Alluxio: A Virtual Distributed File System

Haoyuan Li

May 7, 2018

Acknowledgement

I would like to thank my advisors, Ion Stoica and Scott Shenker, for their support and guidance throughout this journey, and for providing me with great opportunities. They are both inspiring mentors, and continue to lead by example. They always challenge me to push ideas and work further, and share their advice and experience on life and research. Being able to learn from both of them has been my great fortune.

Alluxio: A Virtual Distributed File System

by

Haoyuan Li

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Ion Stoica, Co-chair
Professor Scott Shenker, Co-chair
Professor John Chuang

Spring 2018

Alluxio: A Virtual Distributed File System

Abstract

Alluxio: A Virtual Distributed File System

by

Haoyuan Li

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Ion Stoica, Co-chair

Professor Scott Shenker, Co-chair

The world is entering the data revolution era. Along with the latest advancements of the Internet, Artificial Intelligence (AI), mobile devices, autonomous driving, and Internet of Things (IoT), the amount of data we are generating, collecting, storing, managing, and analyzing is growing exponentially. To store and process these data has exposed tremendous challenges and opportunities.

Over the past two decades, we have seen significant innovation in the data stack. For example, in the computation layer, the ecosystem started from the MapReduce framework, and grew to many different general and specialized systems such as Apache Spark for general data processing, Apache Storm, Apache Samza for stream processing, Apache Mahout for machine learning, Tensorflow, Caffe for deep learning, Presto, Apache Drill for SQL workloads. There are more than a hundred popular frameworks for various workloads and the number is growing. Similarly, the storage layer of the ecosystem grew from the Apache Hadoop Distributed File System (HDFS) to a variety of choices as well, such as file systems, object stores, blob stores, key-value systems, and NoSQL databases to realize different tradeoffs in cost, speed and semantics.

This increasing complexity in the stack creates challenges in multi-fold. Data is siloed in various storage systems, making it difficult for users and applications to find and access the data efficiently. For example, for system developers, it requires more work to integrate a new compute or storage component as a building block to work with the existing ecosystem. For data application developers, understanding and managing the correct way to access different data stores becomes more complex. For end users, accessing data from various and often remote data stores often results in performance penalty and semantics mismatch. For system admins, adding, removing, or upgrading an existing compute or data store or migrating data from one store to another can be arduous if the physical storage has been deeply coupled with all applications.

To address these challenges, this dissertation proposes an architecture to have a Virtual Distributed File System (VDFS) as a new layer between the compute layer and the storage layer. Adding VDFS into the stack brings many benefits. Specifically, VDFS enables global data accessibility for different compute frameworks, efficient in-memory data sharing and management across applications and data stores, high I/O performance and efficient use of network bandwidth, and the flexible choice of compute and storage. Meanwhile, as the layer to access data and collect data metrics and usage patterns, it also provides users insight into their data and can also be used to optimize the data access based on workloads.

We achieve these goals through an implementation of VDFS called Alluxio (formerly Tachyon). Alluxio presents a set of disparate data stores as a single file system, greatly reducing the complexity of storage APIs, and semantics exposed to applications. Alluxio is designed with a memory centric architecture, enabling applications to leverage memory speed I/O by simply using Alluxio. Alluxio has been deployed at hundreds of leading companies in production, serving critical workloads. Its open source community has attracted more than 800 contributors worldwide from over 200 companies.

In this dissertation, we also investigate lineage as an important technique in the VDFS to improve the write performance, and also propose DFS-Perf, a scalable distributed file system performance evaluation framework to help researchers and developers better design and implement systems in the Alluxio ecosystem.

To my family

Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank my advisors, Ion Stoica and Scott Shenker, for their support and guidance throughout this journey, and for providing me with great opportunities. They are both inspiring mentors, and continue to lead by example. They always challenge me to push ideas and work further, and share their advice and experience on life and research. Being able to learn from both of them has been my great fortune.

This thesis is the result of collaboration with many brilliant people, all of whom I am very grateful for having the opportunity to work with. Chapter 2 [84] was joint work with Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Chapter 3 [68] highlights the joint work with Rong Gu, Qianhao Dong, Joseph Gonzalez, Zhao Zhang, Shuai Wang, Yihua Huang, Scott Shenker, Ion Stoica, and Patrick P. C. Lee. Chapter 5 shows the joint work with Bin Fan, Yupeng Fu, Calvin Jia, Gene Pang, Neena Pemmaraju, Aseem Rastogi, and Jiri Simsa. In addition, many people from the AMPLab, AMPLab's sponsors, Alluxio related projects, and Alluxio's developers and users contributed to the advancement of the idea and the implementation of the vision.

Beyond people mentioned above, I would like to thank the following people: Sameer Agarwal, Ganesh Anantharayanan, Peter Bailis, Kaifei Chen, Yanpei Chen, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Aaron Davidson, Michael Franklin, Manu Goyal, Daniel Haas, Ben Hindman, Timothy Hunter, Anand Padmanabha Iyer, Michael Jordan, Dilip Joseph, Randy Katz, Andy Konwinski, Antonio Lupher, Kay Ousterhout, Adam Oliner, Aurojit Panda, Qifan Pu, Justine Sherry, Dawn Song, Evan Sparks, Liwen Sun, Shivaram Venkataraman, Rashmi Vinayak, Di Wang, Jiannan Wang, Patrick Wendell, Reynold Xin, Minlan Yu, David Zats, Kai Zeng, Zhao Zhang, and David Zhu. You made Berkeley an unforgettable experience for me and I learnt a lot from you. Thank you all!

I have been very fortunate to work with the Alluxio open source community, which consists of more than 800 contributors from over 200 companies and universities. The joint effort of this community brought the vision and the technology to fruition. Of these people, I would like to mention our open source Project Management Committee members: Eric Anderson, Andrew Audibert, Cheng Chang, Bin Fan, Yupeng Fu, Rong Gu, Jan Hentschel, Grace Huang, Calvin Jia, Luo Li, Shaoshan Liu, Baolong Mao, Gene Pang, Qifan Pu, Andrea Reale, Mingfei Shi, Jiri Simsa, Pei Sun, Chen Tian, Saverio Veltri, Gil Vernik, Haojun Wang, and Chaomin Yu. Beyond the developers, the users have also contributed and shaped the technology and the vision tremendously by providing great feedback and testing the limits of the system.

I have also been very fortunate to have the support and the trust of my colleagues, advisors, and investors at Alluxio, Inc. The company has been working together with the open source community, to realize the vision, and to bring the technology to hundreds of the leading companies globally, serving many of the most critical infrastructures in the world. For example, as of today, seven out of the ten most valuable Internet companies in the world are running Alluxio. I am humbled by and very proud of the achievements by the collective community.

I have also been very fortunate to have great teachers and mentors at different stages in my education and career. I really appreciate their advice and support.

Last but not least, I want to thank my family and my better half, Amelia. Their love and support made this possible.

Chapter 1

Introduction

The world is entering the data revolution era, along with the advancements of the Internet, Artificial Intelligence, mobile devices, autonomous driving, and Internet of Things (IoT). We are generating, collecting, storing, managing, and analyzing zettabytes of data, and the amount of data is growing exponentially [1]. The data revolution is changing how every industry operates.

Organizations and companies are leveraging tremendous amounts of data to create value. People use data to make decisions or facilitate the decision-making process. Data has already become business critical in many companies, and life critical in certain cases. For example, Internet companies use data to provide better targeted advertisements and user experiences. Financial institutions process data to detect potential fraud in real time. Manufacturing powerhouses study data to track, understand, and better design locomotive and airplane engines. Autonomous cars depend on data to function and to ensure the safety of passengers.

The data revolution has exposed tremendous challenges and opportunities in distributed computer systems. Thanks to the pioneers in both industry and academia over the past two decades, we have seen significant innovation in distributed computation frameworks and distributed storage systems.

In the computation layer, the ecosystem started from the MapReduce framework [50], and grew to many different general and specialized systems. For example, new computation frameworks include: Apache spark [146] for general data processing, Apache Storm [124] and Apache Samza [118] for stream processing, Apache Mahout [22] and H2O [71] for machine learning, Tensorflow [92] and Caffe [81] for deep learning, and Presto [110] and Apache Drill [13] for SQL workloads. Users have access to more than a hundred popular frameworks for various workloads and the number is growing.

In the storage layer, the ecosystem grew from the HDFS [121] to a variety of choices as well. Storage systems have also become more diverse, such as file systems [119, 96, 134, 60, 33, 67, 112], object stores [137, 125], blob stores [99, 36], key-value systems [51, 95, 115, 86, 40, 10, 59], NoSQL databases [17, 11], etc. Examples of storage systems include: Amazon S3 [6], Google Cloud Storage (GCS) [66], Azure Blob Store [97], Swift [125], Ceph [137], GlusterFS [65], and

EMC Elastic Cloud Storage (ECS) [54]. In the meantime, many of these storage systems are backed by technology giants, such as Amazon, Alibaba, EMC, Google, Huawei, HPE, IBM, Microsoft, and Tencent. Given the fast growing data size, more vendors are attracted and entering this ecosystem with new storage solutions.

The speed of innovation in the two layers has provided users many different options. Organizations commonly use multiple storage systems and computation frameworks in their environments. These types of environments bring the following significant challenges.

- Data Sharing and Management: Many organizations, especially large enterprises, are using more than one storage system and more than one computation framework. For example, they may use Amazon S3, along with systems such as GCS, HDFS, and EMC ECS. Efficiently sharing and managing the data siloed (mostly duplicated as well) in various storage systems with the computation frameworks and the applications is difficult.

- Performance: Most storage systems are built for generic workloads, and deployed remotely from the computation systems. In these scenarios, network and hard drive bandwidth often become the performance bottleneck. It is challenging to provide high I/O performance to compute applications.

- Flexibility: As the ecosystem is fast evolving, there are increasingly more overhead for an organization to adopt new technologies. For example, using a new storage system may lead to recompilation or even fresh development of existing applications.

People have been trying to solve the above challenges by taking the following two approaches: a) create a better (faster, cheaper, or easier to use) storage system, which becomes yet another data silo, or b) create a better (faster, easier to program, or more generic) computation framework, which is either and try to convince the ecosystem that a single framework can address all workloads.

However, over the past four decades, we have seen that neither of the two approches have worked. From the storage layer perspective, there will always be another faster, cheaper, and easier to use system to be created to disrupt the existing ones. And it is the same for the computation layer. This trend is going to continue moving forward.

This dissertation argues that instead of building the solution as part of a storage system or a computation framework, or building yet another storage system or computation framework, data centers need a Virtual Distributed File System (VDFS), to overcome the challenges listed above. VDFS abstracts away existing storage systems, and enable applications to interact with data from various storage systems efficiently.

We implement the VDFS in an open source system, called Alluxio[1] (Figure 1.2), formerly Tachyon. We evaluate Alluxio using micro-benchmarks, real-world user application workloads, as well as running it in production environments in hundreds of companies [45].

[1]www.alluxio.org

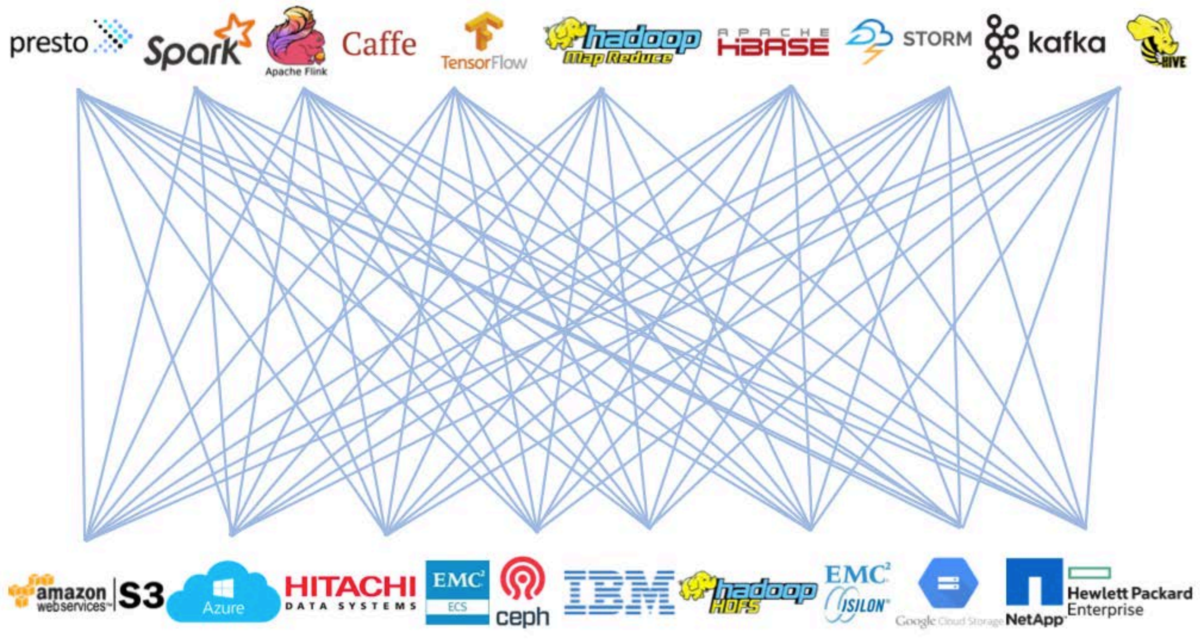


Figure 1.1: Complex Data Ecosystem

Thesis Statement: Data Centers need a Virtual Distributed File System (VDFS). VDFS to data is analogous to Internet Protocol (IP) to Internet.

In the rest of this chapter, we provide the detailed motivation of VDFS, VDFS overview, results highlights, Alluxio's ecosystem and its impact, and the contributions of this research.

1.1 Challenges with the Existing Ecosystem

Given the fast growing data size, researchers and practitioners are developing new computation frameworks and storage systems to address various needs. In the computation framework space, there are general computing systems, such as MapReduce and Spark, and specialized computing systems, such as Presto and Tensorflow. In the storage system ecosystem, different solutions have various trade-offs between total cost of ownership (TCO), performance, manageability, and supported workloads.

With so many options, these systems form a complex ecosystem (see Figure 1.1). To make them work together in different environments, developers need to integrate each computation framework with disparate storage systems. In the meantime, architects try to have each storage system provide various APIs to make it accessible for diverse workloads. Although this seems like a natural approach to integrate the ecosystem, it has several major drawbacks:

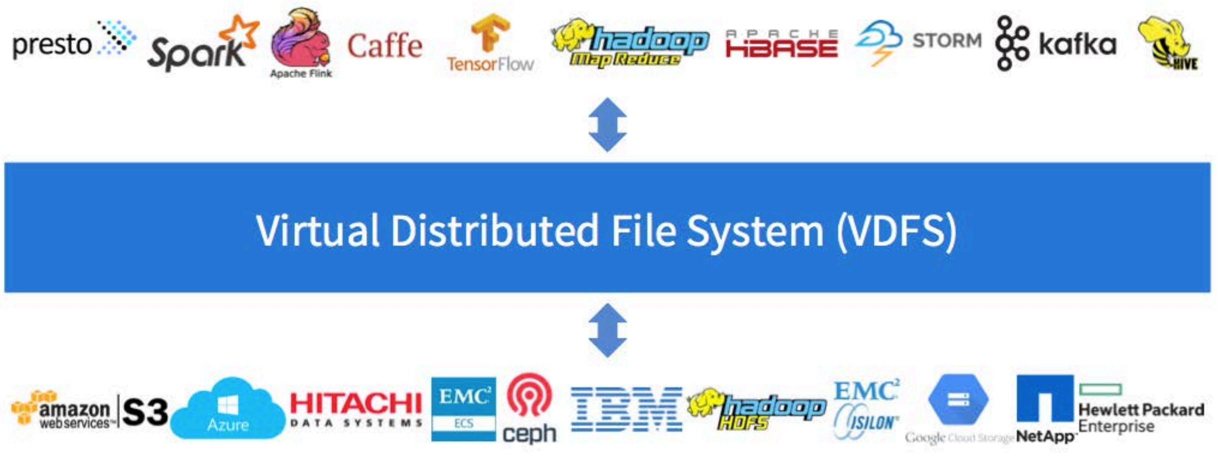


Figure 1.2: Virtual Distributed File System (VDFS)

- Work Duplication: Given the ecosystem stack illustrated in Figure 1.1, developers need to solve similar problems to connect each computation framework to different storage systems. In the meantime, each storage system needs to implement similar APIs to plug into different workloads.
- Data Duplication: For applications to efficiently interact with data from diverse sources, users often perform ETL (Extract, Transform and Load) to move data from many storage systems into a single data repository. This approach often results in duplicated copies of data [148].
- Performance: Most of the storage systems are disk or SSD based, and connected to the computation cluster through a network. Data transfer from the storage to the applications is often the data processing performance bottleneck [8, 34, 48].
- Management and Administration: Given data is permanently stored in various storage systems, and processed by different computation frameworks, both computation application developers and storage administrators have much overhead to learn about all the other systems in the ecosystem. In the meantime, it is difficult to manage data in multiple, diverse, remote data stores [85, 135].

Given the above drawbacks, a data unification layer brings significant value into the ecosystem. A data unification layer can improve data accessibility, performance, and data manageability, but also the convenience to plug future innovated systems into the ecosystem, therefore making it easier and quicker for the industry to adopt innovations.

1.2 Virtual Distributed File System (VDFS)

To address the challenges, we introduce a new abstraction, Virtual Distributed File System (VDFS). VDFS lays between the computation layer and the persistent storage layer. It has two sets of APIs, Southbound APIs (SAPIs) and Northbound APIs (NAPIs), see Figure 1.3. SAPIs enables VDFS to connect with various storage systems, while NAPIs expose different APIs for applications to be able to interact with data through VDFS without any code change and share data efficiently among each other. The insight behind VDFS is that although persistent storage systems expose various APIs, VDFS can abstract all those APIs to expose NAPIs to applications on top of VDFS.

People may have two main concerns around adding the VDFS as a layer in the ecosystem. First, will this further complicate the production environment and make it hard to maintain the software stack? Second, will having this layer in the middle impact the end-to-end performance of the software stack? In the rest of this section, we argue that VDFS does not create these two issues, and in fact, produces a positive impact on those two dimensions.

Typically, by adding another layer into a software stack, it often makes it more complex and harder to manage and operate. However, adding the VDFS simplifies the management and operation. First, VDFS decouples compute and storage systems. From the perspective of the computation layer, applications only need to interact with the VDFS instead of understanding and working with various storage systems. Second, VDFS creates a unified view of the data under a global namespace. From a storage management perspective, VDFS can also simplify data management. For example, without the VDFS, operators need to manually move data among various storage systems and that is likely to duplicate the data. These processes create more work, complexity, and cost to manage the data. The VDFS can solve the data duplication issue and management complexity by intelligently managing the data.

Regarding performance, adding another layer into the software stack means additional layers of code is executed. However, the VDFS manages all the storage media, including Memory, SSD, NVMe, and HDD, in the computation cluster, and can automatically keep the hot data in the most performant storage layer, e.g. DRAM layer, while keeping the cold data in a larger, but less performant layer, e.g. HDD layer. In this case, the hotter the data is, the closer it is to the applications. Therefore the applications would experience much better I/O performance than directly interacting with the persistent storage systems under the VDFS.

For the benefits and values the VDFS provides, we can make the analogy to the IP[2] for Internet or the Virtual File System [117, 82] for an operating system. The IP layer is the narrow waist in the Internet protocol suite[3]. It enables the higher layer to innovate without worrying about the lower IP layer, and vice-versa. In the meantime, the virtual file system is an abstraction layer on top of a concrete file system implementation, and it allows applications to be able to access different types of concrete file systems in a uniform way. In a similar way, the VDFS enables multiple applications to interact and manage data across various storage systems in a unified, performant and efficient

[2]https://en.wikipedia.org/wiki/Internet_Protocol
[3]https://en.wikipedia.org/wiki/Internet_protocol_suite

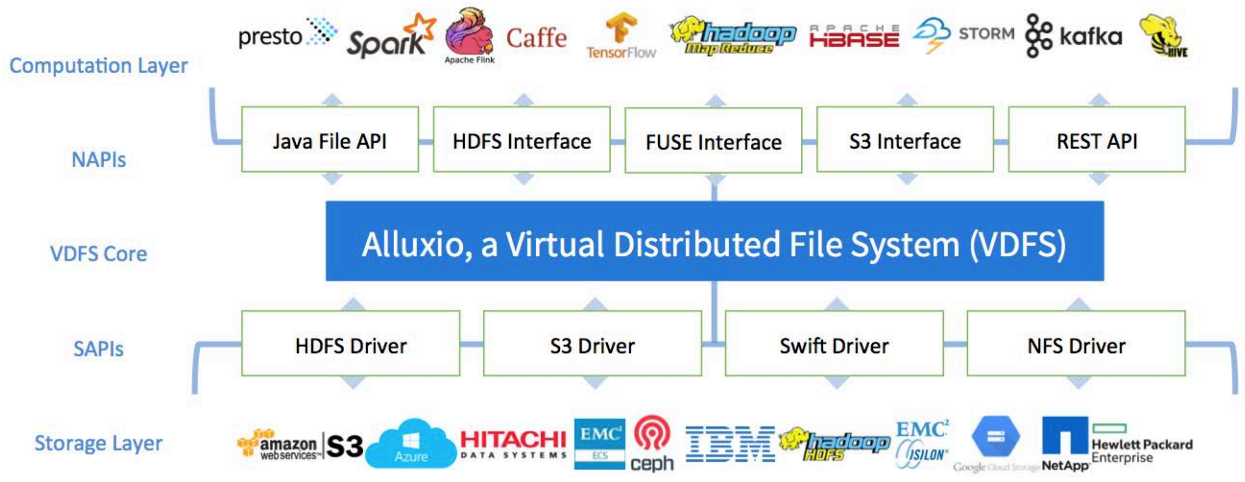


Figure 1.3: VDFS, Alluxio as an example

way.

1.3 Results and Impacts Highlights

We have implemented VDFS in an open source system called Alluxio [3] (formerly called Tachyon). Alluxio supports different computation frameworks and workloads, such as Apache Spark [24], Presto [110], Tensorflow [92], Apache MapReduce [16], Apache HBase [17], Apache Hive [20], Apache Kafka [21], and Apache Flink [14, 37], and various types of storage systems, such as Amazon S3 [6], Google Cloud Storage [66], Microsoft Azure Storage [97], OpenStack Swift [102], GlusterFS [65], HDFS [19], MaprFS, Ceph [137], NFS, Alibaba OSS [2], EMC ECS [54], and IBM Cleversafe [78].

The Alluxio open source project is one of the fastest growing open source projects. With five years of open source history, Alluxio has attracted more than 800 contributors from over 200 institutions, including Alibaba, Baidu, CMU, Google, IBM, Intel, JD, NJU, Red Hat, Tencent, UC Berkeley, and Yahoo. Alluxio is deployed in production by hundreds of organizations and runs on clusters that exceed 1,000 nodes. For example, as of today, seven out of the ten most valuable Internet companies in the world are running Alluxio.

From the data unification perspective, users use Alluxio to connect applications with different storage systems. From the performance perspective, Alluxio can bring up to two orders of magnitude performance improvement. From the data management perspective, Alluxio can provide features such as data migration, and discovery recovery. For example, Baidu uses Alluxio to speedup the throughput of their data analytics pipeline up to 30 times. Barclays use Alluxio to accelerate jobs to reduce the end-to-end execution time from hours to seconds. Qunar performs real-time data analytics on top of Alluxio while leveraging the data from different storage systems at the same time. More examples are listed on Alluxio's website' the powered-by page[4].

1.4 Alluxio Ecosystem

We implemented several NAPIs and SAPIs in Alluxio to integrate different systems and workloads into the Alluxio ecosystem. For applications running on top of Alluxio, they can interact with data from different storage systems through Alluxio without any code change. In some cases, the new stack experiences significant performance improvement over the stack without Alluxio. For storage systems integrated into this ecosystem, they are able to run all workloads Alluxio can support on top of them immediately, with performance boost in many cases.

Here are four NAPI examples. Some of the APIs have more than one language bindings, such as C++, Java, Go, and Python:

- **Hadoop Compatible Filesystem API**: We implemented a Hadoop Compatible Filesystem API to enable existing applications in the big data ecosystem, e.g. Spark, Presto, MapReduce, Hive, etc., to run on top of Alluxio without any code changes.
- **S3 / Swift Compatible API**: By offering an S3 / Swift API, it enables any cloud based applications to interact with data through Alluxio.
- **Filesystem in Userspace (FUSE) API**: By offering a FUSE API, any legacy applications interacting with the local file system can work with Alluxio and interact with data from any storage system, which is integrated into the Alluxio ecosystem.
- **Filesystem REST API**: In addition to the APIs above, by offering a Filesystem REST API, applications written in any language can interact with Alluxio through web-based REST API. The REST API enables many more applications to be able to share data with each other.

Here are two SAPI examples:

- **Filesystem APIs**: We implemented various Filesystem APIs as the SAPIs of Alluxio, in which case, it enables many different filesystems to be plugged into Alluxio ecosystem. For example, these include any Hadoop compatible filesystems, like GlusterFS.
- **Object Storage APIs**: By offering object storage APIs integrations, it expands the ecosystem to many object storage systems, such as Amazon S3, Google Cloud Storage, and Microsoft Azure Cloud Storage.

[4] `https://www.alluxio.org/community/powered-by-alluxio`

Besides various NAPIs and SAPIs, Alluxio can be deployed in a standalone mode, in Kubernetes [35], Mesos [75], DCOS [49], or Yarn [133] mode. In the meantime, Alluxio can be deployed in on-premise, public cloud, or hybrid cloud environments.

1.5 Contributions

In this thesis, I present the VDFS and its open source implementation, Alluxio. The main contributions of this dissertation are the following:

- Propose VDFS, a new abstraction between computation layer and persistent storage layer, which changes the landscape of the ecosystem, enables faster innovation by reducing the complexity of the evolving ecosystem, making data management easier, improving application performance, and increasing business agility.
- Implement VDFS in an open source software called Alluxio. The system is deployed at hundreds of leading companies, serving critical workloads. The Alluxio open source community has attracted more than 800 contributors from over 200 companies.
- Propose Lineage as an important technique in the VDFS to improve write performance.
- Propose DFS-Perf, a scalable distributed file system performance evaluation framework to help researchers and developers better design and implement systems in Alluxio ecosystem.
- Present the lessons learned from the research as well as open source and industry experience and how they can help promote each other.

1.6 Dissertation Overview

This dissertation is organized as follows. Chapter 2 introduces the Lineage concept in VDFS to enable fast durable writes. Chapter 3 covers the Distributed File System Performance Evaluation Framework (DFS-Perf) to help practitioners and researchers better design and implement various storage solutions and their integrations with Alluxio in the VDFS ecosystem. Chapter 4 presents the design and implementation of open source Alluxio, as well as its open source status and impact. Finally, we conclude and discuss potential future work in Chapter 5.

Chapter 2

Lineage in VDFS

2.1 Introduction

Over the past decade, there have been tremendous efforts to improve the speed and sophistication of large-scale data-parallel processing systems. Practitioners and researchers have built a wide array of computation frameworks [89, 91, 93, 109, 146, 145, 118, 124, 129, 101, 30, 110, 38, 56, 139, 79] and storage systems [99, 28, 58, 60, 103, 54, 137] tailored to a variety of workloads.

As the performance of many of these systems is I/O bound, traditional means of improving their speed is to cache data into memory [9]. While caching can dramatically improve read performance, unfortunately, it does not help much with write performance. This is because highly parallel systems need to provide fault-tolerance, and the way they achieve it is by replicating the data written across nodes. Even replicating the data in memory can lead to a significant drop in the write performance, as both the latency and throughput of the network are typically much worse than that of local memory.

Slow writes can significantly hurt the performance of job pipelines, where one job consumes the output of another. These pipelines are regularly produced by perform data extraction with MapReduce, then execute a SQL query, then run a machine learning algorithm on the query's result. Furthermore, many high-level programming interfaces [12, 23, 129], such as Pig [101] and FlumeJava [38], compile programs into multiple MapReduce jobs that run sequentially. In all these cases, data is replicated across the network in-between each of the steps.

To improve write performance, we present *Alluxio* system, a VDFS implementation, with lineage to achieve *high throughput* writes and reads, without compromising fault-tolerance. Alluxio circumvents the throughput limitations of replication by leveraging the concept of *lineage*, where lost output is recovered by re-executing the operations (tasks) that created the output. As a result, lineage provides fault-tolerance without the need for replicating the data.

While the concept of lineage has been used before in the context of frameworks such as Spark, Nectar and BAD-FS [69, 146, 29], there are several important challenges that arise when pushing it into a continuously running distributed storage system. Previous systems do not address these challenges.

The first challenge is *bounding the recomputation cost* for a long-running storage system. This challenge does not exist for a single computing job, such as a MapReduce or Spark job, as in this case, the recomputation time is trivially bounded by the job's execution time. In contrast, Alluxio runs indefinitely, which means that the recomputation time can be unbounded. Previous frameworks that support long running workloads, such as Spark Streaming [145] and BAD-FS [29], circumvent this challenge by using periodic checkpointing. Unfortunately, using the same techniques in Alluxio is difficult, as the storage layer is agnostic to the semantics of the jobs running on the data (*e.g.,* when outputs will be reused), and job execution characteristics can vary widely. In particular, any fixed checkpoint interval can lead to unbounded recovery times if data is written faster than the available disk bandwidth. Instead, we prefer to select which data to checkpoint based on the structure of the lineage graph in a manner that still bounds recovery time.

The second challenge is *how to allocate resources for recomputations*. For example, if jobs have priorities, Alluxio must, on the one hand, make sure that recomputation tasks get adequate resources (even if the cluster is fully utilized), and on the other hand, Alluxio must ensure that recomputation tasks do not severely impact the performance of currently running jobs with possibly higher priorities.

Alluxio bounds the data recomputation cost, thus addressing the first challenge, by continuously checkpointing files *asynchronously* in the background. To select which files to checkpoint and when, we propose a novel algorithm, called the Edge algorithm, that provides an upper bound on the recomputation cost regardless of the workload's access pattern.

To address the second challenge, Alluxio provides resource allocation schemes that respect job priorities under two common cluster allocation models: strict priority and weighted fair sharing [80, 144]. For example, in a cluster using a strict priority scheduler, if a missing input is requested by a low priority job, the recomputation minimizes its impact on high priority jobs. However, if the same input is later requested by a higher priority job, Alluxio automatically increases the amount of resources allocated for recomputation to avoid priority inversion [88].

We have implemented Alluxio with a general lineage-specification API that can capture computations in many of today's popular data-parallel computing models, *e.g.,* MapReduce and SQL. We also ported the Hadoop and Spark frameworks to run on top of it[1]. The project is open source, has more than 40 contributors from over 15 institutions, and is deployed at multiple companies[2].

Our evaluation shows that on average, Alluxio[3] achieves 110x higher write throughput than in-memory HDFS [15]. In a realistic industry workflow, Alluxio improves end-to-end latency by 4x compared to in-memory HDFS. In addition, because many files in computing clusters are tem-

[1] Applications can choose to use the lineage feature or not by a configuration parameter.

[2] Based on the Alluxio Open Source project status in 2013

[3] This paper focus on in-memory Alluxio deployment. However, Alluxio can also speed up SSD- and disk-based systems if the aggregate local I/O bandwidth is higher than the network bandwidth.

porary files that get deleted before they are checkpointed, Alluxio can reduce replication-caused network traffic by up to 50%. Finally, based on traces from Facebook and Bing, Alluxio would consume no more than 1.6% of cluster resources for recomputation.

More importantly, due to the inherent bandwidth limitations of replication, a lineage-based recovery model might be the *only* way to make cluster storage systems match the speed of in-memory computations in the future. This work aims to address some of the leading challenges in making such a system possible.

2.2 Background

This section describes our target workloads and provides background on existing solutions. Section 2.8 describes related work in greater detail.

2.2.1 Target Workload Properties

We have designed Alluxio targeting today's big data workloads, which have the following properties:

- Immutable data: Data is immutable once written, since dominant underlying storage systems, such as HDFS [15], only support the append operation.
- Deterministic jobs: Many frameworks, such as MapReduce [50] and Spark [146], use recomputation for fault tolerance within a job and require user code to be deterministic. We provide lineage-based recovery under the same assumption. Nondeterministic frameworks can still store data in Alluxio using replication.
- Locality based scheduling: Many computing frameworks [50, 146] schedule jobs based on locality to minimize network transfers, so reads can be data-local.
- All data vs. working set: Even though the whole data set is large and has to be stored on disks, the working set of many applications fits in memory [9, 146].
- Program size vs. data size: In big data processing, the same operation is repeatedly applied on massive data. Therefore, replicating programs is much less expensive than replicating data in many cases.

2.2.2 The Case Against Replication

In-memory computation frameworks – such as Spark and Piccolo [109], as well as caching in storage systems – have greatly sped up the performance of individual jobs. However, sharing (writing) data reliably among different jobs often becomes a bottleneck.

The write throughput is limited by disk (or SSD) and network bandwidths in existing data storage solutions, such as HDFS [15], FDS [99], Cassandra [11], HBase [17], and RAMCloud [103]. All these systems use media with much lower bandwidth than memory (Table 2.1).

Media	**Capacity**	**Bandwith**
HDD (x12)	12-36 TB	0.2-2 GB/sec
SDD (x4)	1-4 TB	1-4 GB/sec
Network	N/A	1.25 GB/sec
Memory	128-512 GB	10-100 GB/sec

Table 2.1: Typical datacenter node setting in 2013 [53].

The fundamental issue is that in order to be fault-tolerant, these systems *replicate* data across the network and write at least one copy onto non-volatile media to allow writes to survive datacenter-wide failures, such as power outages. Because of these limitations and the advancement of in-memory computation frameworks [89, 91, 109, 146], inter-job data sharing cost often dominates pipeline's end-to-end latencies for big data workloads. While some jobs' outputs are much smaller than their inputs, a recent trace from Cloudera showed that, on average, 34% of jobs (weighted by compute time) across five customers had outputs that were at least as large as their inputs [41]. In an in-memory computing cluster, these jobs would be write throughput bound.

Hardware advancement is unlikely to solve the issue. Memory bandwidth is one to three orders of magnitude higher than the aggregate disk bandwidth on a node. The bandwidth gap between memory and disk is becoming larger. The emergence of SSDs has little impact on this since its major advantage over disk is random access latency, but not sequential I/O bandwidth, which is what most data- intensive workloads need. Furthermore, throughput increases in network indicate that over-the- network memory replication might be feasible. However, sustaining datacenter power outages requires at least one disk copy for the system to be fault-tolerant. Hence, in order to provide high throughput, storage systems have to achieve fault-tolerance without replication.

2.3 Design Overview

This section overviews the design of Alluxio, while the following two sections (§2.4 & §2.5) focus on the two *main* challenges that a storage system incorporating lineage faces: bounding recovery cost and allocating resources for recomputation.

2.3.1 System Architecture

Alluxio consists of two layers: lineage and persistence. The lineage layer provides high throughput I/O and tracks the sequence of jobs that have created a particular data output. The persistence layer persists data onto storage without the lineage concept. This is mainly used to do asynchronous checkpoints. The persistence layer can be any existing replication based storage systems, such as HDFS, S3, Glusterfs.

Alluxio employs a standard master-slave architecture similar to HDFS and GFS (see Figure 2.1). In the remainder of this section we discuss the unique aspects of Alluxio.

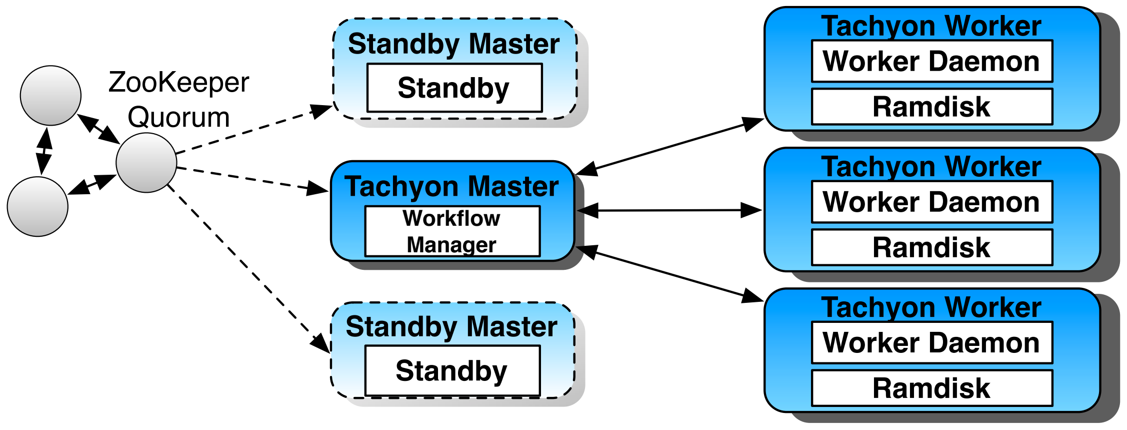


Figure 2.1: Alluxio, formerly Tachyon, Architecture

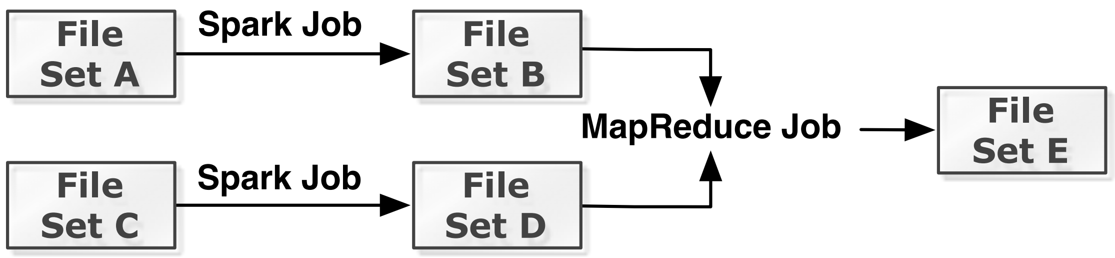


Figure 2.2: A lineage graph example of multiple frameworks

In addition to managing metadata, the master also contains a *workflow manager*. The role of this manager is to track lineage information, compute checkpoint order (§2.4), and interact with a cluster resource manager to allocate resources for recomputation (§2.5).

Each worker runs a daemon that manages local resources, and periodically reports the status to the master. In addition, each worker uses a RAMdisk for storing memory-mapped files. A user application can bypass the daemon and interact directly with RAMdisk. This way, an application with data locality §2.2.1 can interact with data at memory speeds, while avoiding any extra data copying.

2.3.2 An Example

To illustrate how Alluxio works, consider the following example. Assume job *P* reads file set A and writes file set B. Before *P* produces the output, it submits its lineage information L to Alluxio. This information describes how to run *P* (e.g., command line arguments, configuration parameters) to generate B from A. Alluxio records L reliably using the persistence layer. L guarantees that if

Return	**Signature**
Global Unique Lineage Id	createDependency(inputFiles, outputFiles, binaryPrograms, executionConfiguration, dependencyType)
Dependency Info	getDependency(lineageId)

Table 2.2: Submit and Retrieve Lineage API

B is lost, Alluxio can recompute it by (partially[4] re-executing *P*. As a result, leveraging the lineage, *P* can write a single copy of B to memory without compromising fault-tolerance. Figure 2.2 shows a more complex lineage example.

Recomputation based recovery assumes that input files are immutable (or versioned, *c.f.,* §2.9) and that the executions of the jobs are deterministic. While these assumptions are not true of all applications, they apply to a large fraction of datacenter workloads (*c.f.,* §2.2.1), which are deterministic applications (often in a high-level language such as SQL where lineage is simple to capture).

Alluxio improves write performance by avoiding replication. However, replication has the advantage of improving read performance as more jobs read the same data. While Alluxio leverages lineage to avoid data replication, it uses a client-side caching technique to mitigate read hotspots transparently. That is, if a file is not available on the local machine, it is read from a remote machine and cached locally in Alluxio, temporarily increasing its replication factor.

2.3.3 API Summary

Alluxio is an append-only file system, similar to HDFS, that supports standard file operations, such as create, open, read, write, close, and delete. In addition, Alluxio provides an API to capture the lineage across different jobs and frameworks. Table 2.2 lists the lineage API, and Section 2.6.1 describes this API in detail.

The lineage API adds complexity to Alluxio over replication based file systems such as HDFS, S3. However, only framework programmers need to understand Alluxio's lineage API. Alluxio does not place extra burden on application programmers. As long as a framework, e.g. Spark, integrates with Alluxio, applications on top of the framework take advantage of lineage based fault-tolerance transparently[5]. Furthermore, a user can choose to use Alluxio as a traditional file system if he/she does not use the lineage API. In this case, the application would not have the benefit of memory throughput writes, but will still perform no worse than on a traditional replicated file system.

[4]The recomputation granularity depends on the framework integration. For example, it can be job level or task level.

[5]We made this configurable in our Spark and MapReduce integration, which means applications on top of them can choose not to use this feature.

2.3.4 Lineage Overhead

In terms of storage, job binaries represent by far the largest component of the lineage information. However, according to Microsoft data [69], a typical data center runs $1,000$ jobs daily on average, and it takes up to 1 TB to store the uncompressed binaries of all jobs executed over a one year interval. This overhead is negligible even for a small sized data center.

Furthermore, Alluxio can garbage collect the lineage information. In particular, Alluxio can delete a lineage record after checkpointing (*c.f.,* §2.4) its output files. This will dramatically reduce the overall size of the lineage information. In addition, in production environments, the same binary program is often executed many times, *e.g.,* periodic jobs, with different parameters. In this case, only one copy of the program needs to be stored.

2.3.5 Data Eviction

Alluxio works best when the workload's working set fits in memory. In this context, one natural question is what is the eviction policy when the memory fills up. Our answer to this question is influenced by the following characteristics identified by previous works [41, 116] for data intensive applications:

- **A**ccess Frequency: File access often follows a Zipf-like distribution (see [41, Figure 2]).
- **A**ccess Temporal Locality: 75% of the re-accesses take place within 6 hours (see [41, Figure 5]).

Based on these characteristics, we use LRU as a default policy. However, since LRU may not work well in all scenarios, Alluxio also allows plugging in other eviction policies. Finally, as we describe in Section 2.4, Alluxio stores all but the largest files in memory. The rest are stored directly to the persistence layer.

2.3.6 Master Fault-Tolerance

As shown in Figure 2.1, Alluxio uses a "passive standby" approach to ensure master fault-tolerance. The master logs every operation synchronously to the persistence layer. When the master fails, a new master is selected from the standby nodes. The new master recovers the state by simply reading the log. Note that since the metadata size is orders of magnitude smaller than the output data size, the overhead of storing and replicating it is negligible.

2.3.7 Handling Environment Changes

One category of problems Alluxio must deal with is changes in the cluster's runtime environment. How can we rely on re-executing binaries to recompute files if, for example, the version of the framework that an application depends on changes, or the OS version changes?

One observation we make here is that although files' dependencies may go back in time forever, checkpointing allows us to place a bound on how far back we ever have to go to recompute data. Thus, before an environment change, we can ensure recomputability by switching the system into a "synchronous mode", where (a) all currently unreplicated files are checkpointed and (b) all new data is saved synchronously. Once the current data is all replicated, the update can proceed and this mode can be disabled [6].

For more efficient handling of this case, it might also be interesting to capture a computation's environment using a VM image [70]. We have, however, not yet explored this option.

2.3.8 Why VDFS Layer

To achieve memory throughput I/O, we need to avoid replication and thus use lineage information. We believe that it is necessary to push lineage into the VDFS layer because of the followings:

First, a data processing pipeline usually contains more than one job. It is necessary to capture the lineage across jobs. Furthermore, in a production environment, the data producer and its consumers could be written in different frameworks. Therefore, only understanding the lineage at the job level or a single framework level can not solve the issue.

Second, only the VDFS layer (together with storage system) knows when files are renamed or deleted, which is necessary to track lineage correctly and checkpoint data correctly in long-term operations. Since other layers do not have full control on the storage layer, it is possible that a user may manually delete a file. All files depending on the deleted file will lose fault tolerance guarantee.

2.4 Checkpointing

This section outlines the checkpoint algorithm used by Alluxio to bound the amount of time it takes to retrieve a file that is lost due to failures[7]. By a file we refer to a distributed file, *e.g.,* all output of a MapReduce/Spark job. Unlike other frameworks, such as MapReduce and Spark, whose jobs are relatively short-lived, Alluxio runs continuously. Thus, the lineage that accumulates can be substantial, requiring long recomputation time in the absence of checkpoints. Therefore, checkpointing is crucial for the performance of Alluxio. Note that long-lived streaming systems, such as Spark Streaming [145], leverage their knowledge of job semantics to decide what and when to checkpoint. Alluxio has to checkpoint in absence of such detailed semantic knowledge.

The key insight behind our checkpointing approach in Alluxio is that lineage enables us to *asynchronously checkpoint in the background*, without stalling writes, which can proceed at memory-speed. This is unlike other storage systems that do not have lineage information, *e.g.,* key-value

[6]Data deletion can be handled as a similar approach. For example, when a user tries to delete a file A, if there are other files depending on A, Alluxio will delete A asynchronously after the depending data has been checkpointed.

[7]In this section, we assume recomputation has the same resource as the first time computation. In Section 2.5, we address the recomputation resource allocation issue.

stores, which synchronously checkpoint, returning to the application that invoked the write only once data has been persisted to stable storage. Alluxio' background checkpointing is done in a low priority process to avoid interference with existing jobs.

An ideal checkpointing algorithm would provide the following:

1. *Bounded Recomputation Time.* Lineage chains can grow very long in a long-running system like Alluxio, therefore the checkpointing algorithm should provide a bound on how long it takes to recompute data in the case of failures. Note that bounding the recomputation time also bounds the computational resources used for recomputations.
2. *Checkpointing Hot files.* Some files are much more popular than others. For example, the same file, which represents a small dimension table in a data-warehouse, is repeatedly read by all mappers to do a map-side join with a fact table [9].
3. *Avoid Checkpointing Temporary Files.* Big data workloads generate a lot of temporary data. From our contacts at Facebook, nowadays, more than 70% data is deleted within a day, without even counting shuffle data. Figure 2.3a illustrates how long temporary data exists in a cluster at Facebook[8]. An ideal algorithm would avoid checkpointing much of this data.

We consider the following straw man to motivate our algorithm: asynchronously checkpoint every file in the order that it is created. Consider a lineage chain, where file A_1 is used to generate A_2, which is used to generate A_3, A_4, and so on. By the time A_6 is being generated, perhaps only A_1 and A_2 have been checkpointed to stable storage. If a failure occurs, then A_3 through A_6 have to be recomputed. The longer the chain, the longer the recomputation time. Thus, spreading out checkpoints throughout the chain would make recomputations faster.

2.4.1 Edge Algorithm

Based on the above characteristics, we have designed a simple algorithm, called Edge, which builds on three ideas. First, Edge checkpoints the edge (leaves) of the lineage graph (hence the name). Second, it incorporates priorities, favoring checkpointing high-priority files over low-priority ones. Finally, the algorithm only caches datasets that can fit in memory to avoid synchronous checkpointing, which would slow down writes to disk speed. We discuss each of these ideas in detail:

Checkpointing Leaves. The Edge algorithm models the relationship of files with a DAG, where the vertices are files, and there is an edge from a file A to a file B if B was generated by a job that read A. The algorithm checkpoints the latest data by checkpointing the leaves of the DAG. This lets us satisfy the requirement of bounded recovery time (explained in Section 2.4.2).

Figure 2.4 illustrates how the Edge algorithm works. At the beginning, there are only two jobs running in the cluster, generating files A_1 and B_1. The algorithm checkpoints both of them. After

[8]The workload was collected from a 3,000 machine MapReduce cluster at Facebook, during a week in October 2010.

Access Count	1	3	5	10
Percentage	62%	86%	93%	95%

Table 2.3: File Access Frequency at Yahoo

they have been checkpointed, files A_3, B_4, B_5, and B_6 become leaves. After checkpointing these, files A_6, B_9 become leaves.

To see the advantage of Edge checkpointing, consider the pipeline only containing A_1 to A_6 in the above example. If a failure occurs when A_6 is being checkpointed, Alluxio only needs to recompute from A_4 through A_6 to get the final result. As previously mentioned, checkpointing the earliest files, instead of the edge, would require a longer recomputation chain.

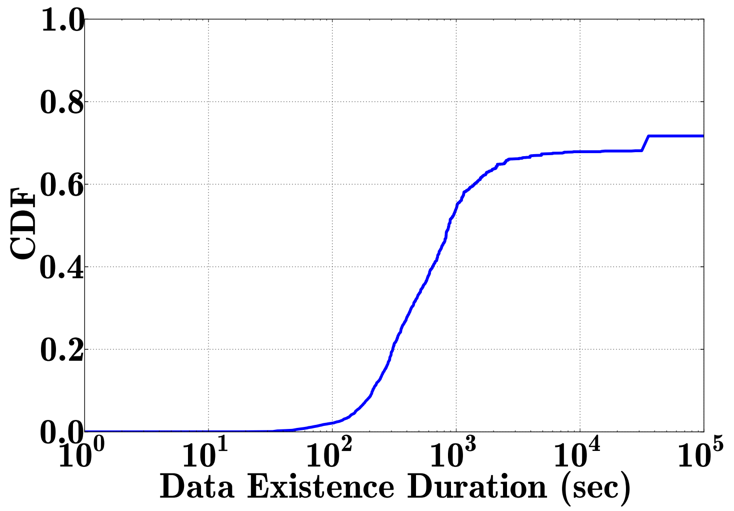
This type of pipeline is common in industry. For example, continuously monitoring applications generate hourly reports based on minutely reports, daily reports based on hourly reports, and so on.

Checkpointing Hot Files. The above idea of checkpointing the latest data is augmented to first checkpoint high priority files. Alluxio assigns priorities based on the number of times a file has been read. Similar to the LFU policy for eviction in caches, this ensures that frequently accessed files are checkpointed first. This covers the case when the DAG has a vertex that is repeatedly read leading to new vertices being created, *i.e.,* a high degree vertex. These vertices will be assigned a proportionally high priority and will thus be checkpointed, making recovery fast.

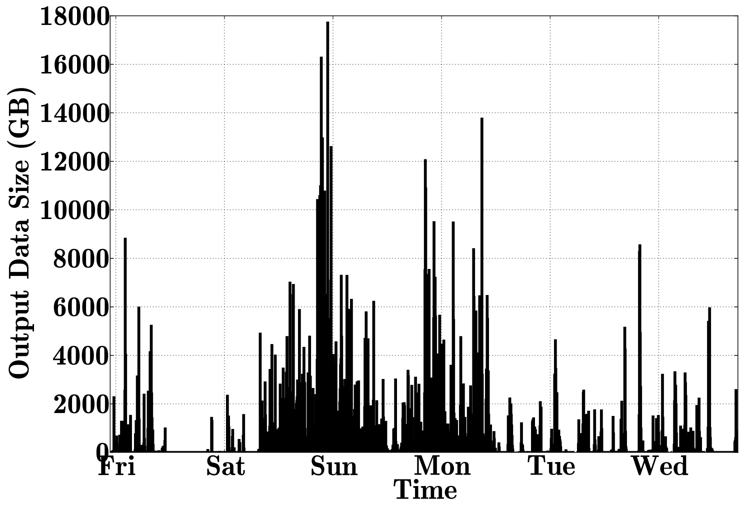
Edge checkpointing has to balance between checkpointing leaves, which guarantee recomputation bounds, and checkpointing hot files, which are important for certain iterative workloads. Here, we leverage the fact that most big data workloads have a Zipf-distributed popularity (this has been observed by many others [9, 41]). Table 2.3 shows what percentage of the files are accessed less than (or equal) than some number of times in a 3,000-node MapReduce cluster at Yahoo in January 2014. Based on this, we consider a file high-priority if it has an access count higher than 2. For this workload, 86% of the checkpointed files are leaves, whereas the rest are non-leaf files. Hence, in most cases bounds can be provided. The number can naturally be reconfigured for other workloads. Thus, files that are accessed more than twice get precedence in checkpointing compared to leaves.

A replication-based filesystem has to replicate every file, even temporary data used between jobs. This is because failures could render such data as unavailable. Alluxio avoids checkpointing much of the temporary files created by frameworks. This is because checkpointing later data first (leaves) or hot files, allows frameworks or users to delete temporary data before it gets checkpointed[9].

[9]In our implementation, Alluxio also allows frameworks to indicate temporary data explicitly by path name.



(a) Estimated temporary data span including shuffle data



(b) Data generation rates at five minutes granularity

Figure 2.3: A 3,000 node MapReduce cluster at Facebook

Dealing with Large Data Sets. As observed previously, working sets are Zipf-distributed [41, Figure 2]. We can therefore store in memory all but the very largest datasets, which we avoid storing in memory altogether. For example, the distribution of input sizes of MapReduce jobs at Facebook is heavy- tailed [8, Figure 3a]. Furthermore, 96% of active jobs can have their entire data simultaneously fit in the corresponding clusters' memory [8]. The Alluxio master is thus configured to synchronously write datasets above the defined threshold to disk. In addition, Figure 2.3b shows that file requests in the aforementioned Facebook cluster is highly bursty. During bursts, Edge checkpointing might checkpoint leafs that are far apart in the DAG. As soon as the bursts

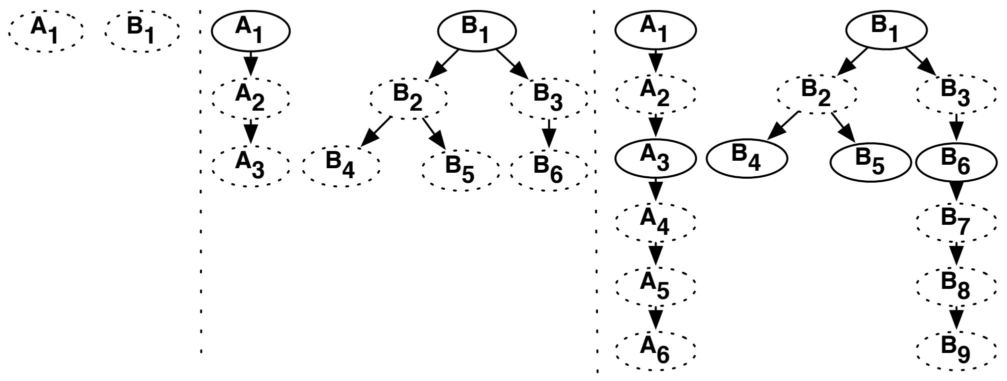


Figure 2.4: Edge Checkpoint Example. Each node represents a file. Solid nodes denote checkpointed files, while dotted nodes denote uncheckpointed files.

finish, Edge checkpointing starts checkpointing the rest of the non-leaf files. Thus, most of the time most of the files in memory have been checkpointed and can be evicted from memory if room is needed (see Section 2.3). If the memory fills with files that have not been checkpointed, Alluxio checkpoints them synchronously to avoid having to recompute long lineage chains. In summary, all but the largest working sets are stored in memory and most data has time to be checkpointed due to the bursty behavior of frameworks. Thus, evictions of uncheckpointed files are rare.

2.4.2 Bounded Recovery Time

Checkpointing the edge of the DAG lets us derive a bound on the recomputation time. The key takeaway of the bound is that recovery of any file takes on the order of time that it takes to read or generate an edge. Informally, it is independent of the depth of the lineage DAG.

Recall that the algorithm repeatedly checkpoints the edge of the graph. We refer to the time it takes to checkpoint a particular edge i of the DAG as W_i. Similarly, we refer to the time it takes to generate an edge i from its ancestors as G_i. We now have the following bound.

Jibberish 1. *Edge checkpointing ensures that any file can be recovered in* $3 \times M$, *for* $M = \max_i\{T_i\}$, $T_i = \max(W_i, G_i)$[10].

This shows that recomputations are independent of the "depth" of the DAG. This assumes that the caching behavior is the same during the recomputation, which is true when working sets fit in memory (*c.f.,* Section 2.4.1).

The above bound does not apply to priority checkpointing. However, we can easily incorporate priorities by alternating between checkpointing the edge c fraction of the time and checkpointing high-priority data $1 - c$ of the time.

[10]We refer the reader to [83] for the proof.

Corollary 2. *Edge checkpointing, where c fraction of the time is spent checkpointing the edge, ensures that any file can be recovered in $\frac{3\times M}{c}$, for $M = \max_i\{T_i\}$, $T_i = \max(W_i, G_i)$.*

Thus, configuring $c = 0.5$ checkpoints the edge half of the time, doubling the bound of Theorem 1. These bounds can be used to provide SLOs to applications.

In practice, priorities can improve the recomputation cost. In the evaluation(§2.7), we illustrate actual recomputation times when using edge caching.

2.5 Resource Allocation

Although the Edge algorithm provides a bound on recomputation cost, Alluxio needs a resource allocation strategy to schedule jobs to recompute data in a timely manner. In addition, Alluxio must respect existing resource allocation policies in the cluster, such as fair sharing or priority.

In many cases, there will be free resources for recomputation, because most datacenters are only 30 –50% utilized. However, care must be taken when a cluster is full. Consider a cluster fully occupied by three jobs, J_1, J_2, and J_3, with increasing importance (*e.g.,* from research, testing, and production). There are two lost files, F_1 and F_2, requiring recomputation jobs R_1 and R_2. J_2 requests F_2 only. How should Alluxio schedule recomputations?

One possible solution is to statically assign part of the cluster to Alluxio, *e.g.,* allocate 25% of the resources on the cluster for recomputation. However, this approach limits the cluster's utilization when there are no recomputation jobs. In addition, the problem is complicated because many factors can impact the design. For example, in the above case, how should Alluxio adjust R_2's priority if F_2 is later requested by the higher priority job J_3?

To guide our design, we identify three goals:

1. *Priority compatibility:* If jobs have priorities, recomputation jobs should follow them. For example, if a file is requested by a low priority job, the recomputation should have minimal impact on higher priority jobs. But if the file is later requested by a high priority job, the recovery job's importance should increase.
2. *Resource sharing:* If there are no recomputation jobs, the whole cluster should be used for normal work.
3. *Avoid cascading recomputation:* When a failure occurs, more than one file may be lost at the same time. Recomputing them without considering data dependencies may cause recursive job launching.

We start by presenting resource allocation strategies that meet the first two goals for common cluster scheduling policies. Then, we discuss how to achieve the last goal, which is orthogonal to the scheduling policy.

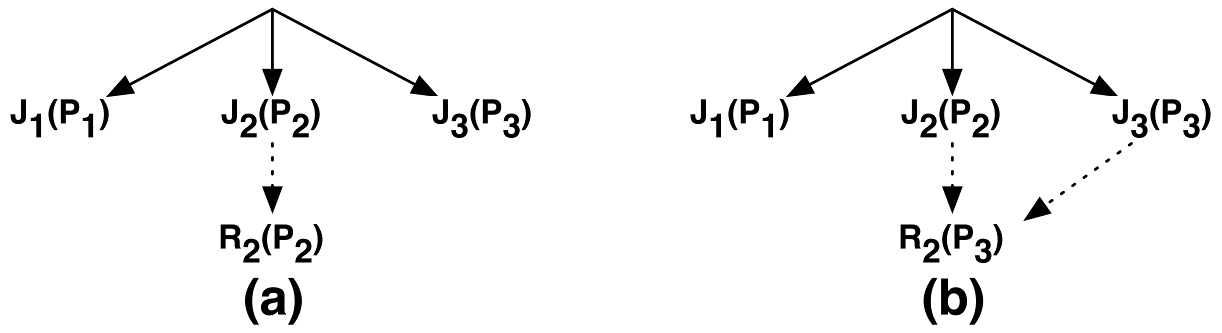


Figure 2.5: Resource Allocation Strategy for Priority Based Scheduler.

2.5.1 Resource Allocation Strategy

The resource allocation strategy depends on the scheduling policy of the cluster Alluxio runs on. We present solutions for priority and weighted fair sharing, the most common policies in systems like Hadoop and Dryad [144, 80].

Priority Based Scheduler In a priority scheduler, using the same example above, jobs J_1, J_2, and J_3 have priorities P_1, P_2, and P_3 respectively, where $P_1 < P_2 < P_3$.

Our solution gives all recomputation jobs the lowest priority by default, so they have minimal impact on other jobs. However, this may cause priority inversion. For example, because file F_2's recomputation job R_2 has a lower priority than J_2, it is possible that J_2 is occupying the whole cluster when it requests F_2. In this case, R_2 cannot get resources, and J_2 blocks on it.

We solve this by priority inheritance. When J_2 requests F_2, Alluxio increases R_2's priority to be P_2. If F_2 is later read by J_3, Alluxio further increases its priority. Figure 2.5a and 2.5b show jobs' priorities before and after J_3 requests F_2.

Fair Sharing Based Scheduler In a hierarchical fair sharing scheduler, jobs J_1, J_2, and J_3 have shares W_1, W_2, and W_3 respectively. The minimal share unit is 1.

In our solution, Alluxio has a default weight, W_R (as the minimal share unit 1), shared by all recomputation jobs. When a failure occurs, all lost files are recomputed by jobs with a equal share under W_R. In our example, both R_1 and R_2 are launched immediately with share 1 in W_R.

When a job requires lost data, part of the requesting job's share[11], is moved to the recomputation job. In our example, when J_2 requests F_2, J_2 has share $(1-a)$ under W_2, and R_2 share a under W_2. When J_3 requests F_2 later, J_3 has share $1-a$ under W_3 and R_2 has share a under W_3. When R_2 finishes, J_2 and J_3 resumes all of their previous shares, W_2 and W_3 respectively. Figure 2.6 illustrates.

[11] a could be a fixed portion of the job's share, *e.g.,* 20%

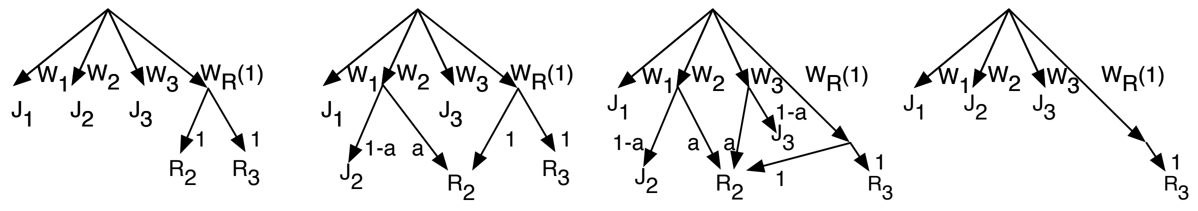


Figure 2.6: Resource Allocation Strategy for Fair Sharing Based Scheduler.

This solution fulfills our goals, in particular, priority compatibility and resource sharing. When no jobs are requesting a lost file, the maximum share for all recomputation jobs is bounded. In our example, it is $W_R/(W_1 + W_2 + W_3 + W_R)$. When a job requests a missing file, the share of the corresponding recomputation job is increased. Since the increased share comes from the requesting job, there is no performance impact on other normal jobs.

2.5.2 Recomputation Order

Recomputing a file might require recomputing other files first, such as when a node fails and loses multiple files at the same time. While the programs could recursively make callbacks to the workflow manager to recompute missing files, this would have poor performance. For instance, if the jobs are non-preemptable, computation slots are occupied, waiting for other recursively invoked files to be reconstructed. If the jobs are preemptable, computation before loading lost data is wasted. For these reasons, the workflow manager determines in advance the order of the files that need to be recomputed and schedules them.

To determine the files that need to be recomputed, the workflow manager uses a logical directed acyclic graph (DAG) for each file that needs to be reconstructed. Each node in the DAG represents a file. The parents of a child node in the DAG denote the files that the child depends on. That is, for a wide dependency a node has an edge to all files it was derived from, whereas for a narrow dependency it has a single edge to the file that it was derived from. This DAG is a subgraph of the DAG in Section 2.4.1.

To build the graph, the workflow manager does a depth-first search (DFS) of nodes representing targeted files. The DFS stops whenever it encounters a node that is already available in storage. The nodes visited by the DFS must be recomputed. The nodes that have no lost parents in the DAG can be recomputed first in parallel. The rest of nodes can be recomputed when all of their children become available. The workflow manager calls the resource manager and executes these tasks to ensure the recomputation of all missing data.

2.6 Implementation

We have implemented Alluxio[12] in about 36,000 lines of Java. Alluxio uses an existing storage system as its persistence layer, with a pluggable interface (we currently support HDFS, S3, GlusterFS, and NFS). Alluxio also uses Apache ZooKeeper to do leader election for master fault tolerance.

We have also implemented patches for existing frameworks to work with Alluxio: 300 Lines-of-Code (LoC) for Spark [146] and 200 LoC for MapReduce [15]. Applications on top of integrated frameworks can take advantage of the linage transparently, and application programmers do not need to know the lineage concept.

The project is open source, has over 40 [13] contributors from more than 15 companies and universities.

2.6.1 Lineage Metadata

This section describes the detailed information needed to construct a lineage.

Ordered input files list: Because files' names could be changed, each file is identified by a unique immutable file ID in the ordered list to ensure that the application's potential future recomputations read the same files in the same order as its first time execution.

Ordered output files list: This list shares the same insights as the input files list.

Binary program for recomputation: Alluxio launches this program to regenerate files when necessary. There are various approaches to implement a file recomputation program. One naïve way is to write a specific program for each application. However, this significantly burdens application programmers. Another solution is to write a single wrapper program which understands both Alluxio's lineage information and the application's logic. Though this may not be feasible for all programs, it works for applications written in a particular framework. Each framework can implement a wrapper to allow applications written in the framework to use Alluxio transparently. Therefore, no burden will be placed on application programmers.

Program configuration: Program configurations can be dramatically different in various jobs and frameworks. We address this by having Alluxio forego any attempt to understand these configurations. Alluxio simply views them as byte arrays, and leaves the work to program wrappers to understand. Based on our experience, it is fairly straightforward for each framework's wrapper program to understand its own configuration. For example, in Hadoop, configurations are kept in *HadoopConf*, while Spark stores these in *SparkEnv*. Therefore, their wrapper programs can serialize them into byte arrays during lineage submission, and deserialize them during recomputation.

Dependency type: We use *wide* and *narrow* dependencies for efficient recovery(*c.f.,* §2.5). *Narrow* dependencies represent programs that do operations, *e.g.,* filter and map, where each output file only requires one input file. *Wide* dependencies represent programs that do operations, *e.g.,*

[12]Based on the Alluxio open source status in 2013
[13]In 2013

shuffle and join, where each output file requires more than one input file. This works similarly to Spark [146].

2.6.2 Framework Integration

When a program written in a framework runs, before it writes files, it provides the aforementioned information (§2.6.1) to Alluxio. Then, when the program writes files, Alluxio recognizes the files contained in the lineage. Therefore, the program can write files to memory only, and Alluxio relies on the lineage to achieve fault tolerance. If any file gets lost, and needs to be recomputed, Alluxio launches the binary program, a wrapper under a framework invoking user application's logic, which is stored in the corresponding lineage instance, and provides the lineage information as well as the list of lost files to the recomputation program to regenerate the data.

Recomputation Granularity Framework integration can be done at different granularity, with different integration complexity and recovery efficiency. An easy but less efficient integration is to recompute data at the job level. Suppose a Map job produces ten files and one file is lost. For job level integration, Alluxio launches the corresponding wrapper job to recompute all ten files. Another solution is to recompute data at the task (within a job) level, which is harder but more efficient than the above approach. With the same example, for task level integration, Alluxio launches the corresponding wrapper job to only recompute the lost file. Our integrations with MapReduce and Spark are at task level.

Configurable Feature Though applications on an integrated framework can have memory throughput writes transparently, we make this feature optional, which is configurable by applications. This is useful if the application does not tolerate any temporary unavailability for the generated data.

2.7 Evaluation

We evaluated Alluxio[14] [15] through a series of raw benchmarks and experiments based on real-world workloads.

Unless otherwise noted, our experiments ran on an Amazon EC2 cluster with 10 Gbps Ethernet. Each node had 32 cores, 244GB RAM, and 240GB of SSD. We used Hadoop (2.3.0) and Spark (0.9).

We compare Alluxio with an in-memory installation of Hadoop's HDFS (over RAMFS), which we dub MemHDFS. MemHDFS still replicates data across the network for writes but eliminates the slowdown from disk.

In summary, our results show the following:

[14]Based on the Alluxio implementation in 2013
[15]We use HDFS (2.3.0) as Alluxio's persistence layer in our evaluation.

- *Performance:* Alluxio can write data 110x faster than MemHDFS. It speeds up a realistic multi-job workflow by 4x over MemHDFS. In case of failure, it recovers in around one minute and still finishes 3.8x faster. Alluxio helps existing in-memory frameworks like Spark improve latency by moving storage off-heap. It recovers from master failure within 1 second.
- *Asynchronous Checkpointing:* The Edge algorithm outperforms any fixed checkpointing interval. Analysis shows that Alluxio can reduce replication caused network traffic up to 50%.
- *Recomputation Impact:* Recomputation has minor impact on other jobs. In addition, recomputation would consume less than 1.6% of cluster resources in traces from Facebook and Bing.

2.7.1 Performance

2.7.1.1 Raw Performance

We first compare Alluxio's write and read throughputs with MemHDFS. In each experiment, we ran 32 processes on each cluster node to write/read 1GB each, equivalent to 32GB per node. Both Alluxio and MemHDFS scaled linearly with number of nodes. Figure 2.7 shows our results.

For writes, Alluxio achieves 15GB/sec/node. Despite using 10Gbps Ethernet, MemHDFS write throughput is 0.14GB/sec/node, with a network bottleneck due to 3-way replication for fault tolerance. We also show the theoretical maximum performance for replication on this hardware: using only two copies of each block, the limit is 0.5GB/sec/node. On average, Alluxio outperforms MemHDFS by 110x, and the theoretical replication-based write limit by 30x.

For reads, Alluxio achieves 38GB/sec/node. We optimized HDFS read performance using two of its most recent features, HDFS caching and short-circuit reads. With these features, MemHDFS achieves 17 GB/sec/node. The reason Alluxio performs 2x better is that the HDFS API still requires an extra memory copy due to Java I/O streams.

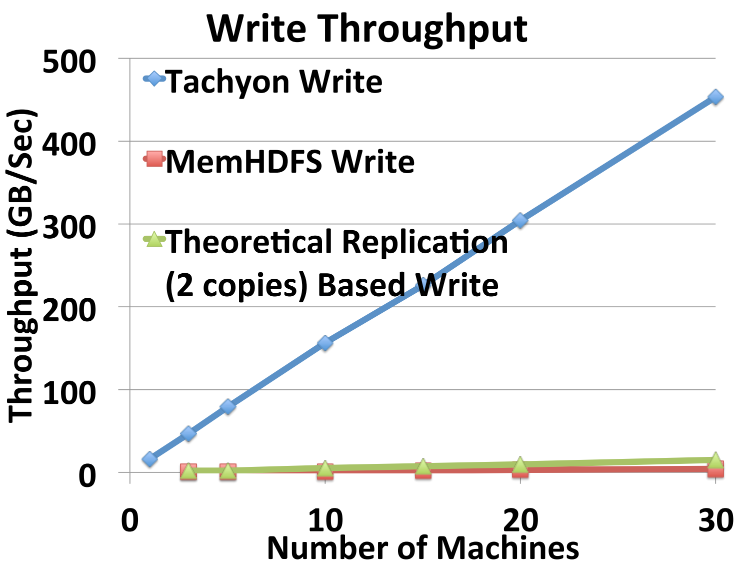
Note that Alluxio's read throughput was higher than write. This happens simply because memory hardware is generally optimized to leave more bandwidth for reads.

2.7.1.2 Realistic Workflow

In this experiment, we test how Alluxio performs with a realistic workload. The workflow is modeled after jobs run at a video analytics company during one hour. It contains periodic extract, transform and load (ETL) and metric reporting jobs. Many companies run similar workflows.

The experiments ran on a 30-node EC2 cluster. The whole workflow contains 240 jobs in 20 batches (8 Spark jobs and 4 MapReduce jobs per batch). Each batch of jobs read 1 TB and produced 500 GB. We used the Spark Grep job to emulate ETL applications, and MapReduce Word Count to emulate metric analysis applications. For each batch of jobs, we ran two Grep applications to pre-process the data. Then we ran Word Count to read the cleaned data and compute the final results. After getting the final results, the cleaned data was deleted.

We measured the end-to-end latency of the workflow running on Alluxio or MemHDFS. To



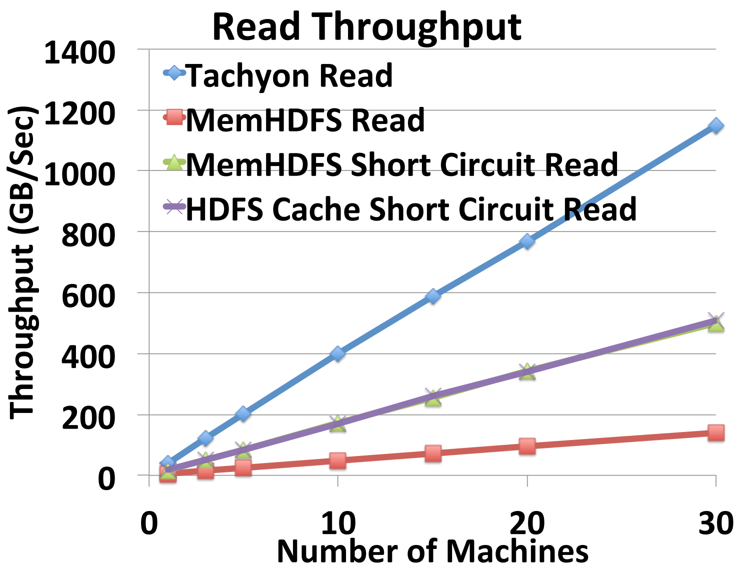


Figure 2.7: Alluxio, formerly Tachyon, and MemHDFS throughput comparison. On average, Alluxio outperforms MemHDFS 110x for write throughput, and 2x for read throughput.

simulate the real scenario, we started the workload as soon as raw data had been written to the system, in both Alluxio and MemHDFS tests. For the Alluxio setting, we also measured how long the workflow took with a node failure.

Figure 2.8 shows the workflow's performance on Alluxio and MemHDFS. The pipeline ran in 16.6 minutes on Alluxio and 67 minutes on HDFS. The speedup is around 4x. When a failure

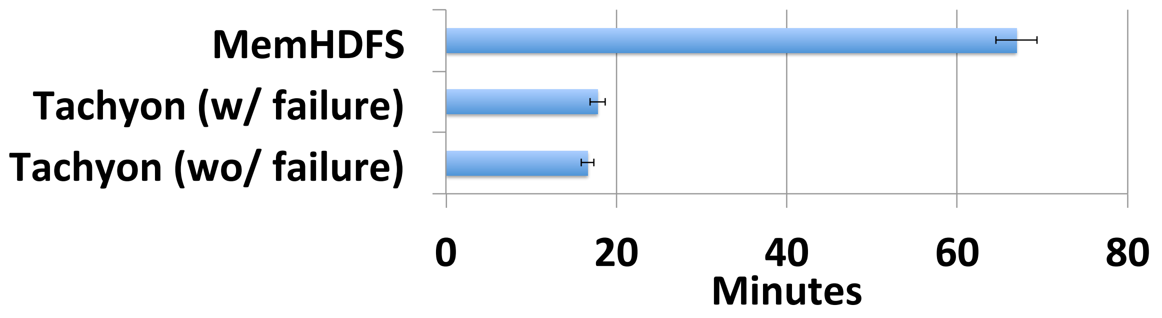


Figure 2.8: Performance comparison for realistic workflow. Each number is the average of three runs. The workflow ran 4x faster on Alluxio, formerly Tachyon, than on MemHDFS. In case of node failure, applications recovers in Alluxio around one minute and still finishes 3.8x faster.

happens in Alluxio, the workflow took 1 more minute, still finishing 3.8x faster than MemHDFS.

With Alluxio, the main overhead was serialization and de-serialization since we used the Hadoop TextInputFormat. With a more efficient serialization format, the performance gap is larger.

2.7.1.3 Overhead in Single Job

When running a single job instead of a pipeline, we found that Alluxio imposes minimal overhead, and can improve performance over current in-memory frameworks by reducing garbage collection overheads. We use Spark as an example, running a Word Count job on one worker node. Spark can natively cache data either as deserialized Java objects or as serialized byte arrays, which are more compact but create more processing overhead. We compare these modes with caching in Alluxio. For small data sizes, execution times are similar. When the data grows, however, Alluxio storage is faster than Spark's native modes because it avoids Java memory management.[16] These results show that Alluxio can be a drop-in alternative for current in-memory frameworks. Apache Spark official release 1.0.0 already uses Alluxio as its default off-heap storage solution.

2.7.1.4 Master Fault Tolerance

Alluxio utilizes hot failovers to achieve fast master recovery. We tested recovery for an instance with 1 to 5 million files, and found that the failover node resumed the master's role after acquiring leadership within 0.5 seconds, with a standard deviation of 0.1 second. This performance is possible because the failover constantly updates its file metadata based on the log of the current master.

[16] Although Alluxio is written in Java, it stores data in a Linux RAMFS.

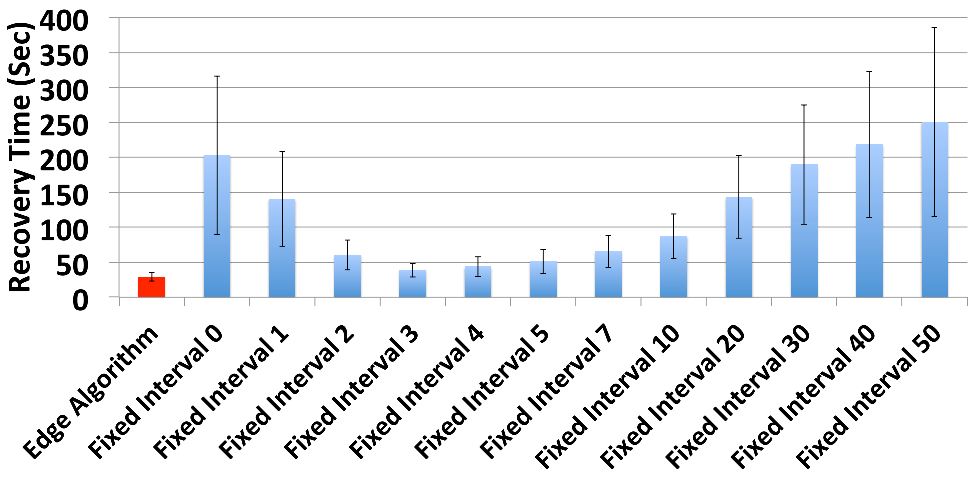


Figure 2.9: Edge and fixed interval checkpoint recovery performance comparison.

2.7.2 Asynchronous Checkpointing

2.7.2.1 Edge Checkpointing Algorithm

We evaluate the Edge algorithm by comparing it with fixed-interval checkpointing. We simulate an iterative workflow with 100 jobs, whose execution time follows a Gaussian distribution with a mean of 10 seconds per job. The output of each job in the workflow requires a fixed time of 15 seconds to checkpoint. During the workflow, one node fails at a random time.

Figure 2.9 compares the average recovery time of this workflow under Edge checkpointing with various fixed checkpoint intervals. We see that Edge always outperforms any fixed checkpoint interval. When too small an interval picked, checkpointing cannot keep up with program progress and starts lagging behind.[17] If the interval is too large, then the recovery time will suffer as the last checkpoint is too far back in time. Furthermore, even if an optimal *average* checkpoint interval is picked, it can perform worse than the Edge algorithm, which inherently varies its interval to always match the progress of the computation and can take into account the fact that different jobs in our workflow take different amounts of time.

We also simulated other variations of this workload, *e.g.,* more than one lineage chain or different average job execution times at different phases in one chain. These simulations have a similar result, with the gap between Edge algorithm and the best fixed interval being larger in more variable workloads.

[17]That is, the system is still busy checkpointing data from far in the past when a failure happens later in the lineage graph.

2.7.2.2 Network Traffic Reduction

Data replication from the filesystem consumes almost half the cross-rack traffic in data-intensive clusters [44]. Because Alluxio checkpoints data asynchronously some time after it was written, it can *avoid* replicating short-lived files altogether if they are deleted before Alluxio checkpoints them, and thus reduce this traffic. Networked bounded applications running in the same cluster can therefore leverage saved bandwidth to have better performance.

We analyze Alluxio's bandwidth savings via simulations with the following parameters:

- Let T be the ratio between the time it takes to checkpoint a job's output and the time to execute it. This depends on how IO-bound the application is. For example, we measured a Spark Grep program using Hadoop Text Input format, which resulted in T = 4.5, *i.e.,* the job runs 4.5x faster than network bandwidth. With a more efficient binary format, T will be larger.
- Let X be the percent of jobs that output permanent data. For example, 60% (X = 60) of generated data got deleted within 16 minutes at Facebook (Fig. 2.3a).
- Let Y be the percentage of jobs that read output of previous jobs. If Y is 100, the lineage is a chain. If Y is 0, the depth of the lineage is 1. Y is 84% in an analytics cluster at Twitter.

Based on this information, we set X as 60 and Y as 84. We simulated 1000 jobs using Edge checkpointing. Depending on T, the percent of network traffic saved over replication ranges from 40% at T = 4 to 50% at T $\geq$ 10.

2.7.3 Recomputation Impact

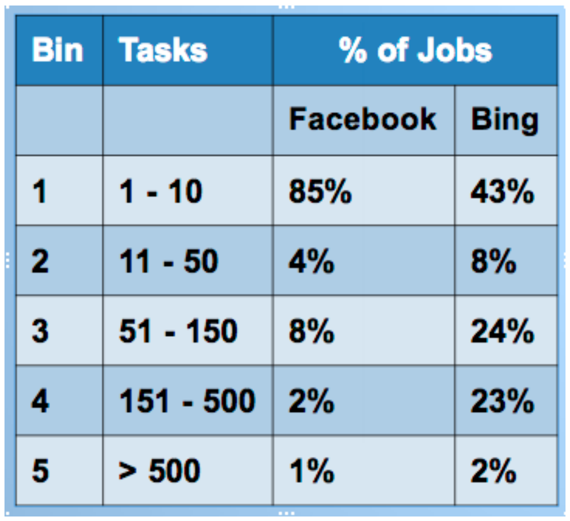
2.7.3.1 Recomputation Resource Consumption

Since Alluxio relies on lineage information to recompute missing data, it is critical to know how many resources will be spent on recomputation, given that failures happen every day in large clusters. In this section, we calculate the amount of resources spent recovering using both a mathematical model and traces from Facebook and Bing.

We make our analysis using the following assumptions:

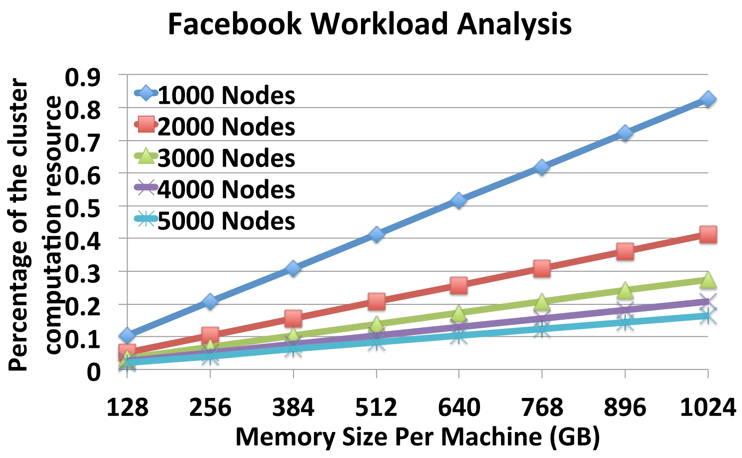
- Mean time to failure (MTTF) for each machine is 3 years. If a cluster contains 1000 nodes, on average, there is one node failure per day.
- Sustainable checkpoint throughput is 200MB/s/node.
- Resource consumption is measured in machine-hours.
- In this analysis, we assume Alluxio only uses the coarse-gained recomputation at the job level to compute worst case, even though it supports fine-grained recomputation at task level.

Worst-case analysis In the worst case, when a node fails, its memory contains only un-checkpointed data. This requires tasks that generate output faster than 200MB/sec: otherwise, data can be checkpointed in time. If a machine has 128GB memory, it requires 655 seconds (128GB / 200MB/sec) to recompute the lost data. Even if this data is recovered serially, and of *all* the other machines



Bin	Tasks	% of Jobs	
		Facebook	Bing
1	1 - 10	85%	43%
2	11 - 50	4%	8%
3	51 - 150	8%	24%
4	151 - 500	2%	23%
5	> 500	1%	2%

Trace Summary



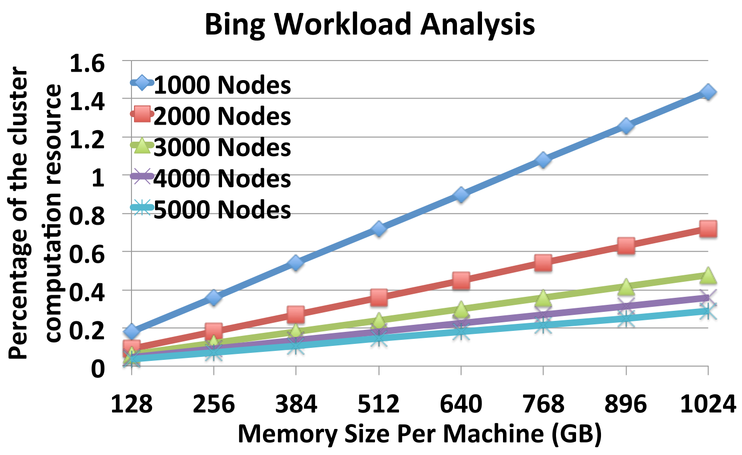


Figure 2.10: Using the trace from Facebook and Bing, recomputation consumes up to 0.9% and 1.6% of the resource in the worst case respectively.

are blocked waiting on the data during this process (*e.g.,* they were running a highly parallel job that depended on it), recomputation takes 0.7% (655 seconds / 24 hours) of the cluster's running time on a 1000-node cluster (with one failure per day). This cost scales linearly with the cluster size and memory size. For a cluster with 5000 nodes, each with 1TB memory, the upper bound on recomputation cost is 30% of the cluster resources, which is still small compared to the typical speedup from Alluxio.

Real traces In real workloads, the recomputation cost is much lower than in the worst-case setting above, because individual jobs rarely consume the entire cluster, so a node failure may not block all other nodes. (Another reason is that data blocks on a failed machine can often be recomputed in parallel, but we do not quantify this here.) Figure 2.10 estimates these costs based on job size traces from Facebook and Bing (from Table 2 in [9]), performing a similar computation as above with the active job sizes in these clusters. With the same 5000-node cluster, recomputation consumes only up to 0.9% and 1.6% of resources at Facebook and Bing respectively. Given most clusters are only 30–50% utilized, this overhead is negligible.

2.7.3.2 Impact of Recomputation on Other Jobs

In this experiment, we show that recomputating lost data does not noticeably impact other users' jobs that do not depend on the lost data. The experiment has two users, each running a Spark ETL pipeline. We ran the test three times, and report the average. Without a node failure, both users' pipelines executed in 85 seconds on average (standard deviation: 3s). With a failure, the unimpacted users's execution time was 86s (std.dev. 3.5s) and the impacted user's time was 114s (std.dev. 5.5s).

2.8 Related Work

Distributed Storage Systems In data analytics, distributed file systems [28, 121, 137], *e.g.,* GFS [60] and FDS [99], and key/value stores [11, 10, 58], *e.g.,* RAMCloud [103] and HBase [17], replicate data to different nodes for fault-tolerance. Their write throughput is bottlenecked by network bandwidth. FDS uses a fast network to achieve higher throughput. Despite the higher cost of building FDS, its throughput is still far from memory throughput. Alluxio leverages the lineage concept in the storage layer to eschew replication and instead stores a single in-memory copy of files. The Apache HDFS community is shifting towards pushing the lineage into the system, which they claimed was partially inspired by Alluxio [114, 77]. The proposed system maintains materialized views in un-replicated memory and can recompute them using the required SQL computations.

BAD-FS [29] separates out a scheduler from storage, which provides external control to the scheduler. Users submit jobs to the scheduler through a declarative workflow language, and thus the system knows the lineage among jobs. However, to make this practical for parallel big data engines (e.g. Spark, MapReduce, Tez), two fundamental issues stood out: (1) ensuring that the interaction between the scheduler and the storage system is correct, e.g. avoiding deadlocks/priority inversion, (2) bounding recomputation time, which otherwise in a 24/7 system can grow unbounded. For the first problem, we provide mechanisms that avoid deadlocks/priority inversion, while respecting the policy of schedulers. For the second problem, we provide an asynchronous checkpointing algorithm that exploits the structure of the lineage graph to continuously perform checkpointing in the background to guarantee bounded recovery time.

Cluster Computation Frameworks Spark [146] uses lineage information within a single job or shell, all running inside a single JVM. Different queries in Spark cannot share datasets (RDD) in a reliable and high-throughput fashion, because Spark is a computation engine, rather than a storage system. Our integration with Spark substantially improves existing industry workflows of Spark jobs, as they can share datasets reliably through Alluxio. Moreover, Spark can benefit from the asynchronous checkpointing in Alluxio, which enables high-throughput write.

Other frameworks, such as MapReduce [50] and Dryad [79], also trace task lineage within a job. However, as execution engines, they do not trace relations among files, and therefore can not provide high throughput data sharing among different jobs. Like Spark, they can also integrate with Alluxio to improve the efficiency of data sharing among different jobs or frameworks.

Caching Systems Like Alluxio, Nectar [69] also uses the concept of lineage, but it does so only for a specific programming framework (DryadLINQ [143]), and in the context of a traditional, replicated file system. Nectar is a data reuse system for DryadLINQ queries whose goals are to save space and to avoid redundant computations. The former goal is achieved by deleting largely unused files and rerunning the jobs that created them when needed. However, no time bound is provided to retrieve deleted data. The latter goal is achieved by identifying pieces of code that are common in different programs and reusing previously computed files. Nectar achieves this by heavily resting on the SQL like DryadLINQ query semantics—in particular, it needs to analyze LINQ code to determine when results may be reused—and stores data in a replicated on-disk file system rather than attempting to speed up data access. In contrast, Alluxio's goal is to provide data sharing across *different* frameworks with *memory speed* and *bounded recovery time*.

PACMan [9] is a memory caching system for data-intensive parallel jobs. It explores different policies to make data warehouse caching efficient. However, PACMan does not improve the performance of writes, nor the first read of any data item. Therefore, the throughput of many applications remains disk-limited—for example, a multi-job MapReduce workflow will still need to replicate data at disk speed between each step.

Lineage Besides the applications in the above fields, lineage has been applied in other areas, such as scientific computing [131, 32], databases [43], and distributed computing [5, 4, 98], for applications such as providing the history of data, which can be used for accountability, security, and data audit. Surveys [43, 32] provide detailed surveys of this work. Alluxio applies lineage in a different environment to achieve memory throughput read and write, which entails a different set of challenges.

Checkpoint Research Checkpointing has been a rich research area. Much of the research was on using checkpoints to minimize the re-execution cost when failures happen during long jobs. For instance, much focus was on optimal checkpoint intervals [132, 142], as well as reducing the per-checkpoint overhead [57, 107, 108]. Unlike previous work, which uses synchronous checkpoints, Alluxio does checkpointing asynchronously in the background, which is enabled by using lineage information to recompute any missing data if a checkpoint fails to finish.

2.9 Limitations and Future Work

Alluxio aims to improve the performance for its targeted workloads(§2.2.1), and the evaluations show promising results. Although many big data clusters are running our targeted workloads, we realize that there are cases in which Alluxio provides limited improvement, *e.g.,* CPU or network intensive jobs. In addition, there are also challenges that future work needs to address:

Random Access Abstractions: Data-intensive applications run jobs that output files to storage systems like Alluxio. Often, these files are reprocessed (*e.g.,* to a DBMS) to allow user-facing applications to use this data. For instance, a Web application might recommend new friends to a user. Thus, enabling Alluxio to provide higher-level read-only random-access abstractions, such as a key/value interface on top of the existing files would shorten the pipeline and enable output of data-intensive jobs to be immediately usable.

Mutable data: This is challenging as lineage cannot generally be efficiently stored for fine-grained random-access updates. However, there are several directions, such as exploiting deterministic updates and batch updates.

Multi-tenancy: Memory fair sharing [111, 62] is an important research direction for Alluxio. Policies such as LRU/LFU might provide good overall performance at the expense of providing isolation guarantees to individual users. In addition, security is also another interesting issue to tackle.

Hierarchical storage: Though memory capacity grows exponentially each year, it is still comparatively expensive to its alternatives. One early adopter of Alluxio suggested that besides utilizing the memory layer, Alluxio should also leverage NVRAM and SSDs. In the future, we will investigate how to support hierarchical storage in Alluxio.

Checkpoint Algorithm Optimizations: In this work, we proposed the Edge algorithm to provide a bounded recomputation time. It also checkpoints hot files first and avoids checkpointing temporary files. However, there could be other techniques to make trade-offs among different metrics, e.g. checkpoint cost, single file recovery cost, and all files recovery cost. We leave those as future work and encourage the community to explore different algorithms in this area.

2.10 Conclusion

As ever more datacenter workloads start to be in memory, write throughput becomes a major bottleneck for applications. Therefore, we believe that lineage-based recovery might be the only way to speed up cluster storage systems to achieve memory throughput. We proposed Alluxio with lineage, a memory-centric VDFS implementation system that incorporates lineage to speed up the significant part of the workload consisting of deterministic batch jobs. We identify and address some of the key challenges in making Alluxio practical. Our evaluations show that Alluxio provides promising speedups over existing storage alternatives. Approaches similar to ours are also being adopted by the HDFS community to leverage memory efficiently on clusters. We have open sourced Alluxio at `www.alluxio.org` for people to use in real environments or as a system research platform.

Chapter 3

Distributed File System Performance Evaluation Framework

3.1 Introduction

We have witnessed the emergence of parallel programming frameworks (e.g., MapReduce [50], Dremel [94], Spark [146, 145]) and distributed data stores (e.g., BigTable [40], Dynamo [52], PNUTS [47], HBase [18]) for enabling sophisticated and large-scale data processing and analytic tasks. Such distributed computing systems often build atop a *distributed file system (DFS)* (e.g., Lustre [33], Google File System [61], GlusterFS [64], CephFS [137], Hadoop Distributed File System (HDFS) [120], Alluxio[1] [84]) for scalable and reliable storage management. A typical DFS stripes data across multiple storage nodes (or servers), and also adds redundancy to the stored data (e.g., by replication or erasure coding) to provide fault tolerance against node failures.

There have been a spate of DFS proposals from both academia and industry. These proposals have inherently distinct performance characteristics, features, and design considerations. When putting a DFS in use, users need to decide the appropriate configurations of a wide range of design features (e.g., fine-grained reads, file backup, access controls, POSIX compatibility, load balance, etc.), which in turn affect the perceived performance and functionalities (e.g., throughput, fault tolerance, security, exported APIs, scalability, etc.) of the upper-layer distributed computing systems. In addition, the characteristics of processing and storage workloads are critical to the development and evaluation of file system implementations [130], yet they often vary significantly across deployment environments. All these concerns motivate the need of comprehensive DFS benchmarking methodologies to systematically characterize the performance and design trade-offs of general DFS implementations with regard to cluster configurations and workloads.

Building a comprehensive DFS benchmarking framework is non-trivial, and it should achieve two main design goals. First, it should be *scalable*, such that it can run in a distributed environment to support stress-tests for various DFS scales. It should incur low measurement overheads

[1]With different storage systems as Alluxio's under storage

in benchmarking for accurate characterization. Second, it should be *extensible*, such that it can easily include general DFS implementations for benchmarking for fair comparisons. It should also support both popular and user-customized workloads to address various deployment environments.

Table 3.1: Comparison of distributed storage benchmark frameworks

Benchmark Frameworks	**Extend to any targeted systems**	**Contain workloads from read world**	**Support to plug in new workload**	**Provide app trace analysis utility**	**Can run without computing framework**	**Support various parallel model**	**Is aimed to benchmark DFS**
TestDFSIO / NNBench	No	No	No	No	No	No	Yes
IOR	No	No	No	No	No	No	Yes
YCSB	Yes	No	Yes	No	Yes	No	No
AMPLab Big Data Benchmark	No	Yes	Yes	No	Yes	No	No
DFS-Perf	Yes	Yes	Yes	Yes	Yes	Yes	Yes

This paper proposes *DFS-Perf*, a scalable and extensible DFS benchmarking framework designed for comprehensive performance benchmarking of general DFS implementations. Table 3.1 summarizes the key features of DFS-Perf compared with existing distributed storage benchmarking systems (see Section 3.3 for details). DFS-Perf is designed as a distributed system that supports different parallel test modes at node, process, and thread levels for scalability. It also adopts a modular architecture that can benchmark large-scale deployments of general DFS implementations and workloads. As a proof-of-concept, DFS-Perf currently incorporates built-in workloads of machine learning and SQL query applications. We have open sourced DFS-Perf[2].

We first review today's representative DFS implementations (Section 3.2) and existing DFS benchmarking methodologies (Section 3.3). We then make the following contributions.

1. We present the design of *DFS-Perf*, a highly scalable and extensible benchmarking framework (Section 3.4) that supports various DFS implementations and big data workloads, including machine learning and SQL workloads (Section 3.5).

2. We use DFS-Perf to evaluate the performance characteristics of four widely deployed DFS implementations, Alluxio, CephFS, GlusterFS, and HDFS, and show that DFS-Perf incurs minimal (i.e., 5.7%) overhead (Section 3.6).

3. We report our experiences of using DFS-Perf to identify and resolve performance bugs of current DFS implementations (Section 3.7).

3.2 DFS Characteristics

Existing DFS implementations are generally geared toward achieving scalable and reliable storage management, yet they also make inherently different design choices for their target applications and scenarios. In this section, we compare four representative open-source DFS implementations,

[2]https://github.com/PasaLab/dfs-perf

Table 3.2: Comparison of characteristics of Alluxio, CephFS, GlusterFS, and HDFS

DFS	**Architecture**	**Storage Style**	**Fault Tolerance**	**I/O Optimization**	**Provided APIs**
Alluxio	centralized	memory-centric; hierarchy	lineage and checkpoint	data locality; multi-level caches	Native API; FUSE; Hadoop Compatible API; CLI
CephFS	centralized / distributed	disk-based	replication; erasure code (optional)	cache tiering	FUSE; REST-API; Hadoop Compatible API
GlusterFS	decentralized	disk-based	RAID on the network	I/O cache	FUSE; REST-API
HDFS	centralized / distributed	disk-based	replication; erasure code (WIP)	data locality	Native API; FUSE; REST-API; CLI

namely Alluxio [126, 84], CephFS [137], GlusterFS [65, 64], and HDFS [73, 120]. We review their different design choices, which also guide our DFS-Perf design. Table 3.2 summarizes the key characteristics of the four DFS implementations.

Architecture. Alluxio, CephFS, and HDFS adopt a *centralized* master-slave architecture, in which a master node manages all metadata and coordinates file system operations, while multiple (slave) nodes store the actual file data. CephFS and HDFS also support multiple distributed master nodes to avoid a single point of failure. On the other hand, GlusterFS adopts a *decentralized* architecture, in which all metadata and file data is scattered across all storage nodes through the distributed hash table (DHT) mechanism.

Storage Style. CephFS, GlusterFS, and HDFS build on *disk-based* storage, in which persistent block devices (e.g., hard disks). On the other hand, Alluxio is *memory-centric*, and uses memory as the primary storage backend. It also supports *hierarchical* storage which aggregates the pool of different storage resources such as memory, solid-state disks, and hard disks.

Fault Tolerance. For fault tolerance, CephFS and HDFS mainly use replication to distribute exact copies across multiple nodes for fault tolerance. They now also support erasure coding for more storage-efficient fault tolerance. GlusterFS implements RAID (which can be viewed as a special type of erasure coding) at the volume level. In contrast, Alluxio can leverage its under storage systems, in the meantime, it can also adopt lineage and checkpointing mechanisms to keep track of the operational history of computations.

I/O Optimization. All four DFSes use different strategies to improve the I/O performance. CephFS adopts a *cache tiering* mechanism to temporarily cache the recent read and written data in memory, while GlusterFS follows a similar caching approach (called *I/O cache*). On the other hand, both HDFS and Alluxio enforce data locality in computing frameworks (e.g., MapReduce, Spark, etc.) to ensure that computing tasks can access data locally in the same node. In particular, Alluxio supports explicit multi-level caches due to its hierarchical storage architecture.

Exported APIs. All four DFSes export APIs that can work seamlessly with Linux FUSE. HDFS and Alluxio export native APIs and a command line interface (CLI) that can work independently without third-party libraries. In particular, both CephFS and Alluxio have the Hadoop-compatible APIs and can substitute HDFS for some computing frameworks including MapReduce and Spark.

Discussion. Because of the variations of design choices, the application scenarios vary across DFS implementations. GlusterFS, CephFS, and Lustre [138, 33] are commonly used in high-performance computing environments. HDFS has been used in big data analytics applications along with the wide deployment of MapReduce. QFS [104] can work in conjunction with MapReduce to provide more efficient file sharing than HDFS. Alluxio provides file access at memory speed across cluster computation frameworks. The large variations of design choices complicate the decision making of practitioners when they choose the appropriate DFS solutions for their applications and workloads. Thus, a unified and effective benchmarking methodology becomes critical for practitioners to better understand the performance characteristics and design trade- offs of a DFS implementation.

3.3 Related Work

Benchmarking is essential for evaluating and reasoning the performance of systems. Some benchmark suites (e.g., [122, 123]) have been designed for evaluating general file and storage systems subject to different workloads. Benchmarking for DFS implementations has also been proposed in the past decades, such as for single-server network file systems [31], network-attached storage systems [63], and parallel file systems (e.g., IOR [105]). Several benchmarking suites are designed for specific DFS implementations, such as TestDFSIO, NNBench, and HiBench [100, 74, 76] for HDFS. To elaborate, TestDFSIO specifies read and write workloads for measuring HDFS throughput; NNBench specifies metadata operations for stress-testing an HDFS namenode; HiBench supports both synthetic microbenchmarks and Hadoop application workloads that can be used for HDFS benchmarking. Instead of targeting specific DFS implementations, we focus on benchmarking for general DFS implementations.

Some benchmarking suites can be used to characterize the I/O behaviors of general distributed computing systems. For example, the AMPLab Big Data Benchmark [7] issues relational queries for benchmarking and provides quantitative and qualitative comparisons of analytical framework systems, and YCSB (Yahoo Cloud Serving Benchmark) [141, 46] evaluates the performance of key-value cloud serving stores. Both benchmark suites are extensible to include user-defined operations and workloads with database-like schemas.

The MTC (Many-Task Computing) envelope [147] characterizes the performance of metadata, read, and write operations of parallel scripting applications. BigDataBench [136] targets big data applications such as online services, offline analytics, and real-time analytics systems, and provides various big data workloads and real-world datasets. In contrast, DFS-Perf focuses on file system operations, including both metadata and file data operations, for general DFS implementations.

One design consideration of DFS-Perf is on workload characterization, which provides guide-lines for system optimizations. Workload characterization in distributed systems has been an active research topic. To name a few, Yadwadkar et al. [140] identified the application-level workloads from NFS traces. Chen et al. [42] studied MapReduce workloads from business- critical deployments. Harter et al. [72] studied the Facebook Messages system backed by HBase and HDFS as the storage layer. Our workload characterization focuses on DFS-based traces derived from real-world applications.

3.4 DFS-Perf Design

We present the design details of DFS-Perf. We first provide an architectural overview of DFS-Perf (Section 3.4.1). We then explain how DFS-Perf achieves scalability (Section 3.4.2) and extensibility (Section 3.4.3).

3.4.1 DFS-Perf Architecture

DFS-Perf is designed as a distributed architecture that runs on top of a DFS that is to be benchmarked. It follows a master-slave architecture, as shown in Figure 3.1, in which the single master process coordinates multiple slave processes, each of which issues file system operations to the underlying DFS. The master consists of a launcher module that schedules slaves to run benchmarks, as well as a set of utility tools for benchmark management. For example, one utility tool is the report generator, which collects results from the slaves and produces performance reports.

Each slave is a multi-threaded process (see Section 3.4.2) that can be deployed on any cluster node to execute benchmark workloads through the underlying DFS.

Figure 3.2 illustrates the workflow of executing a benchmark in DFS-Perf. First, the master launches all slaves and distributes test configurations to each of them. Each registered slave performs initialization work, such as loading the configurations and setting up the workspace, and notifies the master when its initialization is completed. When all slaves are ready, the master notifies them to start executing the benchmark simultaneously. Each slave independently conducts the benchmark test by issuing a set of file system operations, including the metadata and file data operations that interact with the DFS's master and storage nodes, respectively. It also collects the performance statistics from its own running context. Finally, after all slaves issue all file system operations, the master collects the context information to produce a test report of the benchmark.

DFS-Perf performs benchmarking at the file system layer. Thus, it cannot measure the performance of the (lower-layer) internals of a DFS, such as the communications between the DFS's master and storage nodes, or the CPU usage of each DFS entity. Nevertheless, DFS-Perf does not need to modify the internals of a DFS, making it compatible with general DFS implementations.

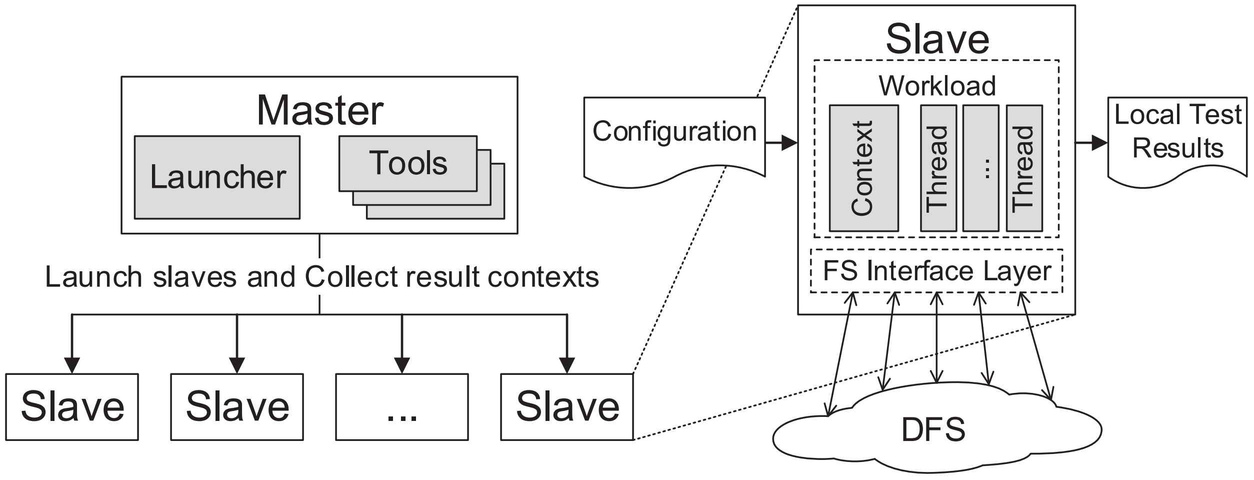


Figure 3.1: DFS-Perf architecture. DFS-Perf adopts a master-slave model, in which the master contains a launcher and other tools to manage all slaves. For each slave, the input is the benchmark configuration, while the output is a set of evaluation results. Each slave runs multiple threads, which interact with the DFS via the File System Interface Layer.

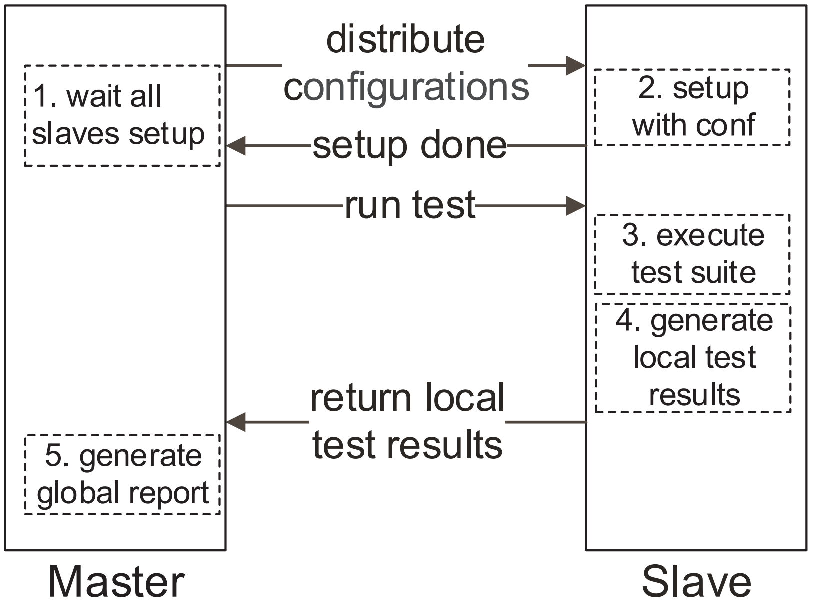

distribute configurations
1. wait all slaves setup
2. setup with conf
setup done
run test
3. execute test suite
4. generate local test results
return local test results
5. generate global report
Master
Slave

Figure 3.2: DFS-Perf workflow. The master launches all slaves and distributes configurations of a benchmark to each of them. Each slave sets up and executes the benchmark independently. Finally, the master collects the statistics from all slaves and produces a test report.

3.4.2 Scalability

To achieve scalability, DFS-Perf parallelizes benchmark executions in a multi-node, multi-process, and multi-thread manner: it distributes the benchmark execution through multiple nodes (or physical servers); each node can run multiple slave processes, and each slave process can run multiple threads that execute the benchmarks independently. The numbers of nodes, processes, and threads can be configured by the users. Such parallelization enables us to stress-test a DFS through intensive file system operations.

To reduce the benchmarking overhead, we only require each slave to issue only a total of *two* round-trip communications with the master, one at the beginning and one at the end of the benchmark execution (see Figure 3.2). That is, DFS-Perf itself does not introduce any communication to manage how each slave run benchmarks on the underlying DFS. Thus, the DFS-Perf framework puts limited performance overhead on the DFS during benchmarking.

3.4.3 Extensibility

DFS-Perf achieves extensibility in two aspects. First, DFS-Perf provides a plug-in interface via which users can add a new DFS implementation to be benchmarked by DFS-Perf. Second, DFS-Perf provides an interface via which users can customize specific workloads for their own applications to run atop the DFS.

3.4.3.1 Adding a DFS

DFS-Perf provides a general *File System Interface Layer* to abstract the interfaces of various DFS implementations. Each slave process interacts through the File System Interface Layer with the underlying DFS, as shown in Figure 3.1.

One design challenge of DFS-Perf is to support general file system APIs. One option is to make the File System Interface Layer of DFS-Perf POSIX-compliant, but general DFS implementations do not support POSIX APIs, which are commonly used in local file systems of Linux/UNIX but may not be suitable for high-performance parallel I/O [55]. Another option is to deploy Linux FUSE inside DFS-Perf, as it is supported by a number of DFS implementations (see Section 3.2). However, Linux FUSE runs at the user level and adds additional overheads to file system operations. Here, we carefully examine the APIs of state-of-the-art DFS implementations and classify the general file system APIs into four categories, which cover the most common basic file operations.

- **Session Management**: managing DFS sessions, including *connect* and *close* methods;
- **Metadata Management**: managing DFS metadata, including *create*, *delete*, *exists*, *getParent*, *isDirectory*, *isFile*, *list*, *mkdir*, and *rename* methods;
- **File Attribute Management**: managing file attributes, such as the file path, length, and the access, create, and modification time;

- **File Data Management**: the I/O operations that transfer the actual file data. Currently they are via InputStream and OutputStream due to the APIs provided by supported DFSes.

To elaborate, we abstract a base class called *DFS* that realizes the abstract methods (or interfaces) of above four categories of general DFS operations. The abstract methods of *DFS* constitute the File System Interface Layer in DFS-Perf. To support a new DFS, users only need to implement those abstract methods in a new class that inherit the base class *DFS*. Users also register the new DFS class into DFS-Perf by adding an identifier (in the form of a constant number) to map the new DFS to the implemented class. DFS-Perf differentiates the operations of a DFS implementation via a URL scheme in the form of *fs://*. Note that users need not be concerned about the synchronization issues of APIs, which are handled by DFS-Perf.

Take CephFS as an example. We add a new class named *DFS_Ceph* and implement all abstract methods for interacting with CephFS. Our implementation comprises only less than 150 lines of Java codes. To specify file system operations with CephFS in a benchmark, users can specify the operations in the form of *ceph://*.

Our current DFS-Perf prototype has implemented the bindings of several DFS implementations, including Alluxio, CephFS, GlusterFS, and HDFS. It also includes the bindings of the local file system.

3.4.3.2 Adding a New Workload

DFS-Perf achieves loose coupling between a workload and the DFS-Perf execution framework. Users can simply define a new workload in DFS-Perf by realizing several base classes and a configuration file, as shown in Table 3.3. The base classes specify the execution logic and the measurement statistics of a workload, while the configuration file consists of the settings of a workload, such as the information about testing data sizes and distributions, the file numbers and locations, and the I/O buffer sizes. All configurations are described in XML format that can be easily configured. Each slave process will take both the base classes and the configuration file of a workload.

3.5 Benchmark Design

A practical DFS generally supports a variety of applications for big data processing. To demonstrate how DFS-Perf can benchmark a DFS against big data applications, we have designed and implemented built-in benchmarks for two widely used groups of big data applications, namely machine learning and SQL query applications. For machine learning, we consider two applications: KMeans and Bayes, which represent a clustering algorithm and a classification algorithm, respectively [22]. For SQL queries, we consider three typical query applications: select, aggregation, and join [106].

Table 3.3: Components for a workload definition: base classes and a configuration file.

Sub-Component	**Meaning**
PerfThread	The class that contains all workload execution logic of a thread.
PerfTaskContext	The class that maintains the measurement statistics and running status.
PerfTask	The class that keeps all threads and conducts the initialization and termination work.
PerfGlobalReport	The class that generates the test report from all workload contexts.
configure.xml	The configuration file that manages the workload settings in XML format.

3.5.1 DFS Access Traces of Applications

We first study and design workloads of the five big data applications based on their access patterns. Tarasov et al [**TarasovKMHPKZ12**] has done the work that can extract workload models from large I/O traces, including I/O size, CPU utilization, memory usage, and power consumption. However, to sufficiently reflect the performance of applications, DFS-Perf focuses on the DFS-related traces, since these traces show how the applications interact with a DFS. Specifically, we run each application on Alluxio with a customized Alluxio client, which records all operation actions. We then analyze the DFS-related traces based on the output logs. In our study, the DFS-related access operations can be divided into four categories: sequential writes, sequential reads, random reads, and metadata operations. The first three types of operations are collectively called file data operations. Here, a sequential read means reading a file sequentially from the head to the end, while a random read means reading a file randomly by skipping to different locations. Since a DFS rarely supports random writes [61], there is no random write operation in our collected access traces.

3.5.1.1 Overview of File Operation Traces

We first analyze the access patterns of the benchmarks. We have collected file operation-related traces, and summarizes the traces of each benchmark in terms of the percentage of read/write operation counts. Figure 3.3 illustrates that these workloads all issue more reads than writes, although they have different read-to-write ratios. The SQL query workloads have more balanced

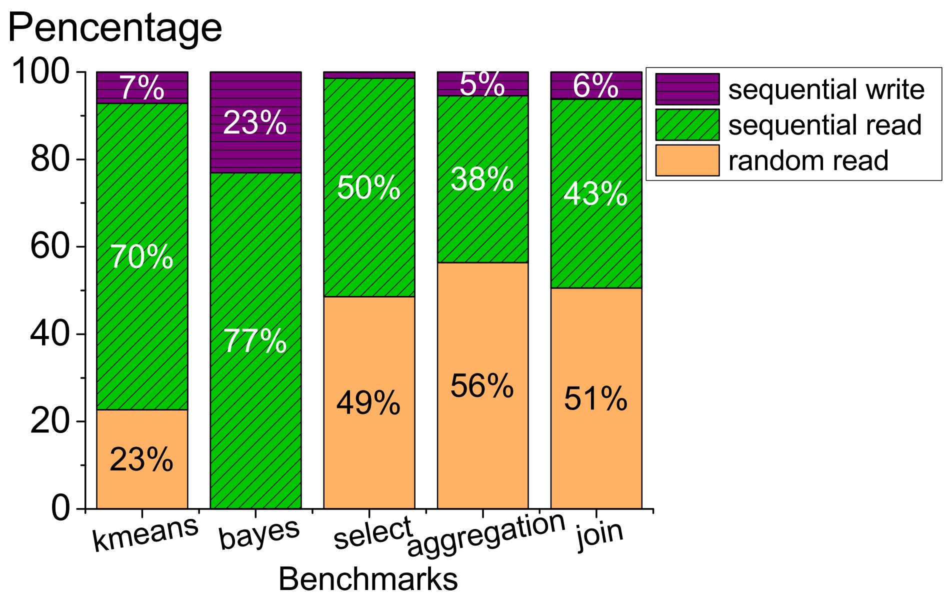


Figure 3.3: Summary of DFS-related access traces of the five applications. All of them issues reads more than writes, and the proportion of sequential and random read is more balanced in SQL query applications than in machine learning applications.

proportions of sequential reads and random reads than machine learning workloads. Also, the aggregation application has the highest percentage of random reads (up to 56%), while the Bayes application has no random read operation at all.

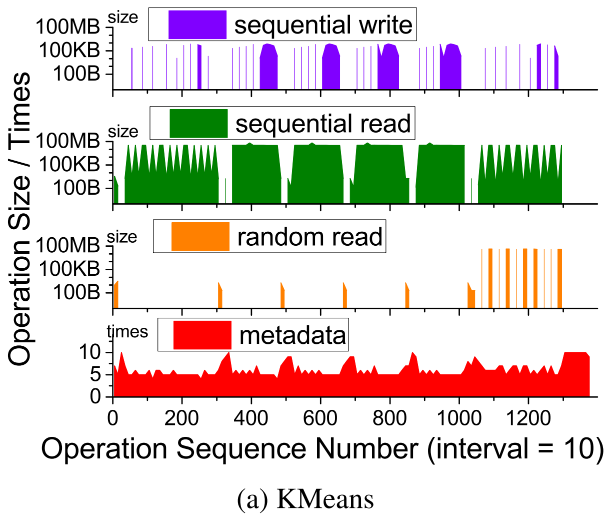
3.5.1.2 Machine Learning Benchmarks

We analyze the DFS access traces of the machine learning benchmarks in detail. Figure 3.4 and Figure 3.5 show the detailed traces. The X-axis is the operation sequence number, each of which corresponds to an operation; the Y-axis is the data sizes of read and write operations, or times of metadata operations. For brevity, in each benchmark, we choose an interval and measure the aggregated operation size or times in each interval. These measurements are affected by various factors in different benchmarks, but we can still summarize the access patterns from their trends.

Figure 3.4 demonstrates the DFS access traces of the KMeans and Bayes training processes. It is obvious that these data analytic applications access a DFS in an iterative fashion. A training process contains several iterations, each of which reads a variable size of data (ranging from several kilobytes to several gigabytes). In the last round, the result determining the size of write operations is the output. Note that sequential reads appear much more frequently than random reads.

3.5.1.3 SQL Query Benchmarks

We now analyze the DFS access traces of the SQL Query Benchmarks. Many big data query systems, such as Hive, Pig, and SparkSQL, convert SQL queries into a series of MapReduce or



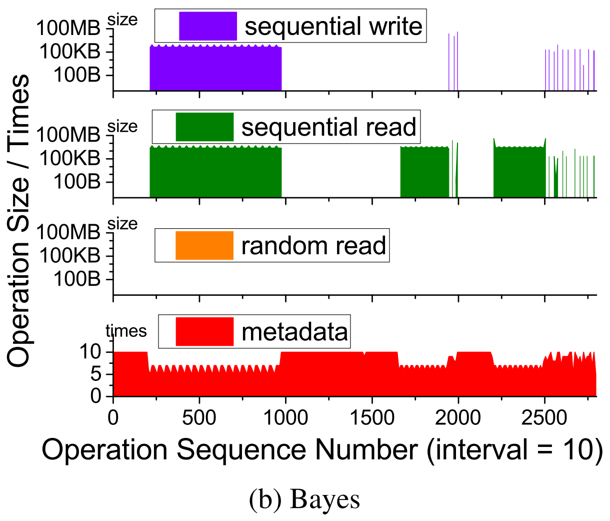
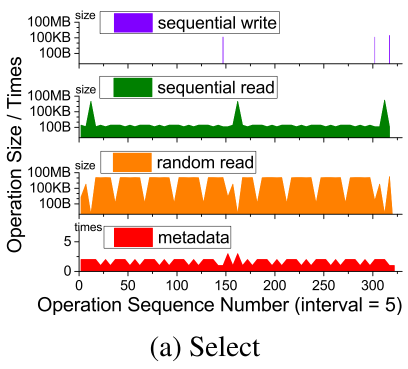
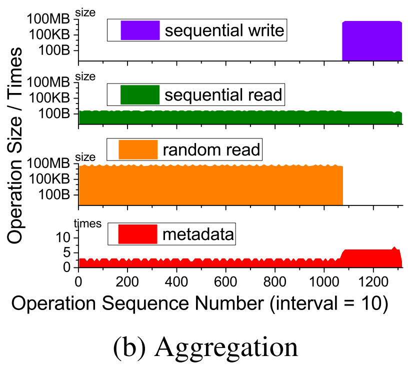


Figure 3.4: DFS Access Traces of Machine Learning Benchmarks. (a) The KMeans training process is iterative. It reads a lot of data in each iteration and writes more in the end. (b) The Bayes training process can be obviously divided into several sub-processes, each of which is filled with many sequential reads and few sequential writes. Note that there is no random read.





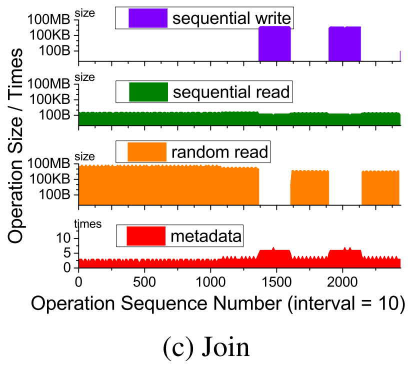


Figure 3.5: DFS Access Traces of SQL Query Benchmarks. The random read size is always more than the sequential read size. (a) Select: At about the halfway mark and the end of the whole process, there are sequential writes with a larger data size. (b) Aggregation: The sequential writes are concentrated at the end of the whole process. (c) Join: This is the longest process and we can obviously see two sequential write sub-processes.

Spark jobs. These applications mainly scan or filter data by sequentially reading data from a DFS in parallel. Also, they use indexing techniques to locate values, and hence trigger many random reads to a DFS.

In Figure 3.5a, we find that the select benchmark has at least thousands of times more random reads than sequential reads. However, the result size is small and thus there are only few sequential writes. The aggregation benchmark is more complex than the select benchmark. As shown in

Figure 3.5b, the aggregation benchmark keeps reading data and executing the computation logic, and finally it writes the result to a DFS. The number of the random reads is still more than that of sequential reads. Also, there is an obvious output process due to the large size of the result. In Figure 3.5c, the whole process of the join benchmark can be split into several read and write sub-processes. The read sub-processes are filled with random reads, while the write sub-processes are mixed with sequential reads and sequential writes.

In summary, the SQL query benchmarks generate more random reads than sequential reads. In addition, the reads and writes are mixed with a certain ratio which is determined by the data size.

3.5.2 Workloads

Based on our collected access traces in Section 3.5.1, we accordingly design five built-in workloads in DFS-Perf to better simulate the distributed I/O characteristics of typical big data applications. Each workload in DFS-Perf has its configurations and execution logic. The configurations make the workload flexible to evaluate testing cases with various scales and loads, while the execution logic represents a set of applications or typical cases. The DFS operators of the workloads include sequential reads, random reads, sequential writes, and metadata operations. Further, we support more sophisticated workloads, which have configurable DFS access patterns of real-world typical applications, such as mixed read/write workloads and iterative read/write workloads.

Metadata Operations Workload: This workload focuses on performing metadata operations such as *create*, *exist*, *rename*, and *delete*. In metadata-centralized DFS, e.g., Alluxio and HDFS, this workload can test the performance of the saturated metadata node. While in metadata-decentralized DFS, e.g., GlusterFS, it can perform a stress test for the correctness and synchronization performance of the whole cluster. Meanwhile, the number of connections is configurable, so that we can use this workload to evaluate the upper limit of concurrency.

Sequential/Random Read/Write Workload: The simple read and write workloads are to sequentially read and write a list of files, respectively. As the basic operations of a file system, they are used to test the throughput of a DFS. In addition, we provide a random read workload to represent the indexing access in databases or the searching access in algorithms. This workload randomly skips a certain length of data and then reads another certain length of data instead of sequentially reading the whole file. Note that a DFS usually provides data streams for reading across the network, and thus the only way for current supported DFSes to perform random skips is to create a new input stream when skipping backward. However, DFS-Perf reserves the randomly reading interface that new DFS can have its own implementation. The write workload is to write content into new files, so it is also the data generator of DFS-Perf with the configurable data sizes and distributions.

Mixed Read/Write Workload: In general, the read-to-write ratio varies across applications. This workload is composed of a mixture of simple read and write workloads with a configurable ratio. It resembles much closer to the real-world applications with heavy reads (e.g., hot data storage like online videos) or heavy writes (e.g., historical data storage like trading information). In addition, mixed read and write workloads are often used to evaluate the cache and eviction performance of a hierarchical DFS.

Iterative Read/Write Workload: The iterative computing pattern is often found in large-scale graph computing and machine learning problems [146]. This workload represents the applications in which the output of the former iteration is the input to the next one. Specifically, we provide two modes called *Shuffle* and *Non-Shuffle* for data accesses. In *Shuffle* mode, each slave process reads files from the whole workspace, which may lead to remote reading across the cluster network; in the *Non-Shuffle* mode, each slave process only reads the files written by itself, which keeps good locality and can benefit from the caching mechanism of some DFS implementations, such as Alluxio.

Irregular Massive Access Workload: Many applications have complex read or write patterns, like web servers. The features that we simulate with these applications are randomization and concurrency. In this irregular massive access workload, files are read or written randomly and concurrently. It can reflect the throughput performance of a DFS cluster close to a real-world situation. Moreover, similar to the iterative workload, this workload also has both *Shuffle* and *Non-Shuffle* modes.

3.5.3 Automatic Workload Generator

In addition to the built-in workloads, DFS-Perf also facilitates users with a tool called the *Workload Generator* that can generate specific workloads automatically. Similar to Section 3.5.1, DFS-Perf provides each supported DFS with a wrapped client implementation that can record all the DFS-related operations. And with a few extra configurations, applications are able to access DFSes with these customized clients. In this way, the Workload Generator can collect and analyze the DFS-related access traces and statistics for an application that runs on a DFS.

Figure 3.6 shows how the workload generator works. First, it traces the DFS-related behaviors of the application and logs the intermediate information. It then analyzes the information to output both the DFS-related statistics and traces. The statistics contain the statistical numbers such as the operation times, total data size, etc. With these statistics, the user can further configure the built-in workloads to match their behaviors to the application. On the other hand, the traces consist of all exact behavior records that represent how the application interacts with the underlying DFS. With these traces, DFS-Perf can generate a new workload which completely replays the DFS-related behaviors of the application, even on different parallel modes. In summary, the workload generator provides customized workloads for DFS-Perf to match the characteristics of real-world applications, so as to measure the pure DFS performance or to compare the performance of different DFS implementations for the same application.

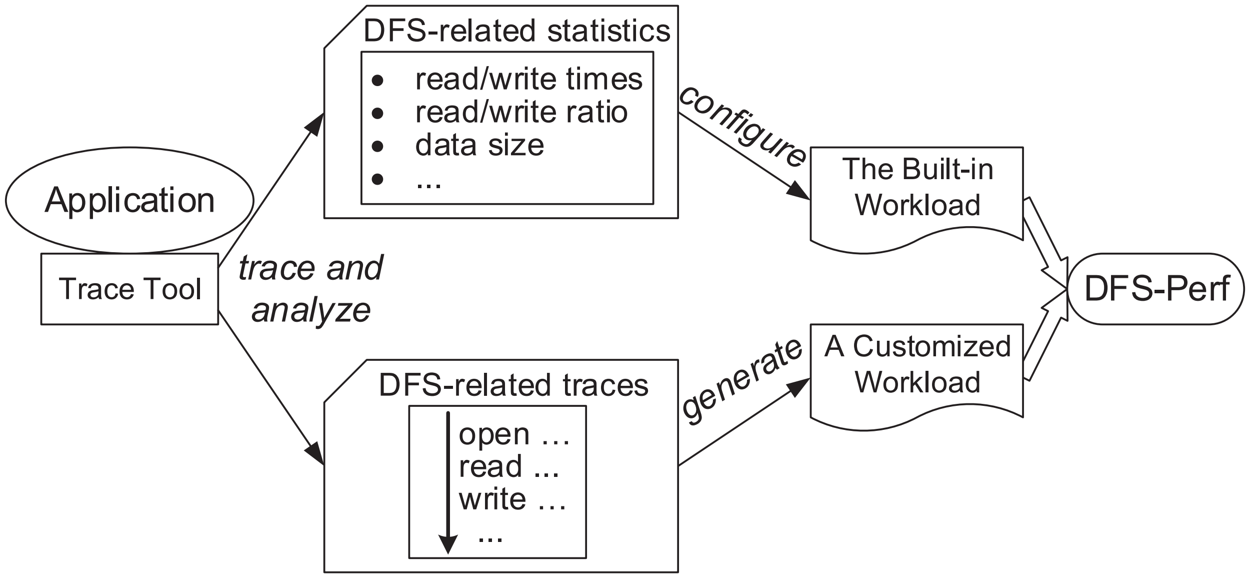


Figure 3.6: The *Workload Generator* collects DFS-related statistics and traces for an application that runs on a DFS. It outputs built-in workloads that are configurable, or customized workloads that match the application.

3.6 Evaluation

In this section, we present evaluation results for using DFS-Perf to benchmark several representative implementations, including Alluxio, CephFS, GlusterFS, and HDFS, on a testbed cluster. The highlights are:

- The characteristics of different DFS implementations have considerable impact on the performance. For example, the centralized and decentralized architectures significantly influence the metadata performance, while the storage style, fault tolerance, and I/O optimization characteristics can lead to various degrees of performance gains by orders of magnitude (Section 3.6.2).
- DFS-Perf is scalable to evaluate the performance upper-bounds of a DFS using different parallel modes. For the evaluated DFS, the throughput difference between the multi-process and multi-thread modes can reach $1.7\times$ (Section 3.6.3).
- DFS-Perf has a negligible overhead (about 5.7%), which ensures reliable benchmarking performance results. On the other hand, TestDFSIO (another DFS benchmarking tool) shows an increasing overhead (nearly 20%) as the concurrency degree increases (Section 3.6.4).

3.6.1 Experimental Setup

We conduct our experiments on a physical cluster with one master node and 40 slave nodes. The master has two Intel Xeon E5-2660v2 CPUs with total 20 cores (40 hyper-thread cores), while each slave node has two Intel Xeon E5-2620v2 CPUs with total 12 cores (24 hyper-thread cores). In other words, we have 40 nodes with 960 hyper-thread cores for processing file data operations. Each node has 64 GB DDR3 memory and 6 TB SAS RAID0 hard disk. All these nodes are connected with 1 Gb/s Ethernet. Each node runs RHEL 7.0 with Linux 3.10.0, Ext4 file system, and Java 1.7.0.

We deploy Alluxio 1.0.1, CephFS 0.94.6, GlusterFS 3.5.6, Hadoop 2.7.0, and our DFS-Perf on all nodes. The Ceph MDS (Metadata Server) runs on the master node and each slave node has a Ceph OSD (Object Storage Device) Daemon. For GlusterFS, each slave node runs both the client and server processes, and we configure it with FUSE. Hadoop runs the NameNode daemon on the master node and the DataNode daemon on each slave node. Alluxio runs the Master daemon on the master node and the Worker daemon on each of slave nodes. Each Alluxio Worker is configured to have 32 GB RAMFS, accounting for a total of 1280 GB memory across all slave nodes. Alluxio is a hierarchical distributed file system which allows users to store data in memory or in the underlying disk-based file system. For brevity, we use *Alluxio (In-Mem)* to refer to Alluxio that uses only in-memory data operations, while using *Alluxio (Through)* to refer to Alluxio that issues all operations to the underlying file system (HDFS in our case) without any in-memory access. We also take the local file system, denoted as *LocalFS*, for comparison. In addition, due to the ease of use, our DFS-Perf is deployed by two simple steps: set the configuration file, then distribute the whole package.

3.6.2 DFS Benchmarking

We apply the built-in workloads in DFS-Perf (Section 3.5.2) to benchmark different DFS implementations, so as to examine how different DFS characteristics affect the performance. For the multi-thread mode, as each slave has 24 hyper-thread cores, each node runs one process with 24 threads. The performance results are illustrated in Figures 3.7 to 3.11.

Metadata Operations. Figure 3.7 first illustrates the metadata performance, in terms of the number of operations per second. As expected, LocalFS outperforms others in the metadata performance, since its metadata operations run in single-node memory without any network or disk interaction. In particular, for DFS metadata operations, we observe that the centralized metadata management can achieve an up to 10$\times$ speedup than the decentralized one. GlusterFS has low metadata performance since its decentralized metadata management needs to synchronize metadata information over the cluster. CephFS has the worst performance although it is metadata-centralized, because the Ceph MDS only handles metadata requests and all metadata is stored in several Ceph OSDs on different nodes. In contrast, Alluxio and HDFS achieve higher metadata performance than GlusterFS since they both store and manage metadata on a single node. However, Alluxio is about 26% slower than HDFS in our evaluation. The reason is that both the Alluxio Master and HDFS NameNode generate edit logs of metadata operations. The Alluxio Master will rotate the edit log when it reaches a certain size, while HDFS performs this work in a separated module named SecondaryNameNode. Thus, HDFS NameNode can deal with more simultaneous metadata requests than the Alluxio Master.

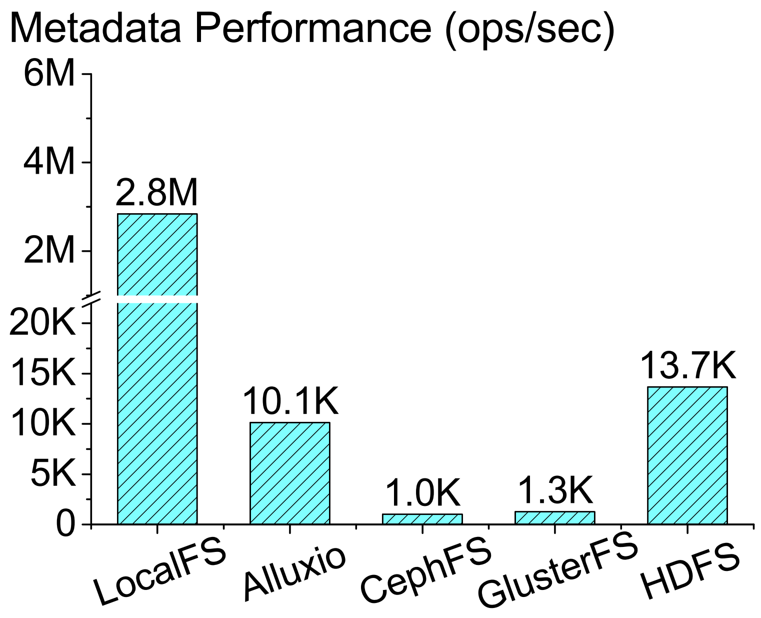


Figure 3.7: Metadata operation performance, in terms of number of operations per second. For LocalFS, the performance is measured on a single node, while for each DFS, the performance is measured on the whole cluster.

Sequential/Random Read/Write. Figure 3.8 shows the throughput results of sequential/random read/write operations. We observe that the memory-centric DFS implementations can achieve higher throughput than the disk- based ones by more than $100\times$. Specifically, *Alluxio (In-Mem)* reads and writes faster than others since all its operations are in local memory. For the write throughput, *Alluxio (In-Mem)* is about $400\times$ faster than GlusterFS and $130\times$ faster than HDFS.

In addition, the fault tolerance mechanism has significant influence on the performance. To maintain fault tolerance, CephFS, GlusterFS, and HDFS need to replicate data across the cluster, while Alluxio uses lineage and checkpoint. HDFS replicates at a block granularity, while GlusterFS needs to synchronize the whole volume due to its RAID-like mechanism, which makes GlusterFS slower than HDFS. CephFS also replicates files and has a similar write throughput to HDFS. Furthermore, HDFS outperforms CephFS and GlusterFS in terms of the read throughput due to data locality. HDFS usually stores a file with a local replica when writing. On the other hand, CephFS uses the CRUSH algorithm [**WeilBMM06**] to determine file locations, while GlusterFS uses consistent hashing; both of them do not address data locally.

Comparing *Alluxio (Through)* and HDFS we can find that *Alluxio (Through)* is a little slower than HDFS in both reading and writing. This is because *Alluxio (Through)* only wraps the APIs of HDFS and it actually stores files on HDFS. In the following experiments, *Alluxio (Through)* also has similar behaviors with HDFS. Moreover, we have already detected a performance issue in the previous version of Alluxio that *Alluxio (Through)* has a critical overhead in the *open* and *close* operations due to the redundant RPC calls when many clients access Alluxio concurrently. We

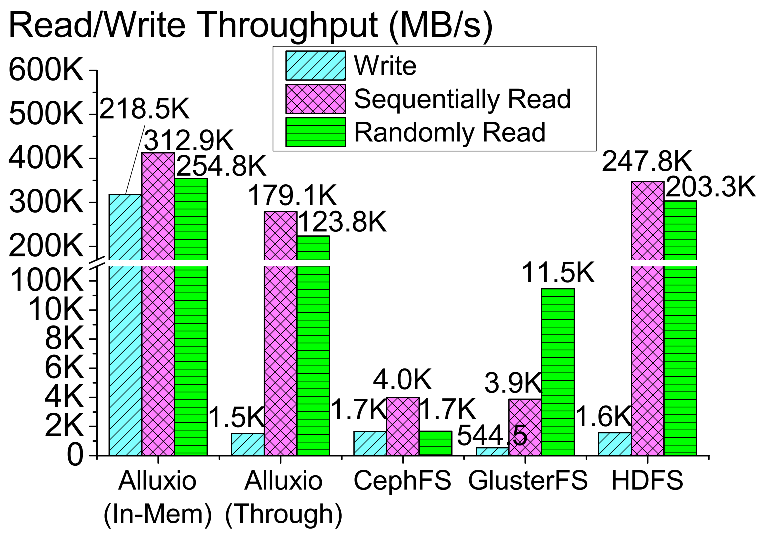


Figure 3.8: Sequential/random read/write throughput. This is the average throughput of the entire cluster.

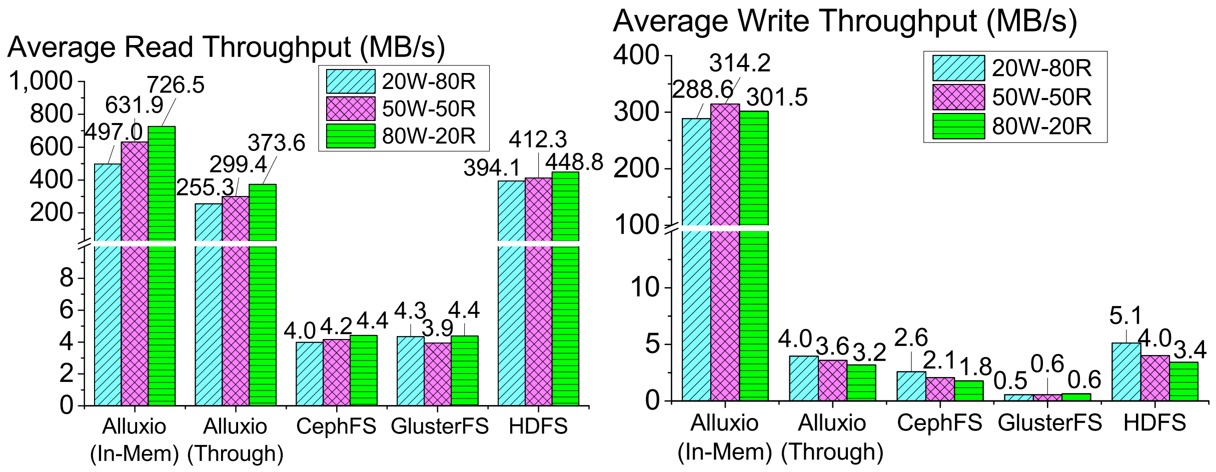
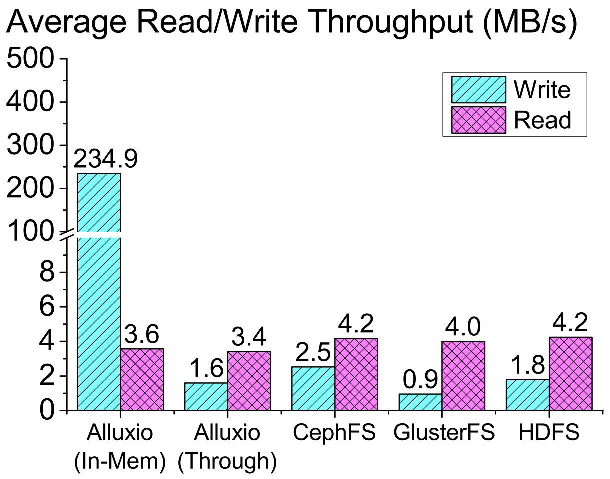


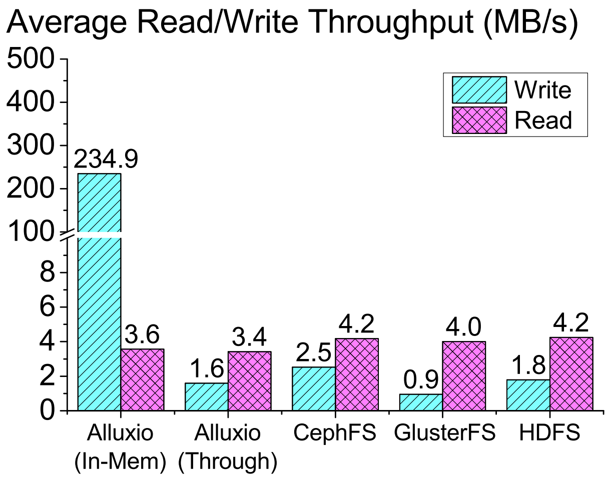
Figure 3.9: Mixed read/write throughput. This is the average throughput of each thread.

have reported it to the Alluxio developer community [128].

Another interesting point is that the random read throughput of GlusterFS is about $3\times$ of the sequential read throughput, while all other DFS implementations have lower random read throughput than the sequential one. The reason is that each GlusterFS client has an I/O cache, which allows the random reads to access cached data from memory. On the other hand, the clients of Alluxio and HDFS do not have such a caching mechanism, but instead always read from the data stores. The *cache tiering* of CephFS is a server-side cache and the network limits the random read throughput.

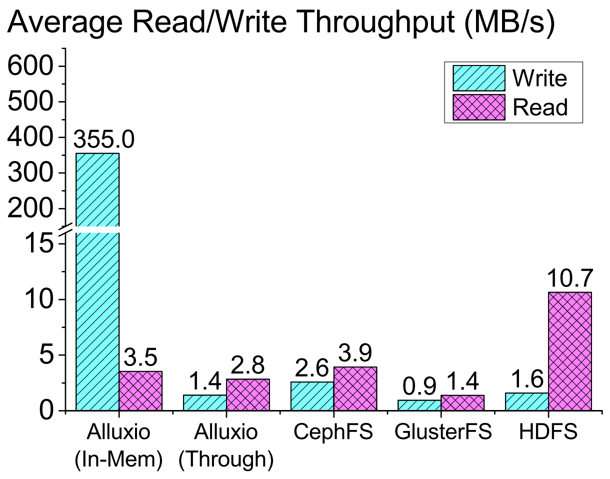


(a) Shuffle

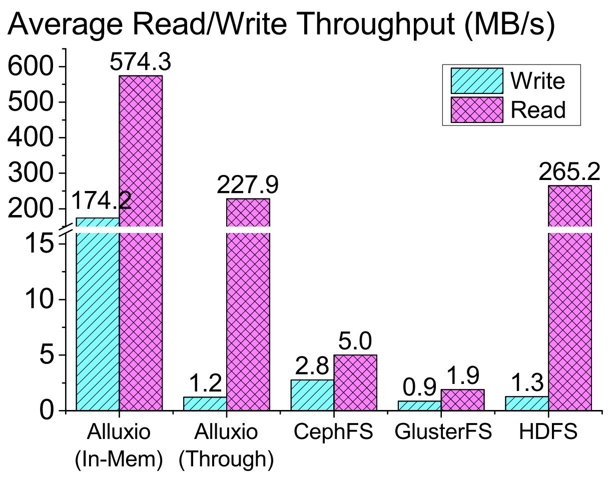


(b) Non-Shuffle

Figure 3.10: Iterative read/write throughput in (a) *Shuffle* Mode and (b) *Non-Shuffle* Mode. This is the average throughput of each thread.



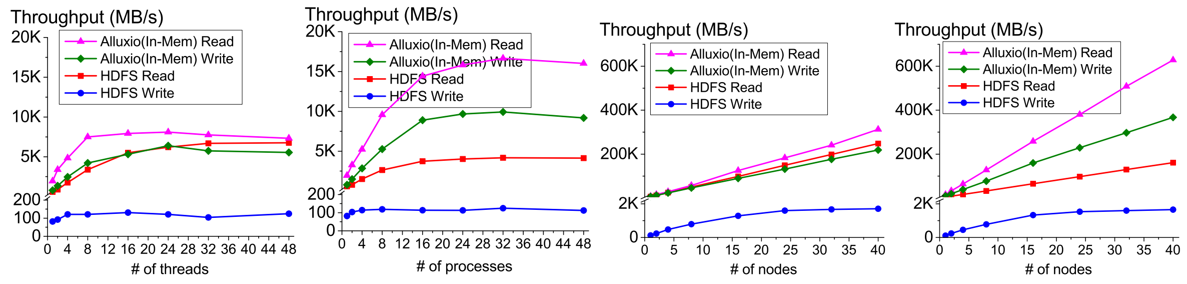
(a) Shuffle



(b) Non-Shuffle

Figure 3.11: Irregular massive access throughput in (a) *Shuffle* Mode and (b) *Non-Shuffle* Mode. This is the average throughput of each thread.

Mixed Read/Write. Figure 3.9 shows the respective read and write throughput subject to different read-to-write ratios. *Alluxio (Through)*, CephFS and HDFS show that the read throughput increases and the write throughput decreases when the write ratio rises. In GlusterFS, both the write and read operations affect the I/O cache regardless of the read-to-write ratio, so the read-to-write ratio is not the major factor to affect the throughput. Finally, since *Alluxio (In-Mem)* has no replications and all operations are in memory, the throughput is closely associated with the system memory bandwidth. In our environment, *Alluxio (In-Mem)* shows an increasing read throughput as the write ratio rises



(a) Single-Node Multi-Thread (b) Single-Node Multi-Process (c) Multi-Node Multi-Thread (d) Multi-Node Multi-Process

Figure 3.12: Scalability of DFS-Perf. In single-node we take the value of 1 process x 1 thread as the baseline and in multi-node we take the value of one node as the baseline. (a) single-node multi-thread. (b) single-node multi-process. (c) multi-node, each node is in 16 threads. (d) multi-node, each node is in 16 processes.

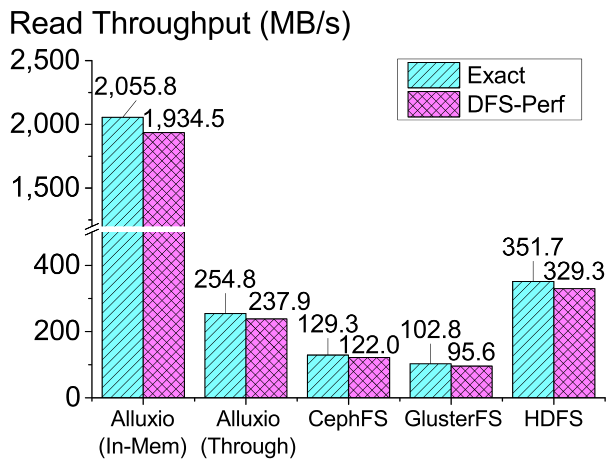
but its write throughput degrades.

Iterative Read/Write. We now consider the iterative workloads, and the performance results are shown in Figure 3.10. Compared the *Shuffle* and *Non-Shuffle* modes, data locality can lead to tens to even hundreds times of performance improvement. The read throughput of HDFS in *Non-Shuffle* mode increases by $70\times$ of the *Shuffle* mode, while this value of *Alluxio (In-Mem)* is more than $100\times$. However, for CephFS and GlusterFS, they distribute files instead of storing files locally, so their throughput is limited by the network.

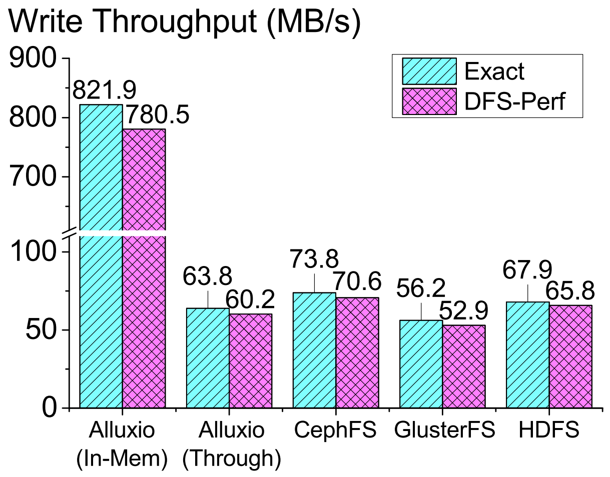
In addition, *Alluxio (In-Mem)* in the *Non-Shuffle* mode reads and writes data through local memory at the same time, which means the read and write may influence each other. This leads to a lower write throughput than in the *Shuffle* mode. However, *Alluxio (In- Mem)* still outperforms others in writes since it does not transfer data over network.

Irregular Massive Access.

Finally, Figure 3.11 shows the throughput of irregular massive access. Compared with the iterative workload results, the read throughput of HDFS in Figure 3.11a is about $2.5\times$ of that in Figure 3.10a, even they are both in *Shuffle* mode. The reason is that HDFS uses three replications by default, which increases the possibility to achieve data locality for irregular access. CephFS also uses replication, but it does not exploit data locality as it always reads from the primary OSD regardless of the location of the client. In addition, the read throughput of *Alluxio (In-Mem)* in Figure 3.11b is about 50% higher than that in Figure 3.10b, because the multi-level caches in Alluxio work better for irregular access than the sequentially iterative access.



(a) Read Comparison



(b) Write Comparison

Figure 3.13: Comparisons between Exact and DFS-Perf. (a) Read Throughput. (b) Write Throughput.

3.6.3 Scalability

In this subsection, we evaluate the scalability of DFS-Perf by comparing its performance in multi-thread and multi-process parallel modes. We choose the throughput results of *Alluxio (In-Mem)* and HDFS as the representative examples. First, we run the workloads from 1 to 48 threads or processes on a single node. Then the same workloads are run from 1 to 40 nodes, either in 24 threads or 24 processes. The speedup performance is shown in Figure 3.12.

In multi-node mode, the throughput of HDFS is limited by the network of the whole cluster, while *Alluxio (In-Mem)* achieves near-linear scalability because it accesses data from local memory. The experimental results in Figure 3.12 also indicate that DFS-Perf has good scalability. Furthermore, this experiment can also be used for performing stress tests in a scalable way to detect the bottleneck of a DFS.

Meanwhile, for the multi-process and multi-thread modes, we find that their behaviors are similar. The throughput grows with the concurrency degree increasing, until the concurrency degree reaches 24, i.e., the hyper-threading upper limit in each of our machines. When the concurrency degree goes beyond the hyper-threading upper limit, the throughput performance stays the same or even decreases a bit. In addition, HDFS performs better in the multi-thread mode (about $1.5\times$) while *Alluxio (In-Mem)* performs better in the multi-process mode (about $1.7\times$) on a single node.

3.6.4 Framework Overhead

In this subsection, we evaluate the overhead of the DFS-Perf framework. The overhead may come from the few communications between DFS-Perf Master and Slaves, and the extra statistics collecting step. And we need to know how much the overhead impacts the performance and whether it will get higher when scaling out.

For comparison, we hard-coded a lightweight tool that directly reads and writes DFS to gather the pure throughput performance metric without any impact on the benchmarking framework itself, namely *Exact* in Figure 3.13 and Figure 3.14. First, we measure the overhead of the DFS-Perf framework by comparing its performance with the pure performance. As shown in Figure 3.13, **DFS-Perf only has 5.7% (3% - 7%) difference on average**, which we believe is little overhead acceptable for the DFS-Perf framework. The negligible overhead of DFS-Perf will ensure reliable benchmarking performance results.

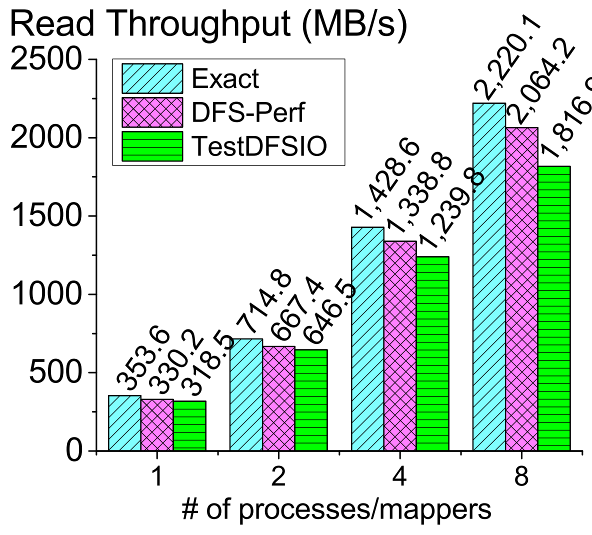
Then, we compare the overhead of DFS-Perf with TestDFSIO, a built-in test tool of HDFS. For fair comparisons, we conduct these experiments on the same DFS, a single node HDFS. Both DFS-Perf and TestDFSIO are configured with the same number of processes or mappers. Figure 3.14 reveals that DFS-Perf has less overhead and achieves more accurate results than TestDFSIO. Moreover, DFS-Perf has a stable difference (about 6.6% on read and 4.4% on write), but TestDFSIO has an increasing difference as the concurrency gets higher (from 9.9% to 12.7% on read and from 6.3% to 19.2% on write). The reason is that TestDFSIO relies on the MapReduce framework, a general parallel processing platform, in which more mappers consume more system and network resources than the DFS-Perf implementation.

3.7 Experience

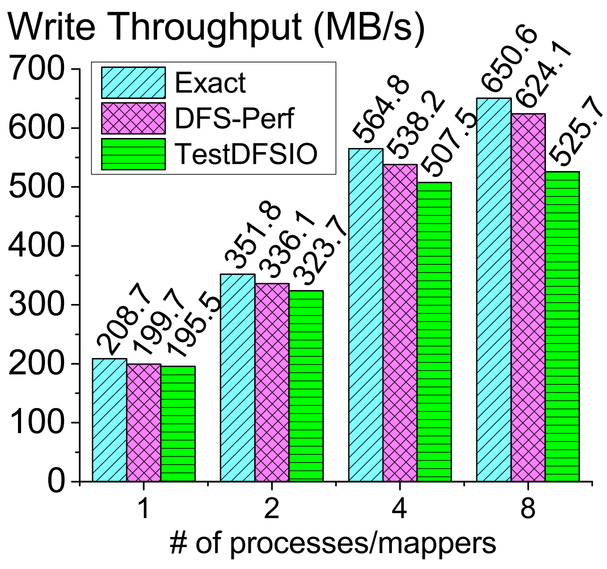
Several performance issues in realistic DFS have being caught by DFS-Perf. We have contributed some our patches to the open source community. In Alluxio, the DFS-Perf sequential read benchmark has detected that there exists critical overhead in the open and close steps when many clients access the file metadata concurrently [128]. With DFS-Perf, we also detected a memory statistics bug in Alluxio [127]. The statistics are important for monitoring and allocating the storage space in Alluxio. This bug only takes effect when multiple users read the same in-memory file concurrently. Thus, the sequential utility tools and unit tests in Alluxio can hardly capture it. Because of DFS-Perf's multi-thread and multi-process testing mechanism, we detected the bug easily. In addition, the *alluxio-perf* module in Alluxio is derived from our DFS-Perf work and now has become the official performance benchmarking tool of the Alluxio project. Besides Alluxio, we also found an issue in GlusterFS regarding the incompatibility between the implementation of the metadata synchronization mechanism and the old Linux kernel interface.

3.8 Conclusion

Big data applications often build on a distributed file system (DFS) for scalable and reliable storage management. We believe that comprehensive performance evaluation of various DFS implementations is important. This paper presented DFS-Perf, a unified benchmarking framework that can be designed for evaluating the performance of various DFS implementations. DFS-Perf is scalable and general enough to adapt to various parallel and distributed environments. Also, it is extensi-



(a) Read Comparison



(b) Write Comparison

Figure 3.14: Comparisons of Exact, DFS-Perf and TestDFSIO. (a) Read Throughput. (b) Write Throughput.

ble with an easy way to add new workloads and plug in new DFS backends. We have designed and implemented a group of representative workloads with the file access patterns summarized from the real-world applications. Moreover, DFS-Perf includes an extensible workload generator that enables users to customize specific workloads automatically. We have conducted extensive testbed experiments to evaluate state-of-the-art DFS implementations using DFS-Perf. Based on the evaluations, we discussed the critical factors that impact the performance of DFS. The experimental results also demonstrate that DFS-Perf has good scalability and negligible overhead (5.7% on average). In future work, we plan to plug in more workloads and DFS backends into DFS-Perf, especially those workloads that can represent other categories of big data applications, e.g., graph processing and key-value stores.

Chapter 4

System Design and Implementation

4.1 Overview

We have designed and implemented Alluxio[1], the first VDFS in the world. It sits between computation frameworks [39] such as Apache Spark [146], Presto [110], Tensorflow [92], and Apache MapReduce [16] and persistent data stores including Amazon S3 [6], Apache HDFS [121], EMC ECS [54], and Ceph [137].

Alluxio (Figure 4.1) presents a set of disparate data stores as a single file system, greatly reducing the complexity of storage APIs, locations, and semantics exposed to applications. For example, an object storage in the cloud and an on-premise distributed file system both appear to applications as different portions of the Alluxio file system namespace.

Alluxio is designed with a memory centric architecture. It enables applications to leverage memory speed I/O by simply using Alluxio. Because Alluxio is between the compute and storage layers, managing compute cluster storage media, it also serves as a transparent cache that enables applications to benefit from data sharing and locality. By being between applications and data stores, Alluxio enables rapid innovation in both the compute and storage layers, because Alluxio abstracts basic functionality and compatibility.

Since open-sourced in April 2013, Alluxio became one of the fastest growing open source projects in the data ecosystem. The Alluxio open source community has over 800 contributors from over 200 organizations, as shown in Figure 4.2.

Alluxio is deployed in production at hundreds of companies and organizations globally, supporting businesses across sectors, such as Financial Services, Technology Sector, Internet Industry, e-Commerce and Retail, Telecommunication, and more, at scales up of to thousands of servers [45].

[1]https://www.alluxio.org/

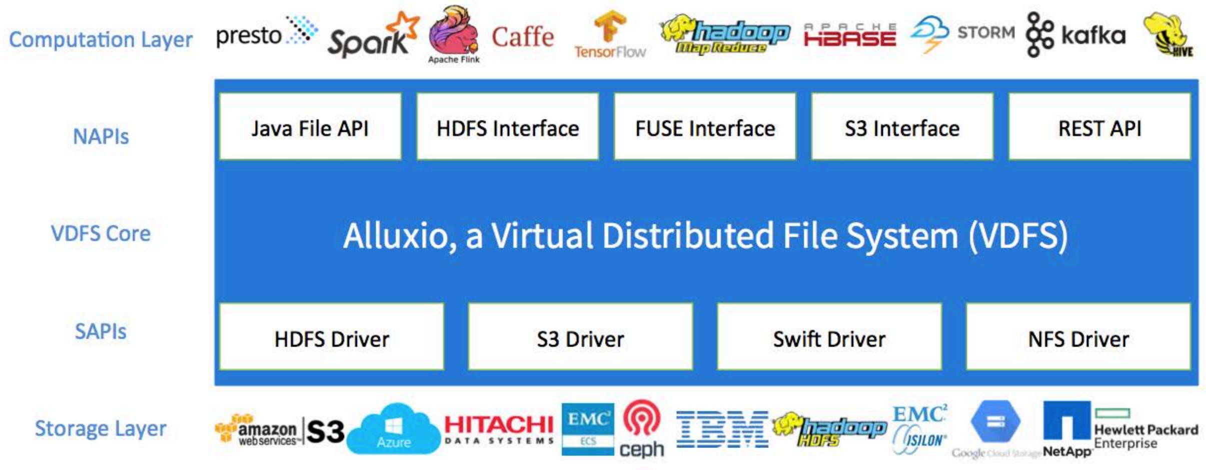


Figure 4.1: Alluxio, a VDFS implementation

4.2 Architecture

Alluxio[2] has a master-worker architecture. Master nodes are managing Alluxio system's metadata, and being responsible for handling metadata requests and cluster registration. Workers are managing each of their deployed servers local storage media, being responsible for handling data requests, and transferring data from Alluxio storage or other data stores to clients. Applications use the Alluxio Client library to interact with Alluxio. Alluxio's the under storage is comprised of different data stores which house the persistent copies of the data.

4.2.1 Masters

The Alluxio master is a quorum of nodes (possibly only one) that manage the state of the Alluxio file system and cluster. Only one master node is actively serving requests at a time; the other master nodes participate in keeping the journal up-to-date as well as act as hot failovers if the active master crashes. Quorum membership and leader election are managed through Apache Zookeeper[25].

The Alluxio Master manages worker membership, registering a unique ID for each worker. Each Alluxio worker is expected to only associate with one Alluxio master. Alluxio masters also assign unique IDs to workers and keep track of the data available on each worker.

The Alluxio Master records the metadata for data blocks which make up files. Blocks are the smallest unit of data in Alluxio, usually ranging from tens to hundreds of megabytes. Each block has a unique ID but can be stored on zero, one, or many worker nodes.

One or more blocks comprise the data portion of a file. The master also keeps track of the file system metadata. It maps a human readable path to a specific file. Along with path information,

[2]Based on Alluxio open sourceo version 1.7.1

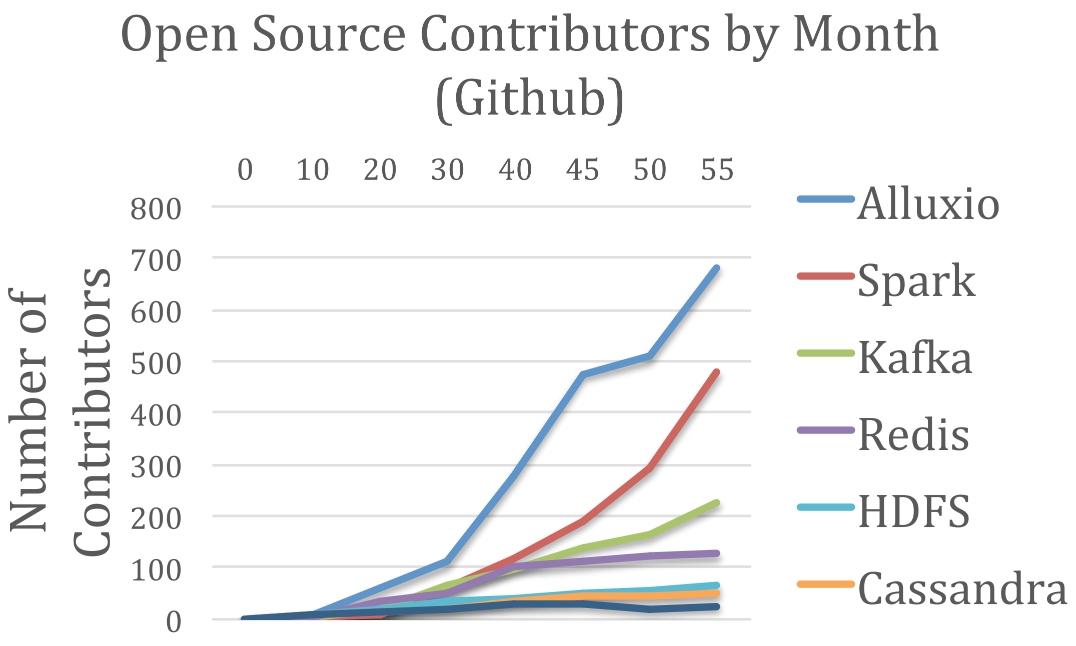


Figure 4.2: Alluxio is one of the fastest growing open-source projects in the data ecosystem

the Alluxio master keeps various file level metadata such as permissions and modification time.

The Alluxio master allows clients and workers to freely join or leave the cluster. Clients typically communicate with the master to make logical updates to the metadata. These requests are independent and stateless, enabling the master to scale up to tens of thousands of concurrent clients. Workers have a periodic heartbeat with the master for as long as they part of the cluster. The heartbeat is low cost and Alluxio clusters have scaled to thousands of nodes in production environments.

The master is also responsible for keeping a journal of events for fault tolerance. The journal is typically stored on distributed storage to avoid a single point of failure, like losing a disk for example. Protocol Buffers are used to serialize journal entries with the added benefit of being easily extended to support future use cases.

Typical deployments have two or more master nodes for master fault tolerance and high availability. Currently, the number of files the Alluxio master supports is proportional to the memory allocated to it. Similarly, allocating more CPUs to the master allows for lower response latency in highly concurrent environments.

4.2.2 Workers

An Alluxio Worker demon is deployed on every server in an Alluxio cluster. The Worker is responsible for managing local storage media and providing data access to the data stores, which Alluxio

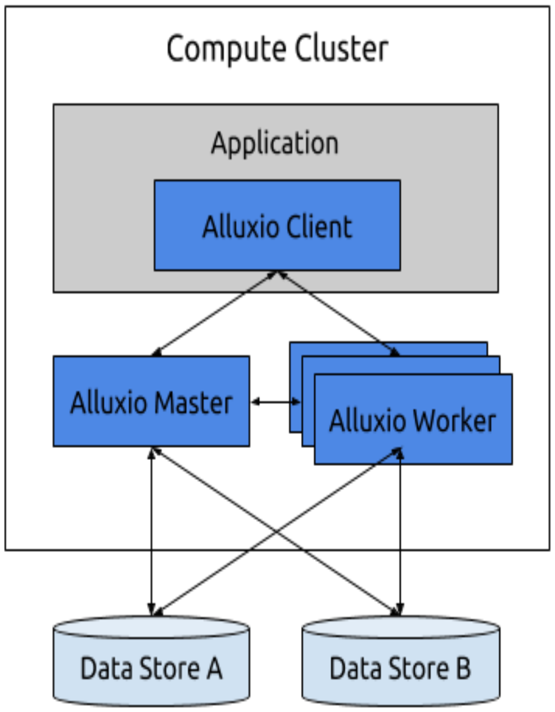


Figure 4.3: Alluxio system components.

abstracts to applications. The Alluxio worker process is lightweight and does not rely on persistent data; the system is designed to tolerate the complete loss of a worker and its local storage.

Deploying the worker process co-located with applications greatly improves the maximum possible read/write throughput to Alluxio by providing data locality. Alluxio has optimizations to allow clients to read and write at memory speed if the client is running on the same node. In workloads that require a large amount of I/O, reducing the network traffic and serving data locally significantly improves performance.

Alluxio workers can be deployed on dedicated nodes as well. This has the benefit of isolating workers, which is helpful in cases where the application nodes are ephemeral or not committed to the cluster. For example, a single set of Alluxio worker nodes can serve several isolated compute clusters belonging to different users. In this way, the compute clusters do not need to be concerned about having any resources taken by Alluxio due to the workload of other clusters.

The primary function of Alluxio workers is to serve data requests from clients. It can do so in one of two ways, either directly from Alluxio local storage or by fetching data from an underlying data store, for example Amazon S3. Data fetched from an underlying data store can be cached in Alluxio local storage for subsequent accesses, based on the policies such as LRU, LFU.

4.2.3 Clients

The Alluxio client is a library that is used by applications to interact with the Alluxio servers. Client library provides several interfaces, including an Alluxio File interface, a Hadoop compatible file system interface, an S3 compatible object store interface, a FUSE interface, and a REST interface. Each interface may have more than one language binding, such as C, Java, Python, Go. Alluxio supports many interfaces to give application developers the most flexibility when working with Alluxio.

- **Hadoop Compatible Filesystem API**: We implemented a Hadoop Compatible Filesystem API to enable existing applications in the big data ecosystem, e.g. Spark, Presto, MapReduce, Hive, etc., to run on top of Alluxio without any code changes.
- **S3 / Swift Compatible API**: By offering an S3 / Swift API, it enables any cloud based applications to interact with data through Alluxio.
- **Filesystem in Userspace (FUSE) API**: By offering a FUSE API, any legacy applications interacting with the local file system can work with Alluxio and interact with data from any storage system, which is integrated into the Alluxio ecosystem.
- **Filesystem REST API**: In addition to the APIs above, by offering a Filesystem REST API, applications written in any language can interact with Alluxio through web-based REST API. The REST API enables many more applications to be able to share data with each other.

More interfaces and language bindings are being added to enlarge Alluxio supported ecosystems and workloads.

4.3 Use Cases

In this section, we introduce several Alluxiocommon use cases.

4.3.1 Enabling Decoupled Compute and Storage

The industry it moving towards the architecture of decoupling compute and storage. This architecture allows improved allocation and utilization of storage and compute resources, brings considerable cost savings. There are two major challenges in this architecture, the performance impact due to lower bandwidth and higher latency and the limitations of creating or modifying an application against a particular storage interface.

Alluxio remedies the performance impact of having remote storage, by managing computation cluster's storage media and providing hot data locality to the applications. Organizations from various industry sectors, including some of the largest Internet companies in the world [27] have

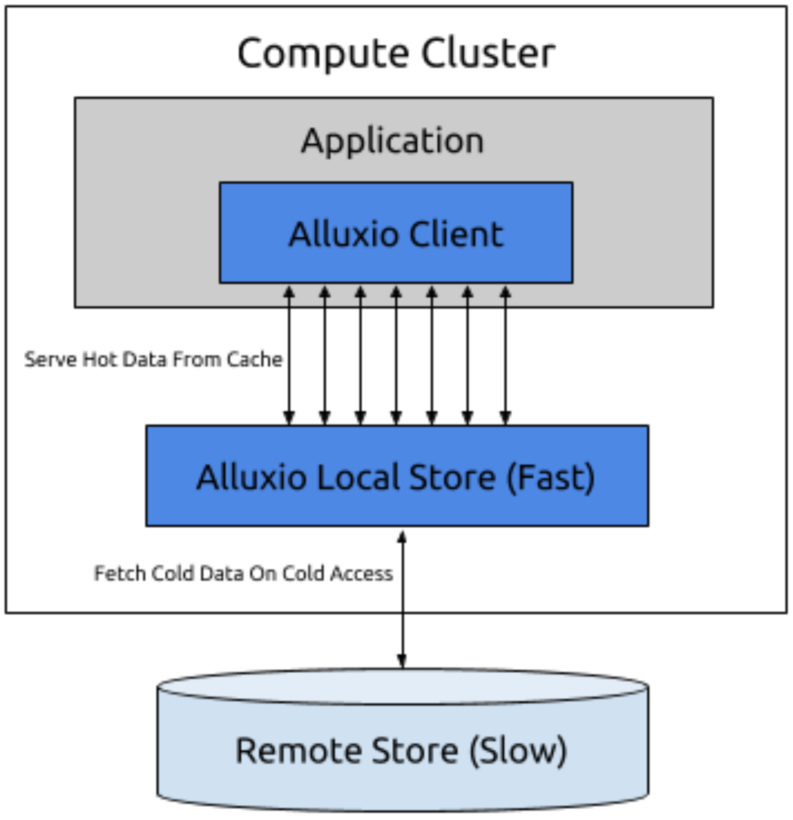

Compute Cluster
Application
Alluxio Client
Serve Hot Data From Cache
Alluxio Local Store (Fast)
Fetch Cold Data On Cold Access
Remote Store (Slow)

Figure 4.4: Alluxio improves I/O performance by levering compute cluster local storage media and policies to provide data locality

used Alluxio to great effect in this manner (see Figure 4.4 below for a visual representation of the architecture).

Alluxio also solves the interface limitation problem by allowing applications to choose from a rich set of standard APIs and connect seamlessly to any data store integrated with Alluxio. This greatly improves the flexibility of application and storage teams. Many companies, including the largest geospatial data distributor [26] use Alluxio to logically decouple compute and storage to enable rapid innovation in the storage and application layers.

4.3.2 Accelerating Data Analytics

Faster time to insight for organizations translates to better-informed decisions, which provides a crucial competitive advantage. The Alluxio memory centric architecture enables top travel booking companies [113] to quickly and accurately analyze user behavior. This analysis then leads to better recommendations and user targeting.

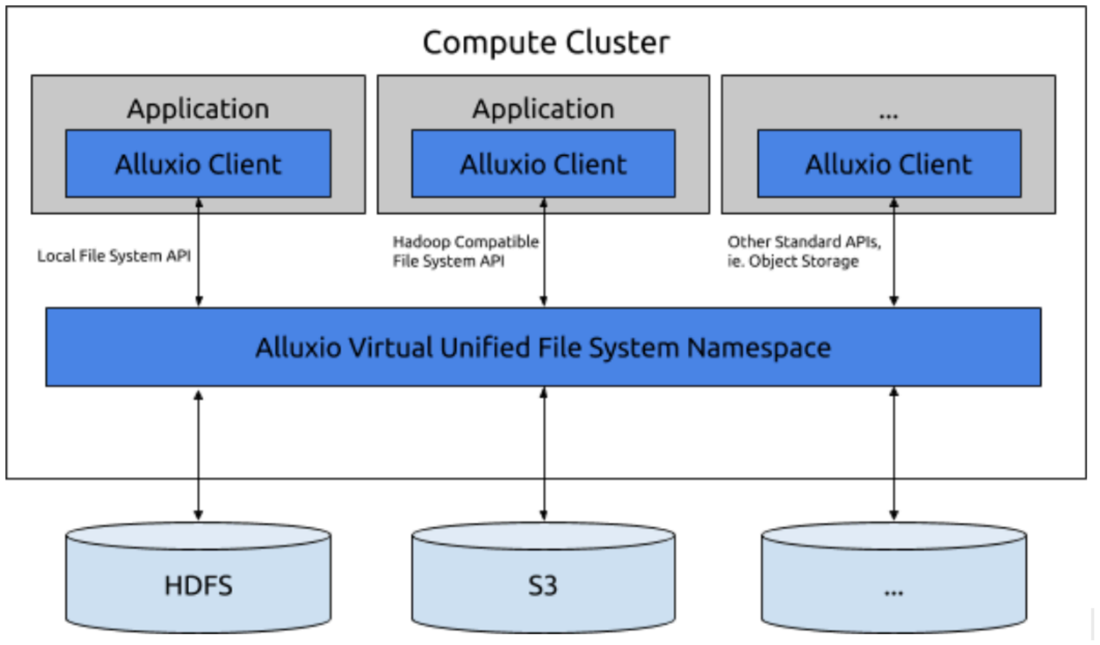

Compute Cluster
Application
Application
...
Alluxio Client
Alluxio Client
Alluxio Client
Local File System API
Hadoop Compatible File System API
Other Standard APIs, ie. Object Storage
Alluxio Virtual Unified File System Namespace
HDFS
S3
...

Figure 4.5: Alluxio provides standard APIs to applications and abstracts disparate data stores in a single unified namespace.

In addition to accelerating I/O, Alluxio enables previously isolated computation to benefit from the data access patterns of other applications. The ability to share data between applications has led to even further performance gains and less data movement. Figure 4.4 shows this for a single application, but multiple applications can utilize the same Alluxio local store.

Machine learning, especially deep learning, is reliant on the available training data sets. Often, it is not the algorithm but the training and validation data that produces the best models. Machine learning workloads have mostly been written with local data in mind, whereas the data necessary to fuel the models are stored in scattered data repositories. By using Alluxio, AI focused companies [87] can easily expand the datasets available for training and validation to all the data repositories owned by the organization (see Figure 4.5). This new architecture improved users' business agility comparing with traditional approaches, which required complex ETL and data tracking to ensure the cleanliness and availability of datasets.

4.3.3 Replacing ETL with Unification

Extraction, transform, load (ETL) jobs are costly and difficult to maintain. Data must be copied from a master source and any updates must be propagated to the copies to ensure data integrity. Before Alluxio, applications or groups typically copy separate version of the data, leading to more complex storage environment, and increasing maintenance requirements and cost.

Alluxio provides applications with data virtualization and unification as shown in Figure 4.5. Organizations such as top financial companies [90] benefit from the huge improvement in manageability and performance gained by using Alluxio as opposed to ETL processes.

Data virtualization allows ETL to be transparent, both in terms of any effort required to obtain the data as well as removing the need for users to manage copies of the data. With Alluxio, the data in Alluxio are automatically synchronized with the underlying storage in the case of updates or other changes when a client accesses them.

Alluxio also provides data unification, enabling applications to access various datasets across different storages, without needing to understand where the source data is located. This is especially helpful as organizations try to derive more insights from different data repositories.

4.4 Examples

In this section, we present different use case examples that leverage the key features of Alluxio to provide data accessibility, performance improvement, and data management, to improve business agility and reduce the IT cost of the users.

4.4.1 Accelerating Data Analytics on Object Storage

Challenges As the volume of data collected by enterprises has grown, there is a continual need to find efficient storage solutions. Owing to its simplicity, scalability and cost-efficiency object storage, including Ceph, has increasingly become a popular alternative to traditional file systems. In most cases the object storage system, on-premise or in the cloud, is decoupled from compute nodes where analytics is run. There are several benefits of this separation.

- Improved cost efficiency - Storage capacity and compute power can be provisioned independently. This simplifies capacity planning and ensures better resource utilization.
- Ease of manageability - A separation of data from compute means that a single storage platform can be shared by different compute clusters. For example, a cluster hosting long-running services emitting data into object storage may run in conjunction with a data processing cluster to derive insights.

However, a consequence of this architecture is that data is remote to the compute nodes. When running analytics directly on the object store, data is repeatedly fetched from the storage nodes leading to reduced performance. This delay may prevent critical insights from being extracted in a timely manner.

The Solution This is addressed by deploying Alluxio on compute nodes, allowing fast ad-hoc analysis of data by bringing performance up to memory speeds with intelligent storage of active data close to computation.

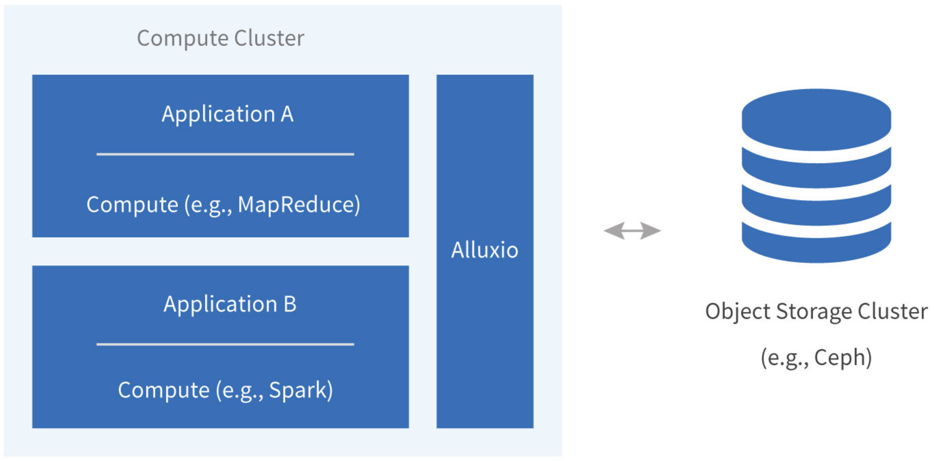


Figure 4.6: Alluxio with Object Store Cluster

The Result We present the results of a comparison between using Alluxio and directly using Ceph in Figure 4.7. Note that in both cases Ceph is accessed using the REST object storage API. The comparison is done with the default configuration for both stacks with one exception. The Swift FS client is configured to use a 512MB block size to match Alluxios default block size instead of its default of 32MB. The larger block size improves I/O bandwidth when using the Swift FS client and improves performance by about 30%. The result presented incorporates this gain. Tuning other parameters may provide performance gain in certain cases for both stacks, but would be difficult to generalize to all workloads.

4.4.2 Analytics on Data from Multiple Locations and Eliminating ETL

Many enterprises have invested extensively in global information technology infrastructure, including multiple data centers collecting petabytes of data. Analyzing data located in multiple data centers worldwide is critical for businesses to understand and improve the usability and reliability of their products. With Alluxio, companies unified data from multiple data centers and eliminated the ETL process while lowering storage cost due to multiple data copies.

Challenges To analyze data resides in different data centers to create a global and comprehensive view of the business, companies typically use time-consuming and error-prone ETL process to transfer the data from multiple locations to a single data center for analysis (See Figure 4.8). For example, one company uses data technologies such as HDFS to store the data and Hive metastore to store the metadata associated with the structured data. Analytics is performed using Hive and Spark SQL to gain insight into user behavior, popular applications, log analysis and more. The volume of data and number of geographic locations presented multiple challenges:

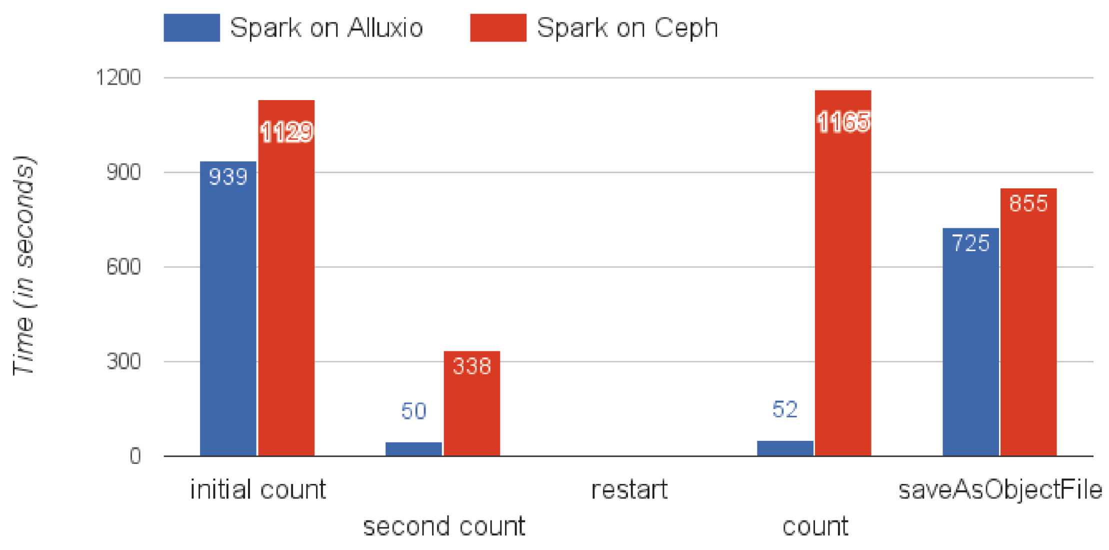


Figure 4.7: Performance comparison of accessing Ceph storage with and without Alluxio.

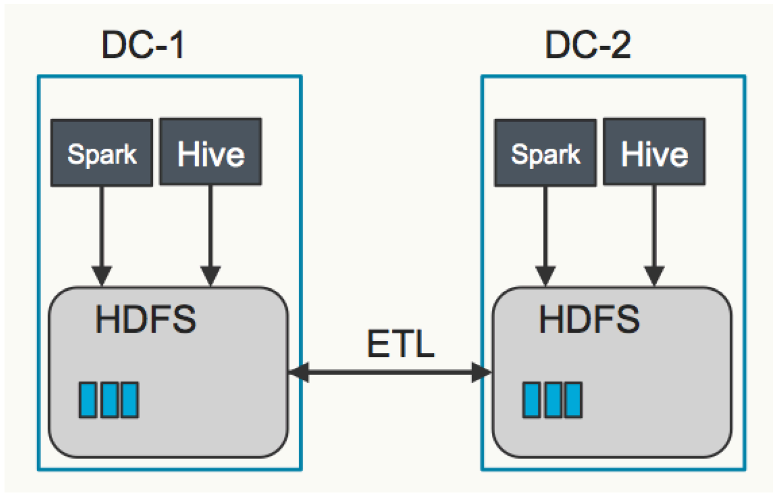


Figure 4.8: Multi Data Centers Infrastructure without Alluxio: ETL processes are required to migrate data between two data centers

- High storage cost due to duplication of data
- Bandwidth and performance limitations transferring data from multiple locations
- Regulations preventing the transfer of certain data and excluding it from analysis

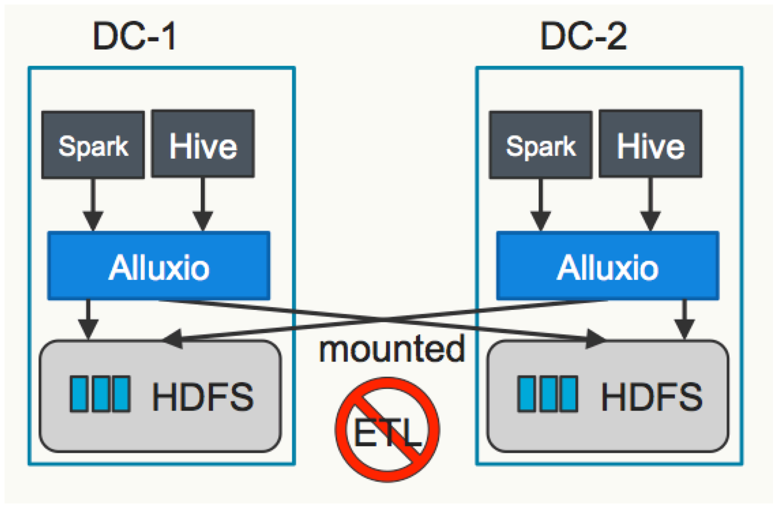


Figure 4.9: Multi Data Centers Infrastructure with Alluxio: ETL processes are no longer required and Alluxio is deployed to provide abstraction across data centers

The Solution Companies addressed the technical challenges by using Alluxio as the data unification and management layer for all data stored worldwide. Various data stores were connected to Alluxio, providing seamless access for multiple applications through the global namespace. No changes to the application were required. With this architecture, companies perform advanced analytics involving cross data center synchronization, joins and unions. Alluxio also temporarily stores the data in memory accelerating performance. Alluxio fits within existing security frameworks and enforces the policies in place, ensuring regulatory and compliance requirements from different countries and jurisdictions are met. Figure 4.9 shows the infrastructure with Alluxio.

The Results With this new architecture, companies now have the infrastructure that allows them to analyze their worldwide data without the need for error prone, time consuming and costly ETL or the need for data duplication. Alluxio maintains the latest copy of the data in memory, or fetches it from HDFS for new requests, so data freshness is assured. With Alluxio, the platform stores data locally in memory from remote HDFS locations and provides transparent access for analytics applications. Alluxio presents the same API to the existing applications, and this enabled the user to achieve the benefits without disrupting the existing stack or changing applications.

With Alluxio integrated in the data processing stack, companies are now able to access and transform massive amounts of data into valuable insights. This meets the business objectives of improved product quality and customer satisfaction at the lowest possible cost for their analytics platform.

Chapter 5

Conclusion

This dissertation proposes a new architecture with a Virtual Distributed File System (VDFS) as a new layer between the compute layer (such as Apache Spark [146], Presto [110], Tensorflow [92], and Apache MapReduce [16]) and the storage layer (such as Amazon S3 [6], Apache HDFS [121], EMC ECS [54], and Ceph [137]). Adding VDFS into the data processing stack brings many benefits. Specifically, VDFS enables unified data access for different compute frameworks, efficient in-memory data sharing and management across applications, high I/O performance and efficient use of network bandwidth, and the flexible choice of compute and storage. Meanwhile, as the single layer to access data and collect data metrics and usage patterns, it also provides users insight into their data, which can also be used to optimize the data access based on workloads.

We have realized a VDFS implementation in Alluxio. Alluxio presents a set of disparate data stores as a single file system, greatly reducing the complexity of storage APIs, locations, and semantics exposed to applications. Alluxio is designed with a memory centric architecture, enabling applications to leverage memory speed I/O.

5.1 Broader Impact

We released Alluxio as an open source project in April 2013. With five years of open source history, Alluxio Open Source project has attracted more than 800 contributors from over 200 institutions globally, including Alibaba, Baidu, CMU, JD.com, Google, IBM, Intel, NJU, Red Hat, Samsung, Tencent, UC Berkeley, and Yahoo. In industry, Alluxio is deployed in production by hundreds of organizations, which includes seven of the ten largest Internet companies worldwide. Many clusters running Alluxio have exceeded 1,000 nodes. In academia, researchers have used Alluxio open source project as a research vehicle. For more information, please visit Alluxio powered-by page hosted at `https://www.alluxio.org`.

5.2 Challenges and Future Work

One important direction of the future work is to extend Alluxio's ecosystem. Many of these items are very challenging.

- **New Application Interfaces** Alluxio currently provides a rich set of file based APIs to access data backed by different stores. As more applications use Alluxio, more sophisticated interfaces will be needed to optimize performance and ease of use. For example, a large number of applications would prefer APIs to access structured data.
- **New Data Store Integrations** Alluxio currently abstracts data stores using file system (e.g., local filesystem, HDFS, NFS and etc) and object store (e.g., AWS S3, GCS, Azure blob store, Alibaba OSS, Ceph and etc) APIs. As users may want to access more types of data and data stores through Alluxio as the access layer, we will need higher level abstractions, such as key-value stores, dataframes, and databases.
- **NAPIs and SAPIs Translation** The ability to efficiently translate the interfaces between south bound and north bound becomes increasingly challenging as the number of interfaces supported increases. These challenges include the translation of the interfaces' semantics, e.g. security.
- **Workload-based Performance Optimizations** Because Alluxio sits in a unique position to collect data metrics and usage patterns across users and applications, it can provide users insight into their data and can also do optimizations in the background. For example, given the data access patterns, Alluxio can prefetch data from under stores into Alluxio space to further improve the performance of cold reads. Also for certain workloads (e.g., SQL workloads), Alluxio can choose better data replication and movement policies to optimize for the locality and performance.
- **Data Compression/Encryption** Besides providing data accessibility, Alluxio can also compress data to better utilize memory capacity. For applications with certain security requirement, Alluxio can also encrypt data before persisting to under stores.
- **New Language Bindings** Alluxio exposes Java APIs natively to applications. In addition to Java, it also supports Python, Golang and REST APIs through a Proxy service. Meanwhile, there is demand from the community to port the client to various popular programming languages like C, C++, Python, Go, Ruby, and Javascript.
- **Non-volatile Memory** Alluxio is designed with a memory centric architecture and leverages DRAM capacity actively. As non-volatile memory emerges as the replacement of DRAM with lower cost and comparable performance, Alluxio can help manage and utilize the resource of non-volatile memory across the compute cluster and make the resource available in a unified manner.

Besides extending Alluxio's ecosystem, future work includes improving the following aspects to improve the user experience and better benefit further future research:

- **System Diagnosis** Debugging a distributed system is complicated because multiple system components and other factors can be involved in a single client request, and issues are often complicated to reproduce and explain. Our experience suggests that making Alluxio easier to debug always provides huge value and reduces the effort to develop any applications. For example, we constantly improve the logging systems, CLI tools, and web interfaces to ease the diagnosis process.

- **Performance Optimizations** One common question on the Alluxio user mailing list is about performance. The performance of distributed applications on Alluxio are hard to tune. Due to the distributed nature, the end-to-end performance can be heavily affected or changed given the cluster resource dynamics. It will be valuable to create tools to derive more insights from these systems to find and understand the performance bottlenecks.

We hope that the community will tackle some of these challenges and open new research directions in distributed systems.

Bibliography

[1] Nitin Agrawal et al. "A five-year study of file-system metadata". In: *ACM Transactions on Storage (TOS)* 3.3 (2007), p. 9.

[2] *Alibaba Object Storage Service (OSS)*. https://www.dellemc.com/en-us/storage/ecs/index.htm.

[3] *Alluxio Community Website*. http://www.alluxio.org.

[4] Lorenzo Alvisi, Karan Bhatia, and Keith Marzullo. "Causality tracking in causal message-logging protocols". In: *Distributed Computing* 15.1 (2002), pp. 1–15.

[5] Lorenzo Alvisi and Keith Marzullo. "Message logging: Pessimistic, optimistic, causal, and optimal". In: *Software Engineering, IEEE Transactions on* 24.2 (1998), pp. 149–159.

[6] *Amazon Simple Storage System*. https://aws.amazon.com/s3.

[7] *AMPLab Big Data Benchmark*. https://amplab.cs.berkeley.edu/benchmark/.

[8] Ganesh Ananthanarayanan et al. "Disk-Locality in Datacenter Computing Considered Irrelevant". In: *USENIX HotOS 2011*.

[9] Ganesh Ananthanarayanan et al. "PACMan: Coordinated Memory Caching for Parallel Jobs". In: *NSDI 2012*.

[10] David G Andersen et al. "FAWN: A fast array of wimpy nodes". In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. ACM. 2009, pp. 1–14.

[11] *Apache Cassandra*. http://cassandra.apache.org/.

[12] *Apache Crunch*. http://crunch.apache.org/.

[13] *Apache Drill*. https://drill.apache.org/.

[14] *Apache Flink*. http://flink.apache.org.

[15] *Apache Hadoop*. http://hadoop.apache.org/.

[16] *Apache Hadoop MapReduce*. http://hadoop.apache.org.

[17] *Apache HBase*. http://hbase.apache.org/.

[18] *Apache HBase*. http://hbase.apache.org/.

[19] *Apache HDFS*. http://hadoop.apache.org/.

[20] *Apache Hive*. https://hive.apache.org/.

[21] *Apache Kafka*. http://kafka.apache.org.

[22] *Apache Mahout*. http://mahout.apache.org/.

[23] *Apache Mahout*. http://mahout.apache.org/.

[24] *Apache Spark*. https://spark.apache.org/.

[25] *Apache Zookeeper*. https://zookeeper.apache.org.

[26] *ArcGIS and Alluxio (2017) ESRI*. https://alluxio.com/resources/arcgis-and-alluxio-using-alluxio-to-enhance-arcgis-data-capability-and-get-faster-insights-from-all-your-data.

[27] *Baidu Queries Data 30 Times Faster with Alluxio (2016) Alluxio, Baidu*. http://alluxio-com-site-prod.s3.amazonaws.com/resource/media/Baidu-Case-Study.pdf.

[28] Jason Baker et al. "Megastore: Providing Scalable, Highly Available Storage for Interactive Services." In: *CIDR*. Vol. 11. 2011, pp. 223–234.

[29] John Bent et al. "Explicit Control in the Batch-Aware Distributed File System". In: *NSDI*. Vol. 4. 2004, pp. 365–378.

[30] MKABV Bittorf et al. "Impala: A modern, open-source SQL engine for Hadoop". In: *Proceedings of the 7th Biennial Conference on Innovative Data Systems Research*. 2015.

[31] Robert Bodnarchuk and Richard Bunt. "A synthetic workload model for a distributed system file server". In: *Proceedings of ACM SIGMETRICS*. 1991.

[32] Rajendra Bose and James Frew. "Lineage Retrieval for Scientific Data Processing A Survey". In: *ACM Computing Surveys 2005*.

[33] Peter J Braam and Rumi Zahir. *Lustre: A scalable, high performance file system*. 2002.

[34] Eric Brewer et al. *Disks for data centers (White paper for FAST 2016)*. https://research.google.com/pubs/pub44830.html.

[35] Brendan Burns et al. "Borg, omega, and kubernetes". In: *Queue* 14.1 (2016), p. 10.

[36] Brad Calder et al. "Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP '11. Cascais, Portugal, 2011, pp. 143–157. ISBN: 978-1-4503-0977-6.

[37] Paris Carbone et al. "Apache flink: Stream and batch processing in a single engine". In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36.4 (2015).

[38] Craig Chambers et al. "FlumeJava: easy, efficient data-parallel pipelines". In: *PLDI 2010*.

[39] Parth Chandarana and M Vijayalakshmi. “Big data analytics frameworks”. In: *Circuits, Systems, Communication and Information Technology Applications (CSCITA), 2014 international conference on*. IEEE. 2014, pp. 430–434.

[40] Fay Chang et al. “Bigtable: A Distributed Storage System for Structured Data”. In: *ACM Trans. Comput. Syst.* 26.2 (June 2008), 4:1–4:26. ISSN: 0734-2071.

[41] Yanpei Chen, Sara Alspaugh, and Randy Katz. “Interactive analytical processing in big data systems: A cross-industry study of MapReduce workloads”. In: *Proceedings of the VLDB Endowment* 5.12 (2012), pp. 1802–1813.

[42] Yanpei Chen, Sara Alspaugh, and Randy H. Katz. “Interactive Analytical Processing in Big Data Systems: A Cross-Industry Study of MapReduce Workloads”. In: *PVLDB* 5.12 (2012), pp. 1802–1813.

[43] James Cheney, Laura Chiticariu, and Wang Chiew Tan. “Provenance in Databases: Why, How, and Where”. In: *Foundations and Trends in Databases 2007*.

[44] Mosharaf Chowdhury, Srikanth Kandula, and Ion Stoica. “Leveraging endpoint flexibility in data-intensive clusters”. In: *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*. ACM. 2013, pp. 231–242.

[45] *Companies and organizations powered by Alluxio*. https://www.alluxio.org/community/powered-by-alluxio.

[46] Brian F. Cooper et al. “Benchmarking cloud serving systems with YCSB”. In: *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*. 2010, pp. 143–154.

[47] Brian F. Cooper et al. “PNUTS: Yahoo!’s hosted data serving platform”. In: *PVLDB* 1.2 (2008), pp. 1277–1288.

[48] Aaron Davidson and Andrew Or. “Optimizing shuffle performance in spark”. In: *University of California, Berkeley-Department of Electrical Engineering and Computer Sciences, Tech. Rep* (2013).

[49] *DCOS*. https://dcos.io/.

[50] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters”. In: *6th Symposium on Operating System Design and Implementation OSDI 2004, San Francisco, California, USA, December 6-8, 2004*. 2004, pp. 137–150.

[51] Giuseppe DeCandia et al. “Dynamo: Amazon’s Highly Available Key-Value Store”. In: *Proc. of SOSP*. Stevenson, Washington, USA, 2007. ISBN: 978-1-59593-591-5.

[52] Giuseppe DeCandia et al. “Dynamo: amazon’s highly available key-value store”. In: *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*. 2007, pp. 205–220.

[53] *Dell*. http://www.dell.com/us/business/p/servers.

[54] *Dell EMC ECS*. https://www.dellemc.com/en-us/storage/ecs/index.htm.

[55] Phillip M. Dickens and Jeremy Logan. "Y-lib: a user level library to increase the performance of MPI-IO in a lustre file system environment". In: *Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing, HPDC 2009, Garching, Germany, June 11-13, 2009*. 2009, pp. 31–38.

[56] Jaliya Ekanayake et al. "Twister: A Runtime for Iterative MapReduce". In: *HPDC 2010*.

[57] E.N. Elnozahy, D.B. Johnson, and W. Zwaenepoel. "The Performance of Consistent Checkpointing". In: *11th Symposium on Reliable Distributed Systems 1994*.

[58] Robert Escriva, Bernard Wong, and Emin Gün Sirer. "HyperDex: A distributed, searchable key-value store". In: *ACM SIGCOMM Computer Communication Review* 42.4 (2012), pp. 25–36.

[59] Sanjay Ghemawat and Jeff Dean. "LevelDB". In: *URL: https://github.com/google/leveldb, http://leveldb.org* (2011).

[60] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. "The Google File System". In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. 2003.

[61] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. "The Google file system". In: *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*. 2003, pp. 29–43.

[62] Ali Ghodsi et al. "Dominant Resource Fairness: Fair Allocation of Multiple Resource Types." In: *Nsdi*. Vol. 11. 2011. 2011, pp. 24–24.

[63] Garth A. Gibson et al. "File Server Scaling with Network-Attached Secure Disks". In: *Proceedings of ACM SIGMETRICS*. 1997.

[64] Gluster. *An Introduction to Gluster Architecture*. White Paper. 2011.

[65] *GlusterFS*. http://www.gluster.org/.

[66] *Google Cloud Storage*. https://cloud.google.com/storage.

[67] *GPFS file system*. http://www-03.ibm.com/systems/software/gpfs/.

[68] Rong Gu et al. *DFS-Perf: A Scalable and Unified Benchmarking Framework for Distributed File Systems.* Tech. rep. UCB/EECS-2016-133. EECS Department, University of California, Berkeley, July 2016.

[69] Pradeep Kumar Gunda et al. "Nectar: Automatic Management of Data and Computation in Data Centers". In: *OSDI 2010*.

[70] Philip J Guo and Dawson Engler. "CDE: Using system call interposition to automatically create portable software packages". In: *Proceedings of the 2011 USENIX Annual Technical Conference*. 2011, pp. 247–252.

[71] *H2O*. https://www.h2o.ai/.

[72] Tyler Harter et al. “Analysis of HDFS under HBase: a facebook messages case study”. In: *Proceedings of the 12th USENIX conference on File and Storage Technologies, FAST 2014, Santa Clara, CA, USA, February 17-20, 2014*. 2014, pp. 199–212.

[73] *HDFS Architecture*. http://hadoop.apache.org/docs/r2.7.0/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html.

[74] *HiBench*. https://github.com/intel-hadoop/HiBench.

[75] Benjamin Hindman et al. “Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center”. In: *NSDI 2011*.

[76] Shengsheng Huang et al. “The HiBench benchmark suite: Characterization of the MapReduce-based data analysis”. In: *Workshops Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, Long Beach, California, USA, March 1-6, 2010*. 2010, pp. 41–51.

[77] Julian Hyde. *Discardable Memory and Materialized Queries*. http://hortonworks.com/blog/dmmq/.

[78] *IBM Cloud Object Storage*. https://www.ibm.com/cloud/object-storage.

[79] Michael Isard et al. “Dryad: distributed data-parallel programs from sequential building blocks”. In: *ACM SIGOPS Operating Systems Review* 41.3 (2007), pp. 59–72.

[80] Michael Isard et al. “Quincy: Fair Scheduling for Distributed Computing Clusters”. In: *SOSP*. Nov. 2009.

[81] Yangqing Jia et al. “Caffe: Convolutional Architecture for Fast Feature Embedding”. In: *Proceedings of the 22Nd ACM International Conference on Multimedia*. MM ’14. Orlando, Florida, USA, 2014, pp. 675–678. ISBN: 978-1-4503-3063-3.

[82] Michael J Karels and Marshall Kirk McKusick. “Towards a Compatible File System Interface”. In: (1986).

[83] Haoyuan Li et al. *Reliable, Memory Speed Storage for Cluster Computing Frameworks*. Tech. rep. UCB/EECS-2014-135. EECS Department, University of California, Berkeley, June 2014.

[84] Haoyuan Li et al. “Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks”. In: *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2014, Seattle, WA, USA, November 3-5, 2014*. 2014, pp. 1–15.

[85] Zhenhua Li et al. “Efficient batched synchronization in dropbox-like cloud storage services”. In: *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer. 2013, pp. 307–327.

[86] Hyeontaek Lim et al. “SILT: A Memory-efficient, High-performance Key-value Store”. In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP ’11. Cascais, Portugal, 2011, pp. 1–13. ISBN: 978-1-4503-0977-6.

[87] Liu, Shaoshan and Sun, Dawei. *PerceptIn Robotics Get a Performance Boost from Alluxio Distributed Storage (2017)*. http://thenewstack.io/powering-robotics-clouds-alluxio.

[88] Douglass Locke et al. "Priority inversion and its control: An experimental investigation". In: *ACM SIGAda Ada Letters*. Vol. 8. 7. ACM. 1988, pp. 39–42.

[89] Yucheng Low et al. "Distributed GraphLab: a framework for machine learning and data mining in the cloud". In: *Proceedings of the VLDB Endowment* 5.8 (2012), pp. 716–727.

[90] *Making the Impossible Possible w/ Alluxio (2016) Alluxio, Barclays*. http://alluxio-com-site-prod.s3.amazonaws.com/resource/media/Making_the_Impossible_Possible_w_Alluxio.pdf.

[91] Grzegorz Malewicz et al. "Pregel: a system for large-scale graph processing". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM. 2010, pp. 135–146.

[92] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[93] Sergey Melnik et al. "Dremel: interactive analysis of web-scale datasets". In: *Proceedings of the VLDB Endowment* 3.1-2 (2010), pp. 330–339.

[94] Sergey Melnik et al. "Dremel: interactive analysis of web-scale datasets". In: *Commun. ACM* 54.6 (2011), pp. 114–123.

[95] *Memcached*. http://memcached.org/.

[96] James Mickens et al. "Blizzard: Fast, Cloud-scale Block Storage for Cloud-oblivious Applications". In: *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. NSDI'14. Seattle, WA, 2014, pp. 257–273. ISBN: 978-1-931971-09-6.

[97] *Microsoft Azure Storage*. https://azure.microsoft.com/en-us/services/storage/.

[98] Edmund B Nightingale, Peter M Chen, and Jason Flinn. "Speculative execution in a distributed file system". In: *ACM SIGOPS Operating Systems Review*. Vol. 39. 5. ACM. 2005, pp. 191–205.

[99] Edmund B Nightingale et al. "Flat Datacenter Storage". In: *OSDI*. 2012, pp. 1–15.

[100] M Noll. *Benchmarking and Stress Testing an Hadoop Cluster With TeraSort, TestDFSIO & Co*. http://www.michael-noll.com/blog/2011/04/09/benchmarking-and-stress-testing-an-hadoop-cluster-with-terasort-testdfsio-nnbench-mrbench. 2011.

[101] Christopher Olston et al. "Pig latin: a not-so-foreign language for data processing". In: *SIGMOD '08*, pp. 1099–1110.

[102] *Open Stack Swift Object Store*. https://www.openstack.org/software/releases/ocata/components/swift.

[103] John Ousterhout et al. "The case for RAMCloud". In: *Communications of the ACM* 54.7 (2011), pp. 121–130.

[104] Michael Ovsiannikov et al. "The Quantcast File System". In: *PVLDB* 6.11 (2013), pp. 1092–1101.

[105] *Parallel filesystem I/O benchmark*. https://github.com/chaos/ior.

[106] Andrew Pavlo et al. "A comparison of approaches to large-scale data analysis". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*. 2009, pp. 165–178.

[107] James Plank. "An Overview of Checkpointing in Uniprocessor and Distributed Systems, Focusing on Implementation and Performance". In: *Technical Report, University of Tennessee, 1997*.

[108] James S. Plank and Wael R. Elwasif. "Experimental Assessment of Workstation Failures and Their Impact on Checkpointing Systems". In: *28th International Symposium on Fault-Tolerant Computing, 1997*.

[109] Russell Power and Jinyang Li. "Piccolo: Building Fast, Distributed Programs with Partitioned Tables." In: *Proceedings of the 9th USENIX conference on Operating systems design and implementation*. USENIX Association. 2010, pp. 293–306.

[110] *Presto*. https://prestodb.io/.

[111] Qifan Pu et al. "FairRide: Near-Optimal, Fair Cache Sharing." In: *NSDI*. 2016, pp. 393–406.

[112] *PVFS file system*. http://www.pvfs.org/.

[113] *Qunar Performs Real-Time Data Analytics up to 300x Faster with Alluxio (2017) Alluxio, Qunar.* http://alluxio-com-site-prod.s3.amazonaws.com/resource/media/Qunar_Performs_Real-Time_Data_Analytics_up_to_300x_Faster_with_Alluxio.pdf.

[114] Sanjay Radia. *Discardable Distributed Memory: Supporting Memory Storage in HDFS*. http://hortonworks.com/blog/ddm/.

[115] *Redis*. http://redis.io/.

[116] Charles Reiss et al. "Heterogeneity and dynamicity of clouds at scale: Google trace analysis". In: *Proceedings of the Third ACM Symposium on Cloud Computing*. ACM. 2012.

[117] R Rodriguez, M Koehler, and R Hyde. "The generic file system". In: *USENIX Conference Proceedings*. 1986, pp. 260–269.

[118] *Samza*. http://samza.apache.org/.

[119] Spencer Shepler, M Eisler, and D Noveck. *Network file system (NFS) version 4 minor version 1 protocol*. Tech. rep. 2010.

[120] Konstantin Shvachko et al. "The Hadoop Distributed File System". In: *IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST 2010, Lake Tahoe, Nevada, USA, May 3-7, 2010*. 2010, pp. 1–10.

[121] Konstantin Shvachko et al. "The hadoop distributed file system". In: *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*. IEEE. 2010, pp. 1–10.

[122] *Solaris FileBench*. `http://filebench.sourceforge.net/wiki/index.php/Main_Page`.

[123] *Storage Performance Council*. `http://www.storageperformance.org/home`.

[124] *Storm*. `http://storm-project.net/`.

[125] *Swift*. `https://docs.openstack.org/swift/latest/`.

[126] *Tachyon*. `http://tachyon-project.org/`.

[127] *Tachyon bug fix of the memory usage statistics*. `https://github.com/amplab/tachyon/pull/354`.

[128] *Tachyon scalability issue in reading from under FS*. `https://tachyon.atlassian.net/browse/TACHYON-257`.

[129] Ashish Thusoo et al. "Hive A Petabyte Scale Data Warehouse Using Hadoop". In: *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*. IEEE. 2010, pp. 996–1005.

[130] Avishay Traeger et al. "A nine year study of file system and storage benchmarking". In: *Trans. Storage* 4.2 (2008), 5:1–5:56.

[131] Amin Vahdat and Thomas E Anderson. "Transparent result caching". In: *USENIX Annual Technical Conference*. 1998.

[132] Nitin H. Vaidya. "Impact of Checkpoint Latency on Overhead Ratio of a Checkpointing Scheme". In: *IEEE Trans. Computers 1997*.

[133] Vinod Kumar Vavilapalli et al. "Apache hadoop yarn: Yet another resource negotiator". In: *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM. 2013, p. 5.

[134] 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*. USENIX Association. 2012, pp. 19–19.

[135] Michael Vrable, Stefan Savage, and Geoffrey M Voelker. "Cumulus: Filesystem backup to the cloud". In: *ACM Transactions on Storage (TOS)* 5.4 (2009), p. 14.

[136] Lei Wang et al. "BigDataBench: A big data benchmark suite from internet services". In: *20th IEEE International Symposium on High Performance Computer Architecture, HPCA 2014, Orlando, FL, USA, February 15-19, 2014*. 2014, pp. 488–499.

[137] Sage A. Weil et al. "Ceph: A Scalable, High-Performance Distributed File System". In: *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*. 2006, pp. 307–320.

[138] *Why use Lustre*. https://wiki.hpdd.intel.com/display/PUB/Why+Use+Lustre.

[139] Reynold S Xin et al. “Shark: SQL and rich analytics at scale”. In: *Proceedings of the 2013 international conference on Management of data*. ACM. 2013, pp. 13–24.

[140] Neeraja J. Yadwadkar et al. “Discovery of Application Workloads from Network File Traces”. In: *8th USENIX Conference on File and Storage Technologies, San Jose, CA, USA, February 23-26, 2010*. 2010, pp. 183–196.

[141] *YCSB*. https://github.com/brianfrankcooper/YCSB/.

[142] John W. Young. “A first order approximation to the optimum checkpoint interval”. In: *Commun. ACM* 17 (9 Sept. 1974), pp. 530–531. ISSN: 0001-0782.

[143] Yuan Yu et al. “DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language”. In: *Proceedings of the 8th USENIX conference on Operating systems design and implementation*. USENIX Association. 2008, pp. 1–14.

[144] Matei Zaharia et al. “Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling”. In: *EuroSys 10*. 2010.

[145] Matei Zaharia et al. “Discretized streams: Fault-tolerant Streaming Computation at Scale”. In: *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP ’13, Farmington, PA, USA, November 3-6, 2013*. 2013, pp. 423–438.

[146] Matei Zaharia et al. “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing”. In: *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association. 2012.

[147] Zhao Zhang et al. “MTC envelope: defining the capability of large scale computers in the context of parallel scripting applications”. In: *The 22nd International Symposium on High-Performance Parallel and Distributed Computing, HPDC’13, New York, NY, USA - June 17 - 21, 2013*. 2013, pp. 37–48.

[148] Benjamin Zhu, Kai Li, and R Hugo Patterson. “Avoiding the Disk Bottleneck in the Data Domain Deduplication File System.” In: *Fast*. Vol. 8. 2008, pp. 1–14.