 300x throughput improvement for large objects as compared to small objects. They report that the throughput of uploading 4MB objects is 1000x times faster than uploading 1KB objects for the same data sizes. Moreover, S3 throttles request rates based on keys, limiting the rate of concurrent reads and writes to each key. Therefore, directly storing data streams in S3 is not feasible.

Alternative cloud providers: Microsoft Azure blob store, Google cloud storage, and Rackspace Swift provide solutions similar to S3 with competitive pricing and features. Therefore, the storage solutions that work with S3 can also be tailored to work with the aforementioned object stores.

2.3.4 Cloud-based Storage Systems for Streaming Data

Most storage systems for streaming data take advantage of the time-based ordering of data to optimize storage and access. These systems are also known as timeseries data management systems (TDMS). Jensen *et al.*, [133] and Bader *et al.*, [25] have thoroughly surveyed existing storage systems for streaming data. Existing systems either provide a sophisticated database targeted towards timeseries data or only the storage system with a limited support for data summarizing queries. Our focus is to develop only the storage system that can be used by applications for stream data analysis originating from financial markets.

Most TDMSes use either a custom-built storage system or a distributed filesystems for data persistence. Among these systems, *Bolt* [111], *ReSpawn* [42], and *Storacle* [51]

provide a hybrid-cloud storage solution for IoT applications. Therefore, these systems cannot be used for our target applications. Druid [290], BTrDB [18], and Alluxio [172] are more suitable for cloud deployments as they support cloud-based persistent storage. However, these systems are not tailored for financial market applications. As a result, these systems cannot take advantage of unique characteristics of financial data streams such as fault-tolerance at the data source. In our Kawkab storage system, we exploit fault-tolerance characteristics of financial data streams to eliminate data replication during data ingestion. In result, Kawkab supports higher data ingestion rates as the data ingestion overhead is low.

Distributed Filesystems in the Cloud

An alternative approach is to store streaming data using existing distributed filesystems such as HDFS. For example, OpenTSDB [213] uses HBase for data storage, and InfluxDB [126] uses a custom built filesystem for data persistence. As others have pointed out [224, 18], HBase incurs high data query latency. Therefore, querying data through HBase is not feasible. Gorilla [224] and BTrDB [18] are high performance storage systems for streaming data. However, Gorilla uses GlusterFS and BTrDB uses Ceph [287] filesystem for data persistence. These systems can be deployed in the cloud using EC2 machines that have several storage devices.

The primary reason for avoiding DFSes in that these systems can incur several times higher cost as compared to Amazon S3 [243]. Running a complete distributed filesystem requires provisioning several virtual machines for a long time, which incurs continuous operational cost. Second, DFSes do not provide any guarantees of data access latency. These systems are optimized for high scalability and fault-tolerance for batch processing, which do not impose strict latency requirements.

2.3.5 Discussion

Existing streaming data systems are not purpose-built for capital market use cases and to store data using a cloud-storage service. Therefore, theses systems are less efficient to store financial data streams. Following the infrastructure-aware design approach, we build *Kawkab*, a streaming data storage system that is purpose-built around financial market data streams and cloud storage services. Kawkab leverages fault-tolerance guarantees provided by financial data providers to optimize data ingestion. Moreover, it leverages the scalability of cloud storage services such as Amazon S3 to support both realtime and

historical data queries in the same system without compromising the latency and staleness of recent data. Chapter 5 presents the design and evaluation of Kawkab.

2.4 Summary

Our review of existing network-aware storage systems for datacenters reveal that either they use only static network information, or they do not combine replica selection or node placement with network path selection, which can result in sub-optimal performance. Therefore, we follow a network/filesystem co-design approach in Mayflower, discussed in the next Chapter, that mutually selects the read replica and the network path to improve their performance.

We found a similar architectural gap in the network-aware consensus protocols, which are commonly used in geo-distributed storage systems. Existing network-aware consensus protocols do not consider exploiting nodes deployment in racks and the presence of a hierarchy of latencies in datacenters. Moreover, they concentrate load on a few nodes and do not efficiently utilize available resources. Our Canopus protocol fills this gap, being designed according to the infrastructure and deployment model in datacenters. We discuss Canopus in Chapter 4.

Finally, the review of storage systems for fast data in a cloud deployment show that existing systems do not provide an adequate solution that is optimized for string financial data streams in a public cloud. We present Kawkab filesystem in Chapter 5, which addresses storage requirements of streaming data storage in capital market use cases.

Chapter 3

Mayflower Distributed Filesystem

In this chapter, we present our Mayflower [235] distributed filesystem that follows the infrastructure-aware design approach for performance optimizations. As we have reviewed in the previous chapters, distributed filesystems such as GFS, HDFS, Ceph, and MapR are commonly used in datacenters to store big data. Such filesystems extensively utilize network resources [60], as a result of the filesystems build a strong and reciprocal relationship with the underlying network. However, most of the existing filesystems either use the network as a black box or do not use dynamic network state information to select the best replica for file read operations. As a result, the file reads are susceptible to in-network congestion. The more advanced filesystems [60, 138] consider the current in-network state for reads and writes to avoid the network bottlenecks. However, such systems do not co-ordinate with the network control plane, which independently schedule the flows without reconciling the flow scheduling decisions with the filesystems. As a result, the file operations can perform poorly when all the paths between the pre-selected endpoints are congested.

We develop Mayflower to fill this performance gap by co-designing it with the network control plane. Instead of first selecting the replica and then the network path, Mayflower jointly selects the best replica and path combination mutually with the SDN controller. In the following sections, we describe the design of Mayflower's replica-path selection methodology and present its performance evaluation.

3.1 Design Overview

In this section, we first describe our assumptions regarding the typical usage model for Mayflower and detail the properties of Mayflower's target workload. We then provide the details of its system architecture.

3.1.1 Assumptions

Our design assumptions are heavily influenced by the reported usage models of Google filesystem (GFS) and HDFS. Mayflower assumes the following workload properties:

- **Size and volume of the files:** The system only stores a modest number of files (on the order of tens of millions). File sizes typically range from hundreds of megabytes to terabytes. The metadata for the entire filesystem can be stored in memory on a single high-end server.
- **Large sequential reads:** Most reads are large and sequential, and clients often fetch entire files. This is representative of the applications that partition work at the file granularity. In these applications, clients fetch and process files one at a time, and the file access pattern is often determined by the file contents.
- **Append-only writes:** File writes are primarily large sequential appends to files; random writes are very rare. Applications mutate data by either extending it through appends, or by creating new versions of it in the application layer and then overwriting the old version using a move operation.
- **Read-heavy workload:** The workloads are heavily read-dominant. Read requests come from both local and remote clients.
- **Network as the bottleneck:** The network is the bottleneck resource due to a combination of high performance SSDs, efficient in-memory caching, and oversubscription in datacenter networks.

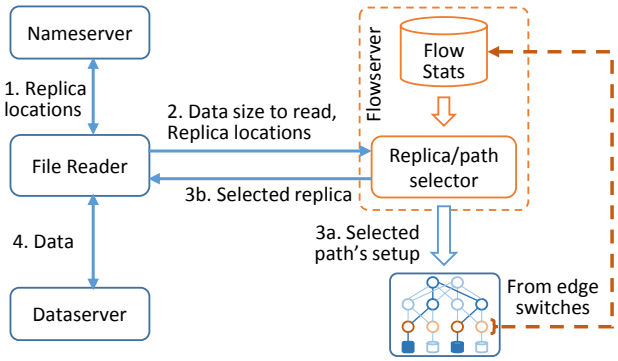


Figure 3.1: Mayflower system components and their interaction in a file-read operation.

3.1.2 Architecture

Mayflower's basic system architecture consists of three main components: Dataserver, Nameserver, and Flowserver. The design principles of Dataserver and Nameserver are the same as their counterparts in GFS and HDFS. Therefore, we leave out their details and only discuss them briefly in this section.

Dataserver: The Dataserver stores actual file data. Each file is partitioned into chunks of configurable size. Similar to HDFS [116], the chunk size is a system-wide parameter that is typically equal to or larger than 128 MB. Each file is represented as a directory in the Dataserver's local filesystem. The file-chunks are stored as individual files along with other metadata information in the file's directory. Each file has a primary Dataserver, which is responsible for ordering all of the `append` requests for the file. The primary Dataserver relays `append` requests to the other replica hosts while servicing the request locally. In order to support atomic `append` requests, the Dataserver only services one `append` request at a time for each file.

Nameserver: Nameserver stores the metadata of the filesystem, including the file-to-chunks and file-to-Dataservers mappings. Clients contact the Nameserver when they need to `create` or `delete` a file, or lookup the size of a file, and cache the results for a Nameserver-specified amount of time to reduce lookup traffic. The mappings in the Nameserver are stored in a persistent database to speed up Nameserver restarts after a graceful shutdown. After an unexpected restart, instead of reading from the possibly stale database, the Nameserver rebuilds the mappings by scanning the files stored at the Dataservers.

For replica placement decisions, Nameserver takes into account system-wide fault-tolerance constraints, such as the replication factor and the number of fault domains. Currently, the Nameserver makes replica placement decisions using only static network topology information. This is because the focus of this work is on improving read performance for read-dominant workloads.

Nameserver is designed as a logically centralized service. We can improve its fault-tolerance by replicating its state across multiple nodes using a state machine replication system such as Paxos [160].

Flowserver: The Flowserver's primary task is to select both the replica and the network path for the Mayflower read operations. The Flowserver models the whole network and makes its selections based on the estimated current network state. It builds its network model by tracking the network paths assigned to the read and write requests, and estimating the network utilization of each flow. The Flowserver can work together with existing network managers, such as DIFANE [291] and DevoFlow [73], to identify non-Mayflower-related elephant flows and estimate their bandwidth utilization. Using this model, the Flowserver can select a replica and path combination that minimizes the average completion time of all non-mice flows in the network, including existing network flows.

As the Flowserver does not explicitly control the bandwidth usage of each flow through bandwidth reservations, the network utilization estimates are often inaccurate. To avoid error propagation and reduce the effect of inaccuracy, the Flowserver periodically fetches the flow-stats from the edge switches. The flow-stats give the recent bytes-count for the flows. As the flow-stats are not collected very frequently, these byte-counts translate to the approximate bandwidth usage of the flows and their remaining size. These approximations reduces the inaccuracy in our estimations during replica and path selection.

In between measurements, the Flowserver tracks flow add and drop requests for the read jobs, and recomputes an estimate of the path bandwidth of each affected flow after each request. This ensures that completion time estimates are accurate, and also reduces the need to poll the switches at short intervals. The polling interval defaults to one second in our implementation.

File read operation: Figure 3.1 illustrates Mayflower components and their interactions with a client in a file-read operation. In this example, the file reader contacts the Nameserver to determine the replica locations of its requested file and then contacts the Flowserver to determine which replicas to read from. The Flowserver executes its replica-path selection algorithm (§ 3.2.2) for replica and path selection. In addition to selecting replicas, the Flowserver also installs the flow path for the request in the OpenFlow switches. Finally, the client contacts the Dataservers to retrieve the file.

Append-only semantics: Mayflower provides file-specific tunable consistency to its clients. In order to reduce reader/writer contention and consistency-related overhead, Mayflower does not support random writes. Instead, files can only be modified using atomic `append` operations. Random writes can be emulated in the application layer by creating and modifying a new copy of the file and using a `move` operation to overwrite the original file.

3.1.3 Consistency

By default, Mayflower provides sequential consistency for each file where clients see the same interleaving of operations. This requires that all `append` requests are sent and ordered by a file's primary replica host. Upon receiving an `append` request, the primary replica host relays the request to the other replica hosts while performing the append locally, and the append completes once the primary receives an acknowledgement from all of the replica hosts. Clients can however send read requests to any replica host and coordination between hosts is not required to service the read request.

Alternatively, Mayflower can be configured to provide linearizability with respect to `read` and `append` requests. The traditional approach to ensure strong consistency is to order all requests by the file's primary replica. However, Mayflower leverages its append-only semantics to only require sending the last chunk's `read` requests to the primary replica host. All other chunk requests can be sent to any of the replica hosts since every chunk except the last one are essentially immutable. Therefore, for large multi-gigabyte files, the vast majority of chunks can be serviced by any replica host while still maintaining strong consistency. The only limitation to this approach is that it cannot provide strong consistency when `read` and `append` requests are interleaved with `delete` requests. Files that are deleted can temporarily appear readable due to client-side caching of the file-to-Dataserver mapping. However, we believe that this is a reasonable tradeoff for improving read performance given that delete requests are relatively rare.

Providing linearizability with interleaving delete requests can be achieved in the following way: When the Nameserver receives a delete request, it marks the file for deletion, but delays the delete for T time, where T is the maximum expiration period for the client-side file-to-Dataserver cache. During the delay period, if a client contacts the Nameserver for the file-to-Dataserver mapping, the Nameserver returns the result with a cache timeout equal to the remaining delay period. After T time, the Nameserver performs the delete operation. In this way, all the clients see the same interleaving of delete requests and cannot read the file from any replica after deletion. The performance impact is small for workloads where delete requests are rare.

3.2 Replica and Path Selection

When a Flowserver receives a replica selection request from a client, it executes our replica-path selection algorithm (§ 3.2.2) to select a replica host and the network path for the request. The replica-path selection process is based on estimated network state that is built using bandwidth estimations and remaining flow size approximations.

Target performance metrics: Our target performance metric is the average *job completion times*. If we assume that the network is the bottleneck for a read operation, the job completion time for a read request can be minimized by maximizing the max-min bandwidth share of the read job's flows. Unlike other flow scheduling systems [10, 60] that are based on link utilization measurements, Mayflower maximizes the max-min bandwidth share because, in addition to link capacity, the max-min calculations also capture the number of existing flows and their bandwidth share in each link. In contrast, absolute link utilization only provides information on available link capacity.

Even though the path with the most bandwidth share is a good choice, it is not always the best choice in highly dynamic settings. This is because new flows affect the path selection for already scheduled flows. We must therefore account for the effect on existing flows when selecting the replica and the network path for a new request.

3.2.1 Problem Statement

Informally, our optimization goal is to select the replica and network path combination among the paths from all replicas to the client, that minimizes the average completion time of Mayflower read jobs. In other words, our goal is to select the network path that minimizes the completion time of both the new flow as well as existing flows, including the previously existing elephant flows belonging to other applications. Our replica and path selection algorithm considers the following criteria: The paths of existing flows, the capacity of each link, the data size of each request, the estimated bandwidth shares of existing flows, and the remaining untransferred data size of existing flows.

A formal description of our problem, which resembles a minimum-cost flow problem, is as follows: Let G be the graph representing the paths from source to destination. Let $b_{i,j}$ be the bottleneck bandwidth on path from node i to node j. Let $c_{i,j}$ be the cost of impact on existing flows on link (i, j) . Let $d_{i,j}$ be the data flowing on link (i, j). Let $I_{i,j}$ be a binary indicator that a flow is assigned to link (i, j). Let s be the supersource, connected to all replicas with 0-cost paths. Let t be a sink node. Let x be the data size. Given G, as well as $c_{i,j}$ and $b_{i,j}$ $\forall$ $(i, j) \in G$, we have:

$$\min_{d_{i,j}, I_{i,j}} \sum_{(i,j)\in G} \frac{d_{i,j}}{b_{i,j}} + I_{i,j} c_{i,j}$$

subject to:

$$\sum_{j:(i,j)\in G} d_{i,j} = \sum_{j:(j,i)\in G} d_{j,i}, \forall i \neq s, t, i \in G$$

$$\sum_{i:(s,i)\in G} d_{s,i} = \sum_{i:(i,t)\in G} d_{i,t} = x$$

$$0 \leq d_{i,j} \leq x I_{i,j}$$

The objective function is formulated such that, in the optimal solution, the cost will be minimized if the binary variable $I_{i,j} = 1$, i.e., if and only if the link (i, j) is on the path used by the flow.

A path consists of several links that may have different bandwidth utilizations. This formulation assumes that the bandwidth of flows along each link in a path is equal to the bottleneck bandwidth in that path. However, a link can be part of multiple paths, and the bandwidth of that link can depend on other paths and flows. To accommodate this, we construct a graph G' to be a tree with the sink node (client) as the root, and all the possible paths to a replica encoded as source nodes, i.e., one link per path to a replica. The cost of using each link in G' is the aggregate cost of using all the links on the corresponding path in G, with the bandwidth set to the bottleneck bandwidth of that path. If paths and costs are precomputed, we can find the least-cost path for the flow by calculating the cost of the links and choosing the link in G', i.e., path in G, with lowest cost. This serves as the basis of our algorithm.

3.2.2 Replica-Path Selection Process

Mayflower's replica-path selection algorithm evaluates all the paths from each replica to the client and selects the path that has the minimum cost

$$\text{Path}_{\min}(P) = \underset{\forall p \in P}{\operatorname{argmin}} \, \text{Cost}(p) \tag{3.1}$$

where P is the set of all distinct paths between the data reader and the replica sources. We restrict to selecting from only the shortest paths between two endpoints.

The cost of each path $p \in P$ is the completion time of the new read job j, and the increase in completion time of the existing jobs in each link along that path

$$\text{Cost}(p) = \frac{d_j}{b_j} + \sum_{\forall f \in F_p} \left[\frac{r_f}{b'_f} - \frac{r_f}{b_f} \right] \tag{3.2}$$

where d_j is the requested data size and b_j is the estimated bandwidth share of a new flow on path p. The first portion of Equation 3.2 estimates the cost of the new flow, while the second portion estimates the impact of the flow on existing flows F_p in path p. The cost of an existing flow $f \in F_p$ is the estimated increase in completion time to download its remaining data r_f when the current bandwidth b_f is decreased to b'_f due to the addition of the new flow in the path. The current bandwidth share and the remaining sizes of the existing flows are measured through flow-stats that are collected from edge switches in the network.

The bandwidth share of the new flow and the change in bandwidth share of the existing flows are estimated through max-min fair share calculations. For each link, given a set of flows that use the link and their bandwidth demands, we equally divide the bandwidth across each flow up to the flow's demand while remaining within the link's capacity. The demand for the existing flows is set to their current bandwidth share whereas the demand of the new flow is set to the link's capacity. The estimated bandwidth b_j of the new flow is its bandwidth in the bottleneck link in the path. The new bandwidth estimate of the existing flows is their bandwidth share when a new flow with bandwidth demand b_j is added to the links in the path.

Background traffic: Mayflower can include information about elephant flows generated by other applications, which we refer to as background flows, to improve the accuracy of its bandwidth utilization estimates. Several existing systems have been proposed to detect elephant flows [10, 291, 73]. We assume that an existing system performs elephant flow detection and informs the path taken by these flows to the Flowserver. The Flowserver adds these flows in its network view which allows correct bandwidth estimates for replica-path selection.

Unknown flow size: Occasionally, the size of a flow is not known (as is the case with background flows), or the provided information is inaccurate. Flow size information is not critical when estimating the cost of a new flow because, as shown in Equation 3.2, the variation in the cost of a new flow derives from its estimated bandwidth. The flow size is used in the estimation of completion time increase for existing flows. If the flow size information is unknown during replica-path selection, then the average elephant flow size

is used as an estimate.

Slack in updating bandwidth utilization: When a path is selected by the Flowserver using the replica-path selection algorithm, the bandwidth utilization for the new flow is set to its estimated bandwidth share. The bandwidth share of the existing flows in the selected path are updated with their new estimated values. To avoid a herd effect, each newly scheduled flow is placed in an *update-freeze state* for a second, which essentially ignores the flow's first bandwidth update. Flows take some time to converge, and because their stats are collected periodically, the first update can coincide with a flow's unstable starting time. This incorrect bandwidth usage information can cause a large number of flows to be assigned to the same path, congesting the network. Therefore, the bandwidth usage of the flows in the update-freeze state is not updated. However, the remaining flow size information is allowed to be updated. Moreover, the replica-path assignments are not blocked during the update-freeze state of flows.

Simplifying bandwidth estimations: The reduction in bandwidth share of flows in a path may result in increase in bandwidth share of flows in other paths. This can indirectly affect nearly all flows in the system. To accurately measure the impact of adding a new flow on a path, we need to not only update the state of the flows on the selected path but also identify and update changes in the bandwidth utilization of flows on other paths. This greatly increases the cost of bandwidth estimation.

For simplicity, we ignore the secondary effects of changes in bandwidth, and estimate and update the bandwidth share of flows in only the paths between replicas and the client. Estimation errors do not accumulate because we periodically update our bandwidth estimates from the edge switches' flow-stats. We also use the flow stat information to update the bandwidth utilization of the remaining flows in the network. This significantly reduces the complexity of the problem while still providing good bandwidth utilization approximations.

Mayflower reduces the overhead of network stats collection by polling only the edge switches. Stats collection for non-edge switches is unnecessary because their link utilizations can be inferred by combining flow path information with flow stats from the edge switches. Moreover, the stats collection period can be relatively large without affecting Mayflower's performance. This is because flow stats are primarily used to reduce small approximation errors in our network model. Mayflower uses a one second stats collection interval by default, which is based on the update interval of a flow's byte-counter in OpenFlow switches [212]. Our experimental results are unaffected when we increase the interval period from one to five seconds. Finally, polling across edge switches is staggered to further reduce the impact of stats collection on the network.

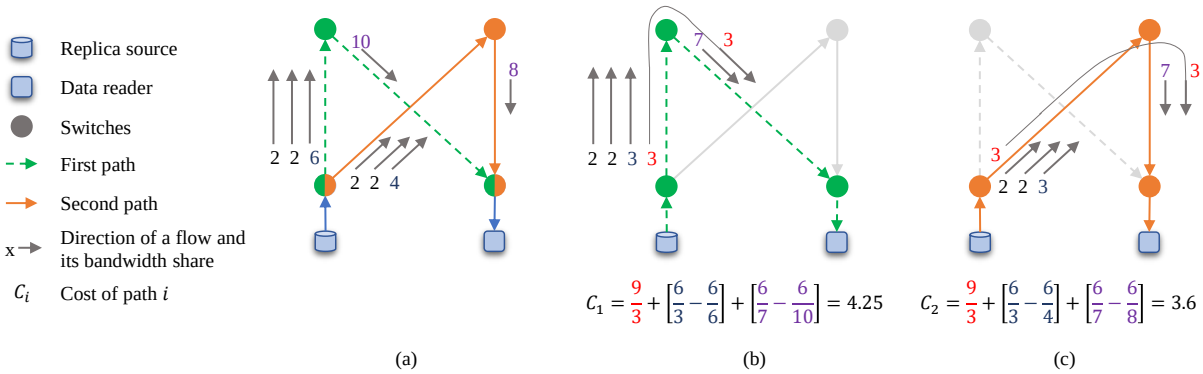


Figure 3.2: An example of cost calculation for replica-path selection: (a) Existing flows' bandwidth share along two paths (10Mbps links). (b) Bandwidth share of the flows if a new flow is added on the first path where C_1 is the cost of adding the new flow on the path. New flow's size is 9Mb whereas the remaining size of the existing flows is 6Mb. (c) Bandwidth share of the flows if the new flow is added on the second path and its corresponding cost is C_2.

An illustrative example: Consider an example where a client reads 9Mb data from a replica source, illustrated in Figure 3.2. The figure shows two edge switches connected to two aggregate switches through 10Mbps links. There are two equal length paths between the reader and the data source. Both paths have three flows on the second link, which connects the edge switch to the aggregate switch, and one flow on the third link, which connects the aggregate switch to the edge switch.

We first evaluate the first path, shown in Figure 3.2b. The second link is the bottleneck link for a new flow because it gives a 3Mbps share to the new flow, compared to a 5Mbps share on the third link. Therefore, the new flow will have a 3Mbps share on the first path and the read job will take $9 \,/\, 3 = 3$ seconds to complete. According to the max-min fair share calculations, the existing flow with 6Mbps share in the second link will be reduced to 3Mbps, and the 10Mbps-flow on the third link will be reduced to 7Mbps. As a result, the completion time of the flows will be increased by $[(6 \,/\, 3) - (6 \,/\, 6)] = 1$ and $[(6 \,/\, 7) - (6 \,/\, 10)] = 0.25$ seconds respectively, assuming the flows' remaining sizes to be 6Mb. Therefore, the estimated cost of the first path, which is a measure of the increase in total completion time, is $3 + 1 + 0.25 = 4.25$. Similarly, the cost of the second path would be 3.6, and therefore the second path will be selected for the read operation.

In this example, the bandwidth share of the new flow is the same for both paths. The

difference in cost is due to the increase in completion time of the existing flows. The second path has an increase of 0.6 seconds in the completion time of the existing flows, compared to 1.4 seconds for the first path.

3.2.3 Reading from Multiple Replicas

Mayflower reads from multiple replicas in parallel if doing so results in a reduction of the completion time. A read job is split into two subflows if the combined cost of the two subflows is smaller than the cost of reading from only one replica. The cost function remains the same as in Equation 3.2. The bandwidth estimate for the two subflows is adjusted as if two new flows are being added in the system.

After selecting the network paths for the subflows and estimating their bandwidth shares, the data read size for each subflow is divided such that the subflows finish at the same time. The subflows are assigned different replicas to avoid encountering the same network bottlenecks.

Multiple replicas selection process: First, a replica-path p_1 is selected for the first subflow f_1 using the replica-path selection algorithm (§ 3.2.2). Assume that the estimated bandwidth share of f_1 is b_1 and its cost is c_1. A temporary flow is then added in the selected path to update the bandwidth estimates of the existing flows in that path. Then another replica-path p_2 is selected for the second subflow f_2, which has estimated bandwidth share b_2 and cost c_2. As p_1 and p_2 may have common links in their paths, f_2 may reduce the bandwidth share of f_1. Therefore, the bandwidth share and cost of f_1 are adjusted to b'_1 and c'_1 according to the reduction in b_1. If the combined cost, $c_t = c'_1 + c_2$, of the two subflows is smaller than c_1, which is the cost of using only one flow for the read job, the replica-paths p_1 and p_2 are selected for the subflows. Otherwise, the temporary changes done by adding a temporary flow in p_1 are rolled back, and only p_1 is selected by the Flowserver. If two subflows are selected, the flow size S_i for each subflow is adjusted proportionally to its estimated bandwidth share: $S_i = d * b_i/b$, where d is the requested data size and $b = b'_1 + b_2$.

The Flowserver returns the replica-paths and the associated data sizes to the client. The client concurrently downloads from the given replicas in chunks of K MB, where K is a tunable parameter that we set to be 32 MB for 128 MB block sizes. If one of the flows finish earlier, it starts downloading the remaining chunks of the other flow, including the last chunk.

Our results show that reading from multiple replicas further reduces the completion time of read jobs by 10% on average. Moreover, the average difference in completion times

between the two subflows of a read job is less than one second when reading a 128 MB block.

3.3 Implementation

We implemented Mayflower in C++, with the exception of the Flowserver which runs as an application in the Java-based Floodlight controller. Our prototype consists of 7,500 lines of C++ code and 3,700 lines of Java code. The replica-path selection function in the Flowserver is exposed as an RPC service. The Flowserver is implemented as a stand alone service and can be integrated with any distributed application through its RPC framework.

The Nameserver stores the filesystem information in a LevelDB [171] key-value store. Files are divided in 128 MB chunks and replicated on three servers. Our choice of block sizes are based on HDFS's default block size [116]. The default replica placement strategy follows an HDFS's placement approach: Two replicas are placed in the same rack as the client, and the third replica is placed in another randomly selected rack.

3.4 Evaluation

We evaluate the effectiveness of Mayflower's replica-path selection using micro-benchmarks that compare it with several other replica-path selection schemes. We also compare the performance of our prototype with HDFS.

3.4.1 Experimental Setup

We conducted our experiments by emulating a 3-tier datacenter network topology using Mininet [170]. As emulating a complete datacenter network in a single machine imposes network size and bandwidth limitations, we partitioned our virtual network into several slices, and distributed these slices across a cluster of 13 machines. This allowed us to emulate a network with 1 Gbps edge links. Each machine in our cluster consists of a 64 GB RAM, a 200 GB Intel S3700 SSD, and two Intel Xeon E5-2620 processors having a total of 12 cores running at 2.1 GHz. The machines are connected through a Mellanox SX6012 switch via 10 Gbps links.

Our testbed consists of 64 virtual hosts distributed across four pods. In our topology, a pod consists of four racks connected to two common aggregation switches. Each pod is

emulated using three physical machines. Two machines emulate the hosts and the top-of-rack switches while the third machine emulates the aggregation switches belonging to that pod. The pods are connected through two core switches that are emulated in a separate dedicated machine. We stitched these network slices together through IP and MAC address translation. Our emulated network has 1 Gbps edge links, and oversubscription is achieved by varying the capacity of higher tier links.

Traffic Matrix: We evaluate Mayflower's performance using several synthetic workloads. Our workloads have the following properties: (1) job arrival follows the Poisson distribution, (2) file read popularity follows the Zipf distribution [17] with the skewness parameter $\rho = 1.1$, and (3) the clients are placed on virtual hosts based on the staggered probability described by Hedera [10]; a client is placed in the same rack as the primary replica with probability R, in another rack but in the same pod with probability P, and in a different pod with probability $O = 1 - R - P$. The replica placement follows conventional constraints of fault tolerance domains. The primary replica is placed in a randomly selected server, the second replica is placed in the same pod as the primary, and the third replica is placed in a different pod.

3.4.2 Selection Schemes

We compared Mayflower's replica-path selection with four other schemes that are a combination of static and dynamic replica selection with various network load balancing methods:

Nearest with ECMP: In this scheme, the closest replica to the client is selected, and the flows are spread across redundant links using ECMP. This represents the schemes where only static information is used for both replica selection and network load balancing.

Sinbad-R with ECMP: In this scheme, a replica is selected based on the current network state by using our read-variant implementation of Sinbad [60], which we call Sinbad-R. Sinbad was originally designed for dynamic replica selection for file write operations. It collects end-hosts' network load information to estimate the network utilization for higher tier links.

In order to support file-read operations, Sinbad-R estimates the link utilization for the traffic travelling up the network hierarchy (i.e., from edge to core) instead of down the hierarchy, which is the case for the original Sinbad scheme. This modification is necessary because file-reads and file-writes have opposing data-flow directions.

Dynamic path selection: To evaluate the effectiveness of Nearest and Sinbad-R replica selection combined with dynamic network load balancing, we coupled them with Mayflower's

network flow scheduler. However, unlike Mayflower's combined replica and path selection, the optimization space is limited to the pre-selected replica source for these schemes. We refer to these methods as Nearest-MF and Sinbad-R-MF in our results.

Mayflower replica-path selection: Mayflower has two replica selection methods: Mayflower-B and Mayflower-P. In Mayflower-B, the Flowserver selects one replica, where as in Mayflower-P, it selects up to two replicas for each requests. The client downloads data from the selected replicas concurrently.

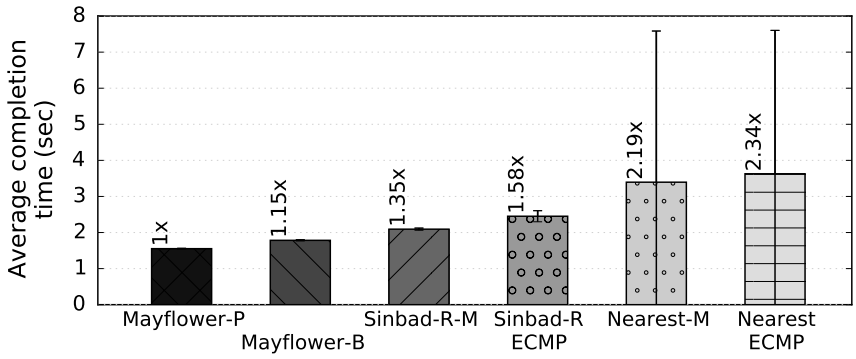
3.4.3 Replica-Path Selection Performance

We first evaluate the performance of Mayflower's replica-path selection using micro-benchmarks. In these experiments, we run a simple client/server application to emulate read jobs. For each read job, a client contacts the Flowserver for replica selection and path setup. The client then downloads 128 MB from the server selected by the Flowserver. Following the design described in Section 3.2.3, the client further breaks down the 128 MB block into 32 MB chunks, and downloads chunks from the selected replicas concurrently. When it finishes downloading a chunk from a replica, it will issue a request for the next chunk from the same replica until all four chunks have been downloaded. To avoid disk bottlenecks in these experiments, the data is stored in memory.

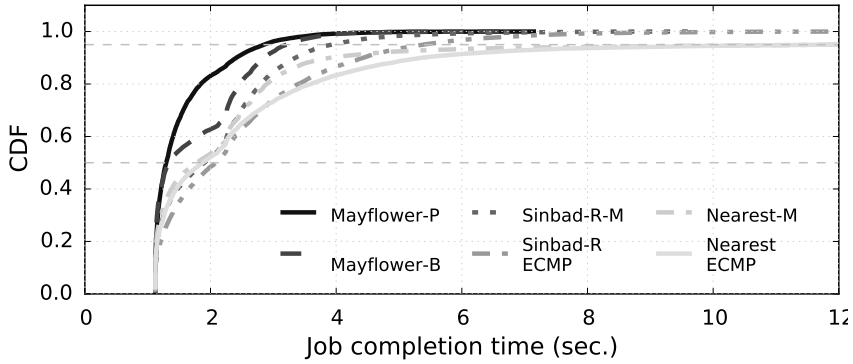
Figure 3.3 shows the performance of Mayflower's replica-path selection in comparison with the other methods (§ 3.4.2). The bars show the average completion time of the read requests and the error bars represent 95% confidence intervals. Mayflower's completion time also includes the query latency to the Flowserver for replica-path selection, which is approximately 4 ms[1] in our experiments. The parameters for this experimental workload iconsists of $\lambda = 0.1$, $\rho = 1.1$, and a client locality distribution of ($R = 0.5$, $P = 0.3$, $O = 0.2$). The results show that Nearest-ECMP, which is commonly used in current deployments, has more than 2.3x the average completion time compared to Mayflower-P. It also shows that replacing ECMP with a dynamic path selection scheme (Nearest-M) only provides marginal improvements over ECMP.

The Nearest replica selection-based approaches perform poorly because the replica selection is static and oblivious to the network state. As half of the clients are in the same rack as the primary replica, the clients have only one replica and path option. Therefore,

[1]Most of the replica-path selection latency is due to installing rules in switches to setup the network path. However, typical file read operations take much longer than replica-path selection latency because block sizes are large, typically between 64 MB and 1 GB, and they take several seconds to read [60].



(a) Average completion times.



(b) CDF of the jobs completion times.

Figure 3.3: Average job completion times of different replica and path selection methods.

the edge link of the primary replica can become congested, resulting in higher completion time.

Sinbad-R-M and Sinbad-R ECMP illustrate that dynamic replica selection can reduce average completion time compared to static replica selection. However, because they select the replicas and paths independently, the Sinbad-based schemes still have an average completion time of more than 1.3x of Mayflower-P. Finally, by intelligently downloading from up to two replicas in parallel, Mayflower-P has 14% lower average completion time than Mayflower-B.

Figure 3.3(b) shows the CDF of job completion time for the same experiment. It illustrates that Mayflower-P has a significantly shorter tail completion time than the other schemes. Its 95^{th} percentile job completion time is 2.81 seconds compared to 3.21 seconds

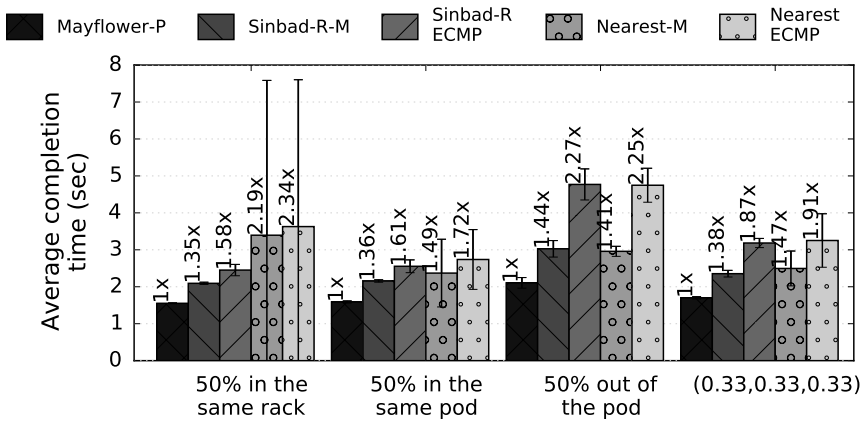


Figure 3.4: Average job completion times with different client locality distributions.

for Mayflower-B and 3.82 seconds for Sinbad-R-M. These results suggest that, in the case where the paths to all replicas are partially congested, Mayflower-P can still achieve a low job completion time by reading from two replicas over two paths with different bottleneck links.

3.4.4 Impact of Client Locality

Figure 3.4 shows the effectiveness of Mayflower's replica-path selection with different client locality distributions. The bars are grouped according to the probability distributions (R, P, O) of the clients being in the same rack R, in the same pod P and in another pod O relative to the location of the primary replica. The bar groups in the Figure have the probability distributions (0.5, 0.3, 0.2), (0.3, 0.5, 0.2), (0.2, 0.3, 0.5) and (0.33, 0.33, 0.33) from left to right. As Mayflower-P consistently performs between 10 to 15% better than Mayflower-B, we omit the results of Mayflower-B from the remaining graphs in this section to improve their readability.

The results show that as we reduce client locality from (0.5, 0.3, 0.2) to (0.3, 0.5, 0.2) causing a larger portion of the traffic to leave the rack, the nearest replica selection schemes surprisingly show a reduction in their average completion time. This is due to a greater dispersion of traffic between replicas, because a client is equally distant from its two closest replicas for a larger percentage of its requests. For the other schemes, this reduction in client locality causes a small increase in average completion time, with the Sinbad and

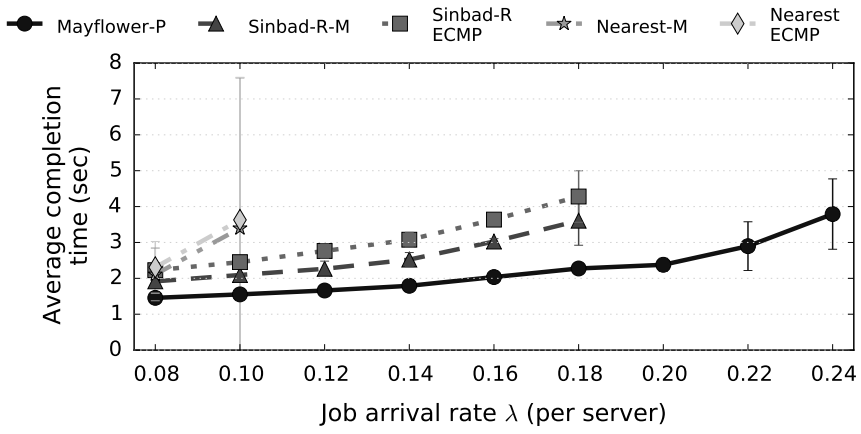


Figure 3.5: Increase in average completion times due to higher job rate.

Nearest-based schemes having an average completion time of more than 1.36x compared to Mayflower-P.

Further reducing the client locality to (0.2, 0.3, 0.5) results in significant increase in average completion time for both Sinbad and Nearest-based approaches. Moreover, the average completion time of ECMP-based approaches increases by nearly 2.27x with this reduction in locality. This is because, with poor client locality, a larger fraction of the requests must traverse the heavily utilized core links. Therefore, efficient load balancing of the core links through dynamic path selection becomes far more important. Finally, the last bar group shows that, in the case where a client distribution is (0.33, 0.33, 0.33), the result is in between the results from (0.3, 0.5, 0.2) and (0.2, 0.3, 0.5).

3.4.5 Impact of Job Rate

In this experiment, we vary the job rate to determine the impact of system load on the completion time of the different replica and path selection methods. Job arrival is modelled as a Poisson process and the job arrival rate (λ) specifies the rate for each server. Thus the job arrival rate of 0.08 means that on average 6 new read jobs are started every second in a 64 node system.

Figure 3.5 shows the results for the common scenario in which a majority of the clients are situated in the same rack as the primary replica of the requested file. We find that, all the methods perform equally well at lower job rate because of the light burden on the

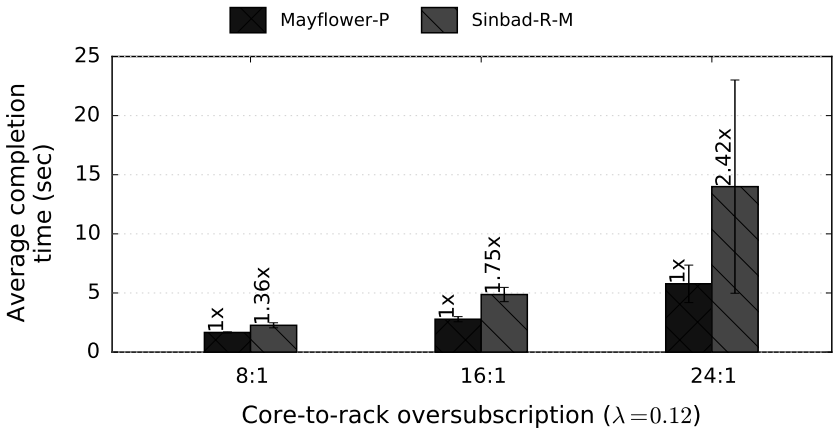


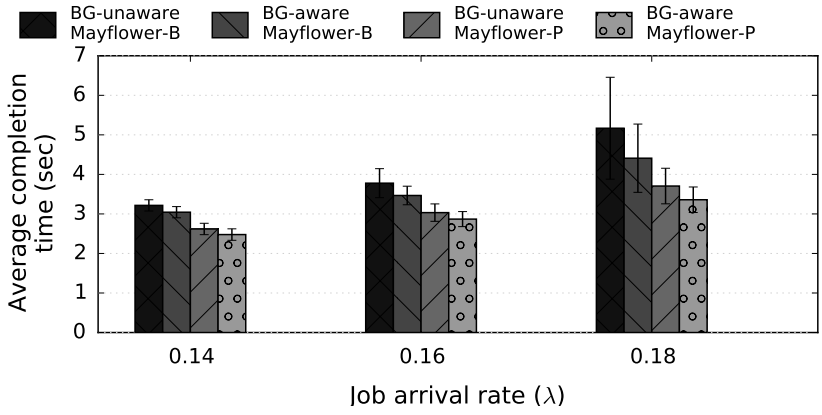
Figure 3.6: Impact of network oversubscription on the average completion time.

system. At higher job rates, links become congested and the performance degrades quickly for all the methods, however, Mayflower has a slower growth in the completion time. The relatively small increase in completion time of Mayflower at higher job rates suggests that Mayflower is more effective at avoiding congestion points in the network than the other methods.

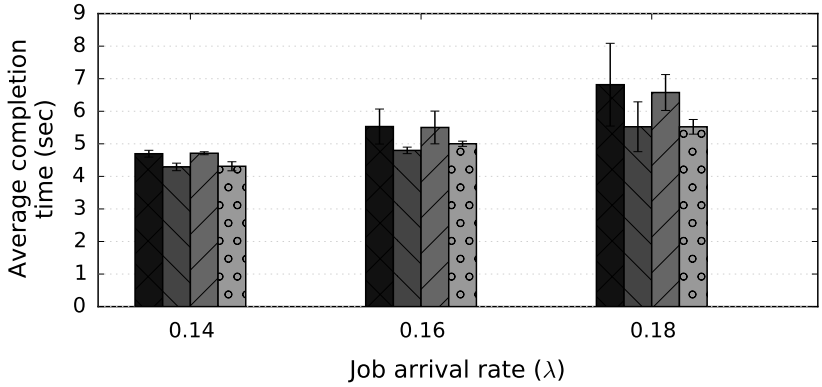
3.4.6 Impact of Oversubscription

The experiments from the previous sections were conducted with an 8:1 oversubscription ratio. Figure 3.6 shows the performance of Mayflower with other network oversubscription ratios. There is no oversubscription within a rack in all of our tested configurations. Due to limitations in our test infrastructure, we are unable to experiment with lower oversubscription while maintaining a 1 Gbps edge link capacity. In this and subsequent Figures, we only show the results for Mayflower and Sinbad-R-M as those are the best among all the different methods.

Higher oversubscription increases the chances for network congestion. Both Mayflower and Sinbad-R-M have a higher average completion time in more oversubscribed networks. However, Mayflower performs better under higher oversubscription than Sinbad-R-M, difference of 2.4x, as it has more replica and path options.



(a) Completion time of Mayflower file read requests.



(b) Completion time of the flows generated in the background.

Figure 3.7: The performance of both Mayflower requests and background traffic.

3.4.7 Effect on Background Traffic

In this experiment, we evaluate the performance of Mayflower-B and Mayflower-P in the presence of other flows in the network. We also evaluate the effect of Mayflower flows on background flows. To generate the background traffic, we run another client/server application in which the clients download data from a server selected randomly using the locality distribution of (0.2, 0.3, 0.5). The size of the background flows are based on a truncated normal distribution with the parameters $N(96, 64)$ and with a lower and upper limit of [32, 256]. We use the job arrival rate $\lambda = 0.08$ for the background flows and vary the arrival rate of Mayflower file read requests, shown by the different bar groups in Figure 3.7.

Figure 3.7(a) shows the completion time of Mayflower read requests. The first bar, background unaware (BG-unaware) Mayflower-B, shows the completion time when Mayflower-B runs while being oblivious to the background flows. The second bar, background aware (BG-aware) Mayflower-B, shows the completion time when Mayflower-B is aware of the background flows. Without background flow awareness, Mayflower-B's bandwidth estimates are inaccurate, which can result in selecting paths that are congested with background flows. When Mayflower-B is aware of the background flows, it can accurately model the utilization of each link, which allows it to avoid congested links and reduce average read completion time compared to BG-unaware Mayflower-B. Compared to Mayflower-B, Mayflower-P is more effective at avoiding congested links by dispersing the load across two partially congested paths with different bottleneck links.

Figure 3.7(b) shows the completion time of the background flows. The Figure highlights the importance of better replica and path selection as it affects the performance of both the background flows and the file read operations. Poor path selection hurts all the elephant flows in the network. This is due to the interdependence between the network and the applications, suggesting the need for increased coordination between the endpoint applications and the network control plane. By making Mayflower aware of the background flows, our results show that we can reduce the completion of both Mayflower request and background flows compared to using background-unaware Mayflower.

3.4.8 Effect of Data Size

Figure 3.8 shows the performance of Mayflower-P with different block size and data rate combinations. Mayflower-P performs consistently better for both small and large block sizes. For smaller block sizes, the average completion time is shorter than the stats collection intervals. Therefore, Mayflower relies primarily on bandwidth utilization estimates to determine flow durations, which is in turn used to determine which replica-path it selects. As a result, for smaller block sizes, there is a smaller performance gap between Mayflower-P and the other methods because of its higher reliance on bandwidth estimates.

3.4.9 Comparison With HDFS

Figure 3.9 shows the performance of our Mayflower prototype compared with HDFS. We used the same network setup and traffic matrix that were used for our replica-path selection evaluation. In this experiment, we use a block size of 256 MB, a client locality distribution of (0.5, 0.3, 0.2) and job arrival rates from $\lambda = 0.06$ to $\lambda = 0.08$. Instead of running

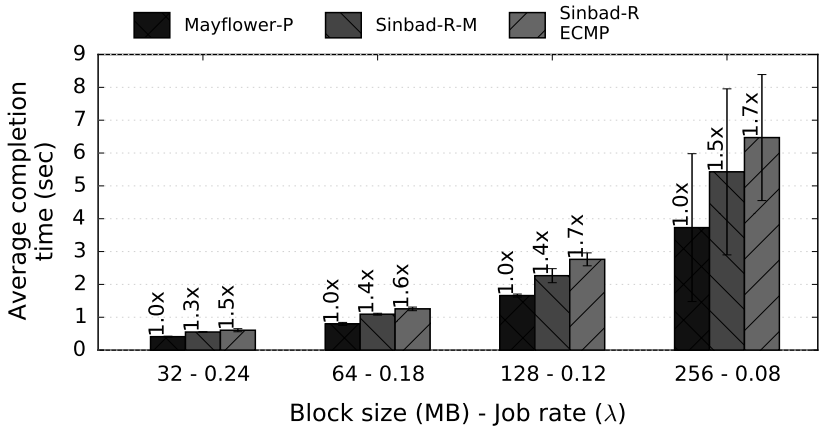


Figure 3.8: Performance impact of the block size.

a client/server application that only simulates file reads and writes, this experiment uses our Mayflower prototype implementation. We configured HDFS to use rack awareness for replica selection – HDFS selects the replica in the same rack where the client is located, if any such replica exists. For network flow scheduling, we performed HDFS experiments with both ECMP and Mayflower flow scheduling. For file placement, we use the same primary replica location for both Mayflower and HDFS.

Mayflower's experimental results are consistent with our simulation results. Mayflower shows a small increase in the completion time as the job arrival rate increases. In contrast, the completion times for HDFS grow rapidly with an increase in the job rate.

3.5 Discussion

Mayflower enables efficient utilization of available replicas and network paths, which reduces in-network congestion and increases bandwidth for file read operations. Its joint replica-path selection approach can be used to improve the performance of read operations in the cases where multiple replicas and network paths are available and a replica is read over the network. Nonetheless, the current design of Mayflower limits its use to a narrow set of deployment scenarios. Below we discuss some of the applications that can benefit from using Mayflower. Moreover, we discuss the limitations of Mayflower's current design and some of the ways to improve the design.

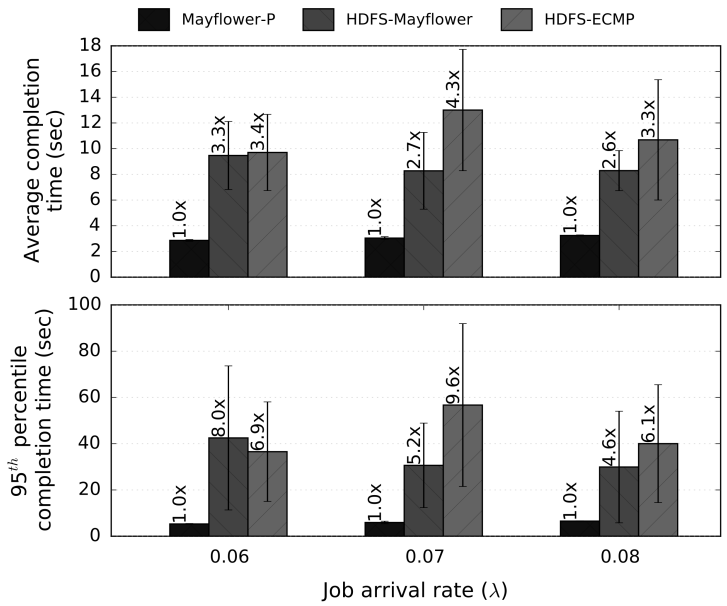


Figure 3.9: Mayflower prototype versus HDFS.

3.5.1 Append-only Semantics and Target Applications

Following the model of HDFS, Mayflower supports file modification only through append operations. As HDFS is a widely used filesystem, Mayflower can be used for many applications despite restricting writes to only file appends. On-demand videos, media encoding, copying virtual machine images in a cloud environment, distributed joins, ad-hoc queries in decisions support systems, and matrix operations are a few examples that require reading data over the network and can benefit from using Mayflower as the underlying filesystem.

3.5.2 Alternative Approach for Network Modeling and Control

Although Mayflower demonstrates high efficacy in reducing latency from network congestion, its use in practice depends on access to the network control plane. A network's control plane is not always accessible to developers and applications. Even if it is accessible, frequently updating rules in switches and periodically fetching stats from the edge switches add strain in switches and increase network overhead in large deployments. In retrospect, it might be better in some deployment scenarios to use an alternative approach where the network is monitored from endpoints. The Flowserver can employ a monitoring service at

endpoints that periodically send stats observed at the endpoints to the Flowserver. The Flowserver can combine the received stats information with its network path information for read jobs and background flows to accurately develop its network model.

In addition to providing network stats, the endpoints can provide remaining flow-size information to the Flowserver. The Flowserver receives expected read size from clients at the beginning of their read operations, and monitors the remaining size of the flows by collecting flow stats from edge switches. It uses the remaining flow size information to calculate replica-path costs for new read jobs. Instead of collecting flow stats from edge switches, the Flowserver can gather transferred bytes between endpoints from its monitoring services at the endpoints. Like CODA [294], a daemon can run on each endpoint that monitors the flows entering or leaving the endpoint and periodically send the flow stats to the Flowserver. The Flowserver can use this information to create a network model based on the information collected from endpoints, and synthesize that network model with network path information for replica-path selection.

Currently the Flowserver controls network paths for read jobs by installing new rules in switches along the paths for each read job. In large deployments where the number of read jobs is high, the path setup can have significant overhead and load on switches. An alternative approach for large and highly dynamic deployments can be to pre-setup network paths and assign the paths to reads jobs through Mayflower's replica-path selection. Network paths can be setup and assigned using five-tuple parameters and network address translations along the paths, or using tunneling protocols. Setting up paths beforehand significantly reduces the overhead of installing and updating rules in switches.

3.5.3 Macro Benchmarks

A workload has several parameters that impact the system performance differently, e.g., popularity of files and variation in the size of reads. Our evaluation of Mayflower using micro-benchmarks show the behavior of the system with various workload parameters. However, our tests do not evaluate the performance with the workloads that are specific to an applications such as data analytics using Hadoop and Spark. Macro-benchmarks provide additional performance analysis and reveal patterns and workloads that are not accurately modeled through synthetic workloads, such as frequency of deletes and their interleaving with reads/writes. For future work, we would like to perform end-to-end performance analysis of Mayflower's replica-path selection by integrating it with HDFS's read operations. Widely used TPC-H, TPC-DS [275, 227] and SPC-2 [258] benchmarks can be used to evaluate our approach with more practical workloads.

3.5.4 Primary Replica Failures

This thesis focuses on optimizing read operations in distributed filesystems using the network/filesystem co-design approach. Therefore, we have not fully implemented the failure handling of the primary replica host. However, we believe that the design and performance of replica-path selection and read operations are not affected when failure handling of the primary host is implemented. One approach is to use an external fault-tolerant configuration service, such as ZooKeeper, to detect the failure of a primary node and decide its replacement among the backup nodes. The clients have to retry their requests that fail due to the primary node's failure.

3.6 Summary

In this chapter, we presented the design and evaluation of Mayflower filesystem. Following the infrastructure-aware design approach, Mayflower is co-designed with SDN control plane to optimize file read operations. Mayflower uses a novel replica and network path selection algorithm that synthesizes information collected from the network and filesystem clients for replica-path selection. Our evaluation of Mayflower using simulations and a deployment on an emulated network showed that existing systems require 1.3x the completion time compared to Mayflower using common datacenter workloads. Higher performance of Mayflower as compared to other approaches highlights the effectiveness of using infrastructure-awareness in improving the performance of distributed storage systems.

Chapter 4

Canopus Consensus Protocol

In this chapter, we describe our contribution to improve the performance of distributed consensus protocols using a network-aware approach. Many distributed storage systems use consensus protocols to replicate data with strong consistency in a geo-distributed deployment. Moreover, big data applications heavily use coordination systems (e.g., Chubby and ZooKeeper) to simplify classic distributed problems such as leader election and configuration management.

Demands for strongly consistent storage systems are increasing as they simplify application development and reasoning for correctness during failures [7]. Strong consistency in geo-distributed storage systems is usually ensured by building a replicated state machine through distributed consensus protocols. However, commonly used distributed consensus protocols, such as Paxos [160, 161] and Raft [210], do not scale to a large number of nodes with high throughput. Consensus across many geo-distributed nodes is needed for planet-scale distributed storage systems and emerging applications such as distributed ledgers. For instance, storing highly popular objects with strong consistency across geo-distributed sites is reported to be an open challenge at Facebook [7]. Popular objects are accessed with high frequency and they demand high throughput per object or shard. Replicating data across tens of datacenters with multiple copies in each datacenter for high performance place significant scalability and high throughput demands.

Despite several efforts to optimize the performance of consensus protocols through multi-leader and leaderless approaches [184, 196, 35, 24, 204, 295, 43], existing consensus protocols do not yield the throughput of millions of operations per second at WAN scale. A fundamental challenge to achieve consensus at large scale with high throughput is to distributed load across all nodes and efficiently utilize all available resources. How-

ever, existing goe-distributed consensus systems, e.g., EPaxos, Mencius, and S-Paxos, are not designed around the underlying cloud infrastructure. Therefore, these systems do not efficiently utilize the network and other infrastructure resources. They send redundant messages over higher oversubscribed links that waste CPU and network resources. Moreover, existing geo-distributed consensus systems, such as D-Paxos, WPaxos, and WanKeeper concentrate load to a few nodes in the consensus cluster. As a result, they under-utilize the available resources. We address these problems in our Canopus [237] protocol. Canopus scales consensus to a large group of geo-distributed nodes by following the infrastructure-aware design approach. Our objective of building Canopus is to scale consensus to tens to hundreds of geo-distributed nodes. A scalable consensus protocol has applications in scalable storage systems that provide strong consistency and distributed ledgers that achieve end-to-end security through hardware assisted trusted computing environments [67]. This chapter present the detailed design and evaluation of Canopus.

4.1 System Assumptions

Canopus is designed based on the following assumptions:

Network topology: Canopus makes only two minor assumptions about the nature of its data center network. These are likely to hold for the vast majority of modern data center networks:

1. **All machines running Canopus in the same datacenter are hosted in the same rack, and are dual-connected to Top-of-Rack (ToR) switches** that provide low-latency connectivity to all the machines in its rack. If the number of Canopus nodes in a datacenter exceeds what can be hosted in a single rack, we assume that we have control over the physical location of the Canopus' nodes within a datacenter, including the location in the racks or whether Canopus runs in a VM or a physical machine.
2. **ToR switches are connected to each other through a hierarchical network fabric**. In general, we expect latency to increase and bandwidth to decrease as hop count between the pair of communicating nodes increases.

Although Canopus will perform correctly (i.e., either achieve consensus or stall) without the next two assumptions, for acceptable performance, we also assume that:

3. **Full rack failures are rare.** More precisely, we assume that although individual machines, ToR switches, and/or links may fail, it is rarely the case that all machines in a rack are simultaneously unreachable from off-rack machines. Single-failure fault tolerance is easily achieved by using dual-NIC machines connected to two ToR switches on each rack. In the rare cases of rack failures, Canopus halts until the racks have recovered. Moreover, rack failures do not cause the system to enter into an unrecoverable state. This is because the nodes persistently store their protocol state, and our system can resume when enough nodes in the failed racks recover.

4. **Network partitions are rare.** A network partition within a rack requires both ToR switches to simultaneously fail, which we assume is rare. Similarly, we assume that the network provides multiple disjoint paths between ToR switches, so that the ToR switches are mutually reachable despite a link or switch failure. Finally, we assume that there are multiple redundant links between data centers, so that a wide-area link failure does not cause the network to partition.

Many modern datacenters use network topologies that meet these two assumptions [254, 92], including Clos-like topologies such as fat-tree [9], VL2 [108], and leaf/spine networks. These failure assumptions simplify the design of Canopus and allow us to optimize Canopus for the common case.

In the unlikely event of a network partition, we believe it is reasonable for a consensus protocol to stall while waiting for the network to recover, resuming seamlessly once the network recovers. Indeed, this is the only reasonable response to a network partition where a majority of consensus participants do not reside in any single partition. We believe that, given both the rarity of network partitions and the severity in which they affect applications and services, it is not worthwhile for a consensus protocol to provide limited availability at the expense of a significant increase in protocol complexity. Planned shutdown of a rack is possible and does not result in any loss in availability.

5. Client-coordinator communication: In Canopus, as in many other consensus systems, clients send key-value read and write requests to a Canopus *node*. We assume that a client that has a pending request with a particular node does not send a request to any other node. This is to allow serialization of client requests at each node. If a client were to concurrently send a write and a read request to two different nodes, the read request could return before the write request even if the read was sent after the write, which would violate program order.

6. Crash-stop failures: We assume that nodes fail by crashing and require a failed node to rejoin the system using a join protocol. The Canopus join protocol closely resembles the join protocol for other consensus systems [211]. Canopus detects node failures by using a method similar to the heartbeat mechanism in Raft.

4.2 Canopus

Canopus is a scalable distributed consensus protocol that ensures live nodes in a Canopus group agree on the same ordered sequence of operations. Unlike most previous consensus protocols, Canopus does not have a single leader and uses a virtual tree overlay for message dissemination to limit network traffic across oversubscribed links. It leverages hardware redundancies, both within a rack and inside the network fabric, to reduce both protocol complexity and communication overhead. These design decisions enable Canopus to support large deployments without significant performance penalties.

The Canopus protocol divides execution into a sequence of *consensus cycles*. Each cycle is labelled with a monotonically increasing cycle ID. During a consensus cycle, the protocol determines the order of pending write requests received by nodes from clients before the start of the cycle and performs the write requests in the same order at every node in the group. Read requests are responded to by the node receiving it. Section 4.3 will describe how Canopus provides linearizable consistency while allowing any node to service read requests and without needing to disseminate read requests.

Canopus determines the ordering of the write requests by having each node, for each cycle, independently choose a large random number, then ordering write requests based on the random numbers. Ties are expected to be rare and are broken deterministically using the unique IDs of the nodes. Requests received by the same node are ordered by their order of arrival, which maintains request order for a client that sends multiple outstanding requests during the same consensus cycle.

4.2.1 Leaf-Only Trees

During each consensus cycle, each Canopus node disseminates the write requests it receives during the previous cycle to every other node in a series of *rounds*. For now, we will assume that all nodes start the same cycle in parallel at approximately the same time. We relax this assumption in Section 4.2.4, where we discuss how Canopus nodes self-synchronize.

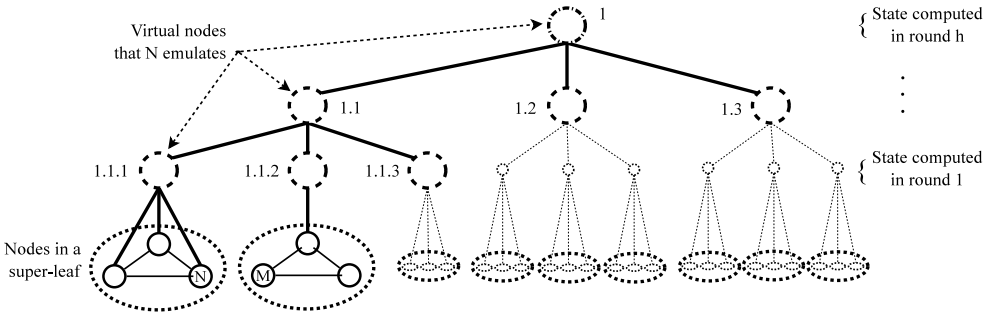


Figure 4.1: An example of a leaf-only tree (LOT). Only the leaf nodes exist physically and the internal nodes are virtual. A leaf node emulates all of its ancestor virtual nodes.

Instead of directly broadcasting requests to every node in the group, which can create significant strain on oversubscribed links in a datacenter network or wide-area links in a multi-datacenter deployment, message dissemination follows paths on a topology-aware virtual tree overlay.

Specifically, Canopus uses a Leaf-Only Tree overlay [13], that allows nodes arranged in a logical tree to compute an arbitrary global aggregation function. We adapt LOT for use in a consensus protocol; from this point on, we will always be referring to our modified version of LOT.

A LOT has three distinguishing properties:

i. Physical and virtual nodes: In LOT, only the leaf-nodes exist physically (in the form of a dedicated process running in a physical machine). Internal nodes are virtual and do not exist as a dedicated process. Instead, each leaf node emulates *all of its ancestor nodes*. For clarity, we denote a leaf node as a *pnode* and an internal node as a *vnode*.

ii. Node emulation: Each pnode emulates all of its ancestor vnodes. i.e., each pnode is aware of and maintains the state corresponding to all of its ancestor vnodes. Thus, the current state of a vnode can be obtained by querying any one of its descendants, making vnodes fault tolerant, and making vnode state access parallelizable.

iii. Super-leaves: All the pnodes located within the same rack are grouped into a single *super-leaf* for two reasons. First, this reduces the number of messages exchanged between any two super-leaves; instead of all-to-all communication between all the pnodes in the respective super-leaves, only a subset of the super-leaf nodes, called its *representatives*, communicate with another super-leaf on behalf of their peers. Second, because all the

pnodes in a super-leaf replicate their common parents' state, a majority of the super-leaf members need to simultaneously fail to cause the super-leaf to fail.

Figure 4.1 shows an example of a leaf-only tree consisting of 27 pnodes. There are three pnodes per super-leaf. The pnode N emulates all of its ancestor vnodes `1.1.1`, `1.1`, and `1`. The root node `1` is emulated by all of the pnodes in the tree.

We assume that during Canopus initialization, each node is provided with the identities and IP addresses of the peers in its own super-leaf.

4.2.2 Consensus Cycle

Each consensus cycle consists of h *rounds* where h is the height of the LOT. Each consensus cycle is labelled by a monotone non-decreasing *cycle ID*. Canopus ensures that nodes concurrently execute the same consensus cycle, which we further explain in Section 4.2.4. In the following sections, we describe the details of the messages exchanged in each round of a consensus cycle.

First round in a consensus cycle: In the first round of a consensus cycle, each pnode prepares a *proposal message* and broadcasts the message to the peers in its super-leaf using a reliable broadcast protocol, which we describe in detail in Section 4.2.3. A proposal message contains the set of pending client requests, group membership updates, and a large random number generated by each node at the start of the consensus cycle. This random number, called the *proposal number*, is used to order the proposals across the nodes in each round in the consensus cycle. A proposal message is also tagged with the cycle ID, the round number, and the vnode ID whose state is being represented by the proposal message. The group membership updates include the membership changes that happened before starting the current consensus cycle. Note that proposal messages sent in the first round are used to inform the receivers about the start of the next consensus cycle.

After receiving the round-1 proposal messages from all the peers in its super-leaf, each pnode independently orders the proposals according to their (random) proposal numbers. The sorted list of proposals represents the partial order of the requests in the current round and constitutes the state of the parent vnode of the super-leaf. A node finishes its first round after it has obtained proposal messages from all its super-leaf peers, and then immediately starts the next round in the consensus cycle.

Round-i in a consensus cycle: In each subsequent round, each node independently prepares a new proposal message, which is used to share the computed state from the previous round with other super-leaves. The list of requests in the proposal message is

simply the sorted list of requests obtained in the previous round. Similarly, the proposal number in the new message is the largest proposal number from the previous round. Thus, a proposal message for round i can be defined as $M_i = \{R'_{i-1}, N'_{i-1}, F'_{i-1}, C, i, v\}$, where R' is the sorted list of proposals from the previous round, N'_{i-1} is the largest proposal number in the previous round, F'_{i-1} is the set of membership updates received in the proposals from the previous round, C is the cycle ID, i is the round number, and v is the vnode ID, or the pnode ID in the case of the first round.

In round i, the pnodes compute, in parallel, the state of their height-i ancestor vnode. For example, the root vnode in Figure 4.1 has a height of 3, and therefore, its state is computed in the third round. As the current state of a vnode is just the merged and sorted list of proposals belonging to its child pnodes or vnodes, each super-leaf independently gathers the proposals of that vnode's children from their `emulators`, where a vnode's emulator is a pnode that is known to have the vnode's state; see § 4.2.6 for details.

To obtain the proposal belonging to a vnode, representatives from each super-leaf select one of that vnode's emulators and sends a proposal-request message to the selected emulator. For instance, to compute the state of the node `1.1`, representative node N in Figure 1 sends a proposal-request to emulators of the vnodes `1.1.2` and `1.1.3` – it already knows the state of the vnode `1.1.1`, which was computed in the previous round.

The proposal-request message contains the cycle ID, round number, and the vnode ID for which the state is required. In response to receiving a proposal-request message, the receiver pnode replies with the proposal message belonging to the required vnode. If the receiver pnode has not yet computed the state of the vnode, it buffers the request message and replies with the proposal message only after computing the state of the vnode. When the super-leaf representative receives the proposal message, it broadcasts the proposal to its super-leaf peers using reliable broadcast. These peers then independently sort the proposals, resulting in each node computing the state of their height-i ancestor. Note that representatives from all super-leaves issue proposal-requests in parallel, and all nodes in all super-leaves independently and in parallel compute the state of their height-i ancestor.

Completing a consensus cycle: The consensus protocol finishes after all nodes have computed the state of the root node, which takes time proportional to the logarithm of the number of Canopus pnodes. At this point, every node has a total order of requests received by all the nodes. The nodes log the requests in their request logs for execution. Each node then initiates the next consensus cycle with its super-leaf peers if they have received one or more client requests during the prior consensus cycle.

4.2.3 Reliable-Broadcast in a Super-Leaf

The nodes in a super-leaf use reliable broadcast to exchange proposal messages with each other. For ToR switches that support hardware-assisted atomic broadcast, nodes in a super-leaf can use this functionality to efficiently and safely distribute proposal messages within the super-leaf.

If hardware support is not available, we use a variant of Raft [211] to perform reliable broadcast. Each node in a super-leaf creates its own dedicated Raft group and becomes the initial *leader* of the group. All other nodes in the super-leaf participate as *followers* in the group. Each node broadcasts its proposal messages to the other nodes in its super-leaf using the Raft log replication protocol in its own Raft group. If a node fails, the other nodes detect that the leader of the group has failed, and elect a new leader for the group using the Raft election protocol. The new leader completes any incomplete log replication, after which all the nodes leave that group to eliminate the group from the super-leaf.

Reliable broadcast requires $2F+1$ replicas to support F failures. If more than F nodes fail in the same super-leaf, the entire super-leaf fails and the consensus process halts.

4.2.4 Self-Synchronization

So far we have assumed that all of the nodes start at the same consensus cycle in parallel. Here, we describe how the nodes are self-synchronized to concurrently execute the same consensus cycle.

Before starting a consensus cycle, a node remains in an idle state. In this state, its network traffic is limited to periodic heartbeat messages to maintain its liveness. A new consensus cycle only starts when the node receives outside prompting. The simplest version of this prompting is when the node receives a client request for consensus. A client request triggers the node to start a new consensus cycle and broadcast a proposal message to other peers in its super-leaf. The proposal message informs the peers that it is time to start the next consensus cycle, if it is not already under way. The representatives, in effect, send proposal-request messages to other super-leaves that trigger them to start the next consensus cycle, if they have not started yet.

Alternatively, a node, while in the idle state, may receive either a proposal message from its peers in its super-leaf, or a proposal-request message from another super-leaf. This indicates that a new cycle has started, and therefore, this node begins the next consensus cycle. In this scenario, the node's proposal message contains an empty list of client requests. As a result, the nodes are self-clocked to execute the same consensus cycle.

4.2.5 Super-Leaf Representatives

A super-leaf representative fetches the state of a required vnode on behalf of its super-leaf peers and broadcasts the fetched state to the peers using reliable broadcast (§ 4.2.3). We use a variant of Raft's leader election and heartbeat protocols [211] to allow nodes in a super-leaf to select one or more super-leaf representatives and detect representative failures. A representative failure triggers a new election to select a replacement representative.

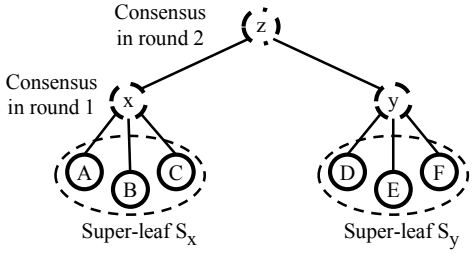
The representatives of a super-leaf are each individually responsible for fetching the states of the tree's vnodes. A representative can fetch the states of more than one vnode. However, to improve load balancing, we recommend that different representatives fetch the states of different vnodes. The vnode to representative assignment can be performed deterministically by taking the modulo of each vnode ID by the number of representatives in a super-leaf. Because representatives of a super-leaf are numbered and ordered (e.g., representative-0, representative-1, etc.), the result of the modulo operation can be used to determine which representative is assigned to a vnode without the need for any communication or consensus.

4.2.6 Emulation Table

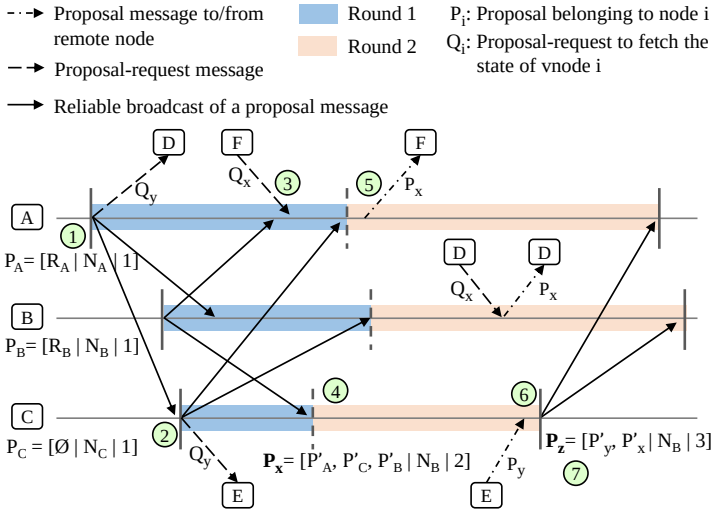
Thus far, we have not discussed how Canopus nodes actually communicate with other nodes. To initiate communication with an emulator of a vnode (a pnode in the sub-tree of the vnode) with a given ID, LOT maintains an *emulation table* that maps each vnode to a set of IP addresses of the pnodes that emulate that vnode. For example, for the LOT given in Figure 4.1, the table would map vnode 1.1 to the set of IP addresses of the nine pnodes that emulate this vnode.

We assume that at system initialization time, the emulation table is complete, correct, and made available to all pnodes. Subsequently, failed pnodes must be removed from the emulation tables maintained by all non-failed pnodes. Note that this itself requires consensus. Hence we maintain emulation tables *in parallel* with the consensus process, by piggybacking changes in the emulation table on proposal messages.

Specifically, in the first round of a consensus cycle, pnodes list membership changes (determined based on an within-super-leaf failure-detector protocol, such as through Raft heartbeats) in their proposal messages. Membership changes for a pnode include the list of pnodes that joined or left its super-leaf before the pnode started its current consensus cycle. At the end of the consensus cycle, all the pnodes have the same set of proposal messages, and thus the same set of membership changes are delivered to each pnode. Therefore,



(a) Six nodes are arranged in two super-leaves in a LOT of height 2.



(b) Timing diagram of the events occurring in the super-leaf S_x, which consists of the nodes A, B, and C.

Figure 4.2: Illustration of a consensus cycle in Canopus.

each pnode applies the same membership updates to the emulation table at the end of each consensus cycle. As a result, each pnode has the same emulation table and same membership view of LOT in each consensus cycle.

4.2.7 An Illustrative Example

We illustrate Canopus using an example in which six nodes are arranged in two super-leaves in a LOT of height 2 as shown in Figure 4.2(a). Nodes A, B, and C comprise super-leaf S_x, and the nodes D, E, and F comprise the super-leaf S_y. The nodes in S_x

and S_y are connected with the common vnodes x and y respectively. The vnodes x and y are connected with the root vnode z. In the first round, the states of vnodes x and y are computed, which establishes consensus within the super-leaves. In the second round, the state of the root vnode z is computed, which establishes consensus between super-leaves. Figure 4.2(b) shows the events occurring in the super-leaf S_x during the consensus cycle. The events are labelled in circles and explained as follows:

1. The consensus cycle starts: Nodes A and B start the first round of the consensus cycle. At the start of the cycle, nodes A and B have the pending requests R_A and R_B. The nodes prepare proposal-messages and broadcast them to their peers in the super-leaf.

Assume that the proposals of nodes A, B, and C are $P_A = \{R_A \mid N_A \mid 1\}$, $P_B = \{R_B \mid N_B \mid 1\}$, and $P_C = \{\phi \mid N_C \mid 1\}$, where R_i is the ordered set of pending requests on node i, N_i is the proposal number, and 1 is the round number. We omit the other information from the proposals for simplicity.

2. A proposal-request message is sent: Node C receives the proposal message from node A and starts its consensus cycle. Assuming that nodes A and C are the representatives for super-leaf S_x, these nodes send proposal-request messages Q_y to D and E respectively to redundantly fetch the state of vnode y. The nodes in S_x require the state of y to compute the state of the ancestor vnode z in the next round.

3. A proposal-request is received for an unfinished round: Node A receives a proposal-request message from F, which belongs to the next round of the current consensus cycle, asking for the proposal P_x. As node A has not yet finished its first round that results in P_x, it buffers the request and does not send any reply until it finishes its first round.

4. The first round is completed: Node C receives the proposal message P_B from B. As node C has received the proposals from all the peers in its super-leaf, it finishes its first round. Node C sorts the requests in the received proposals and its own proposal according the proposal numbers. The sorted list of requests is the consensus result from the first round, and comprises the current state of the vnode x. When nodes A and B finish their first round, they also have the same sorted list of requests.

To continue with the next round, the nodes prepare the proposal-messages that contain the sorted list of requests and the largest proposal number from the last round. In this example, the three nodes prepare the round-2 proposal $P_x = \{P_A, P_C, P_B \mid N_B \mid 2\}$, assuming $N_A < N_C < N_B$.

5. A proposal-request is served: Node A continues in the next round after finishing the first round and prepares the proposal P_x. It replies to the pending proposal-request Q_x, which it received from node F.

6. Coordinating the proposals: Node C receives the reply of its request Q_y for the round-2 proposal P_y, and reliably broadcasts P_y to other peers in its super-leaf. Assume that $P_y = \{P_D, P_E, P_F \mid N_F \mid 2\}$.

7. The consensus cycle is completed: After receiving all the required round-2 proposals, node C finishes the second round of the consensus cycle. At the end of the round, node C has the proposal $P_z = \{P_y, P_x \mid N_B \mid 3\}$, assuming $N_F < N_B$. As node C has now finished calculating the state of the root node, it finishes the consensus cycle. At the end of the cycle, node C has the consensus on the requests, which is $\{R_D, R_E, R_F, R_A, R_C, R_B\}$. Other nodes will have the same ordering of the requests when their second round is finished. Node C applies the requests in its copy of the commit log. Note that all node eventually agree on identical copies of this log.

4.3 Linearizability

Canopus provides linearizability by totally ordering both read and update requests. Interestingly, Canopus enforces global total order *without* disseminating read requests to participating nodes, significantly reducing network utilization for read-heavy workloads. Instead, Canopus nodes delay read requests until all concurrently-received update requests are ordered through the consensus process. Once the update request ordering is known, each Canopus node locally orders its pending read requests with respect to its own update requests such that the request orders of its clients are not violated. This trades off read latency for network bandwidth.

During consensus cycle C_i, the nodes buffer the requests received from the clients. The requests accumulated on a node N_i during cycle C_j form a request set S_i^j. In Canopus, the requests within a request set are sequential and follow the receive order. In contrast, the request sets across the nodes in cycle C_j, which we denote as S_*^j, are concurrent. To order the concurrent sets in S_*^j, Canopus executes the next instance C_{j+1} of the consensus protocol. The consensus process gives a total order of the concurrent sets in S_*^j. The total order of the concurrent sets translates to the linearized order of the requests received during the cycle C_j. Note that the requests in a request set are never separated. We are instead ordering the request sets to determine a total order of requests. At the end of the consensus process, each node has the same total order of write requests, and inserts its read requests in the appropriate positions to preserve its own request set order.

In this approach, each node delays its read requests for either one or two consensus cycles to provide linearizability. If a request is received immediately after starting a consensus

cycle C_j, the read request is delayed for two consensus cycles: The currently executing cycle C_j, and the next cycle C_{j+1} in which the request sets S_i^j are ordered and executed. If the request is received just before completing the currently executing cycle C_j, the read is delayed for only one consensus cycle C_{j+1}.

In a hypothetical situation where a client sends two requests to two different super-leafs, S_i and S_j, without waiting to receive the reply of the first request, the system does not ensure any ordering of the requests because the requests can be concurrent and ordered contrary to the expected ordering by the client. Moreover, in another situation, it may happen that the request to S_i succeeds while a simultaneous request to S_j stalls due to a particular timing of failures. For example, this can happen if the network is partitioned such that S_i receives all the required state to complete the ongoing consensus cycle whereas S_j does not receive the required state. We show in Canopus' correctness proof [236] that eventually all nodes stall during rack failures and network partitions until the fault is recovered. While the consensus process is stalled, new client requests are also stalled as all the requests need to be linearized through the consensus process. Thus, in all cases, the clients do not observe incorrect results.

4.4 Correctness Properties

Similar to Paxos and EPaxos, Canopus provides the properties of nontriviality, agreement, FIFO order of client requests, linearizability, and liveness[1]. Nontriviality and FIFO ordering are met by design. The following theorem ensures agreement, linearizability, and liveness:

Theorem 1: For a height-h LOT consisting of n super-leaves, all the live descendant-nodes $\widehat{E}_w^r$ of the root-vnode w that eventually complete the last round $r = h$ have the same ordered-set of messages.

$$\underset{i,j \in \widehat{E}_w^r}{\forall} \quad M_i^r = M_j^r \tag{4.1}$$

where M_i^r is the set of messages that a live node i has after completing the round r of the current consensus cycle.

We provide a proof sketch for the special case of the first consensus cycle for height-2 tree with n super-leaves. The complete proof [238] involves induction first on the number

[1]Please refer to Reference [196] for definitions.

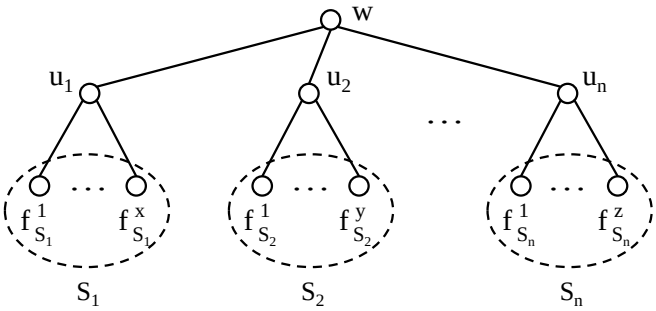


Figure 4.3: A LOT of height 2 with n super-leaves.

of cycles and then on the tree height. The complete proof is available in our extended technical report [236].

The proof assumes that:

A1 All nodes are initialized with the same emulation table and membership view.

A2 The network cannot be partitioned, messages are not corrupted, messages are eventually delivered to a live receiver, and that nodes fail by crashing. Node failures within a super-leaf are detected by using heartbeats and timeouts similar to Raft.

A3 The super-leaf-level structure of LOT does not change: Nodes may leave or join super-leaves, however, super-leaves are not added or removed.

A4 Reliable broadcast functionality is available within a super-leaf, which ensures that all the live nodes in a super-leaf receive the same set of messages.

Proof Sketch: Consider a height-2 LOT consisting of n super-leaves, as shown in Figure 4.3. Each super-leaf S_i has a height-1 parent vnode u_i. The height-1 vnodes are connected with the common root vnode w. A consensus cycle consists of two rounds for a LOT of height 2. In the first round, the nodes share the pending batch of update-requests within their super-leaf using the reliable broadcast functionality (assumption A4). The nodes sort the messages according to the proposal numbers received in the messages. At the end of the first round, each node $f^i_{S_j}$ in a super-leaf S_j has the same ordered-set of messages M^1_j, which represents the state of the height-1 ancestor u_j of the nodes in S_j. To order the messages in the next round, the largest proposal number received in the current round is selected.

In the second round each node $f^i_{S_j}$ calculates the state of its height-2 ancestor w. Therefore, the set of messages that $f^i_{S_j}$ requires to complete the second round is $M^2_j = \forall_{u \in C^2_j} \cup M^1_u$, where C^2_j is the set of children of the height-2 ancestor of any node in the super-leaf S_j.

By Canopus' design, a set of representatives of S_j fetch the missing state M^1_u of each vnode $u \in C^2_j$ from the live descendants of u. By assumption A1 and A2, the representatives will eventually succeed in fetching the state of u if there is at least one live super-leaf descending from u. If the representatives of S_j eventually succeed in fetching the states of all the vnodes $u \in C^2_j$, then by reliable broadcast within the super-leaf (assumption A4), all the nodes in the S_j have the same set of messages M^2_j that are required to complete the second round.

By assumptions A1 and A3, all the nodes have the same view of LOT in the first consensus cycle. Therefore, the representatives of each super-leaf fetch the same state of a common vnode u from any of the descendants of u. This implies that, for all the super-leaves for which the representatives have succeeded in fetching all the required states of height-1 vnodes, the nodes in the super-leaves have same set of messages, which allows the nodes to complete the second round. Thus,

$$\underset{i,j \in \widehat{E}^2_w}{\forall} \quad M^2_i = M^2_j \tag{4.2}$$

the nodes sort the messages according the proposal numbers received with the vnode-states. As the consensus cycle consists of two rounds for a LOT of height 2, equation 4.2 implies that all of the descendants of the height-2 root vnode that complete the first consensus cycle have the same ordered-set of messages. This shows that the live nodes reach agreement and eventually complete the consensus cycle, if a super-leaf does not fail.

If the representatives of a super-leaf S_j fail in fetching the state of at least one vnode $u \in C^2_j$ in the second round, then either (a) all the descendants of u have crashed or they do not have the state of u due to not completing the previous round, or (b) all the representatives of S_j have crashed, which implies that the super-leaf S_j has failed due to insufficient number of live nodes to elect a new representative. In either case, the nodes in S_j cannot complete the second round due to missing a required state, and the consensus process stalls for the nodes in S_j in the second round of the current consensus cycle.

Alternatively, if the nodes in another super-leaf S_k succeed in fetching all the required states, then the consensus process stalls for the nodes in S_k in the next cycle $c+1$ because the live nodes in S_j are stalled in the last cycle c. This shows that the consensus process

stalls for the live nodes due to a super-leaf failure and the nodes do not return a wrong result.

In the subsequent consensus cycles, the assumption A1 does not hold because the membership of LOT changes if a node fails or a new node joins a super-leaf. However, the detailed proof [238] using induction shows that the emulation tables are maintained and all the live nodes that eventually complete a cycle c have the same emulation table in cycle $c+1$. Therefore, Canopus satisfies agreement and liveness properties in any consensus cycle for height-2 LOT. We prove these properties for a LOT of any height-h using induction, for which the height-2 LOT serves as the base case.

4.5 Optimizations

In this section, we discuss two optimizations that enable Canopus to achieve high throughput and low read latency in wide-area deployments.

4.5.1 Pipelining

The completion time of a consensus cycle depends on the latency between the most widely-separated super-leaves. If only one cycle executes at a time, then high latency would reduce throughput as the nodes are mostly idle and are waiting for distant messages to arrive. Therefore, we use pipelining in Canopus to increase the throughput of the system. Specifically, instead of executing only one consensus cycle at a time, a node is permitted to participate in the next consensus cycle if any of the following events occur:

- The node receives a message belonging to the consensus cycle that is higher than its most recently started consensus cycle.
- A periodical timer expires, which serves as an upper bound for the offset between the start of two consensus cycles.
- The number of outstanding client requests exceeds the maximum batch size of requests.

With this change, a node can start exchanging messages with its peers that belong to the next consensus cycle even before the end of the prior cycle. In effect, this allows Canopus to

maintain multiple in-progress consensus cycles. However, log commits only happen when a particular consensus cycle is complete, and always in strict order of consensus cycles. This is no different than a transport layer maintaining a window of transmitted-but-unacked packets.

Two cases need to be addressed to maintain the total order of the requests. First, nodes must always start consensus cycles in sequence (i.e., they cannot skip a cycle). It may happen that a node receives a message belonging to cycle $C_{j \geq i+2}$ where C_i is the most recently started consensus cycle at the node. In that case, the node still starts the next cycle C_{i+1} instead of starting the cycle C_j. Second, nodes always commit the requests from consensus cycles in sequence. Due to parallelism and fetching proposals from different nodes in each cycle, it may happen that a node receives all the required messages to complete a cycle $C_{j \geq i+1}$ before it has completed the cycle C_i. However, to maintain the total order, nodes do not commit the requests from C_j until the requests from all the cycles before C_j have been committed.

4.5.2 Optimizing Read Operations

Canopus delays read operations to linearize them relative to the write requests across all the nodes in the system. Therefore, the read operations must be deferred to the end of the next consensus cycle, which is bounded by the round-trip latency between the farthest super-leaves. However, for read-heavy workload, it is desirable that the read operations are performed with minimal latency. Canopus can optionally support write-leases for a particular key during a consensus cycle to reduce the latency of read operations while preserving linearizability.

For any key, during any consensus cycle either a write lease is inactive, so that *no* writes are permitted to that key, and all nodes can read this key with no loss of consistency or *all* nodes have permission to write to this key (with a write order that is decided at the end of the consensus cycle) and no reads are permitted to this key. Any read requests made to a key during a consensus cycle i such that there is a write lease for the key is deferred to the end of the $(i+1)^{th}$ consensus cycle, to ensure linearizability.

Write leases require the following three modifications to Canopus:

1. Blocking client requests: Following the model in Paxos Quorum Leases [199], our read optimization restricts the clients to sending only *one* request at any one time. This is different than our earlier model, where the clients are permitted many outstanding requests at a time.

2. Write leases: Write requests are committed after acquiring the write-lease for the keys in the requests. Nodes piggyback lease requests with the proposal messages in the next consensus cycle C_{i+1}. At the end of the consensus cycle C_{i+1}, all the correct nodes that complete the cycle have the same set of lease requests. Therefore, all the correct nodes in the system can apply the lease in the same consensus cycle C_{i+p+1}.

3. Reads without delay: A node N performs a read operation for a key y immediately (reading results from committed consensus cycles) if the write-lease for y is not active in any of the currently ongoing consensus cycles. If a node receives a read request for y while the write-lease for y is currently active, the node delays the read request until the end of the next consensus cycle.

4.6 Evaluation

The focus of our evaluation is to compare the throughput and latency of Canopus with other competing systems at different deployment sizes. Our experiments are based on our prototype implementation of Canopus, which consists of approximately 2000 lines of Java code. We select EPaxos as a representative for state-of-the-art decentralized consensus approaches as EPaxos has been reported [196] to perform equal or better than other decentralized consensus protocols. We use the publicly available implementation [197] of EPaxos from the paper's authors for the evaluation. We also compare against ZooKeeper, a widely used coordination service that uses Zab, a centralized atomic broadcast protocol, to provide agreement. In order to provide a fair comparison with ZooKeeper, which includes additional abstractions that may introduce extra overhead, we created ZKCanopus, a modified version of ZooKeeper that replaces Zab with Canopus and performed all of our comparisons to ZooKeeper using ZKCanopus.

4.6.1 Single Datacenter Deployment

In this section, we evaluate the performance of Canopus and EPaxos when all the nodes are located in the same datacenter.

Experimental setup: We conducted our experiments on a three rack cluster where each rack consists of 13 machines. Each machine contains 32GB of memory, a 200GB Intel S3700 SSD, and two Intel Xeon E5-2620 processors. The machines in each rack are connected through a Mellanox SX1012 top-of-rack switch via 10Gbps links. Rack switches are connected through 2x10Gbps links to a common Mellanox SX1012 aggregation switch.

Thus, the network oversubscription is 1.5, 2.5, 3.5, and 4.5 for experiments with 9, 15, 21, and 27 nodes respectively.

For our ZooKeeper and ZKCanopus experiments, both systems were configured to write logs and snapshots of the current system state asynchronously to the filesystem. We use an in-memory filesystem to simplify our experimental setup. To ensure that using an in-memory filesystem does not appreciably affect our results, we perform additional experiments in which logs are stored to an SSD. The results show that, for both ZooKeeper and ZKCanopus, throughput is not affected and median completion time increases by less than 0.5 ms.

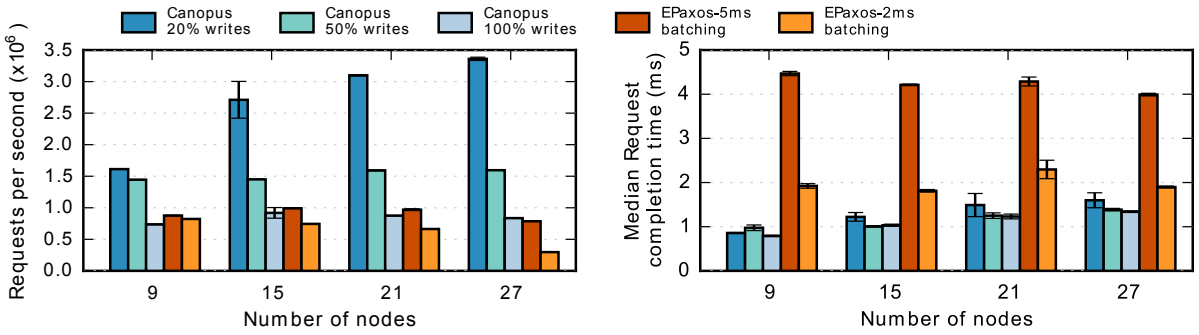
Each experiment runs for 50 seconds and is repeated five times. We discard the first and last 5 seconds to capture the steady state performance. The error bars in the graphs show 95% confidence intervals.

Workload: Linearized read/write requests of key-value pairs can model workloads for various applications such as strongly consistent key-value stores, distributed ledgers, and replicated state machines. Therefore, workloads in our experiments consist of a mix of read and write requests to key-value pairs. We vary the ratio of reads and writes in the workloads to get a general trend of system performance when an application uses Canopus as an external consensus service. Our experiments show performance without the overhead of the application that uses Canopus. However, evaluation using a complete application built on top of consensus would reveal end-to-end system performance that is closer to a real-world deployment. We leave the end-to-end performance analysis of Canopus using a complete application as future work.

Our experimental workloads are driven by 180 clients, which are uniformly distributed across 15 dedicated machines with 5 machines per rack. Each client connects to a uniformly-selected node in the same rack. The clients send requests to nodes according to a Poisson process at a given inter-arrival rate. Each request consists of a 16-byte key-value pair where the key is randomly selected from 1 million keys.

LOT configuration in Canopus: Canopus nodes are organized into three super-leaves in a LOT of height 2. To vary the group size, we change the super-leaf size to 3, 5, 7, and 9 nodes. Due to the size of our cluster, we did not run experiments with taller trees. In our current implementation, nodes reliably broadcast messages within a super-leaf without hardware assistance. Instead, they use the Raft-based reliable broadcast protocol described in Section 4.2.3 in which nodes communicate using unicast messages.

Performance metrics: The two key performance metrics we measure in our experiments are throughput and request completion time. We determine the throughput of a system



(a) Throughput

(b) Median completion time

Figure 4.4: Throughput and median request completion times of Canopus and EPaxos while scaling the number of nodes in the system.

by increasing the request inter-arrival rate until the throughput reaches a plateau. Our empirical results show that, for all of the tested systems and configurations, the plateau is reached before the median request completion time reaches 10 ms. To simplify our testing procedure, our experiments run until the request completion time is above 10 ms and we use the last data point as the throughput result for that run. For each run, we manually verify that we have reached the system's maximum throughput.

As it is difficult to determine the exact inter-arrival rate at which maximum throughput has been reached, together with the common operating practice of running critical systems below their maximum load, we report the median request completion time of the tested systems when they are operating at 70% of their maximum throughput. We believe this is more representative of the completion times that applications will experience when interacting with these systems compared to the median completion time at 100% load.

Comparison with EPaxos

In this experiment, we compare the performance of Canopus with EPaxos at different system sizes. Figure 4.4(a) shows the maximum throughput of the two systems. The first three bars in each group show the Canopus' throughput at 20%, 50%, and 100% write requests, and the last two bars show the throughput of EPaxos at its default batch duration of 5 ms, in which requests are delayed for up to 5 ms in order to collect a larger batch of requests, and a reduced batch duration of 2 ms. For EPaxos, we ensure there is 0% command interference to evaluate its performance under the best possible condition. As

EPaxos sends reads over the network to other nodes, its performance is largely independent of the write percentage; we show its results using the 20% write request workload.

The performance results show that with 20% and 50% write requests, Canopus provides significantly higher throughput than EPaxos with the performance gap increasing with larger system sizes. With 27 nodes and 20% write requests, Canopus provides more than 3 times the throughput of EPaxos with 5 ms batching. This is because Canopus can serve read requests locally without disseminating the request to the rest of the nodes in the group while still providing linearizable consistency. For the 100% write request workload, Canopus provides similar throughput to EPaxos with a 5 ms batching duration. This is in part due to network bandwidth being relatively abundant in a single datacenter deployment.

EPaxos with a 2 ms batching duration shows a significant drop in throughput as we increase the number of nodes in the system. This illustrates that EPaxos has scalability limitations when configured with smaller batch sizes.

Figure 4.4(b) shows the median request completion times of the two systems. From 9 to 27 nodes, Canopus' request completion time is mostly independent of write request percentage, and is significantly shorter than EPaxos.

EPaxos' high request completion time is primarily due to its batching duration. By reducing its batching duration to 2 ms, EPaxos' median request completion time reduces by more than half and is within 1 ms of Canopus' request completion time. However, as we saw previously, EPaxos relies on large batch sizes to scale. Therefore, our results show that EPaxos must tradeoff request completion time for scalability, whereas Canopus' read throughput increases with more nodes, its write throughput remains largely constant up to at least 27 nodes, and its median request completion time only marginally increases going from 9 to 27 nodes.

Comparison with ZooKeeper

In this experiment, we compare the performance of ZooKeeper with ZKCanopus, our Canopus-integrated version of ZooKeeper. This comparison serves three purposes. First, it compares the performance of Canopus with that of a centralized coordinator-based consensus protocol. Second, it shows the effectiveness of Canopus in scaling an existing system by eliminating the centralized coordinator bottleneck. Finally, these experiments evaluate the end-to-end performance of complete coordination services, instead of evaluating just the performance of the consensus protocols.

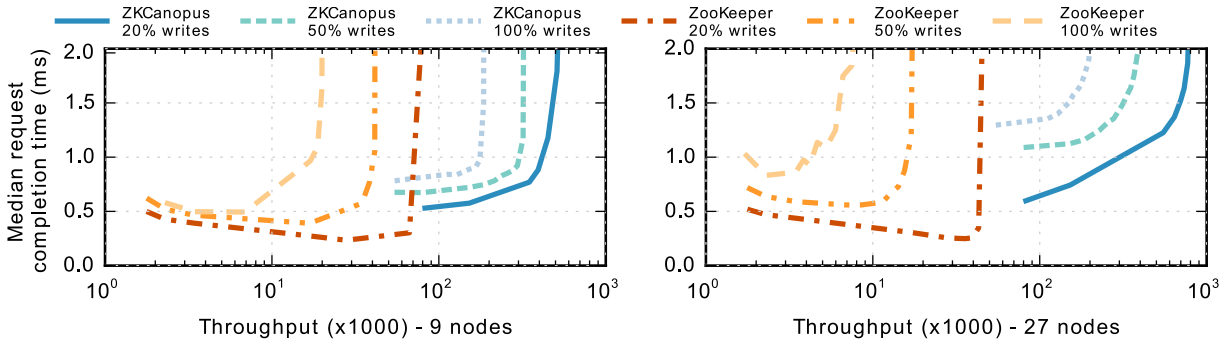


Figure 4.5: Throughput and median request completion times of ZKCanopus and ZooKeeper when the system consists of 9 nodes (left) and 27 nodes (right). Note that the x-axis is in log scale.

We configure ZooKeeper to have only five *followers* with the rest of the nodes set as *observers* that do not participate in the Zab atomic broadcast protocol but asynchronously receive update requests. This choice is mainly to reduce the load on the centralized *leader* node. Although observers do not provide additional fault tolerance, they can improve read request performance by servicing read requests. In the case of ZKCanopus, every node participates fully in the Canopus consensus protocol. We use one *znode* in these experiments and the clients read from and write to the same znode.

Figure 4.5 shows the throughput to median request completion time results for ZKCanopus and ZooKeeper with 9 and 27 nodes. For ZKCanopus, we do not show the request completion time at low throughputs in order to improve the readability of the graphs. The results show that, when the system is not loaded, ZKCanopus has a median request completion time that is between 0.2 ms to 0.5 ms higher than ZooKeeper. This is in part due to the additional network round trips required to disseminate write requests using a tree overlay instead of through direct broadcast. However, we believe the small increase in request completion time is acceptable given the significant improvement in scalability and throughput that ZKCanopus provides over ZooKeeper.

4.6.2 Multi-Datacenter Deployment

In this section, we evaluate the performance of Canopus in a wide area deployment. We run the experiments on Amazon EC2 cloud using three, five, and seven datacenters. Each datacenter consists of three nodes located in the same site. Each node runs on an EC2 c3.4xlarge instance, which consists of 30GB memory and Intel Xeon E5-2680 processor.

Table 4.1: Latencies (ms) between the datacenters used in the experiments. The datacenters are located in Ireland (IR), California (CA), Virginia (VA), Tokyo (TK), Oregon (OR), Sydney (SY) and Frankfurt (FF).

	IR	**CA**	**VA**	**TK**	**OR**	**SY**	**FF**
IR	0.2						
CA	133	0.2					
VA	66	60	0.25				
TK	243	113	145	0.13			
OR	154	20	80	100	0.26		
SY	295	168	226	103	161	0.2	
FF	22	145	89	226	156	322	0.23

Each datacenter has 100 clients that connect to a uniform-randomly selected node in the same datacenter and concurrently perform read and write operations at a given rate. The workload consists of 20% write requests, unless otherwise mentioned. The messages consists of 16-byte key-value pairs. The inter-datacenter latencies are given in the Table 4.1.

For Canopus, each datacenter contains a super-leaf, although the nodes might not be located in the same rack. We enable pipelining to overcome the long delays across datacenters. Each node starts a new consensus cycle every 5ms or after 1000 requests have accumulated, whichever happens earlier. For EPaxos, we used the same batch size as Canopus, and the workload consists of zero command interference. We enable latency-probing in EPaxos to dynamically select the nodes in the quorum. We disable thrifty optimization in EPaxos because our experiments show lower throughput with the thrifty optimization turned on.

Figure 4.6 shows the throughput and median request completion times of Canopus and EPaxos when deployed in three, five, and seven datacenters. Each datacenter consists of three nodes. The vertical lines in the graph show the throughput when the latency touches 1.5 times the base latency.

Scaling across the datacenters, Canopus is able to achieve about 2.6, 3.8, and 4.7 millions requests per second, which is nearly 4x to 13.6x higher throughput than EPaxos. Canopus is able to achieve high throughput because of its more efficient utilization of CPU and network resources. Unlike EPaxos, Canopus does not disseminate read requests across

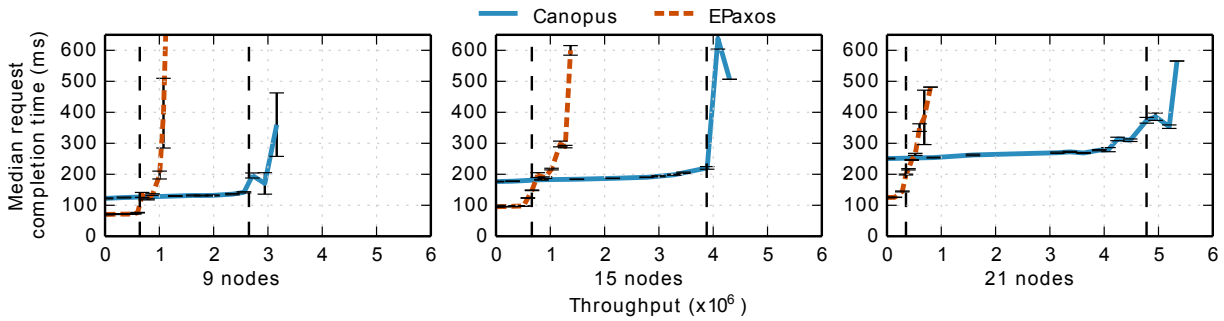


Figure 4.6: Throughput and median request completion times of Canopus and EPaxos in the multi-datacenter experiment with 20% writes workload.

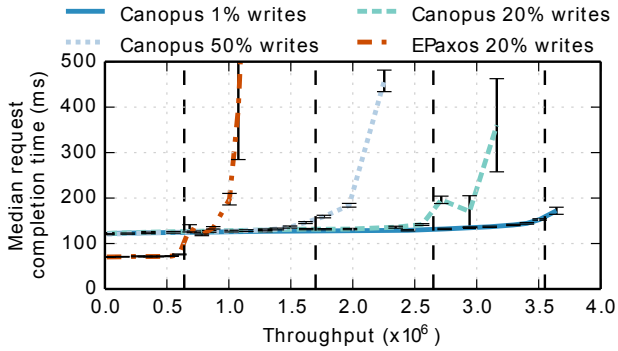


Figure 4.7: Performance of Canopus and EPaxos with different ratios of writes in the workload.

the nodes, not even the read keys, which reduces the number and size of messages exchanged between the nodes. For ready-heavy workloads, this significantly impacts the utilization of CPU and network resources. Furthermore, due to the hierarchical and network-aware design of Canopus, a proposal message is exchanged between each pair of super-leaves only once. This approach places a lower load on wide-area network links compared to the broadcast-based approach used in EPaxos.

Writes to Reads Ratio in the Workload

In this section, we evaluate the performance of Canopus and EPaxos with different ratios of writes in the workload. We run these experiments with nine nodes deployed in three datacenters. Figure 4.7 shows the results when the workload consists of 1%, 20%, and 50%

write requests. For EPaxos, our results show the same results with different write-ratios in the workloads. Therefore, we only show the results for 20% writes workload.

As expected, Canopus has higher throughput for more read-heavy workloads, reaching up to 3.6 million requests per second with 1% writes workload, as compared to the 2.65 millions requests per second with 20% writes workload. However, even with a write intensive workload consisting of 50% writes, Canopus achieves 2.5x higher throughput than EPaxos.

4.7 Discussion

Canopus is a simple and easily understandable consensus protocol that scales to tens of nodes with a high throughput of requests. Canopus achieves these properties at the expense of availability during catastrophic failures that cause a super-leaf to fail. Below we discuss a way that Canopus can be extended to tolerate super-leaf failures.

4.7.1 Tolerating Super-leaf Failures

Canopus achieves these properties at the expense of high availability. Failure of a majority of nodes within a super-leaf can cause a super-leaf to fail and stall the consensus service. Canopus trades-off high availability based on the assumption that complete rack failures are rare due to fault-tolerant design of modern datacenters [179, 180, 254, 247]. In the case of a rack failure or a network partitions, consensus process in Canopus is stalled until the fault is recovered. Thus, crashing of only a few nodes in a super-leaf can prevent all of the nodes from progressing. A simple work around could be to spread the nodes of a super-leaf across multiple racks such that the majority of racks belonging to the same super-leaf do not fail. In a cloud based deployment, a super-leaf can be spread across multiple zones in the same datacenter, where the zones are connected through dedicated high bandwidth network. However, these approaches risk network partitions within super-leafs and do not tolerate disconnection from a complete data center, which can result in stalling the consensus service.

Another way to tolerate super-leaf failures is to use a decentralized consensus protocol, such as Mencius, in the second round of each consensus cycle in Canopus. Super-leafs can elect representatives that can make a separate consensus group across all super-leafs. In the first round, nodes achieve consensus within super-leafs, resulting in the batches of requests to be committed in the ongoing consensus cycle. In the second round, super-leaf

representatives propose the first-round batches through consensus. If a super-leaf fails, consensus in the second round can bypass the failed super-leaf or commit no-ops on its behalf. A failed representative can be replaced with another node in the super-leaf. In this way, Canopus can tolerate failures of less than half of the super-leafs. However, we expect the performance to drop significantly because of additional communication overheads and network round trips due to using a consensus protocol in the second round.

Finally, following the same approach as RCanopus [143], an external membership service can be used to detect super-leaf failures instead of a consensus protocol in the second and subsequent rounds. During network partitions and super-leaf failures, the nodes contact the external membership service to detect super-leaf failures and resolve conflicts. The membership service achieves membership agreement between live nodes through consensus and decides to exclude the state of unresponsive super-leafs from the on-going consensus cycle. As a result, the consensus service is not stalled during super-leaf failures and network partitions. However, this design requires additional round-trip latency in the common case to ensure safety during failures in an asynchronous environment. The nodes must commit the current consensus cycle only after completing the next consensus cycle to detect super-leaf disconnections and use the external membership service to resolve any membership disagreements.

4.8 Summary

In this chapter, we have presented Canopus, a highly parallel, scalable, and network-aware consensus protocol. Canopus scales by leveraging the placement of machines in racks and the hierarchical design of datacenter networks. Canopus virtually group the nodes in the same rack and employs distinguished nodes in the group to fetch the remote state and disseminate within the group. The communication within the groups follow the hierarchical pattern of the network based on a leaf-only tree (LOT) overlay. Our evaluation results show that Canopus can perform millions of requests per second with 27 nodes in a single datacenter deployment. Furthermore, our Canopus-integrated version of ZooKeeper increases the throughput of ZooKeeper by more than 16x for read-heavy workloads. The high performance of Canopus shows the effectiveness of infrastructure-aware design approach in improving the performance of large scale distributed applications.

Chapter 5

Kawkab Distributed Filesystem

In this chapter, we present *Kawkab*, a storage system that we built from the ground-up to address data storage requirements in capital market use-cases. Many financial applications are built on top of financial data generated from capital markets, e.g., competitive pricing analysis, market data terminals [101], best execution regulatory reports, market-making pricing deviation monitoring, and liquidity provider pricing monitoring [256]. Capital markets generate large volumes of data at a high pace in the form of realtime data streams. Message-bursts in the streams can exceed millions of messages per second [79] due to the high frequency of financial events such as trading of stocks. The large volume and pace of streaming data demand high performance at all stages of realtime stream processing, including data storage. Storage systems are required to support peak ingestion rates of millions of messages per second [79] and latency bound of 20ms for recent data [101].

Public cloud infrastructures provide cost-effective storage services that are highly scalable and fault-tolerant. However, ingesting data directly to a cloud storage is not feasible because such cloud storage services can be slow, error prone, and do not support data indexing, which is needed to support fast range queries. Thus, an intermediate storage system is required between stream sources and the cloud storage service for data staging, indexing, and serving realtime and historical data queries. We build Kawkab to serve this purpose while designing it according to cloud and financial markets infrastructure.

High-rate data ingestion: Existing systems commonly follow Lambda [190] and Kappa [149] architectures in which data from streams is initially ingested in a message queue system, such as Amazon Kinesis [15] and Apache Kafka [20]. Applications and storage systems receive data from the messages queues. Data staging using message queues increases data staleness and latency from tens to hundreds of milliseconds [232, 225, 132], which is highly

undesirable in capital market use cases [273, 259]. However, financial data streams often have different fault-tolerance characteristics that can be leveraged to eliminate data replication from the critical path. Financial stream providers publish redundant data streams for fault-tolerance [216, 262] and retain published data for some time for offline access at additional cost [215]. Data receivers can subscribe to ingest data redundantly and may perform data deduplicate during or after data persistence [201]. Moreover, data receivers can recover lost data during ingestion from data providers. We leverage these fault-tolerance properties of financial data streams to eliminate data replication from the critical path during data ingestion, which reduces data ingestion overheads and results in higher scalability and data ingestion rates.

Parallel realtime and historical data queries: Many financial applications access both realtime and historical data in parallel [273, 259]. Realtime and historical queries are typically provided using two separate systems for the two types of queries, such as in the Lambda architecture [190]. Using separate systems results in inefficiencies such as maintaining multiple systems and codebases, and merging results from two different layers [149]. Kawkab separates the two components and leverages the scalability of cloud storage services to support historical data queries without impacting data ingestion and recent data queries. Kawkab uses separate nodes to serve historical data, which fetch data from cloud storage instead of from the nodes that initially ingest the required data. As a result, historical queries do not interfere with realtime queries.

Low-latency realtime queries: Another challenge is to support time-based range queries with low latency for recent data. Applications mostly access recent data within a time window, for which a timeseries index can be built during data ingestion to minimize data search with target timestamps. However, updating the index in the fast path of data ingestion can increase request processing overheads, resulting in lower ingestion rates. Therefore, to support high data ingestion rates and low latency range queries in parallel, building and maintaining the index should incur minimal processing overhead. To achieve this goal, Kawkab introduces a novel timeseries index that is optimized for high rate data ingestion while serving recent data reads with low latency.

This chapter presents the design of Kawkab, and the challenges involved in achieving its performance goals. Moreover, a performance evaluation of Kawkab highlights the effectiveness of designing a system based on the target infrastructure. In the next section we describe the system model and assumptions that we used to design Kawkab.

5.1 Assumptions

Kawkab makes the following assumptions to simplify its design and tailor it to store financial data streams in a public cloud.

Write workload: We assume that the write workload consists of a large number of data streams, which represent financial instruments in capital markets such as stocks. Data streams are bursty, and their collective data rates from multiple streams can peak at tens of million messages per second, though per-stream data rates can be low [79, 262]. Furthermore, the individual message sizes are usually small, ranging from tens of bytes to several hundreds of bytes [217, 262, 101, 256].

Read workload: The read workload comes from applications that perform two types of queries in parallel, realtime data queries and historical data queries [273, 259]. Applications mostly perform realtime queries that access recently ingested data in small time windows. For example, applications may periodically fetch data that is ingested in the last five seconds relative to the current time. Realtime queries have low latency requirements, less than 20 milliseconds [101], because the value of data diminishes over time.

Applications perform historical queries to find trends in data over an extended period of time. A historical data query spans a large time window, such as a few months, which accesses several million messages in a single query [256].

Applications access complete messages that are received from data streams instead of accessing specific fields in the messages. Furthermore, each query accesses only a single data stream. Applications handle multi-stream queries through cross-stream joins.

Fault tolerance: A common approach to provide fault-tolerance during data ingestion is to replicate data on multiple nodes that do not have dependent failures. However, data replication increases the network and storage overheads, which reduce data ingestion rates. Data replication for fault-tolerance is necessary in many cases because data lost during ingestion cannot be recovered from its originating sources. However, as mentioned earlier, financial data providers publish redundant data streams for reliability. Moreover, they retain published data for some duration, e.g., for a trading day [215], and provide additional means to fetch recent data at additional costs. We assume that data stream publishers generate replicated streams and retain published data for offline access for a few hours. Kawkab exploits the unique fault-tolerance characteristic of financial data streams to eliminate data replication on the critical path during ingestion.

5.2 File Abstraction and Data Model

Kawkab uses a *file* abstraction for a data stream. Each file in Kawkab corresponds to a data stream. We define a stream as a continuous sequence of messages belonging to the same financial instrument. For example, messages representing trades of the IBM stock constitute a stream, and File-IBM corresponds to the IBM stock stream. Furthermore, Kawkab stores redundant streams in separate files, e.g., File-IBM-A and File-IBM-B correspond to Stream-IBM-A and Stream-IBM-B in Kawkab. Files in Kawkab are append-only as data streams always move forward with monotonic increasing timestamps.

A file consists of fixed-length data *records*, where a record is a structure that corresponds to a financial event message received from a data stream. Each record has a timestamp field that Kawkab uses for indexing and searching the records based on timestamps. Furthermore, the first 8 bytes of a record are required to be the record's timestamp. This requirement enables Kawkab to perform fast binary search within records without parsing complete messages in range queries.

As applications read complete records based on a time range, Kawkab internally stores records in a row format, i.e., it stores the attributes of a record together instead of storing them separately in different tables. Storing records in row format optimizes I/O for large sequential reads when multiple records are read together based on a time range. However, row format makes range queries more difficult because of requiring full-record reads during search. To support fast range queries, each file in Kawkab has a dedicated timeseries index. The index enables searching of records within given time bounds without fully scanning records in a file.

5.2.1 File Operations

Kawkab supports common file operations: open, close, read, append, file size, and record size. Table 5.2.1 lists the append and read API of Kawkab. The append API supports appending files in batches of records. The read API supports reading individual records as well as a range of records between a start and end timestamp. Individual record access is faster because it doesn't require search in file index.

API Function	Description
Append (String filename, Record[] records)	Appends a batch of records to a file
RecordNum (String filename, long recordNum, boolean fromPrimary)	Returns the record at specified record position in the file. If fromPrimary is true, the record is fetched from the *primary node* of the file, which is the distinguished file writer.
RecordAt (String filename, long timestamp, boolean fromPrimary)	Returns the latest record whose timestamp matches with the given timestamp
ReadRecords (String filename, long minTime, long maxTime, boolean fromPrimary)	Returns all the records between time bounds minTime and maxTime inclusively

Table 5.1: Append and read API in Kawkab

5.3 Architecture

Kawkab consists of three main components: *stream handlers*, *worker nodes*, and a distributed *namespace*. Stream handlers act as a bridge between stream sources and Kawkab worker nodes. They receive messages from data streams and push them to worker nodes for data staging. Worker nodes ingest data from stream handlers and serve read queries. Moreover, they upload ingested data to cloud storage for persistence. The namespace is a distributed file directory that keeps track of the files present in the system. Below we provide further details of these components.

5.3.1 Stream handlers

Financial data providers, such as Options Price Reporting Authority (OPRA) [214] and Vereinigte Wirtschaftsdienste GmbH (vwd) [283], use custom protocols and interfaces to disseminate data through data streams. Receiving data from the streams requires the implementation of provider-specific stream handlers that use data providers' APIs to interact with their systems. For example, OPRA requires data recipients to follow its protocol specifications and network connectivity guidelines [217, 216] to receive data. Messages in data streams have provider-specific formats, which usually differ from the target storage system's data model. Therefore, the messages need to be transformed according to the target storage system's data model.

Kawkab uses a set of *stream handlers* that serve as a bridge between data sources and worker nodes. A stream handler receives messages from a data stream, transforms them

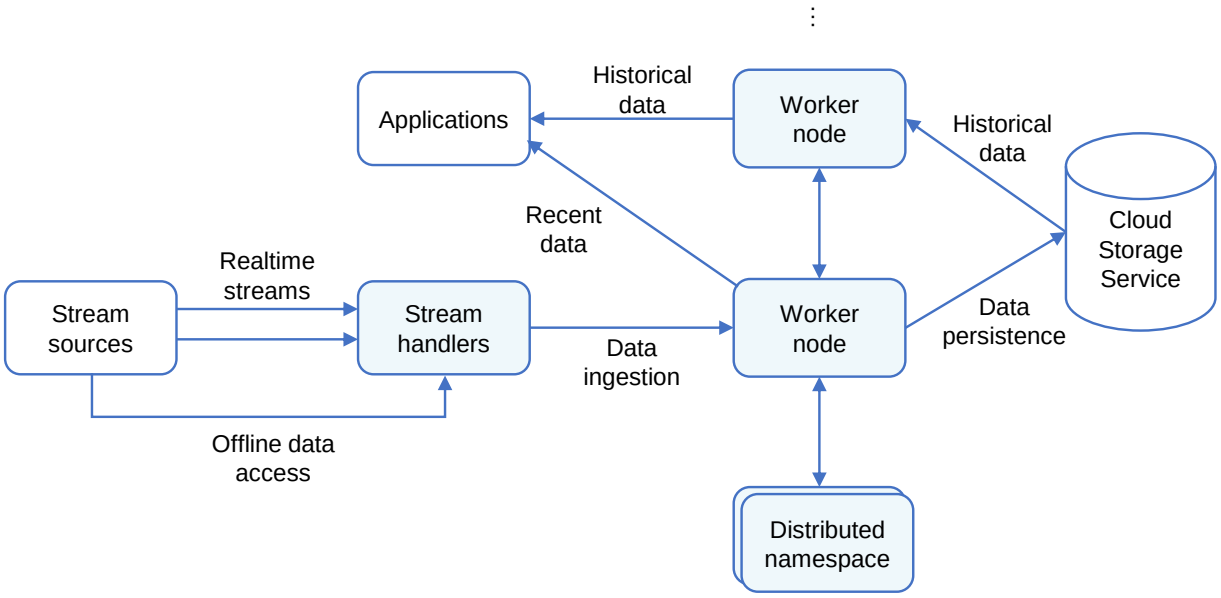


Figure 5.1: Architecture of Kawkab storage system.

into their respective destination record formats, and forwards them to worker nodes to append to their respective files. To append records, stream handlers open stream-specific files and send append requests to the worker nodes. The worker nodes ingest records and eventually persist them in a cloud storage backend.

Handling unordered messages: Data providers often use IP multicast network to publish data streams [216, 101]. Messages can be lost or be received out of order by stream handlers if data providers use UDP-based protocols to publish data streams. Unordered messages can be handled in several ways. Kawkab uses a simple approach where stream handlers buffer incoming messages for ordering and wait for a fixed duration to receive out-of-sequence messages. If a message has not arrived after the fixed duration, the message is ignored. Depending on the application, dropped messages can lead to undesired results. For such applications, stream handlers can be modified to actively fetch the dropped messages from the stream sources to reconstruct the sequence. In our design, we assume that stream handlers always append records in sequence. Kawkab exploits the restriction of time-ordered appends to build a timeseries index that is optimized for searching recent data within a time range.

Eliminating Data Replication: As mentioned earlier, financial data stream providers

often publish streams redundantly and retain the published data for a limited duration, e.g., for a trading day [215]. Instead of replicating received messages from data streams, separate stream handlers in Kawkab receive messages from redundant streams and append them to their respective files. For example, two stream handlers receive Stream-IBM-A and Stream-IBM-B on two separate virtual machines and append them to files File-IBM-A and File-IBM-B. Applications can read data from either stream. Consequently, stream handlers or the worker nodes do not need to actively replicate data to multiple machines during data ingestion.

Failure Handling: In this design, simultaneous failures of stream handlers or both worker nodes that ingest stream replicas can result in losing ingested data. A worker node may take several seconds or minutes to upload ingested data to the cloud storage backend. Ingested data can be lost in rare cases if both worker nodes, handling replicated streams, fail simultaneously before uploading ingested data to the cloud storage. In such cases, Kawkab can recover data through the offline channels provided by the stream providers.

The stream handler whose upload connection to the failed worker node is lost can retrieve the last record from the cloud storage to find out the number of lost records. The lost records can then be retrieved from the stream providers that retain published data for a short duration. However, the throughput of offline channels can be lower than realtime streams, which can result in higher recovery time and unavailability of data to applications until data is recovered. Although this design can result in unavailability of some data during simultaneous failures of nodes that ingest replicated streams, it simplifies the system design and eliminates overheads of replication on the critical path, which results in higher scalability and data ingestion performance.

5.3.2 Worker Nodes

A worker node is a core component of Kawkab that performs all read, write, and indexing operations. Worker nodes mainly serve three purposes. First, worker nodes ingest data from stream handlers at high rates, temporarily store and batch ingested data in locally attached storage, and gradually upload data to the cloud storage backend for persistence. A temporary buffer or staging area is required because data arrival is bursty and message arrival rates can be much higher than the ingestion speeds supported by cloud storage services [79].

The second purpose of worker nodes is to cache ingested data and immediately make it available to applications for realtime data processing. Applications connect to worker nodes and perform realtime and historical data queries. An alternative solution where data

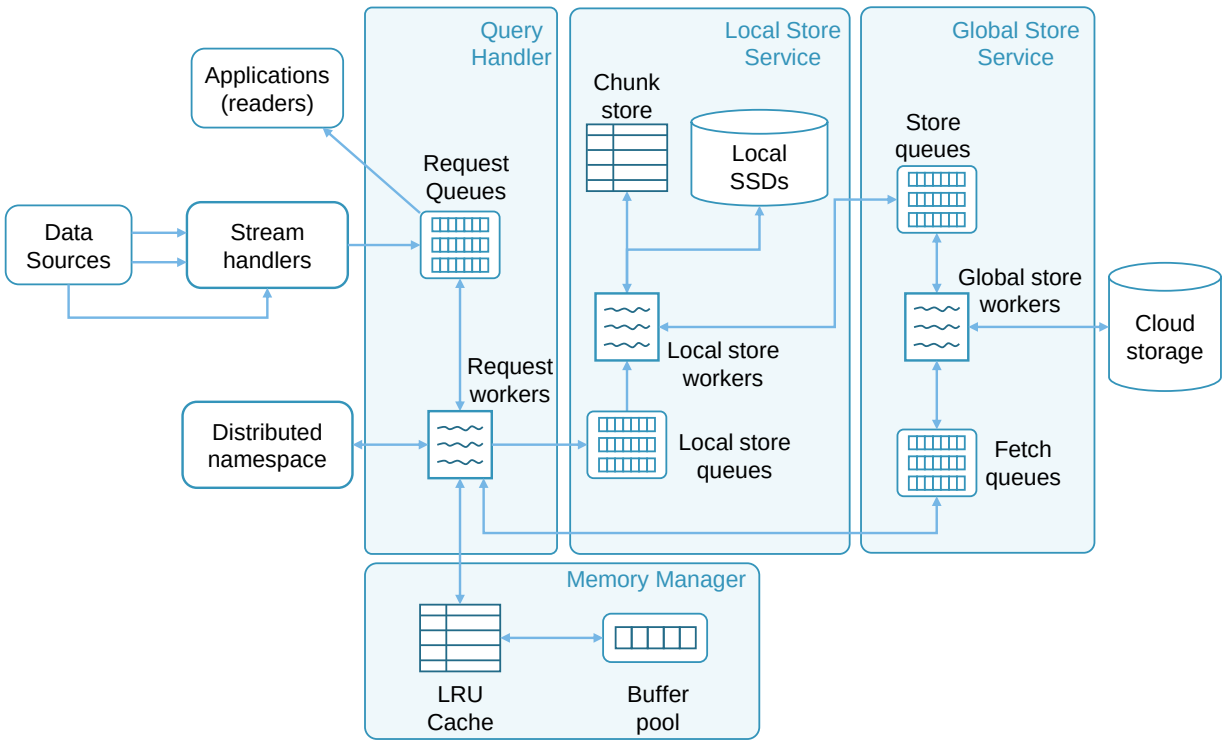


Figure 5.2: Architecture of a worker node in Kawkab

is directly stored and read from cloud storage increases data staleness and latency, which is highly undesirable for realtime data analytics.

Finally, the worker nodes create a timeseries index of received data for each file, which is needed for fast range queries. Cloud services such as Amazon S3 do not support custom indexing methods that can be leveraged to search data based on timestamps. Kawkab creates a timeseries index for each file, which it uses to quickly search recently ingested records based on given time ranges.

Worker Node Architecture

Worker nodes run on virtual machines (VMs) in a public cloud, e.g., using Amazon EC2 machines. Each worker node has a large amount of memory and a number of locally attached storage such as SSDs or Amazon EBS. Figure 5.2 shows the architecture of a

Kawkab worker node. A worker node consists of a set of *query handlers*, a *memory manager*, a *local store service*, and a cloud storage management services called *global store service*.

Query handlers perform the file operations that they receive from stream handlers and applications. Kawkab manages its memory by itself through a buffer-cache, called *memory manager*. Query handlers allocate buffers through the memory manager and perform all file read and write operations using the cached buffers. The local store service manages the *local store* of the node, which comprises the locally attached storage of the node. The local store provides a temporary location to spill ingested data from memory and to batch data for later upload to the cloud storage backend. The global store service manages the reads and writes to the cloud storage backend.

Primary nodes: Initially the node that creates a file or open in append mode becomes the sole writer of the file and is termed as the file's *primary node*. A file's data is ingested only on its primary node. However, the file can be read from any node. This approach eliminates the need for a centralized coordinator or explicit coordination between the nodes during the read and write operations. This approach is similar to *primary-backup* approach in which the readers and writers are allowed to read and write to a file only through the primary replica. The primary node can become the performance bottleneck due to higher load on the primary as compared to the other nodes. To avoid overwhelming the primary nodes, Kawkab allows reading files from any node in the cluster. Reading from non-primary nodes does not create data consistency problems because a file can be modified only through append requests. If a non-primary node does not have required data, the node fetches required data from the primary node or the cloud storage backend and caches fetched data to serve subsequent requests to the same data. In this way, reads to highly popular data can be scaled to a larger number of nodes without overwhelming the primary node.

Data caching on the non-primary nodes can be ineffective in reducing load on the primary if a large number of non-primary nodes are used for the same file. Therefore, reads to the same file through non-primary nodes should be limited to specific and small number of nodes. This can be easily achieved by adding a mapping of file-IDs to designated non-primary nodes in the namespace.

Memory Manager

Each Kawkab node manages its own memory through its centralized memory manager. Memory needs to be managed carefully because memory is a limited resource, and all read/write operations are performed in-memory for high performance. In-memory oper-

ations serve two purposes. First, as incoming data rates can be significantly higher than upload bandwidth to the cloud storage service, in-memory data ingestion enables absorbing high data bursts. Second, reading data from the cloud storage can incur high read latency, which is undesirable in our target use cases. Caching ingested data in memory enables serving reads with low latency for data processing.

The memory manager organizes the memory as an LRU-based buffer cache that is backed by a pool of fixed-length buffers. The buffer pool keeps available a portion of memory for fast data ingestion. Reusing buffers reduces the overhead of allocating memory in the fast path, which results in lower request processing times. Kawkab uses a reference counting cache to allow concurrent readers and the writer of a file to use the same memory buffers.

Memory manager optimizations: As all read and write operations are performed using buffers that are obtained from the memory manager, the memory manager can become a performance bottleneck. This is because the cache in the memory manager is accessed two times per request, both of which require locks for synchronization. The first time it is accessed to obtain a buffer from the cache, and the second time it is accessed to release the buffer back to the memory manager for reference counting and maintaining the buffer pool.

Taking a simple approach, we shard the cache to reduce the contention across the cache. However, sharding alone is not sufficient to support high data ingestion rates. The number of streams is large and data arrival rates per stream can be low. Thus, an append request may consists of only a few records when financial events are spread across data streams. As a result, appends to the same buffer may happen in multiple append requests. Consequently, peak ingestion loads consisting of a large number of small append requests increase contention across each partition.

To address this problem, we reduce the access frequency to the memory manager by not releasing buffers to the memory manager immediately after their use. Instead, a query handler takes the ownership of the buffer it obtains from the cache and releases the buffer after a pre-defined timeout. If an append request to the same file arrives before the timeout, data can be appended to the same buffer without accessing the cache. To achieve this goal, we use a set of non-blocking [1] concurrent queues that delay releasing buffers back to the memory manager for a fixed duration.

Concurrent reads can access the same buffers through the cache, which minimizes data

[1] A *non-blocking* shared data structure guarantees that one of many active processes trying to perform an operation completes the operation in a finite number of steps, regardless of the execution speed of other concurrent processes [118, 191].

staleness. However, the reads are restricted to only successfully appended data to avoid any inconsistent results.

Local Store Service

Each worker node has a local store service that serves two purposes. First, it provides a temporary storage space to spill in-memory ingested data that has yet to be uploaded to the global store. As mentioned earlier, uploads to the cloud storage backend can be slower than incoming data during high data bursts. If data bursts remain for enough duration, the system can eventually run short of free buffers in memory for ingestion if data is uploaded to the global store directly from memory. Therefore, data from memory needs to be stored temporarily in the local store to keep a pool of buffers free in memory for data ingestion.

The second purpose of the local store is to optimize data transfer to the global store by batching data into larger chunks. Cloud storage services like Amazon S3 are optimized for larger blobs, e.g., larger than 4MB, instead of tiny objects [136]. However, buffer sizes in memory should be small, e.g., between 4KB and 256KB, to avoid internal fragmentation. Without storing data in the local store, hundreds of gigabytes of memory would be required to concurrently ingest data for tens of thousands of files, which puts significant memory demand. Kawkab avoids such problems by continuously copying the buffers that have ingested data to larger chunks in the local store. The completely copied buffers can be evicted from the cache and returned to the buffer pool when needed. Moreover, the chunks that become full are transferred to the global store through the global store service. The transferred chunks are deleted from the local store based on LRU policy.

Local store service design: The local store service consists of a set of workers, store queues, and a persistent key-value store, called *chunk map*. The memory manager puts buffers with appended data in the store queues, which the local store workers take for copying them to their respective chunks in the local storage. The chunk map keeps track of the files that are created in the underlying filesystem. Moreover, it keeps track of the files that are already copied to the global store and can be garbage collected.

Global Store Service

The global store service of a node manages reads and writes to a cloud storage system, called *global store*, such as Amazon S3. The global store provides a scalable and fault-tolerant storage of large volumes of data at low cost. Moreover, it minimize the interference between data ingestion and data reads for historical data, which we further explain below.

Leveraging cloud storage for high performance reads: Kawkab differentiates between reads for recent data and historical data. Recent reads involve data that has been ingested in the recent past. Such reads fetch a small number of recently ingested records for realtime analysis. Historical reads fetch larger volumes of data for historical analysis, spanning a large time window.

As it takes time to upload data from the local store to the global store, recently ingested data is more likely be only available on the primary node, i.e., the node that has data ingested from stream handlers. If both realtime and historical data queries are served from the primary node of a file, the load on the primary node increases significantly. As a result, the reads with low performance requirements interfere with data ingestion and realtime queries, which have high performance requirements.

To minimize the impact of historical queries on data ingestion and realtime queries, historical queries are performed on a non-primary node. If the node does not have the required data in memory, it pulls data from the global store instead of the primary node, which significantly reduces the load from the primary node. However, reading data from the global store may incur higher latency, which is acceptable for historical data queries. If a query accesses recent data from a non-primary node, which may not be in the global store, the non-primary node fetches recent data from the primary node and caches the fetched data for subsequent reads to the same data.

5.3.3 Distributed Namespace

Many storage systems keep namespace and file metadata in a centralized metadata node, e.g., NameNode in HDFS. A centralized metadata node can become a performance bottleneck, a single point of failure, and can limit scalability. Kawkab uses separate mechanisms to store namespace and file metadata to avoid such problems. It stores the namespace in ZooKeeper whereas it stores metadata in the cloud storage backend. Namespace in ZooKeeper stores the filename to file ID and primary-node ID mappings.

Metadata storage follows similar storage mechanism as file data storage. Kawkab uses fixed size blocks for metadata and data storage that are persistently stored in the cloud storage backend, which simplifies the system design and leverages the scalability of cloud services for scalable metadata access. Metadata and data storage is further discussed in the next section.

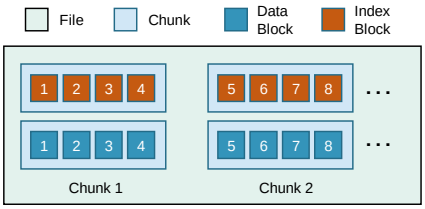


Figure 5.3: File layout in Kawkab

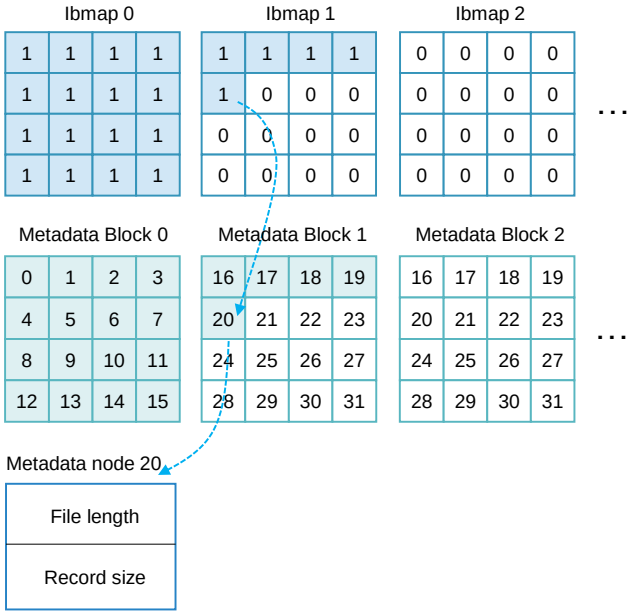


Figure 5.4: Relationship between different types of blocks in Kawkab

5.4 File Organization and Structures

A file in Kawkab consists of three types of fixed-length structures: a *metadata node*, *data blocks*, and *index blocks*. Figure 5.3 shows the layout of a file consisting of these blocks, and Figure 5.4 shows the relationship between the blocks. All blocks are initially copied to the local store for batching purposes. The blocks are persistently stored in the global store. We further describe these structures below.

5.4.1 Data Blocks

Kawkab stores file data in memory in fixed-length blocks, called *data blocks*, which is similar to HDFS and other distributed filesystems. Data blocks are append-only and are immutable once they are completely filled with data and sealed. Data blocks are small in size to reduce internal memory fragmentation. However, for efficiently uploading them to the cloud storage backend, multiple data blocks are batched in larger chunks in the local store. We use 16KB blocks and 4MB chunks in our implementation. Each chunk is itself an append-only file in the underlying filesystem of a node. Chunks that become full are moved to the global store for persistent storage.

Block naming: Unlike other systems, Kawkab does not need to explicitly store file-ID to location mappings, which reduces metadata management overheads. Kawkab builds on existing storage systems, which keep track of the location of their stored files, i.e., the underlying filesystem of a Kawkab node and the cloud storage backend. However, a block need to be addressed uniquely for storage and access from them. Kawkab uses a naming scheme based on file IDs and block numbers to uniquely address blocks.

Each blocks is uniquely named as the combination of `prefix-fileID-chunk#-block#`. A file-ID is a 64-bit number that is globally unique in the system. Chunk and block numbers are 32-bit numbers, which are unique per file. This combination creates a unique name for each block in the system, which is used to create files in the underlying filesystem and to address objects in the cloud storage backend. For instance, block `D-1-2-3` indicates data block `3` in data chunk `2` of file `1`. As block and record sizes are fixed, a block number can be easily calculated from file length or a record number without explicit lookup from a database, which makes block access simpler and faster.

5.4.2 Index Blocks

Kawkab builds a timeseries index for each file that enables applications to query data using time ranges. Similar to file data, Kawkab stores the index in fixed-size *index blocks*. Each index block consists of a sequence of index entries. Each index entry corresponds to a data block and consists of the minimum and the maximum timestamps of the records in the data block. The internal structure of an index block is explained in Section 5.5.

Index blocks are stored and managed the same way as data blocks. Index blocks are append-only and are immutable once they are full with index entries. Moreover, they are batched in the local store in larger chunks. When a chunk becomes full with index blocks, it is moved to the cloud storage backend. Furthermore, Kawkab does not store explicit

addresses of index blocks. Instead, their block numbers are calculated from file length or their position in the index.

Note that an index block does not store any metadata of a file. It only indexes timestamps of records in data blocks. Metadata is stored in a separate data structure, which is described next.

5.4.3 Metadata nodes and metadata blocks

A *metadata node* of a file is a fixed-length structure that stores the metadata and mutable state of the file, such as file length, record size, different timestamps, and file security parameters[2]. Each metadata node has an associated file-ID, which is used to uniquely identify the metadata node and the file.

Kawkab groups multiple metadata nodes in fixed size blocks called *metadata blocks*, which is similar to a Unix-based filesystem. As individual metadata nodes are small, grouping them in larger blocks optimizes I/O for storing and retrieving metadata nodes from the local and global stores. Moreover, it provides a mechanism to share metadata blocks between the files that have temporal locality, which results in lower I/O overheads due to less reads from the local and global stores.

Metadata blocks are uniquely identified by block numbers, which are assigned in sequential order. For example, if each node can use k blocks, the node i uses block numbers from ik to $ik + k$.

Ibmaps: A metadata block may not have all of its metadata nodes assigned to files. Therefore, each Kawkab node needs to keep track of the free and already assigned metadata nodes in the blocks. Like a Unix-based filesystem, Kawkab uses bitmaps, called *ibmaps*, to keep track of the metadata nodes that have been assigned to files. An ibmap is a fixed-size block in which each bit maps to a file-ID, which in turn maps to metadata node numbers. Similar to metadata blocks, ibmaps are uniquely identified by ibmap numbers that are assigned sequentially.

Storage of ibmaps and metadata blocks: Ibmaps and metadata blocks are stored differently based on their update frequency and fault-tolerance requirements. Ibmaps are stored in ZooKeeper whereas metadata blocks are stored in the cloud storage backend similar to other blocks in Kawkab.

[2]In our implementation, we store only file size and record size in a metadata node.

Ibmaps are updated only during file creation and deletion operations, which are not frequent in our target workloads. Moreover, they are required to be consistent with the namespace stored in ZooKeeper. Therefore, ibmaps are stored in ZooKeeper.

Metadata blocks are updated very frequently as they group metadata nodes of many files. A metadata block is updated if any file that has its metadata node in the block is appended. Therefore, metadata blocks are not suitable for storage in ZooKeeper. Instead, metadata blocks are stored in the cloud storage backend. As the blocks are updated rather than being appended, the cloud storage backend is required to support either overwriting objects or multi-version objects so that metadata blocks can be periodically copied to the cloud storage backend.

Non-primary nodes can read metadata blocks from either the cloud storage backend or the primary nodes of files. Reading metadata blocks from the cloud storage backend reduces load on the primary nodes at the expense of reading staled blocks. As historical data queries do not access recent data, reading staled blocks is acceptable. Nevertheless, to enable reading recent data from non-primary nodes, Kawkab's read API supports enforcing to read up-to-date metadata blocks from files' primary nodes.

5.5 Indexing

Our target applications mostly access data that has been received within the last small time window. The latency to read recently ingested data is required to be small, less than 20 ms [101], to retain the time-critical value of data. To address these requirements, Kawkab creates a timeseries index that it uses for time-based record search within a single file. Moreover, the index is optimized to have least impact on data ingestion and to perform faster search for recently ingested data as compared to historical data. Below we describe the index design in detail.

5.5.1 Index structure

Kawkab uses a k-ary *post-order heap* [114] as the main data structure for an index. A post-order heap creates a fully-balanced tree in which nodes are created and inserted in post-order tree traversal ordering. Figure 5.5 shows an example of a post-order heap. Each internal node has k children. The children of a node are created before their parent, from left to right in sequential order. Moreover, the tree grows from bottom to top instead of

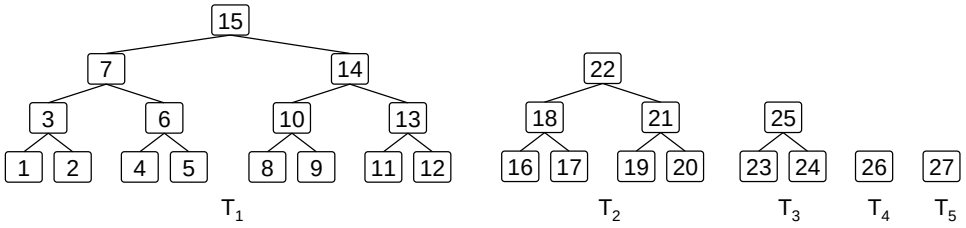


Figure 5.5: A binary post-order heap consisting of 27 nodes in five sub-trees. The nodes are numbered and created in the post-order; first the children are created from left to right, and then the parent node is created.

growing from top to bottom, which is common in B-tree based index structures. Kawkab exploits these properties of post-order heaps in three ways.

First, bottom-up growth in a post-order heap does not require any modifications in the existing nodes when a new node is created. Kawkab uses this bottom-up growth of the index to make the index append-only. As we assume that records in a file are appended in time-order, index entries are also inserted in time order and are always appended to the last node in the index. Consequently, updating the index in the fast path during writes have a negligible impact on data ingestion performance. Moreover, index storage and maintenance is simplified.

Second, we begin index search from the last node in the index, i.e., backwards from the end of the file. As Kawkab caches recently accessed index and data blocks, they are likely be in the cache when recent-data queries are performed. As a result, latency of range queries to access recent data reduces significantly.

Finally, index updates and search do not require full index to be loaded in memory, enabling the index to grow arbitrarily large. Kawkab divides the index into fixed-length blocks that it handles in a similar way as data blocks. Nodes in a post-order heap index correspond to index blocks of a file and are numbered sequentially based on their position in the index. Thus, blocks are accessed based on their numbers or their position in the index. Therefore, index nodes or blocks are not required to explicitly store addresses or names of other index blocks. As a result, index space is maximized to store index entries.

Tree structure: Figure 5.6 summarizes the structure of an index node. Each node in the index tree has k children. Moreover, each index node consists of a header and a body portion. The header of a node consists of k entries corresponding to the k children of the node. The i^{th} header entry consists of the minimum and maximum timestamps in the

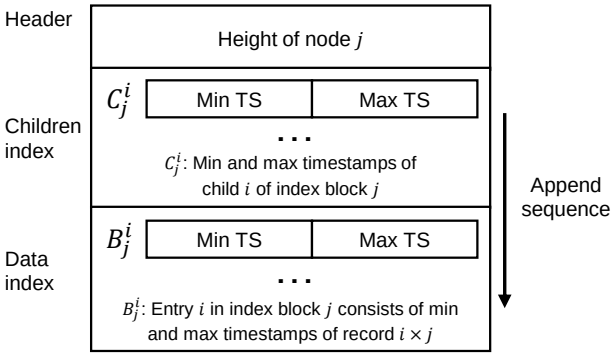


Figure 5.6: Structure of a node in the post-order heap index in Kawkab

sub-tree rooted at i^{th} child of the node, $1 \leq i \leq k$. The body of a node consists of n index entries. An index entry e_j^i, which is the i^{th} index entry in the j^{th} index block, maps to data block number $i \times j$ in its corresponding file. The index entry e_j^i consists of the minimum and maximum timestamps of the records in the corresponding data block.

5.5.2 Building the Index

The index is built by following the logical structure of the k-ary post-order heap structure. Each node in the tree maps to an index block. Initially the index consists of a single block. Each time a new data block is created, the timestamp of the first record in the block is added in the index, which partially fills the last index entry. When the data block becomes full, its last record's timestamp is added in the index, which completes the last index entry. A new index block is created when the last block is completely filled with index entries. If the index block represents an internal node, the block's header is first filled with the minimum and maximum timestamps of the children sub-trees, i.e., the header of the internal node contains the minimum and maximum timestamps of the sub-trees rooted at the node's children.

5.5.3 Index Search

A search query specifies the lower and higher timestamps, t_l and t_h, to search within the index. In the best case, which is searching for recently ingested records, the search accesses only the last index node. In the worst case, the search accesses $\mathcal{O}(k) + \mathcal{O}(log_k n)$ index

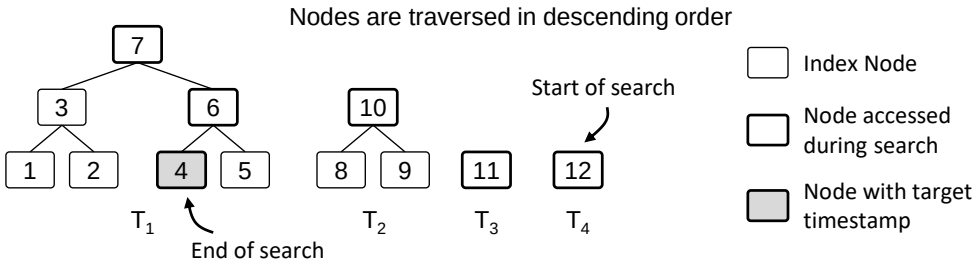


Figure 5.7: Searching a timestamp in a sample binary post-order heap index. The index consists of four sub-trees, T_1 to T_4. Beginning from the last node, N_{12}, the nodes are traversed in the descending order to search a target timestamp.

blocks, where k is the branching factor of the index tree and n is the total number of nodes in the index tree.

Figure 5.7 illustrates searching a timestamp in a sample binary post-order heap index. The index in the example in Figure 5.7 consists of 12 nodes organized in four sub-trees, T_1 to T_4. Index search begins from the last node in the index, which is node N_{12} in Figure 5.7, and traverses root nodes of sub-trees in descending order. If the target timestamp is not located in the sub-tree rooted at the current node, the search continues to the root node of the next sub-tree. If a root node indicates the presence of the target timestamp in its sub-tree, the search traverses down the sub-tree until it reaches the node that contains the target timestamp. If the search reaches the root node of the left-most sub-tree T_1 and the root node does not indicate the presence of the target timestamp, the target timestamp does not exist in the index. Assuming that the target timestamp is located in node N_4 in the example in Figure 5.7, the search traverses nodes $N_{12}, N_{11}, N_{10}, N_7, N_6$, and N_4.

5.6 Implementation

We built a prototype of Kawkab in Java. Our implementation consists of nearly 18k lines of code, which includes the code for server and client library. The system initializes with several configurable parameters such as block and cache sizes, number of local storage devices, number of workers for uploading data to the global store. These parameters can be tuned based on the target deployment scenarios and workloads.

We have benefited from several existing libraries in our implementation. After performance evaluation of gRPC [109], Finagle [97], and Thrift [270] RPC libraries, we used

Thrift RPC for communication over the network. Our local store implementation uses Chronicle Maps [65] library as a persistent key-value store, which keeps track of chunks that are created as files in the underlying filesystem. Chronicle Maps library targets ultra-low latency applications in financial industry and provides high throughput operations. In addition to these libraries, we used Amazon S3 API for cloud storage backend. As a replacement of Amazon S3, we used MinIO [193] for testing in our locally available compute cluster. MinIO is a fast object store that provides an Amazon S3 compatible interface for object storage.

Code Optimizations: A high performance system for realtime data ingestion and queries requires highly optimized append and query execution path [259, 41]. However, optimizing code has been challenging as financial data streams consist of a large number of small messages, which increases request processing overhead and leaves a narrow time window to complete each request. For example, supporting a 500 MB/s burst of 50-byte records leaves only a microsecond to process each request if the load is evenly distributed across 10 cores. A stream is not easily parallelizable either as each stream has only a single writer and can only be handled through a single execution thread. As a result, small request-processing overhead is the key enabler to achieve high performance. In turn, our implementation focused on minimizing code execution path and non-blocking code execution to achieve high ingestion rates.

Our initial prototype consisted of a simple hashmap along with a linked-list as a cache. However, we found synchronization around the cache to be the performance bottleneck as the request handlers accessed the cache twice, for block allocation and deallocation, for each append request. We sharded the cache as a simple optimization, yet we found it to be insufficient to prevent lock contention. To further reduce the contention across a shard, we significantly reduced cache access in the append code path by only allocating blocks from the cache and deferring deallocating the blocks only after a timeout. To achieve this goal, we developed a specialized non-blocking concurrent queue, *timer queue*, which enabled non-blocking execution of most of append requests.

As appends to the same block can be spread across many individual requests, our timer queue data structure supports idempotent enqueue operations, which enables repeated insertions of the same block to the queue without additional overhead. Idempotent and non-blocking enqueue operations proved to be instrumental in several modules in our implementation where two operations are repeatedly performed in a pair. For example, we used timer queues to minimize open/close operations of files in the underlying filesystem, and to handover dirty blocks to the local store for copying locally.

Limitations: Our current implementation uses a large heap size due to the large cache,

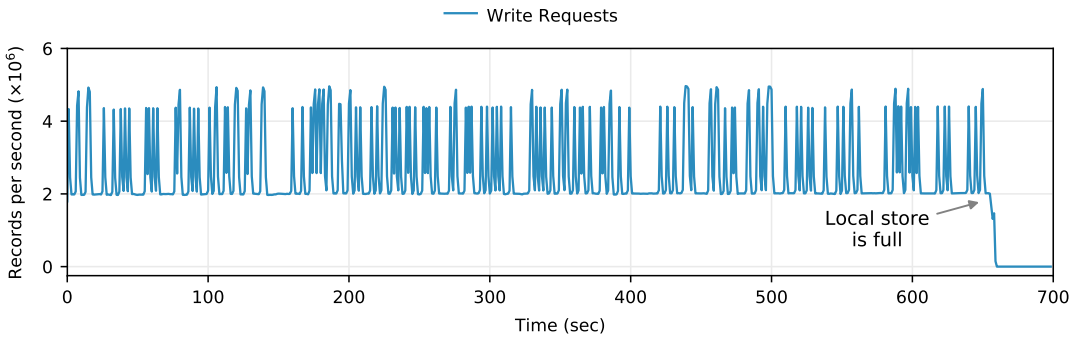
which increase garbage collection times. Although we have used reusable buffers and minimized object creation in our implementation, we observed large garbage collection cycles during our tests. We also observed that increased frequency of garbage collection cycles with higher memory pressure. Therefore, our implementation and system performance can benefit from garbage collection optimizations and using off-heap memory.

5.7 Evaluation

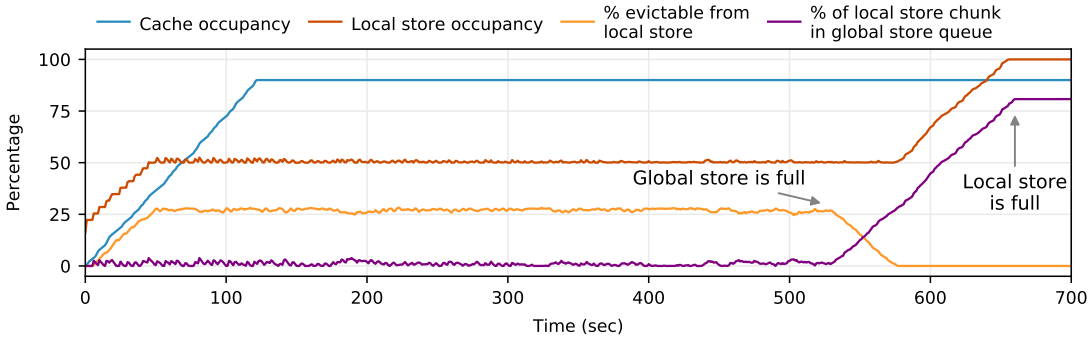
We evaluate our prototype implementation of Kawkab through micro-benchmarks. Our target evaluation metrics are data ingestion rates, request completion times of reading recent records, scalability with respect to the number of nodes, and impact of historical queries on realtime queries and data ingestion. We compare the performance of Kawkab with BTrDB for data ingestion. BTrDB reports higher data ingestion throughput and lower read latency than other timeseries databases [18].

We focus on the evaluation of worker nodes that perform the core functions of storage, indexing, and serving data to applications. To simplify the analysis and test setup, we use a set of clients that emulate stream handlers and applications. The clients act as a separate module that stream handlers and applications can use to interact with Kawkab. We record the throughput and completion times of requests on the client side, which measure the performance of a worker node without any additional overheads of data handling by stream handlers and applications. As a result, the measurements are more insightful and accurate. However, we use synthetic workloads due to lack of access to real data traces. An end-to-end performance measurement using data traces from financial data streams and applications would measure the performance in more realistic settings. In future, we plan to capture data traces from financial data streams and build an application for overall performance analysis of Kawkab.

Experimental setup: Our testbed consists of a cluster of 15 machines that are equipped with a 12-core Intel Xeon E5-2620 CPU, 60GB RAM, and a 200GB Intel S3700 SSD. The machines are connected through 10Gbps links. We configure Kawkab nodes with 16GB cache divided in eight shards. We use 16KB block size and 4MB chunk size. The local store in each Kawkab node consists an SSD. To avoid the expense of Amazon S3 in our tests, we used three dedicated nodes of MinIO [193] storage system to emulate cloud storage backend. MinIO exposes an interface that is compatible with Amazon S3 client library. Accordingly, we use Amazon S3 client library in our implementation to interact with MinIO, which exposes overhead of using Amazon S3 in our tests. Our evaluation is



(a) Throughput of write operations



(b) Status of different modules of the system

Figure 5.8: Data ingestion throughput and the status of different modules in the system during a stable ingestion workload.

based on reads and writes to Kawkab nodes. Therefore, we expect that low latency and high bandwidth to MinIO does not impact the results.

5.7.1 Handling Data Bursts

We first analyze the ability of Kawkab to ingest high data bursts and show the working of different modules in the system. We use a single Kawkab node in this test. We reduce the local store capacity to 25GB and global store space to 60GB in these tests to reduce test time. We generate data traffic using a set of 800 clients that emulate stream handlers. The clients append 64-byte records at a steady rate of 2 million records per second (mps). In addition, they generate bursts of nearly $4.5mps$ with 20% probability to emulate an effect of bursty traffic. Figure 5.8 shows the ingestion throughput of the system (top figure)

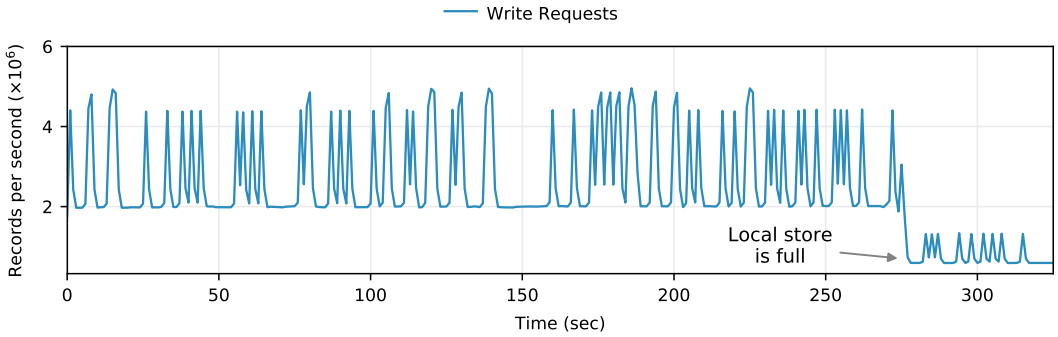
along with the status of the cache, the local store, and the upload queue of the global store service (bottom figure) during the test.

Figure 5.8(a) shows data-ingestion spikes that correspond to incoming data bursts. We observe that the system is able to handle bursts of $4.5mps$ in addition to steadily ingesting records at $2mps$. The system continues to process the requests until the node's local store runs out of space, which itself happens due to the global store being full.

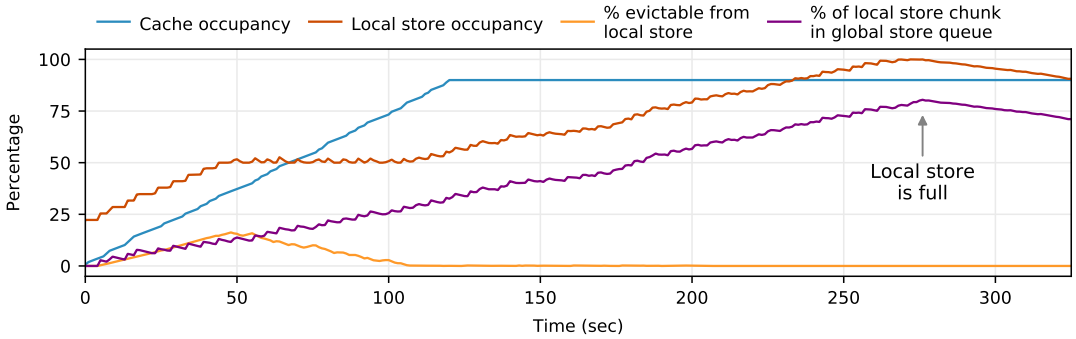
Figure 5.8(b) shows the occupancy of the cache and the local store, fraction of the chunks in the local store that can be deleted, and the length of global store queue during the test. The cache occupancy keeps growing until it becomes 90% occupied, which happens close to 120 seconds in the test. In our implementation, the cache evicts at a low rate until it reaches to the pre-defined 90% occupancy level. At 90%, the cache maintains its buffers aggressively, where it evicts an existing block whenever it adds a new block in the cache. The system is able to maintain the level of the cache because the cached buffers are copied to the local store faster than the average data ingestion rate, keeping enough buffers cleaned for cache eviction.

The local store occupancy line in Figure 5.8(b), which is the second line from the top, shows local store utilization of the node. At the beginning of the test, 16% of the local store is occupied due to metadata, data, and index chunks of 200 files. The local store consists of an SSD that is configured to store $6,400$ chunks (25GB) in this test. As the existing chunks become full, they are added in the global store service queue to be uploaded to the global store. Until the local store reaches half of its capacity, no chunks are deleted from the local store. We observe this effect in the first 50 seconds in the test. Similar to the cache, the local store keeps filling until 50% occupancy, which happens at 50 seconds in the test. During this time, the chunks from the local store are continuously uploaded to the global store, which can be observed from the bottom two lines. The third and fourth lines from the top show the fraction of the local store that can be evicted and the fraction of the local store that is enqueued to be uploaded to the global store.

At 532 seconds, the global store runs out of space. Therefore, after 532 seconds, we observe that the global store queue starts filling while the fraction of evictable chunks from the local store starts decreasing. At 577 seconds in the test, the local store runs out of chunks that can be deleted. Therefore, the local store's occupancy starts increasing, until it reaches 100% of its capacity at 657 seconds in the test. At this point, the system starts declining the append requests that require creating new chunks. The clients backoff exponentially and keep retrying for a fixed number of retries. Eventually they drop the requests. We do not expect this situation to happen in a cloud deployment because cloud storage space is unlikely to become full.



(a) Throughput of write operations during the test



(b) Status of different parts of the system during the test

Figure 5.9: Throughput and status of different modules in the system when the ingestion rate is higher than the upload rate to the global store.

Extended high bursts: The memory unit and the local store eventually run out of space when the upload rate to the global store is lower than the average ingestion rate. In another experiment, we limit the upload rate to the global store by reducing the number of global store worker threads, which upload chunks to the global store. Figure 5.9 shows the behavior of the system when the local store becomes full due to lower bandwidth to the global store. We observe that the global store queue is emptied slower and the queue keeps building, until all of the chunks in the local store are either partially filled or enqueued to be uploaded to the global store. Close to 50 seconds in the test, the local store runs out of the chunks that can be evicted. Therefore, the local store occupancy starts increasing. Close to 270 seconds, the local store becomes full and the system starts dropping the append requests that require creating new chunks in the local store.

5.7.2 Data Ingestion Performance

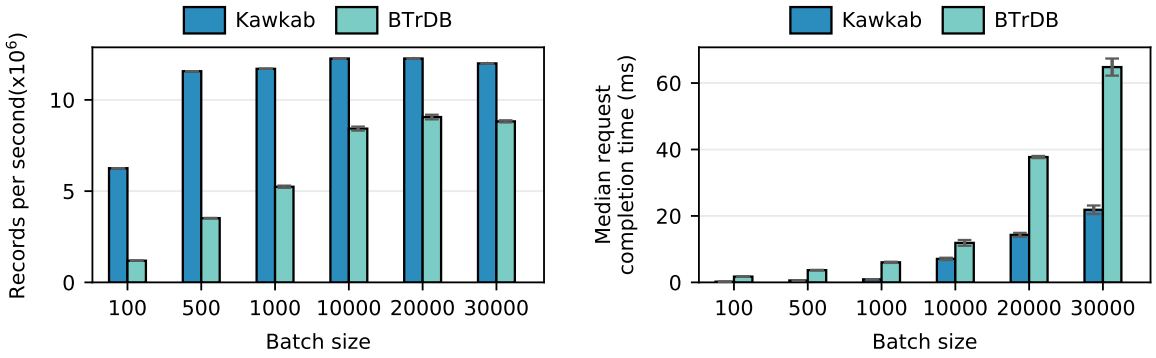
In this section, we evaluate the data ingestion throughput of a single Kawkab node in comparison with BTrDB using different batch sizes in the workloads. Message sizes in financial data feeds are mostly small, ranging from tens to a few hundreds of bytes. Therefore, batching is essential to overcome the network and request processing overheads. During peak loads, records accumulate quickly, leading to low staleness and high data ingestion rates. However, during low traffic, waiting to complete a batch can increase data staleness if the wait time is large. As a result, batching requires a tradeoff between data staleness and data ingestion throughput.

BTrDB configuration: BTrDB uses Ceph [287] object storage system (OSD) as the persistent storage backend. Moreover, it uses Etcd [89] to store cluster configuration. Our testbed for BTrDB's evaluation consists of three Etcd nodes and 11 Ceph OSDs on 11 machines. Ceph OSDs use 60GB loop devices on SSDs for data persistence. A loop device maps its data blocks to another file in the filesystem instead of mapping to a physical disk. We used loop devices because our cluster is shared among many users and creating dedicated partitions for Ceph required creating new disk partitions, which can result in losing existing data. To maximize bandwidth to the Ceph cluster, we disable data replication in Ceph and co-locate BTrDB nodes with Ceph nodes. Ceph benchmarks show 1.13 GB/s average pool bandwidth and 55ms average latency of writing 4MB blocks.

Workload: The workload in this test consists of append requests generated through 800 clients for Kawkab and 200 clients for BTrDB, which are spread across eight machines. Each client connects to a single Kawkab or BTrDB node and appends 16-byte records to a dedicated file. The request arrival follows Poisson distribution at a given arrival rate, which we vary to control traffic generation.

Figure 5.10(a) shows data ingestion throughput of Kawkab and BTrDB. The bars in the figure show the ingestion throughput with batch sizes ranging from 100 to 30,000. The results show that a Kawkab node is able to ingest nearly 6*mps* (96 MB/s) when data is ingested in batches of 100 records, which is 5.3x higher throughput than BTrDB in similar settings. Both systems' throughputs increase with larger batch sizes. However, Kawkab's throughput does not increase with batch sizes larger than 10,000, whereas BTrDB's throughput peaks in the 20,000 batch-size workload. With the batch sizes of 1000, Kawkab's ingestion throughput is nearly 11.5*mps* (184 MB/s), which is 2.2x higher than BTrDB's throughput.

BTrDB's ingestion throughput increases significantly with larger batch sizes. However, a large batch size can add a significant delay to accumulate and ingest, which results



(a) Peak throughput

(b) Median request completion time when the load is 70% of the peak throughput

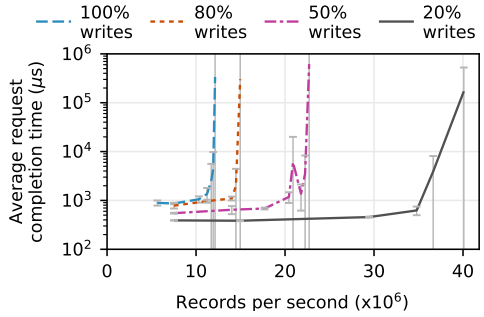
Figure 5.10: Average throughput and median request completion times of append requests with various batch sizes.

in higher data staleness. Figure 5.10(b) shows the median request completion times of write requests when the request rate is 70% of the peak throughput. Kawkab's median request completion times range from 220μs to a millisecond for batch sizes between 100 and 1000. With a larger batch size, request completion times grow faster, ranging from 7ms to 22ms for batch sizes between 10,000 and 30,000. However, BTrDB takes 1.7x (batch size 10,000) to 8x (batch size 100) the median request completion time of Kawkab for the same workloads.

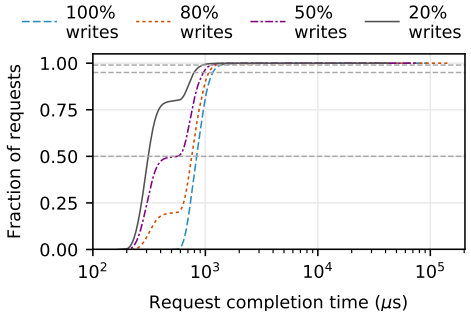
High throughput and short request completion times in Kawkab highlight the importance of reducing request processing overhead in the fast path during data ingestion. One of the ways to reduce request processing overhead is to eliminate data replication in the fast path when possible. Being an infrastructure-aware storage system, Kawkab takes advantage of fault-tolerance guarantees provided by data sources to eliminate data replication in the fast path. Eliminating data replication simplifies the system design and significantly reduce request processing overheads, which results in higher throughput and lower latency for data ingestion. The tradeoff is longer time to recover from failures when lost data cannot be recovered from the local store and the cloud storage backend.

5.7.3 Recent Data Queries Performance

In this section, we evaluate the performance of realtime data queries when they are performed on the same node where data is ingested. The workloads in these tests consist of



(a) Average throughput and request completion times (log y-axis)



(b) CDF of request completion times when the request load is 7 million records per second (log x-axis)

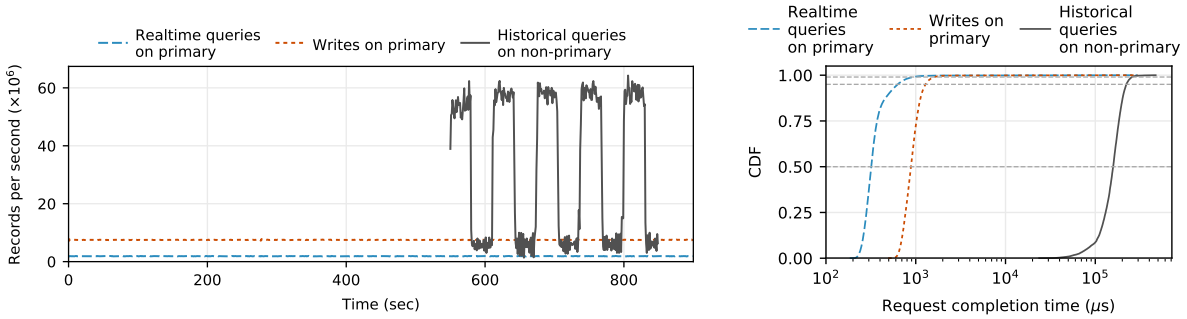
Figure 5.11: Throughput and request completion times of a single Kawkab node with varying ratio of writes in the workloads.

a mix of writes and recent-data range queries. We use 800 clients in these tests, which send requests at a pre-defined request rate and a pre-defined ratio of writes to reads. Each write request consists of a batch of 1000 16-byte records. Moreover, a read request fetches 1000 records from the last 50,000 records in a stream. We control the batch size of reads by using timestamps that are aligned with record numbers. For example, a record search for a time window of 0 to 1000 fetches the first 1000 records in the stream.

Figure 5.11(a) shows the average throughput and request completion times of requests with increasing request load in the system. The lines show request completion times for workloads consisting of 100%, 80%, 50%, and 20% writes. The error bars show 95% confidence interval across five runs. In 80% writes workload, Kawkab is able to process nearly $14mps$ with average request completion time of 1.1ms. The throughput increases and average request completion time decreases with the ratio of reads in the workload. In 20% writes workload, Kawkab is able to process $34mps$ (544 MB/s) with average request completion time of $625\mu s$.

Figure 5.11(b) shows cumulative distribution of request completion times when the request load is $7mps$. We observe that median request completion times are less than 0.9ms in all workloads. Moreover, 99^{th} percentile latency is between 0.95ms (20% writes) and 1.4ms (100% writes).

A 16KB index block in our configuration indexes 680 data blocks, which covers 696,320 records of 16 bytes (1024 records per data block). Therefore, an index lookup in a read request mostly accesses only the last index block in these tests. Our instrumentation



(a) Throughput timeline

(b) Request completion time of read and write requests

Figure 5.12: Impact of historical queries on data ingestion and realtime queries. Historical queries are performed on a separate node, and they fetch data from the cloud storage backend.

during tests show that average index search time is less than 1% of the total read request time. Thus, most of the read request time is spent in data copy, serialization, and network overhead.

5.7.4 Impact of Historical Data Queries

In this section, we evaluate the ability of Kawkab to serve historical data queries without impacting data ingestion and realtime read queries. We use two Kawkab nodes in this test, a *primary node* and a *non-primary node*. The clients ingest data and read recently ingested data from the primary node. They send requests at the rate of 10 *mps*. We use batch size of 1000 for both reads and writes on the primary node.

We run the test for 900 seconds. We do not perform any historical queries in the first 550 seconds to let some files to be uploaded to the global store. After 550 seconds, 10 clients on the non-primary node read historical data in an on/off pattern. The clients request a range of one million records from a random section of lower half of a file. To ensure that the queries are not served from the cache on the non-primary node, we reduce the cache size on the non-primary node to 1GB.

Figure 5.12(a) shows the timeline of throughput in the test. The bottom two lines in the figure show the throughput of reads and writes performed on the primary node. As expected, we do not observe any impact of historical data queries on the throughput of reads and writes performed on the primary node.

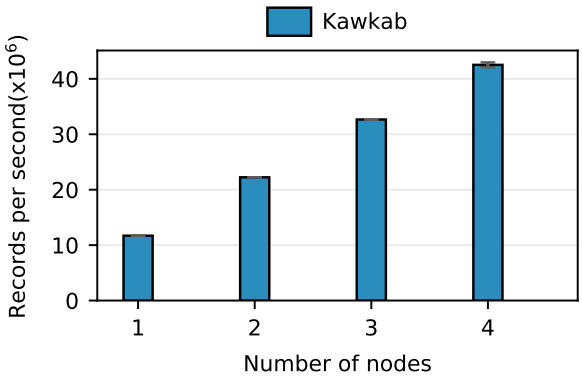


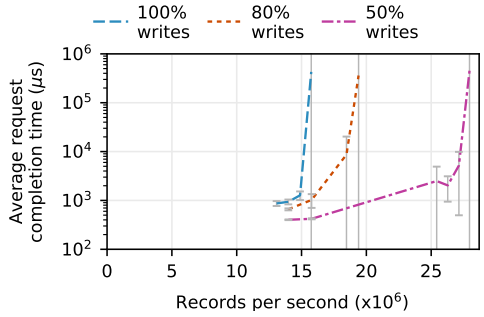
Figure 5.13: Scalability of ingestion throughput of Kawkab with number of nodes.

Figure 5.12(b) shows the CDF of request completion times of the requests performed on both Kawkab nodes. The results show that historical queries on a non-primary node do not interfere with data ingestion and realtime queries on the primary node. At 99^{th} percentile, the reads and writes on the primary node are completed in less than 2ms, which is the same as when the requests are performed without historical data queries. This is because historical data is loaded from the backend nodes instead of the primary node, which is one of the design goals of Kawkab.

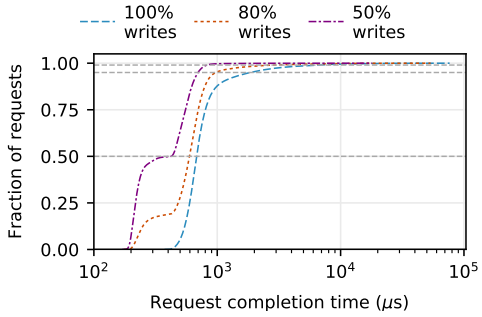
5.7.5 Scaling with the number of Kawkab Nodes

In this section we evaluate Kawkab's ability to scale with the number of nodes in the system. In this test, we run multiple Kawkab nodes. The workload consists of 200 clients per node, which append 16-byte records in batches of 1000 records.We use three MinIO nodes as the cloud storage backend, which are connected through a 10Gig network.

Figure 5.13 shows the peak data ingestion throughput with up to four nodes. The results show that ingestion throughput of Kawkab scales linearly. We observe linear scalability because the nodes ingest data in parallel without any interaction with each other. The scalability of nodes depend on the scalability of the backend storage system. In our test setup, the aggregate upload rate of four Kawkab nodes is less than the ingestion throughput of three MinIO nodes. Our benchmarks show that a single MinIO node can receive 4MB objects with the rate of 990 MB/s in our cluster. Thus, we do not observe throughput saturation with four Kawkab nodes. When the storage backend is the bottleneck, the nodes eventually run out of their local store capacity, which in turn results in unavailability of



(a) Average throughput and request completion times (log y-axis)



(b) CDF of request completion times when the request load is 14 million records per second (log x-axis)

Figure 5.14: Throughput and request completion times of a single Kawkab node with varying ratio of writes in the workloads in a cloud deployment.

free buffers in their caches and failure of append requests. We observe a similar effect in Figure 5.9.

5.7.6 Evaluation in a Cloud Deployment

We validate the results of our local cluster setup by running some experiments in the Amazon cloud. Our cloud deployment consists of a single Kawkab node and four client machines. The Kawkab node runs on c5d.4xlarge EC2 VM whereas the clients run on c5.4xlarge VMs. The VMs consist of 16 vCPUs, 30GB RAM, and 10Gbps network. In addition, the Kawkab node has a locally attached 400GB NVMe SSD. For data persistence, we use Amazon S3 as the cloud storage backend. All chunks are stored in the same bucket in Amazon S3.

We configure the Kawkab node with 8GB cache, 16KB blocks, and 4MB chunks. We limit the local store capacity to 25GB to reduce experiment time by quickly running out of the local store capacity if writes to the SSD become the bottleneck for data ingestion. Similar to previous experiments, we use a set of clients that emulate stream handlers. The clients batch records that arrive at a given Poisson rate and append batched records to their respective files.

Throughput and Latency of Requests

We first evaluate the throughput and latency of write requests and range queries to access recently ingested data. In this test, 600 clients read and write 16-byte records in a batch of 1000 records. Read queries access records from a random range of 1000 records from the last 50000 records in a stream.

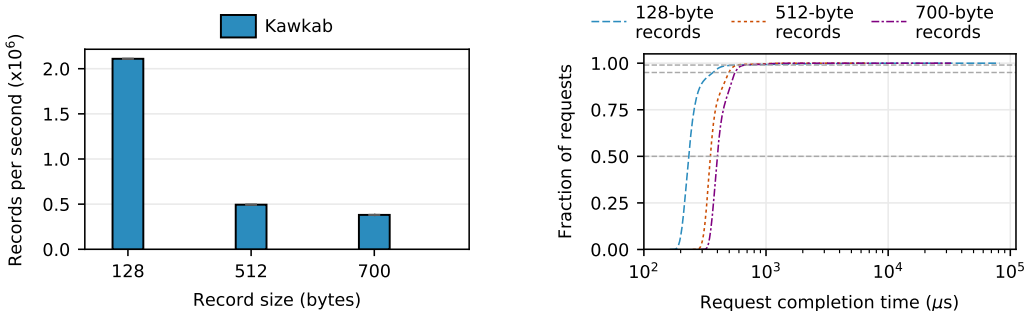
Throughput: Figure 5.14(a) shows the average throughput and completion time of requests with the workloads consisting of 100%, 80%, and 50% write requests. The error bars show 95% confidence intervals across five runs. The results show that a single Kawkab node is able to ingest close to 14.5 million records per second when the workload consists of only write requests. The request throughput increases with the ratio of reads in the workload, increasing up to $25mps$ when half of the requests are writes in the workload.

Request completion times: Figure 5.14(b) shows the CDF of completion times of requests when the load on the system is $14mps$ in the same experiment. The horizontal dashed lines in the figure mark median, 95^{th}, and 99^{th} percentiles. The figure shows that 95% of the write requests complete in less than 2ms when the workload consists of only write requests. At 99^{th} percentile, the completion time of writes is nearly 10ms. The completion time of requests reduces with the increase in the ratio of reads in the workload. Tail latency at 99^{th} percentile reduces to less than a millisecond when half of the requests are reads in the workload.

The throughput and latency results of Kawkab in the cloud deployment are similar to those of the experiments performed in our local cluster. The throughput in these tests is 14% higher in the cloud deployment, which is likely due to the faster CPUs in the VMs. The latency results are the same in both deployments. However, when historical data is accessed from the cloud storage backend, we expect lower read throughput and higher read latency due to different performance properties of the cloud storage backend as compared to our local cluster deployment. Our measurements show that it takes 100ms to more than a second to read a 4MB block from Amazon S3, which results in lower read performance for historical data.

Ingestion Throughput with Large Record Sizes

Previous tests show results with 16-byte records, which evaluate the system's ability to handle small records. In this experiment, we use larger record sizes that are more common in financial data streams. Figure 5.15(a) shows the average write throughput with the record sizes of 128, 512, and 700 bytes. As expected, the throughput drops with larger



(a) Average throughput of writes with varying record sizes

(b) CDF of request completion times

Figure 5.15: Throughput and request completion times of a single Kawkab node with varying record sizes in a cloud deployment.

record sizes whereas the goodput remains the same. The request throughputs are $2mps$, $0.5mps$, and $0.3mps$ for 128-byte, 512-byte, and 700-byte records, respectively. However, goodput of the requests is close to 250 MB/s in all of the three workloads, which is equal to the write speed of the provisioned SSD for the local store. We provisioned a general purpose SSD to save costs, which rate limits the write speed to 250 MB/s.

In our test setup, write bandwidth of the local store is lower than the upload bandwidth to Amazon S3. Our measurements show 270 MB/s upload rate to Amazon S3 with eight worker threads and 4 MB object sizes. As the upload rate to the cloud storage is higher than the write speed to the local store, the local store has limited the sustained ingestion throughput to 250 MB/s in our tests. The ingestion throughput can be increased by provisioning more local store bandwidth.

Figure 5.15(b) shows the CDF of request completion times at the ingestion rates that correspond to the throughputs given in Figure 5.15(a). We observe that the request completion times are less than a millisecond at 99^{th} percentile for the three workloads. The request completion times are significantly smaller in these tests as compared to the previous tests with 16-byte record sizes. The completion times are small mainly due to low write latency to the cache and small queuing delays. Figure 5.15(b) shows the completion times only when the input data rate is lower, but close to the write bandwidth of the local store. If we increase the input load, the demand for clean buffers in the cache exceeds the rate at which the buffers can be cleaned. As a result, the system runs out of the cache and drops excessive write requests.

5.8 Discussion

Kawkab achieves high data ingestion throughput by eliminating data replication during data ingestion. To avoid data replication without losing data during node failures, Kawkab relies on the fault-tolerance guarantees provided by financial data providers, which include publishing data through redundant data streams and retaining published data for a short duration such as a few minutes. However, not all upstream providers, such as banks, offer such fault-tolerance guarantees. In such cases, an alternative approach to achieve high fault-tolerance is to redundantly subscribe to the same data stream, which does not require any modifications in Kawkab. Instead of receiving one data stream by only one stream handler, two stream handlers can separately subscribe to the same data stream and ingest data redundantly. The tradeoff would be the higher monetary costs due to additional subscriptions to receive data.

In the cases where neither upstream providers guarantee fault-tolerance nor data streams can be subscribed redundantly, Kawkab can be extended to add on-demand replication like Alluxio [172] for high availability and data durability. Data can be ingested at different storage tiers that have their own fault-tolerance properties and performance tradeoff. For instance, data can be ingested only in memory, synchronously written to locally attached storage that is backed by Amazon EBS, or replicated to the memories or local stores of other Kawkab nodes. RAMCloud [1] and CRAQ [267] chain replication are practical approaches for data replication on the fast path. The main tradeoffs are higher system complexity and lower performance as compared to Kawkab's current design in terms of ingestion rates, data staleness, and latency to access recent data. We plan to further explore the design space in future to provide higher fault-tolerance guarantees while achieving the same performance goals.

5.9 Summary

Realtime processing of fast streaming data from capital markets represents an important class of applications. Existing timeseries databases and storage systems do not adequately use the available resources and optimization opportunities as they are designed to be infrastructure-oblivious. In this chapter, we have presented the design and evaluation of Kawkab, a high performance streaming data storage systems, which is tailored for financial streaming data storage in a public cloud deployment.

Kawkab supports high data ingestion rates and fast time-based range queries over recent data. Our experimental results using end-to-end benchmarks and synthetic workloads show

that a single Kawkab node can ingest 16-byte records with the throughput of 8 to 9 million records per second. Moreover, Kawkab supports low data staleness and range queries for recent data. Furthermore, Kawkab leverages the scalability of cloud storage services to support historical data queries without impacting data ingestion rates and realtime data queries.

Chapter 6

Conclusion

Big data applications process large volumes of data to reveal trends and insights that help businesses make critical decisions. However, data volumes can be very large, posing challenges in efficiently utilizing available resources for data storage and management. Big data applications thus employ large-scale distributed storage systems to handle the large volume data storage and tackle related challenges. Applications use the storage systems to retrieve and store data on the critical path. Therefore, poorly performing storage systems can become a performance bottleneck for big data analysis.

Distributed storage systems use a large amount of infrastructure resources to meet workload demands of big data applications. These resources include networks, compute and storage devices, and public cloud services. However, most storage systems do not effectively utilize available resources as they are designed to be oblivious to the infrastructure. Although using infrastructure as a black-box simplifies system design, it introduces several inefficiencies such as redundant communication over bandwidth-constraint links, unnecessarily replicating data, and making ineffective use of available replicas. Such problems can be mitigated by making storage system infrastructure-aware. However, effectively using the available information for performance improvements is challenging.

6.1 Contributions

This thesis has explored using infrastructure-aware design to improve the performance of distributed storage systems and has addressed related challenges. In this work, we have developed and evaluated three systems, Mayflower, Canopus, and Kawkab, which

use infrastructure-aware design for performance optimizations. These systems target three categories of storage systems and demonstrate the effectiveness of infrastructure-aware design approach to improve their performance.

The Mayflower filesystem addressed performance problems in reading large files over the network during batched data analysis. Being a network-aware filesystem, Mayflower works closely with the SDN control plane to mutually select the best replica and network path combination for file read operations. The novel file replica and network path selection method of Mayflower minimize the completion time of a new read request with minimal impact on the existing jobs in the system. Our evaluation of a prototype implementation of Mayflower using common datacenter workloads shows that Mayflower is 70% faster than reading files using commonly used replica and path selection approaches. Moreover, compared to always reading data from the nearest replica, our evaluation results show that, depending on network conditions, higher performance can be achieved by reading from a farther replica and opportunistically reading from multiple replicas.

In addition to improving the performance of storage systems used for batch processing, this thesis has addressed the problem of scaling consistent data replication in geo-distributed key-value stores. Several key-value stores use consensus protocols for consistent data replication across geo-distributed nodes. We designed Canopus consensus protocol in this thesis, which exploits the resilient design of modern datacenter infrastructure to achieve higher scalability. Assuming that the network is resilient, is hierarchical, and that rack-failures do not occur, Canopus is optimized to handle high request rates in the common case where a network does not partition and nodes within a rack do not fail simultaneously. Our evaluation of a Canopus prototype shows that it can handle more than four million requests per second with 21 nodes in a geo-distributed deployment.

Our final research contribution targets streaming data storage systems in the capital market industry. Financial data has an associated value that diminishes over time. Therefore, realtime storage and processing of financial data with minimal response time is critical for many applications, such as fraud detection in financial transactions. This thesis presented *Kawkab*, a streaming data storage system, which uses public cloud infrastructure to provide cost-effective storage for capital market use cases. As an infrastructure-aware storage system, Kawkab exploits the fault-tolerance of data at its sources to eliminate the use of message queues and data replication during data ingestion. The outcomes are higher scalability and data ingestion rates, and lower data staleness. In addition, Kawkab enables applications to perform both realtime and historical data queries with high performance using the same system, reducing system management overheads. Furthermore, leveraging the scalability of cloud storage services, Kawkab shifts the load of historical data queries to cloud storage, which eliminates the negative impact of historical query workloads on real-

time data ingestion and queries. Our evaluation of a prototype implementation of Kawkab shows that Kawkab achieves 1.4x to 5x higher ingestion rates than BTrDB under similar experimental settings. Moreover, applications can read recently ingested data with low latency.

The evaluation results of our systems highlight the effectiveness of an infrastructure-aware design in improving resource utilization as compared to an infrastructure-oblivious design. Thus, distributed storage systems, which perform resource intensive operations, can significantly benefit from infrastructure-aware design to improve their performance.

6.1.1 Future Work

In this section, we discuss some possible future directions for further research based on the work presented in this thesis.

Byzantine Fault-tolerance in Canopus

Emerging technologies, such as permissioned blockchains, require agreement on the ordering of requests between a large number of participating nodes. Typically they use Byzantine fault-tolerant (BFT) consensus protocols as the nodes can be malicious and cannot be trusted. Although Canopus does not tolerate Byzantine faults, it can be used in the situations where security and trust are ensured at the endpoints using hardware assistance, such as Microsoft Coco framework. Nevertheless, Canopus is currently being extended to tolerate Byzantine faults [143] in an on-going work.

Indexing Data Summaries in Kawkab

A class of applications use summarized data for analytics, e.g., rolling averages and minimum/maximum prices of financial instruments in fixed time windows are used for dashboard applications. Although supporting summary statistics has not been the focus of this work, such applications can benefit from Kawkab if it can be extended to store and index summary statistics over data in different time ranges. The current design of Kawkab requires fetching data on the application side to compute the summary statistics. However, downloading data only to calculate summary statistics can waste infrastructure resources such as the available network bandwidth. A better approach is to summarize data on Kawkab nodes and send only data summaries to applications, which can be achieved through a few extensions in Kawkab.

Kawkab can support multiple file indexes, which can target different summary statistics and indexes on different record fields. Moreover, Kawkab's indexing mechanism can be extended to index data summaries with different time ranges. Furthermore, metadata blocks can serve as a persistent buffer for partial statistic calculations. With these extensions, Kawkab can support fast summary statistic search for recent records without wasting infrastructure resources.

6.1.2 Concluding Remarks

Data analysis is at the heart of many applications, which require data collection and management at large scales using distributed storage systems. Storage requirements have increased many folds in the recent years due to advancement in technologies that require massive data collections and replication, such as blockchains and applications of machine learning and artificial intelligence. We expect data sizes to further grow rapidly in the coming years, which makes the efficient data storage and management a significant demand. In this thesis, we have focused on improving the performance of distributed storage systems using infrastructure-aware design methods. The evaluation of the systems presented in this thesis show that their design and methods are effective in improving the performance of storage systems, which are commonly used by large-scale distributed applications in datacenters. We believe that the research contributions in this thesis will inspire the adoption of infrastructure-aware design of storage systems for better resource management and higher performance.

References

[1] The RAMCloud storage system, author=Ousterhout, John and Gopalan, Arjun and Gupta, Ashish and Kejriwal, Ankita and Lee, Collin and Montazeri, Behnam and Ongaro, Diego and Park, Seo Jin and Qin, Henry and Rosenblum, Mendel and others. ACM Transactions on Computer Systems (TOCS), 33(3):1–55, 2015.

[2] Michael Abd-El-Malek, William V Courtright II, Chuck Cranor, Gregory R Ganger, James Hendricks, Andrew J Klosterman, Michael P Mesnier, Manish Prasad, Brandon Salmon, Raja R Sambasivan, et al. Ursa Minor: Versatile cluster-based storage. In FAST, volume 5, pages 5–5, 2005.

[3] Sarita V Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. Computer, 29(12):66–76, 1996.

[4] Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, and Tevfik Kosar. Multi-leader WAN Paxos: Ruling the archipelago with fast consensus. Technical report, 2017-01, University at Buffalo, 2017.

[5] Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, Bekir O Turkkan, and Tevfik Kosar. Efficient distributed coordination at wan-scale. In Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on, pages 1575–1585. IEEE, 2017.

[6] Phillipe Ajoux, Nathan Bronson, Sanjeev Kumar, Wyatt Lloyd, and Kaushik Veeraraghavan. Challenges to adopting stronger consistency at scale. In 15th Workshop on Hot Topics in Operating Systems (HotOS XV). USENIX Association, 2015.

[7] Phillipe Ajoux, Nathan Bronson, Sanjeev Kumar, Wyatt Lloyd, and Kaushik Veeraraghavan. Challenges to adopting stronger consistency at scale. In Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems, HOTOS'15, page 13, USA, 2015. USENIX Association.

[8] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: fault-tolerant stream processing at internet scale. Proceedings of the VLDB Endowment, 6(11):1033–1044, 2013.

[9] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication, SIGCOMM '08, pages 63–74. ACM, 2008.

[10] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic flow scheduling for data center networks. In NSDI'10, volume 10, pages 19–19. USENIX, 2010.

[11] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pfabric: Minimal near-optimal datacenter transport. In ACM SIGCOMM Computer Communication Review, volume 43, pages 435–446. ACM, 2013.

[12] Hilfi Alkaff, Indranil Gupta, and Luke M Leslie. Cross-layer scheduling in cloud systems. In Cloud Engineering (IC2E), 2015 IEEE International Conference on, pages 236–245. IEEE, 2015.

[13] André Allavena, Qiang Wang, Ihab Ilyas, and Srinivasan Keshav. LOT: A robust overlay for distributed range query processing. Technical report, CS-2006-21, University of Waterloo, 2006.

[14] S3A Streaming Upload, 2018. http://www.alluxio.org/docs/master/en/Configuring-Alluxio-with-S3.html#experimental-s3a-streaming-upload.

[15] Amazon Kinesis Data Streams, 2018. https://aws.amazon.com/kinesis/data-streams.

[16] Amazon Simple Storage Service (S3). https://aws.amazon.com/s3.

[17] Ganesh Ananthanarayanan, Sameer Agarwal, Srikanth Kandula, Albert Greenberg, Ion Stoica, Duke Harlan, and Ed Harris. Scarlett: coping with skewed content popularity in mapreduce clusters. In Proceedings of the sixth conference on Computer Systems, pages 287–300. ACM, 2011.

[18] Michael P Andersen and David E Culler. Btrdb: Optimizing storage system design for timeseries processing. In 14th {USENIX} Conference on File and Storage Technologies ({FAST} 16), pages 39–52, 2016.

[19] Apache Flink, 2018. https://flink.apache.org.

[20] Apache Kafka, 2018. https://kafka.apache.org.

[21] Apache Samza, 2018. http://samza.apache.org.

[22] Spark Streaming, 2018. https://spark.apache.org/streaming/.

[23] Apache Storm, 2018. http://storm.apache.org.

[24] B. Arun, S. Peluso, R. Palmieri, G. Losa, and B. Ravindran. Speeding up consensus by chasing fast decisions. In 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), DSN'17, 2017.

[25] Andreas Bader, Oliver Kopp, and Michael Falkenthal. Survey and comparison of open source time series databases. Datenbanksysteme für Business, Technologie und Web (BTW 2017)-Workshopband, 2017.

[26] Peter Bailis and Kyle Kingsbury. The network is reliable. ACM Queue, 12(7):20, 2014.

[27] Jason Baker, Chris Bond, James C Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In Proceedings of the Conference on Innovative Data system Research (CIDR), pages 223–234, 2011.

[28] Jonathan Behrens, Ken Birman, Sagar Jha, Matthew Milano, Edward Tremel, Eugene Bagdasaryan, Theo Gkountouvas, Weijia Song, and Robbert Van Renesse. Derecho: Group communication at the speed of light. Technical report, Technical Report. Cornel University, 2016.

[29] Theophilus Benson, Aditya Akella, and David A Maltz. Network traffic characteristics of data centers in the wild. In IMC, pages 267–280. ACM, 2010.

[30] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Understanding data center traffic characteristics. SIGCOMM CCR, 40:92–99, 2010.

[31] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Microte: Fine grained traffic engineering for data centers. In Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies, page 8. ACM, 2011.

[32] Alysson Neves Bessani, Ricardo Mendes, Tiago Oliveira, Nuno Ferreira Neves, Miguel Correia, Marcelo Pasin, and Paulo Verissimo. Scfs: A shared cloud-backed file system. In USENIX Annual Technical Conference, pages 169–180, 2014.

[33] Carlos Eduardo Bezerra, Daniel Cason, and Fernando Pedone. Ridge: high-throughput, low-latency atomic multicast. In Proceedings of the IEEE Symposium on Reliable Distributed Systems (SRDS), pages 256–265. IEEE, 2015.

[34] Martin Biely, Zoran Milosevic, Nuno Santos, and Andre Schiper. S-Paxos: Offloading the leader for high throughput state machine replication. In Proceedings of the IEEE Symposium on Reliable Distributed Systems (SRDS), pages 111–120. IEEE, 2012.

[35] Martin Biely, Zoran Milosevic, Nuno Santos, and Andre Schiper. S-Paxos: Offloading the leader for high throughput state machine replication. In Proceedings of the IEEE Symposium on Reliable Distributed Systems (SRDS), pages 111–120. IEEE, 2012.

[36] How Much Data Do We Create Every Day? The Mind-Blowing Stats Everyone Should Read. www.forbes.com/sites/bernardmarr/2018/05/21/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read.

[37] Peter Bodík, Ishai Menache, Mosharaf Chowdhury, Pradeepkumar Mani, David A. Maltz, and Ion Stoica. Surviving failures in bandwidth-constrained datacenters. In Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, SIGCOMM '12, pages 431–442. ACM, 2012.

[38] Jürg Bolliger and Thomas Gross. A framework based approach to the development of network aware applications. IEEE Transactions on Software Engineering, 24(5):376–390, 1998.

[39] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. ACM SIGCOMM Computer Communication Review, 44(3):87–95, 2014.

[40] Lucas Braun, Thomas Etter, Georgios Gasparis, Martin Kaufmann, Donald Kossmann, Daniel Widmer, Aharon Avitzur, Anthony Iliopoulos, Eliezer Levy, and Ning Liang. Analytics in motion: High performance event-processing and real-time analytics in the same database. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, pages 251–264. ACM, 2015.

[41] Andrew Brook. Low-latency distributed applications in finance. Commun. ACM, 58(7):4250, June 2015.

[42] Maxim Buevich, Anne Wright, Randy Sargent, and Anthony Rowe. Respawn: A distributed multi-resolution time-series datastore. In 2013 IEEE 34th Real-Time Systems Symposium, pages 288–297. IEEE, 2013.

[43] Matthew Burke, Audrey Cheng, and Wyatt Lloyd. Gryff: Unifying consensus and shared registers. In 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20). USENIX Association, 2020.

[44] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In Proceedings of the 7th Symposium on Operating Systems Design and Implementation, OSDI '06, pages 335–350. USENIX Association, 2006.

[45] Chris X Cai, Shayan Saeed, Indranil Gupta, Roy H Campbell, and Franck Le. Phurti: Application and network-aware flow scheduling for multi-tenant mapreduce clusters. In 2016 IEEE International Conference on Cloud Engineering (IC2E), pages 161–170. IEEE, 2016.

[46] Lásaro Jonas Camargos, Rodrigo Malta Schmidt, and Fernando Pedone. Multicoordinated paxos. In Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing, pages 316–317. ACM, 2007.

[47] Svante Carlsson, J Ian Munro, and Patricio V Poblete. An implicit binomial queue with constant insertion time. In Scandinavian Workshop on Algorithm Theory, pages 1–13. Springer, 1988.

[48] Self-driving cars will create 2 petabytes of data, what are the big data opportunities for the car industry? https://datafloq.com/read/self-driving-cars-create-2-petabytes-data-annually/172.

[49] Benjamin Cassell, Tyler Szepesi, Bernard Wong, Tim Brecht, Jonathan Ma, and Xiaoyi Liu. Nessie: A decoupled, client-driven key-value store using RDMA. IEEE Transactions on Parallel and Distributed Systems, 28(12):3537–3552, 2017.

[50] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99), volume 99, pages 173–186, 1999.

[51] Stephan Cejka, Ralf Mosshammer, and Alfred Einfalt. Java embedded storage for time series and meta data in smart grids. In 2015 IEEE International Conference on Smart Grid Communications (SmartGridComm), pages 434–439. IEEE, 2015.

[52] Chain blockchain infrastructure, 2017.

[53] Tushar D Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing, pages 398–407. ACM, 2007.

[54] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. Journal of the ACM (JACM), 43(4):685–722, 1996.

[55] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. Journal of the ACM (JACM), 43(2):225–267, 1996.

[56] 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 Comput. Syst., 26(2), June 2008.

[57] Aleksey Charapko, Ailidani Ailijiang, and Murat Demirbas. PigPaxos: Devouring the communication bottlenecks in distributed consensus. arXiv preprint arXiv:2003.07760, 2020.

[58] Guoqiang Jerry Chen, Janet L. Wiener, Shridhar Iyer, Anshul Jaiswal, Ran Lei, Nikhil Simha, Wei Wang, Kevin Wilfong, Tim Williamson, and Serhat Yilmaz. Real-time data processing at facebook. In Proceedings of the 2016 International Conference on Management of Data, SIGMOD 16, page 10871098, New York, NY, USA, 2016. Association for Computing Machinery.

[59] Wenyan Chen, Kejiang Ye, and Cheng-Zhong Xu. Co-Locating Online Workload and Offline Workload in the Cloud: An Interference Analysis. In 2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS), pages 2278–2283. IEEE, 2019.

[60] Mosharaf Chowdhury, Srikanth Kandula, and Ion Stoica. Leveraging endpoint flexibility in data-intensive clusters. In SIGCOMM'13, pages 231–242. ACM, 2013.

[61] Mosharaf Chowdhury and Ion Stoica. Coflow: A networking abstraction for cluster applications. In Proceedings of the 11th ACM Workshop on Hot Topics in Networks, pages 31–36. ACM, 2012.

[62] Mosharaf Chowdhury and Ion Stoica. Efficient coflow scheduling without prior knowledge. In ACM SIGCOMM Computer Communication Review, volume 45, pages 393–406. ACM, 2015.

[63] Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I Jordan, and Ion Stoica. Managing data transfers in computer clusters with orchestra. ACM SIGCOMM Computer Communication Review, 41(4):98–109, 2011.

[64] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. Efficient coflow scheduling with varys. In ACM SIGCOMM Computer Communication Review, volume 44, pages 443–454. ACM, 2014.

[65] Chronicle Map: High performance, off-heap, key-value store. http://chronicle.software/products/chronicle-map.

[66] CockroachDB, 2018. https://www.cockroachlabs.com.

[67] The Coco Framework, 2017. https://github.com/Azure/coco-framework.

[68] Manuel Coenen, Christoph Wagner, Alexander Echler, and Sebastian Frischbier. Benchmarking financial data feed systems. In Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems, pages 252–253, 2019.

[69] Consul: Service Networking Across Any Cloud, 2018. https://www.consul.io.

[70] Brian F Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. Proceedings of the VLDB Endowment, 1(2):1277–1288, 2008.

[71] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Googles globally distributed database. ACM Transactions on Computer Systems (TOCS), 31(3):8, 2013.

[72] Securities Industry Automation Corporation. Consolidated Tape System Multicast Output Binary Specification, 2020. https://www.ctaplan.com/publicdocs/ctaplan/CTS_Pillar_Output_Specification.pdf.

[73] Andrew R. Curtis, Jeffrey C. Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. Devoflow: scaling flow management for high-performance networks. In SIGCOMM'11, pages 254–265, 2011.

[74] Huynh Tu Dang, Pietro Bressana, Han Wang, Ki Suh Lee, Hakim Weatherspoon, Marco Canini, Fernando Pedone, and Robert Soulé. Network hardware-accelerated consensus. arXiv preprint arXiv:1605.05619, 2016.

[75] Huynh Tu Dang, Marco Canini, Fernando Pedone, and Robert Soulé. Paxos made switch-y. SIGCOMM Computer Communication Review (CCR), 46(2):18–24, May 2016.

[76] Huynh Tu Dang, Marco Canini, Fernando Pedone, and Robert Soulé. Paxos made switch-y. ACM SIGCOMM Computer Communication Review, 46(2):18–24, 2016.

[77] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. Netpaxos: Consensus at network speed. In Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR), pages 5:1–5:7. ACM, 2015.

[78] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. Netpaxos: Consensus at network speed. In Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research, page 5. ACM, 2015.

[79] Market Data Peaks. https://www.marketdatapeaks.net/rates/usa/.

[80] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. Communications of the ACM, 51(1):107113, 2008.

[81] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. Communications of the ACM, 51(1):107113, January 2008.

[82] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. ACM Computing Surveys (CSUR), 36(4):372–421, 2004.

[83] A list of distributed filesystems. https://en.wikipedia.org/wiki/List_of_file_systems#Distributed_file_systems.

[84] Fahad R Dogar, Thomas Karagiannis, Hitesh Ballani, and Antony Rowstron. Decentralized task-aware scheduling for data center networks. In ACM SIGCOMM Computer Communication Review, volume 44, pages 431–442. ACM, 2014.

[85] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. Journal of the ACM (JACM), 34(1):77–97, 1987.

[86] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast Remote Memory. In 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14), pages 401–414. USENIX Association, 2014.

[87] Ted Dunning and Ellen Friedman. Time series databases new ways to store and access data. Oreilly and Associates, 2014.

[88] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. Journal of the ACM (JACM), 35(2):288–323, 1988.

[89] Etcd: A distributed, reliable key-value store. https://coreos.com/etcd.

[90] Ethereum Blockchain Platform, 2017.

[91] What types of financial market data providers are there? https://www.exegy.com/2019/07/types-financial-market-data-providers/.

[92] Nathan Farrington and Alexey Andreyev. Facebook's data center network architecture. In IEEE Optical Interconnects Conference, pages 5–7, 2013.

[93] Facebooks Top Open Data Problems. www.research.fb.com/blog/2014/10/facebook-s-top-open-data-problems/.

[94] Tranquility Data Ingestion Connector. www.research.fb.com/blog/2014/10/facebook-s-top-open-data-problems/.

[95] U. Feige, D. Peleg, P. Raghavan, and E. Upfal. Randomized broadcast in networks. Random Structures and Algorithms, 1:447–460, 1990.

[96] Dennis Fetterly, Maya Haridasan, Michael Isard, and Swaminathan Sundararaman. TidyFS: A simple and small distributed file system. In USENIX annual technical conference, pages 34–34, 2011.

[97] Twitter's finagle rpc. https://twitter.github.io/finagle/.

[98] Finra trade data dissemination service data feed interface specification. https://www.finra.org/sites/default/files/TDDS2.0-version11.pdf.

[99] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. Journal of the ACM (JACM), 32(2):374–382, 1985.

[100] Daniel Ford, François Labelle, Florentina Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in globally distributed storage systems. 2010.

[101] Sebastian Frischbier, Mario Paic, Alexander Echler, and Christian Roth.

[102] The future of streaming analytics in financial services. https://financial-fraud-detection.cfotechoutlook.com/cxoinsights/the-future-of-streaming-analytics-in-financial-services-nid-227.html.

[103] Eli Gafni and Leslie Lamport. Disk paxos. In International Symposium on Distributed Computing, pages 330–344. Springer, 2000.

[104] Vijay K Garg and Michel Raynal. Normality: A consistency condition for concurrent objects. Parallel Processing Letters, 9(01):123–134, 1999.

[105] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In SIGOPS Operating Systems Review, volume 37, pages 29–43. ACM, 2003.

[106] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: Measurement, analysis, and implications. In Proceedings of the ACM SIGCOMM Conference, SIGCOMM '11, pages 350–361. ACM, 2011.

[107] Michel Goossens, Frank Mittelbach, and Alexander Samarin. The LaTeX Companion. Addison-Wesley, Reading, Massachusetts, 1994.

[108] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: a scalable and flexible data center network. Communications of the ACM, 54(3):95–104, 2011.

[109] gRPC: A high performance, open source universal RPC framework. https://grpc.io/.

[110] Indranil Gupta, Robbert Van Renesse, and Kenneth P Birman. Scalable fault-tolerant aggregation in large process groups. In Dependable Systems and Networks, 2001. DSN 2001. International Conference on, pages 433–442. IEEE, 2001.

[111] Trinabh Gupta, Rayman Preet Singh, Amar Phanishayee, Jaeyeon Jung, and Ratul Mahajan. Bolt: Data management for connected homes. In 11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14), pages 243–256, 2014.

[112] Apache Hadoop. http://hadoop.apache.org.

[113] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical report, 1994.

[114] Nicholas JA Harvey and Kevin Zatloukal. The post-order heap. In Proceedings of the 3rd International Conference on Fun with Algorithms, 2004.

[115] HBase. https://hbase.apache.org.

[116] HDFS Design and Architecture. https://hadoop.apache.org/docs/r2.7.3/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html.

[117] HDFS Architecture. https://hadoop.apache.org/docs/r3.3.0/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html.

[118] Maurice Herlihy. Wait-Free Synchronization. ACM Transactions on Programming Languages and Systems, 13(1):124149, January 1991.

[119] Ezra N Hoch, Yaniv Ben-Yehuda, Noam Lewis, and Avi Vigder. Bizur: A key-value consensus algorithm for scalable file-systems. arXiv preprint arXiv:1702.04242, 2017.

[120] Christian E Hopps. RFC 2992: Analysis of an equal-cost multi-path algorithm, 2000. www.tools.ietf.org/html/rfc2992.

[121] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. Flexible paxos: Quorum intersection revisited. arXiv preprint arXiv:1608.06696, 2016.

[122] John H Howard. An overview of the andrew file system. In in Proceedings of USENIX Winter Technical Conference, pages 23–26, 1988.

[123] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference, USENIX ATC '10, page 9. USENIX Association, 2010.

[124] Felix Hupfeld, Toni Cortes, Björn Kolbeck, Jan Stender, Erich Focht, Matthias Hess, Jesus Malo, Jonathan Marti, and Eugenio Cesario. The xtreemfs architecturea case for object-based file systems in grids. Concurrency and computation: Practice and experience, 20(17):2049–2060, 2008.

[125] Hyperledger Blockchain Technologies, 2017. www.hyperledger.org.

[126] InfluxDB: An open source time series database. www.influxdata.com.

[127] Read consistency in dynamodb. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.ReadConsistency.html.

[128] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In ACM SIGOPS operating systems review, volume 41, pages 59–72. ACM, 2007.

[129] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. Consensus in a box: Inexpensive coordination in hardware. In Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation, NSDI '16, pages 425–438, Santa Clara, CA, 2016. USENIX Association.

[130] Raj Jain and Shawn Routhier. Packet trains–measurements and a new model for computer network traffic. IEEE journal on selected areas in Communications, 4(6):986–995, 1986.

[131] Virajith Jalaparti, Peter Bodik, Ishai Menache, Sriram Rao, Konstantin Makarychev, and Matthew Caesar. Network-aware scheduling for data-parallel jobs: Plan when you can. ACM SIGCOMM Computer Communication Review, 45(4):407–420, 2015.

[132] M Haseeb Javed, Xiaoyi Lu, and Dhabaleswar K Panda. Characterization of big data stream processing pipeline: a case study using flink and kafka. In Proceedings of the Fourth IEEE/ACM International Conference on Big Data Computing, Applications and Technologies, pages 1–10, 2017.

[133] Søren Kejser Jensen, Torben Bach Pedersen, and Christian Thomsen. Time series management systems: A survey. IEEE Transactions on Knowledge and Data Engineering, 29(11):2581–2600, 2017.

[134] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-rtt coordination. In 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18). USENIX Association, 2018.

[135] Flavio P Junqueira, Benjamin C Reed, and Marco Serafini. ZAB: High-performance broadcast for primary-backup systems. In Proceedings of the IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN '11), pages 245–256. IEEE, 2011.

[136] Saurabh Kadekodi, Bin Fan, Adit Madan, Garth A Gibson, and Gregory R Ganger. A case for packing and indexing in cloud file systems. In 10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18). USENIX Association, 2018.

[137] Kadena permissioned blockchain, 2017. http://kadena.io.

[138] Elena Kakoulli and Herodotos Herodotou. Octopusfs: A distributed file system with tiered storage management. In Proceedings of the 2017 ACM International Conference on Management of Data, pages 65–78. ACM, 2017.

[139] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using RDMA efficiently for key-value services. In Proceedings of the 2014 ACM Conference on SIGCOMM, SIGCOMM14, page 295306, New York, NY, USA, 2014. Association for Computing Machinery.

[140] Anuj Kalia, Michael Kaminsky, and David G. Andersen. FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram rpcs. In 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), pages 185–201, Savannah, GA, November 2016. USENIX Association.

[141] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. The nature of data center traffic: measurements & analysis. In IMC, pages 202–208. ACM, 2009.

[142] Alfons Kemper and Thomas Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In Data Engineering (ICDE), 2011 IEEE 27th International Conference on, pages 195–206. IEEE, 2011.

[143] Srinivasan Keshav, W Golab, Bernard Wong, Sajjad Rizvi, and Sergey Gorbunov. RCanopus: Making canopus resilient to failures and byzantine faults. arXiv preprint arXiv:1810.09300, 2018.

[144] A Month of Kinesis in Production. https://brandur.org/kinesis-in-production.

[145] Amazon Kinesis Data Firehose. https://aws.amazon.com/kinesis/data-firehose.

[146] Amazon Kinesis Data Streams pricing, 2018. https://aws.amazon.com/kinesis/data-streams/pricing/.

[147] Donald Knuth. The TEXbook. Addison-Wesley, Reading, Massachusetts, 1986.

[148] Cassie Kozyrkov. Automated Inspiration, 2019. https://www.forbes.com/sites/insights-intelai/2019/05/22/automated-inspiration/#73d7d5ea1c44.

[149] Jay Kreps. Questioning the lambda architecture, 2014. https://www.oreilly.com/radar/questioning-the-lambda-architecture/.

[150] Ajay D Kshemkalyani and Mukesh Singhal. Distributed computing: principles, algorithms, and systems. Cambridge University Press, 2011.

[151] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter Heron: Stream processing at scale. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, pages 239–250. ACM, 2015.

[152] Akhil Kumar. Hierarchical Quorum Consensus: A new algorithm for managing replicated data. IEEE transactions on Computers, 40(9):996–1004, 1991.

[153] Katrina LaCurts, Shuo Deng, Ameesh Goyal, and Hari Balakrishnan. Choreo: Network-aware task placement for cloud applications. In Proceedings of the Internet measurement conference (IMC), pages 191–204. ACM, 2013.

[154] Katrina LaCurts, Shuo Deng, Ameesh Goyal, and Hari Balakrishnan. Choreo: Network-aware task placement for cloud applications. In Proceedings of the 2013 conference on Internet measurement conference, pages 191–204. ACM, 2013.

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

[156] Leslie Lamport. The implementation of reliable distributed multiprocess systems. Computer Networks (1976), 2(2):95–114, 1978.

[157] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. Communications of the ACM, 21(7):558–565, 1978.

[158] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Transactions on Computers, 100(9):690–691, 1979.

[159] Leslie Lamport. LaTeX — A Document Preparation System. Addison-Wesley, Reading, Massachusetts, second edition, 1994.

[160] Leslie Lamport. The part-time parliament. ACM Transactions on Computer Systems (TOCS), 16(2):133–169, 1998.

[161] Leslie Lamport. Paxos made simple. ACM SIGACT News, 32(4):18–25, 2001.

[162] Leslie Lamport. Generalized consensus and paxos. Technical report, Microsoft Research, March 2005.

[163] Leslie Lamport. Generalized consensus and paxos. Technical report, MSR-TR-2005-33, Microsoft Research, 2005.

[164] Leslie Lamport. Fast paxos. Springer Distributed Computing, 19(2):79–103, 2006.

[165] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Stoppable paxos. TechReport, Microsoft Research, 2008.

[166] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical paxos and primary-backup replication. In Proceedings of the 28th ACM symposium on Principles of distributed computing (PODC), pages 312–313. ACM, 2009.

[167] Leslie Lamport and Mike Massa. Cheap paxos. In Dependable Systems and Networks, 2004 International Conference on, pages 307–314. IEEE, 2004.

[168] Butler Lampson. The abcd's of paxos. In Proceedings of PODC, volume 1, page 13, 2001.

[169] Butler W Lampson. How to build a highly available system using consensus. In Distributed Algorithms, pages 1–17. Springer, 1996.

[170] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: Rapid prototyping for software-defined networks. In SIGCOMM HotNets workshop, pages 19:1–19:6. ACM, 2010.

[171] Leveldb key-value store. www.github.com/google/leveldb.

[172] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In Proceedings of the ACM Symposium on Cloud Computing, pages 1–15. ACM, 2014.

[173] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. Just say NO to paxos overhead: Replacing consensus with network ordering. In Proceedings of 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), pages 467–483. USENIX Association, 2016.

[174] Jialin Li, Ellis Michael, Naveen Kr Sharma, Adriana Szekeres, and Dan RK Ports. Just say no to paxos overhead: Replacing consensus with network ordering. In OSDI, pages 467–483, 2016.

[175] Min Li, Dinesh Subhraveti, Ali R Butt, Aleksandr Khasymski, and Prasenjit Sarkar. CAM: A topology aware minimum cost flow based resource manager for mapreduce applications in the cloud. In Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing, pages 211–222. ACM, 2012.

[176] Zhaogeng Li, Jun Bi, Yiran Zhang, and Chuang Wang. Eaalo: Enhanced coflow scheduling without prior knowledge in a datacenter network. In Computers and Communications (ISCC), 2017 IEEE Symposium on, pages 1136–1141. IEEE, 2017.

[177] Jimmy Lin. The lambda and the kappa. IEEE Internet Computing, (5):60–66, 2017.

[178] Fagui LIU and Yingyi YANG. D-Paxos: Building hierarchical replicated state machine for cloud environments. IEICE Transactions on Information and Systems, E99.D(6):1485–1501, 2016.

[179] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas Anderson. F10: A fault-tolerant engineered network. In In Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13), pages 399–412. USENIX Association, 2013.

[180] Vincent Liu, Danyang Zhuo, Simon Peter, Arvind Krishnamurthy, and Thomas Anderson. Subways: A case for redundant, inexpensive data center edge links. In Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies, CoNEXT '15, pages 27:1–27:13. ACM, 2015.

[181] Xin Liu and Kenneth Salem. Hybrid storage management for database systems. Proceedings of the VLDB Endowment, 6(8):541–552, 2013.

[182] Internet live stats. www.internetlivestats.com.

[183] Rafik Makhloufi, Guillaume Doyen, Grégory Bonnet, and Dominique Gaïti. A survey and performance evaluation of decentralized aggregation schemes for autonomic management. International Journal of Network Management, 24(6):469–498, 2014.

[184] Yanhua Mao, Flavio Paiva Junqueira, and Keith Marzullo. Mencius: building efficient replicated state machines for wans. In Proceedings of the 8th USENIX conference on Operating systems design and implementation, volume 8 of OSDI '08, pages 369–384, 2008.

[185] Yanhua Mao, Flavio Paiva Junqueira, and Keith Marzullo. Mencius: building efficient replicated state machines for wans. In Proceedings of the 8th USENIX conference on Operating systems design and implementation, volume 8 of OSDI '08, pages 369–384, 2008.

[186] MapR Filesystem. https://mapr.com.

[187] Parisa Jalili Marandi, Marco Primi, and Fernando Pedone. Multi-ring paxos. In Proceedings of the 2012 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN '12, pages 1–12. IEEE, 2012.

[188] Parisa Jalili Marandi, Marco Primi, and Fernando Pedone. Multi-ring paxos. In Proceedings of the 2012 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN '12, pages 1–12. IEEE, 2012.

[189] Parisa Jalili Marandi, Marco Primi, Nicolas Schiper, and Fernando Pedone. Ring paxos: A high-throughput atomic broadcast protocol. In Proceedings of the 2010 IEEE/IFIP International Conference on Dependable Systems Networks, DSN '10, pages 527–536. IEEE, 2010.

[190] Nathan Marz. How to beat the CAP theorem, 2011. http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html.

[191] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, PODC'96, page 267275, New York, NY, USA, 1996. Association for Computing Machinery.

[192] Ginger Milton. Towards a New Data Landscape. https://www.linkedin.com/pulse/towards-new-data-landscape-jennifer-milton.

[193] MinIO Object Store. https://min.io.

[194] Masaaki Mizuno, Michel Raynal, and James Z. Zhou. Sequential consistency in distributed systems. In Theory and Practice in Distributed Systems, pages 224–241. Springer, 1995.

[195] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In Proceedings of the 24th ACM Symposium on Operating Systems Principles, SOSP '13, pages 358–372. ACM, 2013.

[196] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In Proceedings of the 24th ACM Symposium on Operating Systems Principles, SOSP '13, pages 358–372. ACM, 2013.

[197] Iulian Moraru, David G Andersen, and Michael Kaminsky. EPaxos source code, 2014.

[198] Iulian Moraru, David G Andersen, and Michael Kaminsky. Paxos quorum leases: Fast reads without sacrificing writes. In Proceedings of SoCC. ACM, 2014.

[199] Iulian Moraru, David G Andersen, and Michael Kaminsky. Paxos quorum leases: Fast reads without sacrificing writes. In Proceedings of the ACM Symposium on Cloud Computing, pages 1–13. ACM, 2014.

[200] Iulian Moraru, David G Andersen, and Michael Kaminsky. Quorum read leases, 2014.

[201] Gareth W Morris, David B Thomas, and Wayne Luk. Fpga accelerated low-latency market data feed processing. In 17th IEEE Symposium on High Performance Interconnects, pages 83–89. IEEE, 2009.

[202] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. Consolidating concurrency control and consensus for commits under conflicts. In Proceedings of USENIX OSDI, pages 517–532. USENIX Association, 2016.

[203] Ali Munir, Ting He, Ramya Raghavendra, Franck Le, and Alex X Liu. Network scheduling aware task placement in datacenters. In Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies, pages 221–235. ACM, 2016.

[204] Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. Dpaxos: Managing data closer to users for low-latency and mobile applications. In Proceedings of the 2018 International Conference on Management of Data, pages 1221–1236. ACM, 2018.

[205] Edmund B Nightingale, Jeremy Elson, Jinliang Fan, Owen S Hofmann, Jon Howell, and Yutaka Suzue. Flat datacenter storage. In OSDI, pages 1–15, 2012.

[206] Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat. Portland: A scalable fault-tolerant layer 2 data center network fabric. In Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication, SIGCOMM '09, pages 39–50. ACM, 2009.

[207] ObjectiveFS, 2018. https://objectivefs.com/.

[208] Brian M Oki and Barbara H Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In Proceedings of the seventh annual ACM Symposium on Principles of distributed computing, pages 8–17. ACM, 1988.

[209] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In Proceedings of the 2008 ACM SIGMOD international conference on Management of data, pages 1099–1110. ACM, 2008.

[210] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In 2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14), pages 305–319, 2014.

[211] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In Proceedings of USENIX ATC, pages 305–320. USENIX Association, 2014.

[212] OpenFlow Swtich Specification. https://opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf.

[213] OpenTSDB: The Scalable Time Series Database. www.opentsdb.net.

[214] Options Price Reporting Authority. https://www.opraplan.com/.

[215] Siac autolink facility userguide. https://www.opraplan.com/document-library.

[216] Common IP Multicast Distribution Network Recipient Interface Specification. https://www.opraplan.com/document-library.

[217] OPRA Binary Data Recipient Interface Specification. https://www.opraplan.com/document-library.

[218] Oracle Cluster File System for Linux, 2018.

[219] Patrick ONeil, Edward Cheng, Dieter Gawlick, and Elizabeth ONeil. The log-structured merge-tree (lsm-tree). Acta Informatica, 33(4):351–385, 1996.

[220] Leandro Pacheco, Raluca Halalai, Valerio Schiavoni, Fernando Pedone, Etienne Rivière, and Pascal Felber. GlobalFS: A strongly consistent multi-site file system. In Reliable Distributed Systems (SRDS), 2016 IEEE 35th Symposium on, pages 147–156. IEEE, 2016.

[221] Mark Palmer. The Future of Streaming Analytics in Financial Services. https://www.linkedin.com/pulse/future-streaming-analytics-financial-services-mark-palmer.

[222] Swapnil V Patil, Garth A Gibson, Sam Lang, and Milo Polte. Giga+: scalable directories for shared file systems. In Proceedings of the 2nd international workshop on Petascale data storage: held in conjunction with Supercomputing'07, pages 26–29. ACM, 2007.

[223] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. Journal of the ACM (JACM), 27(2):228–234, 1980.

[224] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. Gorilla: A fast, scalable, in-memory time series database. Proceedings of the VLDB Endowment, 8(12):1816–1827, 2015.

[225] István Pelle, János Czentye, János Dóka, and Balázs Sonkoly. Towards latency sensitive cloud native applications: A performance study on aws. In 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), pages 272–280. IEEE, 2019.

[226] Sebastiano Peluso, Alexandru Turcu, Roberto Palmieri, Giuliano Losa, and Binoy Ravindran. Making fast consensus generally faster. In 46th International Conference on Dependable Systems and Networks (DSN), DSN'16, pages 156–167. IEEE/IFIP, 2016.

[227] Meikel Poess, Tilmann Rabl, and Hans-Arno Jacobsen. Analysis of TPC-DS: the first standard benchmark for SQL-based big data systems. In Proceedings of the 2017 Symposium on Cloud Computing, pages 573–585, 2017.

[228] Marius Poke and Torsten Hoefler. DARE: High-performance state machine replication on rdma networks. In Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing (HPDC'15), pages 107–118. ACM, 2015.

[229] Marius Poke, Torsten Hoefler, and Colin W. Glass. Allconcur: Leaderless concurrent atomic broadcast. In 26th International Symposium on High-Performance Parallel and Distributed Computing, HPDC'17, pages 205–218. ACM, 2017.

[230] Lucian Popa, Sylvia Ratnasamy, Gianluca Iannaccone, Arvind Krishnamurthy, and Ion Stoica. A cost comparison of datacenter network architectures. In Proceedings of the 6th International COnference on emerging Networking EXperiments and Techn CoNEXT10, New York, NY, USA, 2010. Association for Computing Machinery.

[231] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. Designing distributed systems using approximate synchrony in data center networks. In Proceedings of 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15), pages 43–57, Oakland, CA, 2015. USENIX Association.

[232] Anna Povzner and Scott Hendricks. 99th percentile latency at scale with apache kafka, 2020. https://www.confluent.io/blog/configure-kafka-to-minimize-latency/.

[233] Yunhui Qiu, Jinyu Xie, Hankun Lv, Wenbo Yin, Wai-Shing Luk, Lingli Wang, Bowei Yu, Hua Chen, Xianjun Ge, Zhijian Liao, et al. FULL-KV: Flexible and ultra-low-latency in-memory key-value store system design on cpu-fpga. IEEE Transactions on Parallel and Distributed Systems, 31(8):1828–1444, 2020.

[234] John Risson and Tim Moors. Survey of research towards robust peer-to-peer networks: search methods. Computer networks, 50(17):3485–3521, 2006.

[235] Sajjad Rizvi, Xi Li, Bernard Wong, Fiodar Kazhamiaka, and Benjamin Cassell. Mayflower: Improving distributed filesystem performance through SDN/Filesystem co-design. In Distributed Computing Systems (ICDCS), 2016 IEEE 36th International Conference on, pages 384–394. IEEE, 2016.

[236] Sajjad Rizvi, Bernard Wong, and Srinivasan Keshav. Canopus: A Scalable and Massively Parallel Consensus Protocol. Technical report, CS-2017-08, University of Waterloo, 2017.

[237] Sajjad Rizvi, Bernard Wong, and Srinivasan Keshav. Canopus: A scalable and massively parallel consensus protocol. In Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies, pages 426–438. ACM, 2017.

[238] Sajjad Rizvi, Bernard Wong, and Srinivasan Keshav. Canopus: A Scalable and Massively Parallel Consensus Protocol. Technical report, CS-2017-08, University of Waterloo, 2017.

[239] Robert B Ross, Rajeev Thakur, et al. Pvfs: A parallel file system for linux clusters. In Proceedings of the 4th annual Linux showcase and conference, pages 391–430, 2000.

[240] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. Inside the social network's (datacenter) network. In Proceedings of SIGCOMM, pages 123–137. ACM, 2015.

[241] Lukas Rupprecht. Network-aware big data processing. 2017.

[242] Lukas Rupprecht, William Culhane, and Peter Pietzuch. SquirrelJoin: Network-aware distributed join processing with lazy partitioning. Proceedings of the VLDB Endowment, 10(11):1250–1261, 2017.

[243] Top 5 Reasons for Choosing S3 over HDFS, 2018. https://databricks.com/blog/2017/05/31/top-5-reasons-for-choosing-s3-over-hdfs.html.

[244] Jerome H Saltzer, David P Reed, and David D Clark. End-to-end arguments in system design. ACM Transactions on Computer Systems (TOCS), 2(4):277–288, 1984.

[245] Mahadev Satyanarayanan, Victor Bahl, Ramón Caceres, and Nigel Davies. The case for vm-based cloudlets in mobile computing. IEEE pervasive Computing, 2009.

[246] Simple binary encoding (sbe). https://www.fixtrading.org/standards/sbe-online/.

[247] Brandon Schlinker, Radhika Niranjan Mysore, Sean Smith, Jeffrey C. Mogul, Amin Vahdat, Minlan Yu, Ethan Katz-Bassett, and Michael Rubin. Condor: Better topologies through declarative design. In Proceedings of the ACM SIGCOMM 2015 Conference, pages 449–463. ACM, 2015.

[248] Frank B Schmuck and Roger L Haskin. GPFS: A shared-disk file system for large computing clusters. In FAST, volume 2, 2002.

[249] Fred B Schneider. Synchronization in distributed programs. Technical report, Cornell University, 1979.

[250] Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. ACM Computing Surveys (CSUR), 22(4):299–319, 1990.

[251] Philip Schwan et al. Lustre: Building a file system for 1000-node clusters. In Proceedings of the 2003 Linux symposium, volume 2003, pages 380–386, 2003.

[252] Xuanhua Shi, Haohong Lin, Hai Jin, Bing Bing Zhou, Zuoning Yin, Sheng Di, and Song Wu. GIRAFFE: A scalable distributed coordination service for large-scale systems. In Proceedings of IEEE Cluster Computing (CLUSTER), pages 38–47. IEEE, 2014.

[253] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies, pages 1–10. IEEE, May 2010.

[254] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, et al. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In Porceedings of SIGCOMM, pages 183–197. ACM, 2015.

[255] Swaminathan Sivasubramanian. Amazon DynamoDB: A Seamlessly Scalable Non-Relational Database Service. In Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, SIGMOD12, page 729730. Association for Computing Machinery, 2012.

[256] Smash.BI Inc. http://www.smash.bi.

[257] Weijia Song, Theo Gkountouvas, Ken Birman, Qi Chen, and Zhen Xiao. The freeze-frame file system. In Proceedings of the Seventh ACM Symposium on Cloud Computing, pages 307–320. ACM, 2016.

[258] Storage Performance Council (SPC). http://storageperformance.org.

[259] Michael Stonebraker, Uundefinedur Çetintemel, and Stan Zdonik. The 8 requirements of real-time stream processing. SIGMOD Record, 34(4):4247, December 2005.

[260] Michael Stonebraker and Ugur Cetintemel. One Size Fits All: An Idea Whose Time Has Come and Gone. In Proceedings of the 21st International Conference on Data Engineering, ICDE05, page 211, USA, 2005. IEEE Computer Society.

[261] Jeremy Stribling, Yair Sovran, Irene Zhang, Xavid Pretzer, Jinyang Li, M Frans Kaashoek, and Robert Morris. Flexible, wide-area storage for distributed systems with WheelFS. In NSDI, volume 9, pages 43–58, 2009.

[262] Hari Subramoni, Fabrizio Petrini, Virat Agarwal, and Davide Pasetto. Streaming, Low-latency Communication in On-line Trading Systems. In Proceedings of International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), pages 1–8. IEEE, 2010.

[263] Hari Subramoni, Fabrizio Petrini, Virat Agarwal, and Davide Pasetto. Streaming, low-latency communication in on-line trading systems. In 2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), pages 1–8. IEEE, 2010.

[264] Pierre Sutra and Marc Shapiro. Fast genuine generalized consensus. In Proceedings of the IEEE Symposium on Reliable Distributed Systems (SRDS), pages 255–264. IEEE, 2011.

[265] Chunqiang Tang, Thawan Kooburat, Pradeep Venkatachalam, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. Holistic configuration management at facebook. In Proceedings of the 25th Symposium on Operating Systems Principles, pages 328–343, 2015.

[266] Tendermint blockchain consensus, 2017. https://tendermint.com.

[267] Jeff Terrace and Michael J Freedman. Object storage on craq: High-throughput chain replication for read-mostly workloads. In USENIX Annual Technical Conference, pages 11–11. San Diego, CA, 2009.

[268] Chandramohan A Thekkath, Timothy Mann, and Edward K Lee. Frangipani: A scalable distributed file system. 31(5), 1997.

[269] Alexander Thomson and Daniel J Abadi. CalvinFS: Consistent wan replication and scalable metadata management for distributed file systems. In FAST, pages 1–14, 2015.

[270] Apache thrift rpc. https://thrift.apache.org.

[271] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Antony, Hao Liu, and Raghotham Murthy. Hive-a petabyte scale data warehouse using hadoop. In Data Engineering (ICDE), 2010 IEEE 26th International Conference on, pages 996–1005. IEEE, 2010.

[272] Ashish Thusoo, Zheng Shao, Suresh Anthony, Dhruba Borthakur, Namit Jain, Joydeep Sen Sarma, Raghotham Murthy, and Hao Liu. Data Warehousing and Analytics Infrastructure at Facebook. In Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, SIGMOD10, page 10131020, New York, NY, USA, 2010. Association for Computing Machinery.

[273] Xinhui Tian, Rui Han, Lei Wang, Gang Lu, and Jianfeng Zhan. Latency critical big data computing in finance. The Journal of Finance and Data Science, 1(1):33–41, 2015.

[274] Brian L Tierney, Dan Gunter, Jason Lee, Martin Stoufer, and Joseph B Evans. Enabling network-aware applications. In High Performance Distributed Computing, 2001. Proceedings. 10th IEEE International Symposium on, pages 281–288. IEEE, 2001.

[275] Transaction Processing Performance Council (TPC). https://tpc.org.

[276] Fung Po Tso, Gregg Hamilton, Rene Weber, Colin S Perkins, and Dimitrios P Pezaros. Longer is better: exploiting path diversity in data center networks. In Distributed Computing Systems (ICDCS), 2013 IEEE 33rd International Conference on, pages 430–439. IEEE, 2013.

[277] Robbert Van Renesse, Kenneth P. Birman, and Werner Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. ACM Transactions on Computer Systems (TOCS), 21(2):164–206, May 2003.

[278] Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In Proceedings of the 6th Conference on Symposium

on Opearting Systems Design & Implementation - Volume 6, OSDI'04, pages 7–7. USENIX Association, 2004.

[279] Robbert Van Renesse and Fred B Schneider. Chain replication for supporting high throughput and availability. In OSDI, volume 4, 2004.

[280] Jorge Veiga, Roberto R Expósito, and Juan Touriño. Performance evaluation of big data analysis., 2019.

[281] Michael Vrable, Stefan Savage, and Geoffrey M Voelker. Bluesky: A cloud-backed file system for the enterprise. In Proceedings of the 10th USENIX conference on File and Storage Technologies, pages 19–19. USENIX Association, 2012.

[282] Marko Vukolić. The quest for scalable blockchain fabric: Proof-of-work vs. bft replication. In International Workshop on Open Problems in Network Security, pages 112–125. Springer, 2015.

[283] Vereinigte Wirtschaftsdienste GmbH. https://www.vwd.com.

[284] Cheng Wang, Jianyu Jiang, Xusheng Chen, Ning Yi, and Heming Cui. Apus: Fast and scalable paxos on rdma. In Proceedings of the 2017 Symposium on Cloud Computing, pages 94–107. ACM, 2017.

[285] Shaoqi Wang, Xiaobo Zhou, Liqiang Zhang, and Changjun Jiang. Network-adaptive scheduling of data-intensive parallel jobs with dependencies in clusters. In Autonomic Computing (ICAC), 2017 IEEE International Conference on, pages 155–160. IEEE, 2017.

[286] Wei Wei, Harry Tian Gao, Fengyuan Xu, and Qun Li. Fast mencius: Mencius with low commit latency. In Proceedings of IEEE INFOCOM, pages 881–889, 2013.

[287] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In Proceedings of the 7th symposium on Operating systems design and implementation, pages 307–320. USENIX Association, 2006.

[288] Sage A Weil, Scott A Brandt, Ethan L Miller, and Carlos Maltzahn. CRUSH: controlled, scalable, decentralized placement of replicated data. In Proceedings of the 2006 ACM/IEEE conference on Supercomputing, page 122. ACM, 2006.

[289] Sage A Weil, Kristal T Pollack, Scott A Brandt, and Ethan L Miller. Dynamic metadata management for petabyte-scale file systems. In Proceedings of the 2004 ACM/IEEE conference on Supercomputing, page 4. IEEE Computer Society, 2004.

[290] Fangjin Yang, Eric Tschetter, Xavier Léauté, Nelson Ray, Gian Merlino, and Deep Ganguli. Druid: A real-time analytical data store. In Proceedings of the 2014 ACM SIGMOD international conference on Management of data, pages 157–168. ACM, 2014.

[291] Minlan Yu, Jennifer Rexford, Michael J. Freedman, and Jia Wang. Scalable flow-based networking with DIFANE. In SIGCOMM'10, pages 351–362, 2010.

[292] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In Proceedings of the 5th European conference on Computer systems, pages 265–278. ACM, 2010.

[293] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: A unified engine for big data processing. Communications of the ACM, 59(11):56–65, October 2016.

[294] Hong Zhang, Li Chen, Bairen Yi, Kai Chen, Mosharaf Chowdhury, and Yanhui Geng. Coda: Toward automatically identifying and scheduling coflows in the dark. In Proceedings of the 2016 ACM SIGCOMM Conference, pages 160–173. ACM, 2016.

[295] Hanyu Zhao, Quanlu Zhang, Zhi Yang, Ming Wu, and Yafei Dai. SDPaxos: Building-efficient semi-decentralized geo-replicated state machines. In Proceedings of the ACM Symposium on Cloud Computing, SoCC'18, New York, NY, USA, 2018. Association for Computing Machinery.

[296] Yangming Zhao, Kai Chen, Wei Bai, Minlan Yu, Chen Tian, Yanhui Geng, Yiming Zhang, Dan Li, and Sheng Wang. Rapier: Integrating routing and scheduling for coflow-aware data center networks. In Computer Communications (INFOCOM), 2015 IEEE Conference on, pages 424–432. IEEE, 2015.